Traduit avec l'accord de l'IEEE, à partir de
la version originale en Anglais paru dans
© l'IEEE Software, Volume 12 (6) pp 42-50, Novembre 1995

Plans Architecturaux – Le Modèle d'Architecture Logicielle à « 4+1 » Vues

Philippe Kruchten
Rational Software Canada
638-650 West 41st Avenue
Vancouver, B.C., V5Z 2M9 Canada
pkruchten@rational.com

Traduction :
La traduction a été réalisée par Celso Gonzalez, Consultant du département Process Management de VALtech (Celso.Gonzalez@valtech.fr)

Avertissement

Cette traduction n'est pas une traduction littérale de l'article de Philippe Kruchten, mais plutôt une traduction libre. La trame et les différentes informations sont issues de son article, mais les représentations ont été revues afin de les faire correspondre à celles utilisées dans la version actuelle du RUP (Rational Unified Process), c'est-à-dire celles d'UML (Unified Modeling Language). Toute erreur s'étant glissée dans le document ne pourrait aucunement être reprochée à Philippe Kruchten, mais serait plutôt issue d'une erreur de traduction ou de compréhension de ma part.

Résumé

Cet article présente un modèle basé sur l'utilisation de plusieurs vues différentes, permettant de décrire l'architecture de systèmes complexes où le logiciel a une influence essentielle sur la réalisation et l'évolution du système comme un tout (software-intensive systems). L'utilisation de ces vues multiples permet de traiter séparément les intérêts des différentes « parties prenantes » de l'architecture : utilisateurs finaux, développeurs, ingénieurs systèmes, chefs de projet, etc., et de traiter séparément les exigences fonctionnelles et non-fonctionnelles. Chacune des cinq vues est décrite, accompagnée d'une notation permettant de la représenter. Les vues ont été conçues en utilisant un processus de développement itératif centré sur l'architecture et piloté par les cas d'utilisation.

Mots clés : architecture logicielle, vue, conception orientée objet, processus de développement logiciel.

Introduction

On a tous lus de nombreux livres et articles dans lesquels un schéma essayait de représenter l'essentiel de l'architecture d'un système. Mais lorsque l'on regarde attentivement les boîtes et les flèches du schéma il devient clair que les auteurs ont dû se démener pour représenter dans un seul schéma plus qu'il ne peut exprimer. Les boîtes représentent-elles des exécutables ? Ou des morceaux de code source ? Ou des machines différentes ? Ou peut-être

des regroupements logiques de fonctionnalités ? Les flèches représentent-elles des dépendances de compilation ? Ou des contrôles de flux ? Ou des flux de données ? Il s'agit, de manière générale, d'un peu de tout cela. Une architecture ne nécessite-t-elle qu'un seul style architectural ? Parfois l'architecture logicielle garde des séquelles d'une conception système qui est allée trop loin en partitionnant prématurément le logiciel, ou d'une accentuation excessive d'un aspect du développement logiciel : le « data engineering », la performance d'exécution, ou la stratégie de développement et l'organisation des équipes. Souvent aussi l'architecture ne répond pas aux préoccupations de tous ces clients (ou stakeholders (parties prenantes) comme ils sont appelés au USC (University of Southern California)). Ce problème a été relevé par de nombreux auteurs : Garlan et Shaw[1], Abowd et Allen au CMU (Carnegie Mellon University) et Clements au SEI (Software Engineering Institute). Comme remède, nous proposons d'organiser la description de l'architecture logicielle en utilisant plusieurs *vues* différentes, chacune répondant à des intérêts spécifiques.

Un Modèle Architectural

L'architecture logicielle traite de la conception et de l'implémentation de la structure haut niveau du logiciel. C'est le résultat de la combinaison d'un certain nombre d'éléments d'architecture de manière à répondre aux principales exigences fonctionnelles et de performance du système, ainsi que d'autres exigences non-fonctionnelles telles que la fiabilité, la capacité à évoluer, la portabilité, et la disponibilité. Perry et Wolfe l'ont élégamment intégré dans cette formule[2] modifiée par Boehm :

Architecture logicielle = {Eléments, Configurations, Justification/Contraintes}
(Software achitecture = {Elements, Forms, Rationale/Constraints})

L'architecture logicielle s'occupe d'abstraction, de décomposition et de composition, de style et d'esthétique. Pour décrire une architecture logicielle, on utilisera un modèle composé de plusieurs *vues* ou perspectives. Afin éventuellement de traiter les architectures importantes et d'un abord difficile, le modèle proposé sera composé de cinq vues principales (cf. fig 1) :

- *La vue logique,* qui est le modèle objet de la conception (quand une méthode de conception orientée objet est utilisée),
- *La vue processus,* qui enregistre les aspects de concurrence et de synchronisation de la conception,
- *La vue de déploiement,* qui décrit le déploiement du logiciel sur le matériel et reflète son aspect distribué,
- *La vue d'implémentation,* qui décrit l'organisation statique du logiciel dans son environnement de développement.

