ACADÉMIE DE MONTPELLIER

UNIVERSITÉ MONTPELLIER II

— **S**CIENCES ET **T**ECHNIQUES DU **L**ANGUEDOC —

THÈSE

présentée au Laboratoire d'Informatique de Robotique
et de Microélectronique de Montpellier pour
obtenir le diplôme de doctorat

SPÉCIALITÉ	:	**INFORMATIQUE**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Information, Structures, Systèmes**

Du découplage à l'assemblage non-anticipé de composants

Conception et mise en œuvre du langage à composants SCL

par

Luc FABRESSE

— Version du 10 décembre 2007 —

Soutenance prévue le 10 décembre 2007 à 14h, devant le jury composé de :

Philippe COLLET, Maître de conférences, I3S, Université de Nice Sophia Antipolis, ... Co-Rapporteur
Christophe DONY, Professeur, LIRMM, Université Montpellier II, ... Directeur de Thèse
Jacques FERBER, Professeur, LIRMM, Université Montpellier II, ... Examinateur
Marianne HUCHARD, Professeur, LIRMM, Université Montpellier II, ... Directrice de Thèse
Philippe LAHIRE, Professeur, I3S, Université de Nice Sophia Antipolis, ... Rapporteur
Mourad OUSSALAH, Professeur, LINA, Université de Nantes, ... Rapporteur
Guy TREMBLAY, Professeur, UQÀM, Université du Québec à Montréal, ... Examinateur

Table des matières

Liste des figures

Liste des tableaux

Liste des listings

Remerciements

CHAPITRE 1

Introduction

There are few problems in computer science that cannot be solved by adding an extra level of indirection. But that usually will create another problem.

David WHEELER.

Préambule

Cette introduction présente successivement le contexte, la problématique et les objectifs de ce travail de thèse. Le contexte est celui de l'ingénierie logicielle et plus spécifiquement le développement logiciel par assemblage de composants (Component-Based Software Development, CBSD). La problématique et les objectifs de cette thèse sont présentés à travers un ensemble de questions que nous nous sommes posées tout au long de ce travail. Enfin, nous terminons cette introduction en précisant l'organisation de ce mémoire.

1.1 Présentation de l'approche à composants

Il existe actuellement en ingénierie logicielle un intérêt grandissant pour les techniques et les outils permettant de développer des applications par assemblage de composants logiciels. Cet intérêt pour les composants résulte aussi bien de la volonté de réduire les coûts de développement en augmentant la réutilisation que de la nécessité d'inventer de nouvelles formes de développement pour prendre en compte la complexité structurelle sans cesse croissante des applications liée à de nouveaux besoins comme la répartition, la fiabilité ou l'évolution. Cette nouvelle ère de l'*orienté composants* commence à peine à se développer alors que l'idée fut proposée pour la première fois en 1968 par McIlroy [McIlroy, 1968]. Originellement, cette approche se fonde sur une analogie entre un électronicien qui fabrique un circuit par assemblage de composants électroniques via des connexions supportant le passage des électrons et un informaticien qui pourrait construire une application informatique par assemblage de *composants logiciels* réutilisables via des « connexions » supportant les communications et les échanges de données entre les composants. Actuellement, cette approche nécessite encore d'être précisée, outillée et expérimentée afin de réellement prendre son essor. Cette idée, encore d'actualité, est parfaitement exprimée dans [Brown et Wallnau, 1998] :

> « *CBSE [Component-Based Software Engineering] is a coherent engineering practice, but we still haven't fully identified just what it is.* »

Avant de poser la problématique de cette thèse (*cf.* section1.2), la suite de cette section est consacrée à la présentation générale du développement d'applications par assemblage de composants à travers ses éléments de base, ses objectifs et son processus de développement centré sur la réutilisation.

1.1.1 Eléments de base de l'approche à composants

Deux notions sont centrales et complémentaires dans l'approche composants : les *composants logiciels* et les *architectures logicielles*.

Un composant logiciel est une brique logicielle élémentaire permettant la construction d'applications. Donner une définition plus précise ou plus formelle de ce terme dans sa généralité reste encore une question ouverte à ce jour. En effet, il existe actuellement autant de propositions de définition [Broy *et al.*, 1998] que d'utilisations différentes des composants. Il est même avancé [Gröne *et al.*, 2005] que le terme «composant» ne peut posséder une unique définition puisque suivant le contexte, ce terme désigne aussi bien des schémas de conception (design patterns) [Gamma *et al.*, 1995], des fonctions ou procédures [McIlroy, 1968], des modules [Fröhlich *et al.*, 2005], des cadres d'application (frameworks) [Szyperski, 2002], des classes [Hamilton, 1997], ou encore des applications complètes [Microsoft, 1996]. On peut tout de même s'appuyer sur les définitions les plus répandues dans la littérature afin de mieux cerner ce concept. La définition la plus consensuelle, proposée en 1996 lors d'un atelier de travail sur la programmation orientée composants [WCO, 1996], est :

> « *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.* »

Cette définition, relativement générale, peut s'interpréter à différents niveaux d'abstraction (spécification, conception, implémentation). Elle donne les caractéristiques externes d'un composant (interfaces, dépendances explicites) et met l'accent sur la capacité des composants à être *assemblés*[1] afin de construire des applications. La vision, plus technique, des composants proposée par le SEI (Software Engineering Institute) [Bachman et *al.*, 2000] est qu'ils sont des unités de réutilisation, de distribution et de déploiement possédant des limites et des dépendances claires et explicites, pouvant être paramétrées (customized) et assemblées. Pour finir, une vision plus précise est proposée dans [Heineman et Councill, 2001] dont voici quelques extraits :

> « *A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.*
> *A component model defines specific interaction and composition standards.*
> *A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model.* »

Cette définition met en évidence la notion de *modèle de composants* et précise la nécessité de concevoir les composants afin de permettre leur assemblage. Un modèle de composants décrit donc les concepts à partir desquels sont définis des composants logiciels (la structure des composants) ainsi que des mécanismes basés sur ces concepts permettant l'assemblage et la communication entre des composants. Il est difficile de faire interopérer (assembler, faire communiquer) des composants non conformes au même modèle. Une application construite à base de composants s'exécute au sein d'une plate-forme logicielle. En s'appuyant sur les spécifications établies par le modèle de composants, la plate-forme peut ainsi interpréter le code des composants afin qu'ils réalisent leurs fonctionnalités. Dans le monde des composants distribués, cette plate-forme est appelée intergiciel (ou middleware en anglais).

Une architecture logicielle est définie [Shaw *et al.*, 1995] ainsi :

> « *The architecture of a software system defines that system in terms of components and of interactions among those components. In addition to specifying the structure and topology of the system, the architecture shows the intended correspondence between the system requirements and elements of the constructed system. It can additionally address system-level properties such as capacity, throughput, consistency, and component compatibility. Architectural models clarify structural and semantic differences among components and interactions. Architectural definitions can be composed to define larger systems. Elements are defined independently so they can be re-used in different contexts.* »

Une architecture logicielle définit la structure macroscopique d'un logiciel en termes de composants, qui assurent les fonctions de calculs, et de *connecteurs* [Shaw, 1996; Mehta *et al.*, 2000] qui relient les composants et coordonnent leurs interactions pour satisfaire des contraintes globales d'intégrité (invariants structurels, coordination, etc.) et des contraintes de qualité (fiabilité, sécurité, évolutivité, etc.).

[1] En français, on préfère le terme « assemblage » à celui de « composition » car la composition de composants n'est qu'une forme particulière d'assemblage consistant à construire des composites.

1.1.2 Objectifs de l'approche à composants

A l'instar de tout projet, le développement d'un logiciel nécessite de maîtriser au mieux les coûts et le temps. Ces deux ressources doivent être maîtrisées durant tout le cycle de vie d'un logiciel ce qui inclut son développement, ses phases de tests mais aussi sa maintenance et ses inévitables évolutions comme le signale la première loi de Lehman [Lehman et Belady, 1985]. Cette gestion est de plus en plus difficile notamment à cause d'une part de la taille des logiciels, et d'autre part du développement des réseaux de communications. En effet, la taille des logiciels ne cesse de croître (on peut citer comme exemple la suite bureautique OpenOffice.org[2] qui a dépassé les 10 millions de SLOC, *Source Line of Code*). D'autre part, l'essor des réseaux informatiques a doté les logiciels de fonctions communicantes qui incluent la distribution mais aussi les possibilités d'interactions avec d'autres applications ou systèmes qui évoluent indépendamment. Dans ce nouveau contexte de développement d'applications à grande échelle, le développement par composants promet, en ce qui concerne le coût et le temps de développement, les améliorations suivantes :

- La diminution des coûts et du temps de développement et de test par la réutilisation ou l'achat de composants existants ;
- La réduction du temps de test et de mise au point des logiciels grâce à l'utilisation de composants testés et éprouvés ;
- La réduction des coûts de maintenance et d'évolution des logiciels car ils sont constitués de composants découplés pouvant être remplacés indépendamment les uns des autres. Cette propriété est appelée l'« extensibilité indépendante » [Szyperski, 2002] (en anglais *independent extensibility*) des applications à base de composants ;
- Une simplicité accrue pour le développement d'applications puisqu'il n'est pas nécessaire de comprendre le fonctionnement interne des composants. Il suffit en effet de connaître les fonctionnalités qu'ils peuvent accomplir ainsi que la façon dont ils communiquent avec leur environnement.

L'approche composants promet aussi une meilleure qualité logicielle. Considérons le modèle de qualité défini par la norme 9126-1 de l'ISO/IEC [Standard, 2001] (International Organization for Standardization/International Electrotechnical Commission) qui est l'un des modèles les plus utilisés. Ce modèle définit six facteurs primaires de qualité : la capacité fonctionnelle (le logiciel répond-il aux besoins fonctionnels, de sécurité et d'interopérabilité ?), la fiabilité (le logiciel est-il capable de fournir ses services dans des conditions et sur une durée spécifiées dans le cahier des charges ?), l'efficacité (le logiciel est-il rapide et économe en consommation mémoire ?), l'utilisabilité (le logiciel est-il facile à utiliser ?), la maintenance (le logiciel est-il facilement modifiable, testable, adaptable ?) et la portabilité (le logiciel peut-il être facilement installé dans un autre environnement ?). Adopter l'approche à composants, par rapport à l'approche à objets notamment, promet un gain de qualité sur au moins trois de ces six facteurs : maintenance, fiabilité et portabilité. Toutefois, il est difficile d'évaluer à ce jour l'impact réel de l'utilisation de l'approche à composants et notamment si les trois autres facteurs ne sont pas dégradés.

[2] http://www.openoffice.org

1.1.3 Un processus de développement centré sur la réutilisation

De plus en plus de travaux visent à définir des méthodes de développement adaptées à l'approche à composants [D'Souza et Wills, 1999; Hassine *et al.*, 2003; Mei, 2004; Atkinson *et al.*, 2001]. La définition de ces nouvelles méthodes de développement a fait émerger de nouveaux acteurs tels que l'architecte, le développeur ou encore le déployeur [Marvie, 2002]. Deux acteurs nous intéressent plus particulièrement : le *développeur de composants* et *l'architecte d'application*. Les rôles complémentaires de ces deux acteurs (*cf.* figure 1.1) sont centraux dans le processus de développement par composants. Le développeur a pour responsabilité de définir et implémenter des composants indépendamment de toute application de sorte qu'ils puissent être utilisés dans différents contextes. À l'inverse, l'architecte construit une application. Il choisit donc des composants sur étagères, les adapte si nécessaire pour les intégrer à son application. On dit que le développeur fait de la conception pour la réutilisation (« *design for reuse* ») alors que l'architecte fait de la conception par la réutilisation (« *design by reuse* ») [Oussalah, 2005]. Il est évident qu'une même personne peut exercer ces deux rôles et qu'un composant peut être créé en vue d'être intégré dans une application donnée. Ce nouveau composant développé doit être complètement indépendant de l'application pour laquelle il est conçu afin qu'il puisse éventuellement être réutilisé dans une autre application.

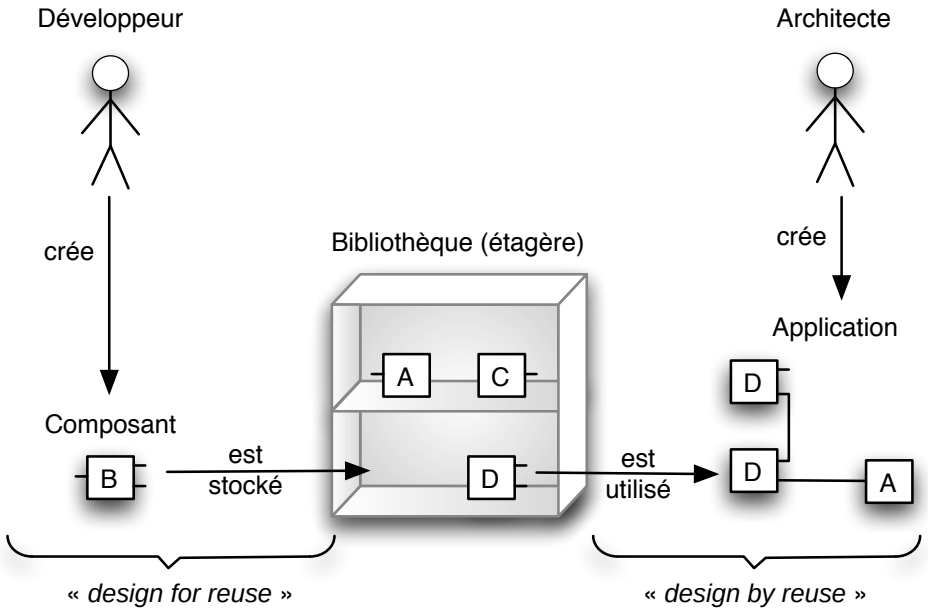

FIG. 1.1 : Vision simplifiée du processus de développement par composants à travers les deux rôles centraux du développeur et de l'architecte.

Dans cette thèse, nous nous concentrons sur ces deux rôles centraux du processus de développement incarnés par le programmeur et l'architecte qui sont à la fois complémentaires et obligatoirement distincts.

Année	Approches à composants				
1989	Durra				
1993	Darwin				
1994	Polylith	Aesop			
1995	COM				
1996	C2	Unicon	Wright		
1997	Acme	*Javabeans*	EJB 1.0	Koala	
1998	SOFA	MALEVA			
1999	Bonobo				
2000	ComponentJ	Comet	Javapod	Services Web	
2001	ArchJava	UML 1.5	Lagoona	Jiazzi	ACOEL
2002	CCM 3.0	Fractal	PECOS	Reo	JAC
2003					
2004	UML 2.0	CCM 4.0	Boxscript		
2005	Picolo				

TAB. 1.1 : Chronologie d'apparition des approches à composants

1.2 Motivations et objectifs de la thèse

Actuellement, de nombreuses propositions (*cf.* table 1.1) se réclament du mode de développement par assemblage de composants logiciels. Cette abondance provient d'une part des industriels qui proposent de plus en plus de technologies comme J2EE ou .NET s'inscrivant dans l'approche composants — en tous cas du point de vue terminologique — et d'autre part des recherches académiques qui s'emploient à formaliser et clarifier les concepts en proposant notamment des langages de description d'architectures (ADL) ou des langages de programmation spécifiques. Malgré un vocabulaire parfois commun (composant, port, interface, service, assemblage, connexion, liaison, connecteur, composition, composite, ...), ces propositions sont souvent disparates de par leurs origines, leurs objectifs, leurs concepts, leurs mécanismes ou encore leurs utilisations. Devant la profusion actuelle de propositions à base de composants, il est difficile de bien comprendre l'essence de l'approche composants.

Dans la suite de cette section, nous présentons tout d'abord les motivations de cette thèse à travers un ensemble de questions et les éléments de réponse actuels, ce qui nous permet ensuite de définir précisément notre problématique et d'énoncer brièvement nos contributions.

Quels sont les nouveaux concepts et mécanismes introduits par la programmation par composants ?
Il est difficile aujourd'hui d'énoncer clairement les apports de la programmation par composants (PPC ou *Component-Oriented Programming* soit COP en anglais) comme on peut le faire pour la programmation par objets (PPO ou *Object-Oriented Programming* soit OOP en anglais). Dans le cas de la PPO, les deux langages fondateurs sont : Simula [Birtwistle *et al.*, 1973], apparu en 1967 et Smalltalk [Goldberg et Robson, 1989], apparu en 1973. Les apports de ces langages par rapport aux lan-

gages procéduraux comme Pascal notamment sont les notions d'« objet » qui encapsule données et traitements, d'« envoi de message » qui permet à un objet receveur de décider de son comportement, d'« héritage » et de « spécialisation » qui permettent une meilleure réutilisation et une extension facilitée. Enoncer aussi clairement et précisément les apports de la PPC est actuellement difficile. On peut toutefois affirmer que le cœur de la PPC est certainement le mécanisme d'« assemblage » des composants. Toutefois, ce mécanisme n'est pas présent dans toutes les propositions, par exemple on ne le trouve pas dans le modèle EJB, et il peut se décliner sous des formes bien différentes dans les autres propositions. On peut aussi arguer qu'un des apports de l'approche à composants, par rapport à l'approche à objets notamment, est que les composants sont dotés d'« interfaces requises » qui permettent d'expliciter et de spécifier clairement les besoins d'un composant par rapport à son environnement. Un composant est également doté d'« interfaces fournies » le protégeant de tout accès direct mais cette notion est moins spécifique de l'approche composants car elle existait dans l'approche objets au travers des propriétés publiques des objets.

L'approche composants n'est-elle qu'une extension de l'approche à objets ?
Actuellement, l'approche objet est considérée comme un paradigme de programmation uniforme dans le sens où il existe des langages de programmation comme Smalltalk où « tout est objet ». Dans ce contexte, peut-on imaginer la même chose pour l'approche composants ou s'agit-il d'une approche ne pouvant se suffire à elle-même ? Apporter une réponse concrète à cette question est actuellement difficile car la majorité des langages à composants (LAC) sont des extensions de langages à objets où les objets et les composants cohabitent de façon non-uniforme. De plus, cela laisse supposer que tous les mécanismes objets font partie intégrante de l'approche à composants. Or, quelques approches à composants spécifiques comme Koala [Ommering, 1998] nous prouvent le contraire puisqu'elles sont implémentées avec des langages de programmation procéduraux comme le langage C. Finalement, répondre à cette question revient à identifier tous les mécanismes nécessaires et suffisants pour faire de la programmation par composants.

A-t-on besoin de langages de programmation dédiés à l'approche composants ?
Oui ! Nous pensons en effet que des langages de programmation dédiés à l'approche composants sont nécessaires car :

- Les langages de description d'architectures [Medvidovic et Taylor, 2000] ne permettent généralement pas d'écrire des descriptions exécutables. Nous sommes en accord complet avec la phrase suivante [Cointe *et al.*, 2004] :

  > « *L'approche langage (de programmation) nous semble devoir être privilégiée car elle autorise une définition concise des modèles de composants et de leurs systèmes d'assemblage mais également car elle permet de disposer de descriptions exécutables.* »

- Bien que les langages de programmation existants (procéduraux ou à objets) soient largement utilisés pour faire de la PPC, ils imposent aux programmeurs de respecter des règles de transformation entre les concepts de l'approche à composants et ceux du langage cible. Des exemples simples de transformations sont : un composant peut être représenté par une classe, les interfaces des composants peuvent être représentées par des interfaces de classes et les interactions entre les composants peuvent définies par des associations ou des envois de messages. L'utilisation de transformation n'est pas satisfaisant pour faire de la PPC car cela complexifie la pro-

grammation à cause de l'utilisation de conventions de nommage ou de schémas de conception dans les cas complexes. De plus, les principaux intérêts de l'approche à composants sont alors perdus au niveau du code source [Perry et Wolf, 1992].

– Les approches à composants actuelles qui permettent de faire de la PPC comme ArchJava [Aldrich *et al.*, 2002b], Julia (implémentation de référence de Fractal [Bruneton *et al.*, 2004]) ou encore Lagoona [Fröhlich *et al.*, 2005] sont disparates. De plus, la plupart de ces langages sont des extensions de langages à objets dans lesquels coexistent les objets et les composants de façon non-uniforme. Tout cela rend difficile la compréhension et la pratique de la PPC.
– Même si on adopte une approche d'ingénierie dirigée par les modèles (IDM ou *Model Driven Engineering* soit MDE en anglais) [Schmidt, 2006], l'utilisation de langages à composants unifiés reste nécessaire. En effet, la génération de code et la rétro-ingénierie seraient très certainement d'autant plus simples que le langage de programmation cible intègre des abstractions et des mécanismes de haut niveau. D'autre part, on ne peut pas envisager à l'heure actuelle de modifier le modèle et de générer à nouveau l'application pour chaque évolution, notamment les évolutions mineures comme les corrections de *bugs*. Il nous semble donc raisonnable de vouloir générer du code source suffisamment simple pour être maintenu et modifié directement dans certains cas ce qui suppose l'existence de langages à composants unifiés.

En résumé, dans cette thèse nous défendons que les langages de programmation à composants sont nécessaires, disparates dans l'existant et ne doivent pas être des extensions de langages à objets car ils proposent des mécanismes inutiles voire nuisibles pour faire de la PPC. Notre problématique est donc la suivante :

Identifier les concepts et les mécanismes fondamentaux de l'approche à composants, les expliciter clairement au sein d'un modèle de composants et proposer un langage à composants minimal qui permette de programmer simplement des applications par assemblage de composants.

En réponse à cette problématique, nous proposons dans cette thèse le langage à composants SCL (*Simple Component Language*). Les objectifs qui ont guidé la spécification de SCL sont parfaitement énoncés dans [Wuyts et Ducasse, 2001][3] :

> « *[...] we feel that there is need for pure component languages. These languages are needed to provide a component developer with a clean and concise vocabulary and semantics for building and composing components.* »

Plus spécifiquement, voici quelques questions à propos des langages à composants (LAC) — dont la plupart ne sont pas abordées dans les travaux existants — auxquelles nous répondons dans cette thèse à travers la conception de SCL :

– Qu'est-ce qu'un langage à composants et que doit-il permettre ?

[3] Pour information, nous ne connaissions pas cet article lors de la conception de SCL, ce qui semble indiquer une certaine convergence des idées dans cette direction.

- Qu'est-ce qu'un port ?
- Qu'est-ce qu'un envoi de message en PPC, le concept est-il encore valide ?
- La notion de receveur courant a-t-elle un sens ?
- Qu'est-ce qu'un attribut ?
- Comment intégrer des données de base (entiers, chaînes de caractères, etc.) ou des objets de base (collections, etc.) de façon uniforme dans un monde de composants ?
- Qu'est-ce qu'assembler, connecter et composer des composants ?
- Comment s'effectue le passage d'arguments et peut-il être assimilé à des connexions ?
- Doit-il y avoir un mécanisme d'héritage ?
- Séparer les préoccupations dans le code source est-il toujours possible avec un LAC contrairement aux langages à objets où cela nécessite des mécanismes supplémentaires tels que ceux introduits par la programmation par aspects [Kiczales *et al.*, 1997] ?

1.3 Organisation de ce mémoire

Outre ce chapitre d'introduction, ce mémoire de thèse est organisé en cinq autres chapitres.

Le chapitre 2 dresse un état de l'art de l'approche à composants. Les motivations générales de cette approche sont présentées à travers trois axes problématiques du génie logiciel : la réutilisation, le passage à l'échelle et la répartition pour lesquels le développement par composants semble mieux adapté que l'approche à objets notamment. Après avoir présenté les principales familles d'approches à composants, certaines approches sont décrites de façon détaillée pour être ensuite comparées. Finalement, en conclusion de ce chapitre, nous montrons les faiblesses de ces approches en ce qui concerne la programmation par assemblage de composants.

Le chapitre 3 présente la spécification du langage à composants SCL. Cette spécification est guidée par des objectifs généraux qui sont définis en début de chapitre. Ensuite, chaque section présente un besoin de la PPC, compare les solutions existantes et finalement intègre une solution en SCL, parfois nouvelle, tout en respectant les objectifs de départ. Avant de conclure ce chapitre, une synthèse du noyau de SCL est présentée.

Le chapitre 4 présente une étude autour de la séparation des préoccupations dans les approches à composants. Après avoir rappelé le principe de séparation des préoccupations, nous présentons la programmation par aspects qui permet une meilleure modularisation de certaines préoccupations dites « transversales » ainsi qu'une étude des principales approches mixtes à composants et à aspects. Nous proposons ensuite deux extensions de SCL respectant les objectifs présentés dans le chapitre précédent. La première permet d'utiliser un même composant SCL de façon standard ou de façon transversale. Contrairement aux approches mixtes dites « symétriques » dont s'inspire cette extension, le programmeur d'un composant SCL n'a pas à écrire de code spécifique pour cela. De même, la deuxième extension permet d'établir des connexions entre les composants basées sur les changements d'états de leurs propriétés.

Le chapitre 5 présente les deux prototypes de SCL. Concernant le prototype le plus avancé, écrit en Smalltalk, nous présentons nos choix de conception, le modèle implémenté, des exemples de code ainsi qu'un prototype d'outil de développement graphique. Nous présentons ensuite le cœur d'un

deuxième prototype en Ruby.

Finalement, un bilan du travail réalisé et les perspectives de ce travail sont décrits dans le chapitre 6.

CHAPITRE 2

Synthèse comparative des approches à composants

The Japanese have a small word – ma – for "that which is in between" – perhaps the nearest English equivalent is "interstitial". The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.

Alan KAY.

Préambule

Dans ce chapitre, nous présentons un état de l'art des travaux les plus représentatifs qui se réclament du développement par assemblage de composants. La section 2.1 commence par une introduction rappelant les principaux avantages — ou promesses — de l'approche à composants. Nous détaillons particulièrement trois axes : la réutilisation, le passage à l'échelle et la répartition. Devant la multitude des approches à composants, nous avons fait une sélection de celles que nous considérons comme principales afin de les détailler dans la section 2.2. La section 2.3 présente une comparaison des approches, notamment celles étudiées en détails, suivant un ensemble de critères que nous jugeons pertinents pour notre problématique. La section 2.4 conclut ce chapitre en proposant une vision synthétique des différentes approches et montre la nécessité de proposer un nouveau langage à composants.

2.1 Pourquoi des composants logiciels ?

SUITE aux nouvelles contraintes auxquelles doit faire face le génie logiciel, les facteurs « coût » et « temps » sont de plus en plus difficiles à maîtriser dans le cycle de vie d'un logiciel. Parmi ces nouvelles contraintes, deux sont centrales : la taille croissante des applications et leur répartition qui est due à l'essor des réseaux informatiques.

Cette section présente les trois principaux axes dans lesquels le développement par composants promet d'être mieux adapté que le développement classique comme le développement par objets notamment. Ces trois axes sont : (*i*) la réutilisation qui permet de réduire directement les coûts et le temps lors des phases de développement et de test, (*ii*) le passage à l'échelle et (*iii*) la répartition qui complexifie le code des applications en imposant l'intégration de code spécifique pour les communications. Chacun de ces trois axes est développé dans une sous-section avec l'objectif de montrer les apports ou les promesses de l'approche à composants suivant cet axe.

2.1.1 La réutilisation

La réutilisation est un des objectifs majeurs du génie logiciel. De nombreux concepts et mécanismes associés ont été inventés pour réutiliser le code :
- fonction et appel de fonction,
- module et importation de module,
- classe et héritage,
- *framework* et paramétrage de *framework*,
- composant et assemblage de composants.

Dans tous les cas, la réutilisation s'effectue en deux étapes : *abstraction* et *utilisation*. L'abstraction consiste à définir sous une forme abstraite et indépendante de tout contexte ce qui peut être réutilisé. L'utilisation consiste à réutiliser dans un contexte donné une abstraction, ce qui peut nécessiter des adaptations et/ou du paramétrage. Par exemple, la définition d'une fonction s'effectue avec des paramètres formels (abstraction) qui sont substitués par des paramètres réels (paramétrage) lors d'un appel de la fonction. Un mécanisme de réutilisation de code définit donc comment décrire les abstractions et comment les utiliser ensuite. On distingue généralement trois types de réutilisation : *boîte noire*, *boîte blanche* et *boîte grise*.

Avec l'approche boîte noire, l'implémentation de l'abstraction n'est pas connue lors de son utilisation et le paramétrage s'effectue à travers des *interfaces* définies explicitement. Typiquement, une fonction peut être considérée comme une boîte noire paramétrable uniquement via ses paramètres qui constituent son interface. Ce type de réutilisation présente l'avantage que l'abstraction peut être aisément changée par une autre présentant les mêmes interfaces. Par exemple, changer l'implémentation d'une fonction[1] sans changer sa spécification externe (signature et sémantique) permet à tous les programmes utilisant cette fonction de profiter de la nouvelle définition sans qu'il soit nécessaire de changer leur code.

[1] On parle ici d'une fonction sans « effet de bord ».

La réutilisation de type boîte blanche repose sur le fait que les détails de l'implémentation de l'abstraction sont connus lors de son utilisation et le paramétrage peut s'effectuer soit à travers les interfaces, soit directement à travers son implémentation. Ce type de réutilisation présente l'avantage de fournir toutes les informations sur le fonctionnement interne de l'abstraction et des possibilités de paramétrage plus fines. Par exemple, cette forme de réutilisation est utilisée pour les *frameworks* objets. Les interfaces explicites de paramétrage d'un *framework* sont les méthodes abstraites déclarées dans les classes qui le constituent. Il est donc possible grâce à la spécialisation de classes de paramétrer un *framework* en définissant les méthodes abstraites ou en redéfinissant des méthodes existantes dans des sous-classes. La redéfinition de méthodes existantes dans le *framework* constitue une réutilisation de type boîte blanche. Cette forme de réutilisation est problématique du point de vue des changements. En effet, si une nouvelle implémentation du *framework* est produite, sans changement de son interface (méthodes et classes abstraites), les programmes clients qui paramétraient le *framework* en redéfinissant directement ses méthodes peuvent ne plus fonctionner.

La réutilisation est donc confrontée à un paradoxe. Pour augmenter le potentiel de réutilisation d'une abstraction, indépendamment d'un contexte d'utilisation, il faut la considérer comme une boîte noire ayant des interfaces explicites. Or, pour une utilisation dans un contexte donné, des mécanismes permettant le paramétrage et l'adaptation de l'abstraction sont nécessaires, ce qui constitue une approche de type boîte blanche. La réutilisation de type boîte grise est un niveau intermédiaire entre les deux formes précédentes. Les détails d'implémentation de l'abstraction peuvent être connus ou révélés pour comprendre sa réalisation mais ne peuvent pas être modifiés par ses clients et son utilisation ne peut s'effectuer qu'à travers ses interfaces.

Actuellement, l'approche objets est la plus utilisée pour le développement de logiciel. Pourtant, la réutilisation de code basée sur l'héritage entre les classes est assez problématique comme le met en évidence, par exemple et entre autres, le problème de la classe de base fragile [Mikhajlov et Sekerinski, 1998]. Ce problème met en évidence le fait que les modifications (du fait des évolutions par exemple) apparemment sûres des classes de base (super-classes) peuvent tout même causer de mauvais fonctionnements dans les sous-classes. Un programmeur ne peut donc déterminer si un changement est sûr en examinant uniquement la classe dans laquelle a eu lieu le changement. Nous reviendrons plus précisément sur ce problème dans la section 3.13 qui traite de l'héritage. Par ailleurs, une classe ne constitue pas une unité de réutilisation satisfaisante comme l'explique [Flatt, 2000] :

> « *[...] class-based languages encourage the reuse of class definitions through extension, but they do not permit the reuse of a class extension in disjoint parts of a class hierarchy.* »

Une classe n'est en effet pas indépendante de tout contexte puisqu'elle est liée à la hiérarchie dans laquelle elle a été définie. Ce problème du *couplage* implicite [Briand *et al.*, 1999; Peschanski *et al.*, 2000] se traduit par le fait que dans le code des méthodes, il est possible d'instancier ou d'utiliser des éléments externes sans que ces liens soient explicités via des interfaces permettant ainsi une réutilisation boîte noire et du paramétrage. La figure 2.1 montre un exemple de couplage implicite entre deux objets. Chaque objet instance de la classe `A` utilisera sa propre instance de la classe `B` pour remplir la fonctionnalité `bar`. Ce couplage entre les objets est implicite car noyé dans le code source qui n'est pas toujours visible et il est aussi fort car non modifiable.

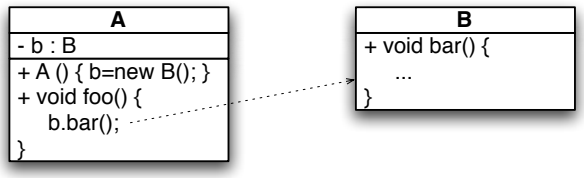

FIG. 2.1 : Couplage fort et implicite entre deux classes d'objets

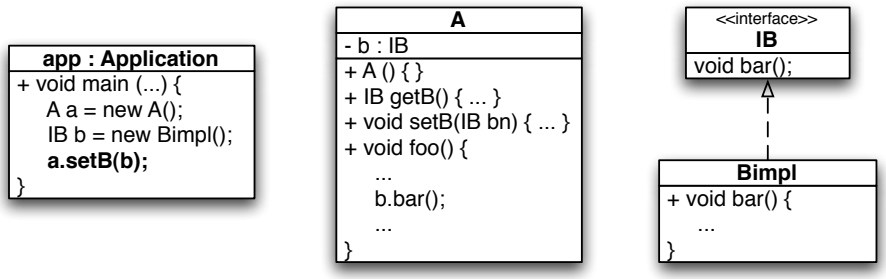

FIG. 2.2 : Couplage faible et explicite entre deux classes d'objets

Une avancée vers plus de découplage serait que le constructeur de la classe `A` possède un paramètre de type `B` afin de valuer l'attribut privé `b`. Lors de l'instanciation de la classe `A`, il serait alors possible de passer en argument une instance de la classe `B`. Le couplage serait alors effectué lors de l'instanciation de la classe `A` et non pas statiquement dans son code. Ce découplage n'est toujours pas satisfaisant car il n'autorise le paramétrage que lors de l'instanciation et non durant toute la vie des objets de type `A`. L'utilisation d'accesseurs (`get` et `set`), d'une interface (au sens objet) et du sous-typage (*cf.* figure 2.2) permet d'expliciter les liens de couplage entre objets et d'autoriser la modification de ces liens durant l'exécution. La classe `A` utilise dans cet exemple l'interface `IB` pour typer son attribut et définir ses accesseurs. Les signatures déclarées dans l'interface `IB` indiquent quelles sont les fonctionnalités réalisées par un objet, ce qui n'introduit pas de couplage dans la classe `A` avec une classe particulière comme la classe `Bimpl` qui indique comment sont réalisées des fonctionnalités lorsqu'elle définit des méthodes.

Bien qu'il soit possible de réaliser le découplage avec un langage à objets, cela nécessite l'utilisation de conventions de programmation qui ne sont pas toujours respectées par les programmeurs. La figure 2.3 montre un exemple schématisé et de haut niveau illustrant le fait que l'approche à composants impose le découplage entre les composants ainsi que l'explicitation des interfaces de communication. Dans cet exemple, le composant `a` appelle (`call`) un service `bar` qu'il ne définit pas (`extern`). Cet appel sera traité par un autre composant, ici `b`, connecté au composant `a` lors de son *assemblage*.

En résumé, l'approche à composants prône l'utilisation systématique d'interfaces explicites afin d'éviter les couplages faibles et implicites et une réutilisation de type boîte noire ou boîte grise. Cette volonté de « protéger » les composants de leur contexte de définition et d'utilisation en explicitant leurs dépendances via des interfaces a pour but de favoriser leur réutilisation. C'est ainsi que l'on espère développer une réutilisation massive du code et même un marché de composants (*component*

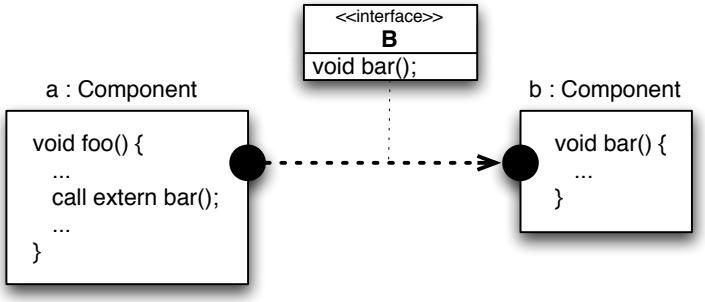

FIG. 2.3 : Couplage faible et explicite entre deux composants

market) [Szyperski, 2002].

2.1.2 Le passage à l'échelle

La taille du code source des applications étant en constante croissance, il est de plus en plus difficile de comprendre, maintenir et tester les programmes. Pour appréhender la taille sans cesse croissante des logiciels, on développe aujourd'hui des techniques de visualisation du code [Langelier *et al.*, 2005; Ducasse *et al.*, 2006; Wettel et Lanza, 2007] afin de mieux percevoir l'architecture d'une application mais aussi ses évolutions. Toutefois, reconstruire l'architecture d'une application [Pollet *et al.*, 2007] est une tâche complexe notamment car elle n'est pas représentée dans le code source lorsqu'on utilise un langage à objets. En effet, les langages à objets sont de plus en plus décrits comme permettant de faire de la programmation à petite échelle (*programming in the small*) [DeRemer et Kron, 1975].

Actuellement, la programmation à grande échelle (*programming in the large*) consiste à partitionner un logiciel en modules de taille suffisamment importante pour posséder des interfaces clairement spécifiées et peu sujettes aux modifications. Les interactions entre les différents modules d'une application peuvent ensuite être définis indépendamment de leurs implantations et de leurs évolutions. L'approche à composants et plus particulièrement sa capacité à mieux exprimer les architectures logicielles [Shaw *et al.*, 1995] en terme de composants interconnectés, semble proposer une vision adaptée pour la programmation à grande échelle où un module est un composant. Il existe d'ailleurs des langages de haut niveau dédiés à la description, parfois à la programmation, à grande échelle comme les ADLs (*Architecture Description Languages*) qui s'attachent à décrire la structure ou le comportement des architectures logicielles en terme de composants et de connecteurs. La description des architectures logicielles est donc le point de départ permettant d'appréhender une architecture dans sa globalité et ainsi proposer des techniques de haut niveau adaptées aux architectures à grande échelle comme le modèle d'évolution d'architecture SAEV [Oussalah *et al.*, 2006] ou encore un mécanisme de remplacement automatique d'un composant par un ensemble de composants interconnectés tout en préservant la qualité de l'architecture de départ [Desnos *et al.*, 2007].

2.1.3 La répartition et l'interopérabilité

La multiplication du nombre des réseaux informatique a profondément bouleversé le développement des applications. Elles ont été dotées de fonctions pour communiquer ou pour accéder à des ressources distantes. La nécessité de découplage entre la partie métier et la partie technique d'une application (comme les communications distantes, la sécurité, etc.) a conduit à l'élaboration d'intergiciels (aussi appelés bus logiciels, *middleware* en anglais). Un intergiciel (*cf.* figure 2.4) est une couche logicielle intermédiaire installée sur les systèmes d'exploitation ayant pour objectif d'offrir une vision uniforme et transparente aux applications communicantes en masquant la répartition géographique, l'hétérogénéité des systèmes, des matériels et des protocoles de communication.

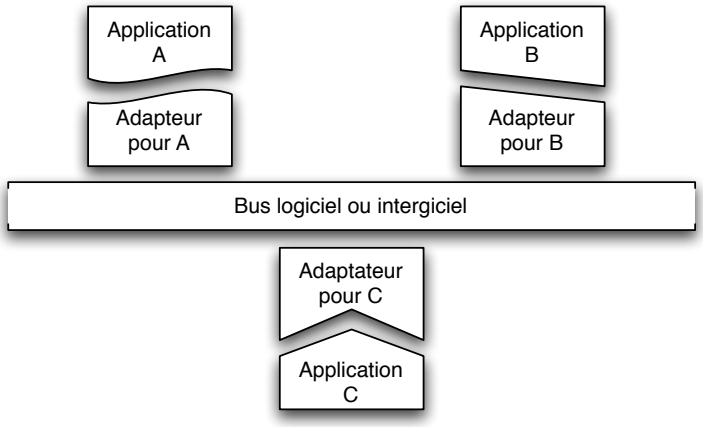

FIG. 2.4 : Principe d'un intergiciel

L'utilisation d'un intergiciel permet d'assurer une indépendance entre le code métier des applications et le code technique qui doit faire face à de nombreuses problématiques de bas niveau comme la concurrence ou encore les pannes physiques. Un intergiciel offre généralement un ensemble de services communs de haut niveau (aussi appelés services *non-fonctionnels*) comme le contrôle de concurrence, les transactions ou la sécurité.

Les intergiciels à objets ou ORB (*Object Request Broker*) permettent l'invocation distante d'une méthode d'un objet. Plusieurs intergiciels à objets existent dont le plus connu est certainement le bus logiciel normalisé CORBA (*Common Object Request Broker Architecture*) de l'OMG (*Object Management Group*).

Toutefois, les architectures à objets répartis ont évolué vers des architectures à composants répartis. Par exemple, l'OMG propose actuellement le modèle de composants CCM (*Corba Component Model*) qui est une évolution du modèle d'objets CORBA. Cette évolution vers les composants dans le domaine des intergiciels a été notamment motivée par les problèmes de couplage évoqués dans la section 2.1.1, mais aussi par la difficulté à bien séparer les préoccupations métiers et techniques (aussi appelées non-fonctionnelles) en utilisant une approche à objets. Avec les composants, le code d'accès aux services non-fonctionnels n'est pas mélangé avec le code métier. Pour cela, les communications entre la couche métier et la couche technique sont contrôlées par une tierce entité au sein

de la plateforme intergicielle. C'est donc cette tierce entité qui utilise le code métier lorsque cela est nécessaire et non l'inverse. Ce principe d'externalisation est parfois appelé inversion de contrôle (*inversion of control*) et il est tout à fait similaire au fonctionnement des *frameworks* où le code écrit par un programmeur est appelé par le code du *framework*. Dans le modèle EJB détaillé dans la section 2.2.4, cette tierce entité est appelé un *conteneur* et il supporte l'exécution d'un composant EJB.

2.2 Présentation détaillée des principales approches à composants

Dans cette section, nous allons décrire de façon détaillée un ensemble de propositions à base de composants. Comme le montre le tableau 1.1 dans le chapitre 1, il existe de plus en plus d'approches s'inscrivant dans le contexte du développement par composants. La section 2.2.1 présente les principales familles d'approches ainsi que les raisons qui nous ont conduits à présenter de façon détaillée les approches (D)COM(+), *Javabeans*, EJB, CCM Fractal, SOFA, ArchJava et UML 2.0. Chacune des sections suivantes est consacrée à l'étude d'une des approches précédentes et est structurée selon le plan suivant :

1. Étude du modèle
   a) Historique et présentation générale
   b) Structure des composants
      i. Structure externe
      ii. Structure interne
   c) Assemblage des composants
      i. Types de liens de connexion
      ii. Types de communications possibles
      iii. Vérification des assemblages
   d) Spécificités du modèle (héritage, déploiement, dynamicité, etc.)
2. Exemple
3. Critique

2.2.1 Les principales familles d'approches à composants

Bien qu'il n'existe pas de classification générale des approches à composants, la littérature [Marvie, 2002; Oussalah, 2005] distingue souvent :

- les approches industrielles comme (D)COM(+)/.NET ou EJB,
- les approches académiques comme les *langages de description d'architectures* (ADLs) (Unicon, C2, WRIGHT, ACME, etc.) ou encore d'autres propositions telles que ArchJava et ComponentJ,
- les normes (ou modèles de référence) comme CCM ou UML 2.0.

Cette classification permet d'expliquer de façon générale les différents objectifs visés par ces différentes propositions. Toutefois, elle ne permet pas de classer facilement toutes les approches à composants comme par exemple l'approche Fractal qui peut être classée dans les trois catégories puisqu'elle est réalisée dans le cadre d'un groupement entre industriels et chercheurs, que le modèle Fractal est une norme et que son implémentation de référence Julia est un langage.

Les approches industrielles visent à fournir un cadre concret permettant le développement d'applications réparties par composants. Elles reposent bien souvent sur des solutions techniques qui rendent complexe l'apprentissage de ce mode de développement.

Les approches académiques sont assez disparates. On distingue généralement les ADLs [Medvidovic et Taylor, 2000] qui permettent de décrire des architectures logicielles des autres propositions. Les ADL sont des langages spécialisés n'ayant pas tous les mêmes objectifs mais ils utilisent bien souvent des concepts similaires : *composant*, *connecteur* et *configuration*. Comme dans [Marvie, 2002], on peut distinguer au moins trois sortes d'ADLs :

- formels comme Wright dont la motivation est de capturer le comportement des applications à des fins de vérifications automatisables,
- de communication comme ACME qui permettent l'échange ou l'intégration d'éléments architecturaux définis à l'aide de langages différents,
- de configuration comme C2 dont l'objectif est d'exploiter les descriptions d'architectures dans le but de générer en partie les composants logiciels ou de supporter l'automatisation du processus de déploiement des applications.

Les ADLs sont une réponse au besoin d'exprimer clairement la structure des applications afin de passer à un mode de développement à grande échelle (*programming in the large*) contrairement aux langages à objets qui à la base sont plus spécialisés dans le développement à petite échelle (*programming in the small*). Toutefois, les descriptions architecturales sont souvent peu reliées à l'implémentation d'une application qui s'effectue toujours dans des langages de programmation tels les langages à objets. Bien que des squelettes de code puissent être générés, il est impossible de garantir que les propriétés architecturales sont effectivement respectées par l'implémentation. Cette constatation est à l'origine du langage ArchJava [Aldrich *et al.*, 2002b] (décrit dans la section 2.2.8). Ce langage hybride tente de faire le lien entre un langage de programmation (Java) et les concepts des ADLs. D'autres approches hybrides similaires existent comme Lagoona [Fröhlich *et al.*, 2005], ComponentJ [Seco et Caires, 2000], ou encore keris [Zenger, 2005].

Les normes ou modèles de référence comme CCM ou UML 2.0 permettent de concilier les approches « haut niveau » des modèles académiques avec les préoccupations pratiques des approches industrielles. Par exemple, UML 2.0 propose un langage graphique de modélisation permettant de décrire un système en terme de composants interconnectés. Ce langage normalisé permet l'échange de modèles architecturaux au même titre que certains ADLs. Le modèle CCM, décrit dans la section 2.2.5, est une extension du modèle des objets distribués CORBA pour supporter le développement par composants distribués.

Face à cette diversité, nous avons choisi de présenter :

(D)COM(+) pour des raisons historiques puisqu'il s'agit de l'un des premiers exemples qualifiés d'approche à composants,

Javabeans car ce modèle — souvent confiné à tort dans le domaine de la construction d'interfaces graphiques — est simple et a apporté selon nous un modèle d'assemblage puissant et novateur encore utilisé aujourd'hui,

EJB car cette approche est particulièrement utilisée actuellement dans les milieux industriels,

CCM car il s'agit d'une norme incontournable,

Fractal car il s'agit d'un modèle récent et unificateur actuellement très utilisé dans le monde académique,

SOFA car bien qu'il soit moins utilisé que Fractal, ce modèle académique propose une vision différente en se focalisant sur les connecteurs ou le remplacement dynamique de composants,

ArchJava car il s'agit d'une approche particulière axée sur le langage de programmation offert aux programmeurs pour implémenter effectivement les composants contrairement aux autres approches qui utilisent les langages de programmation existants,

UML puisqu'il constitue un standard de fait incontournable surtout depuis sa version 2.0 intégrant les structures composites.

Nous avons délibérément écarté l'étude d'un ADL d'une part car il est difficile d'en choisir un qui soit représentatif ; ils ont tous leurs spécificités. D'autres part, nous nous intéressons dans cette thèse plus à la programmation qu'à la description d'architectures. Toutefois, nous présentons les approches Fractal, SOFA ou encore ArchJava qui sont des approches hybrides intégrant des concepts issus des ADLs. Finalement, il est possible de trouver une comparaison détaillée des principaux ADL dans [Medvidovic et Taylor, 2000].

2.2.2 (D)COM(+)

Le modèle

COM (Component Object Model) [Microsoft, 1995; Rogerson, 1997] fut créé par Microsoft en 1995 afin de faire évoluer l'environnement MS-Windows vers les composants. COM est l'une des premières solutions techniques allant dans le sens du développement par composants dont la spécification formelle a été proposée par la suite [Ibrahim, 1998]. COM a connu de nombreuses évolutions notamment DCOM (Distributed COM) qui prend en compte l'aspect distribué des applications ou encore COM+ qui facilite l'intégration de COM et de Java.

COM est une spécification permettant de créer des entités logicielles binaires standardisées appelées des *composants*. COM n'impose pas de contrainte sur l'implémentation des composants mais sur le format binaire des composants. Il est ainsi possible d'implanter un composant COM en n'importe quel langage de programmation (toutefois, les environnements de programmation tels que Microsoft Visual C++ ou plus récemment C# [Hejlsberg *et al.*, 2003], facilitent grandement le développement de composants COM). L'implémentation d'un composant peut donc être une ou plusieurs classes, une bibliothèque de procédures ou de fonctions, etc.

En toute généralité, un composant COM possède une ou plusieurs *interfaces*. Une interface COM spécifie un ensemble de signatures de méthodes et possède les caractéristiques suivantes :

– elle possède un identifiant unique appelé IID (Interface IDentifier), un nombre de 128 bits généré par un algorithme pseudo-aléatoire, afin d'éviter les conflits ;
– elle est immuable et toute modification comme l'ajout, la modification ou le retrait d'une signature de fonction, est impossible ; Cette contrainte facilite la gestion de différentes versions d'une même interface puisqu'elles ont obligatoirement des identités différentes ;
– elle hérite directement ou indirectement de l'interface `IUnknown` ;
– elle est décrite en langage MIDL (Microsoft Interface Definition Language).

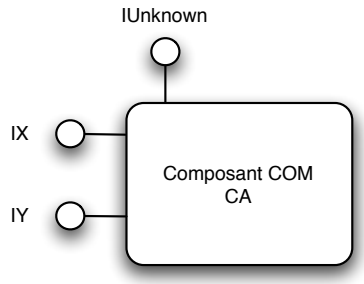

FIG. 2.5 : Représentation graphique d'un objet COM nommé A possédant deux interfaces IX et IY

Les *objets* COM (entité mémoire lors de l'exécution) sont des instances de classes qui ne peuvent être utilisés qu'à travers leurs interfaces. La figure 2.5 montre une représentation graphique d'un objet COM. COM spécifie qu'une interface d'un objet COM doit être un pointeur (accessible par les clients) vers une zone mémoire du composant appelée nœud interface (cf. figure 2.6). Le premier champ de cette zone mémoire est un pointeur vers une table de pointeurs de fonction. Cette table est aussi appelée *vtable* car elle est très similaire à une table des fonctions virtuelles utilisée par les compilateurs du langage C++.

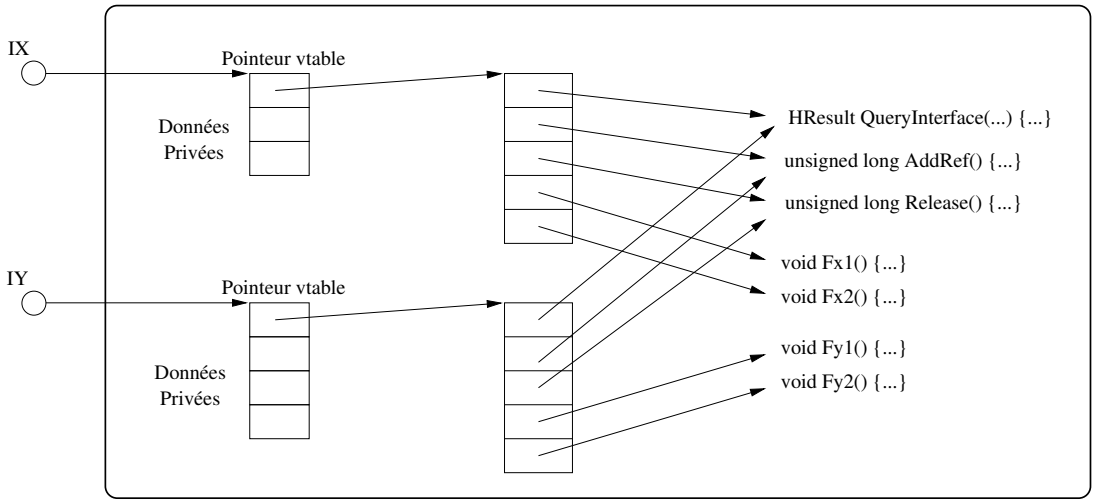

FIG. 2.6 : Organisation interne d'un objet COM ayant deux interfaces fournies IX et IY

Tous les composants possèdent implicitement l'interface `IUnknown` directement ou indirecte-

ment du fait de l'héritage entre interfaces. Ce point est à la base du modèle de communication entre composants COM puisqu'il est toujours possible d'accéder à un composant via cette interface qui offre notamment l'opération `queryInterface` qui permet la navigation entre les différentes *interfaces fournies* du composant. D'après la spécification COM, un composant peut aussi posséder des *outgoing interfaces* [Microsoft, 1995] ; un tel composant est dit *connectable*. Il s'agit d'interfaces déclarées par un composant mais pour lesquelles il est un client c'est-à-dire qu'il utilise les fonctions de ces interfaces. C'est ainsi qu'un composant peut être connecté à des instances de composants offrant ces interfaces (ou une interface dérivée). Toutefois, ces interfaces sont optionnelles, donc rarement utilisées et un composant COM accède directement à un autre composant via l'une de ses interfaces fournie. Le modèle COM supporte les communications synchrones (appels de méthodes) et les communications asynchrones via l'utilisation d'interfaces fournies standardisées. Le modèle de communication entre composants repose sur l'API (*Application Programming Interface*) COM qui fournit des opérations afin que les composants soient connectés c'est-à-dire que les références sur les interfaces soient échangées. La figure 2.7, extraite de la spécification COM [Microsoft, 1995], schématise le fonctionnement de l'API COM afin de permettre les communications entre les composants. Concrètement, cette API est très liée à l'environnement MS-Windows et à la notion de *serveur COM*. Un serveur COM est un fichier binaire (Executable, Dynamic Link Library ou OCX, bibliothèque intégrant une vue graphique) contenant l'implémentation de composants COM. Pour chaque composant qu'il encapsule, le serveur COM possède une fabrique (Design Pattern Factory [Gamma *et al.*, 1995]) chargée de créer des instances du composant. Tout serveur est muni d'un identifiant (CLSID) unique et est enregistré dans un annuaire (i.e la base de registres du système d'exploitation MS-Windows). Grâce à cet annuaire et à l'API COM, il est possible d'accéder dynamiquement et de façon transparente à un composant COM qu'il soit distant ou non.

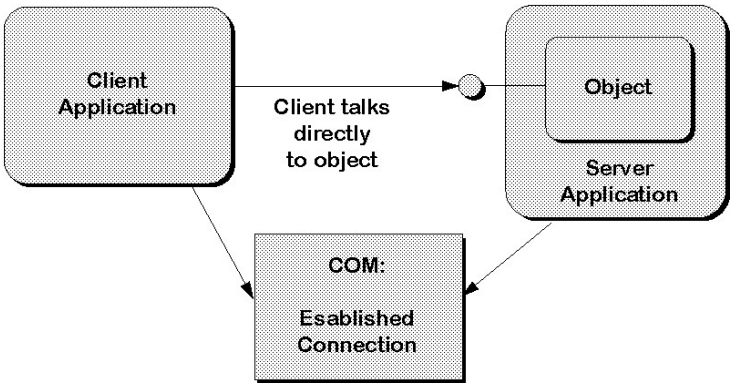

FIG. 2.7 : Principe de la connexion d'objets COM

La réutilisation en COM ne repose pas sur l'héritage d'implémentation. Un composant ne peut pas hériter de l'implémentation d'un autre. Les deux mécanismes de réutilisation de code lors de l'implémentation des composants (cf. figure 2.8 extraite de [Microsoft, 1995]) sont :

- Le *containment*. Un composant implémente certaines de ses opérations en invoquant les fonctions d'un sous-composant via l'une de ses interfaces dont il a stocké l'adresse.
- L'agrégation. Le composant exporte directement l'interface d'un sous-composant. Cette mé-

thode permet de meilleures performances mais nécessite de faire attention lors de l'implémentation. Par exemple, la méthode de navigation entre interfaces doit cacher le fait qu'il existe un sous-composant encapsulé et veiller aux problèmes d'identité.

