Colloque en l'honneur de Jean-François Perrot : "20 ans après, où en sont les objets"

Programmation post-objets : des langages d'aspects aux langages de composants

P. Cointe, J. Noyé, R. Douence, T. Ledoux, J.-M. Menaud, G. Muller, M. Südholt

Projet OBASCO
École des Mines de Nantes et INRIA
www.inria.fr/recherche/equipe/obasco.fr.html
cointe@emn.fr

17 octobre 2003

Résumé

L'utilisation et la gestion des composants sont au cœur des nouvelles architectures logicielles. Pourtant les métiers sont toujours dans l'attente de composants génériques sur étagères, reconfigurables en fonction de l'application visée et de son environnement. Ces vingt dernières années ont certes vu la programmation par objets devenir le paradigme standard de développement des logiciels. Cependant cette approche de la programmation s'avère aujourd'hui trop limitée pour prendre en compte le changement de niveau conceptuel introduit par le passage à l'échelle des applications sur les réseaux. L'un des problèmes non résolus est de donner de la structure aux logiciels pour mieux en maîtriser la complexité et la maintenance et ce à un niveau de granularité suffisamment important.

De ce point de vue, les objets apportent des réponses incomplètes à l'assemblage et à la gestion du cycle de vie des composants logiciels. Le but de cette présentation est d'introduire la période post-objets en discutant l'évolution des objets vers les composants d'une part et celle des objets vers les aspects d'autre part.

1 Introduction à la période post-objets

Aujourd'hui, la complexité et l'ubiquité des systèmes informatiques autant que l'environnement économique compétitif à l'extrême rendent nécessaire le développement d'architectures logicielles ouvertes et adaptables.

Les logiciels sont de complexité croissante, leur réalisation nécessite des investissements toujours plus importants et pourtant le délai d'arrivée sur le marché (*time-to-market*) est un facteur déterminant de succès. Aussi, la production à flux tendu de composants logiciels réellement *plug and play* est apparue une réalité et suscite une demande croissante en terme d'applications réparties construites par assemblage de composants hétérogènes. Parmi les secteurs les plus concernés figurent ceux liés à l'utilisation du réseau comme les télécommunications, le commerce électronique, l'enseignement à distance, mais également de nouvelles disciplines transversales comme la bio-informatique.

De fait, la définition d'architectures logicielles construites à partir de composants pris sur étagère est une idée qui a fait son chemin et qui constitue aujourd'hui un enjeu majeur tant au plan scientifique qu'industriel. Cet enjeu est d'autant plus important que les technologies logicielles associées sont à la frontière de plusieurs domaines clefs [Cen00] :

- les « objets » communicants et autonomes ;
- le transport des données ;
- la gestion des données et du contenu ;
- l'informatique distribuée ;
- le génie logiciel.

1.1 Pourquoi des architectures adaptables ?

> *L'œuvre d'art est un message fondamentalement ambigu, une pluralité de signifiés qui coexistent en un seul signifiant ... Pour réaliser l'ambiguïté comme valeur, les artistes contemporains ont souvent recours à l'informel, au désordre, à l'indétermination des résultats. On est ainsi tenté d'établir une dialectique entre forme et ouverture, qui déterminerait dans quelles limites une œuvre d'art peut accentuer son ambiguïté et dépendre de l'intervention active du spectateur sans perdre pour autant sa qualité d'œuvre. Il faut entendre par œuvre, un objet doté de propriétés structurales qui permettent, mais aussi coordonnent la succession des interprétations, l'évolution des perspectives.*
>
> *Umberto Eco - Opera Aperta (L'œuvre ouverte) - 1962.*

Nous voudrions revenir ici sur les deux raisons qui nous font dire que toute architecture logicielle devrait posséder les propriétés d'ouverture et d'adaptabilité.

La première raison est que lors du développement de ces architectures, il existe une tension très forte entre forme et ouverture. Elle se traduit en deux désirs contradictoires : celui de satisfaire des critères de qualité, fiabilité, sécurité, portabilité et efficacité, en encapsulant les propriétés du logiciel (la boîte noire), et celui d'introduire des points d'entrée (« crochets » ou « coupes ») susceptibles de permettre l'adaptation ultérieure de cette architecture à des domaines, des applications ou des environnements d'exécution non prévisibles a priori. Ces points d'entrée servent finalement à retarder au plus tard les choix les plus pertinents relativement à une stratégie donnée [ABC+96].

La deuxième raison résulte d'une comparaison avec le développement d'autres types d'industrie. Il semble en effet que, après une période d'artisanat de l'industrie du logiciel, culminant avec l'avènement des objets (période 65/95), nous assistions aujourd'hui au début de la phase de standardisation préparatoire à celle de la production en série. Celle-ci devrait conduire à une dernière étape dite de personnalisation permettant *in fine* d'adapter le logiciel à ses usages.

1.2 L'héritage des objets

L'approche objet, dans son pragmatisme, a démontré qu'il était possible de développer des logiciels complexes en utilisant pour « formalisme premier » l'expression, dans le langage de programmation même, du savoir-faire des programmeurs experts [1]. D'où le succès des schémas de conception (*design patterns*) comme outils empiriques de transmission de ce savoir-faire dans le domaine du génie logiciel. Bien sûr, l'ajout de contraintes supplémentaires comme les types pour contrôler la validité de ces programmes, ou les classes et leur hiérarchie pour structurer leur représentation, contribue à rigidifier la forme. Mais ces contraintes ont provoqué « en réaction » l'émergence de techniques comme la métaprogrammation, la réflexion et la programmation par aspects [2] dans le but d'« ouvrir » et d'étendre les langages à objets [Coi00].

Aujourd'hui, Java est un bon exemple d'architecture tentant d'assumer ses contradictions, à savoir d'être un langage ouvert et sécurisé à la fois [3]. Ainsi, un nombre très significatif de travaux de recherche portent sur la résolution de cet antagonisme, donnent sa vitalité à ce langage et inspirent déjà ses successeurs dont le langage C#.

Pourtant, avec beaucoup d'autres, nous constatons que l'approche objet n'est plus suffisante pour exprimer sous forme de modules orthogonaux et réutilisables les services non fonctionnels de beaucoup d'applications [4] comme la persistance, la concurrence, la tolérance aux fautes... L'approche objet n'est pas non plus suffisamment élaborée pour rendre compte de manière transparente de la répartition et de la mobilité de ses applications sur le réseau. Il est donc nécessaire d'introduire d'autres formes de modularité dans la construction des logiciels en développant des notions nouvelles comme celle des *aspects* (modularité transversale liée au découplage des propriétés non fonctionnelles) et des *composants* (modularité intrinsèque liée à l'encapsulation forte des propriétés fonctionnelles et à la neutralité vis-à-vis des langages d'implémentation) sans renoncer pour autant à l'ouverture, gage de flexibilité. Nous postulons donc que l'adaptabilité tant recherchée sera obtenue en appliquant aux aspects et aux composants les techniques introduites avec l'implémentation fine des langages de programmation (à objets) dont la réflexion, l'analyse de programmes, la génération de code et l'évaluation partielle.

1.3 Les enjeux

Comme U. Eco, nous pensons qu'il s'agit de donner une forme, une structure, au matériau, que celui-ci soit solide, acoustique ou logiciel. Intuitivement, cette structure doit être suffisamment flexible pour s'adapter dynamiquement au contexte au moment de son exécution, et ce qu'il s'agisse d'une partition musicale [5] ou d'un programme informatique. Il est à noter que nous parlons ici d'exécution par analogie avec la musique. Il est clair que dans les cas des composants logiciels, il s'agira de s'intéresser à d'autres « temps » de leur cycle de vie : par exemple, ceux de l'implémentation, de la configuration, de l'assemblage, du déploiement... En conséquence, nous devrons également considérer différentes formes d'adaptabilité : adaptabilité statique, adaptabilité dynamique, auto-adaptabilité et surtout proposer une méthodologie pour un développement incrémental.