La description d'une architecture –les décisions prises– peut être organisée autour de ces quatre vues, et illustrée par quelques cas d'utilisations qui deviendront la cinquième vue. L'architecture est en fait partiellement issue de ces cas d'utilisation comme on le verra plus tard.

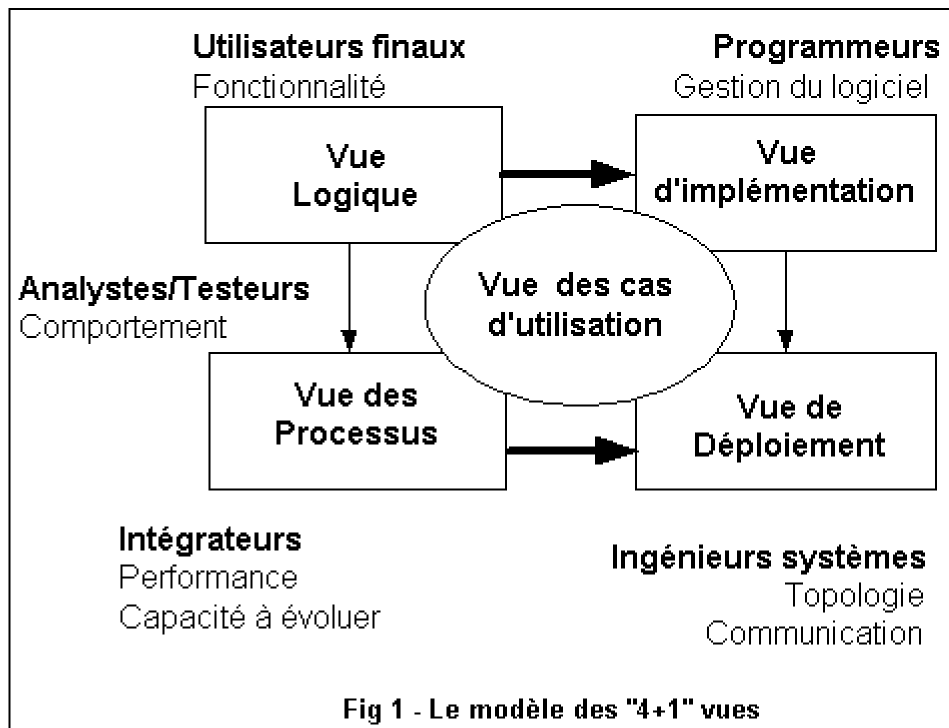

Fig 1 - Le modèle des "4+1" vues

On applique l'équation de Perry et Wolf sur chaque vue, c'est-à-dire, pour chaque vue, on définit le jeu d'éléments à utiliser (composants, conteneurs et connecteurs), on enregistre les formes et les modèles qui fonctionnent, puis on enregistre la raison d'être et les contraintes, reliant l'architecture à certaines des exigences. Chaque vue est décrite par un *plan* utilisant ses propres notations. Les architectes peuvent aussi, pour chaque vue, choisir un certain *style architectural*, permettant la cohabitation, dans un système, de plusieurs styles.

On va maintenant examiner une à une les cinq vues, décrivant pour chacune son objectif, les points de vues qu'elle traite, une notation pour le plan architectural correspondant et les outils que l'on utilise pour la décrire et la gérer. Les exemples donnés correspondent à la conception d'un PABX, provenant de notre travail chez Alcatel Business System et d'un système de Contrôle de Trafic Aérien[3], mais d'une manière simplifiée – le but ici est juste de vous donner un avant-goût des vues et de leur notation et non de définir l'architecture de ces systèmes.

Le modèle des « 4+1 » vues est plutôt « générique » : d'autres notations et d'autres outils peuvent être utilisés, d'autres méthodes de conception peuvent être utilisées, en particulier pour les décompositions logiques et processus, mais on n'a indiqué ici que celles que l'on a utilisées avec succès.

L'Architecture Logique

La Décomposition Orientée Objet

L'architecture logique s'occupe principalement des exigences fonctionnelles - ce que le système doit fournir en termes de services pour ses utilisateurs. Le système est décomposé en un ensemble d'abstractions, prises (pour la plupart) dans le domaine du système, sous la forme d'*objets* et de *classes d'objets*. Ces éléments exploitent les principes de l'abstraction, de l'encapsulation et de l'héritage. Cette décomposition ne sert pas seulement à l'analyse fonctionnelle, mais elle sert aussi à identifier les mécanismes et les éléments de conception communs aux différentes parties du système. On utilisera les *diagrammes de classes* et les *modèles de classe* d'UML[4] pour représenter l'architecture logique. Un diagramme de classe montre un ensemble de classes et leurs relations logiques : association, dépendance,

composition, généralisation, etc. Des groupes de classes en relation les unes avec les autres peuvent être regroupées en paquetages (ou packages). Les modèles de classe se focalisent sur chaque classe ; elles expriment les principales opérations de la classe et identifient les caractéristiques clés de l'objet. Il est important de définir le comportement interne d'un objet, cela est fait avec les diagrammes d'états transitions. Les mécanismes et les services communs sont définis dans des *classes utilitaires.*
Comme alternative à une approche orientée objet, une application qui est vraiment pilotée par les données, pourrait utiliser une autre forme de vue logique, telle que les diagrammes E-R.