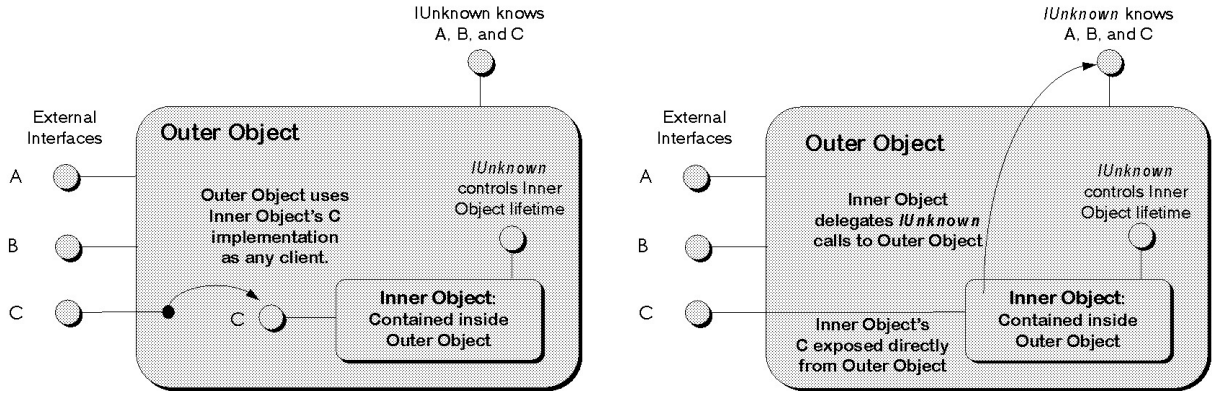

FIG. 2.8 : La réutilisation par *containment* (a) ou par agrégation (b) en COM

Un Exemple

Pour créer un composant, il faut d'abord déclarer ses interfaces en MIDL (*Microsoft Interface Definition Language*) comme l'illustre le listing 2.1. Ce langage standardise la définition des interfaces ainsi que les types de données de base tels que short, char, double ou float mais aussi les types structurés, les tableaux ou les énumérations. Cette normalisation des types permet de définir de nouveaux types de données en MIDL afin que tous les composants puissent les manipuler indépendamment de leur langage d'implémentation.

```
import "unknwn.idl"; // Importation de définitions existantes
[object,
uuid(a03d1420-b1ec-11d0-8c3a-00c04fc31d2f)] // Identifiant généré par Uuidgen.exe
interface IX : IUnknown
{
   HRESULT Fx1([in] short a); // in indique le sens passage de l'argument
   HRESULT Fx2(); // il existe il multitude de mot-clés qualifiant les attributs en MIDL
};
```

LISTING 2.1 : Exemple de fichier MIDL

A partir des déclarations MIDL des outils permettent de générer le code des interfaces et le squelette des implantations de composants dans un langage de programmation donné. La figure 2.2 montre un exemple de squelette C++ d'un composant `CA` possédant deux interfaces `IX` et `IY`. Les interfaces sont implantées en C++ par des classes abstraites et le composant par une classe C++ héritant des classes abstraites `IX` et `IY`. Cette classe contient l'implémentation de toutes les méthodes

déclarées dans IX, IY et IUnknown. Cette implémentation du composant CA en C++ permet d'obtenir une structuration du binaire conforme à la spécification COM. Les types utilisés dans ce programme C++ comme ULONG sont conformes à ceux standardisés par MIDL.

```
#include <windows.h>

static const IID IID_IX = ... ; // Les identifiants générés
static const IID IID_IY = ... ;

class IX : IUnknown {
public:
  virtual HRESULT Fx1(short a) = 0;
  virtual HRESULT Fx2() = 0;
};

class IY : IUnknown {
public:
  virtual HRESULT Fy1() = 0;
  virtual HRESULT Fy2() = 0;
};

// Implémentation du composant CA
class CA : public IX, public IY {
public:
   // Implémentation de l'interface IUnknown
   virtual HRESULT QueryInterface(const IID& iid, void** ppv) {
      if (iid == IID_IUnknown) {
         *ppv = static_cast<IUnknown*>(this);
      }
      else if (iid == IID_IX) {
         *ppv = static_cast<IX*>(this);
      }
      else if (iid == IID_IY) {
         *ppv = static_cast<IY*>(this);
      }
      else {
         *ppv = NULL ;
         return E_NOINTERFACE ; // Erreur, interface non supportée
      }

      reinterpret_cast<IUnknown*>(*ppv)->AddRef();
      return S_OK;
   }

   virtual ULONG  AddRef() { ... }
   virtual ULONG  Release() { ... }

   // Implement interface IX.
   virtual HRESULT Fx1(short a) { ... }
```

```
   virtual HRESULT Fx2() { ... }

   // Implement interface IY.
   virtual HRESULT Fy1() { ... }
   virtual HRESULT Fy2() { ... }
};
```

LISTING 2.2 : Code source de l'implémentation d'un composant COM CA ayant deux interfaces IX et IY

Une fois créé, compilé et packagé (sous la forme d'une DLL par exemple), le composant CA peut être enregistré dans la base de registres afin que les clients COM puissent l'utiliser de manière transparente. Cette inscription du composant est réalisée par un outil qui renseigne plusieurs champs lors de cette inscription et génère notamment un identificateur de classe (CLSID) permettant aux clients COM d'accéder à ce composant. La figure 2.3 montre le code source C++ d'un programme client utilisant la fonction `Fx` du composant `CA` à travers son interface `IX`.

```
#include <windows.h>
#include "IX.h"

// CLSID du composant que l'on veut utiliser
#define CLSID_ComponentCA 419F376-6520-4407-B8B8-740F5B353EDA

int main(void) {
  HRESULT hr;

  // Initialize COM Library
  CoInitialize(NULL) ;

  IX* pIX = NULL ;

  hr = CoCreateInstance( CLSID_ComponentCA, IID_IX, (void**) &pIX, ... );

  if ( SUCCEEDED(hr) ) {
    pIX->Fx2();          // call Fx through interface IX.
    pIX->Release();     // Decr reference counting for automatic deallocation
  }

  // Uninitialize COM Library
  CoUninitialize() ;

  return 0 ;
}
```

LISTING 2.3 : Code source d'un programme client utilisant le composant COM CA

Un client COM commence toujours par initialiser la librairie COM avec la fonction `CoInitialize`. Il peut ensuite utiliser des fonctions particulières telle que `CoCreateInstance` afin de récupérer une

référence sur une interface d'un composant particulier. Cette fonction interroge l'annuaire afin de localiser le serveur COM recherché. Une des fabriques intégrées au serveur crée ensuite une instance de composant implantant l'interface recherchée et une référence sur cette interface est retournée au client. Avec DCOM, le serveur peut être distant et la référence renvoyée au client n'est en réalité que l'adresse d'un *proxy*. Cela est complètement transparent pour le client qui utilise directement la référence d'interface obtenue pour invoquer les fonctions dont il a besoin.

Critique

COM est un des premiers modèles de composants visant à augmenter la réutilisation de code. Ce modèle apporte ou met en avant :

- La notion de composant binaire réutilisable indépendamment de tout langage de programmation,
- La notion d'interface qui permet à un composant d'offrir différents points de vue sur lui-même,
- Un mécanisme de navigabilité entre les interfaces d'un composant grâce à l'opération `QueryInterface`,
- Un mécanisme de réutilisation de code sans héritage,
- La transparence à la localisation des composants via un annuaire,
- L'efficacité car un client ayant obtenu une référence sur l'interface d'un composant peut communiquer directement avec ce dernier sans autre indirection que celle dans la table des fonctions virtuelles.

COM est une des premières réponses au besoin de composants logiciels réutilisables. Toutefois, ce modèle présente les faiblesses suivantes :

- L'intégration forte à l'environnement MS-Windows et aux outils de programmation Microsoft,
- Les interfaces *outgoing* ne sont pas utilisées systématiquement ce qui implique que dans la plupart des cas on ne peut pas considérer qu'il y ait une connexion entre les composants COM puisque le client accède directement à un composant fournisseur via l'une de ses interfaces fournies,
- Le caractère non diffusable d'un composant qui ne contient pas de documentation et qui ne peut pas être adapté facilement,
- Le déploiement, pourtant important, n'est pas spécifié et connaît donc des solutions *ad hoc* qui ont conduit au problème connu sous le nom de l'« enfer des DLL » (appelé *DLL Hell*) se produisant lorsqu'une application utilise des composants COM non répertoriés dans la base de registres MS-Windows,
- Les services non-fonctionnels tels que la persistance, la sécurité, etc. COM+ a répondu partiellement à ce besoin avec Microsoft Transaction Server (MTS) qui permet de gérer automatiquement les transactions.

2.2.3 Javabeans

Le modèle

Le modèle de composants *Javabean*[2] a été développé par Sun Microsystems en 1996 autour de son langage Java. Un *Javabean* est un « *composant logiciel réutilisable qui peut être manipulé graphiquement dans un environnement de développement* » [Hamilton, 1997]. Il faut bien comprendre que tous les *Javabeans* ne sont pas nécessairement des composants graphiques (appelés *widgets*) comme des boutons, des barres de menus, etc. Même si ce modèle est particulièrement bien adapté pour la construction d'interfaces graphiques, son utilisation peut être beaucoup plus large.

Un *Javabean* est une instance d'une classe Java, qui possède des attributs, des méthodes, des *propriétés* et peut émettre et recevoir des *événements* (*cf.* figure 2.9). Les attributs et les méthodes sont des concepts standards en Java, contrairement aux propriétés qui sont des « unités » sémantiques publiques qui affectent l'apparence ou le comportement d'un *Javabean*. Une propriété possède un nom, un type et une valeur qui est accessible en lecture et/ou écriture via des méthodes du *Javabean*. On distingue les propriétés qui n'ont qu'une seule valeur de celles qui ont plusieurs valeurs (*indexed properties*). Les événements sont des objets Java que les composants s'échangent lorsqu'ils sont connectés. Il existe une multitude d'événements prédéfinis et il est même possible d'en définir de nouveaux. Nous verrons dans la suite des exemples d'événements notamment relatifs aux changements de valeurs des propriétés.

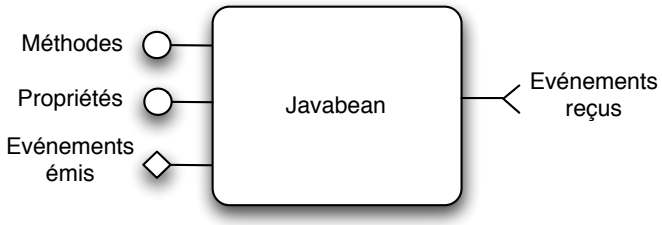

FIG. 2.9 : Structure d'un composant *Javabean*

La connexion de *Javabeans* repose sur la notification et l'écoute d'événements. Chaque *Javabean* écoute un ensemble d'événements auxquels il peut réagir et par ailleurs notifie ses écoutants d'un ensemble d'événements qu'il produit. En résumé, deux *Javabeans* développés indépendamment peuvent être connectés (sans modifier leur code) pourvu que l'un produise et l'autre reçoive des événements de type compatible. La figure 2.10 montre deux connexions entre un *Javabean* `Counter` et ses deux vues, au sens du modèle MVC [Krasner et Pope, 1988] (*Model View Controller*), qui sont aussi des *Javabeans*. Les vues sont enregistrées en tant qu'écouteur du bean `Counter` et reçoivent ainsi les événements signalant ses changements.

Dans l'exemple précédent, le bean `Counter` possède une propriété nommée `value`. Grâce au mécanisme événementiel, un *Javabean* peut notifier les changements de valeur de ses propriétés. De façon non-exclusive, une propriété est dite *liée* (*Bound*) lorsque son bean émet un événement à chaque

[2] Le sens premier du mot anglais *Bean* est Haricot, toutefois, il est ici utilisé avec le sens « grain » donc un *Javabean* est un grain de café.

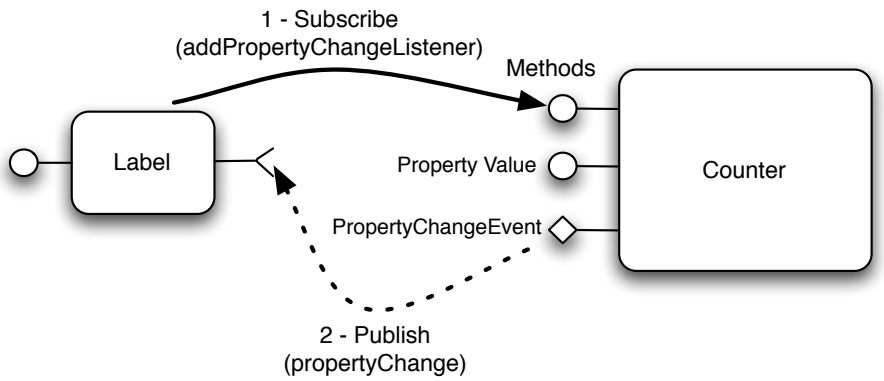

FIG. 2.10 : Connexion d'un *Javabean* LABEL à un *Javabean* COUNTER

fois que sa valeur a été modifiée. Les propriétés liées sont particulièrement utiles pour établir des connexions entre un objet métier et ses vues afin qu'elles se mettent à jour lors de la réception des événements notifiant les changements de valeur de la propriété métier. Cette technique permet de bien séparer le code métier et le code de l'interface tout en garantissant la cohérence à tout instant des informations représentées par les vues. Une propriété est dite *veto* (*Vetoable*) lorsque son bean émet un événement à chaque fois que sa valeur va être modifiée afin de permettre aux écouteurs de s'opposer aux modifications. On dit que les écouteurs ont un droit de *veto* sur les changements de valeur d'une telle propriété. Ces propriétés sont principalement utilisées dans les systèmes concurrentiels, afin de prévenir certains risques d'incohérence lors d'accès en lecture et écriture simultanés.

Un bean possède également un mécanisme standard d'introspection qui fournit, à travers son interface `BeanInfo`, des informations sur lui-même (ses propriétés, les différents types d'événements écoutés et notifiés, ses méthodes). Ce mécanisme d'introspection permet de prendre connaissance de la structure d'un *Javabean* pendant son exécution. Les environnements de développement proposés par Sun comme le *Bean Development Kit* (BDK) ou Netbeans utilisent ce mécanisme afin de fournir des représentations graphiques d'un *Javabean* (fenêtre d'édition de propriétés, connexion de beans graphique, etc).

Un bean sur étagère se présente sous la forme d'une archive (jar) contenant l'implémentation du bean (fichiers java compilés) et des fichiers ressources (fichiers de configuration, images, etc). Cette archive peut alors être facilement déployée. L'exécution d'un *Javabean* est supportée par la machine virtuelle Java qui joue le rôle de conteneur. Cela rend très portables les *Javabeans* puisqu'il existe aujourd'hui une machine virtuelle Java pour la plupart des plateformes matérielles.

Un exemple

La programmation d'un *Javabean* se fait de manière analogue à celle d'une classe Java, mais certains mécanismes spécifiques sont définis selon le schéma de conception « Observateur » [Gamma *et al.*, 1995] et un ensemble de règles de nommage, comme illustré par le code source 2.4.

Pour chaque propriété, un bean possède un ensemble de méthodes respectant des règles de

```java
package compteurbean;

import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.io.Serializable;

// classe d'implémentation du bean Counter
public class Counter implements Serializable {

   protected int value;
   protected PropertyChangeSupport support; // gestionnaire des écouteurs

   public Counter() {
       support = new PropertyChangeSupport(this);
      value = 0;
   }

   public int getValue() { return value; }

   public void setValue(int value) {
      Integer oldValue = new Integer ( this.value );
      this.value = value;

      // signalement d'un changement de valeur de la propriété Value
      // via le gestionnaire d'écouteurs
      support.firePropertyChange( new PropertyChangeEvent( this,
             "value",
             oldValue,
             new Integer( this.value ) ) );
   }

   public void incr() { this.setValue( this.getValue() + 1 ); }
   public void decr() { this.setValue( this.getValue() - 1 ); }
   public void raz() { this.setValue( 0 ); }

   // délégation de l'ajout d'un écouteur
   public void addPropertyChangeListener( PropertyChangeListener cl ) {
      support.addPropertyChangeListener( cl );
   }

   // délégation du retrait d'un écouteur
   public void removePropertyChangeListener( PropertyChangeListener cl ) {
      support.removePropertyChangeListener( cl );
   }
}
```

LISTING 2.4 : Implémentation d'un *Javabean* COUNTER possédant une propriété liée nommée `Value`

nommage permettant d'accéder et modifier la valeur de cette propriété. S'il s'agit d'une propriété mono-valuée, il faut définir les méthodes `PropertyType get<PropertyName>()` et `void set<PropertyName>(<PropertyType>)` comme c'est le cas pour la propriété `Value`. Dans le cas des propriétés multi-valuées, les conventions sont différentes puisqu'elles ajoutent un index pour désigner un élément de la collection.

Le mécanisme de notification d'événement suit le patron « Observateur ». Pour chaque type d'événements qu'il émet, un *Javabean* possède ainsi une méthode d'abonnement (dont le nom suit la convention de nommage `add<EventName>Listener`), une méthode de désabonnement (dont le nom suit la convention de nommage `remove<EventName>Listener`). Il peut aussi définir une méthode de notification des écouteurs ou bien comme dans cet exemple utiliser directement la méthode `firePropertyChange` de l'objet `support` qui facilite l'implémentation des événements *Javabeans*. La notification consiste à instancier la classe d'événements, l'initialiser avec les informations adéquates et transmettre cet événement aux écouteurs. Dans cet exemple, une instance de la classe `PropertyChangeEvent` est créée (dans l'implémentation de méthode `firePropertyChange` qui n'est pas montrée dans le listing précédent) et initialisée avec l'identité de l'objet émetteur, le nom de la propriété dont la valeur a changé ainsi que son ancienne et sa nouvelle valeur.

Pour chaque type d'événements qu'il peut recevoir, la classe du bean implémente une interface sous-type de `EventListener`. Dans le listing 2.5, la classe `MyLabel` implémente l'interface `PropertyChangeListener` afin de recevoir les événements signalant les changements de valeur de propriétés et implémente le(s) méthode(s) de traitement de ce type d'événements définies par cette interface c'est-à-dire la méthode `PropertyChange`.

```
package mygui;

import java.io.Serializable;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeEvent;

// classe d'implémentation du bean MyLabel
public class MyLabel implements Serializable, PropertyChangeListener {
   //...
   void propertyChange(PropertyChangeEvent evt) {
      // Mise à jour de la représentation graphique
   }
}
```

LISTING 2.5 : Implémentation d'un *Javabean* MYLABEL pouvant recevoir des événements de type `PropertyChangeEvent`

La connexion entre deux beans peut être établie par une tierce entité c'est-à-dire en dehors de l'implémentation de ces deux beans. Le code source 2.6 montre le code d'un programme principal établissant la connexion entre deux *Javabeans* `Counter` et `CounterView`.

Un bean peut donc être connecté à n'importe quel autre bean sans aucune modification de leurs implémentations pourvu qu'ils émettent et écoutent des événements de types compatibles. On peut

```
public static void main(String args[]){
   Counter c = new Counter();
   MyLabel l = new MyLabel();
   c.addPropertyChangeListener(l);
}
```

LISTING 2.6 : Programme principal établissant une connexion entre deux composants *Javabeans*

même s'affranchir de cette dernière contrainte grâce à la génération automatique d'un adapteur. C'est d'ailleurs le mode de fonctionnement de l'environnement BDK (Bean Development Kit) proposé en 1997. Pour établir une connexion, un adapteur est généré. Cet adapteur est spécifiquement conçu pour recevoir les événements émis par le bean émetteur et déclencher un traitement sur le bean écouteur. En fait, c'est l'adapteur généré qui est enregistré en tant qu'écouteur du bean émetteur. A chaque réception d'un événement, l'adapteur peut envoyer un message au bean récepteur.

Critique

Le modèle de composants *Javabeans* permet de programmer des composants séparément puis de les connecter (sans les modifier) afin qu'ils accomplissent une tâche et collaborent. Bien qu'il soit peu reconnu, ce modèle a pourtant apporté un modèle de connexion puissant et souple. En effet, la connexion entre deux beans peut être établie par une tierce entité, c'est-à-dire en dehors de l'implémentation de ces deux beans. De plus, via la génération automatique d'adapteurs, aucune relation de compatibilité entre deux *Javabeans* n'est nécessaire pour pouvoir les connecter.

Le modèle *Javabeans* est très primitif et ne supporte que peu de concepts. Il n'y a pas de notion de service requis ou d'interface requise. Un *Javabean* ne peut que fournir des méthodes et des événements via l'enregistrement et la notification. Ce modèle d'assemblage puissant n'est cependant pas suffisant pour définir des applications complexes nécessitant des protocoles de communication autres que celui défini par le schéma de conception « Observateur ». Par ailleurs, il repose essentiellement sur des conventions de nommage en Java ce qui le confine à ce langage de programmation contrairement à d'autres propositions comme COM qui s'est affranchi de tout langage de programmation.

2.2.4 Enterprise JavaBeans (EJB)

Le modèle

Le modèle de composants Enterprise JavaBeans (EJB) constitue le cœur de la plateforme J2EE (Java 2 Enterprise Edition) de Sun permettant de concevoir facilement des applications distribuées. Avant de détailler ce qu'est un *EJB*, la figure 2.11 présente l'architecture générale d'une application J2EE découpée en trois couches logiques (architecture 3-*tiers*) :

1. La couche cliente (*client tier*) qui comprend les clients légers (*thin client*) et les clients lourds (*fat clients*) exécutés par les utilisateurs. Un client léger est un ensemble de pages web générées

dynamiquement et consultables à travers un navigateur. Un tel client est généralement programmé par des JSP (Java Server Page) ou des servlets accessibles via un serveur Web à travers des communications HTTP. Un client lourd est une application standard exécutée par l'utilisateur qui communique à travers le réseau. Ce type de client est de moins en moins utilisé à cause des contraintes de maintenance et de déploiement qu'il impose. En effet, un client léger est toujours à jour puisque qu'il s'agit de pages Web générées dynamiquement et il n'impose aucune configuration (installation, type de matériel, etc.) particulière sur le poste client si ce n'est la présence d'un navigateur.

2. La couche métier (*business tier*) comprend les EJB qui s'exécutent au sein d'un *conteneur* qui est lui-même supervisé par un *serveur*. Un conteneur gère le cycle de vie des instances d'un EJB. Le serveur fournit au conteneur des services non fonctionnels comme la persistance, la gestion des transactions ou encore la sécurité. Cette couche comprend aussi les éléments permettant la génération des clients légers (serveur web, moteur de servlets, etc.) parfois appelé *web-tier*.
3. La couche données (*Enterprise Information System (EIS) tier*) comprend la gestion des ressources comme les systèmes de gestion de bases de données (SGBD), les progiciels de gestion intégrés (PGI), etc.

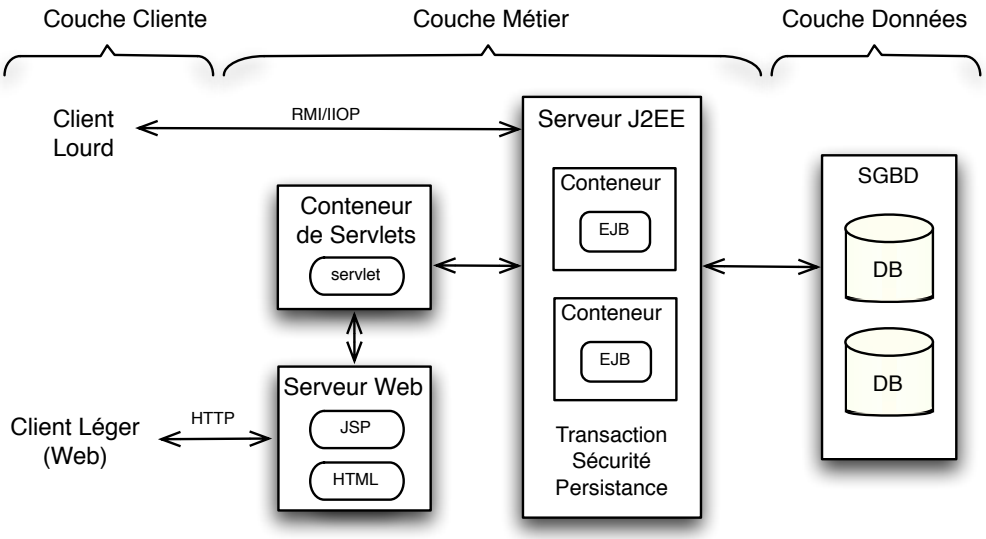

FIG. 2.11 : Les trois niveaux (3-*tiers*) d'une application J2EE

Depuis la version 2.0 de la spécification des EJB, la structure externe d'un EJB (cf. figure 2.12) comprend deux paires d'interfaces. La première paire correspond à des interfaces pour les communications distantes (`Remote`) et la seconde pour les communications locales (`Local`). Chaque paire est constituée :

– d'une *interface maison* permettant sa configuration lors du déploiement, l'accès à ses métadonnées, ainsi que la gestion du cycle de vie de ses instances (création, destruction, recherche en cas de persistance, etc.),

– d'une *interface métier* (ou fonctionnelle) spécifiant les méthodes qu'offre une instance de cet EJB à ses clients.

Ainsi, un EJB peut offrir des services différents à ses clients locaux (s'exécutant sur la même machine) et distants. De plus, les communications via les interfaces locales bénéficient d'une implémentation plus efficace puisque la gestion du réseau n'est pas prise en compte.

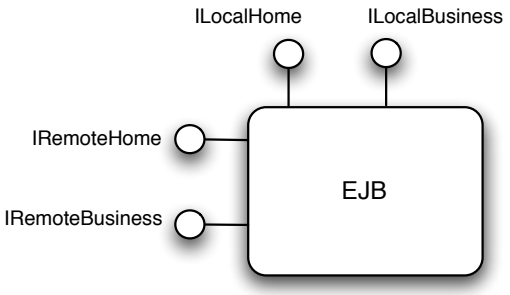

FIG. 2.12 : Structure externe d'un EJB

Il existe trois sortes de beans : les beans session (*session beans*), les beans entités (rebaptisés *entity classes* à partir de la version 3.0) et les beans orientés messages (*message-driven beans*).

Une instance d'un bean session ne peut être utilisée simultanément que par un seul client. Ce type d'EJB permet d'intégrer des fonctionnalités spécifiques à certains clients dans la couche métier. Il existe deux sortes d'EJB session : ceux sans état (*stateless*) et ceux avec état (*stateful*). Un bean session sans état fournit donc un ensemble de services qu'un client invoque indépendamment les uns des autres. Sans état signifie que le bean ne sauvegarde aucun état entre deux requêtes d'un client et le serveur d'EJB peut alors gérer des réserves d'instances (*pools*) afin d'optimiser au mieux les requêtes des clients. Un bean session avec état est souvent dédié à l'accomplissement d'un traitement spécifique nécessitant plusieurs étapes. Par exemple, un bean `WebOrder` pourrait fournir des méthodes permettant d'ajouter et retirer des éléments d'un panier, de valider le panier et enfin de procéder au paiement. Ce bean session avec état ne doit communiquer qu'avec un seul client et maintenir l'état du panier.

Les beans entités peuvent être utilisés simultanément par plusieurs clients, les problèmes de concurrence étant gérés automatiquement par le conteneur du bean. Ce type de beans permet de représenter des concepts métiers (e.g une personne, un compte) dont l'état peut être stocké dans une base de données soit automatiquement pour les beans entité CMP (*Container-Managed Persistency*), soit explicitement par le programmeur pour les beans entités BMP (*Bean-Managed Persistency*).

Les beans orientés messages n'ont pas d'état et sont dédiés à la réception et au traitement de messages asynchrones délivrés par l'API JMS (Java Message Sevice). Ce type de beans, introduit dans la version 2.0 de la spécification EJB, ne présente pas la même structure que les beans sessions et les beans entités. En effet, ces beans ne possèdent pas d'interface maison, ni d'interface métier et ne sont d'ailleurs jamais accédés directement.

Un exemple

L'exemple développé dans cette section utilise l'implémentation de référence de la spécification J2EE 2.0[3] fournie par Sun. L'implémentation d'un bean s'effectue par extension du *framework* J2EE comme indiqué sur la figure 2.13. L'interface maison d'un EJB étend l'interface `javax.ejb.EJBHome`, son interface métier distante étend l'interface `javax.ejb.EJBObject`. Un EJB est implémenté par une classe implémentant l'interface qui correspond au type d'EJB souhaité `javax.ejb.SessionBean` ou `javax.ejb.EntityBean`.

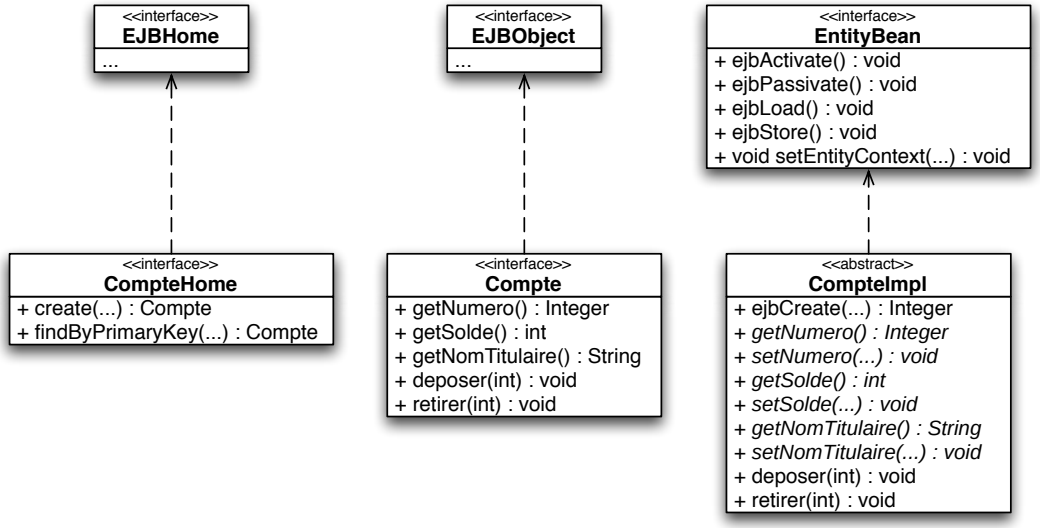

FIG. 2.13 : Diagramme de classes UML d'un EJB COMPTE

Les listings 2.7, 2.8 montrent l'implémentation d'un EJB `Compte`. L'interface maison définit la signature d'une méthode de création de l'EJB `Compte` (cf. ligne 3 du listing 2.7) et la signature d'une méthode de recherche d'instance existante en fonction d'un identifiant (cf. ligne 7 du listing 2.7). L'interface métier de cet EJB (cf. ligne 11 du listing 2.7) définit les méthodes fournies par une instance telles que : `getNumero`, `getSolde` ou `getNomTitulaire`. Ces méthodes sont déclarées comme pouvant lever des exceptions définies dans la spécification EJB comme `RemoteException` signalant les problèmes de réseaux ou `CreateException` signalant les problèmes d'instanciation de l'EJB. Il est aussi possible de définir ses propres exceptions comme nous l'avons avec `OperationImpossible` signalant qu'une opération sur un compte n'est pas réalisable.

La classe abstraite `CompteImpl` constitue une « partie » de l'implémentation de l'EJB entité `Compte`. Le reste de l'implémentation d'un EJB est généré par des outils lors de son empaquetage et de son déploiement en fonction des paramètres spécifiés dans son descripteur de déploiement. Par exemple, le code des accesseurs (`setNumero`, `getNumero`, etc.) sera généré en fonction du mode de persistance choisi. De même le code de la méthode `create` déclarée dans l'interface maison de

[3]La version 3.0 de la spécification a grandement simplifié le code nécessaire à l'implémentation d'un EJB grâce aux possibilités introduites en Java 5 comme les annotations.

```
1  public interface CompteHome extends EJBHome {
     /** Méthode de création */
3    public Compte create( Integer numeroCpte, String nomTitulaire, int soldeInit )
       throws CreateException, RemoteException;
5
     /** recherche un compte à partir de son numéro */
7    public Compte findByPrimaryKey( Integer numeroCpte )
       throws FinderException, RemoteException;
9  }

11 public interface Compte extends EJBObject {
     /** retourne le numéro identifiant le compte */
13   public Integer getNumero() throws RemoteException;

15   /** retourne le solde courant du compte  */
     public int getSolde() throws RemoteException;
17
     /** retourne le nom du titulaire du compte */
19   public String getNomTitulaire() throws RemoteException;

21   /** dépose la somme val sur le compte (val > 0)  */
     public void deposer( int val ) throws RemoteException, OperationImpossible;
23
     /** retire la somme val du compte (val > 0) */
25   public void retirer( int val ) throws RemoteException, OperationImpossible;
   }
```

LISTING 2.7 : Définition des interfaces distantes maison et métier d'un EJB COMPTE

l'EJB n'est pas défini dans sa classe d'implémentation. Une partie de l'implémentation de cette méthode est fournie par la définition de la méthode `ejbCreate`. Les méthodes déclarées dans l'interface `EntityBean` permettent au programmeur du composant de définir ces méthodes et ainsi d'effectuer des traitements particuliers au cours du cycle de vie de l'EJB. Par exemple, la méthode `ejbActivate` sera invoquée lors du transfert en mémoire d'une instance.

Le descripteur de déploiement de l'EJB `Compte` est présenté sur le listing 2.9. Il s'agit d'un fichier XML (heureusement généré par des outils) décrivant d'une part les caractéristiques générales du composant et d'autre part son assemblage avec le serveur J2EE en spécifiant ses besoins du point de vue des services non-fonctionnels (transaction, sécurité, etc.).

Le listing 2.10 montre le code source d'une application cliente de l'EJB `Compte`. Pour qu'un programme client (distant ou non) utilise cet EJB `Compte`, il doit obtenir une référence vers son interface maison afin de rechercher ou créer des instances. Le programme obtient cette référence en interrogeant le service de nommage (Java Naming Directory Interface, JNDI, dans le cas de J2EE). Un tel service permet, d'une part, l'enregistrement d'une ressource distribuée (ici un EJB) dans un annuaire sous un identifiant et d'autre part de rechercher une ressource à partir d'un identifiant.

```
public abstract class CompteImpl implements EntityBean {

   public Integer ejbCreate( Integer numeroCpte, String nomTitulaire, int soldeInit )
   throws CreateException {
      setNumero( numeroCpte );
      setNomTitulaire( nomTitulaire );
      setSolde( soldeInit );
      return null;
   }

   void ejbActivate() { } // méthodes de l'interface EntityBean
   void ejbLoad() { }
   void ejbPassivate() { }
   void ejbRemove() { }
   void ejbStore() { }
   void setEntityContext(EntityContext ctx) { }
   void unsetEntityContext() { }

   public abstract void setNumero( Integer pk );
   public abstract Integer getNumero();

   public abstract String getNomTitulaire();
   public abstract void setNomTitulaire( String nomtit );

   public abstract void setSolde( int s );
   public abstract int getSolde();

   /** retire la somme val (> 0) du compte  */
   public void retirer( int val ) throws OperationImpossible {
      if ( val < 0 )
         throw new OperationImpossible("Retirer une somme negative est impossible !");
      int currentSolde = this.getSolde();
      if ( val > currentSolde )
         throw new OperationImpossible("Solde insuffisant pour ce retrait");
      this.setSolde( currentSolde - val );
   }

   /** depose la somme val (> 0) sur le compte  */
   public void deposer( int val ) throws OperationImpossible {
      if ( val < 0 )
         throw new OperationImpossible("Deposer une somme negative est impossible !");
      this.setSolde( val + this.getSolde() );
   }
}
```

LISTING 2.8 : La classe d'implémentation de l'EJB COMPTE

```
<?xml version="1.0"  encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
  <display-name>compteEJB</display-name>
  <enterprise-beans>    <!-- Description des EJB contenus cette archive -->
    <entity>   <!-- Un EJB entité -->
      <display-name>CompteBean</display-name>
      <ejb-name>CompteBean</ejb-name>
      <home>ejb.tp2.banque.CompteHome</home>    <!-- Les définitions de ses interfaces -->
      <remote>ejb.tp2.banque.Compte</remote>
      <ejb-class>ejb.tp2.banque.CompteImpl</ejb-class>
      <persistence-type>Container</persistence-type> <!-- Politique de persistance CMP -->
      <prim-key-class>java.lang.Integer</prim-key-class> <!-- Type de la clé primaire -->
      <reentrant>False</reentrant>  <!-- Politique de concurrence -->
      <cmp-version>2.x</cmp-version>
      <abstract-schema-name>MaBanque</abstract-schema-name> <!-- Nom table dans la BD -->
      <cmp-field>    <!-- Description des champs de la table -->
        <description>Titulaire du compte</description>
        <field-name>nomTitulaire</field-name>
      </cmp-field>
      <cmp-field>
        <description>Identifiant du compte</description>
        <field-name>numero</field-name>
      </cmp-field>
      <cmp-field>
        <description>Solde courant</description>
        <field-name>solde</field-name>
      </cmp-field>
      <primkey-field>numero</primkey-field>
      <security-identity>
        <description></description>
        <use-caller-identity></use-caller-identity>
      </security-identity>
    </entity>
  </enterprise-beans>
  <assembly-descriptor>    <!-- Caractéritiques d'assemblages avec le conteneur -->
    <container-transaction>   <!-- Politique transactionnelle -->
      <method>
        <ejb-name>CompteBean</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>retirer</method-name>
        <method-params><method-param>int</method-param></method-params>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
     [...]
  </assembly-descriptor>
</ejb-jar>
```

LISTING 2.9 : Exemple de descripteur de déploiement de l'EJB COMPTE

```
public class SimpleCompteClient {
   public static void main ( String [] args ) {
      // obtention du service d'annuaire JNDI
      Context ctx = new InitialContext( new Properties() );

      // obtention d'une référence sur un EJB Compte à travers son interface maison
      Object ref = ctx.lookup("java:comp/env/ejb/EJBCompte");

      // spécialisation en fonction du type du composant
      CompteHome compteHome = (CompteHome)
         PortableRemoteObject.narrow( ref, CompteHome.class );

      try{
         // création d'une instance de l'EJB Compte
         Compte compte = compteHome.create( new Integer(numcpt), "toto", 1000000 );

         // récupération du compte numéro 4
         Compte compte2 = compteHome.findByPrimaryKey(new Integer(4))
      }
      catch(Exception) {
         // Gestion des erreurs
      }
   }
}
```

LISTING 2.10 : Exemple de programme client de l'EJB COMPTE

Critique

Les avantages de ce modèle sont :

- la mise à disposition de services non-fonctionnels que le programmeur n'a pas à écrire comme : la persistance, la gestion des transactions, la sécurité, etc ; il suffit de déclarer et de paramétrer ces services via les fichiers de configuration ou les descripteurs de déploiement des EJB ;
- l'indépendance vis-à-vis d'un système d'exploitation, une machine virtuelle Java étant disponible sur la majorité des systèmes actuels ;
- la standardisation d'unités de déploiement (.jar) qui permettent l'encapsulation et le déploiement des composants sous une forme indépendante contenant toutes les ressources internes et nécessaires au fonctionnement des EJB.

Les principales limitations de cette approche sont les suivantes :

- L'architecture d'une application est cachée dans le code des EJB et des applications clientes. Le code d'interaction entre deux entités est noyé dans le code de l'entité cliente comme dans les programmes à objets. Cette limitation vient du fait qu'il est impossible d'exprimer les interfaces requises d'un EJB ou encore d'expliciter des connecteurs.
- Le manque de standardisation au niveau de l'infrastructure conduit souvent à intégrer dans l'implémentation d'un EJB du code spécifique à un serveur particulier. Cela pose des problèmes

de réutilisation des EJB avec d'autres serveurs.
- L'impossibilité d'ajouter facilement des services non-fonctionnels puisqu'ils sont rendus directement par le serveur J2EE qui ne propose généralement pas de mécanisme d'extension.
- La restriction à deux interfaces métiers (locale et distante) contrairement à un composant COM qui peut offrir plusieurs interfaces métiers caractérisant différentes utilisations.

En définitive, nous pensons que la plate-forme J2EE est une technologie pour construire et maintenir facilement des applications distribuées. Toutefois, cette approche ne remplit pas, selon nous, les critères permettant de la classer dans les approches à composants notamment parce que les EJB ne déclarent pas d'interfaces requises ce qui ne permet pas de faire réellement de la connexion d'EJB.

2.2.5 Corba Component Model (CCM)

Le modèle

CORBA (*Common Object Request Broker Architecture*) est une norme définie par l'OMG[4], dont l'objectif est de fournir des mécanismes permettant le développement d'applications distribuées en s'affranchissant des problèmes de communication, d'hétérogénéité, d'intégration et d'interopérabilité [Object Management Group, 2002]. Les premières normes CORBA ont spécifié un intergiciel pour objets distribués qui supporte l'interopérabilité pour des objets distants et hétérogènes (au sens des langages de programmation et des systèmes d'exploitation). La version 1.0 de la norme, datant de 1999, s'appuie sur le modèle orienté objet (héritage, encapsulation et polymorphisme) et l'architecture client/serveur. Cette version intègre le langage OMG IDL (*Interface Definition Language*) qui permet de décrire les interfaces des objets distribués et utilise le protocole générique GIOP (*General Inter-ORB Protocol*) pour les communications distantes, dont l'instanciation la plus utilisée est IIOP (*Internet Inter-ORB Protocol*) qui repose sur le protocole TCP/IP (*Transmission Control Protocol/Internet Protocol*). Toutefois, les aspects de diffusion, de déploiement, et de construction d'une application complète ne sont pas abordés et c'est dans cette optique que la norme Corba 3.0[5] a introduit un modèle de composants explicite nommé CCM (Corba Component Model) [Object Management Group, 2002] qui est une évolution structurelle des objets distribués Corba. Les composants s'exécutent au sein d'un conteneur qui s'exécute lui-même au sein d'un serveur (de même que dans le modèle EJB). Les communications entre les conteneurs s'effectuent via le bus CORBA qui offre aussi un accès aux services non-fonctionnels.

Un composant CCM encapsule sa représentation interne et son implémentation pour n'offrir à ses clients qu'une surface opaque. La figure 2.14 présente les éléments structurels d'un composant CCM. Un composant est constitué d'attributs permettant sa configuration et de *ports* chacun étant associé à une *interface* décrite en IDL. On distingue quatre sortes de ports : *facettes* (interfaces offertes), les *réceptacles* (interfaces requises), les *sources d'événements* et les *puits d'événements* qui sont des interfaces événementielles.

Un composant CCM offre ses services à travers ses facettes. Elles permettent à différents clients de disposer de l'interface dont ils ont besoin sur un composant et seulement de celle-ci. L'interface

[4] L'(*Object Management Group*) est un groupe constitué d'industriels et de chercheurs.
[5] Cette section décrit la version 3.0 de la norme CCM bien qu'une version 4.0 soit disponible depuis l'année 2006.

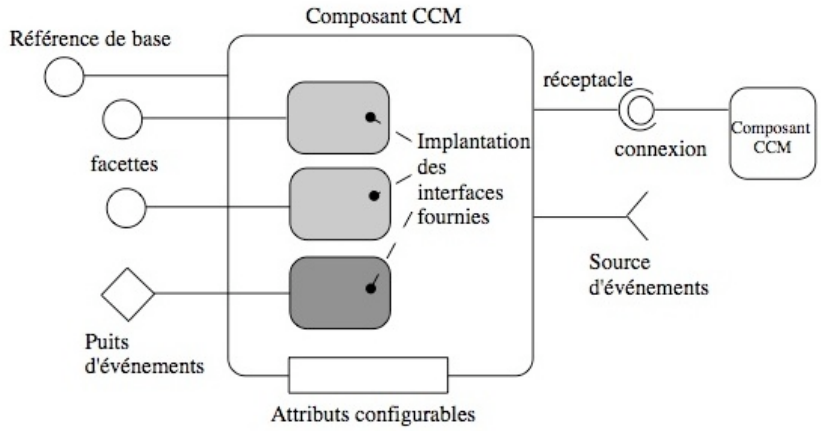

FIG. 2.14 : Structure d'un composant CCM

associée à une facette indique les services (méthodes) fournis. Un service peut appartenir à plusieurs facettes. Il est admis qu'un client change de point de vue sur un composant et donc change de facette, pour cela, il utilise les opérations de navigation entre les multiples facettes qui sont offertes par la *référence de base* du composant. Il s'agit d'une facette qui fournit un point d'entrée vers les possibilités d'un composant (navigation entre les ports, accès aux attributs, aux fonctionnalités, etc.). Un composant possède aussi une *facette de fabrique* (*Component Home*) fournissant les opérations de gestion des instances comme pour les EJB.

Les réceptacles permettent à un composant d'exprimer ses dépendances vis-à-vis d'autres composants fournisseurs de services via leurs facettes. Facettes et réceptacles sont donc les points de connexions complémentaires des composants CCM. Un réceptacle peut accepter une ou plusieurs référence(s) sur des facettes de composants suivant qu'il est multiple ou simple. Lorsqu'un composant reçoit sur l'un de ses réceptacles une référence, cela lui permet d'utiliser toutes les fonctionnalités offertes par le composant (ou l'interface) désigné par cette référence. C'est-à-dire qu'il peut invoquer un service ou s'abonner aux événements levés par ce dernier via ses sources d'événements. Deux opérations de base sont fournies sur un réceptacle : la connexion et la déconnexion. La connexion d'une facette et d'un réceptacle (ou plusieurs dans les cas multiples) suppose une compatibilité de leurs interfaces c'est-à-dire que l'interface de la facette soit un sous-type de l'interface du réceptacle.

Les sources et les puits d'événements sont les homologues respectifs des facettes et des réceptacles dédiés aux communications asynchrones par événements entre les composants. Une source d'événements indique qu'un composant émet un certain type d'événements. Un puits d'événements est rendu public par un composant afin de recevoir des événements d'un certain type en provenance d'un ou plusieurs producteurs. De même que précédemment, suivant l'arité de la source d'événements, un ou plusieurs clients peuvent y souscrire. Les connexions entre une source et un (ou plusieurs) puits sont aussi soumises à la contrainte du sous-typage c'est-à-dire que l'interface de la source doit être un sous-type de l'interface du puits. Ce mécanisme événementiel est à rapprocher de celui du modèle *Javabeans*.

On distingue quatre sortes de composants CCM : les composants *service* (sans état), les composants *sessions* (état non persistant), les composants *process* (état persistant mais pas de référence unique accessible) et les composants *entity* (état persistant et référence unique accessible).

Un exemple

Les interfaces des composants sont décrites en utilisant le langage OMG IDL issu des précédentes normes CORBA dédiées aux objets distribués. De même que pour MIDL (*cf.* section 2.2.2), le langage OMG IDL a pour objectif de masquer l'hétérogénéité en décrivant de façon standardisée les interfaces des objets distribués. OMG IDL standardise ainsi le format binaire des types de données de base, la définition de nouveaux types de données, la déclaration des opérations, etc. La déclaration d'une opération est (entre autres) constituée de son nom, du type du résultat, de la liste de ses paramètres, chacun possédant un type et un mode de passage (`in`, `out`, `inout`) ainsi que de la liste des exceptions qu'elle peut lever. En résumé OMG IDL est un langage de description d'interfaces standardisé complet qui permet l'interopérabilité entre les objets ou les composants communiquants via l'intergiciel CORBA.

Dans la suite, nous présentons un exemple adapté de [Riveill et Merle, 2000]. Le listing 2.11 montre la définition en IDL 3.0 d'un composant DISTRIBUTEUR. Le composant `Distributeur` a trois facettes : `fClient`, `fFournisseur` et `fDepanneur`. La première est destinée aux utilisateurs du distributeur et fournit des méthodes telles que l'insertion de la monnaie ou le choix de la boisson. La deuxième est destinée aux fournisseurs et fournit des méthodes permettant d'ouvrir le distributeur, d'ajouter des matières premières ou de vider le monnayeur. Et enfin la troisième s'adresse aux dépanneurs qui peuvent ouvrir le distributeur mais aussi remplacer des pièces.

Le listing 2.11 montre une description du composant DISTRIBUTEUR complètement indépendante de toute implémentation. C'est le CIF (*Component Implémentation Framework*) qui définit la manière d'implémenter un composant. Le CIF inclut notamment le langage CIDL (*Component Implementation Description Language*) permettant de décrire des implémentations de composants et de leurs maisons. Le listing 2.12 montre un exemple de déclaration CIDL qui précise des choix d'implantation du composant comme sa catégorie (ici entity) et le nom des deux classes C++ `MaisonDistributeurImpl` et `DistributeurImpl` contenant son implémentation (que nous ne détaillons pas ici).

```
module MonApplicationDistributeurImpl {
   composition entity DistributeurImpl {
      home executor MaisonDistributeurImpl {
         implements MonApplicationDistributeur::MaisonDistributeur;
         manages DistributeurImpl;
      };
   };
};
```

LISTING 2.12 : Déclaration CIDL d'un composant CCM

Le CIF fournit aussi des outils de génération de code permettant de générer des implémentations partielles (squelettes de code) de composants à partir de descriptions faites en CIDL (*Component*

```
module MonApplicationDistributeur { // un module IDL permet d'éviter les conflits de noms
   interface FacadeClient {
      void choix_boisson(in string code);
      // ...
   };
   interface FacadeFournisseur {...};
   interface FacadeDepanneur {...};
   interface PriseCourant {...};
   interface PriseEau {...};
   interface PlusDeMonnaieEvt : Components::EventBase {...};
   interface DistributeurVideEvt : Components::EventBase {...};
   interface TemperatureEvt : Components::EventBase {...};

   component Distributeur {
      provides FacadeClient fClient;       // Une facette
      provides FacadeFournisseur fFournisseur;
      provides FacadeDepanneur fDepanneur;

      uses multiple PriseCourant pCourant; // Un réceptacle
      uses PriseEau pEau;

      // Source d'événements acceptant un unique consommateur
      emits PlusDeMonnaieEvt ePdm;

      // Source d'événements acceptant de multiples consommateurs
      publishes DistributeurVideEvt pVide;

      consumes TemperatureEvt cTemp;       // Un puits d'événements

      attribute boolean on;   // Attention, attribute est un raccourci syntaxique
   };                         // en IDL pour définir des accesseurs get/set

   home MaisonDistributeur manages Distributeur {
      // ...
   };
};
```

LISTING 2.11 : Déclaration IDL d'un composant CCM

Implémentation Description Language). Ces implémentations partielles peuvent être générées dans de nombreux langages de programmation car les règles de projection entre les concepts OMG IDL et CIDL et les concepts de la plupart des langages de programmation ont été définies. Par exemple, en C++, un composant CCM peut être implanté par deux classes : l'une pour la fabrique de composants (Maison) et l'autre pour l'implémentation des facettes du composant.

```
// Déclaration de la classe d'implémentation de la maison du composant Distributeur
// Cette classe hérite de la classe MaisonDistributeurImpl générée par les outils CIF
class MaisonDistributeurImpl_ByProgrammer :
   virtual public MonApplicationDistributeurImpl::MaisonDistributeurImpl {
   // méthodes de création d'instances, ...
}

// Déclaration de la classe d'implémentation du composant Distributeur
// Cette classe hérite de la classe DistributeurImpl générée par les outils CIF
class DistributeurImpl_ByProgrammer :
   virtual public MonApplicationDistributeurImpl::DistributeurImpl {
   private:
      char * p_boisson_selectionnee;
      //...
   public:
      virtual void choix_boisson(const char * code) throw(CORBA::SystemException);
      // ...
}

// Implémentation de la méthode choix_boisson
void DistributeurImpl_ByProgrammer::choix_boisson(const char * code)
 throw(CORBA::SystemException) {
   p_boisson_selectionnee = code;
}
```

LISTING 2.13 : Déclaration et réalisation C++ d'un composant CCM

Un composant CCM est ensuite empaqueté sous la forme d'un fichier archive (.zip). Cette archive contient les fichiers de l'implémentation du composant et plusieurs fichiers XML :

- un descripteur CSD (*CORBA Software Descriptor*) décrivant le logiciel (auteurs, licence, compilateur utilisé, etc.),
- un descripteur CCD (*CORBA Component Descriptor*) décrivant les caractéristiques techniques du composant (modèle de transaction, de qualité de service, etc.),
- un descripteur CPF (*Component Property File*) indiquant des valeurs par défaut des attributs des composants,
- un descripteur CAD (*Component Assembly Descriptor*) décrivant les caractéristiques d'assemblage de plusieurs composants (contraintes de connexions, etc).

Le listing 2.14 montre un exemple de fichier CAD décrivant la connexion du puits nommé `cAlert` d'un composant LIGHTALARM avec la source `pVide` d'une instance du composant DISTRIBUTEUR.

```
<componentassembly id="IL300707">
   <description>
      Exemple d'assemblage de composants
   </description>
   <componentfiles>
      <componentfile id="file_distrib">distributeur.zip</componentfile>
      <componentfile id="file_la">lightalarm.jar</componentfile>
   </componentfiles>
   <partitioning>
      <homeplacement id="home_distrib">
         <componentfileref idref="file_distrib" />
         <componentinstanciation id="aDistrib" />
      </homeplacement>
      <homeplacement id="home_la">
         <componentfileref idref="file_la" />
         <componentinstanciation id="aLa" />
      </homeplacement>
   </partitioning>
   <connections>
      <connectevent>
         <consumesport>
            <consumesidentifier>cAlert</consumesidentifier>
            <componentinstanciationref idref="aLa" />
         </consumesport>
         <publishesport>
            <publishesidentifier>pVide</publishesidentifier>
            <componentinstanciationref idref="aDistrib" />
         </publishesport>
      </connectevent>
   </connections>
</componentassembly>
```

LISTING 2.14 : Exemple de fichier CAD

Critique

Les apports de la spécification CCM sont multiples :

- Elle est indépendante de tout langage de programmation (contrairement aux EJB) et de tout système d'exploitation (contrairement à (D)COM(+)).
- Elle supporte la notion d'interface requise via les réceptacles et les puits d'événements, ce qui permet de ne pas « coder en dur » dans l'implémentation d'un composant les interconnexions et les abonnements aux événements.
- Elle clarifie le concept de port qui est un point de connexion d'un composant associé à une interface.
- Elle offre un fort degré de standardisation qui permet aujourd'hui de disposer d'une multi-

tude d'implémentations comme OpenCCM[6] ou Cadena[7] sur lesquelles un composant CCM ou une application construite à base de composants CCM peuvent être déployés semi-automatiquement. Cela est notamment possible car l'utilisation des services non-fonctionnels se fait à travers des interfaces standardisées à différents niveaux : entre le composant et son conteneur, et entre le conteneur et le bus CORBA (contrairement à J2EE).

Bien que le modèle CCM constitue une avancée considérable par rapport aux modèles (D)COM(+) ou EJB, il souffre de plusieurs manques :

- un assemblage de composants n'est pas un composant, ils sont décrits par des fichiers CAD,
- il n'y pas de connecteurs qui permettent l'adaptation et l'abstraction des interactions entre les composants, bien que des extensions de CCM palliant cette limitation aient été proposés [Traverson et Yahiaoui, 2002] depuis,
- l'assemblage entre les composants est réalisé lors du démarrage de l'application, ce qui rend impossible les reconfigurations dynamiques d'architectures à base de composants CCM, toutefois de nombreux travaux ont proposé des extensions de CCM pour supporter les reconfigurations dynamiques comme [Almeida *et al.*, 2001],
- la programmation de composants CCM reste une tâche complexe et fastidieuse, heureusement facilitée par des outils graphiques et de génération de code.

2.2.6 Fractal

Le modèle

Fractal [Bruneton *et al.*, 2002] est un modèle abstrait de composants dont la spécification a été proposée par le consortium Object Web[8] en 2002. Cette spécification connaît actuellement deux implémentations de références : *Julia*[9] en Java et *Cecilia*[10] en C/C++, et d'autres implémentations expérimentales comme *FractTalk*[11] en Smalltalk ou encore *FractNet*[12] en .NET. Un langage de description d'interfaces (IDL) basé sur XML est aussi proposé en Fractal et il peut être traduit en Java ou en OMG IDL.

Un composant Fractal, représenté sur la figure 2.2.6 (issue de [Passama, 2006]), est constitué d'un *contenu* (partie implémentation) encapsulé par une *membrane* (partie contrôle).

Le contenu d'un composant contient l'implémentation de ses fonctionnalités métiers et peut être constitué d'un nombre fini d'autres composants appelés dans ce cas *sous-composants*. Le modèle Fractal est récursif (on utilise aussi le terme *hiérarchique*) et permet donc la construction de *composites* (aussi appelés *super-composants* du point de vue de leurs sous-composants) à partir de compo-

[6] http://openccm.objectweb.org
[7] http://cadena.projects.cis.ksu.edu
[8] http://www.objectweb.org
[9] http://fractal.objectweb.org/julia/index.html
[10] http://fractal.objectweb.org/c.html
[11] http://csl.ensm-douai.fr/FracTalk/
[12] http://www-adele.imag.fr/fractnet/

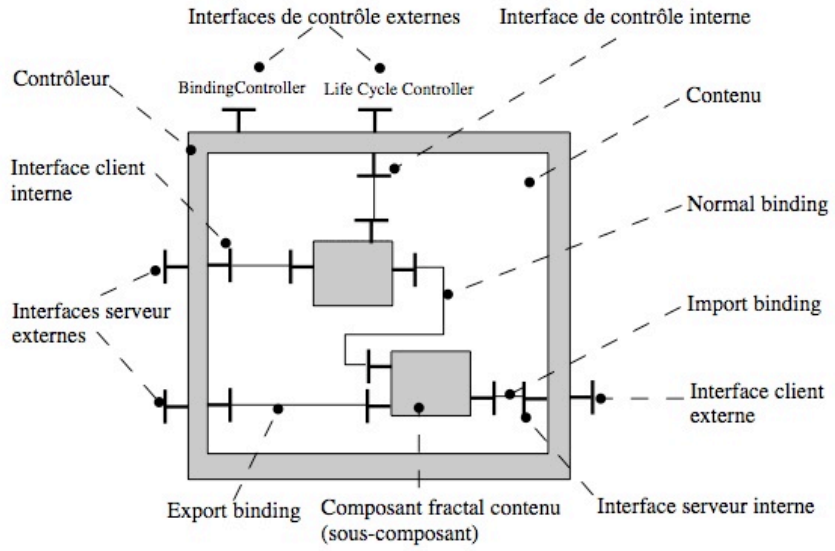

FIG. 2.15 : Structure d'un composant Fractal

sants. Fractal propose aussi le partage (*sharing*) qui permet à un composant d'appartenir à plusieurs composites.

La membrane d'un composant comprend les aspects non-fonctionnels relatifs à l'interaction, l'introspection et l'intercession du composant. Elle est constituée d'*interfaces* qui sont les points d'accès au composant. Les interfaces possèdent trois caractéristiques que nous désignons par les termes suivants :

- « visibilité », une *interface externe* est située sur la paroi extérieure de la membrane ; elle est donc accessible et utilisable par des clients externes au composant alors qu'une *interface interne* n'est utilisable que par son contenu ;
- « orientation », une *interface serveur* est une interface fournie à travers laquelle le composant offre un ensemble d'opérations et une *interface cliente* est une interface requise à travers laquelle le composant émet des invocations de services ;
- « rôle », une *interface de contrôle* standardise les opérations permettant de gérer les aspects non-fonctionnels, les liaisons ou le cycle de vie d'un composant ou de son contenu, alors qu'une *interface fonctionnelle* décrit les opérations métiers du composant.

Ces trois caractéristiques peuvent se combiner. Par exemple, une interface serveur fonctionnelle et externe déclare un ensemble d'opérations offertes par le composant. Une interface serveur externe de contrôle permet d'offrir des opérations permettant de gérer le composant, il existe d'ailleurs un ensemble standard d'interfaces de contrôle :

- `AttributeController` permet de modifier la valeur d'un attribut du composant,
- `ContentController` permet de consulter et modifier le contenu d'un composant (qui n'est donc pas une boîte noire) notamment en ajoutant ou retirant des sous-composants,
- `BindingController` permet de connecter et déconnecter les sous-composants d'un compo-

site,
– `LifeCycleController` permet d'arrêter ou démarrer un composant par exemple.

Un composant ne possède pas forcément toutes ces interfaces. C'est ainsi que la spécification classe les composants en différentes catégories : les *composite components* qui exposent leur contenu, les *primitive components* qui n'exposent pas leur contenu mais possèdent au moins une interface de contrôle et les *base components* qui ne possèdent aucune interface de contrôle. Les composants de base permettent d'arrêter la récursion du modèle Fractal et ils peuvent être considérés comme des objets possédant plusieurs interfaces fonctionnelles [Bruneton *et al.*, 2004].

L'assemblage de composants dans le modèle de composants Fractal repose sur la *liaison* (*binding*) d'interfaces. Il existe plusieurs trois sortes de *liaisons primitives* :

– la liaison normale (*normal binding*) lie une interface cliente externe à une interface serveur externe,
– la liaison d'export (*export binding*) lie une interface cliente interne d'un composite à une interface externe serveur d'un de ses sous-composants,
– la liaison d'import (*import binding*) lie une interface cliente externe d'un sous-composant à une interface interne serveur d'un de ses super-composants.

Les *liaisons composites* désignent un chemin de communication établi à partir d'un ensemble de liaisons primitives et composants de liaison (*binding components*) comme des talons (*stubs*), des squelettes (*skeletons*) ou des adapteurs (*adapters*). Ces composants de liaison sont aussi appelés *connecteurs* en Fractal [Bruneton *et al.*, 2002].