1. Nous pensons à la devise de P. Deutsch, l'implémenteur en chef de Smalltalk : *"Program first, think later"*.
2. Cf. `http://aosd.net`.
3. Voir à ce propos les évolutions du package `java.lang.reflect`.
4. Par opposition au code fonctionnel d'une application défini comme le programme réalisant le service (métier) de base rendu par l'application.
5. *Se renouveler? Oui, mais selon une évolution organique qui développe, étend les possibilités, change les perspectives. En musique, comme en peinture, les exemples sont rares de cette continuité qui préserve l'évolution et la diversité: Klee est l'un des cas exceptionnels où la pensée et l'invention poursuivent sans trêve leur investigation et transforment leur manière de voir et de faire, à tel point qu'il aurait été improbable et hasardeux de prévoir leur trajectoire (page 149 de [Bou89]).*

Nous pensons critique de mener une réflexion de fond sur l'art et la manière d'architecturer à un niveau macroscopique et de manière flexible les logiciels. Nous entendons donner corps à l'image des « composants réutilisables sur étagère » en étudiant la dialectique entre forme et ouverture qui doit se décliner aujourd'hui entre standardisation et personnalisation.

L'approche « langage » est un moyen privilégié pour mener cette réflexion en complémentarité avec l'approche « modèle à la UML/MDA». Cette démarche conduit naturellement à une caractérisation précise et exécutable des concepts manipulés et de leurs interactions. Elle supporte naturellement la construction de logiciel à partir de méthodes et outils éprouvés. L'intérêt de cette approche est qu'elle est largement orthogonale aux standards technologiques EJB, .NET et XML qui sont déjà largement utilisés industriellement mais sans réels fondements scientifiques.

1.4 Vers de nouvelles architectures logicielles

En reprenant les apports et les évolutions du paradigme objet, nous postulons qu'il est possible d'accroître la productivité globale de l'industrie du logiciel en proposant des architectures logicielles plus ouvertes et de meilleure qualité. Pour ce faire, il nous faut explorer les nouvelles techniques pour architecturer des logiciels adaptables construits à base d'aspects et de composants et respectant les principes de simplicité, modularité, minimalité et extensibilité.

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

2 Des objets aux composants

"Gropius created innovative designs that borrowed materials and methods of construction from modern technology. This advocacy of industrialized building carried with it a belief in team work and an acceptance of standardization and prefabrication. Using technology as a basis, he transformed building into a science of precise mathematical calculations[6]*."*

2.1 Bénéfices et limites des objets

L'approche objet a fait la preuve qu'elle facilitait la production de logiciel, d'une part en permettant d'organiser cette production, de l'analyse à l'implémentation, autour d'entités, classes et objets, représentant directement des concepts, relativement stables, du monde réel, d'autre part en fournissant un certain nombre d'éléments d'architecture, schémas de conception et canevas, permettant de structurer les applications tant au niveau microscopique que macroscopique.

Toutefois, parallèlement à la maturation de l'approche et à sa pénétration sur le marché, des limitations importantes sont apparues. D'abord, une réutilisation à grande échelle du logiciel s'est heurtée aux difficultés d'une réutilisation de type *boîte blanche* (les implémentations des entités réutilisées sont visibles et modifiables), fondée sur l'utilisation de l'héritage et d'un langage d'implémentation privilégié. Ensuite, la montée en puissance du réseau des réseaux a rendu nécessaire le passage des objets aux *objets distribués* [OHE96],

6. Pour comprendre ce que devrait être un composant logiciel, il nous semble important d'observer comment d'autres disciplines comme l'architecture, ont abordé le problème de l'industrialisation et de la fabrication en série En particulier, le Bahaus et W. Gropius nous semblent constituer une source d'inspiration historique pour mieux appréhender l'évolution des objets aux composants.

difficiles à maîtriser par des utilisateurs « métier » car requérant une programmation spécifique des services techniques liés à la distribution.

La prise en compte de ces difficultés a résulté en un changement de perspective qui a conduit des objets aux *composants* [Szy02]. Il s'agit essentiellement, comme l'expliquait déjà très bien M.D. McIlroy en 1968 [McI68], d'industrialiser la réutilisation, ce qui suppose de mettre sur pied à la fois une industrie et un marché des composants, comme il en existe dans les autres secteurs (l'automobile et la construction, par exemple) tout en faisant bénéficier les utilisateurs des bonnes propriétés du logiciel (facilité de manipulation tant au niveau concret qu'abstrait).

Dans cette optique, les composants deviennent un niveau de rupture explicite entre un niveau de programmation à petite échelle (*in the small*), bien couvert par les langages à objets, à l'exception notable de la représentation et de la réutilisation des *aspects* (voir section 3), et un niveau de programmation à grande échelle (*in the large*), beaucoup moins bien couvert par ces mêmes langages (malgré l'avancée offerte par la notion de canevas). C'est à l'échelle des composants que l'on va parler d'architecture logicielle.

2.2 La notion de composant logiciel

L'introduction de la notion de composant permet donc de distinguer clairement une étape de production et une étape de consommation (ou utilisation) du logiciel. Cette dernière consiste à *assembler* des composants (statiquement ou dynamiquement). On parle aussi de *composition* (et de *déploiement* pour un assemblage dynamique). Suivant le *modèle* de composant, cet assemblage peut fournir un nouveau composant, réutilisable dans un nouvel assemblage, ou pas.

Afin de ne pas tomber dans le piège de la réutilisation de boîtes blanches, un composant possède un certain degré d'opacité. À la limite, on aboutit à de la réutilisation de type *boîte noire* (l'implémentation de l'entité réutilisée est invisible et ne peut pas être modifiée[7]). C'est le type de réutilisation mis en jeu lors de l'utilisation de bibliothèques.

Le cas intéressant est toutefois le cas intermédiaire de la réutilisation de type *boîte grise* qui va permettre d'adapter le composant à son contexte d'assemblage (donc d'augmenter les possibilités de réutilisation) tout en guidant cette adaptation (afin de simplifier la réutilisation et de la rendre plus sûre). Cette idée, bien qu'implicite dans les définitions de composant (voir, par exemple, [Szy02, CH01, BBB+00]) est, de notre point de vue, caractéristique de la notion de composant.

Pratiquement, un composant est structuré en un *noyau* boîte noire et une *interface* boîte blanche. Cette interface a une existence syntaxique : elle fournit une spécification du composant à l'utilisateur. Elle a aussi une existence sémantique, se transformant (au moins conceptuellement) en *enveloppe* (*wrapper*) lors de l'assemblage.

L'interface du composant guide l'assemblage d'un point de vue syntaxique (ou structurel) d'*interconnexion* et d'un point de vue sémantique (ou comportemental) d'*interaction*, incluant des propriétés de ces interactions, par exemple des propriétés de *qualité de service*. Cette interface est *contractuelle* : elle inclut des déclarations de *propriétés* ou *contrats* [BJPW99] qui aident à la sélection des composants à assembler et au rejet des assemblages non conformes.

L'adaptation du composant à son contexte d'assemblage (nous parlerons par la suite de *configuration*) suppose que l'interface est paramétrée. C'est la prise en compte de ces paramètres qui réalise l'adaptation. Dans le cas de composants distribués, il s'agit notamment de configurer l'utilisation des services techniques sous-jacents. Considérer ces services comme des aspects (voir section 3) introduit un nouveau type de composition.

7. Peu importe ici que cette implémentation soit fournie sous forme de code natif, de code pour une machine virtuelle ou de code source.