Notation pour la vue logique

La notation utilisée pour la vue logique est dérivée d'UML[4]. Elle ne prend en compte que les objets architecturalement significatifs. On utilisera Rational Rose pour réaliser la conception de l'architecture logique.

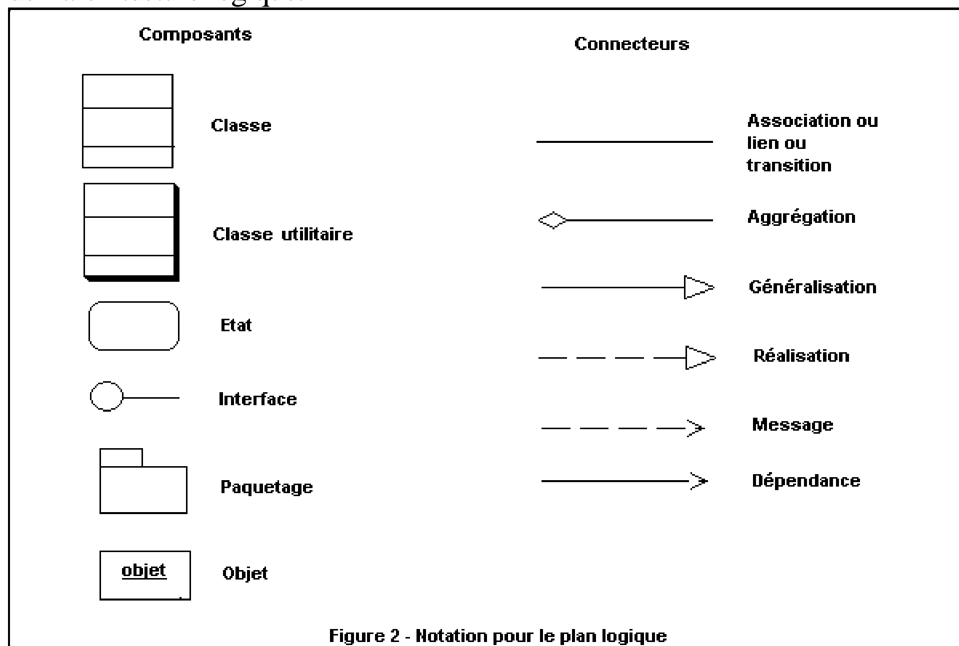

Figure 2 - Notation pour le plan logique

Le style de la vue logique

Le style utilisé pour la vue logique est un style orienté objet. La principale ligne directrice pour la réalisation de la vue logique est d'essayer de conserver un unique modèle objet cohérent pour l'ensemble du système, afin d'éviter la spécialisation prématurée des classes et des mécanismes par site ou par processeur.

Exemples de plans Logiques

La fig 3a montre les principales classes utilisées dans l'architecture du PABX Télic.