Fractal définit aussi un système de types pour les interfaces et les composants. Le type d'une interface de composant comprend : son nom, son *language type* (le type de l'élément définissant l'interface dans un langage de programmation donné), son rôle (cliente ou serveur) mais aussi sa contingence (obligatoire ou optionnelle) et sa cardinalité (singleton ou collection). Ces deux dernières caractéristiques sont particulièrement utiles pour la validation des connexions et plus généralement la vérification d'architectures. Le type d'un composant est défini comme l'ensemble des types de ses interfaces. La relation de sous-typage entre les types des interfaces et ceux des composants repose sur la *substituabilité*. On peut résumer les règles de sous-typage ainsi [Bruneton *et al.*, 2002] :

– Le type d'une interface I_1 est sous-type du type d'une interface serveur (resp. cliente) I_2 si :
  – le nom de I_1^i est le même que celui de I_2,
  – l'interface de langage correspondante à I_1 est une sous-interface (resp. super-interface) de celle correspondante à i_2,
  – si I_2 est obligatoire (resp. optionnelle) alors I_1 est obligatoire (resp. optionnelle) aussi,
  – si I_2 a une cardinalité collection alors I_1 a une cardinalité collection aussi.
– Le type d'un composant C_1 est sous-type d'un composant C_2 si et seulement si chaque type des interfaces clientes de C_1 est sous-type d'un type d'interface de C_2 et que chaque type des interfaces serveurs de C_1 est sous-type d'un type d'interface de C_2.

Ces règles définissent d'une part les liaisons valides en Fractal et d'autre part les substitutions de composants autorisées. Toutefois, la déclaration des types n'est pas obligatoire en Fractal et le programmeur peut décider d'utiliser le système de types ou non en fonction de ses besoins (rien n'est obligatoire en Fractal, la spécification définit d'ailleurs plusieurs niveaux de conformance en fonction

des éléments supportés) qu'il souhaite pour son application.

Un exemple

Fractal ne fait aucune supposition relative au langage de description d'interfaces (IDL) à utiliser. Julia, l'implémentation de référence en Java, intègre un ADL utilisant une syntaxe Java. Toutefois, un autre ADL a été spécifié afin d'être indépendant des langages et purement déclaratif en utilisant une syntaxe XML. Il existe aussi un outil permettant de créer graphiquement des descriptions d'architectures et de générer du code Julia. La figure 2.2.6 montre un exemple d'architecture issu de la spécification Fractal [Bruneton *et al.*, 2004]. Il s'agit d'un serveur web nommé Comanche constitué d'un ensemble de composants tels que le `Receiver` qui reçoit les requêtes HTTP ou encore le `Logger` qui journalise les requêtes reçues.

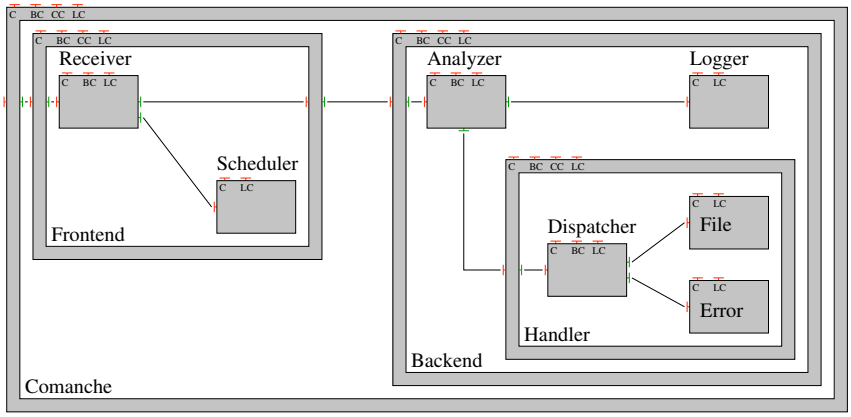

FIG. 2.16 : Architecture d'un composite serveur web en Fractal

Le listing 2.15 montre une partie de la description XML de l'architecture de cette application.

Critique

Fractal est sans conteste une contribution majeure dans le domaine des composants. Ce modèle abstrait est indépendant de toute technologie. Le concept de contrôleur est une explicitation du conteneur dans les modèles EJB et CCM. Le fait qu'un composite soit un composant permet de décrire et de manipuler simplement des architectures à différents niveaux d'agrégation. Cela constitue une réponse au problème du passage à l'échelle évoqué dans la section 2.1.2.

Néanmoins, le modèle Fractal n'intègre pas explicitement la notion de port. Elle est en réalité complètement intégrée dans la notion d'interface. Une interface est ainsi considérée en Fractal à la fois comme un point de connexion d'un composant et comme un contrat fonctionnel. Cela provoque bien souvent des ambiguïtés même si la spécification attire quelque peu l'attention sur la différence entre une *interface de composant* et une *interface de langage* (au sens Java). De plus, le support des connecteurs via les *binding components* semble assez primitif et peu étudié contrairement à d'autres approches comme SOFA.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
 "classpath://org/objectweb/fractal/adl/xml/standard.dtd">

<definition name="example.comanche.Logger">
  <interface name="l" signature="example.comanche.Logger" role="server"
   cardinality="singleton" contingency="mandatory"/>
</definition>

<definition name="example.comanche.BackendType">
  <interface name="rh" signature="example.comanche.RequestHandler" role="server"/>
</definition>

<definition name="example.comanche.Backend" extends="example.comanche.BackendType">
  <component name="ra" definition="example.comanche.RequestAnalyzer"/>
  <component name="rh" definition="example.comanche.Handler"/>
  <component name="l" definition="example.comanche.BasicLogger"/>
  <binding client="this.rh" server="ra.rh"/>    <!-- liaison d'export -->
  <binding client="ra.l" server="l.l"/>   <!-- liaison normale -->
  <binding client="ra.a" server="rh.rh"/>    <!-- liaison normale -->
</definition>

<definition name="...">
   ...
</definition>
```

LISTING 2.15 : Exemple de description d'architecture en Fractal ADL

2.2.7 SOFA/DCUP

Le modèle

SOFA (*SOFtware Applicances*) [Plásil *et al.*, 1998] est un projet, né en 1998, dont l'objectif est de fournir une plate-forme pour le développement d'applications par « composition » d'un ensemble de composants. DCUP (*Dynamic Component UPdating*) propose une architecture et une extension du modèle de composants SOFA afin de supporter le *versionning*, mais aussi le téléchargement et la mise à jour de façon « sûre » et dynamique (pendant l'exécution de l'application) de composants.

En SOFA, une application est vue comme un emboîtement de composants décrit par un descripteur de déploiement. Un composant est instance d'un *template* (ou *component type*). Un *template* est décrit par le *frame* et l'*architecture* de ses instances (cf. figure 2.2.7). Le *frame* est une vision boîte noire d'un composant définissant ses interfaces requises et fournies. De façon optionnelle, un *frame* peut déclarer des propriétés permettant de configurer le composant. Un *frame* peut être implémenté par plusieurs architectures. Une architecture est soit une implémentation opaque dans le cas de composants primitifs, soit une description structurelle comprenant les instanciations des sous-composants et la mise en place de *liaisons* dans le cas de composites.

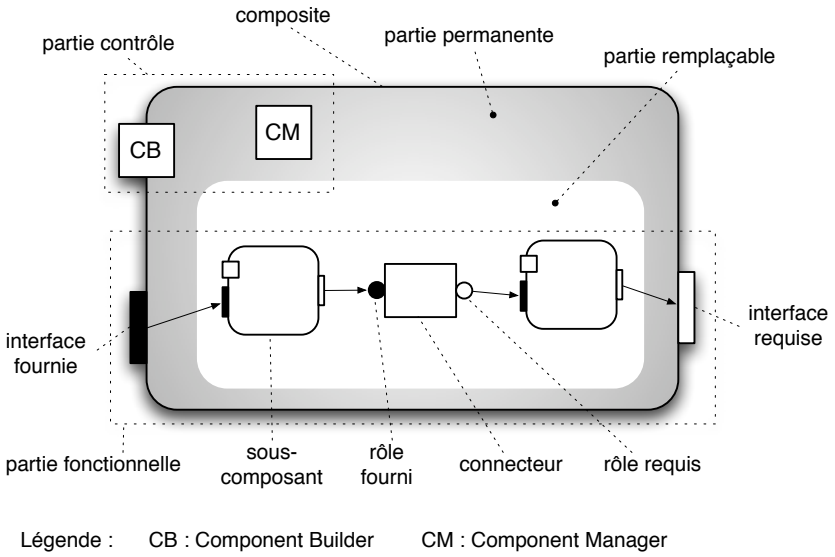

FIG. 2.17 : Structure d'un composant SOFA/DCUP

SOFA distingue quatre sortes de liaisons : (i) les liaisons entre une interface requise et une interface fournie, (ii) les liaisons dites de délégation entre une interface fournie d'un composite et une interface fournie d'un de ses sous-composants, (iii) les liaisons dites de subsomption (*subsuming*) entre une interface requise d'un sous-composant et une interface requise de son composite, (iv) les liaisons d'exemption (*exempting*) qui correspondent aux interfaces non-utilisées. Les trois premiers types de liaisons sont réalisées par des *connecteurs*. Un connecteur implémente la sémantique de l'interaction entre les composants et couvre les besoins d'adaptation entre les composants. Les connecteurs sont aussi considérés comme une solution au problème de l'anomalie de déploiement (*deployment anomaly problem* [Bálek et Plásil, 2001]). Par exemple, choisir un mode de communication distant (Remote Procedure Call par exemple) entre deux composants A et B lors du déploiement nécessiterait normalement de modifier leurs interfaces. L'utilisation d'un ou plusieurs composants adapteurs n'est pas une solution envisageable puisqu'elle implique de modifier l'architecture d'un éventuel composite C englobant A et B à cause d'une contrainte de déploiement. Afin de répondre à ce problème, les connecteurs sont des entités de première classe en SOFA. Chaque type de connecteur implémente la sémantique d'un type d'interaction. Un connecteur est décrit de façon similaire à un composant : par un *frame* de connecteur et une architecture de connecteur. Le frame d'un connecteur est représenté par un ensemble d'instances de rôles. Un rôle est une interface générique de connecteur prévue pour être liée à une interface de composant. L'architecture d'un connecteur décrit le contenu du connecteur. De même que pour un composant, il peut être simple (il contient seulement des éléments primitifs et est directement implémenté dans l'environnement sous-jacent) ou composé (il contient seulement des instances d'autres connecteurs ou composants). Les connecteurs les plus simples, exprimant les appels de procédures (`CSProcCall`), sont implicites et ne doivent pas être spécifiés dans l'architecture. SOFA fournit d'autres types de connecteurs prédéfinis comme `EventDelivery` et `DataStream`. L'utilisateur peut même définir ses propres connecteurs.

Le langage de description d'architecture CDL comporte aussi la possibilité d'attacher des descriptions comportementales sous la forme de *protocoles* [Plásil *et al.*, 1999; Plásil et Visnovsky, 2002] à différents niveaux de description d'un composant (*frame*, architecture et interface). Un protocole est un formalisme à base d'expressions régulières permettant de décrire les invocations d'opérations entrantes (vers le composant) et sortantes (vers un composant connecté) un peu comme des traces. La conformité des protocoles est basée sur l'inclusion des traces.

L'extension DCUP de SOFA propose une extension du modèle de composants SOFA afin de supporter la mise à jour des composants de façon sûre pendant leur exécution. Un composant DCUP (cf. figure 2.2.7) est divisé en deux parties : une permanente et une remplaçable. C'est la partie remplaçable d'un composant qui peut être changée dynamiquement et c'est aussi cette partie qui fait l'objet d'un contrôle de versions. DCUP introduit deux entités de contrôle : le *component manager* (CM) et le *component builder* (CB). CM est le cœur de la partie permanente et est responsable du contrôle du cycle de vie du composant pendant l'exécution. Chaque partie remplaçable (et versionnée) d'un composant est associée à un CB qui est responsable du contenu d'un composant, c'est-à-dire des sous-composants pour un composite ou des objets d'implémentation pour un composant primitif. Un CB gère la création, l'initialisation, le stockage et la restauration de l'état d'un composant avant et après une mise à jour.

SOFA définit le langage CDL (*Component Definition Language*). The langage, basé sur OMG IDL, permet de décrire les interfaces, les *frames* et les architectures des composants et des connecteurs. Les descriptions CDL sont compilées et stockées dans le TIR (*Type Information Repository*). Le TIR gère l'évolution des descriptions de composants et stocke les différentes versions de chaque élément défini en CDL. Dans le TIR, les éléments versionnés sont référencés par leur nom et une spécification de version. Dans les descriptions CDL, un développeur peut faire référence à une version spécifique présente dans le TIR.

Un exemple

L'exemple présenté dans cette section est adapté de [Plásil *et al.*, 1998]. Il s'agit de la réalisation d'un composant composite `Bank` (cf. listing 2.16). Le *frame* du composant BANK (cf. ligne 26 du listing 2.16) montre qu'il possède une interface multiple `clients` et une interface `supervisor`. Un client peut réaliser les actions définies dans l'interface `IClient` alors qu'un superviseur pourra réaliser celles définies dans l'interface `ISupervisor`. La ligne 9 du listing 2.16 montre un exemple très simple de protocole associé à l'interface `IClient`. Ce protocole indique l'ordre autorisé pour les opérations définies dans l'interface, à savoir d'abord `createAccount`, ensuite (la séquentialité est indiquée par `;`) les opérations `deposit`, `withdraw` et `balance` sont réalisables dans n'importe quel ordre (le choix est indiqué par `+`) et autant de fois que nécessaire (indiqué par $*$) pour finir par l'opération `deleteAccount` une seule fois. L'architecture `Bank` montre que ce composant est un composite et décrit les instanciations de ses sous-composants *s* et *ds* et leurs liaisons. Le mot clef **`local`** est ici utilisé pour lier les interfaces `client` à l'implémentation de l'architecture BANK. De même que pour les précédentes propositions, des outils permettent de générer le code des interfaces, des *component builders*, des squelettes d'implémentation (en Java, en C++, etc.) et des descripteurs de déploiement (en XML).

```
module BankDemo {
   interface IClient {
      string createAccount(in float init);
      string deleteAccount(in string account);
      boolean deposit(in string account, in float amount);
      boolean withdraw(in string account, in float amount);
      boolean balance(in string account);
      protocol:
         createAccount; (deposit + withdraw + balance)*; deleteAccount
   };
   interface ISupervisor {
       string assignAccountName();
       boolean canWithdraw(in string account);
       boolean canDelete(in string account);
   };
   interface IAccount {
      float getBalance();
      void setBalance(in float balance);
   };
   interface IDataStore {
      IAccount create(in string account);
      IAccount load(in string account);
      void delete(in string account);
   };

   frame Bank (property: num_of_clients) {
      provides:
         IClient client[num_of_clients];
         ISupervisor supervisor;
      requires:
         IDataStore storeAccess;
   };
};

architecture CUNI Bank version "v1" {
   inst ASupervisor version "v4" s;
   inst ADataStrore version "v2" ds;

   bind s.dataStoreAccess ds.dataStoreAccess;
   bind supervisor s.supervisorAccess;

   for n=1 to num_of_clients {
      bind teller[n] local;
   }
};
```

LISTING 2.16 : Exemple de description CDL d'un composant SOFA/DCUP

Critique

SOFA/DCUP propose un modèle de composants particulièrement intéressant où les composants et les connecteurs sont des entités de première classe. Les composants peuvent être mis à jour dynamiquement grâce à l'extension DCUP. Les connecteurs ont fait l'objet d'une attention particulière en SOFA [Bálek et Plásil, 2001; Balek, 2002]. Comme nous l'avons vu, les connecteurs permettent de masquer complètement la distribution des composants puisque le code de communication se trouve dans les connecteurs et non dans les composants, laissant leurs interfaces vierges de toutes les fonctionnalités liées à un mode de déploiement particulier. Bien qu'il existe des connecteurs prédéfinis en SOFA, il est aussi possible d'en définir de nouveaux spécifiques à des besoins particuliers. SOFA propose aussi la notion de protocole afin de décrire les comportements des composants. La notation utilisée pour les protocoles est suffisamment simple pour être réellement utilisée en pratique et bénéficier d'une technique de vérification des assemblages puissante et formalisée. SOFA propose aussi une modélisation (*SOFANode*) intéressante d'un environnement unique pour le développement, la distribution et l'exécution de composants. Cette modélisation est architecturée autour du TIR (*Type Information Repository*) qui est en quelque sorte une « étagère de composants». La problématique cruciale de gestion des versions des composants a d'ailleurs été intégrée en SOFA, ce qui permet de disposer de différentes versions (partie implémentation) d'un même *frame* dans le TIR.

Le modèle SOFA ne distingue toutefois pas les concepts de port et d'interface qui sont pourtant assez différents et complémentaires comme nous l'avons vu en CCM. Par ailleurs, les connecteurs et le composants n'ont pas tout à fait la même structure (pas de `ComponentBuilder` et `ComponentManager` pour les connecteurs) ce qui ne permet pas aux connecteurs d'être mis à jour dynamiquement comme les composants. La faiblesse majeure de SOFA, selon nous, est l'absence de support pour l'implémentation des composants. Certes, des squelettes de classes peuvent être générés mais il est évident que ce code généré peut être difficile à appréhender et le programmeur doit faire face à des problèmes techniques posés par le langage cible choisi. Une dernière remarque est que SOFA ne propose pas de support pour les services non-fonctionnels contrairement à EJB ou CCM.

2.2.8 ArchJava

Le modèle

ArchJava [Aldrich *et al.*, 2002b; Aldrich *et al.*, 2002a] a été créé en 2001 avec la volonté d'intégrer les concepts apportés par les ADLs comme Darwin [Magee et Kramer, 1996], dont il est inspiré, dans un langage de programmation à objets en l'occurrence Java. Les auteurs d'ArchJava ont constaté que les ADLs favorisent une meilleure conception des architectures logicielles, facilitant leur compréhension et autorisant des approches formelles pour leur analyse. Cependant, les ADLs utilisent souvent leur propre langage qui ne permet généralement pas de produire une application exécutable. Ce découplage dans les approches existantes entre la description d'une architecture et son implémentation ne permet pas d'assurer la non-violation des propriétés architecturales dans l'implémentation. C'est par exemple le cas pour l'intégrité des communications [Moriconi *et al.*, 1995; Luckham et Vera, 1995] qui spécifie que dans une implémentation, un composant ne doit commu-

niquer qu'avec ceux avec lesquels il est connecté dans l'architecture. L'objectif d'ArchJava est donc d'étendre le langage à objets Java afin d'unifier la description d'une architecture logicielle et son implémentation en garantissant que l'implémentation est conforme aux contraintes architecturales définies. ArchJava étend le langage Java en introduisant des concepts architecturaux et un système d'annotation de types nommé *AliasJava*.

Du point de vue architectural, ArchJava a introduit la notion de *classe de composant* (**`component class`**) qui décrit des *composants* assemblables afin de construire des architectures (*cf.* figure 2.2.8). Une classe de composant peut contenir des déclarations de *ports*, de méthodes et d'attributs (au sens Java), de composants encapsulés dits *sous-composants* (*sub-components*) et de *connexions*. Un composant est une instance d'une classe de composant qui communique avec d'autres composants via ses connexions. Les extrémités d'une connexion sont des ports qui sont les points de communication des composants. Un port déclare un ensemble de méthodes fournies (**`provides`**) et requises (**`requires`**). Une méthode fournie est implémentée dans la classe du composant et rendue accessible aux composants connectés à ce port. Inversement, une méthode requise via un port est fournie par un composant connecté via ce port.

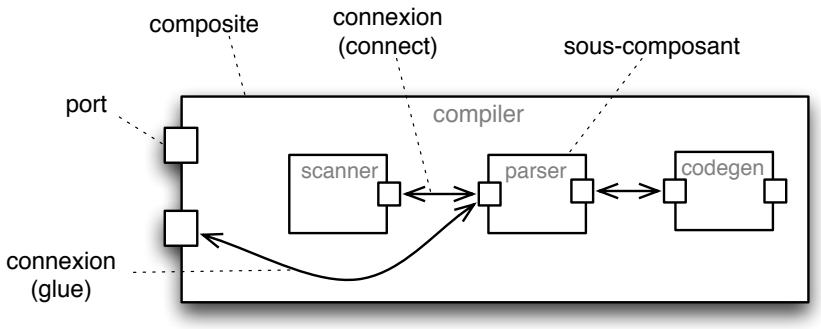

FIG. 2.18 : Architecture d'un composant ArchJava

Les annotations de types d'*AliasJava* ont pour objectif de contrôler statiquement (à la compilation) les échanges de références faits dans une implémentation pour garantir l'intégrité des communications. *AliasJava* a introduit les annotations suivantes : **`unique`** (une donnée référencée par une seule variable), **`owned`** (une donnée appartenant à un domaine défini par un unique objet et accessible par l'ensemble des objets de ce domaine uniquement), **`shared`** (une donnée partagée par les objets du programme comme un singleton [Gamma *et al.*, 1995] par exemple) et **`lent`** (souvent utilisé pour les paramètres ou les variables locales, une référence vers une donnée **`lent`** ne peut pas être stockée par ailleurs). Ces annotations peuvent être vues comme une généralisation des modificateurs Java pour les droits d'accès. Le listing 2.17 montre un exemple (adapté de [Aldrich *et al.*, 2002b]) de code ArchJava et des annotations.

La primitive `connect` permet de lier un ensemble de ports qui sont bidirectionnels (mélange de méthodes requises et fournies) en ArchJava. Par exemple, pour lier les ports $p_1, p_2, ..., p_n$ appartenant respectivement aux composants $c_1, c_2, ..., c_n$, il faut écrire :

```
connect c1.p1, c2.p2, ..., cn.pn;
```

```
public component class Scanner {
   public port out {
      provides Token  nextToken() throws ScanException;
   }
   // ...
}

public component class Parser {
   public port in {
      provides void setInfo(Token symbol, SymTabEntry e);
      requires Token nextToken() throws ScanException;
   }

   public port out {
      provides SymTabEntry getInfo(Token t);
      requires void compile(AST ast);
   }

   public port parseonly {
      provides void parse(String file);
   }

   void parse(String file) {
      Token tok = in.nextToken();  // appel de la méthode nextToken via le port in
      AST ast = parseFile(tok);
      out.compile(ast);
   }

   AST parseFile(Token lookahead) { ... }
   void setInfo(Token t, SymTabEntry e) {...}
   SymTabEntry getInfo(Token t) { ... }
   //...
}

public component class Compiler {
   private final owned Scanner scanner = new Scanner();
   private final owned Parser parser = new Parser();
   private final owned CodeGen codegen = new CodeGen();

   public port parseonly{}

   glue parseonly, parser.parseonly;   // intruction glue pour les délégations
   connect scanner.out, parser.in;  // intruction connect pour les liaisons
   connect parser.out, codegen.in;

   public static void main(shared String args shared [])
      { new Compiler().compile(args); }
   public void compile(shared String args shared [])
      { /* for each file in args do:  ...parser.parse();... */ }
}
```

LISTING 2.17 : Exemple de classes de composants en ArchJava

La sémantique de cette instruction est la suivante : une méthode requise intervenant dans la connexion est liée à une (s'il en existe plusieurs, il s'agit d'un cas d'erreur) méthode fournie de signature compatible (au sens du sous-typage).

Pour spécifier des composants ayant plusieurs connexions avec un même type d'entités — par exemple, le composant BANK (cf. section 2.2.7) qui est connecté à plusieurs composants CLIENT — ArchJava introduit la notion de *port interface*. Un *port interface* est instancié dynamiquement en un port à chaque fois qu'il est utilisé pour établir une connexion. Le port dynamiquement créé joue alors pleinement son rôle de point d'accès du composant. Comme ArchJava vérifie statiquement la validité d'une architecture, il est nécessaire de déclarer dans le code source des *patrons de connexion* (`connect pattern`) qui spécifient les connexions dynamiques.

ArchJava propose aussi un mécanisme d'héritage pour les classes de composants. Une classe de composant peut hériter d'une autre classe de composant ou de la classe Java standard. Lorsqu'une classe de composant hérite d'une classe Java standard, l'intégrité des communications n'est plus assurée par le compilateur ArchJava. Toutefois, ce mode dégradé permet la compatibilité avec les API Java existantes. Lorsqu'une classe de composant hérite d'une autre classe de composant, la sous-classe de composant hérite de tout ce que possède sa super-classe : ses méthodes, ses ports et ses connexions. Il est de plus possible :

- d'ajouter des attributs, des méthodes, des ports, des sous-composants et des connexions,
- de redéfinir des méthodes,
- d'ajouter des méthodes fournie à un port hérité.

Par contre, afin de préserver la substituabilité, il est interdit d'ajouter des méthodes requises à un port hérité. Dans tous les cas, les règles Java s'appliquent et il est impossible par exemple d'hériter la classe de composant COMPILER et d'essayer de redéfinir les sous-composants en les remplaçant par des versions plus spécifiques puisque Java ne supporte pas la covariance.

Un exemple

L'exemple développé dans cette section est adapté de [Aldrich, 2003]. La figure 2.2.8 montre l'architecture d'un composite WEBSERVER et le listing 2.18 montre le code de ce composite. Dans cet exemple, le sous-composant ROUTER accepte des requêtes HTTP et les transmet à un WORKER qui les traite. A chaque nouvelle requête entrante, le ROUTER demande un nouveau WORKER (cf. ligne 29) à travers son port `request` pour traiter la demande. Ce port `request` est connecté au port privé `create` du composite WEBSERVER (cf. ligne 3). Dans l'implémentation de la méthode `requestWorker` du port `create`, une nouvelle instance de WORKER est créé et connectée à travers son port `serve` (cf. ligne 11). Cette connexion dynamique est valide car conforme au pattern de connexion déclaré à la ligne 4.

```
public component class WebServer {
2    private final owned Router r = new Router();
     connect r.request, create;
4    connect pattern Router.workers, Worker.serve;

6    public void run() { r.listen(); }
```

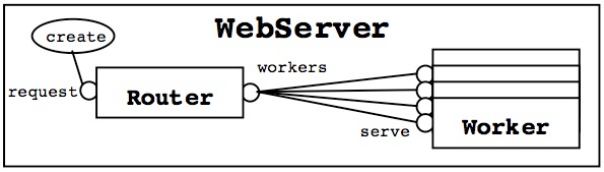

FIG. 2.19 : Architecture d'un composite WEBSERVER en ArchJava