Ces idées sont généralisables à des cycles de vie plus complexes qu'un simple cycle de vie production/assemblage. Ainsi, un cycle de vie classique voit un composant assemblé statiquement, puis déployé, instancié et finalement exécuté. Chaque étape définit un nouveau contexte d'utilisation du composant (plusieurs dans le cas d'un assemblage incrémental ou de conditions changeantes d'exécution) auquel celui-ci peut être adapté.

2.3 État des lieux des travaux académiques

2.3.1 Langages de modules

Le concept de module est d'autant plus proche du concept de composant que l'on considère des systèmes de modules d'ordre supérieur à la ML (voir, par exemple, [Ler95]) qui fournissent des possibilités d'adaptation par paramétrage. Ces systèmes de modules sont généralement associés à des langages spécifiques mais leur applicabilité est bien plus large [Ler00]. L'introduction du concept de *mixin*, initialement développé pour généraliser l'héritage dans les langages à objets [BC90], autorise de plus les définitions récursives de modules et leurs liaisons retardées [FF98, HL02]. Les modules *mixins* fournissent donc une base intéressante pour la définition d'un langage d'interconnexion de composants.

2.3.2 Langages de description d'architectures

Bien que traitant avant tout de la description plutôt que de l'implémentation des architectures logicielles, ces langages [SG96, MT00] fournissent une description explicite des interactions entre *composants* au travers de *connecteurs* ainsi que l'architecture d'un assemblage (ou *configuration*) de composants et de connecteurs. Dans un certain nombre de propositions, cette architecture peut être modifiée dynamiquement par l'intermédiaire d'un *langage de modification d'architecture* (voir, par exemple, [vdHMRRM01, WLF01]).

Le problème de ces langages est que les descriptions fournies sont découplées de leurs implémentations. Il est alors difficile de garantir que les propriétés d'une description sont respectées par l'implémentation. Une solution consiste à munir un langage de programmation d'un niveau de « programmation architecturale » de manière à ce que les préoccupations architecturales soient directement prises en compte lors de l'implémentation. C'est l'approche suivie par ArchJava [ACN02] qui étend Java avec une notion de composant garantissant l'absence d'effets de bord entre composants.

2.3.3 Systèmes de gestion de versions

La notion de configuration a été bien étudiée, principalement du point de vue de la construction de systèmes et de la gestion de versions, par la communauté travaillant sur la gestion de configuration [Est00]. Elle a également été mise en œuvre de manière empirique pour les langages Smalltalk et Java par l'intermédiaire des environnements de développement intégrés à la Envy/VisualAge [TJ88].

2.3.4 Évolution des langages à objets

Outre ArchJava, déjà mentionné, un certain nombre d'extensions de Java introduisant la notion de composants ont d'ores et déjà été proposées. Un des modèles les plus achevés est Jiazzi [8] [MFH01]. Un composant Jiazzi encapsule du code binaire Java tout en supportant des connexions typées entre composants. Cette proposition s'appuie sur un système de modules *mixins*, les *units* [FF98]. L'enveloppe de ces composants reste toutefois très pauvre et ne fournit aucune aide quant à l'ajout de services techniques ou la spécification de protocoles d'interaction entre composants. Deux autres propositions proches sont

8. `http://www.cs.utah.edu/plt/jiazzi`.

JL (*Java Layers*) [CL01] et le modèle de composants typés ComponentJ [SC00]. JL est une extension de Java incluant polymorphisme paramétrique et mixins. ComponentJ se présente comme un calcul d'un noyau impératif typé qui permet d'associer à chaque composant plusieurs interfaces et a la propriété, contrairement aux modèles de composants d'ArchJava, Jiazzi et JL, de considérer les composants comme des entités de première classe.

Bien que plus éloignés de nos préoccupations les composants boîtes blanches proposés dans les travaux de K. Lieberherr, D. Lorenz et M. Mezini [MSL00, LLM99] sont très instructifs quant aux rapports entre canevas, composants et aspects et à l'importance du paramétrage des composants.

Finalement, un certain nombre de travaux issus de la communauté des langages à objets sont aussi de bonnes sources d'inspiration pour ce qui est de la description des interactions [vdBL89, Nie95, YS97]. Ces travaux introduisent la notion de protocole explicite entre objets, décrivant les séquences valides d'appel de méthodes. Ces protocoles peuvent être vus comme des automates à états finis mais aussi intégrés au langage de programmation sous la forme de types.

2.3.5 Intergiciels réflexifs

Dans le cadre de la réflexion, il est naturel de gérer les modifications architecturales en réifiant l'architecture en un métaniveau qui peut être inspecté et modifié à l'exécution. Ces principes ont notamment inspirés les infrastructures à composants K [DC01] et Fractal [CBS02], à comparer avec l'approche plus traditionnelle de [VM01].

2.4 État des lieux des solutions industrielles

Au vu des enjeux industriels, de gros efforts ont été accomplis pour aider au développement d'applications à base de composants. Pour l'instant, ces efforts ont essentiellement portés sur la mise en place d'infrastructures supportant diverses variantes de la notion de composant :

- composants graphiques [Sun97] ;
- composants serveurs[9] : les composants COM+ [Ses00], les *Enterprise JavaBeans* (EJB) [DYK01], le *Corba Component Model (CCM)* [Gro02] ;
- services de la toile (*web services*) présents à la fois dans la plateforme .NET de Microsoft et la plateforme Sun (sous la forme de l'architecture *Sun Open Net Environment* (*Sun ONE*) [Sun01].

2.4.1 Critique

Globalement, ces infrastructures, développées avec un souci légitime de compatibilité de l'existant et d'accès au marché, mais aussi des besoins spécifiques, pas toujours clairement explicités[10], sont très complexes à appréhender et à utiliser, difficiles à étendre et à généraliser.

Cet état de fait est principalement dû à l'absence d'un véritable *modèle de composants*, bien que ce terme soit souvent utilisé, faisant la part des choses entre des détails technologiques d'implémentation (comme le recours à de nombreux autres « standards » de la toile) et les concepts, ni à plus forte raison de *modèle de services*, ou *service* est

9. Nous préférons parler de composant serveur plutôt que de composant distribué dans la mesure où ces propositions intègrent en fait de manière ad hoc des besoins spécifiques aux architectures multi-couches des serveurs d'applications.

10. Pour les composants serveurs, répondre aux besoins des grappes de serveurs du commerce en ligne. Pour les services de la toile, passer d'un modèle de vente de logiciel à un modèle de vente de services.

compris ici comme une généralisation des services techniques mentionnés ci-dessus. Le premier contact avec les spécifications de ces solutions, quand elles existent (c'est le cas des EJB [DYK01] et des composants Corba [Gro02]), est spécialement éprouvant. Il s'avère en effet ardu de trouver la réponse à des questions élémentaires comme « est-il possible d'obtenir un composant par assemblage de composants? » ou encore « dans quelle mesure puis-je m'assurer de la validité d'un assemblage? ».

Ces problèmes se retrouvent à quatre niveaux essentiels :

- la description des composants eux-mêmes au travers de leur interface, plus particulièrement en ce qui concerne la connexion de services et les interactions ;
- la description des assemblages de composants à un niveau permettant de raisonner sur l'architecture de l'assemblage et de garantir des propriétés globales de cohérence des assemblages ;
- la gestion dynamique des assemblages ;
- la gestion de l'adaptation. Alors que les performances sont un souci constant, rien n'est entrepris pour exploiter le fait que le noyau du composant est utilisé dans un contexte particulier. La compilation du composant a lieu en une seule étape, à la fabrication, dans un contexte générique, sans prendre en compte le contexte spécifique dans lequel le composant sera finalement utilisé.