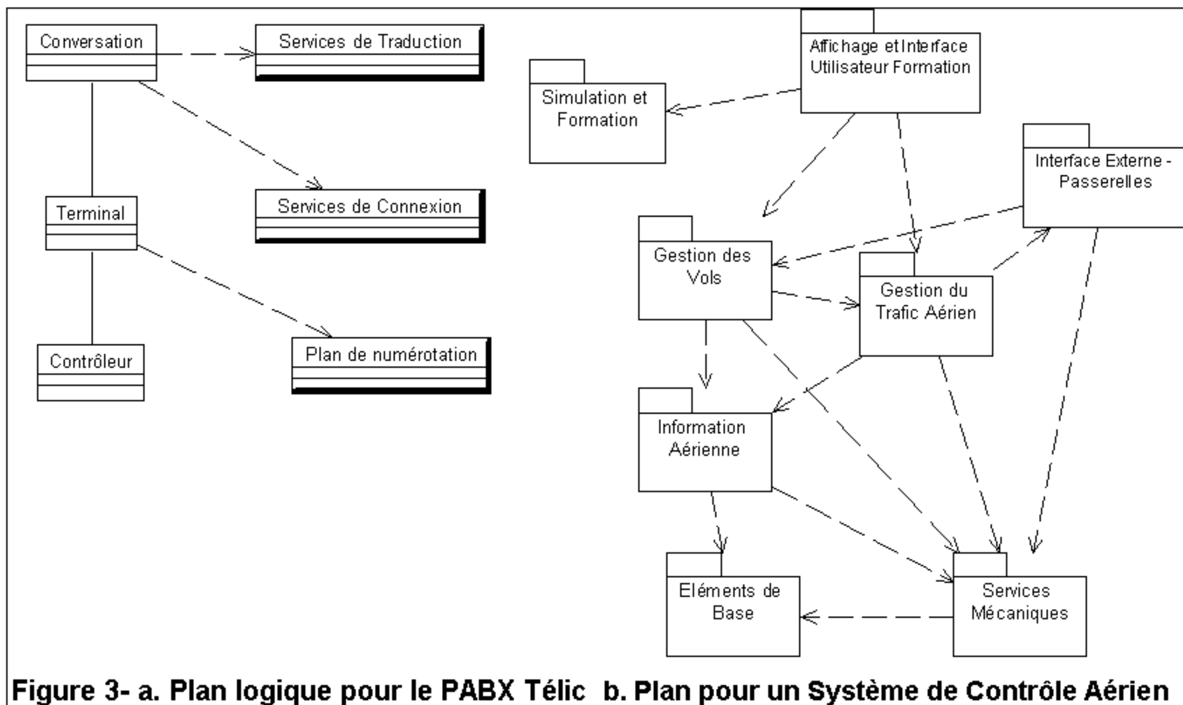

Figure 3- a. Plan logique pour le PABX Télic b. Plan pour un Système de Contrôle Aérien

Un PABX établit des communications entre des terminaux. Un terminal peut être un groupe de téléphone, une « trunk line » (une ligne vers un central), une « tie line » (un PABX privé vers une ligne de PABX), une ligne téléphonique, une ligne de donnée, une ligne RNIS, etc. Des lignes différentes sont supportées par des cartes d'interfaçage de ligne différentes. La responsabilité d'un objet *contrôleur* de ligne est de décoder et d'injecter tous les signaux dans la carte d'interfaçage de ligne, traduisant les signaux spécifiques de la carte à partir de et vers une courte et uniforme séquence d'évènements : démarrer, arrêter, composer, etc. Le contrôleur supporte aussi toutes les contraintes matérielles de temps réel. Cette classe a plusieurs sous-classes afin de donner accès à différents types d'interface. La responsabilité de l'objet terminal est de maintenir l'état d'un terminal, et de négocier les services pour cette ligne. Par exemple, il utilise les services du *plan de numérotation* pour interpréter le numéro composé dans la phase de sélection. La classe *conversation* représente l'ensemble des terminaux engagés dans une conversation. La classe *conversation* utilise les services de *traduction* (répertoire, correspondance entre adresse physique et logique, routes), et les services de *connexion* pour établir une route vocale entre les terminaux.
Pour un plus gros système, qui contient une douzaine de classes architecturalement signifiantes, la fig 3b montre le diagramme de classe de haut niveau d'un système de contrôle de trafic aérien, contenant 8 paquetages (c'est-à-dire, des groupes de classes).

L'Architecture des Processus

La Décomposition du Processus

L'architecture des processus prend en compte certaines exigences non-fonctionnelles, telles que la performance et la disponibilité. Elle traite les problèmes de parallélisme et de distribution, d'intégrité du système, de tolérance aux pannes, et de savoir comment les abstractions principales de la vue logique vont être mises en œuvre dans l'architecture du processus–dans quel thread de contrôle se trouve l'opération d'un objet en cours d'exécution.

L'architecture du processus peut être décrite à plusieurs niveaux d'abstraction, chaque niveau traitant des préoccupations différentes. Au plus haut niveau, l'architecture du processus peut être vue comme un ensemble de *réseaux* logiques, constitués de programmes (appelés « processus ») communiquant entre eux, s'exécutant indépendamment et distribués à travers

un ensemble de ressources matérielles connectées par un LAN ou un WAN. Plusieurs réseaux logiques peuvent exister simultanément, partageant les mêmes ressources physiques. Par exemple, des réseaux logiques indépendants peuvent être utilisés pour mettre en œuvre la séparation entre le système opérationnel en ligne et celui déconnecté ; ainsi que pour mettre en œuvre la coexistence entre les versions de tests et de simulation du logiciel.
Un *processus* est un regroupement de tâches qui forment un exécutable. Les processus représentent le niveau auquel l'architecture du processus peut être tactiquement contrôlée (c'est-à-dire, démarré, restauré, reconfiguré, et arrêté). De plus, les processus peuvent être répliqués pour améliorer la répartition de la charge d'exécution, ou pour augmenter la disponibilité.
Le logiciel est décomposé en un ensemble de *tâches* indépendantes. Une tâche est un thread de contrôle séparé, qui peut être programmé individuellement sur un nœud de traitement.
On peut alors distinguer les tâches majeures, qui sont des éléments architecturaux qui ne peuvent être contactés que de manière unique, et les tâches mineures, qui sont des tâches additionnelles introduites localement pour des raisons liées à l'implémentation (activités cycliques, buffering, time-outs, etc.). Elles peuvent être par exemple implémentées comme des tâches Ada, ou des threads de faible charge.
Les tâches majeures communiquent via un ensemble de mécanismes de communication inter-tâches bien défini : des services de communication synchrone et asynchrone, des appels de procédures distant (rpc), des broadcast d'événement, etc.
Les tâches mineures peuvent communiquer par rendez-vous ou par mémoire partagée. Les tâches majeures ne présument pas de leur colocation dans le même processus ou dans le même nœud de traitement.
Les flux de messages, les charges de processus peuvent être estimés en se basant sur le plan de processus. Il est aussi possible d'implémenter une architecture du processus virtuelle avec des charges de processus factices, et de mesurer les performances sur le système cible, comme l'on décrit Filarey et al. dans leur expérience Eurocontrol.