```
8     private port create {
          provides r.workers requestWorker() {
10            final owned Worker newWorker = new Worker();
              r.workers connection = connect(r.workers, newWorker.serve);
12            return connection;
          }
14    }
  }
16
  public component class Router {
18    public port interface workers {
          group stream;
20        requires void httpRequest(stream InputStream in,stream OutputStream out);
      }
22    public port request {
          requires this.workers requestWorker();
24    }
      public void listen() {
26        unique ServerSocket<stream> server = new ServerSocket(80);
          while (true) {
28            unique Socket<stream> sock = server.accept();
              this.workers conn = request.requestWorker();
30            conn.httpRequest(sock.getInputStream(), sock.getOutputStream());
          }
32    }
  }
34
  public component class Worker extends Thread {
36    public port serve {
          group stream;
38        provides void httpRequest(stream InputStream in,stream OutputStream out) {
              this.in = in; this.out = out; start();
40        }
      }
42    public void run() {
          // gets requested file and sends it on the output stream
44    }
  }
```

LISTING 2.18 : Code source des composants WEBSERVER, ROUTER et WORKER en ArchJava

Critique

ArchJava est l'une des rares approches à tenter d'intégrer dans l'implémentation les descriptions architecturales normalement décrites par les ADLs. Cela a pour objectif de faciliter la compréhension des programmes, de garantir l'intégrité des communications entre les composants grâce au système de type *AliasJava* et d'assurer une bijection entre le code et l'architecture durant les phases de maintenance ou d'évolution de l'application. Un autre atout d'ArchJava est certainement le fait qu'il est une extension de Java dont la syntaxe est largement connue ce qui permet un apprentissage rapide [Bennouar *et al.*, 2006] et contribue à son essor comparé à d'autres propositions bien plus lourdes à mettre en œuvre.

ArchJava souffre de quelques faiblesses du point de vue de la dynamicité. Tout d'abord, l'intégrité des communications est vérifiée statiquement, ce qui impose de déclarer à l'avance des patrons de connexions réduisant ainsi les possibilités à l'exécution. Par ailleurs, il n'est pas possible d'enlever des connexions. ArchJava n'offre aucune possibilité pour décrire les aspects comportementaux d'une architecture comme c'est le cas en SOFA avec les protocoles. Le contrôle de l'intégrité des communications par le typage impose des contraintes de programmation avec l'utilisation d'annotations de types qui complexifient l'écriture et la lecture du code (comme le prouve l'exemple développé ci-dessus). Nous pensons que la principale faiblesse d'ArchJava est justement d'être une extension de Java. En effet, cela suppose que tous les concepts et mécanismes objets (de Java) sont compatibles avec l'objectif visé. En ArchJava, les objets et les composants supportent des contraintes différentes. Par exemple, une référence sur un objet peut être échangée entre composants ou objets lors de l'exécution alors qu'une référence sur un composant ne peut pas, justement pour éviter de violer l'intégrité des communications. Cela se traduit par le fait que tous les paramètres des méthodes sont obligatoirement définis par une classe Java standard comme c'était le cas pour `InputStream` ou `Token` dans les exemples présentés. Dans ce contexte, comment décider ce qui doit être un objet et ce qui doit être un composant lors de la conception d'une application ? Et cette décision ne sera-t-elle pas décisive pour les futures évolutions de l'application ?

2.2.9 UML 2.0

Le modèle

UML (*Unified Modeling Language*) [OMG, 2004] est un langage graphique normalisé, défini par l'OMG, permettant la modélisation de systèmes. Dans sa version 1.x, ce langage intégrait déjà la notion de composants, considérés comme des unités modulaires, déployables et interchangeables d'un système, encapsulant l'implémentation et n'exposant qu'un ensemble d'interfaces. Cette vision assez bas niveau des composants pour un langage de modélisation a changé à partir d'UML 2.0 [Cheesman et Daniels, 2000]. Les composants sont maintenant des unités de structuration abstraites qui représentent des sous-parties d'un système, pouvant être modélisées selon différents points de vue et raffinées tout au long du cycle de développement.

UML 2.0 a introduit des concepts et mécanismes, inspirés des ADLs, permettant de décrire les systèmes en terme de composants interconnectés. Un *composant* (*Component*) est une extension de la notion de *classe structurée* (*StructuredClass*), elle-même étant une extension indirecte (via *Struc-*

turedClassifier notamment) de la notion de *Classifier*. Un composant possède une *structure externe* constituée de *ports* (*cf.* figure 2.2.9). Les ports isolent un composant de son environnement en précisant les interactions entre sa *structure interne* et son environnement.

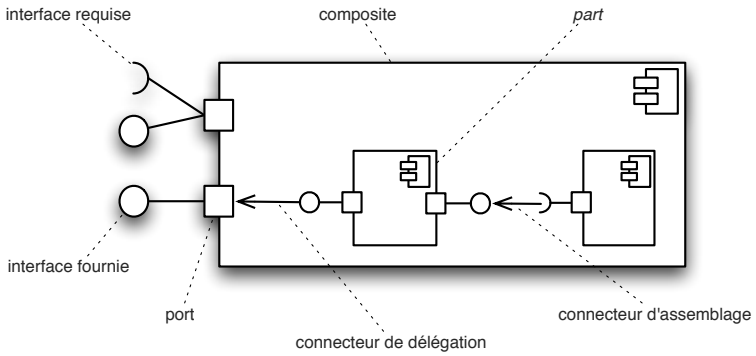

FIG. 2.20 : Structure d'un composant UML

Plusieurs ports peuvent être définis pour un composant, ce qui permet de distinguer des interactions différentes selon le port à travers lequel elles sont réalisées. Les ports renforcent le découplage entre un composant et son environnement afin qu'il puisse être réutilisé dans un autre environnement conforme aux contraintes imposées sur le port. Les contraintes associées à un port peuvent être des *interfaces fournies*, des *interfaces requises* ou des *protocoles*. Un port peut être associé à plusieurs interfaces fournies et plusieurs interfaces requises définissant ainsi l'offre et la demande correspondant à une collaboration à travers ce point de communication du composant. Les interfaces décrivent des contraintes statiques (nature et signature des opérations) qui doivent faire l'objet d'une implémentation par un *classifier* (classe ou composant). Les protocoles sont des descriptions de contraintes dynamiques qui peuvent s'effectuer à l'aide de diagramme d'états dont nous donnons un exemple dans la suite.

Les collaborations entre deux ou plusieurs composants se traduisent par des *connecteurs* représentant les possibilités de communication entre plusieurs instances de composants. Il existe deux sortes de connecteurs en UML : les *connecteurs d'assemblage* (*assembly connector*) et les *connecteurs de délégation* (*delegation connector*). Un connecteur d'assemblage permet de relier des composants qui fournissent et requièrent des services compatibles. Un connecteur d'assemblage est défini entre une interface requise (resp. un port requis) et une interface fournie (resp. port fourni). Les connecteurs de délégation sont utilisés pour la construction de composites, c'est-à-dire de composants ayant une structure interne constituée de *parts* (des *Property* qui dérivent de *TypedElement*) et de connecteurs. Les connecteurs de délégation permettent de relier les contrats externes d'un composant (interfaces ou ports) aux contrats externes de ces *parts* afin que de rediriger les requêtes d'opérations entrantes. De même que pour les ports, un connecteur peut être décrit par un protocole et il est ensuite possible de vérifier la validité d'une connexion en vérifiant la conformité des protocoles.

Un exemple

La figure 2.2.9 présente l'exemple d'un composite STORE tiré de la spécification de UML [OMG, 2004]. Ce composite est constitué de trois sous-composants (*parts*), respectivement dédiés à la gestion des ordres de commandes (ORDER), des produits (PRODUCT) et des clients (CUSTOMER). Ce composite contient aussi quatre connexions : deux connexions de délégation et deux connexions d'assemblage. On peut remarquer que dans cet exemple, les interfaces requises et fournies sont directement attachées à un composant et non à un port. Cela provient du fait que la notion de composant est définie en UML comme une extension de celle de classe. Cela est assez déroutant sachant que les ports des composants devraient être leurs seuls points d'interactions.

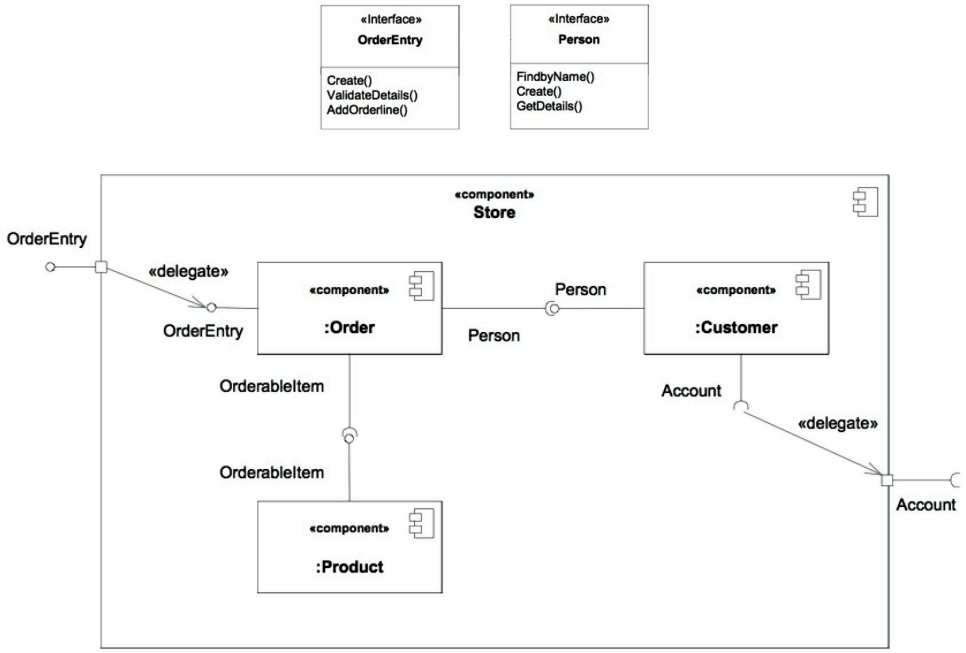

FIG. 2.21 : Un composite STORE en UML

La figure 2.2.9 montre un exemple de protocole attaché à l'interface `OrderEntry`. Ce protocole permettra la validation des connexions établies via cette interface.

Critique

A partir de UML 2.0, un pas vers la description d'architecture a été franchi par rapport aux versions précédentes. Les concepts de *port*, de *connecteur* et de *diagramme d'architecture* ont été introduits. UML permet aussi bien les descriptions structurelles des architectures que les descriptions comportementales via les protocoles notamment. La plupart des ADLs ont les mêmes objectifs que UML à savoir la modélisation des systèmes. Depuis sa version 2.0, UML est bien mieux adapté pour décrire

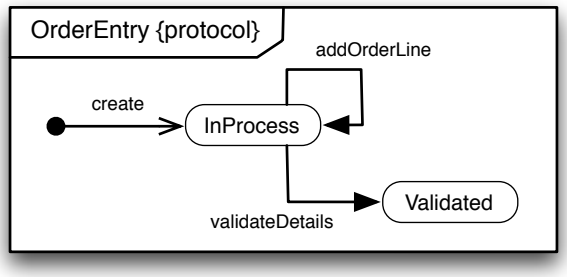

FIG. 2.22 : Exemple de protocole associé à une interface en UML

des architectures logicielles qu'il ne l'était dans ses versions précédentes comparé aux ADLs [Medvidovic *et al.*, 2002]. Actuellement, non seulement il supporte les principales constructions des ADLs mais il offre aussi un mécanisme d'extension (sous la forme de profil) puissant qui permet de l'adapter lorsqu'il n'offre pas en standard un concept ou un mécanisme dont on pourrait avoir besoin.

UML souffre de plusieurs défauts dont le principal est sa complexité. Il est en effet difficile de maîtriser complètement les mécanismes UML (surtout les plus récents) qui manquent souvent de précisions dans la spécification. C'est ainsi que nous avons montré que des interfaces peuvent être attachées directement à un composant alors qu'elles devraient toujours être attachées à un port d'un composant. UML souffre comme la plupart des ADLs d'un découplage fort avec l'implémentation.

2.3 Comparaisons des approches à composants

Dans cette section, nous comparons les différentes approches à composants et notamment celles étudiées en détails dans la section précédente.

2.3.1 Les objectifs visés

Comme on a pu le constater lors des différentes présentations détaillées de la section précédente, chaque approche est caractérisée par un ou plusieurs objectifs principaux. Le tableau 2.1 présente, de manière synthétique, les objectifs spécifiques des différentes approches.

2.3.2 Niveaux d'abstraction

La plupart des approches à composants ne couvrent pas l'ensemble du cycle de développement d'une application, depuis sa spécification, considérée comme le plus haut niveau d'abstraction, jusqu'à son code binaire, représentant le plus bas niveau. De plus, comme nous l'avons souligné dans la section précédente, certaines approches se spécialisent pour un domaine particulier. La figure 2.23 présente un repère dont les axes sont justement ces deux critères : niveaux d'abstraction et domaine d'application.

	Focalisation
(D)COM(+)	Réutilisation dans l'environnement MS-Windows
Javabeans	Assemblage graphique (*à la souris*) de *Javabeans*
EJB	Développement d'applications Web selon une architecture multi-niveaux
CCM	Développement d'applications réparties et hétérogènes à l'aide de composants
Fractal	Modèle hiérarchique et récursif de composants supportant le partage de sous-composants
ArchJava	Extension de Java garantissant l'intégrité des communications entre les composants dans l'implémentation
SOFA	Mise à jour dynamique des composants et connexion de composants via des connecteurs avec vérification de la compatibilité comportementale basée sur la notion de protocole
UML	Conception d'applications à l'aide de composants (extension des *classifiers*) éventuellement composites et de connexions
Wright	Description et analyse formelles du comportement des composants, des connecteurs et des configurations (descriptions d'assemblages de composants via des connecteurs)

TAB. 2.1 : Les objectifs spécifiques de quelques approches à composants

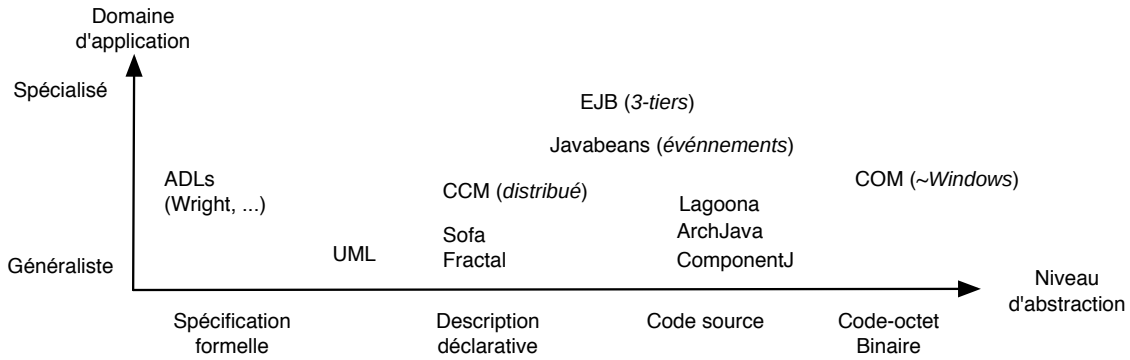

FIG. 2.23 : Niveau d'abstraction et domaine d'application : deux axes de classification des approches à composants

Les ADLs sont des approches qui peuvent être aussi bien généralistes que spécialisées, par contre elles s'intéressent principalement à la spécification des architectures. Depuis sa version 2.0, UML s'inscrit dans la même veine que les ADLs mais il ne propose pas de support formel contrairement à de nombreux ADLs. Les modèles de composants Fractal et SOFA sont assez généralistes et indépendants de l'implémentation. CCM est plus orienté pour le développement d'applications distribuées. Les *Javabeans* et les EJB sont des approches plus spécialisées car elles sont très liées à l'environnement Java. De plus, les EJB se focalisent essentiellement sur les applications web construites en couches. Les approches ArchJava, ComponentJ ou encore Lagoona sont des langages de programmation offrant au programmeur des facilités (primitives, structures de contrôle, etc.) pour mettre en pratique la programmation par composants. Finalement, COM est la proposition qui ne s'intéresse qu'au format binaire des composants bien que cela ait changé depuis l'avènement de .NET.

2.3.3 Comparaison générale

Le tableau 2.2 présente une synthèse des différentes approches étudiées jusqu'ici. Les critères de comparaison classés par thème sont les suivants :

- **La structure** :
  - *Itf. fournies*. Un composant offre-t-il une ou plusieurs interfaces qui définissent différents points de vue ?
  - *Itf. requises*. Un composant exprime-t-il ses dépendances externes notamment sous la forme d'une ou plusieurs interfaces requises ?
  - *Ports*. Un composant possède-t-il des ports qui constituent ses points d'interaction et de connexion ?
  - *Composites*. Le modèle propose-t-il la notion de composite c'est-à-dire de composant qui encapsule d'autres composants interconnectés ?
  - *Connecteurs*. Le modèle réifie-t-il la notion de connexion ?
- **Le code source** :
  - *Desc. d'itf.* En quel langage sont écrites les descriptions d'interfaces ?
  - *Desc. de comp*. En quel langage sont écrites les descriptions de composants ?
  - *Implémentation*. En quel langage de programmation (LP) sont implémentés les composants ?
- **Le déploiement** :
  - *Unité déployée*. Sous quelle forme sont diffusés les composants ?
  - *Automatisation*. Le modèle propose-t-il un support automatisé ou semi-automatisé du déploiement ?
  - *Description*. En quel langage est décrit ou configuré le déploiement ?
  - *Serv. non-fonct.*. Est-il possible de configurer des services non-fonctionnels pour un composant ?
- **L'exécution** :
  - *Environnement*. Quel est l'environnement d'exécution des composants ?
  - *Connexion*. Est-il possible de connecter dynamiquement des composants ?
  - *Déconnexion*. Est-il possible de déconnecter dynamiquement des composants ?
  - *Remplacement*. Est-il possible de remplacer dynamiquement un composant ?

		(D)COM(+)	*Javabeans*	EJB	CCM	Fractal	SOFA	ArchJava	UML
Structure	Itf. fournies	n	✗	4[a]	n	n	n	n	n
	Itf. requises	✗	✗	✗	n	n	n	n	n
	Ports	✗	✗	✗	Unidir.[b]	✗	✗	Bidir.	Bidir.
	Composites	✓	✗	✗	✗	✓	✓	✓	✓
	Connecteurs	✗	✗	✗	✗	✗[c]	✓	✓	✓
Code Source	Desc. d'itf.	MSIDL	Java	Java	OMG IDL	IDL[d]	IDL	ArchJava	UML
	Desc. de comp.	LP[e]	Java	Java	CIDL	ADL[f]	CDL	ArchJava	UML
	Implémentation	Squel.+LP	Java	Java	LP[g]	Squel.+LP	Squel.+LP	ArchJava	/
Déploiement	Unité déployée	Dll, Exe	Jar	Jar, Ear, War	Zip	✗	inconnue[h]	✗	/
	Automatisation	✗	✗	✓	✓	✗	✓	✗	/
	Description	✗	✗	XML	XML	✗	XML	✗	Diag.
	Serv. non-fonct.	✗	✗	✓	✓	✗	SOFANode	✗	/
Exécution	Environnement	Windows	JVM	J2EE	CORBA	DI[i]	SOFAnode	JVM	/
	Connexion	✓	✓	✗	✗	✓	✓	✓[j]	/
	Déconnexion	client	client	✗	✗	✓	✓	✗	/
	Remplacement						✓		/

TAB. 2.2 : Synthèse générale des approches à composants

Légende : ✓ : supporté ✗ : non supporté / : non pertinent <vide> : non spécifié mais possible n : arité multiple Diag. : Diagrammes

[a] Interfaces de fabrique locale et distante et interfaces métiers locale et distante.
[b] Les ports peuvent être unidirectionnels ou bidirectionnels suivant s'ils acceptent ou non les communications dans les deux sens.
[c] Les *binding components* n'offrent qu'un support limité des connecteurs.
[d] Fractal n'impose pas d'IDL particulier.
[e] N'importe quel Langage de Programmation adapté (à objets ou non)
[f] Fractal n'impose pas d'ADL particulier, bien que fractal-ADL (langage déclaratif XML) soit proposé.
[g] Les descriptions des composants écrites dans un langage abstrait souvent déclaratif sont utilisées pour générer des squelettes de code dans un LP.
[h] L'environnement SOFAnode détermine la forme de diffusion des composants.
[i] L'environnement d'exécution dépend de l'Implémentation choisie, par exemple ce sera la JVM pour l'implémentation Julia, etc.
[j] Toute connexion dynamique doit être conforme à un *pattern de connexion* déclaré statiquement.

On peut remarquer que les modèles industriels se focalisent en premier lieu sur des problématiques pragmatiques telles que le déploiement, le packaging ou encore les services non-fonctionnels. Ils manquent pour la plupart d'un réel modèle de composants comme c'est le cas pour COM, *Javabeans* ou EJB. Avec le modèle de composants CCM, l'OMG fait le pont entre les problématiques pragmatiques et les concepts apportés par les ADLs. Fractal est un modèle de composants réflexif qui met en avant les notions de composites (représentation des configurations dans les ADLs par des composants) et de partage. SOFA se concentre sur la dynamicité avec le remplacement dynamique de composants ou encore sur les connecteurs qui sont la réification des connexions entre les composants. ArchJava est une extension du langage Java afin d'exprimer l'architecture des applications directement dans le code source et ainsi assurer l'adéquation entre les descriptions architecturales et l'implémentation.

2.3.4 Niveaux de découplage

Dans la section 2.1.1, nous avons présenté la notion de découplage. Nous nous concentrons dans cette section sur les approches permettant l'implémentation effective des composants. Nous avons aussi intégré Java dans cette comparaison afin de comprendre où se situent les langages à objets. Le tableau 2.3 présente une comparaison des différentes approches en regard de cette propriété de découplage. Pour cela, nous distinguons les critères suivants :

Code côté fournisseur Un composant fournissant un service possède-t-il du code spécifique (par exemple la gestion des écouteurs en *Javabeans*), écrit par son programmeur et relatif aux connexions ou aux communications afin que ce service puisse être utilisé ?

Code côté client Un composant client utilisant un service externe possède-t-il une référence explicite sur le composant fournisseur ? Le client possède-t-il du code spécifique relatif aux connexions ou aux communications afin de pouvoir utiliser un service externe ? Par exemple, les modèles EJB et (D)COM(+) imposent d'intégrer du code dans le composant client afin de rechercher une référence sur l'interface maison du composant fournisseur dans un annuaire.

Changer le fournisseur Est-il possible de changer dynamiquement le fournisseur d'un service pour un composant client ?

Découplage C'est la synthèse du tableau qui propose 3 niveaux de découplages (faible, moyen et élevé) que nous établissons en fonction des deux derniers critères précédents. En effet, le premier critère (code dans le fournisseur) n'est pas une limitation en terme de couplage (seulement une contrainte pour le programmeur) puisque le fournisseur ne possède jamais de références « figées » sur ses clients.

(D)COM(+) et EJB sont les approches les moins satisfaisantes du point de vue du découplage. CCM et ArchJava sont moyennes puisqu'elle ne permettent pas le changement dynamique de fournisseur dans leur versions de base. Les approches les plus pertinentes en terme de découplage sont : *Javabeans*, Fractal et SOFA. *Javabeans* est une approche particulière dont la communication repose entièrement sur le schéma de conception Observateur. Cela permet en effet un bon niveau de découplage mais impose des contraintes fortes sur la programmation des composants. Fractal et SOFA

	Code côté fournisseur	Code côté client	Changer le fournisseur	**Découplage**
Java	o	•[a]	∼[b]	faible
(D)COM(+)	o	•	∼[c]	faible
Javabeans	•[d]	o [e]	✓	élevé
EJB	o	•[f]	∼	faible
CCM	o	o	✗	moyen
Fractal	o	o	✓	élevé
SOFA	o	o	✓	élevé
ArchJava	o	o	✗	moyen

TAB. 2.3 : Niveau de découplage des composants dans les différentes approches

Légende : o : absence de code spécifique • : présence de code spécifique
✓ : supporté ✗ : non supporté ∼ : partiellement supporté

sont donc les propositions les plus abouties dans ce domaine et l'approche SOFA se démarque encore grâce au support des connecteurs.

2.3.5 Assemblage

L'assemblage de composants est un mécanisme central qui repose dans la plupart des modèles sur différents types de liaisons entre les ports (ou les interfaces pour les modèles sans ports). Le tableau 2.4 propose une synthèse en ce qui concerne l'assemblage entre les composants. Ce tableau ne prend pas en compte les propositions (D)COM(+), EJB et *Javabeans* car ces approches ne présentent pas les mécanismes comparés. Nous utilisons les critères de comparaison suivants :

- **Type de liaisons**. Quels sont les différents types de liaisons supportés ?
- On distingue généralement deux niveaux de conformités :
  - **Syntaxique**. Comment est vérifiée la validité syntaxique des liaisons ?
  - **Comportementale**. Comment est vérifiée la validité comportementale des liaisons ? Ce niveau de vérification relève de la sémantique des liaisons et vise à garantir la robustesse dynamique des communications effectuées via une liaison.
- **Types de donnés échangés**. Que peuvent s'échanger les composants comme données ?

[a] Un objet client doit intégrer le code lui permettant de posséder une référence sur un objet fournisseur (instanciation, accesseurs)

[b] Si l'on a utilisé des accesseurs dans le client (*cf.* figure 2.2), le changement dynamique est possible

[c] Possible si le client a été doté d'*interfaces outgoing* ou peut-être en modifiant directement une entrée dans la base de registre Windows

[d] Le code de gestion des Observateurs (abonnement, désabonnement, signalement)

[e] La génération automatique d'un adapteur permet de s'affranchir de code spécifique du côté client (souscripteur)

[f] Interrogation de l'annuaire JNDI, etc.

[g] Peut-être en modifiant directement une entrée de l'annuaire JNDI

	CCM	**Fractal**	**SOFA**	**ArchJava**	**UML**
Types de liaisons	Facette—Réceptacle Puits—Source	IR—IF IF—IF sc. IR sc.—IR	IR—IF IF—IF sc. IR sc.—IR	Port—Port Port—Port sc.	Port—Port Port—Port sc.
Conform. syntax.	sous-typage i.	sous-typage i.	sous-typage i.	sous-typage i.	sous-typage i.
Conform. comportem.	✗	✗	protocoles	✗	protocoles
Types de Données	IDL[a]	IDL	IDL	Objet[b]	*Instances*[c]

TAB. 2.4 : Comparaison du mécanisme d'assemblage de composants de différentes approches

Légende : IR : Interface Requise (cliente en Fractal) IF : Interface fournie (serveur en Fractal) sc. : sous-composant —: liaison ✗: non supporté

[a] N'importe quelles données dont le type est décrit en IDL.
[b] Uniquement les instances de classes (et pas celle des classes de composants) peuvent être passées en argument en ArchJava.
[c] Le méta-modèle UML ne décrit pas les entités présentes à l'exécution.

Dans les approches supportant les composites, on distingue généralement les liaisons de délégation entre un composite et l'un de ses sous-composants et les liaisons « normales » entre deux composants. CCM distingue aussi les liaisons de types requis/fourni (communication synchrone) et les liaisons événementielles (communication asynchrone). ArchJava proposait aussi dans ses premières versions la notion de service diffusé (`broadcast`) qui s'apparentait à des liaisons événementielles (bien qu'il s'agissait de communications synchrones aussi).

La vérification des connexions entre les composants repose dans la majorité des modèles sur la relation de sous-typage entre les interfaces pour la conformité syntaxique et sur les protocoles pour la conformité comportementale. Les protocoles peuvent être décrits à l'aide de formalismes variés comme on a pu le constaté en SOFA et en UML. ArchJava propose aussi un système de typage plus complet pour contrôler l'intégrité des communications entre les composants. Cela passe par un contrôle des données échangées entre les composants afin d'assurer l'intégrité des communications. Les approches à composants sont souvent sous-spécifiées en ce qui concerne les types de données échangeables et cela se traduit par des choix propres à chaque implémentation. Par exemple, l'implémentation Julia de Fractal permet de passer en paramètre n'importe quel objet Java ce qui peut se traduire par la violation de l'intégrité des communications lors de l'exécution [Léger *et al.*, 2006].

2.3.6 Synthèse

Après avoir étudié toutes ces propositions, nous pensons que le critère du découplage est celui qui est fondamental dans l'approche composants. En effet, le découplage favorise d'une part la réutilisation des composants dans divers contextes et d'autre part la dynamicité puisque ces liens de couplage explicites peuvent être modifiés à tout moment. En regard de ce critère, nous proposons le classement suivant :

- approches à objets (distribuées) : EJB, (D)COM(+)
- approches à composants : *Javabeans*, CCM, Fractal, SOFA, ArchJava, UML 2.0

Les EJB et les composants COM ne constituent pas, selon nous, des approches à composants du fait que le code de l'interaction entre un composant client et un composant fournisseur est encore noyé dans le code métier. Nous pouvons résumer cela par la phrase suivante qui résume notre vision de la différence principale entre l'approche à objets et l'approche à composants :

> En POO, lors d'un envoi de message, un objet appelant détient une référence sur un objet receveur. Suivant la manière dont cette référence est acquise (statiquement ou dynamiquement) et suivant si elle peut être changée dynamiquement, elle peut constituer un *lien de couplage fort* qui réduit la potentialité de réutilisation de l'objet appelant.
> En POC, un composant client ne détient pas de référence sur un composant fournisseur mais ils sont mis en relation par une connexion modifiable (*lien de couplage faible*), rendue possible grâce à l'explicitation des dépendances requises.

2.4 Conclusion

Cette thèse s'intéresse en premier lieu à la programmation par assemblage de composants. En regard des comparaisons précédentes et plus particulièrement la manière dont sont implémentés les composants (*cf.* tableau 2.2), nous distinguons quatre principales familles d'approches relativement aux techniques d'implémentation :

1. Respect de conventions : *Javabeans*, (D)COM(+)
2. Extension de framework : EJB, picolo, Julia
3. IDL+Génération de squelettes+Implémentation : CCM, SOFA, Fractal
4. Langage à composants : ArchJava, ComponentJ, Lagoona, keris, piccola, boxscript

La première famille consiste à définir des normes ou standards de programmation qui permettront la réutilisation et l'assemblage des composants. Par exemple, *Javabeans* impose de respecter des conventions syntaxiques telles que l'utilisation d'accesseurs pour définir des propriétés ou encore l'utilisation du schéma de conception Observateur pour permettre l'assemblage des *Javabeans*. Le respect de ces conventions incombe au programmeur qui ne peut s'en affranchir sous peine de restreindre les réutilisations potentielles de son composant et même rendre impossible son assemblage. (D)COM(+) repose sur un standard binaire ce qui permet de rendre les composants indépendants de tout langage de programmation. Toutefois, un programmeur doit choisir un compilateur générant du binaire conforme à la norme et doit même dans certains cas écrire son code source de telle sorte que le compilateur génère du binaire conforme. Cela impose une bonne connaissance du fonctionnement du compilateur. L'arrivée de .NET a quelque peu pallié cet inconvénient puisque n'importe quel langage conforme à la CLS (*Common Language Specification*) possède un compilateur vers du code-octet (*bytecode*) conforme.

La seconde famille impose de programmer les composants en étendant un *framework* comme c'est le cas pour les EJB. Cela rend l'implémentation des composants dépendante du *framework* en question et du langage dans lequel ont été développés les composants. Toutefois, l'implémentation des composants est facilitée par rapport à la première famille puisque le programmeur est déchargé d'une grande quantité de code à écrire.

La troisième famille concerne particulièrement les approches qui se veulent indépendantes de tout langage de programmation. Les interfaces et les composants sont décrits à l'aide d'un langage déclaratif standardisé tel que OMG IDL ou SOFA CDL. Des squelettes d'implémentation des composants peuvent ensuite être générés grâce à des outils de génération de code utilisant ces descriptions. Le programmeur doit ensuite compléter ces squelettes.

La dernière famille comprend des langages de programmation qui proposent des abstractions et/ou des mécanismes afin de programmer des composants mais aussi des applications par assemblage de composants. Par analogie avec les langages à objets, nous appelons ces approches des *langages à composants*.

Seuls les langages à composants proposent de supporter le paradigme composant au niveau de l'implémentation. Bien que la seconde approche tente de s'abstraire des langages d'implémentation, il nous semble que ceux-ci sont importants puisqu'ils déterminent en grande partie la facilité

de maintenance, de compréhension et de rétro-ingénierie du code source (*cf.* figure 2.4). Utiliser des langages de programmation qui n'offrent pas un bon niveau de découplage, ni un réel support de la POC, est tout à fait possible mais cela se traduit inévitablement par la complexification de toutes les activités se basant sur le code source. Cela conduit à un éloignement entre la conception et l'implémentation puisque les abstractions utilisées sont différentes. Cela impose bien souvent l'utilisation de conventions (schémas de conception ou encore règles syntaxiques), partiellement prises en charge par des outils de génération de code.

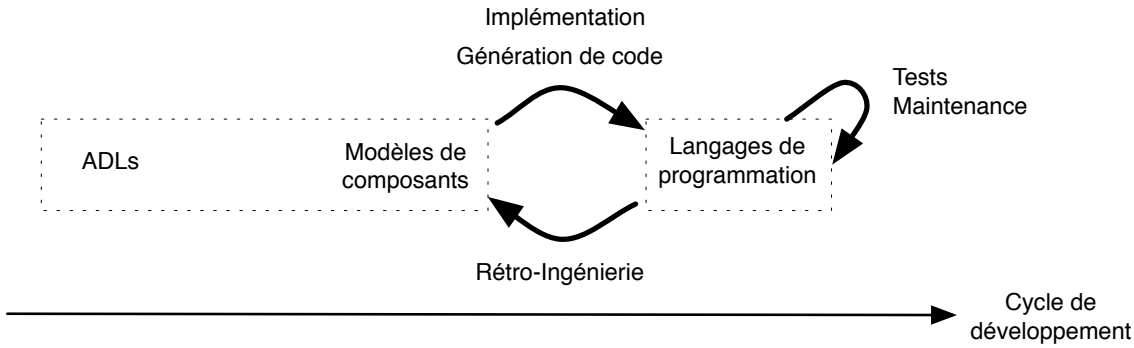

FIG. 2.24 : Les langages de programmation : dernier maillon de la chaîne de développement (par composants)

Les langages à composants ont donc pour objectif principal d'offrir un support au niveau du langage de programmation pour faire de la POC. Ces langages proposent des concepts de haut niveau spécifiques pour l'approches à composants, ce qui facilite l'implémentation des composants mais aussi la génération de code ou encore la rétro-ingénierie du code source écrit dans ces langages. De plus, le test mais aussi la maintenance (évolutions mineures) sont des phases importantes du cycle de développement qui s'effectuent au niveau du code source. Les langages à composants sont donc indispensables à l'essor du développement par composants.

Les principaux langages à composants existants sont actuellement construits par extension de langages à objets existants comme ArchJava ou ComponentJ qui sont des extensions du langage Java. Or, les langages à objets ne sont peut-être pas vraiment adaptés pour faire de la programmation par composants comme cela est souligné dans [Fröhlich et Franz, 1999; Beugnard, 2005]. Il semble que peu de travaux se soient intéressés à vraiment construire un langage à composants en se concentrant sur les problématiques au cœur de cette approche. Nous pouvons citer par exemple les travaux sur le langage Lagoona [Fröhlich *et al.*, ; Fröhlich *et al.*, 2005] qui est un langage dans lequel le concept de *classe* a été revu et qui introduit deux nouveaux mécanismes : *stand-alone messages* et *generic message forwarding*. Bien que ce langage soit intéressant, il ne permet pas de décrire une architecture logicielle en terme de composants interconnectés comme on pourrait le faire en ArchJava par exemple. Lagoona ne s'est pas assez inspiré, selon nous, des récents travaux sur les approches à composants.

C'est dans ce contexte que nous proposons dans le chapitre 3 une étude visant à définir les abstractions et mécanismes de base d'un langage à composants. Notre objectif est de construire un langage à composants en nous concentrant en premier lieu sur les problématiques centrales de ce mode de développement à savoir le découplage et l'assemblage.

CHAPITRE 3

Spécification de SCL : un langage à composants minimal

Un langage de programmation est censé être une façon conventionnelle de donner des ordres à un ordinateur. Il n'est pas censé être obscur, bizarre et plein de pièges subtils (ça ce sont les attributs de la magie).

Dave SMALL

Préambule

Ce chapitre présente le cœur du travail de cette thèse qui est la spécification du langage SCL *(Simple Component Language) [Fabresse* et al.*, 2007a; Fabresse* et al.*, 2007b]. Nous commençons par présenter les motivations et les objectifs qui ont guidé la conception de* SCL*. Ensuite, chaque section est une discussion autour d'un besoin de la programmation par composants afin d'y répondre au mieux dans* SCL*. Les choix faits sont ainsi discutés et argumentés tout au long de ce chapitre. Finalement, après une synthèse du noyau de* SCL*, ce chapitre se termine par un bilan de ce qui a été réalisé en mettant en exergue les spécificités de* SCL*.*

3.1 Motivations et Objectifs

Au delà des discours et de l'existence de différents modèles et langages, la programmation par composants (PPC) est encore peu répandue et nous disposons de peu d'expérimentation de sa pratique. Notre but premier est d'examiner point par point les besoins du programmeur afin de déterminer quels doivent être les traits essentiels d'un langage à composants (LAC), ceux qui sont accessoires voire inutiles ou nuisibles.

Cette volonté de créer un nouveau langage de programmation pour faire de la programmation par composants (PPC) est née de plusieurs constats. Le premier est que les langages de programmation traditionnels (procéduraux ou objets) ne sont pas adaptés à la PPC et pourtant utilisés pour mettre en pratique ce mode développement. Ils imposent aux programmeurs de respecter des conventions ou des schémas de conception afin d'implémenter les concepts de l'approche composant en utilisant ceux du langage de programmation utilisé. Par exemple, la programmation d'un composant *Javabeans*, s'effectue avec le langage Java en respectant des conventions de nommage et en appliquant le schéma de conception Observateur [Gamma *et al.*, 1995] (*cf.* chapitre 2). Cela complexifie l'implémentation mais aussi le test, la maintenance ou encore l'évolution du code source de l'application qui ne profite pas directement et simplement des avantages apportés par l'approche composant. Notre deuxième constat concerne les langages de programmation tels que ArchJava, ComponentJ, Lagoona ou Piccola. Bien que ces langages intègrent effectivement des abstractions et des mécanismes spécifiques pour faire de la PPC, ils ne présentent pas tous les mêmes alors qu'ils utilisent souvent un vocabulaire commun. Cette disparité invite à mieux identifier et définir les besoins de la PPC.

Finalement, on peut résumer ces problématiques soulevées par ces deux constats par les questions générales suivantes : Qu'est-ce qu'un langage à composants ? Quelles sont ses structures de données et de contrôles fondamentales ?

Tout d'abord, en accord avec les objectifs de cette thèse et la synthèse des approches à composants présentés respectivement dans les chapitres 1 et 2, nous posons la définition suivante :

Définition 1 (Langage à composants) *Langage de programmation permettant d'une part de programmer des composants logiciels réutilisables (*design for reuse*) qui pourront être mis à disposition dans des bibliothèques (on parle aussi de composants sur étagères) et d'autre part de programmer des applications par assemblage de composants logiciels pré-fabriqués (*design by reuse*) c'est-à-dire de décrire des architectures logicielles en terme d'assemblage de composants choisis dans des bibliothèques.*

Cette définition générale a pour but de fixer les idées sur la signification de la notion de « langage à composants » (LAC) dans cette thèse. Un LAC est destiné à être utilisé à la fois par un programmeur de composants et par un architecte d'application (*cf.* chapitre 1). Parmi les approches à composants actuelles, seules certaines sont conformes à cette définition comme ArchJava, ComponentJ, Piccola ou encore Lagoona. En effet, tous ces langages permettent de faire de la PPC et proposent de nouvelles structures de données et de contrôle à cet effet contrairement aux propositions qui utilisent principalement Java comme Julia (l'implémentation de référence de Fractal) qui est un framework écrit en Java.

Dans la suite de ce chapitre nous décrivons SCL (*Simple Component Language*) qui est notre proposition de langage à composants. Avec SCL, nous souhaitons proposer un langage à composants le plus minimal possible. Notre idée est de ne pas intégrer un mécanisme (notamment objet) s'il n'est pas indispensable au développement par composant. On peut énoncer plus précisément cet objectif de la façon suivante :

Objectif 1 *Proposer un langage à composants qui intègre les concepts et mécanismes nécessaires et suffisants pour faire de la programmation à base de composants.*

Ce premier objectif souligne notre volonté de minimalité afin de produire un langage simple et cohérent. Cette idée est bien exprimée par une phrase de Saint-Exupéry : « *Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher* »[1]. Nous ne prétendons pas cependant vouloir atteindre la perfection.

Notre second objectif est de proposer un langage de programmation qui permette au programmeur de s'exprimer facilement et donc de proposer des abstractions et mécanismes de haut niveau. Comme le souligne J. Privat dans sa thèse [Privat, 2006], « *un bon langage de programmation doit permettre au programmeur de s'exprimer facilement, il doit donc être le plus proche possible du mode de pensée humain* ». Cela doit permettre une meilleure *expressivité* du langage qui est un indicateur de la capacité des programmeurs à s'exprimer succinctement et directement. Ces propos sont résumés par l'objectif suivant :

Objectif 2 *Les concepts et mécanismes intégrés dans un langage à composants doivent être de haut niveau et en adéquation avec l'approche composant afin de faciliter l'expressivité des programmeurs.*

Une des difficultés actuelles de l'approche composant est de construire des composants réutilisables dans différentes applications. Or, ils sont souvent originellement conçus pour les besoins d'une application particulière. Cette pratique semble inévitable surtout si aucun des composants disponibles sur étagère ne satisfait les besoins exigés. Nous pensons que cette problématique doit être prise en compte directement au niveau des langages de programmation. En effet, c'est au langage à composants d'offrir les mécanismes permettant de définir un composant indépendamment du contexte d'exécution pour lequel il est destiné a priori. Cette volonté de rendre les composants indépendants de leur environnement (de définition, d'exécution au sein d'une application particulière, etc.) a pour objectif de maximiser leur réutilisation potentielle. Ceci nous donne notre dernier objectif.

Objectif 3 *Un langage à composants doit garantir autant que possible l'écriture de composants indépendamment de tout contexte.*

Ce troisième objectif est fondamental pour comprendre les choix effectués dans SCL. Dans la suite, on parlera de *non-anticipation* lors de la définition d'un composant c'est-à-dire de la nécessité de ne pas pré-câbler dans le code d'un composant ce qui pourrait empêcher son utilisation dans

[1] Extrait du livre « Terre des hommes » paru en 1939.

certains contextes. Autrement dit, tout code relatif à un contexte d'utilisation particulier, comme celui relatif aux communications avec des entités extérieures par exemple, ne doit donc pas être intégré dans le code d'un composant.

Ces trois principaux objectifs étant posés, les sections qui suivent détaillent maintenant les choix effectués pour SCL. Pour cela, nous avons adopté une démarche constructive où l'on identifie un à un les besoins de la PPC et intégrons les concepts et/ou mécanismes couvrant ce besoin en s'inspirant des propositions existantes décrites dans le chapitre 2. Nous pensons que cette démarche contribue à remplir notre premier objectif. Dans tous les cas, l'ensemble des choix effectués lors de la conception de SCL ont été décidés dans l'optique de remplir au mieux ces trois objectifs.

3.2 Vocabulaire de base

Pour discuter des concepts liés à la spécification de SCL, nous avons besoin de définir un vocabulaire de base. Nous définissons ce vocabulaire de base à l'aide de termes volontairement généraux afin de ne pas introduire d'ambiguïtés avec l'existant ou de choix implicites.

Un vision imagée des composants est : « *A component is a static abstraction with plugs* » [Nierstrasz et Dami, 1995]. Les prises (*plugs* aussi appelées *hooks*) d'un composant constituent ses uniques points d'accès au même titre que les pattes d'un composant électronique.

Définition 2 (Prise) *Points d'accès à un composant permettant son assemblage.*

Elles contribuent à renforcer l'encapsulation du composant qui ne peut être accédé que via l'une de ses prises et augmentent l'indépendance des composants par rapport à leur environnement (d'autres composants) en rendant explicite les interactions faites avec celui-ci. A travers ses prises, un composant fournit une ou plusieurs *fonctionnalités*.

Définition 3 (Fonctionnalité) *Traitement réalisé par un composant.*

Pour utiliser une fonctionnalité offerte par un composant, il faut l'*invoquer*. Un composant peut ne pas être directement utilisable c'est-à-dire que l'invocation d'une de ses fonctionnalités ne fonctionnera pas s'il n'est pas *assemblé* au préalable. Nous distinguons deux mécanismes d'assemblage de composants : la *connexion* et la *composition* qui sont respectivement détaillés dans les sections 3.7 et 3.11. De façon générale, connecter des composants consiste à lier leurs prises. Ces *liaisons* permettent de satisfaire les dépendances des composants exprimées via leurs prises en utilisant les fonctionnalités fournies par d'autres composants via leurs prises aussi. La composition de composants consiste à construire un nouveau composant en connectant des composants existants. Ces nouveaux composants sont dits *composites* car ils sont eux-mêmes constitués de composants plus élémentaires appelés *sous-composants*.

Illustrons ce vocabulaire de base à travers une analogie avec le langage C. Par exemple, considérons les fichiers objets (.o) comme des composants. Les prises d'un composant sont les symboles présents dans la table des symboles du fichier objet. On distingue les prises femelles (symbole non

défini correspondant à une fonction appelée mais non définie dans un fichier .c) et les prises mâles (symbole défini correspondant à une fonction définie dans un fichier .c). On peut voir un composite comme un fichier de bibliothèque statique (.a) obtenu par agrégation de fichiers objet. Cette analogie montre ici ses limites puisqu'un composite peut normalement fournir de nouvelles fonctionnalités non présentes dans ses sous-composants. Dans cette analogie, la connexion de composants est réalisée par l'outil d'assemblage (*linker*) lorsqu'il lie chaque symbole non-défini avec un unique symbole défini. En annexe A.1, nous montrons qu'il est possible, dans une moindre mesure et en respectant des règles précises, d'adopter un style de programmation par composants en utilisant des langages de programmation standards comme le C ou Java.

Dans la suite de ce chapitre, nous définissons d'une part la structure des composants en SCL c'est-à-dire comment sont représentés leurs prises et leurs fonctionnalités, et d'autre part les mécanismes tels que l'invocation d'une fonctionnalité, la liaison de prises, la connexion et la composition de composants.

3.3 Faut-il des descripteurs de composant ?

Il existe deux grandes familles de langages à objets : les langages à classes (ou descripteurs) et les langages à prototypes [Lieberman, 1986; Abadi et Cardelli, 1996]. Les langages à classes sont aujourd'hui très largement répandus même si des langages à prototypes sont encore utilisés dans des domaines spécifiques comme par exemple le web avec Javascript [Flanagan, 1998]. Dans le monde objet, les termes *classe* et *instance* désignent respectivement les descriptions d'objets dans le code et les objets eux-mêmes lors de l'exécution. Par analogie, on peut se demander si un langage à composants doit être construit sur un modèle avec des descripteurs[2] ou avec des prototypes.

Dans le monde composant, il n'existe pas deux termes bien identifiés pour désigner les descripteurs et les instances. Ainsi, le terme « composant » peut aussi bien désigner un descripteur instanciable qu'une instance c'est-à-dire une entité en mémoire à l'exécution. Cette ambiguïté de vocabulaire se traduit par des différences dans la littérature et les langages. Par exemple, un descripteur est désigné par le terme « classe de composant » (mot-clé `component class`) en ArchJava alors qu'il est désigné par celui de « composant » (mot-clé `component`) en ComponentJ. Ainsi, lorsqu'on parle d'un « composant », il s'agit d'une instance d'une classe de composant en ArchJava alors qu'il s'agit descripteur en ComponentJ.

En dehors de ce problème de vocabulaire, on constate que la plupart des langages à composants sont construits sur un modèle à descripteurs et non sur un modèle à prototypes. Le seul exemple à ma connaissance de langage à composants à prototypes a été proposé dans [Zenger, 2002]. Un composant ne possède ni état ni identité et est créé via des primitives de *raffinement* de composants existants. Le composant de bootstrap est nommé `component` et il ne fournit et ne requiert aucun service. Les services d'un composant ne peuvent être utilisés directement et ce dernier doit préalablement être *instancié*. En effet, ce langage distingue les notions de *composant* et d'*instance de composant*. Le vocabulaire est assez surprenant pour un langage à prototypes et témoigne du manque de précision en ce qui concerne l'approche prototypique pourtant revendiquée. Toutefois cette proposition a le

[2] On n'utilise pas le terme « classe » pour éviter toute ambiguïté avec le monde objet.

mérite de soulever la question de savoir si un langage à composants peut être (ou doit être) construit sur un modèle à prototypes puisque la plupart des langages à composants (ArchJava, ComponentJ, etc.) sont actuellement bâtis sur un modèle à descripteurs.

Les arguments pour ou contre l'utilisation de descripteurs dans le monde des composants nous semblent similaires à ceux avancés dans le monde des objets [Dony *et al.*, 1992]. Les arguments donnés en faveur des langages à prototypes sont :

- la simplicité ; cet argument n'a été retenu dans le monde des objets puisque la plupart des LOO sont actuellement construits sur un modèle à descripteurs et il ne semble pas non plus s'imposer dans le monde des composants ;
- l'indépendance des prototypes qui ne sont pas liés à un descripteur ; cet argument ne constitue pas une limitation des langages à descripteurs dans le monde des composants puisqu'un composant est empaqueté sous la forme d'une archive contenant son descripteur notamment (*cf.* chapitre 2) afin d'être facilement mis sur étagère ou déployé.

A l'inverse, la principale critique faite aux langages à prototypes est la difficulté à décrire des abstractions (caractéristiques partagées par toute une famille d'objets) de façon réellement satisfaisante bien que des solutions aient été proposées dans certains langages comme les *traits* en SELF [Ungar et Smith, 1987]. Dans l'optique de réaliser des applications de taille importante, ce manque d'abstraction et donc de structuration est une forte limitation. En SCL, nous avons donc choisi d'adopter une approche avec descripteurs.

Choix 1 *Un* composant *est une entité logicielle présente à l'exécution, instance d'un* descripteur de composant.

Ce choix en faveur d'une approche à descripteurs soulève deux questions. La première : *Est-ce les descripteurs ou les composants qui sont mis sur étagère et réutilisés ?* Tout d'abord, rappelons qu'une étagère est considérée ici comme une bibliothèque d'archives (fichiers jar par exemple) contenant des entités logicielles réutilisables dans différentes architectures (applications). Dans la majorité des approches à composants, l'archive mise sur étagère contient essentiellement un ou plusieurs descripteurs. En effet, un composant étant une entité présente à l'exécution, elle peut difficilement être mise sur étagère afin d'être réutilisée. Toutefois, le modèle *Javabeans* offre la possibilité d'inclure un composant *Javabeans* sérialisé (enregistré sur disque) dans une archive. Cela permet notamment de mettre sur étagère des composants initialisés (possédant des valeurs) par exemple. Dans les autres approches, des valeurs initiales peuvent aussi être spécifiées dans un fichier de configuration inclut dans l'archive et lu lors de l'instanciation du descripteur. Les deux approches offrent donc les mêmes possibilités et nous avons choisi en SCL de mettre uniquement les descripteurs sur étagère. Si nous avions choisi l'approche proposée par le modèle *Javabeans*, il aurait fallu traiter différemment les archives contenant des descripteurs (instanciation) de celles contenant des composants (clone).

Choix 2 *Ce sont les descripteurs de composant qui sont mis sur étagère.*

La seconde question est : *Assemble-t-on des descripteurs ou des composants ?* En effet, l'assemblage peut s'adresser aussi bien aux entités conceptuelles (descripteurs) qu'aux entités à l'exécution

(composants). D'ailleurs, les principales approches à composants n'offrent pas toutes les mêmes possibilités sur ce point [Lau et Wang, 2005b; Lau et Wang, 2005a]. Par exemple, en *Javabeans* et ArchJava, seul l'assemblage d'instances de composants est possible. Par contre, en ComponentJ ou en Scala [Odersky et Zenger, 2005], ce sont les descripteurs de composants qui sont assemblables. En Scala, les composants — attention il s'agit de descripteurs — sont des classes et l'assemblage de composants correspond essentiellement à un mécanisme de combinaison de classes basé sur des *mixins*. Dans ce genre d'approches, l'instanciation d'un descripteur n'est possible que si toutes ses dépendances requises sont satisfaites. Ces modèles sont généralement sûrs mais contraignant puisqu'ils interdisent d'avoir des composants ayant des dépendances partiellement satisfaites. A l'inverse, en ArchJava, toutes les dépendances d'un composant peuvent ne pas être nécessaires si seulement une partie de ses fonctionnalités sont utilisés dans l'application. Nous pensons que l'idéal serait qu'un langage à composants qui propose des mécanismes d'assemblage aussi bien au niveau des descripteurs que des instances. En effet, cela permettrait de définir les descripteurs mais aussi de construire une application avec les mêmes mécanismes d'assemblage unifiés. Aucun langage n'intègre cela actuellement et il semble que cet objectif relève de la problématique de faire un langage à composants réflexif. En effet, dans un tel langage, les descripteurs seraient des composants à part entière et pourraient être assemblés. Dans cette thèse nous avons fait le choix suivant pour SCL :

Choix 3 *Un composant est une entité logicielle* assemblable.

Avec ce choix nous fixons que ce sont les instances qui sont assemblées. Comme nous l'avons dit, l'idéal serait de pouvoir assembler aussi bien des descripteurs de composants que des instances. Ce choix constitue donc une étape nous permettant de définir un langage simple. Il laisse toutefois la perspective de proposer une version réflexive de SCL où les descripteurs de composants seront des composants et par conséquent assemblables. De plus, assembler des instances plutôt que des descripteurs offre de bien meilleures perspectives en terme de dynamicité.

3.4 Comment représenter les fonctionnalités des composants ?

Les fonctionnalités représentent les traitements que peuvent accomplir des composants. Dans certains modèles, un composant n'a qu'une seule fonctionnalité. Par exemple, les *processus Unix* peuvent être considérés comme des composants accomplissant des traitements sur leurs flux d'entrée (*stdin*), de sortie (*stdout*) et d'erreur (*stderr*). Deux processus peuvent être combinés en redirigeant le flux de sortie du premier vers l'entrée standard du second par exemple. Ce style architectural, souvent appelé *pipes and filters*, est représenté sur la figure 3.1.

Toutefois, dans la plupart des modèles actuels, un composant possède plusieurs fonctionnalités qui sont représentées par des *services*. Un service est généralement une fonction (ou opération) définie par un composant, qui possède un nom, des paramètres[3] et un résultat. Un service est aussi

[3]Le terme « paramètre » (ou « paramètre formel ») permet de désigner une variable liée dans une fermeture lexicale, par exemple *x* est un paramètre dans la définition de fonction suivante *f(x)...*, alors que le terme « argument » (ou « paramètre réel ») permet de désigner la valeur substituée à un paramètre lors de l'application d'une fonction, par exemple *3* est un argument dans l'appel de fonction suivant *f(3)*.

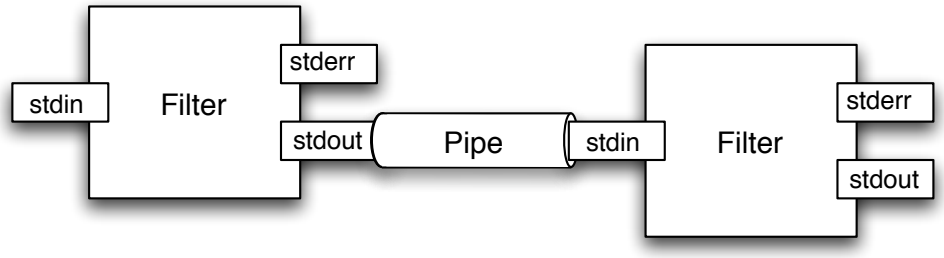

FIG. 3.1 : Les processus Unix (*pipes and filters*) vus comme des composants

défini comme un ensemble de fonctions dans certains modèles. Par exemple, un service de gestion de compte peut se décomposer en trois sous-services : consultation, retrait et dépôt.

L'*invocation de service* (ce mécanisme est présentée de façon plus complète et détaillée dans la section 3.9) est le terme utilisé dans cette thèse pour désigner le mécanisme permettant à un composant d'exécuter un service suite à la réception d'une invocation et émettre des invocations de services à d'autres composants. Ce terme permet de ne pas le confondre avec l'*envoi de message* dans le monde objets. Comme nous le verrons dans la suite, la différence avec l'envoi de message est liée au mécanisme d'assemblage spécifique au monde des composants. Le définition suivante permet de fixer le consensus adopté par la majorité des modèles actuels :

Définition 4 (Service fourni) *fonction définie dans le code source d'un composant et offerte aux autres composants pour qu'ils puissent l'invoquer.*

Un service fourni peut être assimilé à une *méthode publique* dans le monde objet. Un composant peut aussi posséder des services qu'il ne fournit pas afin de factoriser son implémentation par exemple.

Définition 5 (Service interne) *service non-accessible à l'extérieur du composant qui le définit.*

Un composant exprime aussi les services qu'il requiert d'autres composants. Ces *services requis* ne sont pas définis par le composant qui peut tout de même les invoquer dans son code.

Définition 6 (Service requis) *service nécessaire au fonctionnement d'un composant (invoqué dans le code de ses services fournis) et fourni par d'autres composants.*

3.5 Que sont les prises des composants ?

Nous avons défini les prises d'un composant comme étant ses points d'accès permettant son assemblage (*cf.* définition 2). Dans cette section, nous allons essayer de déterminer la nature de ces prises et leur rôle exact.

D'après la section précédente, on peut légitimement supposer que les services peuvent jouer le rôle de prise des composants. La connexion de composants serait alors réalisée par la liaison d'un service requis avec un service fourni. Ce mécanisme de connexion basé sur l'appariement de services rappelle l'analogie avec le langage C présentée dans la section 3.2. Toutefois, ce mécanisme est souvent critiqué pour sa granularité trop fine : « *[...] component systems should offer the possibility to connect many required and provided services at once* » [McDirmid *et al.*, 2001a]. Nous pensons en effet qu'il faut proposer un mécanisme de connexion offrant un niveau de granularité plus important afin de permettre un meilleur passage à l'échelle. Cela suppose donc que les prises des composants aient un niveau de granularité intermédiaire compris entre celui d'un unique service et celui d'un composant entier.

Les *interfaces de composants* sont des prises de granularité intermédiaire : « *a component can only be accessed through well defined interfaces* » [Szyperski, 2002]. Attention, il ne faut pas confondre avec la notion d'« *interface* » au sens des langages à objets comme Java avec celle d'« *interface de composants* » même si ce n'est pas sans rapport comme nous le verrons dans la suite. Les approches à composants utilisent des concepts différents pour représenter les interfaces de composants comme : les ports, les interfaces (au des LOO), les interfaces de port [Aldrich *et al.*, 2002b], les protocoles [Plásil et Visnovsky, 2002], les contrats [Jézéquel et Meyer, 1997] ou encore les réseaux de Pétri [Bastide et Barboni, 2006]. Par exemple en Fractal, les interfaces de composants — appelées interfaces par abus de langage [Bruneton *et al.*, 2004] — sont des points d'accès du composants décrits par une interface Java. A cause de cet abus de langage, les interfaces de composants sont souvent confondues avec les interfaces Java alors qu'elles remplissent des fonctions différentes comme nous le verrons dans la suite. En UML, les interfaces de composants sont représentées à l'aide des concepts de port et d'interface. Un composant UML fournit ou requiert des interfaces à travers ses ports mais il peut aussi directement (sans passer par un port) implémenter ou requérir une interface. En ArchJava, les composants ont des ports ; chaque port est décrit par une seule interface qui spécifie les services fournis et/ou requis à travers ce port. Face à cette diversité, nous devons intégrer une représentation simple et suffisamment générale des interfaces de composants en SCL.

Tout d'abord, nous pensons qu'il faut distinguer au moins deux problématiques dans lesquelles sont utilisées les interfaces de composants :

- l'assemblage des composants où les interfaces de composants sont vues comme les points de connexion des composants et sont le support des invocations de services. Le concept de port est généralement utilisé pour représenter cette facette des interfaces de composants.
- la description des composants où les interfaces de composants sont vues comme des descriptions (syntaxiques, comportementales, de qualités) permettant par exemple la vérification des assemblages ou la validation des utilisations du composant. Les notions d'interface (au sens des LOO), de protocole ou de contrat sont par exemple utilisées pour représenter cette facette des interfaces de composants.

La suite de cette section présente ces deux problématiques séparément en se concentrant dans un premier temps sur le concept de port et ses variations puis sur le concept d'interface au sens large (interface des LOO, protocole, contrat). Nous détaillons ensuite un exemple en SCL afin d'illustrer les choix que nous avons fait en ce qui concerne les interfaces de composant.

3.5.1 Notion de port

Les ports remplissent une double fonction pour le composant, ils sont le support de ses connexions et de ses communications via l'invocation de ses services fournis et requis. L'étude des approches existantes nous a permis de distinguer les *ports unidirectionnels* et les *ports bidirectionnels*. Les ports unidirectionnels regroupent un ensemble de service fournis ou requis comme en ComponentJ ou Fractal. On parle alors de *port fourni* et de *port requis*. A travers un port bidirectionnel des services peuvent être requis et fournis comme c'est le cas en ArchJava ou en UML.

Dans les deux cas, un port définit un point de vue et une politique de sécurité. Les ports requis définissent les points de vue que le composant peut avoir sur des composants externes tandis que les ports fournis définissent les points de vue que les composants externes pourront avoir sur ce composant. De même, lorsqu'un composant est accédé via l'un de ses ports, ce dernier est le garant d'une politique de sécurité, c'est-à-dire que les composants clients ne pourront utiliser que les services disponibles via ce port. En permettant de panacher des services requis et des fournis au sein d'un même port, les ports bidirectionnels permettent de décrire plus précisément les dépendances entre les services. Ils peuvent être considérés comme le point de vue du composant sur une collaboration (au sens d'UML).

Bien que cela puisse sembler restrictif au premier abord, nous avons choisi d'intégrer, des ports unidirectionnels en SCL. Ce choix 4 est essentiellement motivé par notre objectif 3 de non-anticipation. En effet, utiliser deux ports, l'un requis et l'autre fourni à la place d'un seul port bidirectionnel lors de la conception d'un composant permet ensuite de laisser l'architecte décider si le composant qui va utiliser les services fournis est le même que celui qui va fournir les services requis. Imposer que ce soit le même composant client lors de la conception est restrictif et relève de la description de contraintes d'assemblage décrites dans la section 3.5.2. Notre idée est de fournir des composants sur étagère simples et peu contraints de sorte qu'ils puissent être adaptés et réutilisés dans différents contextes. Les ports uniderectionnels sont aussi plus simples à comprendre et à utiliser en pratique. De plus, la perspective de construire (si nécessaire) des ports bidirectionnels en agglomérant des ports unidirectionnels reste envisageable.

Choix 4 *Un composant est doté de ports* unidirectionnels.

Suite à ce choix 4, la définition 7 pose le vocabulaire que nous utilisons dans la suite.

Définition 7 (Port requis (resp. *Port fourni*)) *point d'assemblage d'un composant possèdant un nom et à travers lequel est requis (resp. fourni) un ensemble de services.*

Il est à noter qu'en SCL, deux ports d'un même composant ne peuvent pas posséder le même nom mais peuvent exporter le même service fourni (s'il s'agit de ports fournis) ou requérir des services de signatures identiques (s'il s'agit de ports requis).

Notation graphique. La figure 3.2 présente les conventions graphiques utilisées dans la suite. Nous n'avons pas utilisé directement la notation UML car elle trop dépendante du modèle de composants

sous-jacent. Par exemple, en UML ce sont les interfaces qui sont fournies ou requises et non les ports. La notation que nous proposons s'inspire toutefois largement de la notation UML. Un composant est un rectangle. Les ports sont représentés sur la périphérie des composants. Un demi-cercle convexe représente un port fourni alors qu'un demi-cercle concave représente un port requis.

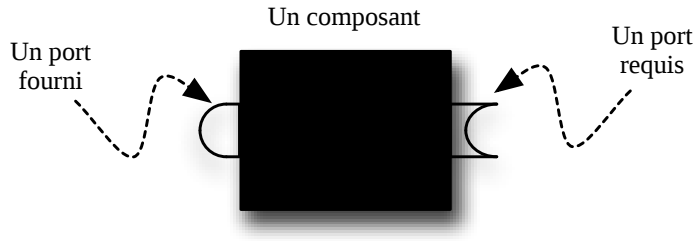

FIG. 3.2 : Présentation de conventions graphiques adaptées à SCL

3.5.2 Notion d'interface

De façon générale, les interfaces sont un support de description des composants permettant de spécifier comment ils peuvent être assemblés ou utilisés au sein d'une architecture. Les interfaces se situent soit à un niveau local (associées à un port) soit à un niveau global (associées à un composant). Les interfaces définissent des contrats généralement classés en quatre niveaux [Beugnard *et al.*, 1999] :

Syntaxique : ces contrats spécifient les signatures des services fournis et requis (nom, paramètres, résultat, exceptions).

Comportementaux : ces contrats spécifient comment peut être utilisé un composant.

De synchronisation : ces contrats sont nécessaires dans un contexte distribué et/ou concurrentiel afin de spécifier le comportement des composants en terme de synchronisations entre les invocations de services.

De qualité de service : ces contrats sont généralement cruciaux dans le domaine de l'informatique embarquée ou temps réel afin d'assurer des contraintes de qualité au sein d'une architecture globale.

Les approches à composants proposent des notations formelles ou informelles adaptées à leurs besoins pour décrire des contrats de ces différents niveaux. Les contrats syntaxiques sont les plus simples et peuvent être définis à l'aide de langages de définition d'interface (*Interface Definition Language* – IDL) ou directement avec la notion d'*interface* des LOO (comme les interfaces Java). D'ailleurs, les principaux langages de programmation à composants se limitent généralement à ce niveau de contrat via l'utilisation d'interfaces comme en Fractal ou ComponentJ. En ArchJava, une interface spécifie aussi pour chaque service s'il est requis ou fourni puisque chaque port est bidirectionnel et est décrit par une seule interface. SOFA supporte des contrats comportementaux puisqu'il permet de décrire les enchaînements d'invocations de services valides – notion de *protocole* – avec un formalisme à base d'expressions régulières. Un tel formalisme a aussi été utilisé pour décrire les

protocoles de services web [Tremblay et Chae, 2005]. Prenons l'exemple d'un composant dédié à des communications réseaux fournissant les trois services suivants : `open(adr)` (ouvre une connexion réseau à l'adresse spécifiée par le paramètre `adr`), `send(data)` (envoie les données `data` à travers la connexion) et `close` (ferme la connexion). Un protocole pour ce composant permet de décrire que l'ordre d'invocation de ces trois services pour une utilisation valide est : `open` (invoqué une seule fois), puis `send` (autant de fois que nécessaire) et finalement `close` (une seule fois). D'autres formalismes peuvent être utilisés pour exprimer les contrats comportementaux comme les machines à états d'UML, les langages à automates [de Alfaro et Henzinger, 2001] ou encore les protocoles symboliques [Pavel *et al.*, 2005]. Les contrats de synchronisation ou de qualité de service sont peu caractérisés dans les langages à composants existants et les solutions apportées à ces problématiques restent confinées dans des ADL dédiés à la spécification et ne permettant pas de produire directement une application exécutable.

Le domaine de la vérification des architectures logicielles à base de composants vise à valider les assemblages et les utilisations des composants en se basant sur le respect des contrats établis par les interfaces des composants. Vérifier la conformité de contrats comportementaux dépend du formalisme choisi. Par exemple, si les comportements des composants sont décrits avec des réseaux de Pétri à objets (RdPO) [Passama, 2006], vérifier la validité d'une architecture revient à construire et valider le RdPO de l'architecture globale à partir des RdPO des composants et de leurs connexions. Dans le cas du système *ConFract* [Collet *et al.*, 2005], un contrat a une portée (interface, composant, ensemble d'interfaces, etc.) et il est créé à partir de spécifications exécutables écrites en CCL-J (*Component Constraint Language for Java*. Le respect des contrats est ensuite automatiquement vérifié pendant l'exécution.

Du point de vue syntaxique, on constate que la conformité des interfaces entre elles est souvent établie par la compatibilité des types qui leur ont été associés. Lorsque deux composants sont connectés via leurs ports, cela suppose que les types de leurs interfaces sont compatibles. Par exemple, un port requis typé par une interface I_1 et connecté à un port fourni typé par une interface I_2 suppose que le type défini par I_1 est un super-type de celui défini par I_2. On distingue généralement deux sortes de systèmes de types : ceux basés sur des noms (*named type systems*) comme en Java et ceux basés sur la structure (*structural type systems*) comme en Objective CAML [Loulergue, 2004]. L'utilisation de noms permet de capturer la sémantique [Büchi et Weck, 1998] :

> « *[...] types stand for semantical specification. While the conformance of an implementation to a behavioral specification cannot be easily checked by current compilers, type conformance is checkable. By simply comparing names, compilers can check that several parties refer to the same standard specification.* »

Les systèmes de types structurels [Cardelli, 1997] sont moins expressifs mais offrent un meilleur découplage entre les entités puisque la relation de sous-typage est déduite de la structure des interfaces et non d'un nom commun. Par exemple, spécifier qu'un « composant requiert une pile » est plus sémantique que « un composant requiert deux services `push` et `pop` ». Toutefois, dans le premier cas, il doit exister une interface pile (de façon globale) et le composant ne pourra être connecté qu'à un autre composant fournissant cette même interface pile ou l'un de ses sous-types. Dans le deuxième cas, une interface peut être un sous-type d'une autre sans qu'elles aient de relation directe puisque la relation de sous-typage est déduite de la structure.

En SCL, nous avons choisi d'intégrer un modèle simple en ce qui concerne la description des composants. Ainsi, une interface est locale et attachée à un port.

Choix 5 *Une* interface *est associée à un port.*

Actuellement, nous n'avons pas intégré à SCL d'interfaces globales à un composant. Cela serait certainement nécessaire pour définir des contrats globaux afin de faire de la vérification d'architectures ou décrire des contraintes portants sur plusieurs ports.

Toutefois, nous nous limitons pour l'instant à des vérifications du niveau syntaxique pour lesquelles des interfaces de ports sont suffisantes.

Choix 6 *Une* interface *spécifie un ensemble de signatures des services. La* compatibilité *d'interface repose sur la relation de sous-typage entre leurs types qui est basée sur l'inclusion des ensembles de signature de services.*

Une interface n'a pas besoin d'un nom en SCL et la relation de sous-typage est structurelle. Cette décision est motivé par notre volonté de découplage des composants. En effet avec un système de types basé sur les noms, deux composants ne peuvent être connectés que si les types de leurs interfaces sont en relation directe. Une approche structurelle semble offrir une meilleure indépendance des composants lors de leur définition au détriment de la sémantique comme nous l'avons vu. Cette vision est parfois appelée « *Duck typing* » (typage du canard) [Anantharam, 2001] :

> « *This method . . . [of] . . . just relying on what methods it supports is known as "Duck Typing", as in "if it walks like a duck and quacks like a duck. . . ". The benefit of this is that it doesn't unnecessarily restrict the types of variables that are supported. If someone comes up with a new kind of list class, as long as it implements the join method with the same semantics as other lists, everything will work as planned.* »

3.5.3 Exemple en SCL

En résumé, une interface de composant est représentée en SCL par un port unidirectionnel décrit par une interface. Un exemple de composant SCL est montré à la figure 3.3. Le composant `pm` est une instance du descripteur PASSWORDMANAGER. Chacun de ses ports est décrit par une interface qui est représentée sur le schéma par un rectangle en pointillé contenant les signatures de services décrites par cette interface. Ce composant permet d'une part de générer des mots de passe composés de lettres et de chiffres via son service `generatePwd` ou composés de chiffres uniquement via son service `generateADigitsOnlyPwd` et d'autre part de vérifier qu'un mot de passe n'est pas trop simple (taille suffisante, mot inexistant dans un dictionnaire, etc.) via son service `isValid`. Pour fournir ces trois services, un composant PASSWORDMANAGER a besoin d'un service de génération de nombre aléatoires nommé `getRandomNumber` qu'il requiert à travers son port `Randomizer`.

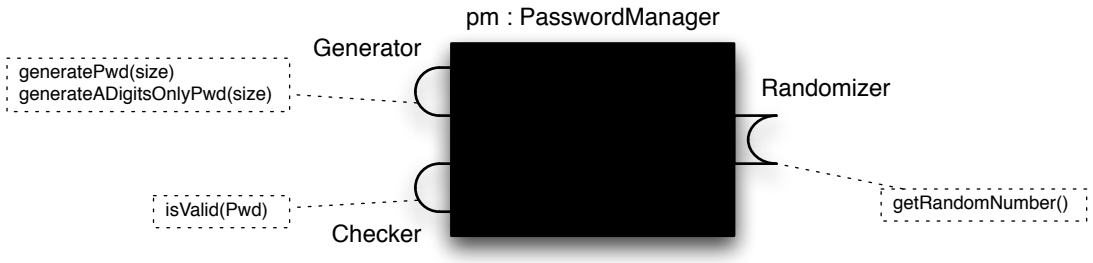

FIG. 3.3 : Représentation graphique d'un composant `pm` instance d'un descripteur de composant PASSWORDMANAGER, possédant deux ports fournis `Generator` et `Checker` et un port requis `Randomizer`

Le code source 3.1 présente la déclaration du descripteur de composant PASSWORDMANAGER en SCL. Cette déclaration est faite dans un pseudo-langage proche de la syntaxe du langage Ruby[4]. Dans ce pseudo-langage, nous déclarons les interfaces à part pour faciliter la lecture mais leur définition pourrait être intégrée dans le code du descripteur de composant. Les signatures de services sont très simples (nom du service et arité) ce qui est suffisant pour nos besoins en SCL. La partie implémentation des services est détaillée dans les sections suivantes. La dernière ligne de code de cet exemple montre une instanciation de ce descripteur de composant via l'opérateur `new`.

```
interface IGenerator { generatePwd(size); generateADigitsOnlyPwd(size) }
interface IChecker { isValid(pwd) }
interface IRandomizer { getRandomNumber() }

descriptor PasswordManager {
   providedport Generator, IGenerator
   providedport Checker, IChecker
   requiredport Randomizer, IRandomizer

   def generatePwd(size) { ... }
   def generateADigitsOnlyPwd(size) { ... }
   def isValid(pwd) { ... }
}

pm := new PasswordManager
```

LISTING 3.1 : Déclaration d'un descripteur de composant PASSWORDMANAGER

[4]Pour la présentation de SCL, nous n'utilisons pas la syntaxe du prototype actuel (cf. chapitre 5) car elle est très proche de celle du langage Smalltalk qui est bien souvent mal connue. Nous avons donc opté pour un pseudo-langage (dérivé de Ruby) ayant une syntaxe plus proche des langages à objets actuels (Java, C#) tout en étant en typage dynamique.

3.6 Les *objets primitifs* sont-ils des composants ?

Un *objet primitif* est nativement manipulé par l'architecture cible. Les types de ces objets sont entre autres : les petits entiers, les booléens, les caractères et les flottants.

Dans les langages à objets comme Smalltalk, les objets primitifs sont représentés de façon uniforme au niveau du modèle mais bénéficient tout de même d'une implémentation spécifique pour des raisons d'efficacité. Il existe plusieurs techniques (e.g mise en boîte appelée *boxing*) permettant de simuler qu'un objet primitif est instance d'une classe tout en optimisant ses traitements par rapport à une instance réelle. Tous les langages à objets ne traitent pas uniformément les objets primitifs et les objets comme c'est le cas en C++ par exemple. Java distingue aussi les types primitifs (int, float, boolean, etc.) de ceux définis par des classes. Toutefois, depuis sa version 1.5, Java supporte supporte l'*auto-boxing* permettant la conversion automatique d'un objet primitif en une instance de classe et inversement. Considérer les objets primitifs comme des objets permet de proposer au programmeur une vision unifiée. Il manipule ainsi les objets primitifs comme des objets standards pouvant recevoir des messages et profiter du polymorphisme.

La question que l'on se pose dans cette section est la suivante : *Peut-on considérer les objets primitifs comme des composants dans un langage à composants ?* La plupart des modèles existants n'abordent pas cette problématique sauf Fractal qui considère que les objets Java multi-interfacés sont des composants de base ce qui permet de mettre fin à la définition récursive des composants [Bruneton *et al.*, 2002]. Par contre ArchJava distingue clairement les objets Java (instance d'une classe) et les composants (instance d'une classe de composant). Cette cohabitation entre les objets et les composants en ArchJava est problématique pour le programmeur. Par exemple, seul un objet peut être passé en argument lors d'un envoi de message. Afin de proposer une vision unifiée au programmeur en SCL, nous avons fait le choix suivant :

Choix 7 *Un objet primitif doit être considéré comme un composant possédant un unique port fourni nommé* `defaut` *à travers lequel il fournit un ensemble de services.*

Ce choix 7 permet d'intégrer les objets primitifs comme des composants comme l'illustre la figure 3.4. De façon analogue à Smalltalk où un entier est présenté au programmeur comme une instance de classe `SmallInteger` alors qu'il est traité d'une façon spécifique par la machine virtuelle, un entier est présenté au programmeur SCL comme un composant.

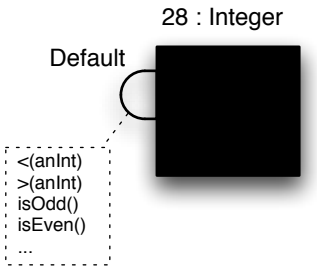

FIG. 3.4 : Un entier vu comme un composant en SCL

Cette solution en SCL permet d'unifier la vision des objets primitifs et des composants au niveau du modèle. Bien évidemment un interprète de SCL devra traiter ces objets primitifs de façon spécifique (*cf.* chapitre 5).

Dans le cas d'une implémentation avec un langage à objets, cette solution permet même de considérer un objet comme un composant n'ayant qu'un seul port nommé `default` à travers lequel toutes les méthodes publiques (y compris celles héritées) de l'objet sont fournies. Afin d'unifier complètement objets et composants, nous avons fait le choix suivant en SCL :

Choix 8 *Tout composant possède un port nommé* `default` *à travers lequel tous les services fournis par le composant sont accessibles. L'instanciation d'un descripteur de composant (via la primitive* **`new`***) retourne une référence sur le port* `default` *du composant nouvellement créé.*

Ce choix 8 nous permet d'unifier les objets et les composants car tous les composants possèdent un port nommé `default`. Nous imposons aussi que le mécanisme d'instanciation retourne une référence vers le port `default` du composant instancié au lieu d'une référence vers le composant directement. Empêcher la détention de référence directe sur un composant permet d'assurer que toutes les communications s'effectuent via un port et contribue à assurer l'intégrité des communications comme nous le verrons dans la section 3.9.

3.7 Qu'est-ce que lier des prises de composants ?

Le mécanisme de connexion est central dans l'approche composants (cf. chapitre 2) et se présente sous diverses formes dans les propositions existantes. En toute généralité, un composant a des *prises* et connecter des composants revient à brancher leurs prises. En SCL, les ports sont les prises des composants et la connexion de composants repose donc sur la *liaison de ports*[5]. Dans la suite, deux composants sont dits *connectés* s'il existe une liaison entre leurs ports.

On distingue les mécanismes de liaison de ports n-aires et binaires. Par exemple, ArchJava propose la primitive n-aire **`connect`** permettant de lier directement un ensemble de ports, alors que Fractal propose la primitive binaire **`bindFc`** qui lie un port requis à un port fourni. Bien que ces approches intègrent respectivement des ports bidirectionnels et unidirectionnels, elles soulèvent la question suivante : *Le mécanisme de liaison de ports doit-il être n-aire ou binaire ?*

En SCL, nous avons choisi d'intégrer un mécanisme de liaison de port binaire car il est moins sujet aux ambiguïtés comme l'est celui de ArchJava quand il existe plusieurs services fournis candidats pour un même service requis. De plus, les liaisons décrites par le mécanisme n-aire peuvent se décomposer en liaisons binaires dans la plupart des cas. Les seuls cas où cela n'est pas possible, c'est lorsqu'on veut associer des services requis à travers un même port requis à des services fournis par différents ports. Ces cas peuvent être aisément traités avec des composants adapteurs [Gamma *et al.*, 1995] ou les connecteurs de SCL que nous présentons dans la section 3.10. Par ailleurs, dans le cas de

[5] Nous avons choisi le terme liaison et non pas connexion que nous réservons aux composants afin d'éviter toute ambiguïté.

liaisons binaires, la validité d'une liaison entre ports est plus facilement vérifiée que dans le cas n-aire puisqu'elle repose uniquement sur la compatibilité des deux interfaces qui leur sont associées.

Choix 9 *Un port requis peut être lié à un unique port fourni si leurs interfaces sont compatibles.*

La liaison de ports permet d'établir que les invocations de services requis à travers un port aboutiront à l'exécution des services fournis via un autre port. Un port requis ne peut donc être lié qu'à un unique port fourni. Par contre, un port fourni peut être lié à plusieurs ports requis. Actuellement, les contrats définis par les interfaces des ports sont uniquement syntaxiques et la compatibilité entre interfaces repose sur l'inclusion ensembliste comme l'indique le choix 6. La figure 3.5 montre une liaison entre le port requis `Randomizer` d'un composant `pm` et le port fourni `Generator` d'un composant `rng`. La représentation graphique d'une liaison de port est une flèche à tirets directement empruntée à la notation UML. Le sens de la flèche indique le sens des invocations de services.

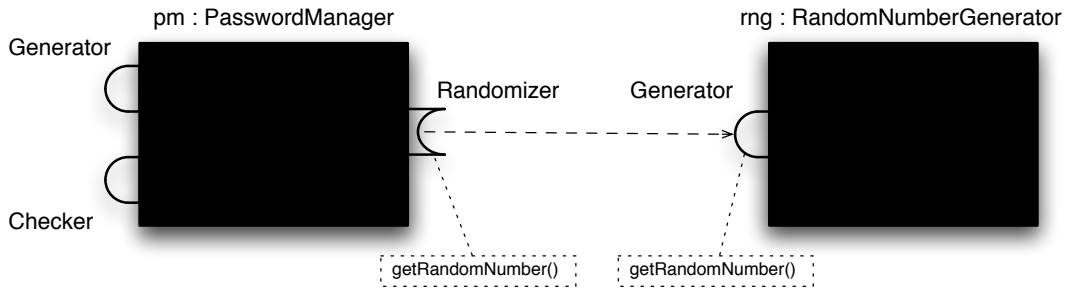

FIG. 3.5 : Un exemple de liaison de ports

Le code source 3.2 explicite comment lier des ports de composants. La construction syntaxique **`bind`** *port_requis* **`to`** *port_fourni* permet de mettre en place une liaison de ports. Cette structure sera enrichie dans les sections suivantes.