2.4.2 À propos de l'approche dirigée par les modèles

Le besoin de travailler à un plus haut niveau d'abstraction a conduit l'*Object Management Group* (OMG) [11] à proposer l'approche *Model-Driven Architecture* (MDA) [ORM01] qui vise à s'affranchir des plateformes technologiques. Cette approche consiste à décrire des architectures logicielles à l'aide de langages de « modélisation », principalement UML, pour ensuite « compiler » ces modèles, selon les besoins, vers une ou plusieurs des différentes infrastructures précédemment évoquées.

Cette approche bien que fédératrice et ambitieuse a pour inconvénient de rendre difficile la réflexion de fond en la noyant sous des considérations technologiques dues au contexte de définition tant des outils de modélisation que des infrastructures cibles.

3 Des objets aux aspects

Separation of concerns is a key guiding principle of software engineering. It refers to the ability to identify, encapsulate, and manipulate only those parts of software that are relevant to a particular concept, goal or purpose. Concerns are the primary criteria for decomposing software into smaller, more manageable and comprehensible parts that have meaning to software engineers [EFA01].

Depuis 1996, sous l'impulsion d'une partie de la communauté des langages à objets, la programmation par aspects s'affirme progressivement comme un nouveau paradigme de programmation [ABC+96, KLM+97]. Nous développons ici, une explication de cette évolution des objets vers les aspects.

3.1 Un besoin récurrent : séparer les préoccupations

La définition de systèmes complexes dans des domaines aussi différents que les interfaces homme machine, les architectures logicielles, les bases de données, les systèmes distribués, ..., a révélé un problème récurrent : l'existence de fonctionnalités que les abstractions

11. Le consortium à vocation industrielle à l'origine de Corba.

du domaine ne permettent pas de représenter convenablement [Kru95]. Ces fonctionnalités, aujourd'hui appelées *aspects*, sont disséminées dans les différentes composantes d'une application et ne bénéficient donc pas d'une encapsulation adéquate tant au niveau des modèles de conception que des langages de programmation.

Cette faiblesse dans l'expression d'un point de vue particulier, est manifeste dans la réalisation « objet » du serveur Apache, telle qu'elle est décrite par G. Kiczales dans son tutoriel sur AspectJ[12]. Certaines préoccupations, comme l'analyse syntaxique de termes XML ou la reconnaissance de motifs dans des URL, sont bien encapsulées dans une seule classe. Par contre, le code implémentant la notion de session de connexion est disséminé dans une cinquantaine de classes différentes et ce malgré la présence d'un mécanisme de réutilisation comme l'héritage.

Cette incapacité à regrouper au sein d'une même entité logicielle le programme traitant d'un même problème introduit des surcoûts liés à la redondance de code non ou mal factorisé, à la maintenance de code disséminé dans des parties de l'application sémantiquement déconnectées et à l'absence de lisibilité des solutions [HL95]. Elle conduit à envisager puis à développer une nouvelle forme de programmation dite *par aspects* pour laquelle il est possible : (i) d'identifier, d'encapsuler et de manipuler les propriétés relevant d'un domaine particulier sous forme de « modules » indépendants, (ii) d'insérer ensuite ces modules au bon endroit dans l'application pour qu'elle prenne en compte un nouveau concept, un nouveau but, un nouveau besoin, un nouveau service ... .

3.2 (Bénéfices) et limites des objets

3.2.1 Limites des mécanismes de base

L'une des faiblesses de l'approche par objets apparaît donc dès qu'il s'agit d'exprimer des propriétés transversales à une hiérarchie de classes, propriétés dont l'implémentation traverse cette hiérarchie de spécialisation.

On touche alors aux limites de la modularité et de la réutilisation permises par l'encapsulation et l'héritage.

Pourtant, de nombreux travaux ont été menés pour introduire une meilleure séparation des préoccupations[13]. Ils se caractérisent par la volonté de pousser les mécanismes sous-tendant le modèle objet à leurs limites. Ainsi, l'étude des actes des conférences OOPSLA et ECOOP, montre que :

- dès 1986, les encapsulateurs de G. Pascoe ont donné lieu à une série de réalisations sur l'interception des envois de message pour adapter/contrôler le comportement des objets receveurs et introduire de nouveaux traitements (préoccupations) en cours d'exécution. On pourra en particulier citer l'introduction de messages asynchrones en Actalk, la réalisation des filtres de composition en Sina, une implémentation de l'héritage multiple en Smalltalk76 pour Thinglab ;
- la mise en œuvre de l'héritage multiple dans des langages comme les Flavors et CLOS a permis d'appréhender via les mixins et la combinaison des méthodes la difficulté à expliciter puis à composer des éléments de structure et de comportement. Fort de cette expérience les concepteurs de Java ont réduit l'usage de l'héritage multiple aux interfaces, c'est-à-dire aux simples spécifications ;
- le développement de la programmation par métaclasses en CLOS et Smalltalk a reposé sur l'utilisation du mécanisme d'instanciation pour exprimer au métaniveau des propriétés de classes réutilisables comme par exemple la persistance, la mémoïsation ou la sérialisation des instances. Ce style de programmation a révélé le problème de compatibilité des métaclasses dû au couplage fort entre instanciation et héritage.

12. `http://www.aspectj.org`.
13. En anglais : *Separation Of Concern* ou SOC.

3.2.2 À propos du schéma de conception MVC

Indépendamment des travaux visant à étendre le modèle objet, le développement du *Model-View-Controller*[14] Smalltalk a constitué dès 1981, une autre mise en œuvre - mais au niveau méthodologique - de la séparation des préoccupations. Elle nous paraît aujourd'hui exemplaire car i) elle a contribué à l'émergence de la notion d'événements, ii) elle préfigure la programmation par aspects d'aujourd'hui.

L'idée est de séparer dans la réalisation d'une application, la partie relevant de son domaine propre (par exemple la modélisation d'un compte bancaire), des parties traitant de son affichage (ses vues sur l'écran) et de ses interactions avec les périphériques externes (la souris et le clavier pour opérer des dépôts ou retraits bancaires). Le MVC préconise donc de concevoir/implémenter l'application en toute indépendance (sous forme d'une hiérarchie de classes), puis de se préoccuper ensuite seulement des problèmes d'affichage et de contrôle. Au niveau implémentatoire se pose bien sûr la question de coupler les éléments de la trilogie de manière à ce que, lors d'un changement du modèle provoqué par un contrôleur, l'information soit automatiquement répercutée à la vue pour son réaffichage. Concrètement, l'apport du MVC au domaine de la programmation par aspects est d'avoir établi qu'à condition d'instrumenter le source d'une classe (accès en écriture aux variables d'instance), et de disposer d'un mécanisme de notification à base d'événements, il était possible de réaliser dans le domaine des interfaces homme-machine une bonne séparation des préoccupations.

3.2.3 Apport de la réflexion

The most important contribution of the reflection community is to clearly illustrate the potential of separation of concerns [Tho02].

Une autre contribution des « objets » au domaine de la séparation des préoccupations [15], résulte des travaux menés dans le domaine des architectures réflexives. L'approche réflexive part de l'hypothèse qu'il est possible de décomposer une application en un niveau de base représentant le « pourquoi » de celle-ci, c'est-à-dire sa raison première, et un ou plusieurs métaniveaux explicitant le « comment » , c'est-à-dire la description des entités et mécanismes utilisés. Il s'agit donc bien de séparer le cœur générique de l'application de sa description et de son paramétrage au métaniveau [BSLS01].