Notation pour la vue des Processus
De nouveau la notation que l'on utilise pour la vue des processus est issue de la notation UML, et elle se focalisera sur les éléments architecturalement significatifs (Fig 4).

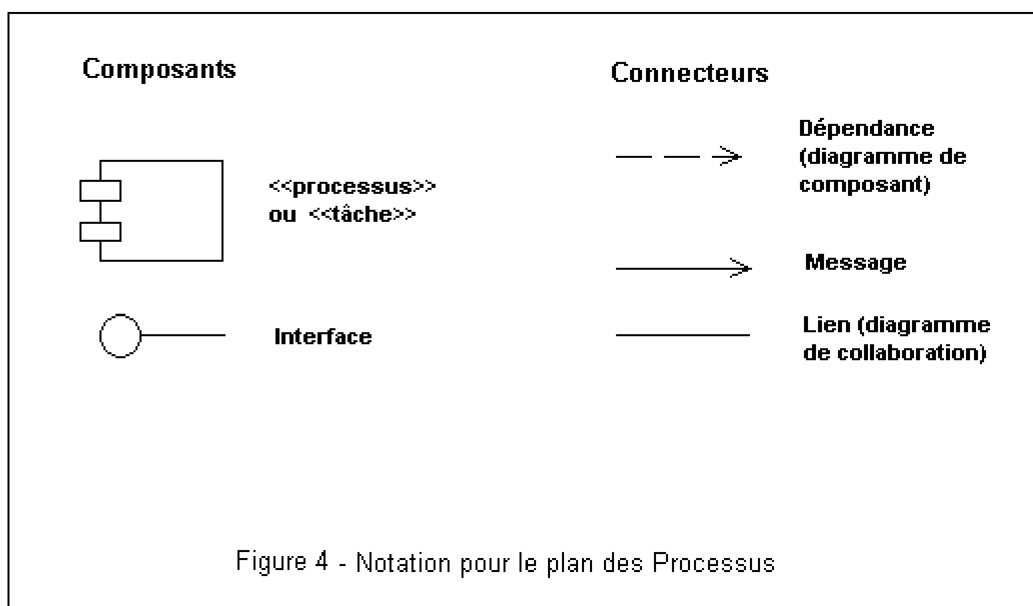

Figure 4 - Notation pour le plan des Processus

On a utilisé le produit UNAS (Universal Network Architecture Services : Services Universels d'Architecture Réseaux) de TRW pour structurer et implémenter l'ensemble des processus et

des tâches (et de leurs redondances) dans des réseaux de processus. UNAS contient un outil – le Software Architects Lifecycle Environment (SALE)– qui accepte cette notation.
SALE permet la description graphique de l'architecture du processus, en incluant les spécifications de chemins possibles de communication entre tâches, à partir de laquelle le code source C++ ou Ada peut-être automatiquement généré. Le bénéfice de cette approche pour spécifier et implémenter l'architecture du processus est que les changements peuvent être facilement incorporés, sans grand impact sur le logiciel de l'application.

Style pour la vue du processus
Plusieurs styles peuvent convenir à la vue des processus. Par exemple, à partir de la taxonomie de Garlan et Shaw[1] on peut avoir : des « tuyaux-et-filtres » (pipes-and-filters) ; ou du client/serveur, avec des variantes comme plusieurs clients/un serveur ou plusieurs clients/plusieurs serveurs. Pour des systèmes beaucoup plus complexes, on pourrait utiliser un style similaire à l'approche «groupes de processus » du système ISIS[5] tel que l'a décrit K. Birman avec une autre notation et d'autres outils.