```
pm := new PasswordManager
rng := new RandomNumberGenerator

bind pm.Randomizer to rng.Generator
```

LISTING 3.2 : Liaison d'un port requis et d'un port fourni

3.8 Les liaisons multiples

Dans cette section, nous traitons le problème des relations multiples entre les composants. Par exemple, la relation entre un composant `Bank` et des composants `Client`. Une telle relation est « multiple » dans le sens où une banque peut posséder de nombreux clients.

Dans les approches existantes décrites dans le chapitre 2, deux approches existent pour traiter ces relations multiples :

– SOFA (*cf.* section 2.2.7) et Fractal (*cf.* section 2.2.6) permettent de fixer une cardinalité multiple à une interface de composant ce qui ce traduit dans le code du composant par un tableau ou une liste en tant qu'attribut de la classe implémentant le composant ;
– ArchJava (*cf.* section 2.2.8) permet l'ajout dynamique de ports grâce aux notions de *port interface* et de *connect pattern*. Ainsi, pour chaque nouveau composant `Client`, le composant `Bank` est automatiquement doté d'un port permettant de le lier à ce nouveau composant.

La solution que nous proposons en SCL est intermédiaire.

Choix 10 *Un composant peut posséder des* collections de ports.

Définition 8 (Collection de ports) *Collection ordonnée et nommée de ports requis ou de ports fournis. Chaque port de la collection peut être accédé par un index.*

Par ce choix 10 et la définition 8 nous posons qu'un composant SCL peut posséder des *collection de ports*. Les collections de ports sont déclarées par le programmeur SCL au même titre que les ports. Toutefois, la taille de ces collections n'est pas fixée par le programmeur — contrairement à SOFA— et un interprète SCL peut décider par exemple d'adopter une politique d'allocation de ports à la demande comme en ArchJava. Dans la suite on utilise le terme *port multiple* pour désigner une collection de ports. Ceci est un abus de langage car une collection de ports n'est pas un port. Toutefois, ce terme est intuitif et analogue aux « prises multiples » électriques.

Notation graphique. La figure 3.6 présente la convention graphique pour les ensembles de ports requis (resp. fournis) qui reprend la graphie d'un port requis (resp. fourni) en ajoutant une sur-épaisseur afin d'illustrer la multiplicité.

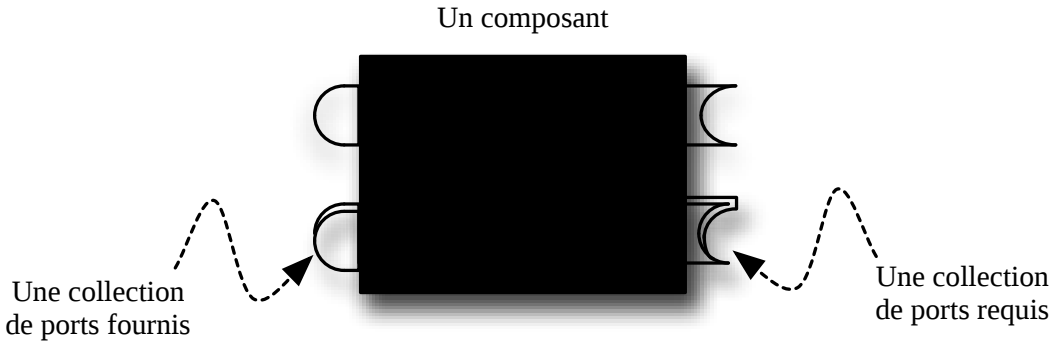

FIG. 3.6 : Représentation graphique des ports multiples en SCL

3.9 Les bases de l'invocation de services

Dans cette section nous présentons les choix qui nous conduits à définir qu'en SCL, une invocation de service référence :

– un port à travers lequel l'invocation est émise,
– un nom de service à exécuter (similaire au *sélecteur* dans le monde objet),
– un ensemble d'arguments.

Par exemple : `unPort.selecteur(arg1,arg2)`

3.9.1 Le port d'émission

Les invocations de services sont effectuées dans le code source des services des composants. Syntaxiquement, une invocation de service est très similaire à un envoi de message si ce n'est qu'elle s'effectue à travers un port appelé port d'émission.

Choix 11 *Une invocation de service s'effectue toujours via un port d'un composant.*

Ce choix 11 est motivé par la nécessité d'assurer l'*intégrité des communications* dans les architectures logicielles [Luckham *et al.*, 1995] c'est-à-dire que toute communication entre composants doit passer par une liaison décrite dans l'architecture. Comme les liaisons sont établies entre les ports, imposer que les invocations de services s'effectuent aussi via les ports rend explicite toutes les dépendances qu'elles induisent. En effet, s'il était possible d'invoquer directement un service d'un composant à partir d'un autre, cela introduirait une « dépendance cachées dans le code » entre ces deux composants qui ne serait pas décrite au niveau de l'architecture. De plus, pour les invocations effectuées via un port requis, comme illustré sur la figure 3.7, le composant destinataire est inconnu de `pm` lors de l'implémentation et donc impossible à référencer. C'est lors de l'assemblage que le composant destinataire sera choisi et fixé via le mécanisme de liaison de ports. Les invocations de services à travers un port requis provoqueront alors l'exécution des services de ce composant destinataire.

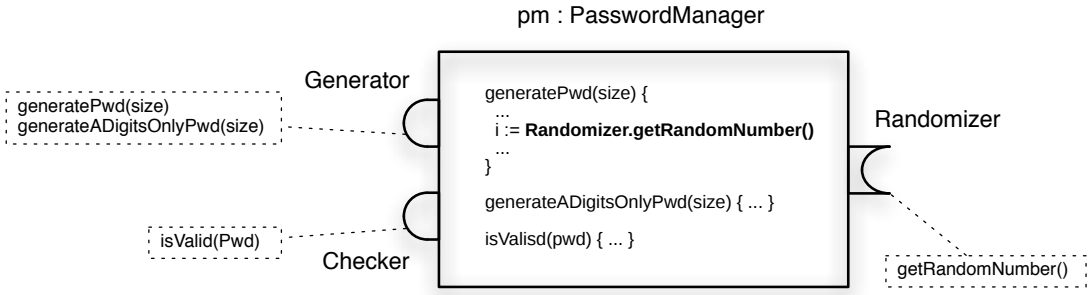

FIG. 3.7 : Invocation du service requis `getRandomNumber` à travers le port `Randomizer` du composant `pm`

Les invocations via les ports requis permettent un meilleur découplage puisque l'établissement ou le retrait des liaisons de ports permet de fixer le composant qui traitera effectivement les invocations de services faites à travers ces ports. Dans ce contexte, *les invocations de services à travers des ports fournis sont-elles nécessaires ?* Contrairement aux invocations via des ports requis, elles ne favorisent pas le découplage puisque le port d'émission référencé dans le code est un port fourni et de ce

fait il appartient au composant destinataire qui traitera effectivement l'invocation. Bien qu'elles ne favorisent pas le découplage et fixent le composant destinataire, les invocations de services à travers des ports fournis nous ont semblé nécessaires pour deux raisons : invoquer des services internes et invoquer des services fournis par des sous-composants dans les composites comme nous le verrons dans la section 3.11.

Puisque les invocations de services s'effectuent toujours via un port et que les services internes ne sont pas fournis, comment est-il possible d'invoquer des services internes ? Ce problème est généralement pas ou peu abordé dans les langages à composants existants. En Julia et en ArchJava, les composants sont implémentés par une classe Java et il est donc possible d'invoquer un service interne via l'envoi de message en utilisant la pseudo-variable `this`. En SCL, nous posons les choix suivants :

Choix 12 *Tout composant possède un port interne nommé* `self` *via lequel tous les services définis dans le composant sont fournis.*

Définition 9 (Port interne) *port (requis ou fourni) d'un composant qui n'est pas accessible en dehors de son implémentation.*

Les choix 12 et la définition 9 permettent de proposer une solution intégrée et uniforme pour l'invocation des services internes. Un port interne n'est utilisable que dans l'implémentation de son composant au même titre que les interfaces internes en Fractal. Un tel port peut aussi être décrit par une interface, supporter des liaisons de ports et des invocations de services. Toutefois, il est invisible et inaccessible pour les autres composants. Tout composant est muni d'un port interne `self` et c'est pourquoi nous l'omettons souvent dans nos représentations graphiques. La figure 3.8 montre un exemple de représentation de ce port interne.

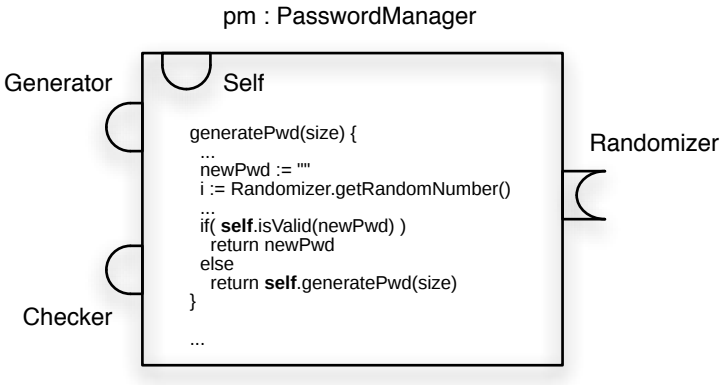

FIG. 3.8 : Représentation du port interne `self` d'un composant

Dans la suite, les ports accessibles de l'extérieur d'un composant sont appelés *ports externes* pareillement aux interfaces externes de Fractal. L'utilisation que nous proposons pour les ports internes requis est détaillée dans la section 3.11.

3.9.2 Les paramètres

Les paramètres de services et le passage des arguments lors de l'invocation de service soulèvent de nombreuses questions : Qu'est-ce qu'un paramètre ? A-t-on réellement besoin de paramètres sachant qu'il est possible d'utiliser la connexion de composants ? Si oui, qu'est-ce qu'un passage d'arguments ? Nous allons montrer que ces questions ne sont pas vraiment traitées dans les langages existants bien qu'elles constituent un enjeu majeur pour l'intégrité des communications et l'uniformité du langage.

Dans les approches existantes, les composants peuvent s'échanger des données diverses via le passage d'arguments et le retour de résultats lors des invocations de services. Une approche consiste à standardiser le format des données échangeables comme c'est le cas dans le modèle des *processus Unix* (*Pipe and Filter*) où les données échangées sont toujours des flux de caractères ou encore dans le modèle des *Services Web* avec le format SOAP spécifiant comment sont structurés les requêtes de services, les données et les réponses. Les langages tels que ArchJava, ComponentJ ou encore Julia reposent entièrement sur le passage d'arguments du langage Java. En Julia, un composant Fractal est implémenté par un ensemble d'objets Java chacun représentant une partie du composant (interface, membrane, contenu) et ayant des responsabilités différentes. Le programmeur a accès à ces différents objets, il peut donc passer ces objets comme arguments lors des envois de messages. Cette possibilité conduit à « la violation de l'intégrité des communications » [Léger *et al.*, 2006] c'est-à-dire que des objets Java représentants deux composants différents peuvent communiquer directement (avec un échange préalable de références) sans passer par les interfaces et les liaisons de leurs composants. Face à ce problème, la solution développée dans [Léger *et al.*, 2006] est un contrôle dynamique des communications entre objets afin de vérifier leur validité par rapport à l'architecture. Nous pensons qu'il est possible de s'affranchir de ce contrôle dynamique en spécifiant clairement dans le langage SCL ce que les composants peuvent échanger et comment ils peuvent le faire dans le respect de l'intégrité des communications. Cela implique que le programmeur SCL ne devra pas avoir accès aux objets sous-jacents à un composants dans le cas d'une implémentation dans un LOO.

Actuellement, aucune opération n'est possible directement sur un composant. Les invocations de service et les connexions s'effectuent via les ports et c'est pourquoi nous avons fait le choix suivant en SCL :

Choix 13 *Les paramètres des invocations de services sont des références sur des ports.*

Par ce choix 13, nous répondons à notre question initiale en fixant la nature des paramètres. Les paramètres étant des références sur des ports, quelle est la différence entre un paramètre et un port requis ? Pour illustrer cette différence, la figure 3.9 présente deux conceptions différentes du descripteur PASSWORDMANAGER. Dans le cas (a), le service `generatePwd` prend un paramètre *size* alors que dans le cas (b), le descripteur `pm2` possède un port `Configurator` à travers lequel il requiert un service `getSize` qu'il invoque dans l'implémentation du service `generatePwd`.

La différence entre les paramètres des services et les ports requis est, hormis la syntaxe, la portée de l'identificateur (*scope*) et sa durée de vie (*extent*). C'est la même différence qu'entre un paramètre et un attribut dans un LOO. La question suivante se pose alors : faut-il garder les paramètres des services en SCL alors qu'il serait possible de n'utiliser que des ports requis ? Utiliser systématiquement

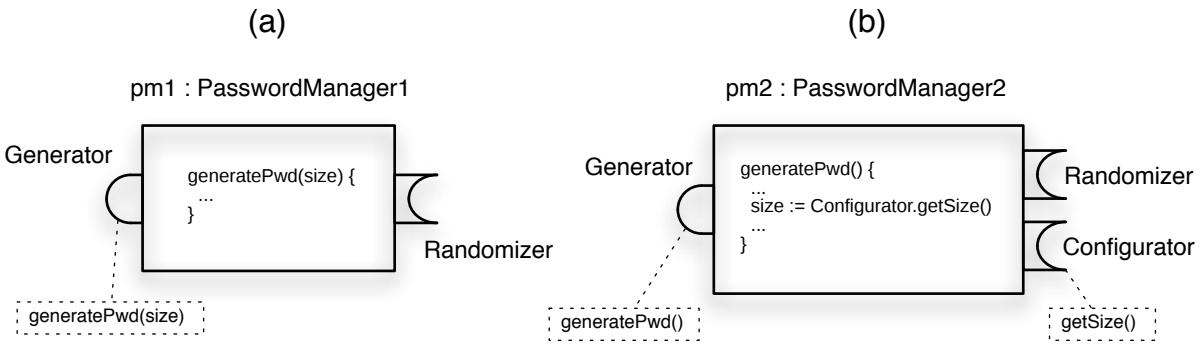

FIG. 3.9 : La dualité des paramètres de service et des ports requis

des ports requis plutôt que des paramètres permet d'assurer l'intégrité des communications puisque le mécanisme d'assemblage est utilisé. En effet, s'il est possible de passer des références sur des ports fournis en tant qu'arguments, l'intégrité des communications pourrait être violée de la même façon qu'en Julia c'est-à-dire en stockant cette référence pour l'utiliser ultérieurement. Toutefois, il n'est pas envisageable de n'utiliser que des ports requis du fait des différences en les paramètres des services et les ports requis. La solution intégrée en SCL est la suivante :

Choix 14 *Le passage d'arguments s'effectue par l'établissement et le retrait automatique de liaisons de ports. Tout composant est doté d'une collection de ports requis nommée* `args`*. Lors d'une invocation de service, les arguments (des ports)* $\langle a_1, a_2, ..., a_n \rangle$ *sont chacun liés respectivement à* $\langle args[1], args[2], ..., args[n] \rangle$*. Les identificateurs des paramètres sont en réalité des alias des identificateurs des ports* `args`*. A la fin de l'exécution du service, toutes les liaisons des ports de* `args` *sont supprimées.*

Nous avons donc choisi de décrire le mécanisme de passage d'arguments de SCL en terme de liaisons de ports. Par ce choix, nous proposons pour SCL un mécanisme uniforme et respectueux de l'intégrité des communications puisqu'il est impossible que les composants communiquent en dehors d'une connexion. La figure 3.10 présente un exemple d'invocation de service utilisant ce mécanisme dont les étapes de traitement sont les suivantes :

1. Emission d'une invocation de service via un port. Dans la figure 3.10, le composant `am` émet, via son port `PwdGenerator`, une invocation de service ayant pour sélecteur `generatePwd` et pour seul argument, une référence sur l'unique port du composant primitif représentant la valeur 6 (cf. choix 7).
2. Le composant destinataire (`pg` dans notre exemple) reçoit et traite l'invocation. Nous reviendrons sur la façon dont un composant traite une invocation de service dans la section 3.12.
3. Les ports passés en argument de l'invocation sont liés aux ports `args` du composant destinataire (cf. figure 3.10).
4. Le service est exécuté. Un mécanisme d'*aliasing* transparent (mis en place par l'évaluateur par exemple) permet au programmeur d'utiliser les noms qu'il a spécifié pour les paramètres

dans l'implémentation et ne lui impose pas d'utiliser directement les ports `args`. Dans notre exemple, le code source du service `generatePwd` du composant `pg` utilise l'identificateur `size` comme nom de paramètre et non `args[1]`.

5. A la fin de l'exécution du service, toutes les liaisons des ports `args` sont supprimées.

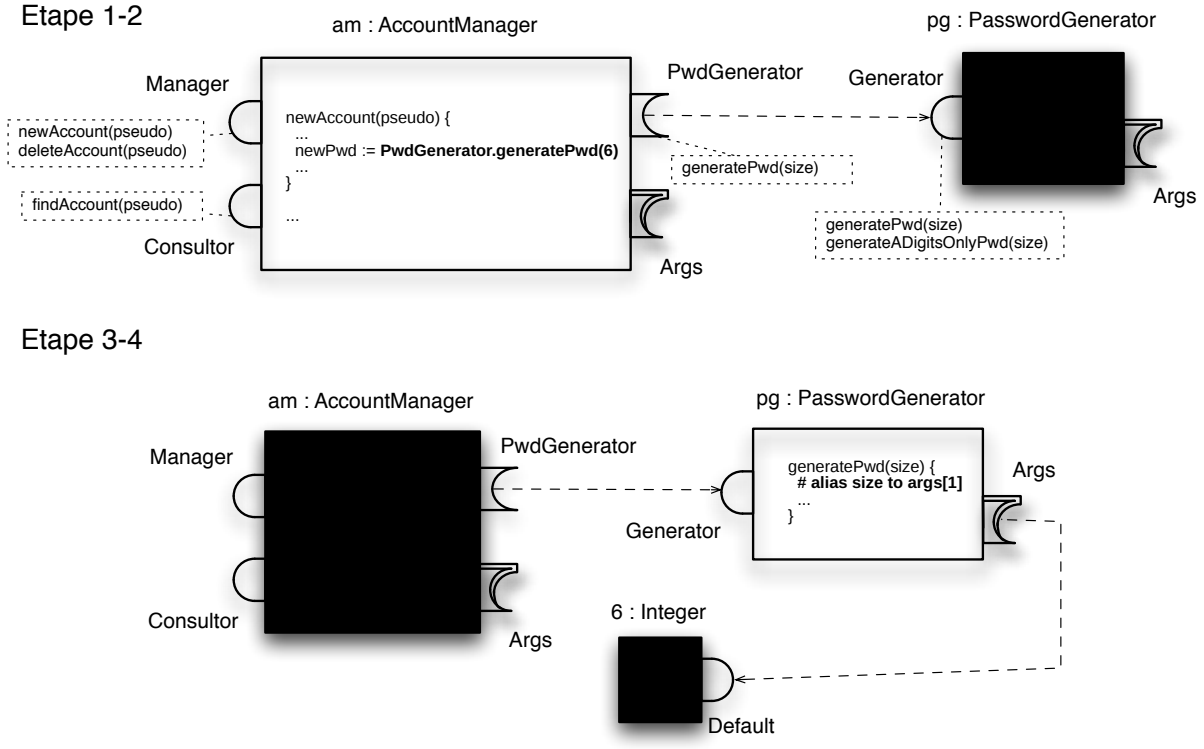

FIG. 3.10 : Illustration du traitement d'une invocation de service en SCL

3.10 Les connexions doivent-elles être réifiées ?

Les connexions entre les composants sont réalisées par des liaisons de ports. Bien que le mécanisme de liaison de ports soit utilisé dans la majorité des modèles actuels, il présente au moins les deux inconvénients suivants : la difficulté à résoudre les incompatibilités et le passage à l'échelle.

Une incompatibilité (*mismatch* en anglais) [Garlan *et al.*, 1995] se produit lorsque la connexion de deux composants a du sens mais qu'elle est impossible à établir à cause de problèmes syntaxiques ou structurels. Ce problème est souvent évoqué sous des noms différents comme la compatibilité des composants du point de vue des branchements (*plug-compatible components*) [Achermann *et al.*, 2001] en Piccola ou encore les conflits d'interfaces (*interface conflicts*) [Fröhlich, 2000] en Lagoona. Ces incompatibilités sont inévitables [Sametinger, 1997] puisque elles sont la conséquence de la non-anticipation des connexions. En effet, un composant n'est pas développé pour être connecté ou pour

coopérer avec un composant donné (un type de composants en particulier) mais avec d'autres composants fournissant les services qu'il requiert. Ces incompatibilités peuvent être levées par l'application du schéma de conception *adapteur* [Gamma *et al.*, 1995] c'est-à-dire la création d'un composant spécifique permettant de faire collaborer des composants qui ne le pourraient pas directement. La réification des connexions sous la forme de *connecteurs* [Shaw, 1996] permet de résoudre plus facilement ces incompatibilités structurelles. Les connecteurs permettent aussi :

- une meilleure séparation entre le code relatif à l'interaction entre les composants et le code métier. Typiquement, le code relatif aux communications distantes est à inclure dans les connecteurs.
- d'associer la sémantique des connexions aux connecteurs et éventuellement de le rendre adaptable au cas par cas.
- réutiliser directement des protocoles de communication complexes.
- capitaliser sur les connecteurs c'est-à-dire de proposer des bibliothèques de connecteurs réutilisables chacun proposant un protocole de communication particulier.

Tous ces avantages nous conduisent à réifier les connexions en SCL.

Choix 15 *Les* connecteurs *sont des entités de première classe (manipulables dans le langage).*

Suite à ce choix 15, la question suivante se pose : qu'est-ce qu'un connecteur ? Il existe deux approches : celles où les connecteurs ne sont pas des composants comme en ArchJava et en SOFA et celles où les connecteurs sont des composants comme en Fractal. En ArchJava [Aldrich *et al.*, 2003], les composants sont décrits par des classes de composants (décrivant les ports) alors que les connecteurs sont décrits par des classes standards utilisant l'API `java.lang.reflect` afin de manipuler et d'adapter si besoin les envois de message échangés par les composants via leurs liaisons de ports. De même, en SOFA les connecteurs ne sont pas des composants bien qu'ils peuvent être construits à partir de composants comme nous l'avons vu dans la section 2.2.7. En Fractal par contre, un connecteur est un composant (*binding component*) standard jouant le rôle d'adapteur pour d'autres composants. En SCL, nous avons choisi cette dernière approche.

Choix 16 *Un* connecteur *est un composant.*

Ce choix 16 est essentiellement motivé par notre volonté d'uniformité et de simplicité. A ce stade, SCL propose une solution identique à celle de Fractal en ce qui concerne les connecteurs qui sont des composants standards jouant un rôle d'adapteur.

Cette solution n'est toutefois pas suffisante comme nous allons le voir à travers l'exemple de la figure 3.11. Dans cet exemple, les composants `pm` et `rng` ne peuvent être connectés directement car le premier requiert un service nommé `getRandomNumer` alors que le second fourni un service nommé `random`. L'utilisation du composant adapteur `pm2rng` permet de résoudre cette incompatibilité. Bien que cette solution fonctionne, nous pensons qu'elle pose plusieurs problèmes. Tout d'abord, le composant `pm2rng` doit obligatoirement fournir un service nommé `getRandomNumer` afin d'être lié au composant `pm`. Ce nom de service imposé est en rapport ni avec la sémantique du code source, ni avec celle du composant adapteur. En effet, cet adapteur ne fournit pas un service générant des nombres

aléatoires mais adapte le service requis *random* afin qu'il puisse être utilisé par le composant `pm`. Cette adaptation est nécessaire car le service requis `getRandomNumber`, utilisé pour générer des mots de passe, doit retourner des nombres aléatoires compris entre 1 et 26 (pour le choix d'une lettre de l'alphabet) alors que le service `random` retourne des flottants entre 0 et 1. Par ailleurs, la question qui se pose est de savoir comment, avec cette solution, il serait possible de définir des adapteurs génériques. Dans cet exemple, le composant adapteur `pm2rng` a été construit de façon *ad hoc* afin de faire collaborer les deux composants `pm` et `rng` et il ne pourrait certainement pas être réutilisé pour adapter d'autres composants. Enfin, cette solution est mal adaptée pour le passage à l'échelle puisque l'écriture d'un adapteur peut s'avérer très fastidieuse dans le cas d'une connexion de plusieurs composants ayant de nombreux services.

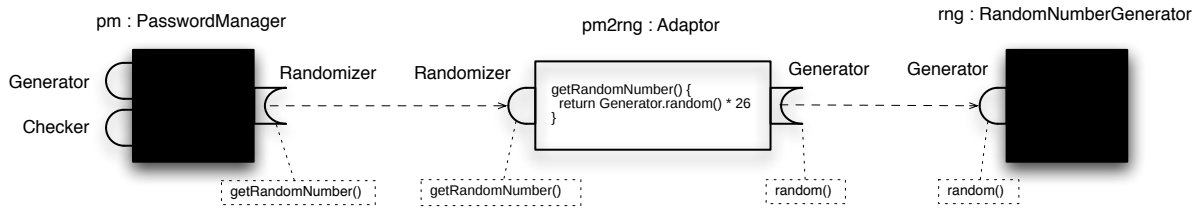

FIG. 3.11 : Utilisation d'un composant adapteur

Face à toutes ces limitations, une solution consiste à utiliser une approche par génération de code. La génération de code est par exemple utilisée dans l'environnement *Javabeans* pour générer automatiquement des adapteurs Dans les cas d'adaptation triviaux, le code généré est complètement masqué à l'architecte. Dans les autres cas (comme notre exemple de la figure 3.11), le code généré n'est qu'un squelette que l'architecte doit compléter. Nous n'avons pas retenu la génération de code pour SCL car elle contredit nos objectifs initiaux de simplicité et de fournir des abstractions et mécanismes de haut niveau. En effet, le code généré peut être complexe, difficile à maintenir et à faire évoluer au fil des évolutions.

Nous proposons une solution novatrice à ce problème en SCL. Cette solution est fondée sur les constats suivants :

- un composant adapteur ne peut pas être utilisé directement comme connecteur à cause de la conception *ad hoc* que cela implique comme nous l'avons montré ci-dessus (*cf.* figure 3.11) ;
- du point de vue de la réutilisation et de la dynamicité, les composants adapteurs ne sont pas très adaptés car ils doivent être conçus spécifiquement c'est-à-dire dotés de ports et de services spécifiques en fonction des composants à connecter ;
- le code des connecteurs relève, du point de vue des communications entre les composants, du méta-niveau puisqu'il manipule les invocations de services et les résultats afin de les adapter.

Le reste de cette section est consacrée à la présentation de notre solution en SCL concernant les connecteurs. Le choix 17 pose les bases de cette solution.

Choix 17 *Tout port fourni `pf` d'un composant `c` peut être associé à un service particulier du composant appelé* service glue *dont l'unique paramètre est une invocation de service. Ce service glue est exécuté lorsqu'une invocation de service `i` est reçue par `c` à travers `pf` et qu'aucun service fourni à travers `pf` ne correspond au sélecteur demandé dans `i`.*

La figure 3.12 et le code source 3.3 illustrent ce choix 17 en reprenant l'exemple précédent de la figure 3.11 et en utilisant un service *glue*. Dans cet exemple, un connecteur binaire est utilisé. Il s'agit d'un composant instance du descripteur MYADHOCBINARYCONNECTOR. Ce composant possède le port fourni `Incoming` à travers lequel il reçoit des invocations de services et le port requis `Outgoing` à travers lequel il émet des invocations de service. Le port `Incoming` ne fournit aucun service et est associé à un service glue. Ce service glue est exécuté pour toutes les invocations de service reçue via le port `Incoming` puisque qu'aucun service n'est fourni via ce port. L'implémentation de ce service glue est dédiée à l'adaptation de composants instances des descripteurs PASSWORDMANAGER et RANDOMNUMBERGENERATOR.

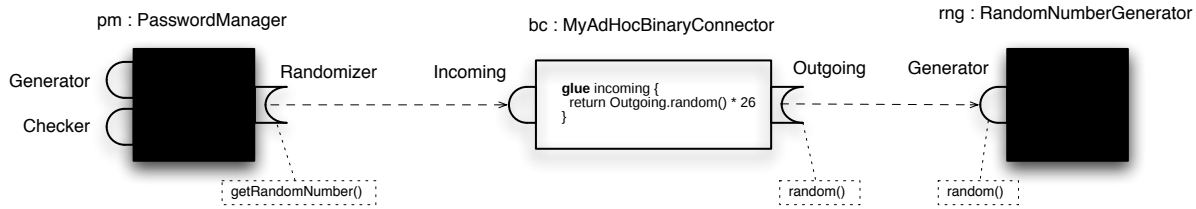

FIG. 3.12 : Utilisation d'un connecteur binaire

```
interface IOutGoing, { random() }

descriptor MyAdHocBinaryConnector {
   providedport Incoming
   requiredport Outgoing, IOutGoing

   glue Incoming {
      # la pseudo variable « si » (service invocation) permet au programmeur
      # d'accéder à l'invocation de service courante dans un service glue
      return Outgoing.random() * 26
   }
}

pm := new PasswordManager
rng := new RandomNumberGenerator
bc := new MyAdHocBinaryConnector

bind pm.Randomizer to bc.Incoming
bind bc.Outgoing to rng.Generator
```

LISTING 3.3 : Définition et utilisation d'un connecteur binaire

En SCL, il est donc possible d'attacher un service *glue* à un port fourni d'un composant (primitive **glue**) qui est exécuté si le composant ne possède pas le sélecteur demandé. Cette notion de « service glue » est fortement inspiré de la méthode `doesnotunderstand:` en Smalltalk qui a d'ailleurs été reprise dans de nombreux langages à objets comme Ruby notamment. Cette possibilité d'attacher un service glue à un port fourni n'est pas spécifique aux connecteurs en SCL mais En SCL, la frontière

entre composant et connecteur n'est pas structurelle puisque les connecteurs sont des composants (cf. choix 16) mais vraiment sémantique c'est-à-dire que c'est le rôle joué par un composant dans une architecture qui détermine s'il est un connecteur.

Dans l'exemple précédent, nous avons défini le descripteur MyAdHocBinaryConnector pour établir une connexion binaire spécifique aux composants à adapter. Ainsi, cette solution à base de « services glue », ne permet toujours pas la définition de connecteurs génériques et réutilisables. Dans l'exemple précédent, seul le service glue est vraiment dépendant des composants PasswordManager et RandomNumberGenerator. Ainsi, nous avons définit des connecteurs réutilisables en Scl tout en laissant la possibilité à l'architecte de fixer un code glue particulier. Le code source 3.4 illustre cela en présentant l'instruction **bind** ...**to** ...**glue** ... qui créé une instance du connecteur générique BinaryConnector et la paramètre par le code glue spécifié par l'architecte. Ce code glue peut accéder à l'invocation de service courante via la pseudo-variable **si** ainsi qu'aux ports du connecteur dans lequel il sera exécuté. Dans le cas d'un BinaryConnector par exemple, le programmeur peut faire référence aux ports `Incoming` et `Outgoing`.

```
pm := new PasswordManager
rng := new RandomNumberGenerator

# Par défaut, un BinaryConnector est automatiquement créé pour lier deux ports
# lorsque du code glue est spécifié. Dans le code glue, le programmeur peut accéder à :
# - l'invocation courante vi la pseudo variable si
# - aux ports du connecteur dans lequel va s'exécuter le code glue (e.g Outgoing)
bind pm.Randomizer to rng.Generator glue {
   return Outgoing.random() * 26
}

# Un connecteur particulier peu être spécifié par le programmeur
bind c1.r1 to c2.p2 connector new TCPConnector

# Utilisation d'un connecteur spécifique et de code glue
bind c1.r1 to c2.p2 connector new TCPConnector glue {
   return Outgoing.random() * 26
}
```

Listing 3.4 : Utilisation de connecteurs prédéfinis et exemples de facilités syntaxiques

Comme nous venons de le voir, il est possible de construire des connecteurs génériques en Scl pour lesquels le code glue sera spécifié par l'architecte. Jusqu'ici, nous nous sommes essentiellement concentré sur des connexions binaires (à deux composants) et donc sur le cas du connecteur BinaryConnector. Toutefois, l'établissement de connexions n-aire (plusieurs composants) nécessite aussi des connecteurs et des constructions syntaxiques adaptées. Nous proposons ainsi une forme générale de connecteurs en Scl. Cette forme générale est décrite par le descripteur Connector dont un exemple est donné sur la figure 3.13 et dont la spécification est posée par le choix 18.

Choix 18 *Un connecteur est constitué de deux collections de ports nommées `Adaptees` et `Targets`. Un connecteur est en charge de traiter les invocations de services qu'il reçoit à travers ses ports `Adaptees` en*

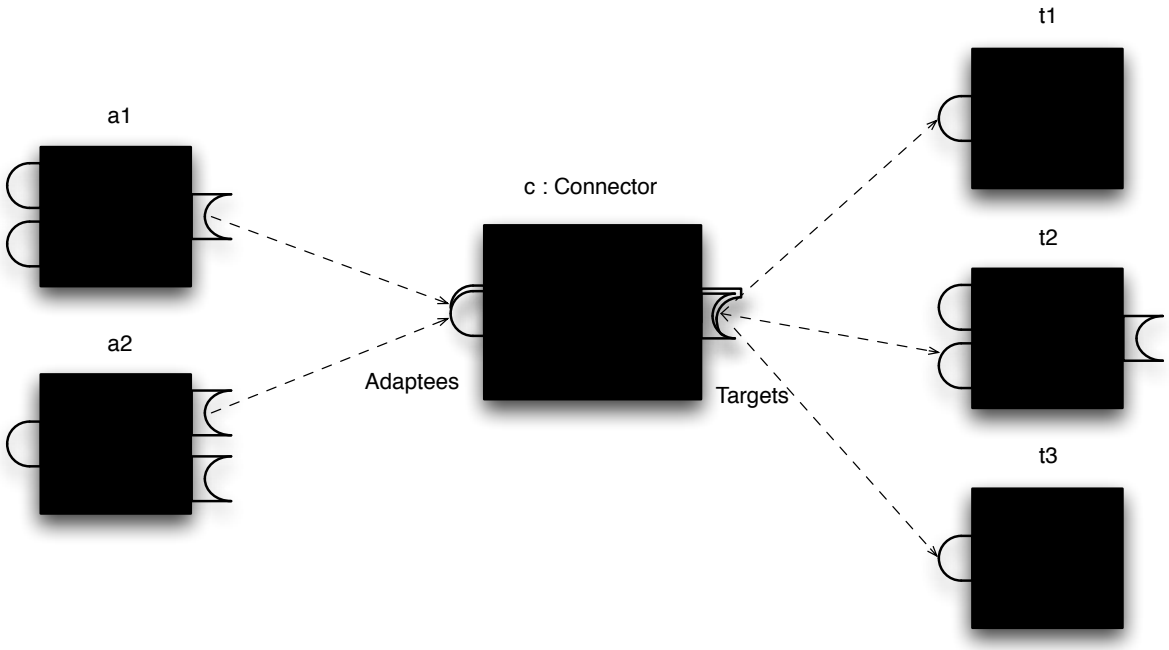

FIG. 3.13 : Forme générale d'un connecteur en SCL

invoquant si nécessaire des services à travers ses ports `Targets`.

En SCL, un connecteur joue le rôle d'adapteur [Gamma *et al.*, 1995] entre les composants connectés à ses ports `Adaptees` et les composants connectés à ses ports `Targets`. On peut remarquer que le connecteur binaire BINARYCONNECTOR n'est qu'une restriction de cette forme générale n'ayant qu'un seul port `Adaptees` nommé `Incoming` et qu'un seul port `Targets` nommé `Outgoing`. De même que pour le cas binaire, du *sucre syntaxique* permet de s'affranchir de la définition des descripteurs de connecteurs dans les cas simples comme le montre le code source 3.5 sur quelques exemples. La primitive **`bind`**...**`to`**...**`glue`**... a été enrichie pour s'adapter au cas n-aire. Ainsi, lorsque du code glue est spécifié, il est associé à tous les ports `Adaptees` du connecteur sauf si un port est explicitement spécifié.

3.11 De l'intérêt des composants composites ?

Dans les approches à composants, un composant est dit *composite* s'il est constitué de composants appelés *sous-composants* (ou *composants internes*). Ce concept est à la base du mécanisme de *composition* (comparable à la relation de composition entre classes en UML).

Tout d'abord : *Pourquoi a-t-on besoin de composites ?* Pour répondre à cette question, rappelons que l'expression des services requis d'un composant permet son découplage et augmente ainsi son potentiel de réutilisation. Cette technique d'extraction des dépendances du code source sous la forme

```
# Par défaut, un Connector est automatiquement créé pour lier plus de deux ports
# lorsque du code glue est spécifié.
bind a1.r1, a2.r1
to t1.p1, t2.p2, t3.p1
glue {
   for i in 1..3
      targets[i].perform(si)
}

# un Connector spécifique peut être choisi.
bind a1.r1, a2.r1
to t1.p1, t2.p2, t3.p1
connector new Broadcaster

# Le code glue peut être associé à un port donné du connecteur.
bind a1.r1, a2.r1
to t1.p1, t2.p2, t3.p1
glue Adaptees[1] {
   # glue code
}
glue Adaptees[2] {
   # glue code
}
```

LISTING 3.5 : Exemples de constructions syntaxiques facilitant l'utilisation des connecteurs

de services requis est parfois appelée *factoring out* [Seco et Caires, 2000]. Dans l'exemple de la figure 3.3, un PASSWORDGENERATOR peut être utilisé avec n'importe quel autre composant fournissant un service de génération de nombres aléatoires. Bien que cette technique de *factoring out* permet un meilleur découplage, elle pose aussi des problèmes de passage à l'échelle et de réutilisation. En effet, il est actuellement impossible de réutiliser directement un assemblage de composants, par exemple un PASSWORDGENERATOR connecté à un RANDOMNUMBERGENERATOR comme présenté sur la figure 3.12. Cela signifie que pour chaque application dans laquelle l'architecte veut intégrer un PASSWORDGENERATOR, il devra intégrer un RANDOMNUMBERGENERATOR et écrire à nouveau le code glue nécessaire à la connexion de ces deux composants. Les composites sont notamment une réponse à ces besoins, à savoir :

- encapsuler plusieurs composants et leur assemblage afin de masquer certains détails dans une architecture logicielle,
- réutiliser directement des assemblages de composants.

Nous pensons qu'un langage à composants doit permettre la manipulation (définition, réutilisation, remplacement, maintenance, etc.) d'assemblage de composants. Pour cela, les ADLs ont été les précurseurs en proposant le concept de *configuration*. Dans l'ADL WRIGHT par exemple, une *configuration* représente un ensemble de composants connectés via des connecteurs. Contrairement à une configuration, un composite est un composant. Les modèles tels que Fractal et ArchJava proposent le concept de *composite* pour représenter des assemblages de composants. Ces modèles sont

dits *hiérarchiques* car un composite peut être décomposé en un assemblage de *sous-composants* interconnectés ; chaque sous-composants pouvant être un composite ou un composant de base. Nous avons choisi d'intégrer une approche similaire à celle des modèles hiérarchique en SCL.

Choix 19 *Un* composite *est un composant possédant un ou plusieurs sous-composants.*

Considérer les composites comme des composants (*cf.* 19) permet :
- de les mettre sur étagère et de les réutiliser ;
- de les doter de ports requis permettant ainsi d'encapsuler des architectures « partielles » configurables via le mécanisme d'assemblage ;
- de rendre les architectures complexes plus compréhensibles puisqu'elles peuvent être examinées à différents niveaux de granularité suivant que l'on détaille ou non le contenu d'un composite.

Avant d'aller plus loin, il nous semble nécessaire de présenter une difficulté — complètement oubliée dans les approches existantes — concernant la conception des composites qui nécessite de ne pas appliquer le principe de *factoring out*. En effet, lorsqu'un programmeur développe un composite, il choisit et fixe les sous-composants qu'il utilise. Cela induit nécessairement un couplage entre un composite et ses sous-composants. La conception d'un composite nécessite donc de faire un choix entre ce qui doit être externalisé (via des ports requis) afin de garder un fort potentiel de réutilisation et ce qui doit être internalisé en utilisant des sous-composants afin de masquer les détails de fonctionnement. L'utilisation systématique de composites sans externalisation des besoins ne permet pas de définir des composants réutilisables. Il est toutefois impossible d'empêcher cela dans un langage à composants. On peut toutefois limiter cet effet néfaste de la composition en imposant que chaque descripteur soit définit séparément (*cf.* choix 20).

Choix 20 *Un descripteur de composite ne contient pas les descripteurs de ses sous-composants mais il y fait référence.*

Par ce choix 20, nous interdisons l'imbrication des descripteurs en SCL (si l'on fait un parallèle avec Java, ce choix interdirait les *inner* classes) . Notre objectif est que tous les descripteurs de composants (même ceux des sous-composants) soit mis sur étagère afin d'être réutilisés. Ce choix est motivé par le fait qu'un composant ne doit pas uniquement être conçu comme un sous-composant encapsulé dans un composite donné mais comme un composant mis sur étagère et éventuellement utilisé en tant que sous-composant.

En SCL comme dans la plupart des approches à composants, un composite contrôle le cycle de vie (création/destruction) et l'assemblage de ses sous-composants. Les sous-composants sont privés à leur composite et inaccessibles (voire invisibles) en dehors de son implémentation. La figure 3.14 montre un exemple de composite et le code source 3.6 montre le code du descripteur de ce composite en SCL. Le service `init` d'un composant permet son initialisation et est donc utilisé ici pour instancier les sous-composants. La ligne 10 du code source 3.6 montre l'établissement d'une connexion entre les deux sous-composants `pg` et `rng`. La ligne 15 montre aussi une liaison de ports entre le port fourni `Generator` du composite et un port fourni de son sous-composant `pg`. Nous appelons ces liaisons

entre ports de même nature des *liaisons de délégation* (*cf.* définition 10). Syntaxiquement, la même primitive **bind** ...**to** ... est utilisée pour les liaisons et les délégations car il n'y a pas d'ambiguïté.

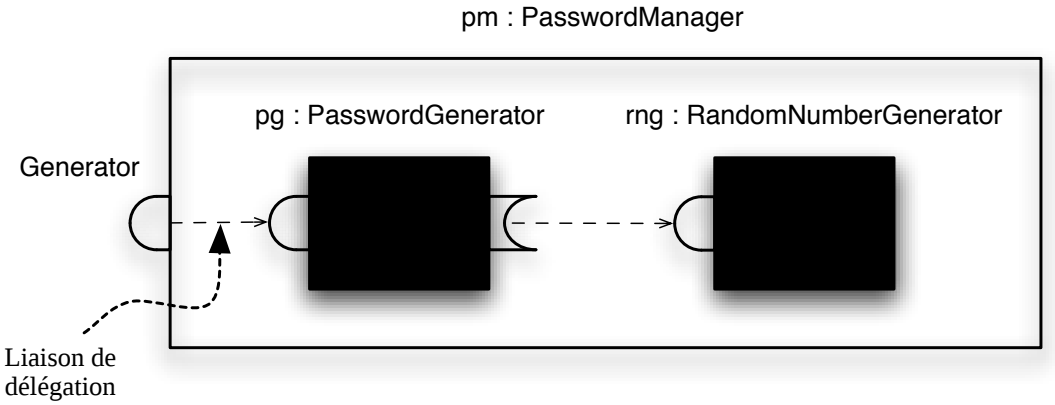

FIG. 3.14 : Utilisation d'un composite pour encapsuler un assemblage de composants

```
1  descriptor CDPasswordManager {
       providedport Generator
3
       def init {
5          # intanciation des sous-composants
           pg := new PasswordGenerator
7          rng := new RandomNumberGenerator

9          # connexion des sous-composants
           bind pg.Randomizer to rng.Generator glue {
11             # glue code
           }
13
           # liaison de délégation entre deux ports fournis
15         bind Generator to pg.Generator
       }
17 }
```

LISTING 3.6 : Descripteur d'un composite

Définition 10 (Liaison de délégation) *Liaison orientée entre deux ports de même nature (requis ou fourni). Les invocations de services reçues par le port origine d'une telle liaison sont transmises sans modification au port cible de la liaison.*

Concernant la gestion interne des composites, les questions suivantes se posent :

– Est-il possible, et si oui comment, de référencer un sous-composant dans un composite ?

– Comment invoquer les services des sous-composants dans l'implémentation des services du composite ?

Dans le code source 3.6, les sous-composants sont instanciés dans le service `init` du composite. Rappelons que l'instanciation d'un descripteur retourne une référence vers le port nommé `default` du composant instancié (*cf.* choix 8). Les variables `pg` et `rng` contiennent donc des références sur des ports et sont locales au service `init`. Pour pouvoir faire référence aux ports des sous-composants dans le code des services du composite, il faudrait stocker ces références dans des « attributs ». Nous allons en SCL utiliser des ports internes requis (cf. choix 9) à cet effet. La figure 3.15 montre un composite doté du port requis interne `Randomizer`. Ce port requis interne est lié à un port fourni du sous-composant `rng`. Dans le code du service `generateACharsOnlyPwd`, le service `random` est invoqué via ce port interne requis `Randomizer`. Cette invocation de service est tout à fait standard et obéit aux mêmes règles que celles décrites dans la section 3.9. En résumé, les sous-composants peuvent être référencés via des ports internes requis et les services des sous-composants peuvent être invoqués via le mécanisme standard d'invocation de service. Outre l'uniformité, l'avantage de cette solution est la possibilité d'utiliser des connecteurs entre un composite et ses sous-composants afin d'adapter les sous-composants aux besoins du composite à l'aide de code glue.

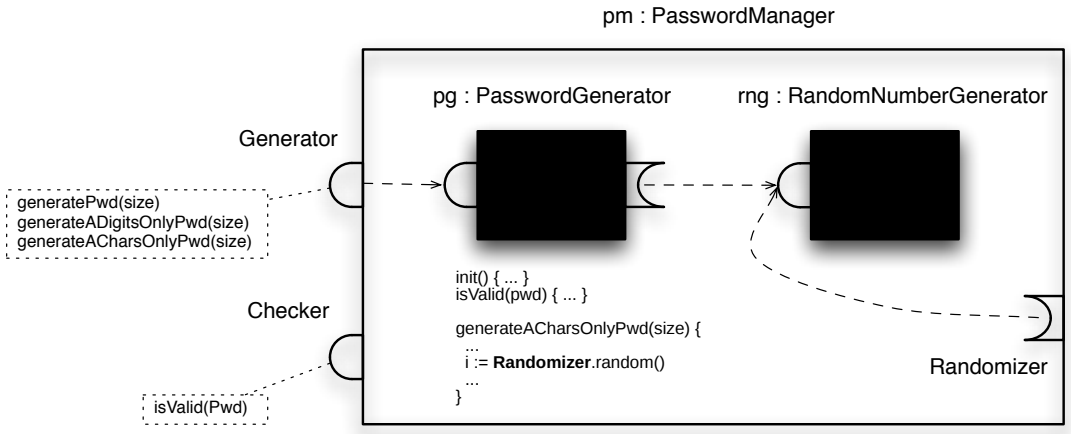

FIG. 3.15 : Exemple de composite possédant un port interne requis

Dans cet exemple présenté à la figure 3.15, le composite exporte trois services. Seul le service `generateACharsOnlyPwd` est défini dans le code du composite. Les deux autres services sont fournis par le sous-composant `pg`. Nous expliciterons cela dans la section suivante qui traite en détail l'invocation de service en tenant compte notamment des liaisons de délégation.

Pour finir, nous souhaitons faire remarquer qu'en SCL les composites ne sont vraiment qu'un moyen de coupler des composants. La figure 3.16 montre une architecture utilisant les mêmes trois composants que dans figure 3.15 mais sans composite. Le composant `pm` de cette architecture n'est pas un composite car les composants `pg` et `rng` peuvent être utilisés par d'autres composants de l'architecture (ils ne sont pas internes à `pm`). On peut aussi remarquer que la délégation de port permet de s'affranchir d'un port requis normalement nécessaire (le port `RG` est présent dans la figure 3.16 alors qu'il n'existe pas dans la figure 3.15).

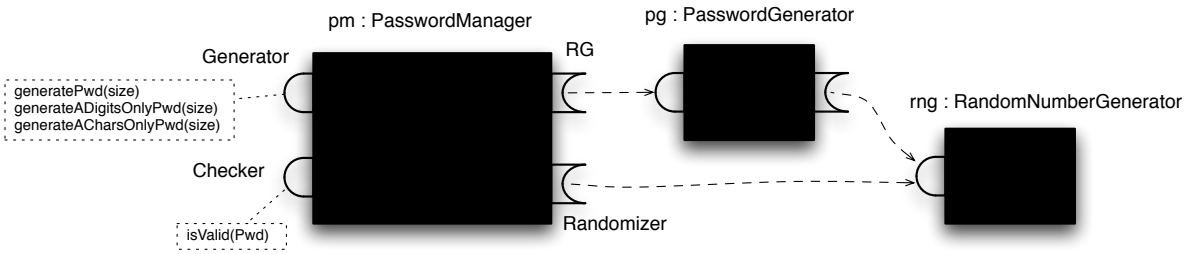

FIG. 3.16 : Exemple d'architecture équivalente sans composite

3.12 L'invocation de services

Contrairement aux langages tels que Julia, ArchJava ou encore ComponentJ, SCL propose un mécanisme d'invocation de service (*cf.* section 3.9) qui n'est pas l'envoi de message car problématique pour faire de la PPC [Beugnard, 2005] et surtout qui tient compte de l'assemblage des composants. Dans cette section nous définissons de façon plus précise (que dans la section 3.9) ce mécanisme central d'invocation de service en tenant compte notamment de l'introduction des connecteurs et des liaisons de délégation en SCL. Deux cas sont distingués :

- l'émission d'une invocation de service via un port requis,
- la réception d'une invocation de service via un port fourni.

L'algorithme 1 présente le traitement d'une émission d'invocation de service via un port requis. Cet algorithme comprend trois cas. Le premier (*cf.* figure 3.17 cas 1) correspond à une liaison standard entre un port requis et un port fourni. L'invocation est dans ce cas transmise au port fourni lié et traitée selon l'algorithme 2 (décrit ci-après). Le second cas (*cf.* figure 3.17 cas 2) correspond à une liaison de délégation. L'invocation est alors transmise au port requis de délégation qui traite l'invocation selon ce même algorithme 1. Le troisième et dernier cas correspond à une dépendance non satisfaite puisque le port requis d'émission ne fait l'objet d'aucune liaison. Le traitement de ce cas d'erreur devrait se traduire par la levée d'une exception. Toutefois, cela sort du cadre de cette thèse car nécessiterait d'introduire en SCL un système de gestion d'exceptions adapté à la programmation par composants comme cela a été étudié dans [Souchon, 2005].

Algorithme 1 : Emission d'une invocation de service via un port requis

Données :
  i : une invocation de service
  r_1 : le port requis à travers lequel i est émise
if r_1 *est lié à un port fourni* p_2 **then** `/* cas 1 */`
  transmettre i via p_2
else
  if r_1 *est lié par délégation à un port requis* r_2 **then** `/* cas 2 */`
    déléguer i via r_2
  else `/* r1 non connecté */`
    cas d'erreur

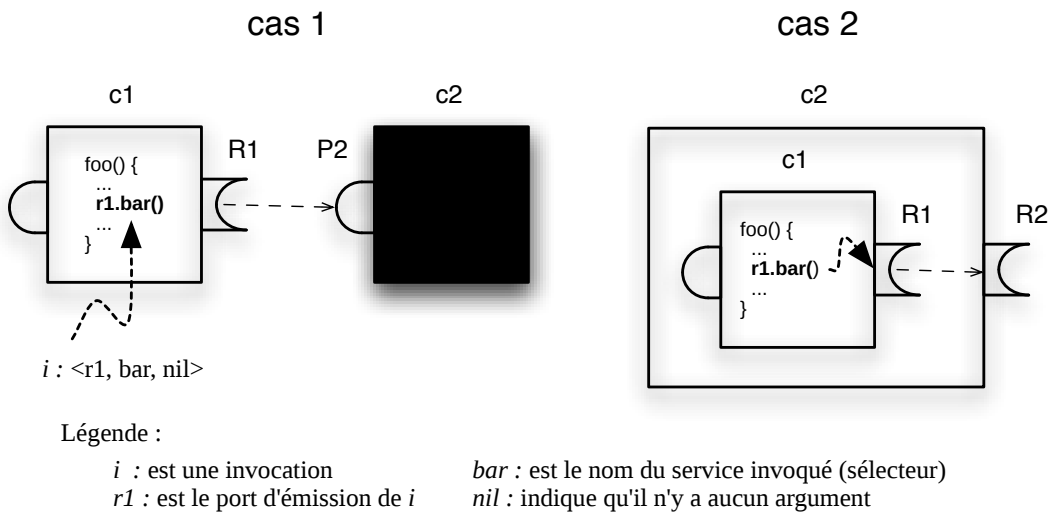

FIG. 3.17 : Les principaux cas lors de l'émission d'une invocation de service à travers un port requis

L'algorithme 2 présente le traitement de la réception d'une invocation via un port fourni. Cet algorithme distingue quatre cas. Le premier (*cf.* figure 3.18 cas 1) consiste à délégué l'invocation de service reçue à un autre port fourni qui traite à son tour l'invocation selon ce même algorithme 3.17. Dans le second cas (*cf.* figure 3.18 cas 2), le service demandé est exécuté puisque défini par le composant auquel appartient le port de réception. Dans le troisième cas (*cf.* figure 3.18 cas 2), c'est le service *glue* du port de réception qui est exécuté car le composant ne définit pas le service demandé. Le dernier cas est un cas d'erreur où le port de réception n'est pas lié, n'est pas délégué et ne possède pas de service *glue* associé.

Algorithme 2 : Réception d'une invocation de service via un port fourni

Données :
i : une invocation de service
p_1 : le port fourni à travers lequel i est reçue