Dans le cas d'un langage de programmation, ses principaux éléments doivent être réifiés au métaniveau. Par exemple, la continuation et l'environnement en Scheme, les classes, leurs méthodes et les messages d'erreur en Smalltalk, les classes et les fonctions génériques en CLOS, la description des classes et de leurs membres en Java. Dans le cas d'un langage à classes, la réification est opérée sous forme de classes particulières dont les instances sont nommées métaobjets et dont les méthodes définissent les protocoles des métaobjets ou MOP. L'idée est ensuite d'utiliser ces classes et leurs MOP pour décrire au métaniveau les mécanismes de base, spécialiser ces mécanismes par héritage, puis réfléchir leurs applications au niveau de base. On distingue alors réflexion de structure et de comportement selon que l'on s'attaque à la représentation des propriétés statiques des objets (encapsulation, initialisation et héritage des champs) ou à celles dynamiques de leurs interactions (envoi de messages, ...). Par ailleurs, l'on discernera entre les protocoles dits d'introspection permettant simplement d'examiner la structure et le comportement et ceux dits d'intercession permettant de les modifier.

Au niveau académique, l'expression de différents MOP a mis en exergue les différents aspects techniques qui pouvaient être ajoutés réflexivement à un système [BL01]. En particulier, J.-P. Briot et R. Guerraoui discutent de l'introduction de la distribution par

14. Devenu depuis l'un des archétypes des schémas de conception.
15. Voir Parnas http://www.acm.org/classics/may96

l'intermédiaire de bibliothèques spécialisées de métaobjets [BG96]. Ainsi, les métaobjets du système FRIENDS permettent de gérer tolérance aux fautes et aux intrusions [FP98]. Ces travaux sur les MOP ont également permis de distinguer les différents temps du cycle de vie des objets (compilation, chargement, exécution ...) pendant lesquels pouvaient s'opérer l'introduction des « coupes » [Coi99, TNCC03], ainsi que d'aborder les problèmes de composition et d'interaction des métaobjets [MMC95, BS99].

3.3 Programmer avec des aspects

3.3.1 Le modèle général

Il fonde la définition d'une application sur un programme « principal » et un ensemble d'aspects indépendants. Ce programme, dit *de base*, détermine la sémantique métier de l'application. Ce programme et les aspects sont ensuite entrelacés à l'aide d'un outil spécifique, le *tisseur d'aspects* (*aspect weaver*) pour produire un nouveau programme exécutable adapté à un usage particulier.

La définition du programme de base utilise un langage traditionnel (en général à objets, typiquement Java). Ce programme sert de référentiel et, comme dans le cas de systèmes réflexifs, il est instrumenté pour introduire les *points de jonction* servant à ancrer les aspects. Les aspects, à leur tour, sont définis séparément soit à l'aide d'un langage généraliste, soit à l'aide d'un ou plusieurs langages dédiés. Ce modèle fait apparaître - a minima - les besoins pour l'expression :

- d'un *langage de coupes* pour définir où et quand l'aspect peut modifier le programme de base. Il peut s'agir d'un sous-ensemble des points de jonction caractéristiques de l'exécution d'un programme comme la modification d'un champ d'un objet (cf. le MVC Smalltalk), la réception d'un appel de méthode par un objet (cf. les encapsulateurs ou les métaobjets), la création d'un objet (cf. les métaclasses et leurs méthodes d'allocation) ;
- d'un *langage d'actions* pour décrire comment l'aspect contribue à l'exécution du programme de base. Une action peut s'exprimer, comme en AspectJ, par la déclaration d'une combinaison d'appels de méthodes exécutée au moment d'une coupe (cf. page 335 de [KHH+01]) ;
- finalement, du *langage d'aspects* lui-même pour décrire sous forme d'un élément réutilisable un aspect, c'est-à-dire la déclaration des coupes et des actions associées ainsi que (éventuellement) un état représentant le contexte d'exécution. Ce langage doit en outre expliciter les relations entre aspects, comme par exemple, la relation `dominate` d'AspectJ, indiquant que les actions d'un aspect interviennent avant celles d'un autre aspect.

3.3.2 Les langages dédiés

Un langage dédié (*Domain-Specific Language*, ou DSL) est un langage de programmation spécifique à un domaine d'application. Un langage dédié permet de capturer l'expertise sur son domaine ; il est alors possible d'effectuer des vérifications relatives à la cohérence des programmes du domaine considéré. Ces langages ont démontré leur intérêt dans des domaines qui requièrent une expertise très spécifique tels que les systèmes d'exploitation, en simplifiant la programmation de sous-systèmes tels que les pilotes [MRC+00], le réseau [HML02, MLTR03] ou les ordonnanceurs [BM02, BDMS02]. Dans le contexte de la programmation par aspects, il peut sembler naturel d'associer à chaque aspect un langage dédié. En effet, non seulement les préoccupations des progammeurs du logiciel de base et des aspects sont différentes, mais également les expertises. À ce titre, la vérification de propriétés peut être un avantage crucial lorsque différentes expertises doivent cohabiter. Il

est toutefois à noter que les réalisations actuelles ont tendance à utiliser un seul et même langage pour représenter aspects et code de base.

3.4 État des lieux

Le besoin de structurer les applications en aspects se manifeste aux trois grandes étapes de la modélisation, de l'implémentation et de l'exécution. En reprenant ces trois phases, nous recensons des travaux de nature essentiellement académiques.

3.4.1 Au niveau des modèles

Les équipes de M. Aksit [Aks96] et de J.-M. Jézéquel [HJMAF99, HJPP02] travaillent à l'introduction de la notion d'aspects [EFA01] dès la phase de conception. Ainsi, Triskell propose d'utiliser des spécifications reposant sur des diagrammes UML pour modéliser le programme de base et les aspects. Ces modèles sont ensuite transformés en des squelettes de programmes. Malheureusement, ces travaux ne sont pas encore transposables directement aux phases de mise en œuvre du fait de l'absence de mécanisme support au niveau des langages d'implémentation.

3.4.2 Au niveau des langages

Le problème de la séparation des préoccupations a longtemps été abordé de manière plus ou moins ad hoc, depuis l'insertion manuelle de code jusqu'à la définition de canevas pour composants logiciels. Dans des cas simples mais fréquents, l'aspect est intégré dans le langage à l'aide d'un ensemble de mots-clés (par exemple en Java `synchronized` pour la synchronisation, `throws/catch` pour la levée/capture d'exceptions). Les travaux les plus récents se situent soit dans le prolongement de l'approche réflexive [BSL03] soit dans la conception de nouveaux langages à objets prenant en compte les notions de coupes, d'actions et d'aspects [EFA01, Tho02].

Approche réflexive

- AspectJ propose une extension de Java introduisant explicitement la notion d'aspects. La définition d'un aspect est inspirée de celle d'une classe mais intègre un langage élémentaire de coupes (permettant de parler en terme d'expressions logiques principalement de l'appel et du retour des méthodes/constructeurs, de l'accès aux champs) et un langage d'actions permettant d'exprimer celles-ci sous forme de corps de méthode Java[16].

  Au niveau implémentatoire, un tisseur opère par transformation des programmes sources.

- Les filtres de composition (*composition filters*)[17] s'inscrivent dans la lignée des encapsulateurs de la programmation par objets[18]. La simplicité du modèle permet la composition séquentielle de ces filtres. Cependant, un filtre étant attaché à un seul objet, il est difficile de prendre en compte des aspects concernant simultanément plusieurs objets [AWB+94].

16. *Dynamic crosscutting in AspectJ is based on a small but powerful set of constructs. Join points are well-defined points in the execution of the program; pointcuts are a means of referring to collections of join points and certain values at those join points; advices are method-like constructs used to define additional behavior at join points; and aspects are units of modular crosscutting implementation, composed of pointcuts, advices and ordinary Java member declarations.* Page 329 de [KHH+01].