Exemple de plan des Processus

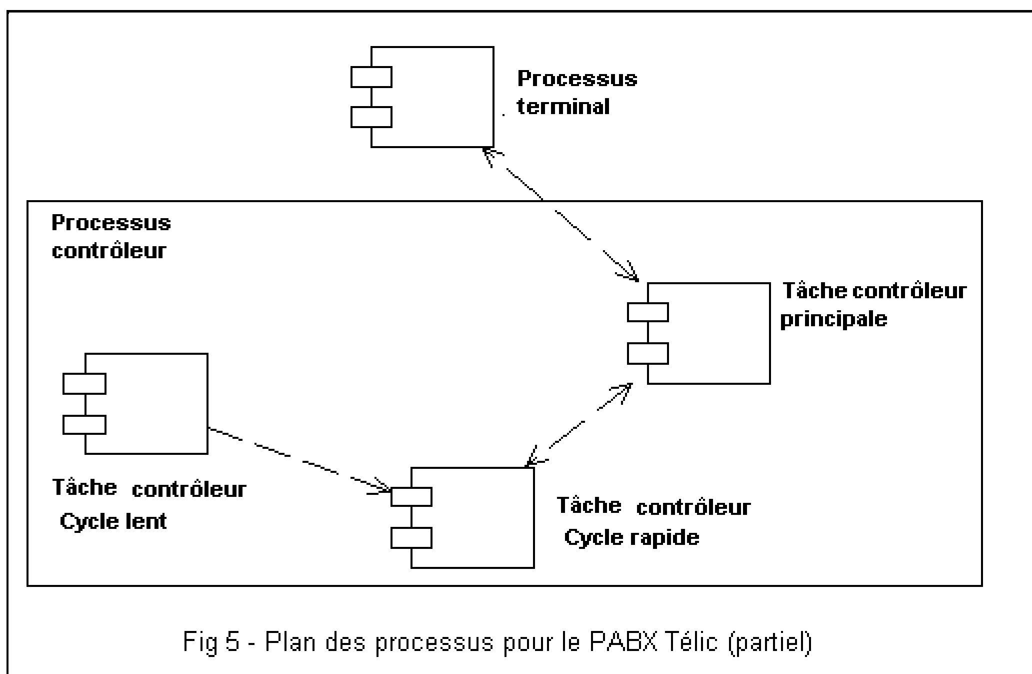

Fig 5 - Plan des processus pour le PABX Télic (partiel)

Tous les terminaux sont gérés par un seul *processus terminal*, qui est piloté par les messages de sa « queue » d'entrées. Les objets contrôleurs sont exécutés par une des trois tâches qui composent *le processus contrôleur* : une *tâche de cycle lent* sonde tous les terminaux inactifs (200 ms), place tout terminal devenant actif dans la liste d'exploration des *tâches à cycle rapide* (10ms), qui détecte tout changement d'état significatif, et les passe à la *tâche contrôleur principal* qui interprète les changements et les communique par message au terminal correspondant. Le passage du message à travers le processus de contrôle est fait via la mémoire partagée.

L'Architecture d'Implémentation

La décomposition en sous-système
L'architecture d'implémentation se focalise sur l'organisation des modules dans l'environnement de développement. Le logiciel est découpé en petits morceaux –des

bibliothèques de programme, ou des *sous-systèmes*– qui peuvent être développés par un seul développeur ou un petit nombre d'entre eux. Les sous-systèmes sont organisés en *couches*, chaque couche fournit une interface précise et limitée pour les couches supérieures.
L'architecture d'implémentation du système est représentée par des diagrammes de modules et de sous-systèmes, montrant les relations d' « import » et d' « export ». L'architecture d'implémentation complète ne peut être décrite que lorsque tous les éléments du logiciel ont été identifiés. Il est toutefois possible de lister les règles qui gouvernent l'architecture d'implémentation : le partitionnement , le regroupement, la visibilité.
L'architecture d'implémentation prend principalement en compte les exigences internes liées à la facilitation du développement, à la gestion du logiciel, à la ré-utilisation ou à la factorisation, et les contraintes imposées par les outils ou le langage de programmation. La vue d'implémentation sert de base à l'affectation des exigences, à l'affectation du travail aux développeurs (ou même pour l'organisation des équipes), à l'évaluation des coûts et des plannings, à la mesure de la progression du projet, à la réflexion sur la réutilisation logicielle, à la portabilité et la sécurité. C'est la base de l'établissement d'une ligne de produit.

Notation pour le Plan d'Implémentation
De nouveau la notation UML limitée aux objets architecturalement signifiant.