```
if p1 est lié par délégation à un port fourni p2 then                    /* cas 1 */
    déléguer i via p2
else
    if composant(p1) possède le service s demandé par i then             /* cas 2 */
        exécuter s
    else
        if p1 est associé à un service glue g then                       /* cas 3 */
            exécuter g
        else                          /* p1 non connecté et sans code glue associé */
            cas d'erreur
```

Ces algorithmes ont été conçus pour prendre en compte des cas problématiques ou nécessitant une attention particulière tels que ceux présentés sur la figure 3.19.

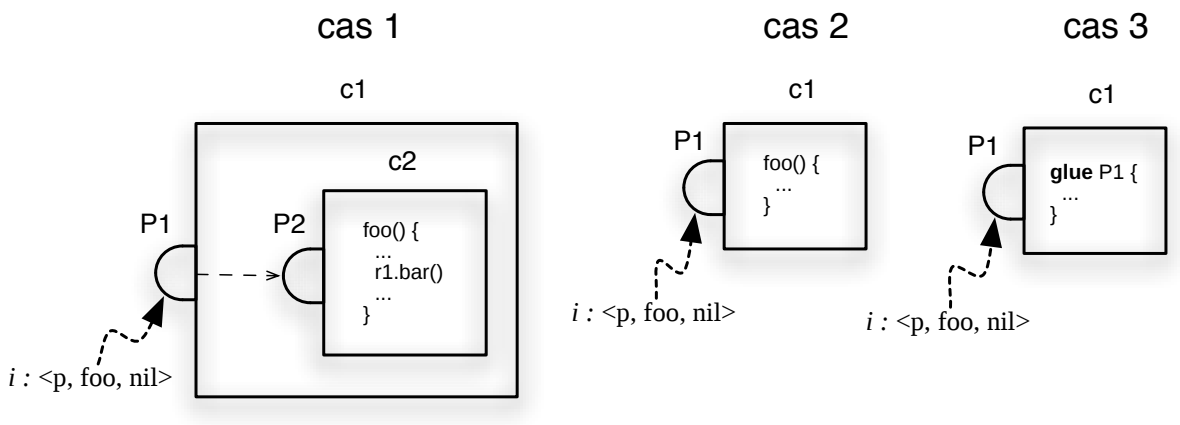

FIG. 3.18 : Les principaux cas lors de la réception d'une invocation de service à travers un port fourni

Dans le cas (a), une invocation de service *i* est reçue via le port `p1` du composite `c1`. Cette invocation est traité selon l'algorithme 2 aboutissant à l'exécution du service `foo` définit dans le sous-composant `c2` à cause de la liaison de délégation entre les ports `p1` et `p2`. Le service `foo` du composite `c1` n'est pas exécuté car nous avons choisit de rendre prioritaire la liaison de délégation par rapport à la présence du service dans le composite. Ce choix permet d'éviter les problèmes de « masquage » qui surviendraient avec l'autre alternative (priorité au service défini dans le composite par rapport à la liaison de délégation) si le composite `c1` possédait plusieurs sous-composants fournissant le service `foo`. En effet, notre solution laisse la possibilité à l'architecte de décider quel composant doit traiter l'invocation via mise en place ou non de liaisons de délégation. Dans notre exemple, si on enlève la liaison de délégation entre les ports `p1` et `p2`, la même invocation aboutirai à l'exécution du service `foo` du composite `c1`.

Dans le cas (b), une invocation de service est émise via le port `r1` du sous-composant `c1` dans le code du service `foo`. Le traitement de cette invocation par l'algorithme 1 abouti au cas d'erreur. Nous n'avons en effet pas intégré de règle permettant d'exécuter automatiquement le service `bar` du composite. Une telle règle serait se problématique pour fournir des services différents à deux sous-composants qui nécessitant un service nommé `bar`. Dans notre exemple, pour l'invocation du service `bar` via `r1` dans `c1` aboutisse à l'exécution du service `bar` du composite `c2`, l'architecte doit établir une liaison entre le port `r1` du sous-composant `c1` et le port `self` du composite `c2`.

En définitive, nous n'avons intégré en SCL aucune règle implicite en ce qui concerne l'invocation de service. Les liaisons étant explicitement mise en place par l'architecte, nous les considérons comme prioritaires.

3.13 Faut-il de l'héritage ?

Dans le monde des objets, la réutilisation repose essentiellement sur les classes et l'héritage même si de plus en plus de travaux proposent de nouvelles formes de réutilisation [Kiczales *et*

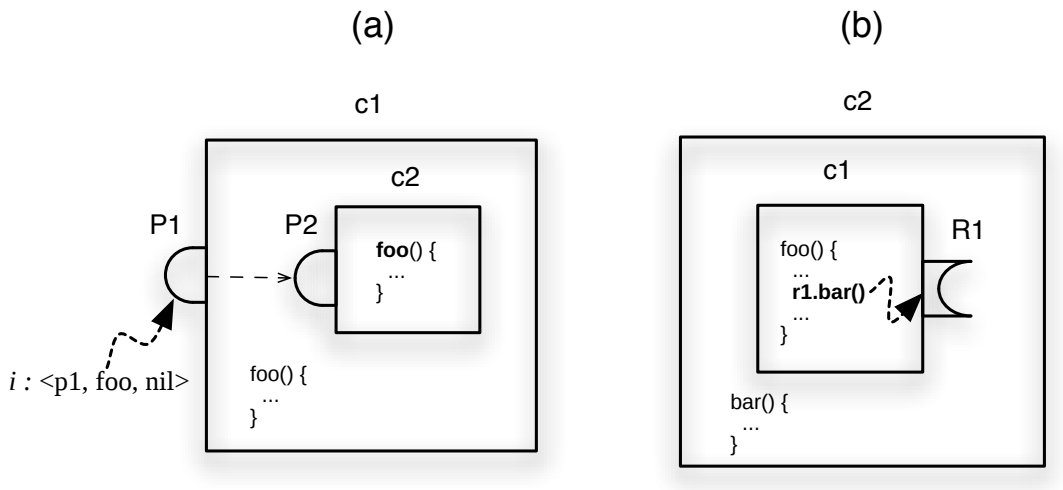

FIG. 3.19 : Cas particuliers d'invocations de service

al., 2001; Lahire et Quintian, 2006]. Bien que le mécanisme d'héritage soit universellement utilisé dans les langages à objets, il pose également de nombreux problèmes [Taenzer *et al.*, 1989; Hürsch, 1994]. Dans cette section, nous discutons les questions suivantes :

- Un mécanisme d'héritage est-il intéressant le monde des composants ?
- Si oui, quelle forme doit prendre ce mécanisme ?

L'héritage est un mécanisme encore largement discuté dans le monde des composants [Weck et Szyperski, 1996; Szyperski, 2002; Opluštil, 2003; Lahire *et al.*, 2004]. En effet, ce mécanisme semble poser des problèmes allant l'encontre des principes de l'approche à composants (découplage et encapsulation). Avant d'expliquer ces problèmes, rappelons que dans les LOO, une classe a deux catégories de clients :

- ceux qui l'instancient (*instantiating clients*),
- ceux qui en héritent (*inheriting clients*).

Ce sont les clients de la deuxième catégorie qui sont à la fois problématiques et puissants. Puissants car ils permettent :

- de faire de la description différentielle via l'ajout de propriétés (attributs et méthodes),
- de redéfinir des méthodes,
- de paramétrer des *framework* en héritant des classes (éventuellement abstraites) le constituant.

Ces clients sont également problématiques car une sous-classe est un client privilégié de sa super-classe pouvant accéder à des détails internes « cassant » ainsi l'encapsulation de la super-classe : « *Inheritance breaks encapsulation* » [Snyder, 1987]. L'héritage est en effet un mécanisme de réutilisation « boîte blanche », contrairement à la composition d'objets qui est un mécanisme de réutilisation « boîte noire », dont l'une des conséquences identifiée est le problème de la classe de base fragile [Mikhajlov et Sekerinski, 1998]. La figure 3.20 illustre ce problème. Supposons l'existence d'une classe `Set` dont une instance représente un ensemble. Cette classe possède les méthodes : `add` pour ajouter un élément et `addAll` pour ajouter tous les éléments contenus dans un autre ensemble. Supposons

que l'on veuille écrire une classe `CountingSet` en héritant de la classe `Set` et en ajoutant le code permettant à un ensemble de maintenir le nombre d'éléments qu'il contient. Pour cela, le programmeur de la classe `CountingSet` doit connaître l'implémentation des deux méthodes `add` et `addAll` de la classe `Set`. En effet, si la méthode `add` est « appellée » dans le code de la méthode `addAll`, la classe `CountingSet` ne doit redéfinir que la méthode `add` (*cf.* figure 3.20 cas (a)). Par contre, si l'implémentation de la méthode `addAll` n'utilise pas la méthode `add` et ajoute directement les éléments, la classe `CountingSet` doit redéfinir les deux méthodes `add` et `addAll` (*cf.* figure 3.20 cas (b)). Cet exemple montre que dans de nombreux cas, l'implémentation d'une super-classe doit être connue pour réaliser une sous-classe.

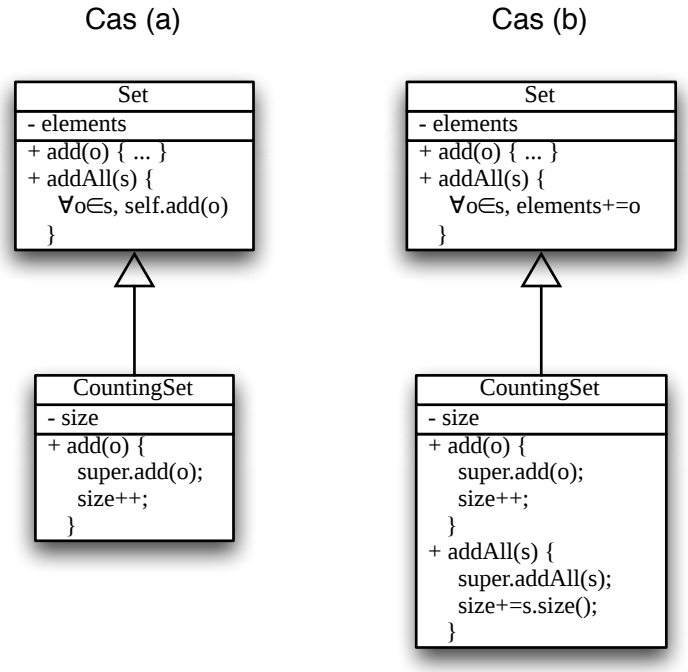

FIG. 3.20 : Illustration du problème de la classe de base fragile

L'héritage est donc en contradiction avec les principes de l'approche à composants sur au moins les deux points suivants :

- il impose d'avoir accès code source (au moins une description fine de l'implémentation),
- il induit un couplage implicite entre une classe et toutes ses super-classes (directes et indirectes) puisque la modification d'une classe modifie l'ensemble de ses sous-classes.

En conséquence, nous pensons que le mécanisme d'héritage entre classes ne peut pas s'appliquer directement aux descripteurs de composant à cause de son caractère intrusif. Toutefois, il nous semble important d'offrir les mêmes possibilités dans un langage à composants que celles offertes par l'héritage dans un LOO à savoir : la description différentielle, la redéfinition et la réutilisation par paramétrage. Pour cela, il existe deux catégories d'approches : celles au niveau des descripteurs et celles au niveau des composants.

Parmi les langages proposant un mécanisme d'héritage au niveau des descripteurs, nous pouvons citer :

- UML (*cf.* section 2.2.9) puisque qu'au niveau du méta-modèle UML, la classe `Component` hérite de la classe `Class`, conférant ainsi la possibilité à un `Component` d'hériter d'un autre `Component` ou même d'une `Class`.
- ArchJava (cf section 2.2.8) permet à une classe de composant d'hériter soit une classe Java standard, soit une autre classe de composant.

Dans ces deux propositions, le mécanisme d'héritage standard entre les classes est appliqué aux descripteurs de composant ce qui nous parait être un mauvais choix pour les raisons que nous détaillées ci-dessus. En SCL, nous n'avons pas définit de mécanisme d'héritage au niveau des descripteurs de composants. Nous pensons que cela relève de la problématique de proposer un mécanisme d'assemblage au niveau des descripteurs. Nous avons déjà évoqué cette problématique dans la section 3.3 (discussion autour du choix 3) en précisant qu'il serait intéressant de proposer une version réflexive de SCL afin d'utiliser les mêmes mécanismes d'assemblages au niveau des composants et au niveau des descripteurs.

Comme cela a été souligné dans le monde des objets, il est possible d'utiliser la délégation pour faire de l'héritage [Stein, 1987]. Dans le monde des composants, les composites (comparables à des objets composites) peuvent donc être utilisés pour de faire de la définition incrémentale, la redéfinition de services et le paramétrage. C'est d'ailleurs ce que proposent, dans une certaine mesure, les auteurs de ComponentJ [Seco et Caires, 2000]. La figure 3.21 illustre ces possibilités en SCL :

Cas 1 Construire une sous-classe ajoutant de nouvelles propriétés peut se réaliser en créant un composite qui exporte l'ensemble des fonctionnalités de ses sous-composants (*cf.* cas 1 de la figure 3.21).

Cas 2 L'héritage est monotone en POO (ajout de propriétés uniquement) alors qu'un composite peut masquer des fonctionnalités offertes par ses sous-composants. Cela provient du fait qu'un composite n'est pas forcément un sous-type de tous ses sous-composants.

Cas 3 Une classe avec des méthodes abstraite peut s'apparenter à un composant ayant des services requis (*cf.* cas 2 de la figure 3.21).

Cas 4 La redéfinition d'une méthode dans une sous-classe peut être en partie réalisé par la création d'un composite. Attention, les invocations dans les sous-composants ne profitent pas de la redéfinition comme c'est le cas pour les super-classes (*cf.* cas 3 de la figure 3.21). On aurait pu mettre en place ce mécanisme en SCL via des liens de délégation implicites entre les ports `self` des sous-composants et le port `self` du composite (*cf.* section 3.12). Toutefois, cela poserait le « problème de sous-composants fragiles » de façon analogue à celui de la classe de base fragile. Si le concepteur d'un composant veut laisser la possibilité de redéfinir un service, il doit le faire explicitement via l'utilisation d'un service requis.

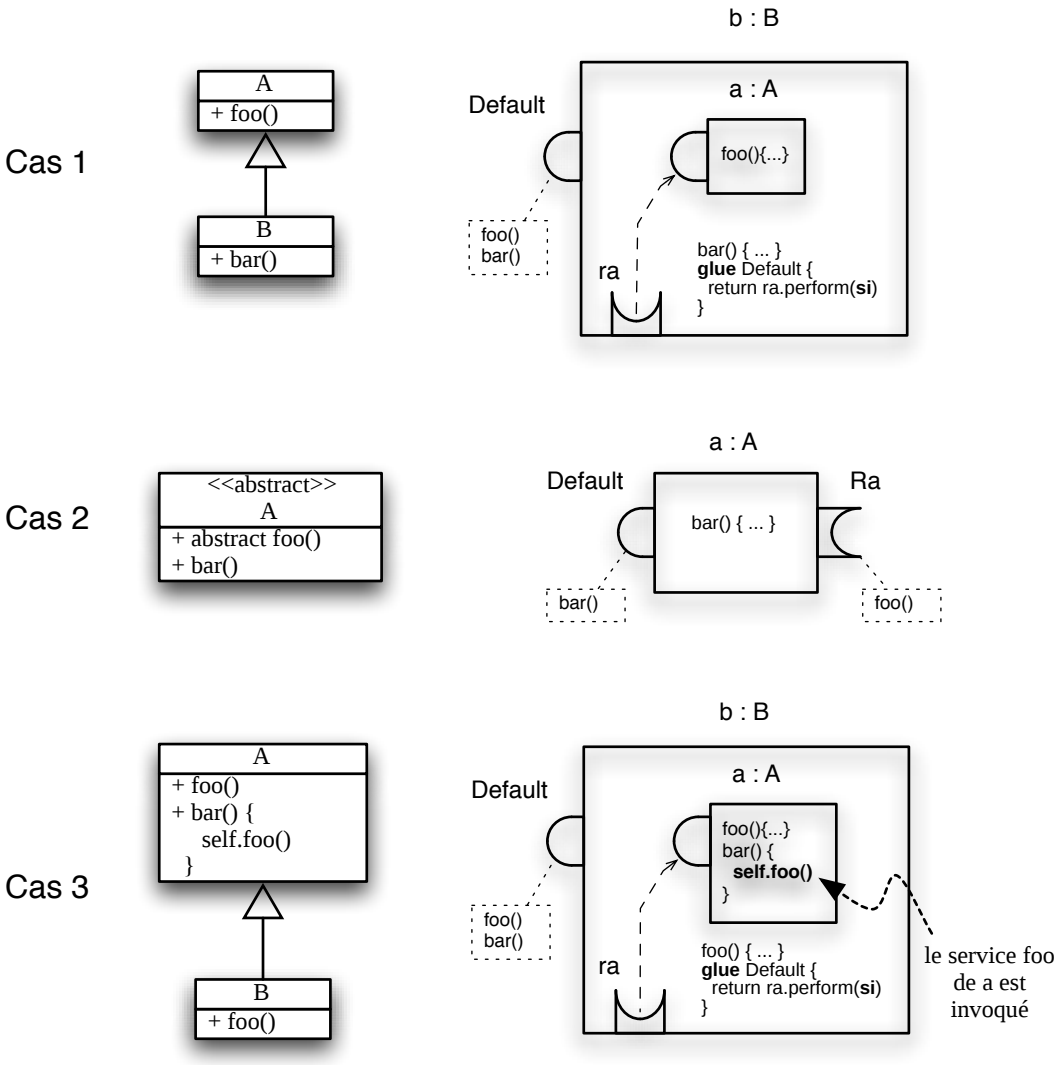

FIG. 3.21 : Analogies entre héritage et composition

3.14 Synthèse

On peut résumer de façon générale la spécification de SCL par les énoncés suivants :

1. Tout *composant* est instance d'un *descripteur de composant*.
2. Un descripteur définit le comportement de ses instances au moyen de services et la structure de ses instances au moyen de *ports requis* et de *ports fournis* dont certains peuvent être privés aux instances (*port interne*).
3. Un port requis `r` d'un composant `c` est un point d'accès pour `c` vers le monde extérieur. Les interactions possibles entre `c` et le monde extérieur via `r` sont spécifiées par l'*interface* de `r`.
4. Un port fourni `f` d'un composant `c` est un point d'accès pour le monde à un sous-ensemble des services fournis par `c`. L'interface de `f` spécifie les interactions possibles via `f`.
5. L'interface d'un port permet de spécifier quels services peuvent être invoqués via ce port et selon quel ensemble de contraintes (ordre des invocations, contrats, protocole des invocations, ...). Les interfaces sont réduites à un ensemble de signatures de services.
6. Pour pouvoir communiquer, les composants doivent être *connectés*. Connecter deux composants s'effectue via la *liaison* de leurs ports. Deux composants c_1 et c_2 sont connectés si au moins un port de c_1 est lié à un port de c_2.
7. Les composants communiquent au moyen d'*invocations de services* via leurs ports. Une invocation de service est constituée d'un port d'émission, d'un sélecteur (nom du service demandé) et de paramètres (références sur des ports).
8. Un composant c_1 peut invoquer un service d'un composant c_2 s'ils sont connectés c'est-à-dire qu'un port requis r_1 de c_1 est lié à un port fourni p_2 de c_2. Dans ce cas, l'invocation de service est émise par c_1 via r_1 et reçue par c_2 via p_2.
9. Quand un composant reçoit une invocation de service `i` via l'un de ses ports fournis `p`, il la traite selon l'algorithme 2 (*cf.* section 3.12).
10. Quand un composant émet une invocation de service `i` via l'un de ses ports requis `r`, il la traite selon l'algorithme 1 (*cf.* section 3.12).
11. Un composant peut jouer le rôle d'adapteur entre plusieurs autres, c'est-à-dire qu'il adapte les requêtes de services qu'il reçoit avant de les transmettre à d'autres composants. Tout composant standard peut jouer ce rôle d'adapteur et on le qualifie alors de *connecteur*. La programmation de connecteurs est facilitée SCL grâce à une forme générale de connecteurs et à la possibilité d'attacher un *service glue* à un port fourni.
12. Un *composite* est un composant qui encapsule un ou plusieurs composants appelés *sous-composants*. Les sous-composants sont privés à leur composite. Les liaisons de *délégation de ports* permettent au composite de rediriger les invocations de services vers ses sous-composants.

3.15 Bilan

Dans ce chapitre, nous avons présenté une argumentation concernant la conception d'un langage à composants. Nous avons commencé par définir notre vision d'un langage à composants ainsi que nos objectifs pour réalisation un tel langage. Nous avons ensuite détaillé de façon ordonnée, les problèmes que nous avons rencontrés lors de la spécification de notre langage à composants nommé SCL (*Simple Component Language*). Pour chaque problème, nous avons décrit les principales solutions existantes et présenté ensuite notre solution pour SCL ainsi que les raisons qui ont conduites à intégrer cette solution particulière. Ainsi, nous pensons que SCL propose des abstractions et des mécanismes adaptés à la programmation par composants ce qui n'est pas toujours le cas des langages comme Julia, ArchJava ou ComponentJ comme nous avons essayé de le montrer tout au long du chapitre. La section 3.14 fait la synthèse de toutes les constructions du langage SCL. Rappelons toutefois les points qui font la spécificité du langage SCL :

- l'intégration uniforme des objets de base qui sont considérés comme des composants,
- le mécanisme d'invocation de services et notamment celui du passage d'arguments en terme de connexions temporaires préservant ainsi l'intégrité des communications,
- le port interne nommé `self` permettant de faire des auto-références en utilisant l'invocation de service standard,
- la possibilité d'associer un *service glue* à un port fourni et la standardisation des connecteurs qui facilitent la résolution de conflits lors de l'assemblage des composants.

CHAPITRE 4

Séparation des préoccupations en SCL

C'est par la séparation qu'on évalue la force des liens.
Gérard GÉVRY.

Préambule

SCL *doit permettre la définition de composants réutilisables dans différents contextes pour lesquels ils n'ont pas été spécifiquement conçus. La modularisation des préoccupations que cela suppose est actuellement difficile à réaliser que ce soit avec une approche à composants ou avec une approche à objets. Ce chapitre a donc pour objectif de proposer une solution en ce qui concerne la séparation des préoccupations en* SCL*. Pour cela, nous commençons par présenter le principe de séparation des préoccupations. Nous décrivons ensuite la programmation par aspects (PPA ou* Aspect-Oriented Programming *soit AOP en anglais) qui est la solution permettant de mettre en œuvre ce principe dans le monde des objets grâce à l'introduction du concept d'*aspect *et du mécanisme de (*tissage*) notamment. Nous étudions ensuite les principales extensions des approches à composants qui ont intégré des idées issues de la PPA et nous proposons enfin deux extensions de* SCL *permettant la séparation des préoccupations. La première permet à un même composant* SCL *d'être utilisé de façon standard ou transversale grâce à une extension du concept de « port » et à l'ajout de nouveaux types de liaisons. La deuxième permet à l'architecte de réaliser des connexions basées sur les changements d'états des composants en affranchissant le programmeur de ces composants de l'écriture du code normalement indispensable pour réaliser ce type de connexion. Finalement, la sixième section conclut ce chapitre dans lequel nous avons développé les idées décrites dans [Fabresse, 2004; Fabresse* et al.*, 2004; Fabresse* et al.*, 2006a; Fabresse* et al.*, 2006b].*

4.1 Présentation de la programmation par aspects

LE principe de *séparation des préoccupations* (*separation of concerns*) [Parnas, 1972; Dijkstra, 1976] préconise le découpage d'une application en entités de taille plus réduite et aussi indépendantes que possible les unes des autres afin de faciliter la maintenance, la compréhension et la réutilisation. Cette modularisation du logiciel est toutefois difficile à mettre en place notamment pour certaines préoccupations qui ne peuvent être encapsulées dans des entités logicielles telles que des objets ou des composants comme la gestion des *logs* (journaux), des transactions ou encore la sécurité (identification, droits d'accès, etc.). Ces préoccupations ne relèvent généralement pas du domaine métier de l'application et elles sont désignées par le terme de *préoccupations transversales* (en anglais, *cross-cutting concerns*). Le code des préoccupations transversales est souvent noyé dans le code fonctionnel et distribué sur l'ensemble du programme notamment à cause de sa nature transversale. Dans la suite de cette section, nous présentons la *programmation par aspects* [Kiczales *et al.*, 1997], originellement proposée comme une extension de la programmation par objets, qui propose une solution face à ce problème.

La PPA propose notamment la notion d'*aspect* permettant d'encapsuler ces préoccupations transversales et le mécanisme de *tissage* pour intégrer le code des aspects et le code métier. La figure 4.1 détaille le principe général de la programmation par aspects. Les applications construites en utilisant les approches à objets ou à composants possèdent une architecture pouvant se résumer à du code métier (fonctionnel) qui utilise du code technique (non-fonctionnel). Par exemple la journalisation (*logging*) est typiquement une préoccupation transversale qui nécessite que le code métier fasse explicitement appel au code technique de journalisation pour chaque élément nécessitant d'être journalisé. La PPA propose d'encapsuler le code technique dans des aspects qui seront ensuite tissés avec le code métier — lequel ne contenant aucune référence explicite au code technique — afin de produire une nouvelle application intégrant la fonctionnalité technique.

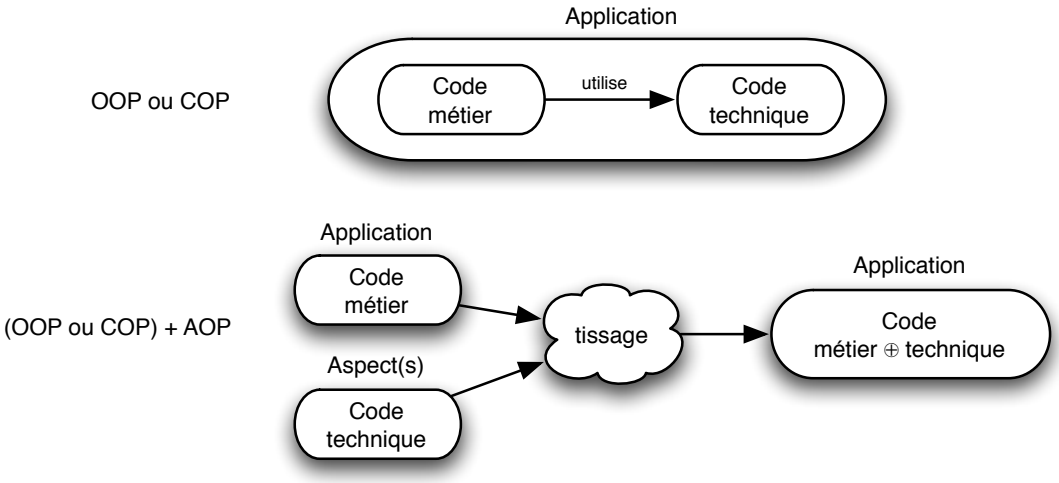

FIG. 4.1 : Principe de fonctionnement de la programmation par aspects

Dans cette section, nous allons présenter les principaux concepts et mécanismes de la PPA à tra-

vers le langage AspectJ [Kiczales *et al.*, 2001][1] qui est une référence dans ce domaine et un bon moyen d'illustrer la PPA. Cette présentation est synthétique et ne couvre pas tout le domaine de la PPA [Kiczales *et al.*, 1997], ni tout le langage AspectJ [2].

4.1.1 Point de jonction

Un point de jonction (*Join Point*) est un point précis et bien défini dans l'exécution d'un programme où il est possible d'exécuter du code transversal. Les principaux points de jonction disponibles en AspectJ sont les suivants :

- *method call* : point de jonction lors d'un envoi de message particulier,
- *method execution* : point de jonction lorsque le code d'une méthode est exécuté,
- *constructor call* / *execution* : points de jonction analogues aux points de jonction précédents mais pour les constructeurs,
- *field reference* : point de jonction lors de l'accès à un attribut,
- *field set* : point de jonction lors de la modification de la valeur d'un attribut.

À tout point de jonction sont associés trois éléments : l'objet courant (`this`), l'objet cible (`target`) et un tableau d'arguments (`args`).

4.1.2 Point de coupe

Un point de coupe (*Pointcut*) désigne un ensemble de points de jonction et expose éventuellement certaines valeurs dans le contexte d'exécution de ces points de jonction. En AspectJ, il existe un nombre fixé de primitives permettant de désigner des points de coupe comme :

- `call(MethodPattern)`, désigne un ensemble de points de jonction correspondant à des appels de méthodes dont la signature est conforme au pattern spécifié,
- `get(FieldPattern)`, désigne un ensemble de points de jonction correspondant à un accès à un attribut dont la signature correspond au pattern spécifié,
- `set(FieldPattern)`, désigne un ensemble de points de jonction correspondant à la modification d'un attribut dont la signature correspond au pattern spécifié.

Il existe un nombre fixé de points de coupe en AspectJ (nous ne les avons pas tous présentés) qui peuvent être combinés à l'aide d'opérateurs booléens : `&&` (et logique), `||` (ou logique) et `!` (négation). Le listing 4.1 montre un exemple de définition d'un point de coupe en AspectJ. Ce point de coupe est nommé `change` et comprend tous les points de jonction correspondant soit à l'appel de la méthode `setValue` de la classe `Counter`, soit à l'appel de la méthode `setPas` de la classe `Counter`. Dans la suite de cette section, nous utilisons cet exemple d'une classe `Counter` munie d'accesseurs (`get` et `set`) pour les propriétés `Valeur` et `Pas` (il s'agit de la valeur à ajouter au compteur lors de son incrémentation).

[1] Il existe d'autres langages à aspects comme HyperJ (`http://www.alphaworks.ibm.com/tech/hyperj`) ou encore AspectWerkz (`http://aspectwerkz.codehaus.org`).

[2] `http://www.eclipse.org/aspectj/`

```
pointcut change():
   call( void Counter.setValue(int) ) ||
   call( void Counter.setPas(int) );
```

LISTING 4.1 : Définition énumérative d'un point de coupe en AspectJ

Il est possible d'utiliser des « caractères joker » (*wildcards*) pour définir des points de coupes. Le listing 4.2 illustre cela en définissant le point de coupe `change` comme l'ensemble des points de jonction correspondants aux appels des méthodes publiques de la classe `Counter` dont le type de retour est `void`, le sélecteur commence par `set` et les arguments sont quelconques.

```
pointcut change():
   call( public void Counter.set*(..) );
```

LISTING 4.2 : Définition d'un point de coupe en AspectJ à l'aide de *wildcards*

4.1.3 Advice

Un *advice* associe un point de coupe à du code qui sera exécuté à chaque point de jonction défini par le point de coupe. On peut assimiler un *advice* à une sorte de méthode définissant le code non-fonctionnel qui sera exécuté lors de certains points de jonctions définis par un point de coupe. En AspectJ, cinq sortes d'*advices* sont disponibles[3] :

- *before advice* est exécuté avant l'exécution du point de jonction, par exemple avant le début de l'exécution de la méthode s'il s'agit d'un point de jonction *method call*,
- *after advice* est exécuté après l'exécution du point de jonction (*cf.* listing 4.3). Par exemple, dans le cas d'un point de jonction *method call*, l'*advice* est exécuté après l'exécution du corps de la méthode, juste avant que le flot de contrôle ne retourne à l'appelant. Un *after returning advice* correspond à un retour « normal » de la méthode, un *after throwing advice* correspond à un retour exceptionnel de la méthode par la levée d'une exception et un *after advice* correspond au retour de la méthode qu'il soit « normal » ou exceptionnel.
- *around advice* est exécuté à la place du point de jonction et peut éventuellement déclencher l'exécution du point de jonction.

```
after() : change() {
   System.out.println("A Counter has changed");
}
```

LISTING 4.3 : Un exemple d'*advice* en AspectJ

[3] Les *advices* ne sont pas sans rappeler le mécanisme de *combinaison de méthodes* en CLOS [Bobrow *et al.*, 1986] lui-même provenant des *démons* dans les langages à *frames* tel que Flavors [Moon, 1986].

4.1.4 Déclaration « inter-type »

Une déclaration *inter-type* (*inter-type declaration* aussi appelée *introduction*) consiste à déclarer des attributs et des méthodes en dehors de la hiérarchie des classes (dans un aspect comme nous le verrons dans la suite) et rattachés ensuite à des classes existantes. Contrairement aux *advices*, les déclarations *inter-type* sont traitées statiquement à la compilation et permettent de modifier la structure des classes en ajoutant des attributs ou des méthodes mais aussi en modifiant le graphe d'héritage c'est-à-dire la super-classe et les super-interfaces. Le listing 4.4 montre un exemple d'aspect modifiant la structure de la classe `Counter` afin d'en faire un *Javabeans* (*cf.* section 2.2.3). Pour cela, la super interface `Serializable` est ajoutée ainsi que des méthodes de gestion d'observateurs.

```
aspect CounterObserving {
   declare parents: Counter implements Serializable;

   private PropertyChangeSupport Counter.support = new PropertyChangeSupport(this);

   public void Counter.addPropertyChangeListener(PropertyChangeListener listener) {
      support.addPropertyChangeListener(listener);
   }

   public void Counter.removePropertyChangeListener(PropertyChangeListener listener) {
      support.removePropertyChangeListener(listener);
   }

   public void Counter.hasListeners(String propertyName) {
      support.hasListeners(propertyName);
   }
}
```

LISTING 4.4 : Exemple de modification de la structure d'une classe existante en AspectJ

Ce mécanisme est à rapprocher du mécanisme d'héritage puisqu'il offre les mêmes possibilités, à la différence que la classe existante est modifiée lors de la compilation au lieu de créer une nouvelle sous-classe. Modifier la classe existante permet à toutes les autres classes utilisant la classe modifiée de profiter des ajouts. Ce mécanisme est aussi sujet à de nombreux conflits [Kiczales *et al.*, 2001].

4.1.5 Aspect

Un *aspect* est une unité de modularité encapsulant une préoccupation transversale. Comme nous l'avons vu précédemment, la déclaration d'un aspect contient des déclarations *inter-type*, des *advices* et des points de jonction. Le listing 4.5 montre le code de l'aspect `CounterObserving` complet intégrant les *advices*.

En AspectJ, un aspect est similaire à une classe, sans toutefois être une classe. Un aspect peut étendre d'autres aspects et même une classe ou encore implémenter une interface. Les aspects sont instanciés non pas directement via l'opérateur `new` mais automatiquement.

```
aspect CounterObserving {
   declare parents: Counter implements Serializable;

   private PropertyChangeSupport Counter.support = new PropertyChangeSupport(this);

   public void Counter.addPropertyChangeListener(PropertyChangeListener listener) {
      support.addPropertyChangeListener(listener);
   }

   public void Counter.removePropertyChangeListener(PropertyChangeListener listener) {
      support.removePropertyChangeListener(listener);
   }

   public void Counter.hasListeners(String propertyName) {
      support.hasListeners(propertyName);
   }

   // définition d'un pointcut nommé changes dont la cible est un point
   pointcut changes(Point p): target(p) && call(void Point.set*(int));

   // lors des changements d'un point p, on notifie les écouteurs
   void around(Point p): changes(p) {
      String propertyName =
         thisJoinPointStaticPart.getSignature().getName().substring("set".length());
      int oldValue = p.getValue();
      int oldPas = p.getPas();
      proceed(p);
      if (propertyName.equals("Value")){
         firePropertyChange(p, propertyName, oldValue, p.getValue());
      } else {
         firePropertyChange(p, propertyName, oldPas, p.getPas());
      }
   }

   void firePropertyChange(Point p, String property, double oldval, double newval) {
         p.support.firePropertyChange( property,
                              new Double(oldval),
                              new Double(newval));
   }
}
```

LISTING 4.5 : Exemple de transformation d'une classe Java `Counter` en un composant *Javabeans* avec AspectJ

4.1.6 Tissage

Il s'agit du processus visant à combiner le programme de base (code métier) constitué d'objets ou de composants avec les aspects afin de produire une application étendue avec les comportements définis par les aspects. On distingue généralement le tissage statique (à la compilation) et le tissage dynamique (à l'exécution). AspectJ effectue le tissage statiquement lors de la compilation. L'application compilée résultante intègre le code métier et celui des aspects. La phase de tissage doit faire face à des conflits [Sanen *et al.*, 2006; Pulvermüller *et al.*, 2001] (ce problème est aussi appelé *feature interaction problem*) lorsque différents aspects sont à mettre en œuvre pour un même point de jonction. Différentes politiques de résolution des conflits sont possibles, depuis une résolution manuelle jusqu'à une résolution automatique par le tisseur en fonction de critères d'ordonnancement des aspects par exemple.

4.1.7 Synthèse et remarques

Finalement, la programmation par aspects peut se résumer ainsi [Filman et Friedman, 2000] :

> « *AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers.* »

La quantification consiste à exprimer des conditions sur la structure statique ou la dynamique d'un programme de base afin de déclencher des traitements non-fonctionnels. La non-anticipation (*obliviousness*) indique que les programmes de base sont écrits par des programmeurs n'ayant pas connaissance de ces quantifications. Cette propriété de non-anticipation n'est pas vérifiée dans certaines approches à aspects, souvent appelées *canevas*[4], où le code de base est écrit avec des « trous » dans lesquels viendront s'insérer les nouvelles fonctionnalités définies par la suite. L'approche par canevas n'est pas satisfaisante car elle impose au programmeur de l'application de base de « prévoir ces trous ».

Comme cela est souligné dans [Filman et Friedman, 2000], de nombreux systèmes permettent de mettre en pratique la PPA de façon plus ou moins complète sans pour autant être des langages à aspects comme c'est le cas des systèmes à base de règles, de méta-programmation ou encore de génération de code. En résumé, les principaux éléments d'un langage à aspects (ou plus généralement une approche à aspects) sont les suivants :

Un modèle de points de jonction qui définit les points de jonctions disponibles dans un programme. Ce modèle est fondamental puisque les points de jonction déterminent les points d'adaptations. On distingue généralement les points de jonction statiques et les points de jonction dynamiques qui dépendent de valeurs uniquement connues lors de l'exécution [Stoerzer et Hanenberg, 2005].

Un langage de coupe permettant de sélectionner un ensemble de points de jonctions. Les langages de coupe doivent proposer des constructions permettant de sélectionner des points de jonctions statiques ou dynamiques. Le langage de points de coupe permettant de désigner des

[4] Certainement en référence aux *frameworks*.

points de jonction d'AspectJ est relativement simple puisqu'il s'agit essentiellement de patrons syntaxiques et d'opérateurs booléens.

Un modèle d'aspects qui définit la structure d'un aspect. Par exemple, en AspectJ les aspects contiennent des définitions *inter-types*, des *advices* et des points de coupe. Cette définition rend les aspects en AspectJ peu réutilisables puisqu'ils contiennent des points de coupes dont la définition est généralement très liée à l'application de base. Il existe d'ailleurs des travaux qui ont proposé de séparer d'un côté, la définition des aspects et de l'autre, l'application métier afin d'augmenter leur réutilisation [Lieberherr *et al.*, 1999].

Un tisseur est un compilateur ou un interprète qui mélange un programme de base et des aspects pour produire un programme qui met en œuvre de multiples préoccupations. Le tissage peut être effectué statiquement ou dynamiquement. Lorsqu'il est réalisé statiquement, il est généralement impossible d'enlever ou ajouter dynamiquement des aspects. Dans tous les cas, il doit permettre la résolution de conflits soit en offrant au programmeur la capacité de les résoudre, soit en intégrant une politique de résolution automatique.

4.2 Aspects et approches à composants

De même que les approches à objets, les approches à composants souffrent d'insuffisances en ce qui concerne la modularisation des préoccupations transversales [Lieberherr *et al.*, 1999; Duclos *et al.*, 2002]. Dans la suite de cette section, nous présentons divers travaux concernant l'intégration des aspects et des composants. Nous présentons ces travaux en fonction des approches à composants sur lesquelles ils sont basés :

- des approches à composants réparties comme CCM ou EJB,
- des modèles de composants (souvent académiques) comme Fractal,
- des langages à composants (au niveau du code source) comme FuseJ.

4.2.1 Aspects et approches à composants industrielles

Comme nous l'avons vu dans la section 2.1.3, les approches à composants réparties reposent sur des intergiciels qui fournissent des services non-fonctionnels comme la persistance, les transactions et la sécurité. Ces services non-fonctionnels sont des fonctionnalités transversales qui ne peuvent être encapsulées par des composants. Il s'agit d'aspects que le programmeur peut mettre en œuvre soit déclarativement dans le descripteur de déploiement soit par programmation grâce aux méthodes fournies par le *framework* d'implémentation d'un composant. Les composants n'accèdent pas directement aux services non-fonctionnels, sinon le code d'accès serait mélangé au code métier. Pour cela, les communications entre la couche métier et la couche technique sont contrôlées par une tierce entité au sein de la plateforme intergicielle : le *conteneur* de composant qui utilise le code métier lorsque cela est nécessaire et non l'inverse. Ce principe d'externalisation, parfois appelé inversion de contrôle (*inversion of control*), est tout à fait similaire au fonctionnement des *frameworks* où le code écrit par un programmeur est appelé par le code du *framework*.

La réalisation de ces fonctionnalités transversales dans les approches à composants réparties est assez rigide et limitative. Les principales insuffisances de ces modèles sont les suivantes :

- Les fonctionnalités transversales sont souvent simples et limitées. Par exemple, le modèle de sécurité de J2EE est basé sur le contrôle d'accès à l'aide de rôles mais il ne propose pas d'héritage de droits d'accès entre un rôle et un sous-rôle.
- Le paramétrage des fonctionnalités transversales est basé sur l'extension de *framework* par héritage comme en J2EE, ce qui limite les extensions possibles aux méthodes déclarées par le *framework*.
- L'ajout de nouvelles fonctionnalités transversales est souvent impossible ou alors nécessite de modifier directement l'implémentation de l'intergiciel.

De nombreux travaux proposent d'utiliser des techniques de la programmation par aspects pour prendre en compte ces fonctionnalités transversales dans les approches à composants réparties et pallier ces limitations. Parmi les extensions des approches à composants réparties avec des aspects, on peut citer AspectJ2EE [Cohen et Gil, 2004], JBossAOP [Pawlak *et al.*, 2004], AES [Choi, 2000] ou encore CVM [Duclos *et al.*, 2002] qui sont des extensions du modèle EJB intégrant des possibilités offertes par les approches à aspects. Une comparaison de ces principales approches a été réalisée dans [Oussalah, 2005] et nous en rappelons ici les éléments essentiels :

- L'approche boîte noire est conservée et un aspect ne peut modifier directement l'implémentation d'un composant (contrairement à AspectJ qui peut modifier l'implémentation d'une classe) cela afin de garantir la possibilité d'utilisation d'un composant développé par un tiers. Ainsi, les modèles de coupes n'autorisent généralement pas les définitions *inter-types* ou alors de façon limitée comme c'est le cas en JBossAOP avec des *mixins*.
- La plupart de ces approches sont basées sur l'extension du conteneur de composants. En effet, le conteneur est l'entité privilégiée pour l'intégration des aspects puisqu'il traite toutes les requêtes émises et reçues par un composant. Le modèle de coupes proposé est donc généralement basé sur l'interception des appels de méthodes par le conteneur. Ainsi, les modèles de coupes supportent tous au minimum les appels de méthodes (`call` en AspectJ) puisqu'ils ne remettent pas en cause le modèle boîte noire. Certaines approches proposent toutefois des coupes spécifiques. Par exemple, AspectJ2EE permet de spécifier une coupe spécifique pour les invocations de méthodes distantes via le mot-clef `remotecall` (en plus de celle similaire à `call` en AspectJ pour les appels locaux) qui permet de définir le traitement d'un appel à la fois du côté du client et du côté du serveur, c'est-à-dire à différents niveaux de l'application (*tier-cutting*).
- Le tissage prend des formes variées dans ces approches. Toutefois, elles ont toutes pour objectifs de préserver la compatibilité avec le support d'exécution standard du modèle industriel de composants sous-jacent. C'est ainsi que les techniques de tissage utilisées s'appuient sur des mécanismes comme la génération d'encapsulateurs (*wrappers*). Par exemple, en AspectJ2EE le tissage est effectué lors du déploiement par génération de sous-classes des classes d'implémentation de l'EJB.

Toutes les approches présentées dans cette section sont construites à partir des modèles de composants dits industriels et se spécialisent dans le domaine des applications réparties dont les spécificités contraignent la mise en œuvre de l'approche à aspects. De plus, ils sont pour la plupart basés

sur des approches de type canevas qui ne satisfait pas complètement le besoin de non-anticipation. Dans la section suivante, nous présentons d'autres approches construites par extension de modèles de composants académiques comme Fractal.

4.2.2 Aspects et modèles de composants

Des extensions de modèles de composants (comme Fractal par exemple) sont actuellement proposées afin d'offrir une meilleure modularisation des préoccupations transversales. Dans la suite de cette section, nous détaillons deux approches qui nous paraissent représentatives de cette volonté d'intégrer les concepts et mécanismes de l'approche à aspects dans un modèle de composants en l'occurrence Fractal.

Fractal-AOP

Fractal-AOP [Fakih *et al.*, 2004] propose une extension du modèle de composants Fractal afin de supporter la PPA. Cette extension repose sur l'ajout de deux nouvelles interfaces de contrôle `cEC` et `sIC`, de composants de tissage et de composants d'*advice* (*cf.* figure 4.2).

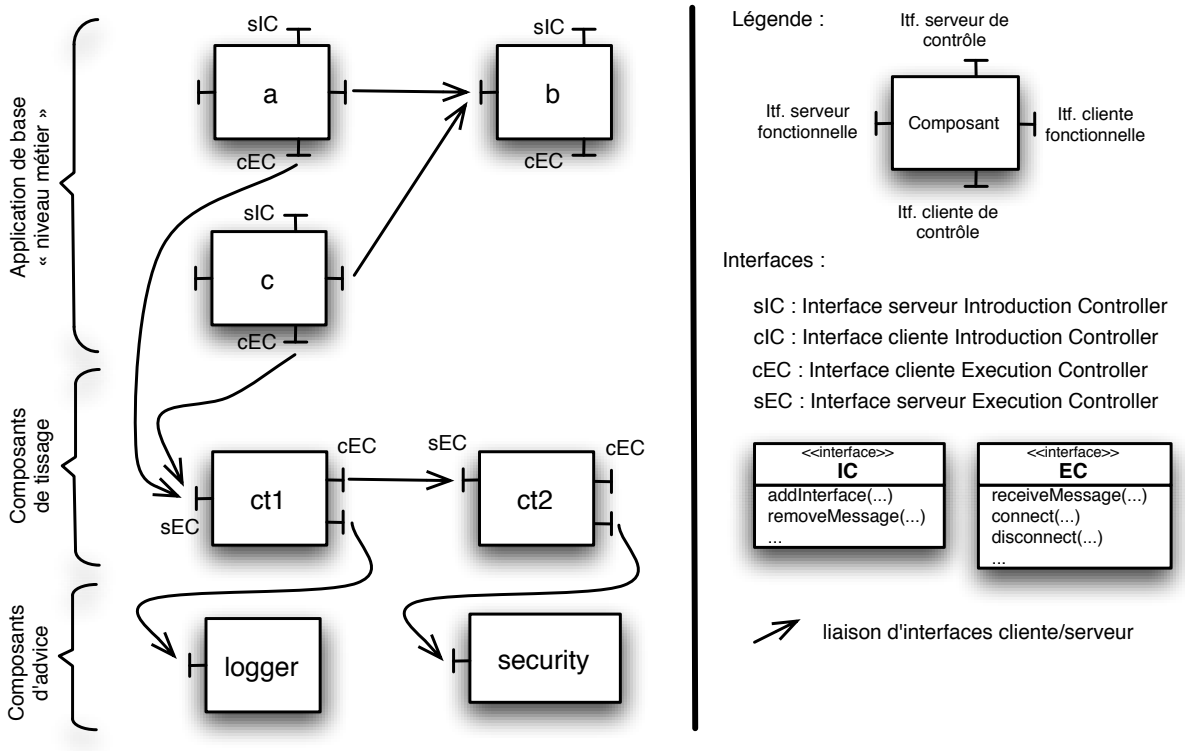

FIG. 4.2 : Schéma général du modèle Fractal-AOP

Examinons tout d'abord les deux nouvelles interfaces associées aux composants métiers `a`, `b` et `c` dans la figure 4.2 :

- *Client Execution Controller* (`cEC`) qui est une interface de contrôle cliente donnant accès aux occurrences des points de jonction dans le flot d'exécution du composant. En effet, à chaque point de jonction, la membrane du composant va émettre une invocation d'opération via cette interface afin que les *composants d'aspects*, décrits ci-dessous, puissent intervenir et effectuer leurs traitements. Par exemple, cette interface déclare les deux opérations suivantes : `receiveMessage` qui définit un point de jonction lors de la réception d'un message par le composant à travers l'une de ses interfaces de rôle serveur et `connect` qui définit un point de jonction lors de la connexion d'une interface du composant.
- *Server Introduction Controller* (`sIC`) qui est une interface serveur permettant aux aspects d'effectuer des introductions, c'est-à-dire des changements transversaux statiques modifiant la structure d'un composant. Par exemple, cette interface déclare les opérations suivantes : `addInterface` qui permet d'ajouter une interface à un composant ou encore `removeMessage` qui permet d'enlever une opération d'une interface d'un composant. Bien évidemment, les introductions nécessitent de vérifier les liaisons déjà établies pour s'assurer de leur validité après modification.

En résumé, les points de jonctions disponibles en Fractal-AOP [Fakih *et al.*, 2004] sont :

- écriture/lecture des attributs,
- connexion/déconnexion de deux composants,
- ajout/suppression d'un sous-composant à un composite,
- changement de l'état de cycle de vie d'un composant,
- ajout/suppression d'un attribut/opération/interface au composant,
- réception d'un message sur les interfaces de rôle serveur,
- émission d'un message sur les interfaces de rôle client.

Un aspect est mis en place via des *composants d'advice* et des *composants de tissage*. Un composant d'*advice* définit les traitements d'un aspect particulier à l'aide de composants primitifs ou composites. Un composant de tissage met œuvre le tissage entre les composants métiers et les composants d'*advice*, c'est-à-dire qu'il gère les définitions de coupes, les règles de tissage et est en charge de l'invocation des opérations des composants d'*advice*. Pour cela, un composant de tissage possède une interface fonctionnelle `sEC` (*Server Execution Controller*) sous-type de l'interface cliente cEC à travers laquelle il reçoit des invocations d'opération correspondant aux points de jonctions définis dans l'interface cEC des composants métiers connectés. Le composant de tissage peut alors filtrer les points de jonctions et déclencher les traitements des composants d'*advice* si nécessaire.

Dans le cas des composites, les interfaces `sIC` et `cEC` sont enrichies et proposent des points de jonction spécifiques permettant de déterminer le sous-composant auquel ils se rattachent. Les interfaces `cEC` des sous-composants peuvent être liées (au choix de l'architecte) à celle du composite, de même que l'interface sIC peut être liée à celle des sous-composants.

Les conflits de tissage sont résolus par l'architecte qui peut définir un ordre entre les composants de tissage en les liant via leurs interfaces `sEC` et `cEC`. Cette solution est inspirée du schéma de conception « chaîne de responsabilité » [Gamma *et al.*, 1995]. Dans la figure 4.2, le composant de tissage

ct_1 directement connecté au composant métier est prioritaire par rapport au composant de tissage suivant dans la chaîne ct_2 connecté à ct_1 via leurs interfaces respectives `sEC` et `cEC`.

Critique. Les avantages de cette solution sont d'une part sa simplicité et d'autre part la compatibilité avec le modèle d'exécution de Fractal puisqu'il n'y a pas de nouvelles abstractions ou de mécanismes spécifiques. Fractal-AOP propose un modèle de points de jonction qui est un modèle « boîte blanche » autorisant d'une part les introductions et même les points de coupe sur les sous-composants d'un composite. Le choix d'un modèle boîte blanche s'explique certainement par le fait qu'un composite peut exporter son contenu en Fractal. En Fractal-AOP, les aspects sont construits à partir de composants d'*advice* et de tissage mais ne sont pas des composants. Ce choix de ne pas réifier les aspects sous la forme de composites est volontaire de la part des auteurs qui s'interrogent sur les implications d'un tel choix notamment du point de vue de la répartition des aspects.

Fractal Aspect Component (FAC)

FAC [Pessemier *et al.*, 2006] est aussi une extension du modèle de composants Fractal qui vise à intégrer le développement par aspects et le développement par assemblage de composants. En FAC, tous les concepts de la PPA sont modélisés en utilisant ceux de l'approche à composants. FAC distingue les éléments suivants (*cf.* figure 4.3) : *composant aspectisable* (*aspectisable component*), *composant d'aspect* (*aspect component*), *liaison d'aspect* (*aspect binding*) et *domaine d'aspect* (*aspect domain*).

Tout d'abord, les points de jonction supportés par FAC sont les réceptions et les émissions d'appels d'opérations à travers les interfaces des composants aspectisables. Les composants aspectisables sont des composants métier qui peuvent être tissés avec des aspects (composants A, B et C sur la figure 4.3). Par rapport à des composants Fractal standards, les composants aspectisables fournissent une interface nommée `WI` (*Weaving Interface*) contenant les opérations permettant le tissage du composant comme `setAspectBinding` qui permet de tisser un composant d'aspect à un point de coupe donné du composant ou encore `unsetAspectBinding` qui effectue l'opération inverse.

Un composant d'aspect encapsule une préoccupation transversale. Le listing 4.6 montre un exemple de définition de composant d'aspect qui correspond au composant `ac1` de la figure 4.3. Il s'agit d'un composant Fractal standard fournissant une interface fonctionnelle serveur nommé `ACI` (*Aspect Component Interface*) fournissant l'opération `invoke`. Cette opération prend en paramètre un point de jonction et elle définit un *around advice* c'est-à-dire le code à exécuter avant et après ce point de jonction. Contrairement à AspectJ, FAC ne supporte que les *around advice* qui sont finalement les plus généraux puisqu'un *advice* de type *before* ou *after* peut être mis en place avec un *around advice*.

En Fractal, les composants sont liés via leurs interfaces clientes et serveurs. En FAC, ces liaisons sont appelées *liaisons régulières* pour les différencier des *liaisons d'aspects* qui permettent de lier un composant via son interface `WI` à un composant d'aspect via son interface `ACI`. Ces liaisons d'aspects n'obéissent pas aux mêmes contraintes que les liaisons régulières comme le sous-typage des interfaces client et serveur et n'offrent pas la même sémantique puisque dans le cas d'une liaison d'aspect, l'opération `invoke` de l'interface `ACI` est appelée pour chaque point de jonction défini lors de la liaison.

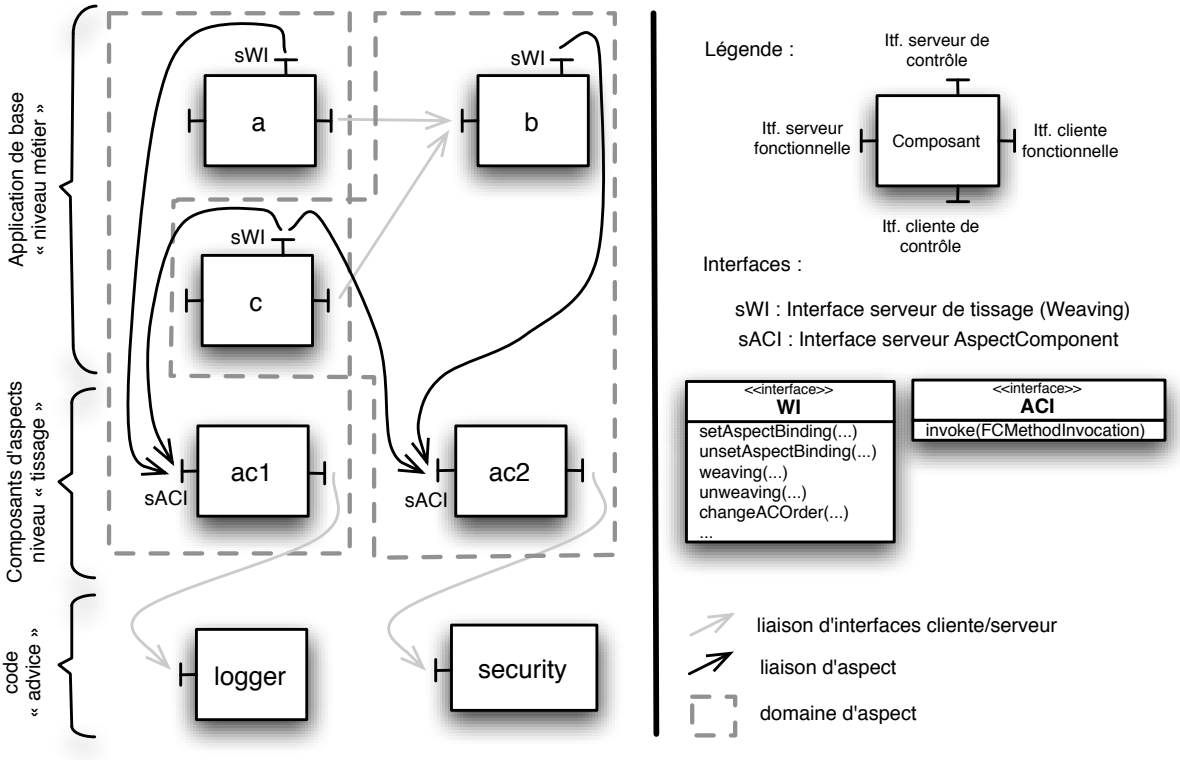

FIG. 4.3 : Schéma général du modèle FAC

```
@AspectComponent
public class AC1 implements AspectComponent {
   //  AC1 possède une interface client nommée logger
   @Requires(name = "logger")
   private ILogger logger;

   //  Implémentation de l'interface AspectComponent
   public Object invoke(FcMethodInvocation m) throws Throwable {
      logger.log(m.getComponentName() + " before method " + m.getMethod().getName());
      Object ret = m.proceed(); //  Exécution de la méthode interceptée
      logger.log("/// Trace AC: after method "+ m.getMethod().getName());
      return ret;
   }
}
```

LISTING 4.6 : Exemple de définition d'un composant d'aspect en FAC

Un domaine d'aspect est la réification sous la forme d'un composite du domaine d'application d'un composant d'aspect. Ce composite est créé automatiquement et il contient un composant d'aspect ainsi que tous les composants métiers tissés avec ce composant d'aspect. Un composant métier peut appartenir à plusieurs domaines d'aspect puisque le modèle Fractal supporte le partage de sous-composants. Les conflits de tissage sont résolus localement pour chaque composant aspectisable via l'interface `WI` qui fournit le service `changeACOrder`. Ce service permet d'ordonner les composants d'aspects tissés sur un composant aspectisable.

FAC propose aussi une extension de l'ADL Fractal (langage XML) afin de supporter la définition des liaisons d'aspect et la définition des coupes via des expressions régulières comme le montre le listing 4.7. La balise **`weave`** décrit le tissage du composant d'aspect `ac1` avec chaque sous-composant du composite `monApplication` (`rootComp="this"`) de sorte que les opérations reçues par les interfaces serveurs de ces sous-composants soient interceptées (attribut `pcd`). Le domaine d'aspect est nommé `logDomain`.

```
<definition name="monApplication" >
   <interface name="r" signature="java.lang.Runnable" role="server"
      cardinality="singleton" contingency="mandatory"/>
   <component name="a" definition="componentA" />
   <component name="b" definition="componentB" />
   <component name="c" definition="componentC" />

   <-- Définition du component d'aspect -->
   <component name="ac1" definition="componnetAC1" />

   <-- Liaisons « standards » -->
   <binding client="this.r" server="a.r" />
   <binding client="a.ci" server="b.si" />
   <binding client="c.ci" server="b.si" />

   <-- Définition du tissage -->
   <weave rootComp="this" ac="ac1" pcd="SERVER *;*;*" aDomain="logDomain" />
   <controller desc="aspectComposite"/>
</definition>
```

LISTING 4.7 : Exemple de définition de tissage en FAC

Critique. FAC propose une intégration des aspects et des composants en s'intéressant particulièrement à l'application des concepts issus des composants aux concepts de la PPA. C'est ainsi qu'une préoccupation transversale est encapsulée dans un composant d'aspect, le domaine d'application d'un aspect est aussi réifié sous la forme d'un composite et les liaisons d'aspect représentent les liens de tissage entre les composants d'aspects et les composants aspectisables. Contrairement à Fractal-AOP, FAC propose un modèle de point de jonction de type boîte noire et a introduit un nouveau type de liaison d'interfaces.

4.2.3 Aspects et langages à composants

Dans cette section, nous présentons les approches à composants et à aspects, au niveau des langages de programmation. Il existe une multitude d'approches visant à combiner les approches aspects et composants au niveau des langages de programmation comme Jiazzi [McDirmid *et al.*, 2001b], *Aspectal Collaborations* [Lieberherr *et al.*, 2003], Caesar [Mezini et Ostermann, 2003] ou encore FuseJ [Suvée *et al.*, 2005; Suvée *et al.*, 2006]. Nous présentons dans la suite de cette section le langage FuseJ qui est le plus récent et aussi le plus proche de SCL. Une étude incluant Jiazzi [McDirmid *et al.*, 2001b], *Aspectal Collaborations* [Lieberherr *et al.*, 2003] et Caesar [Mezini et Ostermann, 2003] (mais pas FuseJ) est disponible dans [Oussalah, 2005].

En FuseJ, un composant est constitué de *gates* (ensemble de méthodes) fournies ou requises. La connexion de composants est réalisée via des *connecteurs* et des *linklets* (liaisons) entre les *gates*. Différentes sortes de *linklets* existent : les *regular* (requis/fournis ou délégation) et les *crosscutting* (*before, after, around*) qui correspondent aux *advices* sur les appels de services. La figure 4.4 montre une architecture d'application intégrant des composants et des aspects en FuseJ. La listing 4.8 présente la déclaration de la connexion entre les composants A, C et LOGGER.

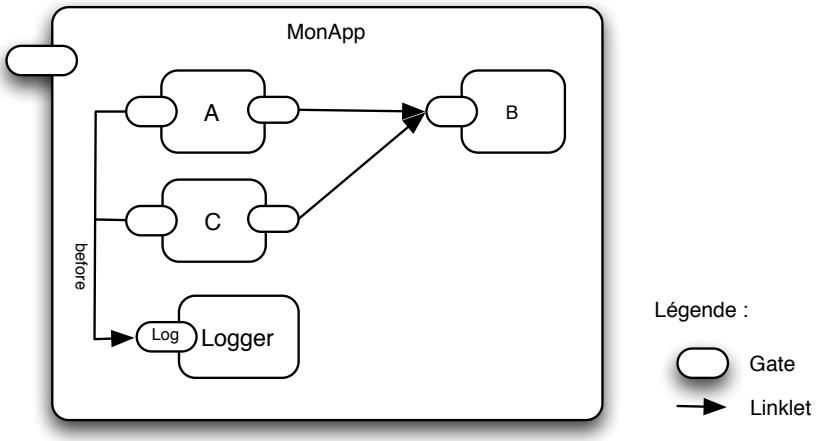

FIG. 4.4 : Schéma général du modèle FuseJ

Les auteurs de FuseJ n'abordent pas les problèmes de conflits. Par exemple, est-il possible d'intégrer deux *linklets* de type *before* sur une même *gate*? Et dans l'affirmative, quelle est leur priorité respective?

Critique. L'approche proposée par FuseJ est similaire à celle de FAC dans le sens où de nouveaux types de liaisons sont introduits. De plus, ces nouveaux *linklets* sont comparables aux composants d'aspects de FAC. Toutefois, FuseJ semble moins avancé que FAC ou Fractal-AOP actuellement puisque les conflits ne sont actuellement pas traités.

```
configuration MonApp ... {
   linklet log {
      execute:
         Logger.log(String st);
      before:
         A.*(..);
         B.*(..);
      where
         st = Source.getMethodSignature();
   }
}
```

LISTING 4.8 : Exemple de code en FuseJ

4.3 Séparer les préoccupations en SCL

Comme nous l'avons montré jusqu'ici, les approches à aspects et à composants peuvent être intégrées afin d'apporter une meilleure modularisation des préoccupations transversales dans les applications construites par assemblage de composants. Dans cette section, nous présentons la façon dont SCL a été étendu afin de supporter la PPA sans pour autant remettre en question les choix que nous avons effectués dans le chapitre 3. Chaque sous-section qui suit est une discussion visant à déterminer s'il est nécessaire d'intégrer un nouvel élément en SCL et dans l'affirmative, de choisir sous quelle forme faire cette intégration afin de permettre la séparation des préoccupations transversales.

4.3.1 Quels points de jonction ?

Boîte noire ou boîte blanche ? AspectJ et Fractal-AOP proposent un modèle de points de jonction de type boîte blanche. Avec un tel modèle de points de jonctions, les aspects peuvent être tissés de façon précise et fine dans une application de base en accédant par exemple aux sous-composants des composites, aux attributs privés, etc. Il est même possible de modifier la structure des éléments grâce aux introductions notamment. Bien évidemment, un modèle boîte blanche suppose l'accès au code source de l'application de base. Les coupes deviennent alors très dépendantes de l'application pour laquelle elles sont définies ce qui peut réduire les possibilités de réutilisation des aspects lorsque leur définition intègre des définitions de coupe. *A contrario*, AspectJ2EE ou encore FAC proposent des modèles de points de jonction de type boîte noire. Ces modèles ne permettent à un aspect d'accéder directement à l'implémentation des composants constituant l'application de base. Cette dernière approche, respectueuse de l'encapsulation et du découplage, semble mieux adapté au développement par composants où un composant doit pouvoir être remplacé par un autre composant offrant les fonctionnalités adéquates indépendamment de son implémentation. C'est pourquoi nous avons fait le choix suivant pour SCL :

Choix 21 *Le modèle de points de jonction est de type boîte noire.*

Non-anticipation. Comme nous l'avons indiqué dans la section 4.1.7, la propriété de non-anticipation (*obliviousness*) est importante. Toutefois, elle n'est pas supportée par toutes les approches comme c'est le cas pour les approches à composants réparties. Fractal-AOP et FAC ne semblent pas supporter complètement cette propriété. Pour être « aspectisables », les composants métiers doivent posséder les interfaces `cEC` et `sIC` dans le cas de Fractal-AOP et l'interface `sWI` dans le cas de FAC. Si un programmeur doit décider de doter son composant d'interfaces particulières afin qu'il puisse supporter le tissage, on peut considérer que la propriété de non-anticipation est violée. Par contre, si tous les composants sont (automatiquement) dotés d'une interface de tissage sans que le programmeur n'ait à s'en soucier ni à écrire de code spécifique pour cela, la non-anticipation n'est pas violée. Cette propriété de non-anticipation nous semble cruciale, aussi bien pour le développement des composants (*cf.* objectif 3) que pour l'intégration des aspects.

Choix 22 *Tous les composants peuvent être tissés indépendamment de leur conception et de leur implémentation.*

Finalement, en considérant les deux choix précédents, nous avons intégré les points de jonction suivants en SCL :

- les invocations de services à travers un port,
- la connexion/déconnexion d'un port de composant.

En SCL, les ports étant les uniques points d'accès d'un composant, ils sont donc les candidats idéaux pour supporter un modèle de points de jonction de type boîte noire. De plus, un composant possède toujours des ports et est donc toujours « aspectisable » (respect de la non-anticipation). Les réceptions d'invocation de service via les ports fournis et l'émission d'invocations de service via les ports requis sont les éléments minimum d'un tel modèle. Nous avons aussi intégré les points de jonction correspondant à la connexion et la déconnexion d'un port présents en Fractal-AOP. Au niveau des *advices*, SCL supporte les *advices* de type *before* et *after* qui peuvent être combinés afin de supporter ceux de type *around*, contrairement à FAC qui supporte uniquement les *advices* de type *around*. Ce choix n'est absolument pas restrictif ni pour SCL, ni pour FAC puisque les possibilités sont équivalentes. Nous verrons dans la suite que cela permet de s'affranchir en SCL de déclencher explicitement le traitement d'un point de jonction comme c'est le cas en FAC via la méthode `proceed` d'un point de jonction (*cf.* listing 4.6).

4.3.2 Qu'est-ce qu'un *advice* ?

On distingue généralement deux sortes d'approches à aspects :

- Les approches asymétriques comme AspectJ, Caesar ou JAsCo qui considèrent les aspects comme des entités différentes de celles constituant l'application de base. Par exemple, en AspectJ, un aspect est une entité différente d'une classe possédant des *advices* qui ne sont pas des méthodes.
- Les approches symétriques comme HyperJ, FAC, Fractal-AOP ou FuseJ représentent le programme de base et les préoccupations transversales (aspects, *advices*, etc.) avec les mêmes entités, en l'occurrence des composants et des connexions.

Cette seconde approche nous semble unifiée et promettre une meilleure réutilisation. En effet, un composant défini et mis sur étagère peut ensuite être utilisé pour construire la partie de base ou la partie transversale d'une application. Cela n'est pas forcément possible avec les approches asymétriques. En AspectJ, si l'on définit un aspect de journalisation par exemple, il ne peut être utilisé de façon standard puisque ce n'est pas une classe. En SCL, nous avons choisi d'adopter une approche symétrique et donc de représenter les concepts de la PPA à l'aide des concepts de l'approche à composants.

Choix 23 *Les services des composants peuvent être utilisés comme préoccupation transversale (*advice*).*

Le code transversal peut être encapsulé dans un composant. Par exemple, on peut construire un composant LOGGER fournissant le service `log`. Cela permet d'assurer que le code transversal pour une application donnée puisse être utilisé comme du code de base pour une autre application. Séparer la définition du code transversal de son utilisation avec une application donnée, en ne spécifiant pas les points de coupe notamment, permet d'augmenter ses possibilités de réutilisation [Lieberherr *et al.*, 1999]. Toutefois, les appels à du code transversal (le service `log` par exemple) ne peuvent être indépendants du code de l'application puisqu'ils sont par définition disséminés dans le code des composants métiers. Cette problématique est l'objet de la section suivante.

4.3.3 Comment effectuer le tissage ?

Fractal-AOP et FAC représentent deux approches différentes en ce qui concerne le tissage. Fractal-AOP utilise la liaison « standard » (entre une interface requise et une interface fournie) pour lier un composant à un composant de tissage, lui-même lié à un composant d'*advice*. Cela suppose de doter tous les composants métiers d'une interface cliente, nommée `cEC` en Fractal-AOP, définissant les points de jonctions et permettant sa liaison avec les composants de tissage. FAC n'utilise pas la liaison standard et a introduit la liaison d'aspect pour lier un composant métier à un composant d'aspect. Toutefois, ces liaisons d'aspects sont aussi établies entre des interfaces spécifiques des composants métiers (dits aspectisables en FAC) et des composants d'aspects.

En SCL, nous avons adopté une approche similaire à celle de FAC en introduisant de nouveaux types de liaison de ports. Cela permet d'une part de bien distinguer les liaisons de type requis/fournis de celles relevant des aspects et d'autre part de ne pas introduire de port particulier pour mettre en place ces liaisons afin de rendre tous les composants potentiellement aspectisables sans effort du programmeur. La figure 4.5 présente un exemple d'intégration d'aspects dans une application en SCL. Cet exemple reprend l'architecture à deux composants utilisée précédemment dans la figure 3.12 et ajoute la fonctionnalité transversale de journalisation en utilisant un composant FILELOGGER standard, un connecteur standard et une nouvelle sorte de liaison de ports de type *bSI* (*before service invocation*). Lorsqu'une invocation de service est reçue par l'un des composants `pm` ou `rng` via leurs ports fournis liés par une liaison de type *bSI*, cette invocation est d'abord transmise via cette liaison avant d'être traitée. Le connecteur `adHocConnector` reçoit alors cette invocation qui la traite normalement et exécute son *code glue* dans cet exemple.

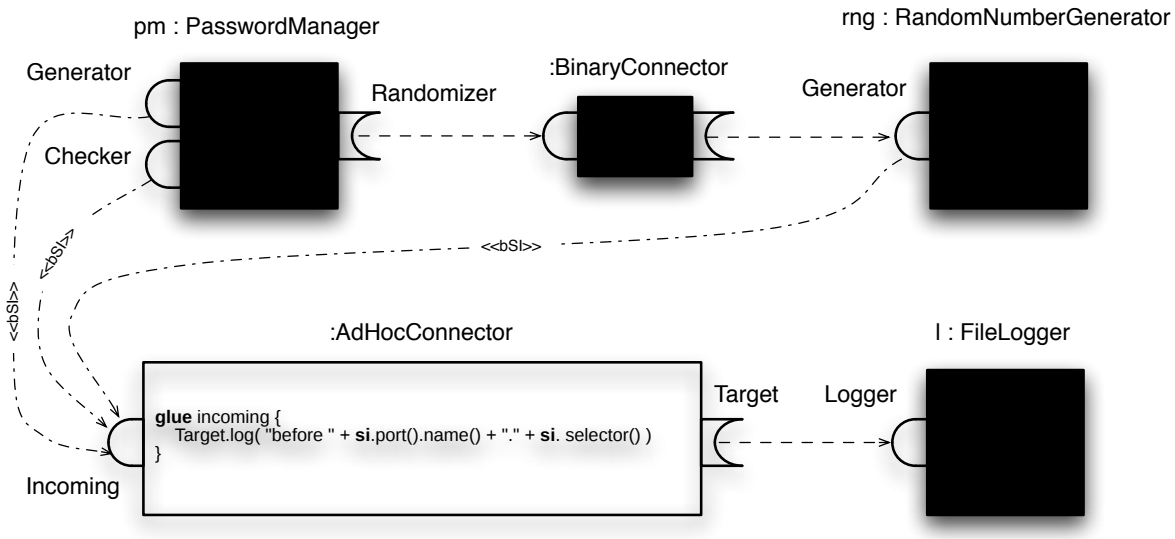

FIG. 4.5 : Schéma général de l'intégration des préoccupations transversales en SCL

Les éléments permettant de faire de la PPA en SCL sont : de nouveaux types de liaison de ports et des connecteurs spécifiques facilitant la programmation. Ces nouveaux types de liaison, que l'on nomme *liaisons d'aspects* dans la suite, sont :

- *before/after service invocation* (**bSI**/**aSI**), il s'agit des points de jonction avant ou après la réception (resp. émission) d'une invocation de service pour port fourni (resp. requis),
- *before/after connection* (**bCon**/**aCon**),
- *before/after disconnection* (**bDiscon**/**aDiscon**).

Le listing 4.9 montre comment établir les liaisons représentées sur la figure 4.5. Similairement aux liaisons normales, de nouvelles constructions syntaxique telles que **bindbsi** <*port*>+ **to** <*port*> sont introduites. En plus du sucre syntaxique (**bind** ...**to** ...**glue**), il est possible de définir des connecteurs qui facilitent l'utilisation de ces liaisons.