17. `http://trese.cs.utwente.nl/composition_filters`.

18. La métaphore des filtres est inspirée de la photographie et indique que ceux-ci fonctionnent comme des intercepteurs de messages permettant de contrôler les appels de méthodes (envoi ou réception de message). Voir 3.2.1.

– JAC[19] permet de choisir les coupes liant programme de base et les aspects en configurant son système d'exécution [PSLF01]. Une contribution notable est sa bibliothèque d'opérateurs spécialisés de composition. Par contre, JAC ne s'intéresse pas à l'utilisation de langages dédiés pour la définition des aspects.

Évolution des langages à objets

– Demeter[20] étudie depuis l'origine l'expression des relations ou interactions entre objets indépendamment de la hiérarchie des classes [Lie94]. Récemment, ces interactions ont été isolées sous la forme d'un aspect. Les derniers résultats obtenus portent sur la spécification des stratégies de parcours du graphe d'interactions dynamiques.

– Définis par le même groupe, les *aspectual components* proposent un modèle simple de composants et de connecteurs qui sont mis en œuvre sous forme de canevas à objets [MSL00]. Cette notion rend explicite, à la façon des modules, les interfaces fournies et requises par un composant [ML98]. Ces composants peuvent être considérés comme des aspects et leur composition comme un tissage. Cette première proposition ouvre des perspectives intéressantes quant à la convergence de la programmation par aspects et par composants mais reste très ad hoc (c'est-à-dire liée à la structure des canevas sous-jacents) et focalisée sur une vision structurelle plutôt que comportementale des programmes.

3.4.3 Au niveau des applications

Intergiciels industriels Les modèles de composants Corba, EJB et .NET font grand usage de la notion de service qu'il faut rapprocher de celle d'aspect. En effet, ces modèles offrent comme service de base la communication entre objets distribués, complété par d'autres services supports aux mécanismes de transactions, de sécurité, ou de persistance ... Ces services fournissent les abstractions nécessaires aux programmeurs qui doivent cependant insérer manuellement, aux bons endroits (c'est-à-dire aux points de coupe), les appels correspondants (c'est-à-dire les actions) dans les applications. Pour faciliter l'intégration de ces services durant les différentes étapes du cycle de vie des composants, le modèle des EJB introduit les notions de *conteneur* et de *descripteur de déploiement*. Le conteneur joue le rôle de l'enveloppe décrite en 2 ou de l'encapsulateur présenté en 3.2. Il introduit une coupe, alors que le descripteur de déploiement participe au tissage du composant et des services en définissant un langage d'actions.

Intergiciels réflexifs La mise en œuvre des services dans ces modèles de composants industriels souffre de nombreuses déficiences. Premièrement, elle se fait d'une manière complètement ad hoc et ne permet pas l'intégration de services non anticipés. Ensuite, il n'existe pas de support linguistique pour représenter ces services (même sous forme de canevas). Finalement, la composition d'aspects prédéfinis sous forme de services n'est permise que d'une manière très restreinte. De ce fait, et dans l'attente d'une programmation par aspects plus aboutie, la communauté de recherche sur les intergiciels utilise de manière assez systématique les techniques de la réflexion pour traiter les problèmes de la séparation et de la composition des services. À cet égard les travaux les plus significatifs sont ceux sur CodA [McA95], GARF [GGM96], OpenCorba [Led99], ADAPT [BCRP98] et JavaPod [Bru01].

Systèmes d'exploitation Le développement de systèmes d'exploitation est traditionnellement considéré comme une activité en marge du génie logiciel. En fait, le manque

19. `http://jac.aopsys.com`.
20. `http://www.ccs.neu.edu/research/demeter/`.

de méthodologies se traduit fréquemment par le développement de systèmes fermés et non évolutifs. Trop souvent, les concepteurs de systèmes sacrifient la généricité à la performance. Y. Coady, du groupe de G. Kiczales, a récemment montré qu'il est possible d'appliquer la programmation par aspects à l'écriture efficace d'algorithmes de caches pour la gestion de la mémoire virtuelle. Son approche consiste à associer un cœur générique et des aspects réalisant sa personnalisation [CKF+01, CKFS01]. Cette meilleure séparation entre algorithmes (politiques) et mécanismes de base facilite leurs évolutions et leurs extensions.

À l'heure actuelle, il se trouve que la majorité des systèmes d'exploitation du commerce sont écrits en C. Ce constat a conduit Y. Coady à proposer une extension de C appelée AspectC qui applique les concepts d'AspectJ à un langage de bas-niveau. Cette problématique du tissage d'aspects dans des systèmes complexes existants, souvent écrits en C, est très importante pour de nombreux domaines d'applications. Elle nous conforte dans l'idée de conduire une réflexion sur la définition d'un modèle d'aspects aussi uniforme et indépendant du langage de base que se peut.

3.5 Critiques des approches actuelles

3.5.1 Instrumentation du programme de base

Cette transformation pose le double problème de la violation de l'encapsulation et de la modification des applications. Comment garantir alors la protection et la préservation de ses propriétés[21] ? Par ailleurs, dans la mise en œuvre d'un langage de coupes, comment veiller à la non-prolifération des aspects dynamiques dans la lignée de la réflexion et du tout métaobjet ?

3.5.2 Passage à l'échelle : interférence et composition

Autant la programmation par aspects réduite à l'utilisation d'un petit ensemble d'aspects orthogonaux se conçoit bien, autant la prise en compte a priori d'un grand nombre de dimensions mal connues reste un problème ouvert. Dès lors que les aspects représentent un élément de réutilisabilité, il est indispensable de disposer au niveau de leurs langages d'un mécanisme de composition. Celui-ci présuppose que les aspects sont complètement orthogonaux ou qu'il est possible de décrire les interférences entre ces aspects.

Dans les faits, il est fréquent que deux aspects concernent une même entité du programme de base[22]. À supposer que soit vérifiée la compatibilité entre ces deux aspects, se pose alors, de manière assez analogue à la résolution des conflits en héritage multiple, le problème d'expliciter leur ordre de composition.

Un modèle formel pour la définition d'aspects et de tisseurs est proposé dans [DFS02]. Dans ce cadre, un tisseur est vu comme un moniteur d'exécution (qui entrelace l'exécution du programme de base avec celles des aspects) et un aspect définit un motif de trace qui indique quand il doit être déclenché. Ce modèle a permis de définir une analyse statique de conflits entre aspects et de proposer plusieurs techniques de résolution de conflits utilisant des opérateurs de composition d'aspects.

3.5.3 Limite dans l'expressivité

Du fait d'une séparation insuffisante entre le langage de coupes et le langage d'actions, il est difficile de traiter déclarativement les aspects. En particulier, il est souvent impossible

21. À rapprocher des anciens travaux sur la macro-expansion hygiénique en Lisp [KFFD86].

22. Par exemple, si un aspect de sécurité permet d'encrypter (puis décrypter) les arguments des méthodes et si un autre aspect de mémoïsation permet de conserver la liste des appels de ces méthodes, se pose la question d'enregistrer ces appels avant ou après le cryptage.

de raisonner sur un ensemble de coupes que l'on voudrait lier entre elles. Ainsi AspectJ, ne fournit (presque exclusivement) que des coupes correspondant à des points individuels de l'exécution du programme de base ; la définition d'aspects concernant un ensemble de coupes nécessite de gérer manuellement (au niveau des actions) un état représentant la relation entre ces points.

Nous étudions une alternative à cette gestion manuelle dans [DMS01]. Les séquences de points qui définissent une coupe sont spécifiées à l'aide d'expressions proches de grammaires.