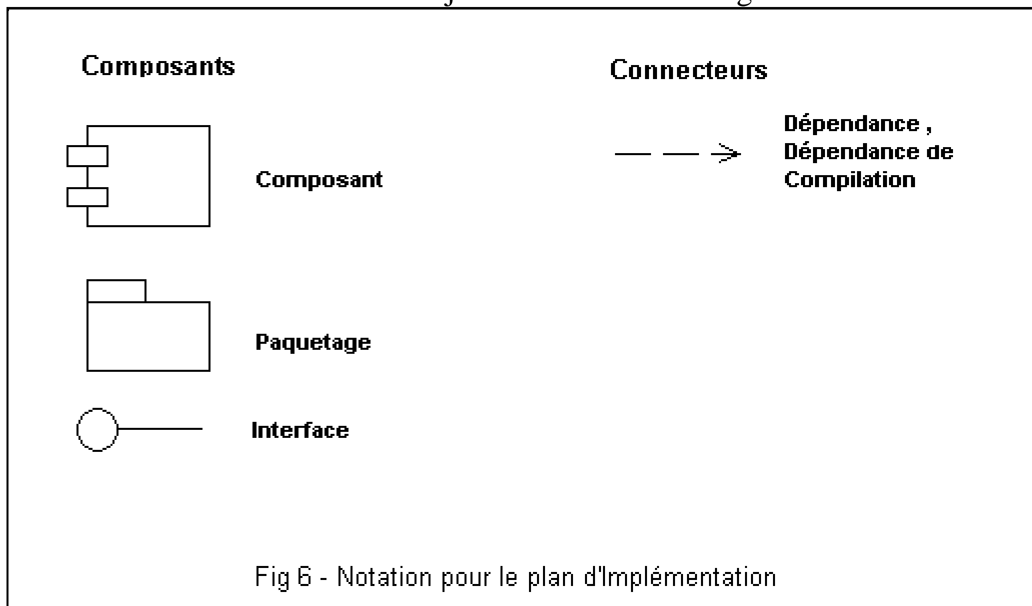

Fig 6 - Notation pour le plan d'Implémentation

L'environnement de développement Apex de Rational supporte la définition et la mise en œuvre de l'architecture d'implémentation, la stratégie en couche décrite ci-dessous, et l'application des règles de conception.
Rational Rose peut dessiner les plans d'implémentation au niveau des modules et des sous-systèmes, en conception et en rétro-conception à partir du code source pour les langages Ada, C++ et Java.

Style de la vue d'implémentation
On recommande l'utilisation d'un style en couche pour la vue d'implémentation, en définissant 4 à 6 couches ou sous-systèmes. Chaque couche a une responsabilité bien définie. La règle de conception est qu'un sous-système appartenant à une certaine couche ne peut dépendre que d'un sous-système appartenant à la même couche que lui ou à une couche inférieure. Ceci afin de réduire le développement de réseaux de dépendances très complexes entre modules et de permettre la mise en œuvre de stratégies simples de réalisation couche par couche.

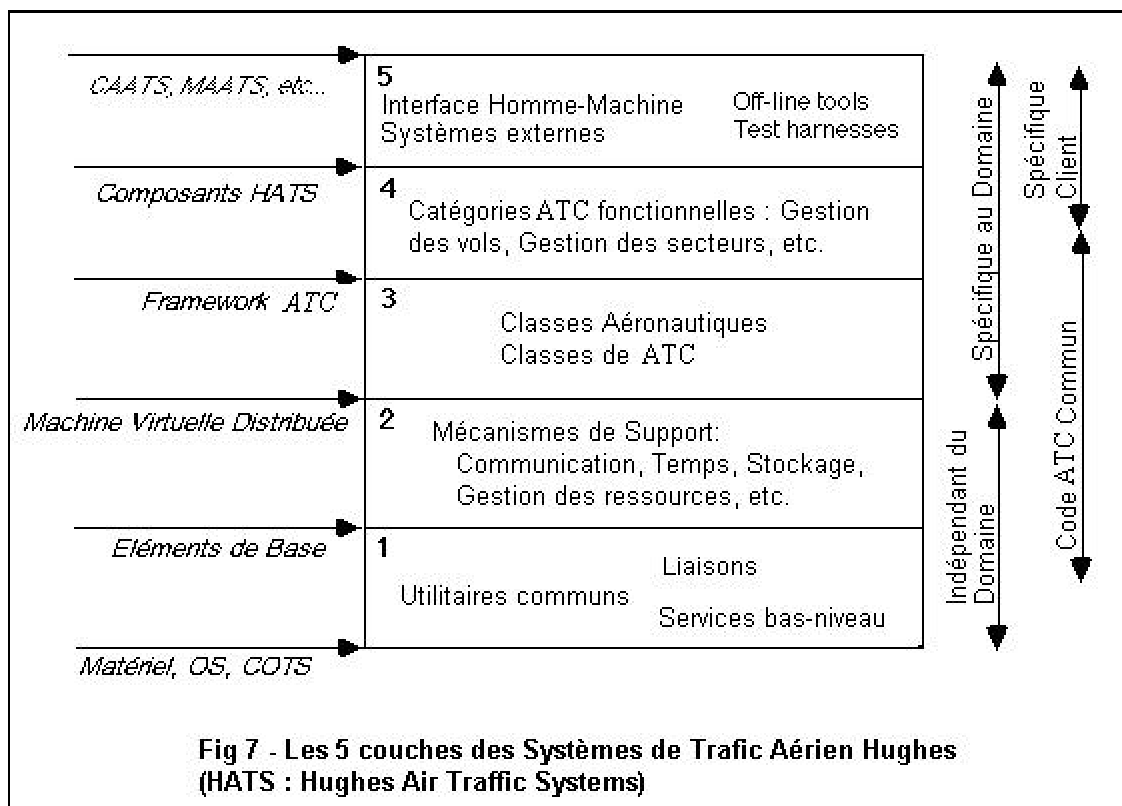

Fig 7 - Les 5 couches des Systèmes de Trafic Aérien Hughes (HATS : Hughes Air Traffic Systems)