```
pm := new PasswordManager
rng := new RandomNumberGenerator
l := new FileLogger

bind pm.Randomizer to rng.Generator glue {
   # ...
}

bindbsi pm.Generator, pm.Checker, rng.Generator to l.log glue {
   Target.log( "before " + si.port().name() + "." + si. selector() )
}
```

LISTING 4.9 : Définition d'une liaison d'aspects de type *bSI* en SCL

Dans le code d'un composant, il est possible d'utiliser des liaisons d'aspects sur les ports internes comme l'illustre le listing 4.10. Cela n'est absolument en contradiction avec le modèle boîte noire puisque les ports internes sont justement accessibles uniquement depuis l'implémentation de leur composant.

```
descriptor CDPasswordManager {
   providedport Generator

   def init {
      # ...
      bindbsi self to (new FileLogger).Logger glue {
         Target.log("internal CDPassword call of " + si.selector )
      }
   }

   # ...
}
```

LISTING 4.10 : Une liaison de type *bSI* sur un port interne en SCL

Les ports des connecteurs peuvent eux aussi être liés via des liaisons d'aspects puisque ce sont des composants. Tisser des aspects directement sur les ports des connecteurs ou des composants est tout à fait similaire.

Le tissage repose donc sur les liaisons d'aspects en SCL. Ainsi, il est possible de tisser dynamiquement des aspects en établissant des liaisons mais aussi d'en enlever en supprimant les liaisons.

Enfin, un conflit de tissage se produit lorsqu'un port possède deux liaisons d'aspect du même type. De même qu'en FAC, les conflits peuvent être résolus par l'architecte en spécifiant un ordre d'exécution entre les liaisons d'aspects en conflit. Le listing 4.11 montre un exemple de résolution (mot-clef **priority**) en spécifiant le niveau de priorité (un entier) pour chaque liaison du même type pour un port. Attention, deux liaisons du même type pour un même port doivent posséder une priorité qui ne peuvent être identiques.

```
pm := new PasswordManager
lf := new FileLogger
lbd := new BDLogger

bindbsi pm.Generator to lf.log priority 1 glue {
   # ...
}

bindbsi pm.Generator to lbd.log priority 2 glue {
   # ...
}
```

LISTING 4.11 : Résolution de conflits explicite entre des liaisons d'aspects via des niveaux de priorité

Afin d'éviter la résolution de conflits explicite systématiquement, nous avons introduit une règle de priorité implicite en SCL qui est que l'ordre entre les liaisons est le même que celui de la mise en place des liaisons. Ainsi, dans l'exemple du listing 4.11, préciser les priorités est inutile puisque la première liaison établie est bien celle qui est prioritaire.

4.3.4 Critique

SCL propose donc un modèle de points de jonction de type boîte noire comme la plupart des modèles industriels et FAC afin de garantir le découplage. SCL intègre aussi des liaisons d'aspects spécifiques pour les points de jonction qu'il supporte. Ces liaisons explicitent le couplage entre le code de base et le code transversal. Ensuite, les connecteurs de SCL et la souplesse apportée par le code glue permettent de séparer les préoccupations au mieux. La propriété de non-anticipation (*obliviouness*) est respectée puisque les liaisons d'aspects « s'attachent » directement sur les ports des composants. Un programmeur de composant ne doit pas explicitement rendre son composant aspectisable en le dotant d'interfaces particulières comme c'est le cas par exemple en FAC ou Fractal-AOP. Du point de vue de la dynamicité, le tissage est effectué dynamiquement et est réversible en SCL puisqu'il est complètement basé sur la connexion comme en Fractal-AOP et FAC. Du point de vue de l'intégration des aspects, FuseJ propose une solution comparable à celle de SCL bien que la gestion des conflits ne soit pas traitée. Par contre, du point de vue des composants, FuseJ est bien moins unifié que SCL car il utilise une classe Java standard pour implémenter un composant négligeant ainsi, dans le code, la notion de « port » par exemple. Le principal désavantage de SCL est actuellement le manque de langage de coupes. En effet, établir des liaisons d'aspect est bien plus fastidieux qu'utiliser un langage de coupes déclaratif qui permet au programmeur de décrire succinctement les coupes. Toutefois, il est possible de construire des connecteurs en SCL facilitant la mise en place des liaisons d'aspects en se basant sur des descriptions à base d'expression régulières.

4.4 Les propriétés observables des composants

Dans cette section, nous présentons une problématique générale des approches à composants concernant la possibilité d'établir, de façon non-anticipée, des connexions entre les composants basées sur les changements d'état de leurs propriétés. Cette problématique relève de la séparation des préoccupations car l'établissement de telles connexions entre les composants nécessite normalement que leur programmeur ait écrit du code spécifique à cet effet. Nous pensons que le code métier des composants ne doit pas contenir de code transversal permettant l'établissement de telles connexions. Il existe peu de solutions à ce problème dont la plus notable est certainement SuperGlue [McDirmid et Hsieh, 2006] que nous présentons brièvement avant de détailler celle que nous avons intégrée à SCL.

4.4.1 Problématique

Généralement les propriétés de composants se résument à un couple d'accesseurs (services permettant d'accéder ou modifier une valeur interne du composant nommés par convention `get` et `set`)

comme c'est le cas en CCM, Fractal ou encore ArchJava. Dans ces conditions, il est difficile d'établir des connexions basées sur les changements d'état des composants. Pourtant, cela est particulièrement utile notamment dans le cas des interactions entre les entités *modèles* et les entités *vues* d'une application architecturée selon le modèle MVC [Krasner et Pope, 1988].

De nombreuses techniques existent pour déclencher des traitements en réaction à des changements d'état comme le schéma de conception Observateur [Gamma *et al.*, 1995] ou encore les *attachements procéduraux* [Minsky, 1975] qui permettent d'attacher des procédures à l'accès d'un attribut afin qu'elles soient automatiquement exécutées lors de chaque accès à cet attribut. Plus généralement, cette problématique relève du protocole de communication *publish/subscribe* [Eugster *et al.*, 2003] qui est particulièrement utile et permet un bon découplage entre les entités comme énoncé dans [Garlan et Shaw, 1993] : *"The main invariant in this style is that announcers of events do not know which components will be affected by those events"*. Toutefois, la mise en place de ce protocole de communication doit être effectuée par le programmeur des composants ce qui contredit le principe de non-anticipation à deux niveaux :

Côté émetteur. Le programmeur d'un composant qui notifie ses changements d'état, doit explicitement écrire du code relatif à la gestion des écouteurs (ajout/retrait) et leur notification. C'est le cas dans le modèle *Javabeans* (*cf.* listing 2.4) mais aussi dans la plupart des autres modèles comme CCM où il faut équiper le composant d'une source d'événements et lever les événements explicitement dans le code du composant ou en Fractal où il faut utiliser une interface cliente optionnelle à travers laquelle on invoque les services de notification.

Côté récepteur. Il peut être nécessaire qu'un composant soit programmé de façon particulière pour recevoir des notifications. Par exemple, en CCM, le composant doit être équipé d'un puits d'événements dont le type est compatible avec ceux à recevoir.

Dans la plupart des cas, l'utilisation d'un adapteur [Gamma *et al.*, 1995] permet de s'affranchir des contraintes du côté récepteur alors que cela est impossible pour celles du côté émetteur.

En résumé, un langage à composants doit offrir selon nous :

- la possibilité de déclarer l'état exporté par un composant lors de sa définition, sans compromettre l'encapsulation c'est-à-dire révéler la façon dont est stocké cet état en interne,
- la possibilité de connecter des composants en se basant sur les changements de valeurs de leurs propriétés, sans qu'ils aient été programmés spécifiquement pour supporter l'abonnement et la notification (respect de la non-anticipation).

4.4.2 Les solutions existantes

La notion de propriété existe dans de nombreux langages de programmation à objets comme Delphi[5] ou plus récemment C# [Hejlsberg *et al.*, 2003]. Dans ces langages, il existe des constructions syntaxiques particulières pour déclarer des propriétés comme le montre le listing 4.12 et les utiliser simplement c'est-à-dire en masquant les accesseurs qui sont pourtant utilisés lors de l'exécution.

[5] http://www.borland.com/fr/products/delphi/index.html

```
class TimeStamp
{
   // Nombre de seconde depuis le 1er janvier 1970 à 00:00:00 GMT (Norme)
   private double s;

   public int Time {  // Propriété Time accessible en lecture écriture
      get {
         return s;
      }
      set {
         s = value;
      }
   }

   public int Year {  // Propriété Year accessible en lecture
      get {
         return computeYear(s);
      }
   }

   private int computeYear(double s ) {
      // ...
   }
}

class Test
{
   public static void Main() {
      TimeStamp ts = new TimeStamp();
      ts.Time = 1234567.89;      // Modification de la valeur de la propriété Time
      Console.WriteLine(ts.Year);   // Accès à la valeur de la propriété Year
   }
}
```

LISTING 4.12 : Les propriétés en C#

L'idée de doter les propriétés de comportement de notification est issue du modèle *Javabeans*. Dans les langages actuels les propriétés ne notifient pas automatiquement leurs changements d'état et cela doit être mis en place par le programmeur ce qui n'est pas une solution envisageable du point de vue de la non-anticipation. Ce code de notification étant fastidieux et récurrent, des constructions syntaxiques simplifient son écriture comme c'est le cas en C# [Hejlsberg *et al.*, 2003] avec les *delegates*, mais cela n'est toujours pas satisfaisant.

Dans le domaine des composants, le langage de programmation SuperGlue [McDirmid et Hsieh, 2006] adresse ce problème. En SuperGlue, un composant est soit une instance d'un *atom* (implémenté en Java), soit d'un *compound* (implémenté en SuperGlue). Dans tous les cas, un composant est constitué de *signals* exportés ou importés. Un signal représente une valeur typée variant dans le temps (*time-varying values*). Le listing 4.13 montre la déclaration de deux atoms, leur instanciation et le code de deux connexions. La connexion est réalisée via l'opérateur d'égalité. La ligne 13 de cet exemple, signifie que lors de toute modification de valeur du signal `temperature`, la valeur du signal `text` sera automatiquement mise à jour. La deuxième connexion présentée dans ce listing (*cf.* ligne 16) utilise la syntaxe classique d'un **`if`** mais la sémantique est différente. Les conditions sont des *gardes* et dès que l'une d'elles est vérifiée (à tout instant de l'exécution du programme), l'action correspondante est exécutée.

```
1  atom Thermometer {
       export temperature : int;
3  }

5  atom Label {
       import text : String;
7      import color : Color;
   }
9
   let model = new Thermometer;
11 let view = new Label;

13 view.text = " " + model.temperature + " C";  // une connexion en SuperGlue

15 // Autre exemple de connexion
   if (model.temperature > 30) view.color = red;
17 else if (model.temperature < 0) view.color = blue;
   else view.color = black;
```

LISTING 4.13 : Exemple de code en SuperGlue

Ce rapide survol de SuperGlue permet de constater que ce langage est vraiment spécialisé pour réaliser des connexions basées sur les changements d'état des propriétés. D'ailleurs, la connexion de composants repose uniquement sur la liaison de *signals*. Le prototype actuel de SuperGlue utilise le schéma de conception Observateur [Gamma *et al.*, 1995] pour détecter les changements d'état des signaux.

SCL présente la possibilité d'établir des connexions de type requis/fourni et de type aspects. Dans

la section suivante, nous présentons une extension de SCL afin de réaliser aussi des connexions basées sur les changements d'état sans qu'elles aient été prévues par les programmeurs des composants.

4.4.3 Notre solution en SCL basée sur les aspects

Afin de répondre à la problématique présentée ci-dessus, nous avons intégré à SCL la notion de *propriété observable* inspirée du modèle *Javabeans* et qui s'apparente à la notion de signal de SuperGlue. En SCL, une propriété est déclarée par le programmeur d'un composant qui est alors automatiquement doté de deux ports : le *port d'accès* et le *port de notification* de la propriété.

Le port d'accès est un port fourni du composant exportant les accesseurs de la propriété (*getter* et *setter*). Comme c'est le cas dans les autres langages comme C# (*cf.* listing 4.12), la représentation interne de la valeur d'une propriété reste encapsulée et masquée dans l'implémentation du composant.

Le port de notification d'une propriété est un port requis à travers lequel le composant invoque des services de notification lors des modifications de la propriété. Par exemple, le service de notification `nac` (*Notify After Change*) est invoqué après la modification de la valeur de la propriété avec l'ancienne et la nouvelle valeur de la propriété comme paramètres. Les propriétés de SCL sont toutes *liées* au sens du modèle *Javabeans* sans effort du programmeur.

La figure 4.6 montre un exemple d'utilisation des propriétés dans une application *client de messagerie instantanée* (*chat client*). Dans cet exemple, le composant CHATCLIENT possède une propriété nommée `ChatText` dont la valeur contient le texte des messages envoyés et reçus. Le mécanisme de connexion standard de SCL est utilisé pour connecter le port de notification de cette propriété au port `Display` du composant CHATCLIENTGUI.

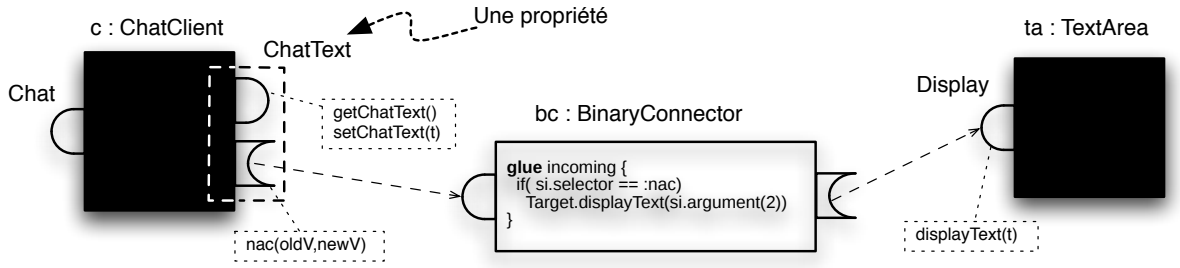

FIG. 4.6 : Utilisation des propriétés pour établir des connexions basées sur les changements d'état

Le listing 4.14 montre la déclaration du composant CHATCLIENT et notamment celle de la propriété `ChatText`. La déclaration des propriétés s'effectue via le mot-clef **`property`** dans notre langage. Les accesseurs de la propriété sont des services définis par le programmeur du composant[6]. Dans cet exemple, les accesseurs de la propriété `ChatText` accèdent au port interne `ct`.

[6]Cette tâche est souvent facilitée par du sucre syntaxique dans les langages lorsque la propriété n'accède qu'à un seul attribut.

```
interface IChat { join(serverAddress); leave; send(message) }

descriptor ChatClient {
   providedport Chat, IChat

   intern requiredport ct
   intern requiredport nickname
   intern requiredport socket

   property ChatText {
      # definir le port d'accès permet de choisir le nom du port et ceux des accesseurs
      # une convention pourrait être utilisée pour faciliter la tâche du programmeur
      accessport AccessCT, {getChatText; setChattext(v)}
   }

   def getChatText { return ct.getText }
   def setChattext(v) { ct.setText(v) }

   def init {
      bind ct to new Text
      # ...
   }

   # exécuté lorsque l'utilisateur envoie un message
   def send(message) {
      socket.send("<" + nickname.getValue() + "> " + message)
   }

   # exécuté lors de la réception d'un message à travers la socket
   def receive(message) {
      ct.concat("\n" + message)
      # attention, cette dernière instruction a modifié la valeur de la propriété ChatText
      # bien que l'accesseur en écriture cette propriété n'ait pas été utilisé
   }
}
```

LISTING 4.14 : Déclaration de propriété en SCL

Dans le code du service `receive`, un service est invoqué via `ct` et l'exécution de ce service peut modifier le sous-composant de telle sorte que la valeur de la propriété `ChatText` sera elle aussi modifiée. L'interprète de SCL doit détecter les changements de valeur des propriétés pour effectuer les notifications. La détection de ces changements peut être mise en place via les liaisons d'aspects présentées dans la section précédente. La figure 4.7 montre une possible organisation interne du composant `ChatText` permettant d'effectuer automatiquement les notifications de changement d'état. Bien évidemment la mise en place de cette organisation est effectuée par le compilateur ou l'évaluateur puisque la notification ne doit pas reposer sur le programmeur. De plus, si un programmeur souhaite effectuer lui-même une notification, il peut le faire directement et simplement en invoquant le service `nac` à travers le port de notification de la propriété.

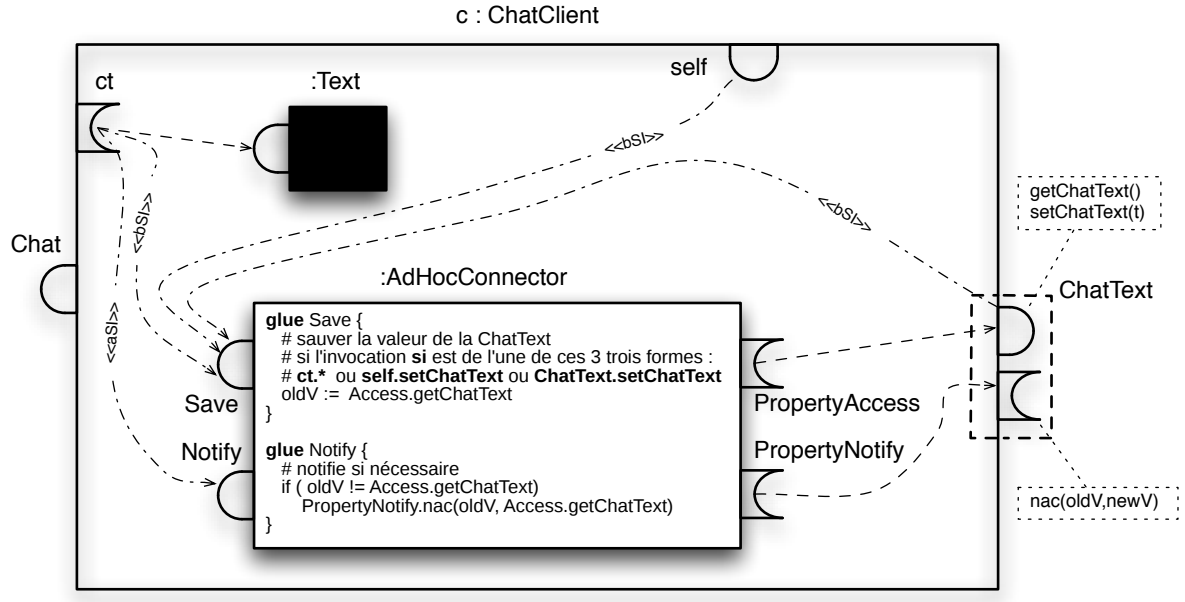

FIG. 4.7 : Une organisation interne détectant les changements de valeurs de propriété qui peut être générée par un compilateur ou évaluateur

Critique. Contrairement à SCL, SuperGlue ne supporte que les connexions basées sur les changements d'état des composants. L'ajout de propriétés observables en SCL permet d'offrir une possibilité d'assemblage supplémentaire à l'architecte, augmentant ainsi les possibilités de réutilisation des composants, sans déroger au principe de non-anticipation. Toutefois, le programmeur de composants peut toujours fournir directement des accesseurs, sans déclarer de propriété. Cela constitue actuellement une limitation de cette extension de SCL qu'il faudrait lever en imposant la déclaration systématique de propriétés pour exporter de l'état.

4.5 Conclusion

Dans ce chapitre, nous avons présenté deux extensions de SCL.

La première concerne la séparation des préoccupations en utilisant des techniques issues de la programmation par aspects. Nous avons choisi d'adopter une approche symétrique dans laquelle les aspects sont représentés et traités de la même façon que les composants standards. Notre idée est que le code réutilisable (métier ou transversal) doit être défini dans les composants et l'utilisation d'un composant dans un contexte donnée doit s'effectuer via l'établissement de connexions. C'est ainsi que tout composant écrit en SCL peut être utilisé de façon « normale » ou transversale via le mécanisme de connexion qui a été enrichi par de nouveaux types de liaisons appelées liaisons d'aspects.

La deuxième extension de SCL présente la notion de *propriété observable* afin de permettre la définition de connexions basés sur les changements d'états. Ces connexions reposent sur la notification automatique des changements d'états des propriétés sans anticipation de la part des programmeurs. Peu d'approches à composants offrent un tel mécanisme alors qu'il nous semble vital pour la définition de composants réutilisables bien qu'il soit difficile à mettre œuvre efficacement comme cela est le cas pour SuperGlue [McDirmid et Hsieh, 2006].

CHAPITRE 5

Prototypes de SCL

Make it work.
Make it work right.
Make it work right and fast.

Edsger DIJKSTRA, Donald KNUTH, C.A.R. HOARE

Yes, we are old dudes who know Smalltalk.
Listen and learn ! :-)

Mike MILINKOVICH
Executive Director of the Eclipse Foundation, 2006

Préambule

Ce chapitre présente deux prototypes de SCL. *Le premier est implémenté en Smalltalk ou plus exactement en Squeak qui est une implémentation de Smalltalk. Nous présentons d'abord ce premier prototype, qui est le plus abouti, à travers nos choix de conception et des exemples de code. Nous présentons aussi une ébauche d'environnement graphique pour développer en* SCL. *Ensuite, le deuxième prototype, écrit en Ruby, est rapidement présenté avant de conclure ce chapitre.*

5.1 Pourquoi Smalltalk ?

UN premier prototype de SCL a été réalisé en Squeak [Ingalls *et al.*, 1997]. Squeak[1] est une implémentation libre [2], moderne et portable du langage et de l'environnement Smalltalk [Goldberg et Robson, 1989]. Smalltalk est un langage à objets réflexif et dynamiquement typé. Les langages dynamiquement typés offrent une flexibilité et des qualités encore reconnues [Nierstrasz *et al.*, 2005]. Les principales caractéristiques du modèle objet de Smalltalk sont l'héritage simple et la notion de méta-classe implicite c'est-à-dire qu'une métaclasse est créée automatiquement pour chaque classe du système sans que le programmeur n'ait à la définir. Du fait de la réflexivité, Smalltalk est un langage de programmation uniforme (« tout est objet ») et ouvert ce qui facilite l'écriture d'extensions. Squeak [Ingalls *et al.*, 1997; Black *et al.*, 2007] est une implémentation de Smalltalk écrite en Smalltalk[3] ce qui facilite les extensions de la machine virtuelle Squeak.

Outre les qualités de Smalltalk et de Squeak, nous avons aussi constaté que la majorité des prototypes actuels de langages à composants sont des extensions de Java. Il semble donc intéressant de proposer une alternative dans un environnement différent afin de mieux distinguer les spécificités des langages à composants de celles du langage Java.

5.2 Choix d'implémentation

L'implémentation d'un langage de programmation nécessite le développement de la chaîne de compilation et/ou d'interprétation allant de l'analyseur de code source (*parseur*) jusqu'au générateur de code machine. Bien évidemment, il existe de nombreux outils facilitant la construction de cette chaîne comme les *compiler compiler* qui permettent de générer le code source de *parseurs*, d'interpréteurs ou de compilateurs à partir de descriptions (syntaxiques et sémantiques) d'un langage de programmation. D'ailleurs, SmaCC[4] (*Smalltalk Compiler Compiler*) a été utilisé dans les premières versions du prototype de SCL pour générer un *parseur* à partir de la description d'une grammaire. Cette technique implique de modifier la grammaire, de générer un nouveau *parseur* et surtout de modifier l'interprète de l'arbre syntaxique à chaque fois que la syntaxe subit une évolution. Du fait du caractère évolutif de notre prototype, nous avons adopté une approche plus souple et bien mieux adapté aux évolutions constantes.

Nous avons décidé d'implémenter SCL en utilisant directement les constructions syntaxiques et les mécanismes de Smalltalk. Cette technique est particulièrement utilisée actuellement pour implémenter des *Domain-Specific Languages* (DSL) [Mernik *et al.*, 2005]. Elle permet de s'affranchir de l'écriture d'un *parseur* et d'un interpréteur d'arbres syntaxiques mais elle rend toutefois le langage implémenté complètement dépendant du langage hôte (*internal to a base langage*).

Smalltalk facilite grandement la définition de DSL notamment grâce à sa syntaxe basée sur des sélecteurs à mots-clefs mais aussi grâce à la notion de *bloc* (fermeture lexicale) :

[1] http://www.squeak.org
[2] http://www.gnu.org/philosophy/free-sw.html
[3] Plus exactement en un sous-ensemble de Smalltalk nommé Slang.
[4] http://www.refactory.com/Software/SmaCC/

> « *Smalltalk seems much more a notation, that can do nothing but build domain specific languages. [...] When you create a domain specific language in Smalltalk, your code never looks different than code provided by the compiler writer himself, it's one syntax to rule them all. [...] Smalltalk shares this trait with languages such as Lisp and Scheme, truly growable languages that put you, the programmer, in charge of what language you want, not some compiler writer who might take six more years to add some feature you just got to have now. When you need a new language feature in Smalltalk, you simply add it. [...] Having a succinct notation for the delayed evaluation of code, and allowing that code to be placed into variables and passed around, is all one needs to build virtually any domain specific language they can dream up.* » [Leon, 2006]

Un autre avantage à utiliser directement les constructions de Smalltalk est l'accès direct aux outils de développement Smalltalk (*browser*, *debugger*, etc.) avec du code écrit dans le nouveau langage.

5.3 Amorçage de l'implémentation

L'amorçage (*bootstrap*) de l'implémentation de SCL repose sur la représentation des composants et des descripteurs. La figure 5.1 montre une implémentation « idéale » du noyau de SCL en Squeak reposant sur les deux classes `Component` et `ComponentDescriptor`. La classe `Component` est un descripteur de composants instance de la méta-classe `ComponentDescriptor` qui décrit les descripteurs de composants.

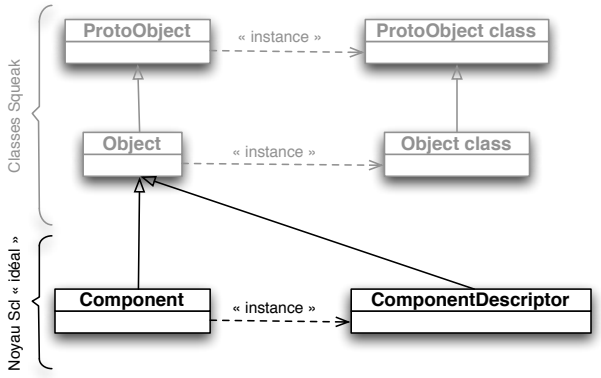

FIG. 5.1 : Implémentation « idéale » du noyau de SCL

Nous pensons que cette implémentation est « idéale » dans le sens où elle n'intègre que très peu de mécanismes existants en Smalltalk. En effet, la méta-classe `ComponentDescriptor` hérite de la classe `Object` (et non de la classe `Class`) ce qui permet d'implémenter uniquement les comportements propres à SCL sans hériter ceux spécifiques à Smalltalk comme l'héritage entre les classes par exemple. Toutefois, nous n'avons pas retenu cette implémentation « idéale » car :

– les méta-classes sont implicites en Smalltalk et créées automatiquement. Il n'est donc pas possible de construire une méta-classe héritant de la classe `Object` sans utiliser une extension de

Smalltalk telle que MetaclassTalk [Bouraqadi, 1999] ;
- il faudrait réécrire un mécanisme d'instanciation (allocation mémoire, etc.) pour SCL puisqu'il est défini en Squeak dans la classe `Behavior` ;
- les outils standards de l'environnement Smalltalk manipulent des classes et ne pourraient donc pas être utilisés avec des descripteurs de composants qui ne sont pas des classes.

La figure 5.2 montre l'implémentation actuelle du noyau de SCL utilisant la méta-classe `Component class` pour représenter les descripteurs de composants. Cette méta-classe hérite indirectement de la classe `Class` ce qui facilite le développement (l'instanciation est disponible) ainsi que l'incorporation dans l'environnement Smalltalk (les descripteurs de composants sont des classes).

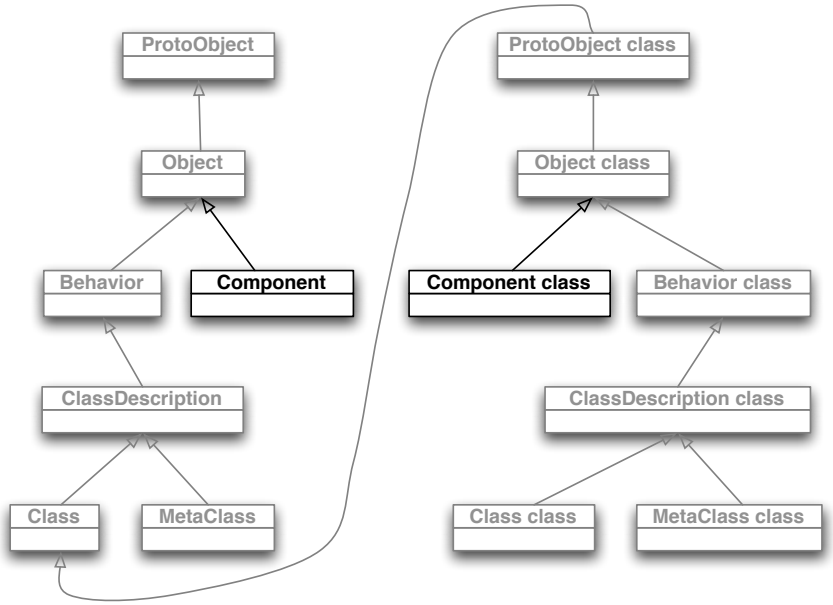

FIG. 5.2 : Implémentation actuelle du noyau de SCL

5.4 Architecture générale du prototype

La figure 5.3 montre l'architecture générale de l'implémentation de SCL. Cette architecture comprend quatre « niveaux », chacun étant constitué d'éléments différents :

1. Le premier niveau est celui des classes de Squeak telles que `Object` ou `Object class`.
2. Le second niveau comprend les classes relevant de la mise en œuvre de SCL telles que `Component` ou `Component class`. Dans la suite, nous décrivons plus précisément les classes de ce niveau et leurs relations.
3. Le troisième niveau est celui des descripteurs de composants. C'est dans ce niveau qu'un programmeur SCL écrit des descripteurs de composants. Nous verrons dans la suite des exemples de définition de descripteurs en SCL.

4. Enfin, le quatrième niveau est celui des instances c'est-à-dire des composants qui peuvent être assemblés. Le composant `SclBuilder` est important comme nous le verrons dans la suite.

Un utilisateur du langage SCL (programmeur ou architecte) ne doit accéder qu'aux éléments des niveaux 3 et 4 de cette architecture. C'est pourquoi nous allons nous efforcer de masquer autant que possible les niveaux 1 et 2 aux utilisateurs de SCL comme nous le verrons à travers les exemples de code.

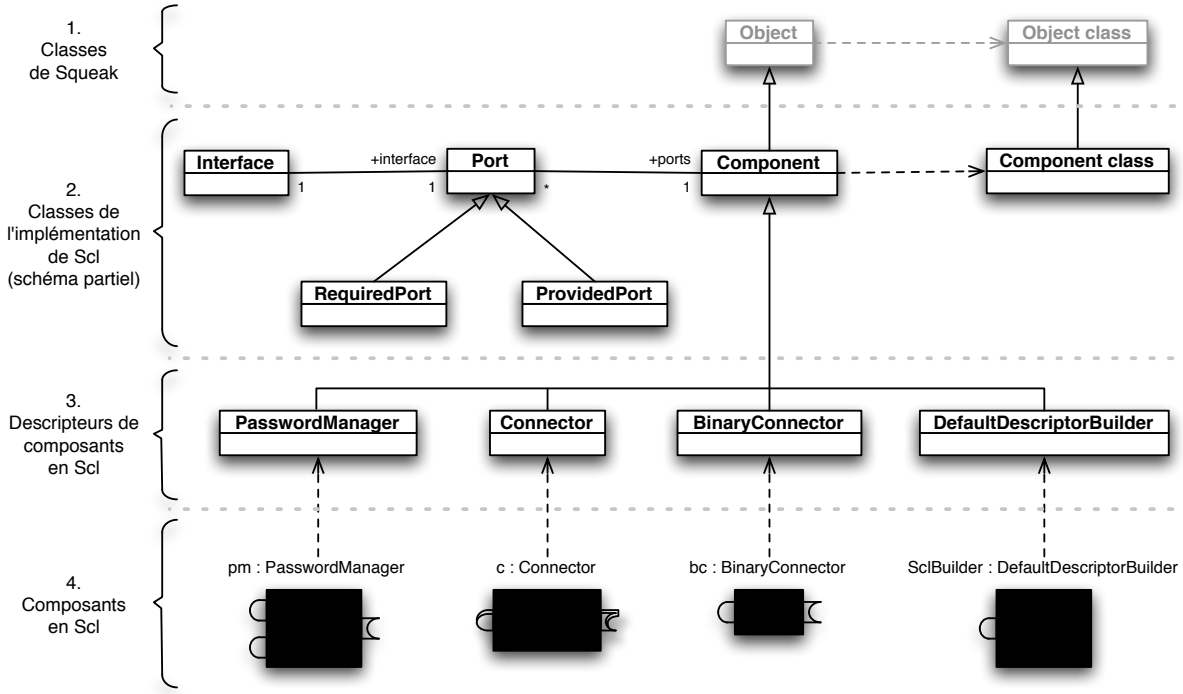

FIG. 5.3 : Les quatre « niveaux » de l'implémentation de SCL en Smalltalk

5.5 Le modèle implémenté

Le deuxième niveau de cette architecture est détaillé par le schéma UML présenté à la figure 5.4. Cette figure présente le modèle de SCL où la classe `Component` (qui représente les compsants) est centrale. Cette classe est une instance de la méta-classe `Component class` qui introduit un dictionnaire de ports de façon analogue à la classe `Behavior`, dont elle hérite (*cf.* figure 5.2), qui introduit un dictionnaire de méthodes. Un composant est donc constitué de ports (`Port`) appartenant éventuellement à une collection de ports (`MultiPort`). Un port est décrit par une interface (`Interface`) qui comprend un ensemble de signatures de service (`Service`). On distingue les ports requis (`RequiredPort`) et les ports fournis (`ProvidedPort`) ainsi que les deux ports particuliers que sont `self` (`SelfPort`) et `default` (`DefaultPort`). Un port fourni peut être associé à un service glue

(`GlueService`) qui est un service particulier du composant. Les liaisons sont représentées par des associations au niveau des ports :

- les liaisons standards permettent de lier un `RequiredPort` à un `ProvidedPort`,
- les liaisons de délégation permettent de lier deux ports de même nature.

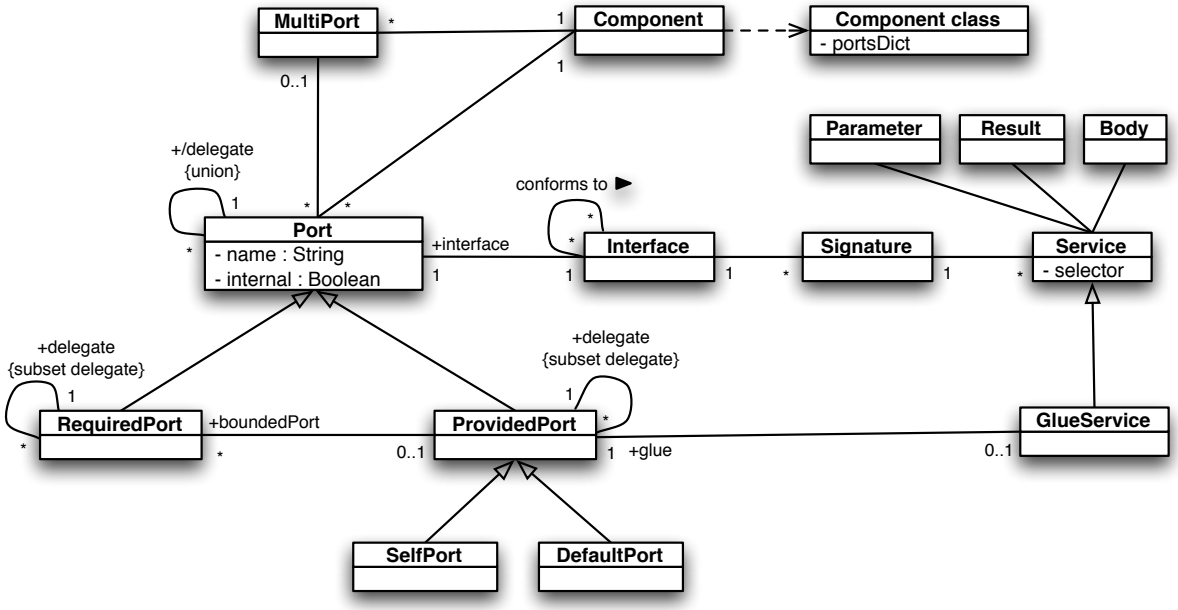

FIG. 5.4 : Schéma UML du modèle SCL implémenté en Smalltalk

Le listing 5.1 montre un exemple de définition d'un descripteur de composant en SCL (il s'agit de celui décrit dans le listing 3.1). Dans ce code, le composant `SclBuilder` (*cf.* figure 5.3) est utilisé. Ce composant est un singleton [Gamma *et al.*, 1995] permettant la construction de descripteurs de composant tout masquant aux programmeurs les détails d'implémentation. La déclaration de chaque port comprend :

- son nom qui est un symbole en Smalltalk et commence donc par le caractère #,
- son interface qui est une collection[5] de symboles correspondants à des sélecteurs.

Ensuite, les codes source (uniquement les points clés) des services sont montrés. Le premier service (`getRandomCharacterWithLetters:`) est interne et son code contient une invocation via le port requis `Randomizer` du composant. Nous reviendrons dans la suite sur la syntaxe de l'invocation de service qui est la même que celle de l'envoi de message en Smalltalk (le caractère espace). Les codes source des autres services montrent plusieurs exemples d'invocation dont certaines à travers le port `Self`. Finalement, ce descripteur est instancié via la primitive **`newC`**. Nous verrons dans la section suivante (*cf.* section 5.6) pourquoi cette primitive est nommée **`newC`** et non `new`. Rappelons toutefois que l'instanciation d'un descripteur retourne une référence sur le port nommé `Default` du composant nouvellement créé (*cf.* choix 8 page 86).

[5] Squeak a introduit la syntaxe avec des accolades pour permettre l'écriture de tableaux de façon littérale.

```
(SclBuilder newDescriptor: #PasswordManager)          "Définition d'un descripteur"
   requiredPort: #Randomizer -> {#getRandomNumber} ;
   providedPort: #Generator -> { #generatePwd: . #generateADigitsOnlyPwd: } ;
   providedPort: #Checker -> {#isValid:}.

PasswordManager>>getRandomCharacterWithLetters: b
"retourne un chiffre choisi aléatoirement"
"b indique si le caractère peut être une lettre minuscule"
   "..."
   i:= Randomizer getRandomNumber.                    "invocation via le port Randomizer"
   "..."

PasswordManager>>generatePwd: size                    "Définition du service generatePwd"
"Génère un mot de passe dont la taille est égale à size"
"à partir de lettres minuscules et de chiffres choisis aléatoirement"
   | generatedPwd i |
   "..."
   i := Self getRandomCharacterWithLetters: true.     "invocation interne via le port Self"
   "..."
   (Checker isValid: generatedPwd)                    "invocation via le port Checker"
      ifTrue: [ ^generatedPwd ]
      ifFalse: [ ^Generator generatePwd: size ]

PasswordManager>>generateADigitsOnlyPwd: size
"Génère un mot de passe de taille size avec des chiffres uniquement"
   "..."
   i := Self getRandomCharacterWithLetters: false.    "invocation interne via le port Self"
   "..."

PasswordManager>>isValid: aPwd
"retourne un booléen indiquant si aPwd est valide en vérifiant sa difficulté par exemple"
   "..."

pm := PasswordManager newC.
```

LISTING 5.1 : Définition partielle du descripteur PASSWORDMANAGER en SCL

5.6 L'intégration des objets de base

L'intégration des objets de base nécessite de modifier le mécanisme d'instanciation de Smalltalk. Lors de l'instanciation d'une classe, il faut retourner une référence sur le port nommé `Default` de l'instance nouvellement créée et non sur l'instance directement. En toute généralité, il aurait fallu redéfinir la méthode `basicNew` de la classe `Behavior` qui permet de créer une instance mais cela est impossible en Squeak car elle est une primitive de la machine virtuelle. Notons que la redéfinition de la méthode `new` définie dans la classe `Behavior` ne permet pas d'atteindre notre objectif car de nombreuses classes redéfinissent cette méthode et utilisent directement la méthode `basicNew`. Pour implémenter notre mécanisme d'instanciation, nous avons donc défini les deux méthodes suivantes :

- `newC` (contraction de *newComponent*) dans la classe `Object class` qui permet d'instancier un composant à partir d'une classe et retourne une référence vers le port `Default` de ce composant,
- `defaultPort` qui permet d'obtenir un objet représentant le port par défaut de n'importe quel objet Smalltalk. L'interface de ce port est initialisée avec l'ensemble des signatures (un sélecteur contient l'arité de la méthode en Smalltalk) de messages auxquels cet objet peut répondre.

```
"Création d'un composant Scl à partir d'une classe Smalltalk"
Object class>>newC
   ^ self new defaultPort

"Accesseur au port Default d'un objet"
Object>>defaultPort
   ^ (Port newNamed: #Default provides: self allSelectors)
      component: (self) ;
      yourself
```

Dans l'exemple suivant, nous montrons des utilisations de ces méthodes pour intégrer les objets de base Smalltalk sous la forme de composants en SCL :

```
"Instanciation de la classe OrderedCollection"
col := OrderedCollection newC.
"la variable col contient ici une référence un port fourni nommé Default"
col size.            "invocation du service size"

"les entiers sont considérés automatiquement comme des instances"
"de la classe SmallInteger dont odd est une méthode"
unComposantEntier := 1 defaultPort.
"la variable unComposantEntier contient une référence sur un port Default"
unComposantEntier odd   "invocation du service odd"
```

Dans le cas des littéraux comme les entiers ou les chaînes de caractères, le programmeur SCL doit stocker la référence vers leur port par défaut en utilisant la méthode `defaultPort`. En effet, les littéraux ne profitent pas directement de la méthode `newC` définie dans la classe `Object class` car ils ne sont jamais vraiment instanciés mais traités spécifiquement par la machine virtuelle Squeak. Afin d'uniformiser la vision offerte au programmeur SCL, nous avons aussi défini la méthode `newC` dans la classe `Object` de la façon suivante :

```
Object>>newC
   ^self defaultPort
```

En résumé, le programmeur SCL doit toujours utiliser la primitive[6] **`newC`** pour instancier des descripteurs et des classes ou pour obtenir une référence sur le port par défaut d'un objet :

[6] Nous utilisons le terme « primitive » et non celui de « méthode » pour mettre l'accent sur le fait que le code ce mécanisme doit être considéré comme « opaque » par le programmeur SCL puisqu'il relève du second niveau dans l'architecture de notre prototype.

```
pm := PasswordManager newC.

c := OrderedCollection newC.
c size

entier := 28 newC.
entier odd
```

5.7 L'invocation de service

La syntaxe de l'invocation de service est identique à celle de l'envoi de message en Smalltalk. Toutefois, nous avons bien utilisé les algorithmes 1 et 2 (*cf.* page 103) détaillés dans le chapitre 3. Pour illustrer le fonctionnement de l'invocation de service, plaçons-nous dans le contexte de l'exemple présenté sur la figure 3.5 où une instance `pm` du descripteur PASSWORDMANAGER est connectée (par une liaison) à un composant `rng` instance du descripteur RANDOMNUMBERGENERATOR. Dans ce contexte, considérons le traitement d'une invocation de service émise via le port `Randomizer` du composant `pm`. L'objet représentant le port `Randomizer` ne possède pas de méthode nommée `getRandomNumber` et sa méthode `doesNotUnderstand:` est exécutée à la place[7]. Dans la classe représentant les ports (`Port`), nous avons redéfini cette méthode pour qu'elles appelle la méthode `invoke:` du même port qui traite correctement l'invocation de service. Ensuite, nous avons redéfini cette méthode `invoke:` dans les sous-classes de `Port` afin que chacune traite spécifiquement les invocations qu'elle reçoit. Les listing 5.2 et 5.3 montrent le code des méthodes :

- `doesNotUnderstand:` qui est redéfinie dans la classe `Port` et qui transmet les envois de message à la méthode `invoke:` en les transformant en invocation de service préalablement,
- `invoke:` qui est la méthode de traitement des invocations de service que ce soit pour un port requis ou pour un fourni car la différence de traitement est encapsulée dans la méthode `computeReceiverObject`,
- `computeReceiverObject` est donc définie de façon abstraite dans la classe `Port` et redéfinie dans les classes `RequiredPort` et `ProvidedPort` de façon spécifique.

Cette technique d'implémentation présente l'avantage de proposer une syntaxe simple pour les invocations de service. En effet, il est tout à fait possible d'utiliser en SCL la syntaxe suivante pour l'invocation de service :

```
aPort invoke: #selector with: { arg1 . arg2 }
```

Cette syntaxe est rendue possible par la définition de la méthode suivante dans la classe `Port` :

```
Port>>invoke: selector with: args
   ^ self invoke: (ServiceInvocation selector: selector arguments: args)
```

[7] Il s'agit du mécanisme de base en Smalltalk lorsqu'un objet ne possède pas de méthode correspondant au sélecteur du message qu'il reçoit.

```
"Code de la méthode doesNotUnderstand: redéfinie dans la classe Port"
Port>>doesNotUnderstand: aMessage
   "l'envoi de message est transformé en invocation de service"
   "et passé en argument de l'envoi du message invoke à self"
   ^self invoke: (ServiceInvocation copy: aMessage sentThrough: self)

"Code de la méthode invoke: définie dans la classe Port"
Port>>invoke: aServiceInvocation
   | res receiverObject |

   "si l'interface du port ne contient pas le sélecteur demandé"
   (self interface selectors includes: aServiceInvocation selector) ifFalse: [
      "alors on laisse le traitement d'erreur s'effectuer"
      ^super doesNotUnderstand: aServiceInvocation asMessage
   ].

   "Calcul de l'objet auquel transmettre cette invocation ?"
   receiverObject := self computeReceiverObject

   "l'invocation est transmise via les liaisons d'aspects bsi"
   self processBeforeServiceInvocationBindings: aServiceInvocation.

   "l'invocation est transmise à l'objet receveur calculé"
   res :=  aServiceInvocation sendTo: receiverObject.

   "avant de retourner le résultat de l'invocation, on vérifie que la référence retournée"
   "est une référence sur un objet représentant un Port"
   "et pas sur un objet représentant un composant par exemple"
   "car dans ce cas, le résultat est fixé au port courant"
   (res isKindOf: Port) ifFalse: [ res := self ].

   "le résultat est stocké dans l'invocation"
   aServiceInvocation result: res.

   "l'invocation est transmise via les liaisons d'aspects asi"
   self processAfterServiceInvocationBindings: aServiceInvocation.

   "le résultat est finalement retourné"
   ^res

"La méthode computeReceiverObject est abstraite dans la classe Port"
Port>>computeReceiverObject
   self subclassResponsability
```

LISTING 5.2 : Méthodes de la classe `Port` permettant le traitement des invocations de services

```
"Redéfinition de la méthode computeReceiverObject dans la classe RequiredPort"
RequiredPort>>computeReceiverObject
   (self isBound) ifTrue:[
      "le port est lié sera le receveur"
      ^ self boundPort
   ] ifFalse: [
      (self isDelegated) ifTrue:[
         "le port délégué sera le receveur"
         ^ self delegate
      ] ifFalse: [
         "stoppe l'exécution et lève une exception"
         self error: self name, ' : not bound and not delegated'
      ]
   ]

"Redéfinition de la méthode computeReceiverObject dans la classe ProvidedPort"
ProvidedPort>>computeReceiverObject
   (self isDelegated) ifTrue:[
      "le port délégué sera le receveur"
      ^ self delegate
   ] ifFalse: [
      "l'objet représentant le composant sera le receveur"
      ^ self component
   ]
```

LISTING 5.3 : Méthodes permettant le traitement des invocations de service dans les classes `RequiredPort` et `ProvidedPort`

Toutefois, cette syntaxe est assez lourde à l'utilisation et c'est pourquoi nous préférons celle de l'envoi de message même si :

- elle prête à confusion avec l'envoi de message lors de la lecture du code au premier abord,
- elle est sujette à des conflits. Par exemple, si le sélecteur d'une invocation de service correspond à un sélecteur d'une méthode du port receveur, alors cette méthode est exécutée alors qu'il faudrait réaliser une invocation de service à destination d'un composant. Pour limiter cet inconvénient, les noms des méthodes dans la classe des ports pourraient être préfixés.

5.8 Les liaisons et les connecteurs

Comme nous l'avons vu ci-dessus, les liaisons sont mises en place au niveau des ports. En fonction du type de la liaison et de la nature du port, les invocations de services sont traitées différemment. La primitive **`bindTo:`** permet de mettre en place des liaisons de délégation et des liaisons standards (*cf.* figure 5.4). De façon analogue, il existe des primitives spécifiques pour les liaisons d'aspects : **`bindBSITo:`** et **`bindASITo:`**.

Le listing 5.5 présente des utilisations de connecteurs. Les connecteurs sont paramétrables par

```
pm := PasswordManager newC.
rng := RandomNumberGenerator newC.

"mise en place d'une liaison entre les ports Randomizer et Generator"
(pm port: #Randomizer) bindTo: (rng port: #Generator).

"invocation de service à travers le port Default de pm"
pm generatePwd: 6 newC
```

LISTING 5.4 : Une liaison de port en SCL

du *code glue* qui est représenté par un *bloc* en Smalltalk (délimités syntaxiquement par des crochets). Un *bloc* réifie une fermeture lexicale.

Il est possible de définir ses propres connecteurs en SCL. Cela nécessite d'écrire un descripteur et de prévoir son paramétrage via des *blocs*.

5.9 Les propriétés

Un composant peut déclarer des propriétés observables en SCL (*cf.* section 4.4.3). Le listing 5.6 montre la déclaration du composant `ChatClient` que nous avions présentée dans le listing 4.14. Ce composant est doté d'une propriété nommée `ChatText`.

```
(SclBuilder newDescriptor: #ChatClient)
   internalRequiredPort: #ct ;
   internalRequiredPort: #nickname ;
   internalRequiredPort: #socket ;
   providedPort: #Chat -> { #join: . #leave . send: } ;
   providedPort: #AccessCT -> { #getChatText . #setChatText: } ;
   property: #ChatText accessPort: #AccessCT.   "déclaration de la propriété ChatText"

ChatClient>>init
   ct bindTo: Text newC
   "..."

ChatClient>>getChatText
   ^ ct getText

ChatClient>>setChattext: v
   ct setText: v
```

LISTING 5.6 : Déclaration de propriétés en SCL

Le listing 5.7 montre le code permettant d'établir une connexion, entre un CHATCLIENT et un CHATCLIENTGUI, basée sur les changements d'état de la propriété `ChatText` (*cf.* figure 4.6).

```
"Supposons que les variables pm, rng, c1..c5 contiennent des références vers des ports"

"Résolution d'une incompatibilité entre pm et rng (cf.  listing 3.4)"
BinaryConnector newC
   source: (pm port: #Randomizer)
   target: (rng port: #Generator)
   glue: [ :source :target :si | "Paramètres du code glue"
      ^ target random * 26
   ].

"Utilisation d'un autre connecteur binaire avec du code glue aussi"
TCPConnector newC
   source: (pm port: #Randomizer)
   target: (rng port: #Generator)
   glue: [ :s :t :si |
      ^ t random * 26
   ].

"Utilisation d'un connecteur n-aire pour faire du broadcast"
Connector newC
   sources: { c1.r1 . c2.r1 }
   targets: { c3.p1 . c4.p2 . c5.p1 }
   glue: [ :s :t :si |
      (1 to: 3) do: [ :i | (t at: i) perform(si) ]
   ]

"Associer du code glue à un seul port source"
Connector newC
   sources: { c1.r1 . c2.r1 }
   targets: { c3.p1 . c4.p2 . c5.p1 } ;
   glueSourceAt: 1 with: [ :s :t :si |
      (1 to: 3) do: [ :i | (t at: i) perform(si) ]
   ] ;
   glueSourceAt: 2 with: [ :s :t :si |
      (1 to: 3) do: [ :i | (t at: i) perform(si) ]
   ]
```

LISTING 5.5 : Utilisation de connecteurs et de *code glue* en SCL

```
client := ChatClient newC.
clientGui := ChatClient newC.

BinaryConnector newC
   source: (client notifyPort: #ChatText)
   target: (clientGui port: #Display)
   glue: [ :s :t :si |
      (si selector == #nac:value:oldValue:) ifTrue: [
         t displayText: (si arguments at: 2)
      ]
   ].

"ci-dessous, la même connexion est définie en utilisant "
"un connecteur spécifique simplifiant l'écriture de ce type de connexions"
BinaryNACConnector newC
   property: (client property: #ChatText)
   target: (clientGui port: #Display)
   glue: [ :property :target :newValue :oldValue |
      t displayText: newValue
   ]
```

LISTING 5.7 : Mise en place d'une connexion basée sur les changements de valeur d'une propriété en SCL

5.10 Vers un environnement de développement graphique

La figure 5.5 présente une capture d'écran d'un outil de développement visuel permettant l'assemblage des composants SCL. Ce prototype d'outil permet actuellement à un architecte de « prendre à la souris » des composants dans une bibliothèque et de les « poser » dans la palette de construction des assemblages. Cette action a pour effet de créer une instance du descripteur choisi dans la bibliothèque. La liste des instances, c'est-à-dire des composants présents dans l'architecture, est présentée en dessous de la bibliothèque. En suite, il est possible d'utiliser l'outil de liaison (le bouton avec la flèche) pour lier un port requis avec un port fourni. Une fois la liaison établie et réussie (contrainte de compatibilité des interfaces), il est possible d'entrer du code SCL dans la zone de saisie en bas à gauche pour invoquer les services des composants. Les résultats peuvent être visualisés dans la fenêtre en bas à droite qui est la sortie standard. Les liaisons peuvent être enlevées en supprimant les flèches entre les ports. Cet environnement d'assemblage visuel est un prototype simple d'outil nécessaires à la compréhension et à la mise en pratique du développement par composants. Dans l'annexe B, nous présentons brièvement les environnements de développement visuels adaptés à l'approche à composants que nous avons découvert.

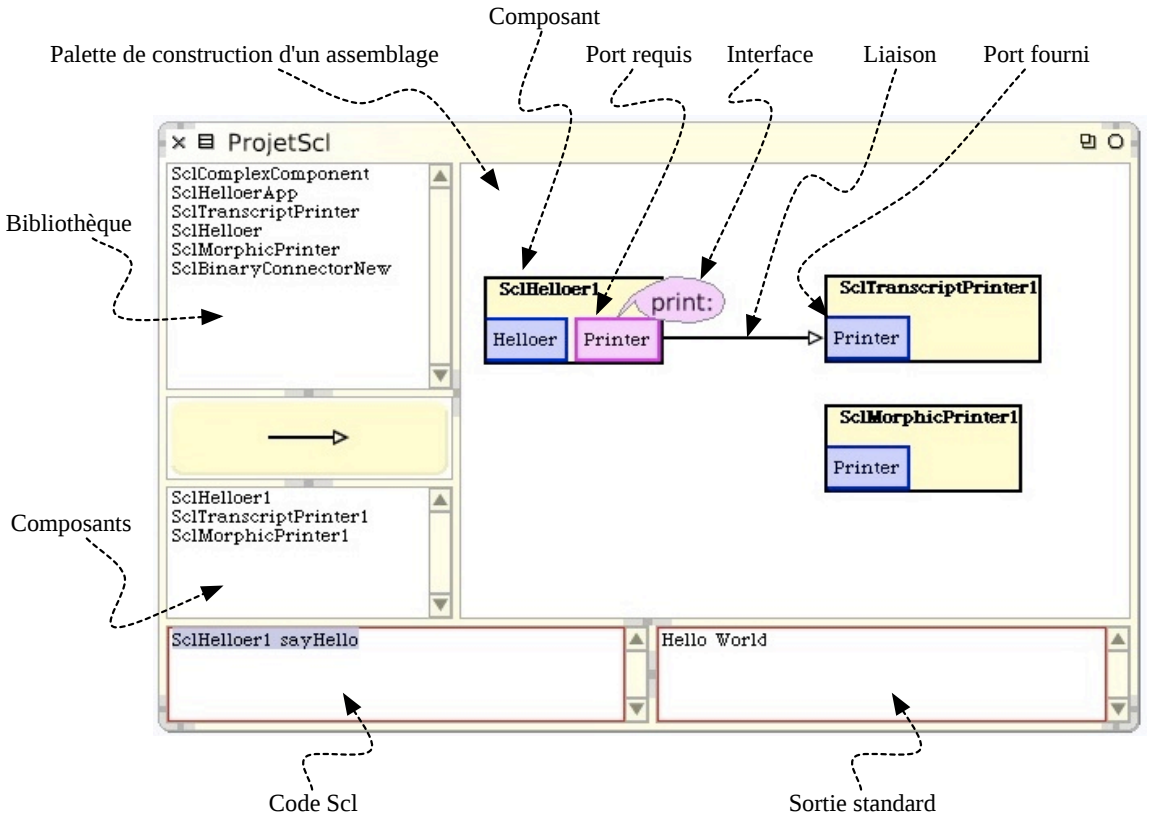

FIG. 5.5 : Capture d'écran d'un prototype d'outil visuel pour l'assemblage de composants SCL

5.11 Le cœur d'un prototype en Ruby

Dans cette section, nous présentons brièvement le cœur d'un prototype de SCL écrit avec le langage à objets Ruby [Anantharam, 2001].

Pourquoi Ruby ? Contrairement à Smalltalk, Ruby utilise une syntaxe mieux connue actuellement. C'est donc uniquement la barrière de la syntaxe qui nous a incité à développer un autre prototype que celui en Smalltalk. Notre choix s'est porté sur Ruby car :

- sa syntaxe est simple et relativement proche de tous les langages actuels tels que Java ou C#,
- il est inspiré de Smalltalk et possède donc des caractéristiques similaires : langage à objets réflexif, réification des fermetures, héritage simple, etc,
- il est très bien adapté à la création de DSL.

Implémentation. Nous avons fait exactement les mêmes choix de conception que ceux présentés dans la section 5.2 concernant le prototype en Smalltalk. De ce fait, l'amorçage, l'architecture générale

ou encore le modèle implémenté sont très similaires. Le listing 5.8 présente un exemple de code SCL écrit avec ce protoype en Ruby. Nous pouvons constater que la syntaxe de SCL est très proche de celles que nous avons utilisée dans les chapitres 3 et 4.