3.5.4 Absence de bases formelles

Très peu de travaux portent sur les fondements de la programmation par aspects, à l'exception notable de ceux de R. Lämmel [Läm99], J. Andrews [And01] et M. Wand [WKC02].

R. Lämmel présente une formalisation en terme de transformations de grammaires engendrant des arbres de syntaxe abstraite. Ces grammaires, bien que potentiellement très expressives, sont d'un niveau d'abstraction très bas et, par conséquent, inapproprié pour donner un modèle formel complet. Plus récemment, il propose une sémantique opérationnelle « grand pas » fondée sur les encapsulateurs [Läm02] et donc plus proche des modèles opérationnels. J. Andrews développe un calcul de processus pour formaliser le tissage mais dans le cadre particulier de l'étude de la concurrence. Le point commun à ces deux approches est qu'elles ne s'intéressent qu'à la formalisation de langages dédiés et n'envisagent pas l'expression d'un langage générique intégrant l'expression des coupes et des actions.

De ce point de vue, l'approche du groupe de M. Wand semble la plus intéressante. Elle porte sur la définition (en Scheme) d'un cadre formel pour la définition de langages d'aspects. Néanmoins, l'utilisation de ce formalisme pour prouver, par exemple, la préservation des propriétés du programme de base au progamme tissé, semble encore loin d'être évidente.

4 Vers un continuum objet, aspect et composant

Dans la lignée des objets, l'objectif du projet OBASCO est de concevoir et de développer une nouvelle génération de langages rendant compte d'un modèle de programmation bien fondé et complétant l'approche ascendante des infrastructures industrielles à la EJB, .net et CCM.

Dans la tradition de la machine Flex d'Alan Kay et des travaux sur la réflexion, nous revendiquons une vision uniforme permettant de décrire aussi bien un "objet système" que l'assemblage de composants d'entreprise distribués.

Notre hypothèse est que l'approche par composants telle qu'elle est aujourd'hui présentée n'est pas assez aboutie pour construire de grandes architectures logicielles. Dans cette quête du Graal, nous postulons qu'il existe un continuum entre les objets, les aspects et les composants. Il nous faut donc travailler à la fois sur la compréhension de la programmation par aspects mais aussi sur celle de la programmation par composants. Il nous semble que cette approche doit permettre de faire fructifier plus de 20 ans de travaux sur les systèmes et langages à objets tout en préparant les ruptures technologiques de demain!

Remerciements

Le premier auteur tient à remercier chaleureusement Jean-François en l'honneur duquel ce colloque est organisé. Il n'oublie pas qu'il fut son élève dans le milieu des années soixante-dix. C'est sous son impulsion et celle de Patrick Greussay qu'il se mit passionnément à l'étude des langages Lisp, Plasma et Smalltalk. Ces (presque) trente années de collaboration furent très riches en discussions en particulier sur l'art de bien programmer.

Ce texte est un extrait de la proposition de projet INRIA OBASCO. Différents relecteurs ont contribué à l'amélioration de par la qualité de leur remarques et de leurs interactions. Nous souhaitons tout particulièrement remercier Isabelle Attali, Gérard Boudol, Yves Caseau, Charles Consel, Jacky Estublier, Rachid Guerraoui, Michel Mauny et Yan Vitek.

5 Références bibliographiques

[ABC+96] M. Aksit, A. Black, L. Cardelli, P. Cointe, and al. Strategic research directions in object oriented programming. *ACM Computing Surveys*, 28(4):691–700, December 1996.

[ACM01] ACM. *Proceedings of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, Vienna, Austria, September 2001.

[ACN02] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, Orlando, FL, USA, May 2002. IEEE Computer Society Press.

[Aks96] M. Aksit. Composition and separation of concerns in the object-oriented model. *ACM Computing Surveys*, 28A(4), 1996.

[And01] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[AWB+94] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, volume 791, pages 152–184. Springer-Verlag, 1994.

[BBB+00] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Volume ii: Technical concepts of component-based software engineering, 2nd edition. Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon Software Engineering Institute, May 2000.

[BC90] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of ECOOP-OOPSLA*, Ottawa, Canada, October 1990.

[BCRP98] G.S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, London, 1998. Springer-Verlag.

[BDMS02] L. P. Barreto, R. Douence, G. Muller, and M. Südholt. Programming OS schedulers with domain-specific languages and aspects: New approaches for OS kernel engineering. Int. Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD, April 2002.

[BG96] J.-P. Briot and R. Guerraoui. Objets pour la programmation parallèle et répartie : intérêts, évolutions et tendances. *Technique et Science Informatique*, 15(6), 1996.

[BJPW99] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Software*, pages 38–45, June 1999.

[BL01] M.N. Bouraqadi and T. Ledoux. Le point sur la programmation par aspects. *Technique et Science Informatique*, 20(4), 2001.

[BM02] L.P. Barreto and G. Muller. Bossa: a language-based approach to the design of real-time schedulers. In *10th International Conference on Real-Time Systems (RTS'2002)*, pages 19–31, Paris, France, March 2002.

[Bou89] P. Boulez. *Le pays fertile*. NRF, Gallimard edition, 1989.

[Bru01] E. Bruneton. *Un support d'exécution pour l'adaptation des aspects non-fonctionnels des applications réparties*. Thèse de doctorat, INPG, October 2001.

[BS99] M.N. Bouraqadi-Saâdani. *Un MOP Smalltalk pour l'étude de la composition et de la compatibilité des métaclasses. Application à la programmation par aspects*. PhD thesis, École des Mines et Université de Nantes, July 1999.

[BSL03] N.M. Bouraqadi-Saâdani and T. Ledoux. Supporting AOP using reflection. Addison-Wesley, 2003. To appear.

[BSLS01] M.N. Bouraqadi-Saâdani, T. Ledoux, and M. Südholt. A reflective infrastructure for coarse-grained strong mobility and its tool-based implementation. Technical Report 01-7-INFO, École des Mines de Nantes, September 2001. Invited presentation at the *International Workshop on "Experiences with reflective systems"*, (held in conjunction with Reflection 2001 [YM01]).

[CBS02] T. Coupaye, E. Bruneton, and J.-B. Stéfani. The fractal composition framework. Technical report, The ObjectWeb Group, 2002.

[Cen00] Technologies clés 2005. Ministère de l'économie, des finances et de l'industrie, September 2000. Les éditions de l'industrie, Paris.

[CH01] B. Councill and G.T. Heineman. Definition of a software component and its elements. In G.T. Heineman and W.T. Councill, editors, *Component-Based Software Engineering – Putting the Pieces Together*, pages 5–19. Addison-Wesley, 2001.

[CKF+01] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. Suan Ong. Structuring system aspects. *Communications of the ACM*, 44(10), October 2001. Special issue on AOP.

[CKFS01] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)* [ACM01], pages 88–98.

[CL01] R. Cardone and C. Lin. Comparing frameworks and layered refinement. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 285–294, Toronto, Canada, May 2001. IEEE Computer Society Press.

[Coi99] P. Cointe, editor. *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *Lecture Notes in Computer Science*, Saint-Malo, France, July 1999. Springer-Verlag.

[Coi00] P. Cointe. Les langages à objets. *Technique et Science Informatique*, 19(1-2-3), 2000.

[DC01] J. Dowling and V. Cahill. The K-component architecture meta-model for self-adaptive software. In Yonezawa and Matsuoka [YM01], pages 81–88.

[DFS02] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, pages 173–188, Pittsburgh, USA, October 2002.

[DMS01] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In A. Yonezawa and S. Matsuoka, editors, *Proceedings of the 3rd International Conference on Reflection 2001*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, Kyoto, Japan, September 2001. Springer-Verlag.

[DYK01] L.G. DeMichiel, L.Ü. Yalçinalp, and S. Krishnan. *Enterprise JavaBeans^{TM} Specification*. Sun Microsystems, August 2001. Version 2.0, Final Release.

[EFA01] T. Elrad, R. Filman, and Bader A. Aspect-oriented programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[Est00] J. Estublier. Software configuration management: a roadmap. In A. Finkelstein, editor, *The Future of Software Engineering, Proceedings of FoSE'00*, pages 279–289, Limerick, Ireland, June 2000. ACM Press.

[FF98] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 236–248. ACM SIGPLAN Notices, 33(5), May 1998.

[FP98] J.-C. Fabre and T. Perennou. A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers*, 47(1):78–95, 1998.

[GGM96] R. Guerraoui, B. Garbinato, and K. Mazouni. Lessons from designing and implementing GARF. In *Object-Oriented Parallel and Distributed Computing*. Springer-Verlag, 1996.

[Gro02] Object Management Group. CORBA components. Adopted Specification formal/02-06-65, OMG, June 2002. Version 3.0.

[HJMAF99] W-M. Ho, Jézéquel J-M., Le Guennec A., and Pennaneac'h F. UMLAUT: an extensible UML transformation framework. In J. Hall Robert and E. Tyugu, editors, *14th IEEE conference on Automated Software Engineering (ASE'99)*, Florida, USA, October 1999.

[HJPP02] W-M. Ho, J.-M. Jézéquel, F. Pennaneac'h, and N. Plouzeau. A toolkit for weawing aspect oriented designs. In *1st conference on Aspect Oriented Software Development (AOSD'02)*, Enschede, The Netherlands, April 2002.

[HL95] W. Hürsch and C. Videira Lopes. Separation of concerns. Technical Report NU-CCS-95-03, Northeastern University, 1995.

[HL02] T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In D. Le Métayer, editor, *Programming Languages and Systems - 11th European Symposium on Programming, ESOP 2002*, Lecture Notes in Computer Science, pages 6–20, Grenoble, France, April 2002. Springer-Verlag.

[HML02] D. He, G. Muller, and J.L. Lawall. Distributing MPEG movies over the Internet using programmable networks. In *The 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 161–170, Vienna, Austria, July 2002.

[KFFD86] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duda. Hygienic macro expansion. In *ACM Conference on Lisp and Functional Programming*, Cambridge, August 1986.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming - 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming - 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finnland, June 1997. Springer-Verlag.

[Kru95] P.B. Kruchten. The $4 + 1$ view model of architecture. *IEEE Software*, pages 42–50, 1995.

[Läm99] R. Lämmel. Declarative aspect-oriented programming. In *PEPM'99*, 1999.

[Läm02] R. Lämmel. A semantical approach to method-call interception. In Gregor Kiczales, editor, *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD)*, pages 41–55. ACM Press, April 2002.

[Led99] T. Ledoux. OpenCorba: a Reflective Open Broker. In Cointe [Coi99], pages 197–214.

[Ler95] X. Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 142–153, San Francisco, CA, USA, January 1995. ACM Press.

[Ler00] X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[Lie94] K. Lieberherr. *The Art of Growing Adaptative Object-Oriented Software.* PWS Publishing Company, 1994. Traité multi-volumes sur les sciences de l'ingénieur, série 4C. À paraître.

[LLM99] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999.

[McA95] J. McAffer. Meta-level programming with CodA. In W. Olthoff, editor, *ECOOP'95 - Object-Oriented Programming - 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 190–214, Aarhus, Denmark, August 1995. Springer-Verlag.

[McI68] M.D. McIlroy. Mass produced software components. In P. Naur and Randell B., editors, *Proceedings of the NATO Conference on Software Engineering*, pages 138–155, Garmish, Germany, October 1968. NATO Science Committee.

[MFH01] S. McDirmid, M. Flatt, and W.C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings of OOPSLA*, 2001.

[ML98] M. Mezini and K.J. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Conference on Object-Oriented*, pages 97–116, 1998.

[MLTR03] G. Muller, J. Lawall, S. Thibault, and E.V.J. Rasmus. A domain-specific language approach to programmable networks. *IEEE - Transactions on Systems, Man and Cybernetics*, 2003. To appear.

[MMC95] P. Mulet, J. Malenfant, and P. Cointe. Towards a methodology for explicit composition of metaobjects. In *Proceedings of OOPSLA'95*, pages 316–330, Austin, Texas, October 1995. ACM-Sigplan.

[MRC+00] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 17–30, San Diego, California, October 2000.

[MSL00] M. Mezini, L. Seiter, and K. Lieberherr. Component integration with pluggable composite adapters. In M. Aksit, editor, *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer Academic Publishers, 2000.

[MT00] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[Nie95] O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, chapter 4, pages 99–121. Prentice Hall, 1995.

[OHE96] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects - Survival Guide*. Wiley, 1996.

[ORM01] Architecture Board ORMSC. Model Driven Architecture (MDA). Technical Report ormsc/2001-07-01, OMG, July 2001.

[PSLF01] R. Pawlak, L. Seinturier, Duchien L., and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In Yonezawa and Matsuoka [YM01].

[SC00] J.C. Seco and L. Caires. A basic model of typed components. In E. Bertino, editor, *ECOOP 2000 - Object-Oriented Programming - 14th European Conference*, volume 1850 of *Lecture Notes in Computer Science*, pages 108–128, Sophia Antipolis and Cannes, France, June 2000. Springer-Verlag.

[Ses00] R. Sessions. *COM+ and the battle for the Middle Tier*. Wiley, 2000.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[Sun97] Sun Microsystems. *JavaBeans^{TM}*, July 1997. Version 1.01.

[Sun01] Sun Microsystems. Sun Open Net Environment (Sun ONE) Software Architecture. White Paper, 2001.

[Szy02] C. Szyperski. *Component Software*. Addison-Wesley, 2002. 2nb edition.

[Tho02] D. Thomas. Reflective Software Engineering - From MOPS to AOSD. *Journal Of Object Technology*, 1(4):17–26, October 2002.

[TJ88] D. Thomas and K. Johnson. Orwell - a configuration management system for team programming. In *Proceedings of OOPSLA'88*, pages 135–142. ACM-Sigplan, September 1988.

[TNCC03] E. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection : Spatial and temporal selection of reification. In *Proceedings of OOPSLA'2003*, Anaheim, USA, October 2003. ACM-Sigplan.

[vdBL89] J. van den Bos and C. Laffra. PROCOL: A parallel object language with protocols. In N. Meyrowitz, editor, *OOPSLA'89, Conference Proceedings*, pages 95–102, New Orleans, Louisiana, USA, October 1989. ACM SIGPLAN Notices, 24(10).

[vdHMRRM01] A. van der Hoek, M. Mikic-Rakic, R. Roshandel, and N. Medvidovic. Taming architectural evolution. In *Proceedings of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)* [ACM01], pages 1–10.

[VM01] M. Vadet and P. Merle. Les containers ouverts dans les plates-formes à composants. In *Journées Composants - JC 2001*, Besançon, France, October 2001.

[WKC02] M. Wand, G. Kiczales, and C.Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming, 2002. Soumis pour publication.

[WLF01] M. Wermelinger, A. Lopes, and J.L. Fiadeiro. A graph based architectural (re)configuration language. In *Proceedings of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)* [ACM01], pages 21–32.

[YM01] A. Yonezawa and S. Matsuoka, editors. *Proceedings of the 3rd International Conference on Reflection 2001*, volume 2192 of *Lecture Notes in Computer Science*, Kyoto, Japan, September 2001. Springer-Verlag.

[YS97] D.Y. Yellin and R.E. Strom. Protocol specifications and component adaptors. *ACM Transactions of Programming Languages and Systems*, 19(2):292–333, March 1997.