```
componentDescriptor :CDPasswordManager do
    providedPort :in1, [:generatePwd, :generateADigitsOnlyPwd]
    providedPort :in2, [:isValid]
    requiredPort :out, [:getRandomNumber]

    def generatePwd
      # ...
      size=out.getRandomNumer(taille)
      for i in 1..size
        j=out.getRandomNumer(25)
        # ...
      end
      pwd
    end
end

componentDescriptor :CDRandomNumberGenerator do
    providedPort :in, [:rand]

   def init ; srand ; end
   def rand ; rand ; end # retourne un nombre compris dans [0.0..1.0[
end

componentDescriptor :CDMyAdHocConnector do
    providedPort :source1, IRandomizer
    requiredPort :target1

   # le service _glue1 est défini comme service glue associé à un port fourni source1
   glue :source1, :_glue1
   def _glue1( sel, *args )
      ((target1.rand*args[0])+1).to_i
   end
end

# new retourne une référence sur le port par défaut du composant créé
pm = CDPasswordManager.new
rng = CDRandomNumberGenerator.new
c = CDMyAdHocConnector.new

pm.out.bindTo(c.source1)   # liaison de ports
# Pour chacun de ses ports externes (comme out ou source1), un composant est doté
# d'un service portant le nom du port qui retourne une référence sur ce port
c.target1.bindTo(rng.in)
puts pm.in1.generatePwd    #invocation de service
```

LISTING 5.8 : Exemple de code SCL écrit avec le prototype en Ruby

L'amorçage de ce prototype et le développement des principales entités permet aujourd'hui d'exécuter des exemples utilisant des liaisons de type requis–fourni essentiellement. Les principaux éléments actuellement manquant dans ce prototype sont :

- les ports collection,
- l'intégration des objets de base,
- les liaisons d'aspects,
- les propriétés.

En regard de ce qui est déjà programmé, nous pensons que le développement de ces éléments manquants ne devrait pas poser de difficulté particulière par rapport au prototype écrit en Smalltalk.

5.12 Conclusion

Dans ce chapitre, nous avons présenté deux prototypes de SCL écrits respectivement en Smalltalk et en Ruby. L'implémentation du prototype écrit en Smalltalk est bien plus détaillée car c'est actuellement la plus avancée des deux. Nous avons aussi présenté les arguments qui nous ont poussés à choisir le langage Smalltlak et nos choix d'implémentation dont celui de ne pas faire de *parseur*. Concernant ce prototype écrit en Smalltalk, nous avons :

- abordé le problème d'amorçage du prototype qui impose que la classe représentant un descripteur de composant hérite de la classe `Class`,
- présenté l'architecture en couches du prototype et notre volonté de masquer aux utilisateurs de SCL les couches relevant de l'implémentation,
- décrit le modèle UML implémenté,
- intégré les objets de base et les composants en proposant la primitive **`newC`**,
- détaillé l'implémentation de l'invocation de service qui utilise la même syntaxe que celle de l'envoie de message mais dont le fonctionnement est conforme à celui définit dans le chapitre 3,
- illustré l'assemblage de composants à l'aide de liaisons, de connecteurs et de *code glue* à travers des exemples,
- défini des propriétés de composants afin d'établir des connexions basées sur leurs changements d'état,
- montré, pour finir, un outil visuel pour assembler des composants SCL.

Ensuite, nous avons présenté le prototype écrit en Ruby en commençant par les raisons qui nous ont conduites à l'écrire et en terminant par un exemple de code.

Le point d'amélioration de ces prototypes est très certainement l'efficacité. En effet, par nos choix nous avons clairement privilégié l'évolutivité à l'efficacité. Il est difficile de quantifier l'efficacité des prototypes actuels mais l'implémentation que nous avons choisie pour l'invocation de service (délégation explicite entre les objets représentant les ports) laisse supposer des performances médiocres.

CHAPITRE 6

Bilan et Perspectives

On ne fait jamais attention à ce qui a été fait ;
on ne voit que ce qui reste à faire.

Marie CURIE.

Préambule

Ce chapitre dresse le bilan du travail réalisé au cours de cette thèse et présente cinq perspectives qui nous intéressent suite à ce travail. La première concerne la formalisation de SCL *sous la forme d'une sémantique opérationnelle. La seconde propose l'étude d'une version réflexive de* SCL *où les descripteurs seraient des composants assemblables. La troisième, plus pragmatique, soulève le besoin de concevoir des outils de développements adaptés à la PPC. Dans la quatrième perspective, nous présentons notre vision à long terme de ce travail qui est d'aller vers une plateforme d'expérimentation extensible des concepts et mécanismes relatifs à la PPC. Enfin, dans la cinquième et dernière perspective que nous présentons, nous détaillons notre vision en ce qui concerne la vérification des assemblages actuellement limitée en* SCL.

CETTE thèse s'inscrit dans le domaine de recherche du développement par composants, actuellement très actif à cause de la difficulté de concevoir, de maintenir et de faire évoluer des applications de plus en plus complexes. En effet, les approches à composants promettent une meilleure modularisation des logiciels de grande taille facilitant ainsi leur conception, leur maintenance et leur évolution. Malgré de nombreuses propositions (technologies, modèles) à composants, on utilise encore aujourd'hui principalement des langages de programmation à objets provoquant ainsi une rupture entre les modèles architecturaux et le code source.

Dans ce mémoire, nous avons abordé cette problématique en étudiant les principales approches à composants (*cf.* chapitre 2) ce qui nous a permis d'isoler les deux notions qui nous semblent fondamentales pour la programmation par composants (PPC) : le *découplage* et la *non-anticipation*.

Le découplage consiste à créer les composants indépendamment de leurs contextes de définition ou d'utilisation. En effet, les composants sont créés dans un contexte donné et généralement conçus pour un contexte donné c'est-à-dire une application particulière alors qu'ils devraient être conçus pour être réutilisés (*design for reuse*). Afin d'augmenter leurs possibilités de réutilisation, ils doivent tout de même pouvoir être adaptés à différents contextes d'utilisation. Pour cela, le découplage impose une encapsulation forte des composants en les dotant de ports et/ou d'interface qui spécifient toutes les interactions entre un composant et son environnement.

La non-anticipation — rarement mise en avant dans les travaux sur les composants — nous semble primordiale pour le développement par composants. Par non-anticipation, nous désignons le fait que les programmeurs de composants ne doivent en aucun cas écrire de code spécifique et non-métier pour permettre l'assemblage ou l'utilisation d'un composant d'une façon particulière et donc prévue lors de sa création. En effet, dans un tel contexte, les possibilités de réutilisation des composants seraient directement proportionnelles à la capacité de leur programmeur à prévoir un maximum d'utilisations potentielles de leur composants.

Le manque d'un langage de programmation simple et vraiment dédié à la programmation par composants qui respecte ces deux notions clés nous a conduit à proposer dans le chapitre 3 le langage à composants SCL (*Simple Component Language*). Avec SCL, notre objectif est de faciliter l'écriture d'applications par assemblage de composants logiciels en offrant au programmeur des abstractions et des mécanismes de haut niveau pour ce mode de développement. La spécification de ce langage a été faite en adoptant une démarche constructive, c'est-à-dire que nous avons identifié un à un les besoins de ce mode de développement et intégré les concepts et/ou mécanismes minimaux couvrant ce besoin à partir de l'étude des propositions existantes. Cette démarche nous permet de penser que SCL synthétise les concepts et mécanismes fondamentaux des approches existantes. Toutefois, nous avons aussi traité des problématiques cruciales qui pour certaines ne sont pas (ou peu) abordées par les approches existantes comme :

- l'intégration des types de bases (entiers, chaînes de caractères, etc.) qui sont considérés comme des composants en SCL ;
- la nécessité d'effectuer des auto-références qui nous a conduit à introduire le port interne `self` ;
- le passage d'arguments, souvent problématique pour le contrôle de l'intégrité des communications, qui repose sur la connexion en SCL ;
- la nécessité d'adapter les composants lors de leur assemblage à cause des incompatibilités structurelles ce qui nous a amené à introduire la notion de connecteur et de code *glue*.

Par ailleurs, une extension de SCL a été proposée dans le chapitre 4 pour supporter la séparation des préoccupations en s'inspirant des techniques issues de la programmation par aspects. En effet, l'approche à composants — comme l'approche à objets — ne permet pas de bien modulariser les préoccupations transversales. En s'inspirant de certaines approches mixtes à aspects et à composants dites symétriques, nous avons introduit en SCL de nouveaux types de liaisons de ports. Toutefois, contrairement à ces approches, nous avons respecté nos deux principes initiaux de découplage et de non-anticipation, ce qui permet à un même composant SCL d'être assemblé de façon standard ou de façon transversale indépendamment de sa définition. De plus, la souplesse du mécanisme d'assemblage de composants de SCL grâce aux connecteurs et au code *glue* permet d'effectuer le tissage via le mécanisme d'assemblage standard en utilisant ces nouveaux types de liaisons de ports.

Nous avons aussi intégré la notion de *propriété observable* en SCL afin d'autoriser les connexions basées sur les changements d'états des propriétés. Dans les approches existantes, les propriétés sont souvent restreintes à un couple d'accesseurs alors que le modèle *Javabeans* a largement montré qu'il est intéressant de doter les propriétés de comportement signalant leurs changements de valeurs. Cela permet d'établir des connexions basées sur les changements de valeur de propriétés. Dans les approches existantes (*Javabeans* y compris), le programmeur d'un composant doit écrire du code spécifique afin de rendre possible l'établissement de telles connexions. Nous proposons en SCL un mécanisme permettant au programmeur de s'affranchir de toutes ces lignes de code contredisant la non-anticipation. Ainsi, un programmeur déclare des propriétés (au même titre qu'il déclarerait des accesseurs) ce qui permet ensuite d'établir des connexions basées sur les changements de valeurs de ces propriétés.

Afin de concrétiser SCL, un premier prototype a été développé en Smalltalk (*cf.* chapitre 5). Ce prototype a été implémenté en utilisant les capacités réflexives et la souplesse de la syntaxe de Smalltalk. La notion de bloc (fermeture lexicale) a été particulièrement utile pour simplifier la définition des connecteurs et du code glue. Nous avons aussi présenté une ébauche d'environnement de développement graphique. Le cœur d'un deuxième prototype a été écrit en Ruby car il utilise une syntaxe proche des langages les plus connus comme Java ou C# alors que Smalltalk utilise une syntaxe souvent mal connue.

Les perspectives de ce travail sont multiples tant sur le plan conceptuel qu'opérationnel. Nous détaillons ici celles qui nous intéressent le plus et témoignent de notre vision de l'avenir pour SCL.

Formalisation. Tout d'abord, nous souhaitons formaliser le cœur de SCL sous la forme d'une sémantique opérationnelle. Nous avons d'ailleurs commencé ce travail avec la collaboration de Guy Tremblay en utilisant l'outil LETOS [Hartel, 1999]. Le premier résultat de ce travail est la formalisation de la syntaxe abstraite du langage SCL, c'est-à-dire la définition des principales constructions syntaxiques (sous forme abstraite) ainsi que les principaux concepts sémantiques de SCL. La prochaine étape pour obtenir une sémantique opérationnelle consiste à décrire la partie dynamique de SCL qui s'articule essentiellement autour du mécanisme d'invocation de service.

Réflexivité. Nous envisageons aussi la possibilité d'utiliser le même mécanisme d'assemblage aussi bien au niveau des composants qu'au niveau des descripteurs. Notre volonté d'unifier le niveau des composants et celui des descripteurs provient du fait que dans certaines approches, l'assemblage est

effectué au niveau des descripteurs et non au niveau des instances comme en SCL. C'est par exemple le cas avec les *traits* [Schärli *et al.*, 2003; Nierstrasz *et al.*, 2006] qui sont essentiellement des groupes de méthodes (fournies et requises) utilisés pour construire des classes grâce un mécanisme de composition. De même, dans le langage Scala [Odersky et Zenger, 2005] les composants sont des classes et l'assemblage repose sur la composition à base de *mixins*. Dans ces approches, les classes et les traits sont des entités différentes et non unifiées n'offrant pas les mêmes possibilités alors qu'elles sont toutes les deux vouées à être réutilisées. Pour atteindre cet objectif d'unification en SCL, nous pensons qu'il faut réfléchir à une version réflexive de SCL. En effet, si les descripteurs deviennent des composants assemblables et instanciables, il serait alors possible d'utiliser le même mécanisme d'assemblage au niveau des descripteurs et des composants. Cela soulève bien sûr de nombreuses questions telles que : Un descripteur peut-il être un composant ? Les ports et les services peuvent-ils être des composants ? Peut-on imaginer construire un descripteur de composants en assemblant des composants ? Comment traiter la méta-régression ? ... Il semble possible d'apporter des éléments de réponses à certaines de ces questions. Par exemple, on peut considérer un service comme un composant où les paramètres sont des ports requis (liés aux arguments lors de l'invocation), l'implémentation est du code glue associé à un port fourni permettant l'invocation du service. Toutefois, cette perspective nécessite un travail important qui pourra éventuellement impliquer des changements profonds dans le cœur de SCL.

Environnement de développement. En plus d'un interpréteur, nous souhaitons concevoir des outils spécifiques pour le développement par composants. Les nouveaux besoins de ce mode de développement comme la mise à disposition dans une bibliothèque, la recherche de composants ou encore l'assemblage de composants sont de plus en étudiés et des modèles sont proposés. Toutefois, il existe en pratique peu d'outils pour ces nouvelles tâches et encore moins dans un environnement intégré. Nous pensons que la mise en pratique de la programmation par composants nécessite des outils spécifiques. Il existe d'ailleurs actuellement des propositions d'environnements visuels dédiés à la programmation par assemblage. Nous avons réalisé un premier pas dans ce sens avec l'outil de développement graphique que nous avons présenté dans le chapitre 5.

Vers une plateforme d'expérimentation extensible. Nous souhaitons que SCL constitue une base pour comprendre, étudier mais aussi expérimenter les concepts et mécanismes relatifs au développement par composants. En effet, SCL est bien plus simple que les approches industrielles qui comme nous l'avons vu sont souvent complexes, difficiles à mettre œuvre et à étendre. Nous pensons aussi que l'uniformité de SCL en fait un meilleur candidat pour expérimenter l'approche à composants que d'autres approches telles que Julia ou ArchJava. Fort de ces atouts, nous pensons qu'il serait intéressant de disposer pour SCL d'un noyau formalisé et d'un environnement d'expérimentation facilement extensible. Cela comprendrait la possibilité d'étendre le noyau de SCL mais aussi les outils de l'environnement de développement afin d'expérimenter facilement les solutions proposées à de nombreuses problématiques relatives au développement par composants. Comme nous l'avons fait en définissant des extensions pour les aspects et les propriétés observables, nous souhaiterions pouvoir facilement définir des extensions opérationnelles avec un minimum d'effort.

La vérification des assemblages. La vérification des assemblages est actuellement limitée en SCL et nécessite des travaux supplémentaires. Actuellement la plupart des approches opérationnelles s'appuient sur un système de type et des règles de sous-typage pour vérifier la cohérence syntaxique des liaisons. Cela suppose qu'il existe une relation de sous-typage entre les types des éléments à assembler. Cette contrainte peut s'avérer contradictoire avec notre principe de non-anticipation si le système de type utilisé repose sur des noms comme en Java. Nous pensons qu'il serait intéressant d'essayer d'utiliser des techniques d'inférence de types comme c'est le cas en Objective CAML [Loulergue, 2004] où le programmeur n'explicite jamais les annotations de types ; elles sont inférées.

ANNEXE A

Programmation par composants avec des langages de programmation actuels

Les absurdités d'hier sont les vérités d'aujourd'hui et les banalités de demain.
Alessandro MORANDOTTI.

Annexe en cours d'écriture...

Préambule

*Dans ce chapitre, nous montrons comment utiliser les langages de programmation existants pour programmer par assemblage de composants. La COP (*Component-oriented programming*) peut en effet, selon nous, être mise en pratique dans la plupart des langages de programmation existants si l'on respecte certaines règles de programmation. Bien évidemment, suivant le langage choisi et les règles à respecter, la POC peut être plus ou moins facile en œuvre.*

Notre objectif est de montrer dans cette annexe que si le programmeur respecte des règles de programmation données, il peut faire de la POC dans les langages de programmation actuels. Dans les sections qui suivent, nous présentons les règles à respecter pour faire de la POC en C et en Java. Nous avons établit ces règles afin de maximiser le découplage entre les entités logicielles.

A.1 PPC en C

Dans cette section, nous montrons qu'il est possible d'adopter un style de programmation par composants avec un langage procédural et modulaire comme C. On aurait sans doute pu utiliser n'importe quel langage de cette famille comme ADA ou Pascal pour faire cette démonstration.

- composant compilé = fichier .o ou fichier .a
- composant = un fichier .c d'implémentation
- propriétés/attributs d'un composant = les variables "static" (portée du fichier)
- services fournis d'un composant = les fonctions définies dans le fichier c
- services requis d'un composant = les signatures "extern" déclarées

```
/***************
uncompteur.c
Un composant compteur
***************/
#include "compteur_implementation.c"

/* les propri\'et\'es */
static int val = 0;
```

```
/***************
compteur.c
Un composant compteur
***************/

extern int val;

/* les services fournis */
void incr() { val ++; }
int value() { return val; }
```

```
/***************
test.c
Un composant Test fournissant un service main
****************/

/* Les services requis par ce composant */
extern int printf(const char * __restrict, ...);
extern void incr();
extern int value();
```

```
/* Un service fournit */
int main () {
   int a = value();
   printf( "%i \n", a);

   incr();

   a = value();
   printf( "%i \n", a);

   return 0;
}
```

La compilation des composants s'effectue avec gcc afin d'obtenir un .o constituant dans notre exemple la forme de diffusion des composant :

```
$ gcc -c compteur.c
$ gcc -c test.c
$ gcc -nodefaultlibs -o test uncompteur.o compteur.o test.o libc.a libgcc.a
```

La dernière ligne assemble ces composants pour qu'ils fonctionnent ensembles. Pour cela, nous utiliserons le linker de gcc :

Le composant Test requiert trois services dont deux sont fournis par le composant Compteur et le dernier par la bibliothèque libc.a elle même possédant des dépendances satisfaites par libgcc.a. L'exécutable obtenu (test) correspond au résultat d'un assemblage de composants logiciels où toutes les dépendances ont été satisfaites. L'exécution de notre application test construite par assemblage de composants nous donne :

```
$ ./test
0
1
```

A.2 PPC en Java

Les langages à objets sont actuellement les plus utilisés. Ils peuvent être utilisés pour faire de la programmation par composants si l'on respecte des schémas de conception et des conventions. Nous avons choisi d'utiliser le langage Java pour cette étude car il est le langage le plus utilisé par les chercheurs pour prototyper des langages à composants (ArchJava, ComponentJ, Lagoona, Julia, JPiccola, ...). L'objectif est de comprendre comment il serait possible de faire de la programmation par composants en utilisant le langage Java.

– composant compilé = fichier .class ou .jar

- composant = une classe java
- propriétés/attributs d'un composant = les attributs de la classe
- services fournis d'un composant = les méthodes
- services requis = les signatures définies dans les interfaces qui typent les attributs

Pour faire de la programmation par composants en Java, il faut suivre un ensemble de règles.

- Ne jamais faire d'instanciation dans une classe (sauf dans les classes de composants composites)
- Toujours typer les attributs d'une classe par une interface afin de ne pas fixer le type concrêt
- Toujours munir la classe d'accesseurs. (passer au constructeur ne suffit pas. Cela figerait pour la durée de vie de l'instance)

```
public interface Printer {
   public void print(String s);
}

public class Helloer {
   private Printer printer;

   public Printer getPrinter() { return printer; }
   public void setPrinter(Printer p) { printer = p; }

   public void sayHello() { printer.print("Hello world !"); }
}

public class ConsolePrinter implements Printer {
   public void print(String s) {
      ...
   }
}

public class MonApplication {
   public void main(String args[]) {
      h = new Helloer();
      p = new ConsolePrinter();

      h.setPrinter(p);
      h.sayHello();
   }
}
```

Le composant Helloer requiert un Printer i.e requiert un composant sachant afficher. Ainsi construit, le composant Helloer pourra utiliser n'importe quel composant qui est un Printer.

MonApplication est un exemple d'application qui connecte une instance de ConsolePrinter à une instance de Helloer. Le Helloer utilisera dans cette application un ConsolePrinter. Pendant l'exécution d'une application, on peut changer le Printer qu'utilise le Helloer.

ANNEXE B

Environnements de développement visuels basés sur l'approche à composants

En toute chose, c'est la fin qui est essentiel.
Aristote.

Annexe en cours d'écriture...

Préambule

Ce chapitre présente différentes technologies que nous avons découvertes et qui relèvent du développement par assemblage de composants à travers un environnement visuel.

B.1 Quartz Composer

Quartz Composer[1] est un *environnement de développement visuel* fournit avec Mac Os X depuis sa version Tiger (10.4).

B.2 Automator

http://www.apple.com/fr/macosx/features/automator/

http://www.apple.com/fr/macosx/theater/automator.html

B.3 Pipes

Pipes[2] est une application web proposée par Yahoo! qui fournit une interface visuelle pour construire des applications web en agrégeant des flux rss ou d'autres services provenant de sites différents. La définition proposée par le site officiel est la suivante :

> « *Pipes is a powerful composition tool to aggregate, manipulate, and mashup content from around the web.* »

B.4 Scratch

http://scratch.mit.edu

[1] http://developer.apple.com/graphicsimaging/quartz/quartzcomposer.html
http://developer.apple.com/documentation/GraphicsImaging/Conceptual/QuartzComposer/index.html

[2] http://pipes.yahoo.com

Bibliographie

[Abadi et Cardelli, 1996] Martin Abadi et Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[Achermann *et al.*, 2001] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, et Oscar Nierstrasz. Piccola – a Small Composition Language. In *Formal Methods for Distributed Processing – A Survey of Object-Oriented Approaches*, éditeurs Howard Bowman et John Derrick, pages 403–426. Cambridge University Press, 2001.

[Aldrich *et al.*, 2002a] Jonathan Aldrich, Craig Chambers, et David Notkin. Architectural reasoning in archjava. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 334–367, London, UK, 2002. Springer-Verlag.

[Aldrich *et al.*, 2002b] Jonathan Aldrich, Craig Chambers, et David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *ICSE*, pages 187–197. ACM, 2002.

[Aldrich *et al.*, 2003] Jonathan Aldrich, Vibha Sazawal, Craig Chambers, et David Notkin. Language support for connector abstractions. In *ECOOP*, éditeur Luca Cardelli, volume 2743 de *Lecture Notes in Computer Science*, pages 74–102. Springer, 2003.

[Aldrich, 2003] J. Aldrich. *Using Types to Enforce Architectural Structure*. PhD thesis, University of Washington, August 2003.

[Almeida *et al.*, 2001] João Paulo A. Almeida, Marten Van Sinderen, et Lambert Nieuwenhuis. Transparent Dynamic Reconfiguration for CORBA. In *DOA '01: Proceedings of the Third International Symposium on Distributed Objects and Applications*, page 197, Washington, DC, USA, 2001. IEEE Computer Society.

[Anantharam, 2001] Parasuram Anantharam. Programming ruby. *SIGSOFT Software Engineering Notes*, 26(4):89–89, 2001.

[Atkinson *et al.*, 2001] Colin Atkinson, Barbara Paech, Jens Reinhold, et Torsten Sander. Developing and Applying Component-Based Model-Driven Architectures in KobrA. *edoc*, 00:0212, 2001.

[Bachman et *al.*, 2000] F. Bachman et *al.* Volume ii : Technical concepts of component-based software engineering, 2nd edition. Rapport Technique 2, The Software Engineering Institute (SEI), Carnegie Mellon University, mai 2000. *http://www.sei.cmu.edu/publications/documents/00.reports/00tr008/00tr008title.html*.

[Bálek et Plásil, 2001] Dusan Bálek et Frantisek Plásil. Software connectors and their role in component deployment. In *Proceedings of DAIS'01*, Krakow, Poland, September 2001. Kluwer Academic Publishers.

[Balek, 2002] Dusan Balek. Connectors in software architectures, 2002.

[Bastide et Barboni, 2006] Rémi Bastide et Eric Barboni. Software Components: a Formal Semantics Based on Coloured Petri Nets. *Electronic Notes in Theoretical Computer Science*, 160:57–73, 2006.

[Bennouar *et al.*, 2006] D. Bennouar, T. Khammaci, et A. Henni. The Design of a Complex Software System with ArchJava. *Journal of Computer Science*, 2(11):807–814, september 2006. Science Publications.

[Beugnard *et al.*, 1999] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, et Damien Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, 1999.

[Beugnard, 2005] Antoine Beugnard. Peut-on réaliser des composants avec un langage á objets ? In *Proceedings of LMO'05 Conference (Langages et Modèles à Objets)*, pages 47–60, Bern, Switzerland, March 2005. Hermes Editions.

[Birtwistle *et al.*, 1973] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, et Kristen Nygaard. *SIMULA Begin*. Auerbach Publishers, Philadelphia, Penn., 1973.

[Black *et al.*, 2007] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, et Damien Pollet. *Squeak by Example*. Square Bracket Associates, 2007. with Damien Cassou and Marcus Denker.

[Bobrow *et al.*, 1986] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, et Frank Zdybel. Commonloops: merging lisp and object-oriented programming. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 17–29, New York, NY, USA, 1986. ACM Press.

[Bouraqadi, 1999] N. Bouraqadi. *Un MOP Smalltalk pour l'étude de la composition et de la compatibilité des métaclasses. Application à la programmation par aspects (A Smalltalk MOP for the Study of Metaclass Composition and Compatibility. Application to Aspect-Oriented Programming - In French)*. Thèse de doctorat, Université de Nantes, Nantes, France, Juillet 1999.

[Briand *et al.*, 1999] Lionel C. Briand, John W. Daly, et Jürgen K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Trans. Software Eng.*, 25(1):91–121, 1999.

[Brown et Wallnau, 1998] Alan W. Brown et Kurt C. Wallnau. The Current State of CBSE. *IEEE Software*, 15(5):37–46, 1998.

[Broy *et al.*, 1998] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, Frantisek Plásil, Gustav Pomberger, Wolfgang Pree, Michael Stal, et Clemens A. Szyperski. What characterizes a (software) component? *Software - Concepts and Tools*, 19(1):49–56, 1998.

[Bruneton *et al.*, 2002] E. Bruneton, T. Coupaye, et J.B. Stefani. The Fractal Composition Framework. Rapport technique, The Object Web Consortium, Juillet 2002.

[Bruneton *et al.*, 2004] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, et Jean-Bernard Stefani. An Open Component Model and Its Support in Java. In *CBSE*, éditeurs Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, et Kurt C. Wallnau, volume 3054 de *Lecture Notes in Computer Science*, pages 7–22. Springer, 2004.

[Büchi et Weck, 1998] Martin Büchi et Wolfgang Weck. Compound types for Java. In *OOPSLA'98: Proceedings of the 13th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 362–373, New York, NY, USA, 1998. ACM Press.

[Cardelli, 1997] Luca Cardelli. *The Handbook of Computer Science and Engineering*, chapitre 103, Type Systems, pages 2208–2236. CRC Press, Boca Raton, FL, 1997.

[Cheesman et Daniels, 2000] John Cheesman et John Daniels. *UML components: a simple process for specifying component-based software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Choi, 2000] Jung Pil Choi. Aspect-Oriented Programming with Enterprise JavaBeans. *edoc*, 00:252, 2000.

[Cohen et Gil, 2004] Tal Cohen et Joseph Gil. *AspectJ2EE = AOP + J2EE*, volume 3086. January 2004.

[Cointe *et al.*, 2004] Pierre Cointe, Jacques Noyé, Rémi Douence, Thomas Ledoux, Jean-Marc Menaud, Gilles Muller, et Mario Südholt. Programmation post-objets : des langages d'aspects aux langages de composants. *RSTI - L'objet*, 10(4):119–143, 2004.

[Collet *et al.*, 2005] Philippe Collet, Roger Rousseau, Thierry Coupaye, et Nicolas Rivierre. A Contracting System for Hierarchical Components. In *Component-Based Software Engineering, 8th International Symposium (CBSE'2005)*, volume 3489 de *LNCS*, pages 187–202, St-Louis (Missouri), USA, Mai 2005. Springer Verlag.

[de Alfaro et Henzinger, 2001] Luca de Alfaro et Thomas A. Henzinger. Interface automata. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 109–120, New York, NY, USA, 2001. ACM Press.

[DeRemer et Kron, 1975] Frank DeRemer et Hans Kron. Programming-in-the-large versus Programming-in-the-small. In *Proceedings of the international conference on Reliable software*, pages 114–121, New York, NY, USA, 1975. ACM Press.

[Desnos *et al.*, 2007] Nicolas Desnos, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, et Guy Tremblay. Automated and unanticipated flexible component substitution. In *Proceedings of the 10th ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE2007)*, éditeurs H. W. Schmidt et al., volume 4608 de *LNCS*, pages 33–48, Medford, MA, USA, July 2007. Springer.

[Dijkstra, 1976] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1976.

[Dony *et al.*, 1992] Christophe Dony, Jacques Malenfant, et Pierre Cointe. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. In *OOPSLA*, pages 201–217, 1992.

[D'Souza et Wills, 1999] Desmond F. D'Souza et Alan Cameron Wills. *Objects, components, and frameworks with UML: the catalysis approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Ducasse *et al.*, 2006] Stéphane Ducasse, Tudor Gîrba, et Adrian Kuhn. Distribution Map. In *Proceedings International Conference on Software Maintainance (ICSM 2006)*, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society.

[Duclos *et al.*, 2002] Frédéric Duclos, Jacky Estublier, et Philippe Morat. Describing and using non functional aspects in component based applications. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 65–75, New York, NY, USA, 2002. ACM Press.

[Eugster *et al.*, 2003] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, et Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computer Survey*, 35(2):114–131, 2003.

[Fakih *et al.*, 2004] Houssam Fakih, Noury Bouraqadi, et Laurence Duchien. Aspects and software components: A case study of the FRACTAL component model. In *International Workshop on Aspect-Oriented Software Development (WAOSD 2004)*, éditeurs Minhuan Huang, Hong Mei, et Jianjun Zhao, Septembre 2004.

[Filman et Friedman, 2000] R. Filman et D. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA'00*, Octobre 2000.

[Flanagan, 1998] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[Flatt, 2000] Matthew Raymond Flatt. *Programming languages for reusable software components*. PhD thesis, 2000. Adviser-Matthias Felleisen.

[Fröhlich *et al.*,] Peter H. Fröhlich, Andreas Gal, et Michael Franz. On Reconciling Objects, Components, and Efficiency in Programming Languages.

[Fröhlich *et al.*, 2005] Peter H. Fröhlich, Andreas Gal, et Michael Franz. Supporting software composition at the programming-language level. *Science of Computer Programming, Special Issue on New Software Composition Concept*, 56(1-2):41–57, April 2005.

[Fröhlich et Franz, 1999] Peter H. Fröhlich et Michael Franz. Component-Oriented Programming in Object-Oriented Languages. Rapport Technique 99-49, Department of Information and Computer Science, University of California, Irvine, CA 92697-3425, USA, Octobre 1999.

[Fröhlich, 2000] Peter H. Fröhlich. Component-Oriented Programming Languages: Messages vs. Methods, Modules vs. Types. In *Proceedings of the Workshop on Programming Languages and Computer Architecture*, Bad Honnef, Germany, Mai 2000. Technical Report 2007, Institute for Computer Science and Applied Mathematics, Christian-Albrechts-University, Kiel, Germany.

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

[Garlan *et al.*, 1995] David Garlan, Robert Allen, et John Ockerbloom. Architectural Mismatch, or, Why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, pages 179–185, Seattle, Washington, April 1995.

[Garlan et Shaw, 1993] David Garlan et Mary Shaw. An Introduction to Software Architecture. In *Advances in Software Engineering and Knowledge Engineering*, éditeurs V. Ambriola et G. Tortora, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

[Goldberg et Robson, 1989] Adele Goldberg et David Robson. *Smalltalk-80: The Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[Gröne *et al.*, 2005] Bernhard Gröne, Andreas Knöpfel, et Peter Tabeling. Component vs. component: Why we need more than one definition. In *ECBS*, pages 550–552. IEEE Computer Society, 2005.

[Hamilton, 1997] Graham Hamilton. JavaBeans. API Specification, Sun Microsystems, Juillet 1997. Version 1.01.

[Hartel, 1999] Pieter H. Hartel. LETOS - a lightweight execution tool for operational semantics. *Software: Practice and Experience*, 29(15):1379–1416, 1999.

[Hassine *et al.*, 2003] I. Hassine, D. Rieu, A. Front-Conte, et F. Bounaas. Modélisation et formalisation d'une démarche de développement à base de composants métier. TBA, 2003.

[Heineman et Councill, 2001] éditeurs George T. Heineman et William T. Councill. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Hejlsberg *et al.*, 2003] Anders Hejlsberg, Scott Wiltamuth, et Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Hürsch, 1994] Walter L. Hürsch. Should Superclass be Abstract? In *Proceedings ECOOP'94 : 8th European Conf. Object-Oriented Programming*, éditeurs M. Tokoro et R.Pareschi, volume 821 de *LNCS*, pages 12–31. Springer Verlag, july 1994.

[Ibrahim, 1998] R. Ibrahim. COMEL: A formal model for COM, 1998.

[Ingalls *et al.*, 1997] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, et Alan Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 318–326, New York, NY, USA, 1997. ACM Press.

[Jézéquel et Meyer, 1997] Jean-Marc Jézéquel et Bertrand Meyer. Design by Contract: The Lessons of Ariane. *Computer*, 30(1):129–130, 1997.

[Kiczales *et al.*, 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, et John Irwin. Aspect-oriented programming. In *11th Europeen Conf. Object-Oriented Programming*, éditeurs Mehmet Akşit et Satoshi Matsuoka, volume 1241 de *LNCS*, pages 220–242. Springer Verlag, 1997.

[Kiczales *et al.*, 2001] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, et William G. Griswold. An Overview of AspectJ. In *ECOOP*, éditeur Jørgen Lindskov Knudsen, volume 2072 de *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[Krasner et Pope, 1988] Glenn E. Krasner et Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. In *Journal of Object-Oriented Programming*, volume 1, pages 26–49, Août-Septembre 1988.

[Lahire *et al.*, 2004] Ph. Lahire, G. Arévalo, H. Astudillo, A.P. Black, E. Ernst, M. Huchard, T. Oplustil, M. Sakkinen, et P. Valtchev. Mechanisms for Specialization, Generalization and Inheritance, 2004.

[Lahire et Quintian, 2006] Philippe Lahire et Laurent Quintian. New Perspective To Improve Reusability in Object-Oriented Languages. *Journal Of Object Technology (JOT)*, 5(1):117–138, 2006.

[Langelier *et al.*, 2005] Guillaume Langelier, Houari A. Sahraoui, et Pierre Poulin. Visualisation et analyse de logiciels de grande taille. In *Langages et Modèles à Objets 2005*, pages x–y, Mars 2005.

[Lau et Wang, 2005a] K. K. Lau et Z. Wang. A Survey of Software Component Models. Technical reports, Department of Computer Science, University of Manchester, Avril 2005.

[Lau et Wang, 2005b] Kung-Kiu Lau et Zheng Wang. A taxonomy of software component models. In *EUROMICRO-SEAA*, pages 88–95. IEEE Computer Society, 2005.

[Lehman et Belady, 1985] éditeurs M. M. Lehman et L. A. Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[Leon, 2006] Ramon Leon, Octobre 2006. `http://feeds.feedburner.com/~r/onsmalltalk/KhHm/~3/43846539/`.

[Lieberherr *et al.*, 1999] K. Lieberherr, D. Lorenz, et M. Mezini. Programming with Aspectual Components, 1999.

[Lieberherr *et al.*, 2003] Karl Lieberherr, David H. Lorenz, et Johan Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, 46(5):542–565, Septembre 2003.

[Lieberman, 1986] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 214–223, New York, NY, USA, 1986. ACM Press.

[Loulergue, 2004] Frédéric Loulergue. Développement d'applications avec Objective CAML by E. Chailloux, P. Manoury and B. Pagano, O'Reilley, 2003. *Journal of Functional Programming*, 14(5):592–594, 2004.

[Luckham *et al.*, 1995] David C. Luckham, James Vera, et Sigurd Meldal. Three Concepts of System Architecture. Rapport Technique CSL-TR-95-674, 1995.

[Luckham et Vera, 1995] David C. Luckham et James Vera. An Event-Based Architecture Definition Language. *IEEE Trans. Software Eng.*, 21(9):717–734, 1995.

[Léger *et al.*, 2006] Marc Léger, T. Coupaye, et Thomas Ledoux. Contrôle dynamique de l'intégrité des communications dans les architectures à composants. In *Langages et Modèles à Objets*, éditeur S. Vauttier R. Rousseau, C. Urtado, pages 21–36. Hermès-Lavoisier, 2006.

[Magee et Kramer, 1996] Jeff Magee et Jeff Kramer. Dynamic structure in software architectures. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, New York, NY, USA, 1996. ACM Press.

[Marvie, 2002] Raphael Marvie. *Séparation des préoccupations et méta-modélisation pour environnements de manipulation d'architectures logicielles à base de composants*. PhD thesis, Université des Sciences et Techniques de Lille, LIFL, Villeneuve d'Ascq, Décembre 2002.

[McDirmid *et al.*, 2001a] S. McDirmid, M. Flatt, et W. Hsieh. Mixing COP and OOP, 2001.

[McDirmid *et al.*, 2001b] Sean McDirmid, Matthew Flatt, et Wilson C. Hsieh. Jiazzi: New-age components for old-fashioned java. In *OOPSLA*, pages 211–222, 2001.

[McDirmid et Hsieh, 2006] Sean McDirmid et Wilson C. Hsieh. Superglue: Component programming with object-oriented signals. In *ECOOP*, éditeur Dave Thomas, volume 4067 de *Lecture Notes in Computer Science*, pages 206–229. Springer, 2006.

[McIlroy, 1968] M. D. McIlroy. Mass produced software components. In *Proceedings, NATO Conference on Software Engineering*, éditeurs P. Naur et B. Randell, Garmisch, Germany, Octobre 1968.

[Medvidovic *et al.*, 2002] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, et Jason E. Robbins. Modeling software architectures in the Unified Modeling Language. *ACM Trans. Software Eng. Methodol.*, 11(1):2–57, 2002.

[Medvidovic et Taylor, 2000] Nenad Medvidovic et Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *Software Engineering*, 26(1):70–93, 2000.

[Mehta *et al.*, 2000] Nikunj R. Mehta, Nenad Medvidovic, et Sandeep Phadke. Towards a taxonomy of software connectors. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 178–187, New York, NY, USA, 2000. ACM Press.

[Mei, 2004] Hong Mei. ABC: Supporting Software Architectures in the Whole Lifecycle. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference on (SEFM'04)*, pages 342–343, Washington, DC, USA, 2004. IEEE Computer Society.

[Mernik *et al.*, 2005] Marjan Mernik, Jan Heering, et Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computer Survey*, 37(4):316–344, 2005.

[Mezini et Ostermann, 2003] Mira Mezini et Klaus Ostermann. Conquering aspects with Caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM Press.

[Microsoft, 1995] Microsoft. The Component Object Model Specification, 1995.

[Microsoft, 1996] Microsoft. DCOM technical overview. Microsoft Windows NT Server white paper, Microsoft Corporation, 1996.

[Mikhajlov et Sekerinski, 1998] Leonid Mikhajlov et Emil Sekerinski. A Study of the Fragile Base Class Problem. *Lecture Notes in Computer Science*, 1445:355+, 1998.

[Minsky, 1975] M. Minsky. A Framework for Representing Knowledge. In *The Psychology of Computer Vision*, éditeur P. Winston, pages 211–281. mgh, ny, 1975.

[Moon, 1986] David A. Moon. Object-oriented programming with flavors. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–8, New York, NY, USA, 1986. ACM Press.

[Moriconi *et al.*, 1995] Mark Moriconi, Xiaolei Qian, et R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Trans. Software Eng.*, 21(4):356–3, 1995.

[Nierstrasz *et al.*, 2005] Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse, Markus Gälli, et Roel Wuyts. On the revival of dynamic languages. In *Proceedings of Software Composition 2005*, éditeurs Thomas Gschwind et Uwe Aßmann, volume 3628, pages 1–13. LNCS 3628, 2005. Invited paper.

[Nierstrasz *et al.*, 2006] Oscar Nierstrasz, Stéphane Ducasse, et Nathanael Schärli. Flattening Traits. *Journal of Object Technology*, 5(4):129–148, may 2006.

[Nierstrasz et Dami, 1995] Oscar Nierstrasz et Laurent Dami. Component-oriented software technology. In *Object-Oriented Software Composition*, éditeurs Oscar Nierstrasz et Dennis Tsichritzis, pages 3–28. Prentice-Hall, 1995.

[Object Management Group, 2002] Object Management Group. *Manual of Corba Component Model V3.0*, 2002. *http://www.omg.org/technology/documents/formal/components.htm*.

[Odersky et Zenger, 2005] Martin Odersky et Matthias Zenger. Scalable Component Abstractions. In *OOPSLA*, éditeurs Ralph Johnson et Richard P. Gabriel, pages 41–57. ACM, 2005.

[OMG, 2004] Object Management Group OMG. Uml 2.0 superstructure specification. Rapport technique, Object Management Group, 2004.

[Ommering, 1998] Rob C. Van Ommering. Koala, a component model for consumer electronics product software. In *ESPRIT ARES Workshop*, éditeur Frank van der Linden, volume 1429 de *Lecture Notes in Computer Science*, pages 76–86. Springer, 1998.

[Opluštil, 2003] Tomáš Opluštil. Inheritance in Architecture Description Languages. In *WDS 2003 - Proceedings of Contributed Papers*, éditeur Jana Šafránková, pages 124–131, Prague, Czech Republic, 2003. Matfyzpress, MFF UK.

[Oussalah *et al.*, 2006] M. Oussalah, N. Sadou, et D. Tamzalit. SAEV :a Model to Face Evolution Problem in Software Architecture. In *Proceedings of the International ERCIM Workshop on Software Evolution*, pages 137–146, Lille, France, april 2006.

[Oussalah, 2005] M. Oussalah. *Ingénierie des composants : concepts, techniques et outils*. Editions Vuibert, 2005.

[Parnas, 1972] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Comm. ACM*, 15(12):1053–1058, Décembre 1972.

[Passama, 2006] Robin Passama. *Conception et développement de contrôleurs de robots - Une méthodologie basée sur les composants logiciels*. PhD thesis, Université de Montpellier 2, LIRMM, Juin 2006.

[Pavel *et al.*, 2005] Sebastian Pavel, Jacques Noyé, et Jean-Claude Royer. Un modèle de composant avec protocole symbolique. In *Journée du groupe Objets, Composants et Modèles*, Bern, Suisse, 2005.

[Pawlak *et al.*, 2004] Renaud Pawlak, Jean-Philippe Retaillé, et Lionel Seinturier. *Programmation orientée aspect pour Java/J2EE*. Eyrolles, 2004.

[Perry et Wolf, 1992] Dewayne E. Perry et Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[Peschanski *et al.*, 2000] F. Peschanski, T. Meurisse, et J.-P. Briot. Les composants logiciels : Evolution technologique ou nouveau paradigme ? In *In Actes de la conférence OCM'2000*, pages 53–65, 2000.

[Pessemier *et al.*, 2006] N. Pessemier, L. Seinturier, L. Duchien, et T. Coupaye. A model for developing component-based and aspect-oriented systems. In *Proceedings of the 5th International Symposium on Software Composition (SC'06)*, volume 4089 de *Lecture Notes in Computer Science*. Springer, Mars 2006.

[Plásil *et al.*, 1998] F. Plásil, D. Bálek, et R. Janecek. Sofa/dcup: Architecture for component trading and dynamic updating. In *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*, page 43, Washington, DC, USA, 1998. IEEE Computer Society.

[Plásil *et al.*, 1999] Frantisek Plásil, Miloslav Besta, et Stanislav Visnovsky. Bounding component behavior via protocols. In *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 387, Washington, DC, USA, 1999. IEEE Computer Society.

[Plásil et Visnovsky, 2002] Frantisek Plásil et Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Software Eng.*, 28(11):1056–1076, 2002.

[Pollet *et al.*, 2007] Damien Pollet, Stephane Ducasse, Loic Poyet, Ilham Alloui, Sorana Cimpan, et Herve Verjus. Towards A Process-Oriented Software Architecture Reconstruction Taxonomy. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 137–148, Washington, DC, USA, 2007. IEEE Computer Society.

[Privat, 2006] Jean Privat. *De l'expressivité à l'efficacité, une approche modulaire des langages à objets. Le langage PRM et le compilateur* `prmc`. PhD thesis, Université de Montpellier 2, LIRMM, Juillet 2006.

[Pulvermüller *et al.*, 2001] Elke Pulvermüller, Andreas Speck, Maja D'Hondt, Wolfgang DeMeuter, et James O. Coplien. Feature interaction in composed systems, ecoop 2001 - proceedings, tr no. 2001-14. Rapport Technique 2001-14, Sep 2001.

[Riveill et Merle, 2000] Michel Riveill et Philippe Merle. La programmation par composants. In *La programmation par composants*, numéro H2759. Techniques de l'Ingénieurs, Décembre 2000.

[Rogerson, 1997] Dale Rogerson. *Inside COM*. Microsoft Press, Redmond, WA, USA, 1997.

[Sametinger, 1997] Johannes Sametinger. *Software engineering with reusable components*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[Sanen *et al.*, 2006] Frans Sanen, Eddy Truyen, Wouter Joosen, Andrew Jackson, Andronikos Nedos, Siobhan Clarke, Neil Loughran, et Awais Rashid. Classifying and documenting aspect interactions. In *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, éditeurs Yvonne Coady, David H. Lorenz, Olaf Spinczyk, et Eric Wohlstadter, pages 23–26, Bonn, Germany, Mars 2006. Published as University of Virginia Computer Science Technical Report CS–2006–01.

[Schmidt, 2006] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[Schärli *et al.*, 2003] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, et Andrew Black. Traits: Composable Units of Behavior. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 de *LNCS*, pages 248–274. Springer Verlag, July 2003.

[Seco et Caires, 2000] João Costa Seco et Luís Caires. A basic model of typed components. *Lecture Notes in Computer Science*, 1850:108–129, 2000.

[Shaw *et al.*, 1995] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, et Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *Software Engineering*, 21(4):314–335, 1995.

[Shaw, 1996] Mary Shaw. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *ICSE '93: Selected papers from the Workshop on Studies of Software Design*, pages 17–32, London, UK, 1996. Springer-Verlag.

[Snyder, 1987] Alan Snyder. Inheritance and the development of encapsulated software systems. In *Research Directions in Object-Oriented Programming*, pages 165–188. 1987.

[Souchon, 2005] Frédéric Souchon. *SaGE, Un Système de Gestion d'Exceptions pour la Programmation Orientée Message : Le Cas des Systèmes Multi-Agents et des Plates-Formes à Base de Composants Logiciels*. PhD thesis, Université de Montpellier 2, LIRMM, 2005.

[Standard, 2001] ISO/IEC Standard. Software Engineering – Product Quality – Part 1: Quality Model. ISO Standard 9126-1, ISO/IEC, 2001. http://iso.org.

[Stein, 1987] Lynn Andrea Stein. Delegation is inheritance. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 138–146, New York, NY, USA, 1987. ACM Press.

[Stoerzer et Hanenberg, 2005] Maximilan Stoerzer et Stefan Hanenberg. Software Engineering Properties of Languages and Aspect Technologies. In *Software Engineering Properties of Languages and Aspect Technologies*, éditeurs Lodewijk Bergmans, Kris Gybels, Peri Tarr, et Erik Ernst, Mars 2005.

[Suvée *et al.*, 2005] Davy Suvée, Bruno De Fraine, et Wim Vanderperren. FuseJ: An architectural description language for unifying aspects and components. In *Software Engineering Properties of Languages and Aspect Technologies*, éditeurs Lodewijk Bergmans, Kris Gybels, Peri Tarr, et Erik Ernst, Mars 2005.

[Suvée *et al.*, 2006] Davy Suvée, Bruno De Fraine, et Wim Vanderperren. A symmetric and unified approach towards combining aspect-oriented and component-based software development. In *CBSE*, éditeurs Ian Gorton, George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski, et Kurt C. Wallnau, volume 4063 de *Lecture Notes in Computer Science*, pages 114–122. Springer, 2006.

[Szyperski, 2002] C. Szyperski. *Component Software: Beyond Object-Oriented Programming (2nd Edition)*. Addison-Wesley, 2002.

[Taenzer *et al.*, 1989] D. Taenzer, M. Ganti, et S. Podar. Problems in Object-Oriented Software Reuse. In *Proceedings of ECOOP'89 : European Conf. Object-Oriented Programming*, éditeur S. Cook, volume 821, pages 25–38. Cambridge University Press, july 1989.

[Traverson et Yahiaoui, 2002] Bruno Traverson et Nesrine Yahiaoui. Connectors for CORBA Components. In *OOIS '02: Proceedings of the 8th International Conference on Object-Oriented. Information Systems*, pages 458–463, London, UK, 2002. Springer-Verlag.

[Tremblay et Chae, 2005] Guy Tremblay et Junghyun Chae. Towards specifying contracts and protocols for Web services. In *MCeTech Montreal Conference on eTechnologies*, éditeurs H. Mili et F. Khendek, pages 73–85, January 2005.

[Ungar et Smith, 1987] David Ungar et Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, New York, NY, USA, 1987. ACM Press.

[WCO, 1996] *Workshop on Component-Oriented Programming (WCOP'96)*. Springer-Verlag, 1996.

[Weck et Szyperski, 1996] W. Weck et C. Szyperski. Do we need inheritance, 1996.

[Wettel et Lanza, 2007] Richard Wettel et Michele Lanza. Visualizing Software Systems as Cities. In *VISSOFT'07 (4th IEEE International Workshop on Visualizing Software For Understanding and Analysis)*, éditeur IEEE CS Press, pages 92–99, 2007.

[Wuyts et Ducasse, 2001] Roel Wuyts et Stéphane Ducasse. Composition Languages for Black-Box Components. In *First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, 2001.

[Zenger, 2002] Matthias Zenger. Type-safe prototype-based component evolution. In *Proceedings of the European Conference on Object-Oriented Programming*, Malaga, Spain, June 2002.

[Zenger, 2005] Matthias Zenger. Keris: evolving software with extensible modules: Research articles. *J. Software Maint. Evol.*, 17(5):333–362, 2005.

Publications

[Fabresse *et al.*, 2004] Luc Fabresse, Christophe Dony, Marianne Huchard, et Olivier Pichon. Vers des composants logiciels interfaçables. In *8ème Colloque Agents Logiciels, Coopération, Apprentissage et Activité humaine (ALCAA'04)*, pages 33–48, 17–18 Juin 2004.

[Fabresse *et al.*, 2006a] Luc Fabresse, Christophe Dony, et Marianne Huchard. Connexion non-anticipée de composants en SCL : Une voie pour l'évolution des logiciels. In *Atelier sur l'Evolution du Logiciel*, pages 1–7, 21 Mars 2006.

[Fabresse *et al.*, 2006b] Luc Fabresse, Christophe Dony, et Marianne Huchard. Unanticipated Connection of Components based on their State Changes Notifications. In *International Workshop on Evaluation and Evolution of Component Composition (EECC'06). In proceedings of 18th International Conference on Software Engineering and Knowledge Engineering (SEKE 2006)*, pages 702–708. Knowledge System Institute (KSI), 5–7 July 2006.

[Fabresse *et al.*, 2007a] Luc Fabresse, Christophe Dony, et Marianne Huchard. Foundations of a Simple and Unified Component-Oriented Language. *Journal of Computer Languages, Systems & Structures*, 2007. To appear. doi :10.1016/j.cl.2007.05.002.

[Fabresse *et al.*, 2007b] Luc Fabresse, Christophe Dony, et Marianne Huchard. SCL : a Simple, Uniform and Operational Language for Component-Oriented Programming in Smalltalk. In *Advances in Smaltalk, Proceedings of 14th International Smalltalk Conference (ISC), September 4–8, 2006*, éditeur De Meuter, Wolfgang, volume 4406, pages 91–110. LNCS, Springer-Verlag, April 2007.

[Fabresse, 2004] Luc Fabresse. Programmation de composants interfaçables. In *Actes de JOCM, Journées du groupe Objets, Composants et Modèles*, pages 19–24, 16 Mars 2004.

Résumé : Les composants logiciels sont des entités logicielles réutilisables promettant une réduction des coûts liés au développement, à la maintenance et à l'évolution d'un logiciel. Actuellement, de nombreuses propositions se réclament du mode de développement par assemblage de composants logiciels. Malgré un vocabulaire parfois commun (composant, port, interface, service, connexion, connecteur), ces propositions sont disparates de par leurs origines, leurs objectifs, leurs concepts ou encore leurs mécanismes. De plus, elles restent difficiles à utiliser en pratique par manque de véritables langages de programmation sémantiquement fondés et concrètement utilisables. Devant tant de profusion, nous nous intéressons, dans cette thèse, aux problématiques suivantes : qu'est-ce qu'un langage à composants ? Quels sont les avantages de ces langages ? Comment réaliser un langage à composants ?

Cette thèse propose donc SCL, un langage de programmation à composants permettant de mettre en pratique réellement la programmation par composants (PPC). De par sa conception, SCLse veut être un langage : (*i*) minimal dans le sens où toutes les abstractions et tous les mécanismes qui lui sont intégrés répondent à un besoin identifié ; (*ii*) simple car ses abstractions et ses mécanismes sont de haut niveau ; (*iii*) détaillé car nous avons abordé un ensemble de questions souvent oubliées dans les autres propositions comme l'auto-référence, le passage d'arguments ou le statut des composants de base (collections, entiers, etc) dans un monde unifié ; (*iv*) dédié à la PPC, car prenant en compte les deux préoccupations que nous jugeons majeures en PPC et qui sous-tendent toute l'étude : le *découplage* et la *non-anticipation*.

Le coeur de SCLrepose principalement sur les concepts de composant, de port, de service, de connecteur et de *code glue* ainsi que sur les mécanismes de liaison de ports et d'invocation de service. La séparation des préoccupations au niveau du code occupe une part importante de l'étude qui établit un lien entre la programmation par composants et la programmation par aspects (PPA). SCLpropose dans ce cadre une amélioration des approches dites « symétriques » de la PPA, via une unification des concepts d'aspect et de composant et via un ensemble de différents types de liaisons de ports qui permettent d'utiliser un même composant de façon standard ou de façon transversale. SCLintègre aussi un mécanisme général permettant d'établir des connexions entre composants basées sur les changements d'états de leurs propriétés sans que leurs programmeurs n'aient à écrire une ligne de code spécifique à cet effet. Deux prototypes de SCLsont également présentés, le premier et le plus abouti est écrit en Smalltalk et le second en Ruby.

Mots clés : Langage à composants, séparation des préoccupations, découplage, assemblage non-anticipé, SCL

Abstract: Component-based software development (CBSD) promises costs reduction during the development, the maintenance and the evolution of a software. Currently, a lot of propositions claim to be "component-oriented" but they generally differ in several points like their objectives, their abstractions and their mecanisms. Moreover, it is difficult to use CBSD in practice because of the lack of semantically founded and really usable component-oriented languages (COL). In this context, we address the folowing problematics: What is a COL? What are the advantages of those languages? How to realise a COL?

This work purposes the programming language SCL to enable component-oriented programming (COP) in practice. SCL has been built to be: (*i*) miminal because all its abstractions and mechanisms respond to an identified need; (*ii*) simple because these abstractions and mechanisms are of a high-level; (*iii*) detailled because during the conception of SCL we study a lot of crucial points usually forgotten by other propositions such as self-references, arguments passing based on connections or considering base components (collections, integer, etc) in a unified world; (*iv*) dedicated to COP because it integrates the two key points that we identified: *decoupling* and *unanticipation*.

The core of SCL is built upon the following concepts: component, port, service, connector, *glue code* and the following mechanisms: port binding, service invocation. We also study the separation of concerns in the source code. In this area, SCL improves the "symmetric" aspect-oriented approaches and enables the use of a component as a regular component or a crosscutting component. Deeper in the separation of concerns, SCL also enables component connections based on state changes of their properties. Component programmers do not have to write special code in order to enable this kind of connections. Currently, two prototypes of SCL are available. The more advanced one is written in Smalltalk and the other one has been started in Ruby.

Keywords: Component-oriented language, separation of concerns, decoupling, unanticipated assembly, SCL

LIRMM — 161, rue Ada — 34392 Montpellier Cedex 5 — France