Exemple d'architecture d'implémentation
La fig 7 représente en cinq couches l'organisation des développements d'une ligne de produits de systèmes de Contrôle de Trafic Aérien développée par Hughes Aircraft au Canada[3]. C'est l'architecture d'implémentation correspondant à l'architecture logique de la fig 3b.
Les couches 1 et 2 constituent une infrastructure distribuée, indépendante du domaine, qui est commune à la ligne des produits et qui le protège contre les variations sur les plateformes matérielles, sur les systèmes d'exploitation, ou de produits prêt à l'emploi tels que les systèmes de gestion de base de données. A cette infrastructure, la couche 3 ajoute un framework ATC (Air Traffic Control : Contrôle de Trafic Aérien) pour réaliser une architecture logicielle spécifique au domaine.
Une palette des fonctionnalités est construite dans la couche 4 à partir de ce framework. La couche 5 est très dépendante du client et du produit, et elle contient la plupart des interfaces utilisateurs et des interfaces avec les systèmes externes. Environ 72 sous-systèmes sont distribués à travers les 5 couches, chacun contenant de 10 à 50 modules, et ils peuvent être représentés dans des plans supplémentaires.

L'architecture de déploiement

Appliquer le logiciel sur le matériel
L'architecture de déploiement prend principalement en compte les exigences non fonctionnelles du système comme la disponibilité, la fiabilité (tolérance aux pannes), la performance (débit), et la modularité. Le logiciel s'exécute dans un réseau d'ordinateurs, ou nœuds de traitement (ou juste nœuds en abrégé). Les différents éléments identifiés – réseaux, processus, tâches et objets – doivent être appliqués sur les différents nœuds. On utilisera plusieurs configurations physiques différentes : certaines pour le développement et les tests,

d'autres pour le déploiement du système sur différents sites ou pour différents clients. L'application du logiciel sur les nœuds doit toutefois être très flexible et avoir un impact minimal sur le code source lui-même.

Notation pour le Plan de Déploiement

Les plans de déploiement peuvent devenir très complexe sur de gros systèmes. Ils prennent donc différentes formes, avec ou sans la correspondance avec la vue des processus.

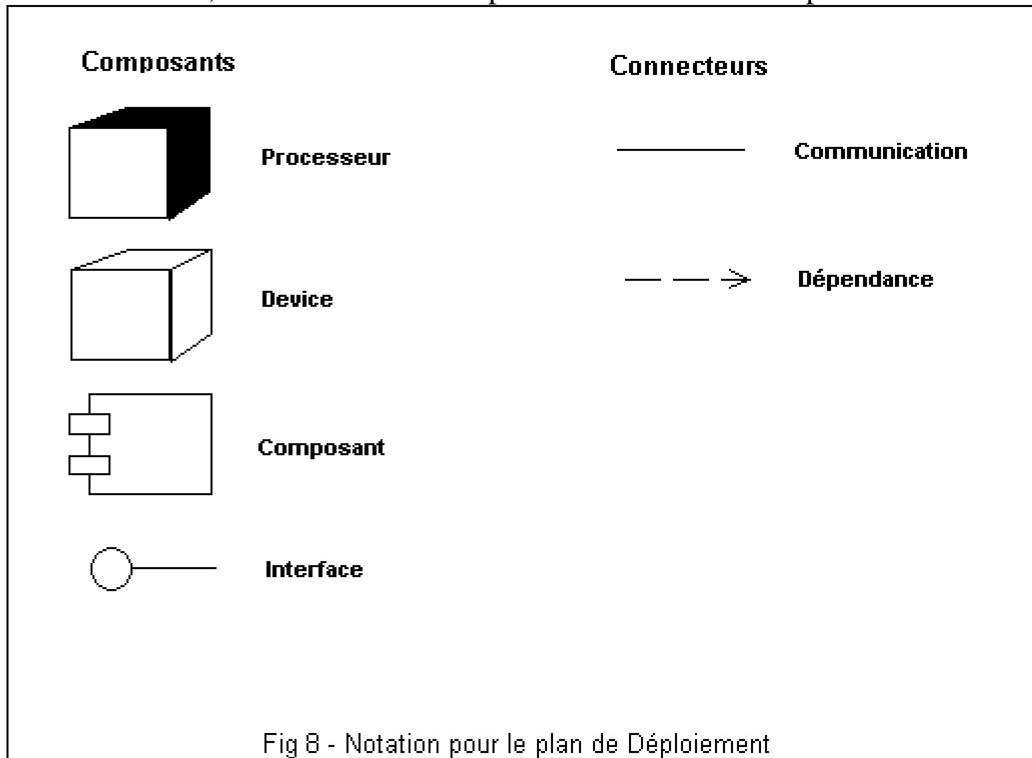

Fig 8 - Notation pour le plan de Déploiement

UNAS de TRW fournit des méthodes orientées données, permettant d'appliquer l'architecture des processus sur l'architecture physique et permettant une grande flexibilité dans les changements sans impact sur le code source.

Exemple de plan de Déploiement

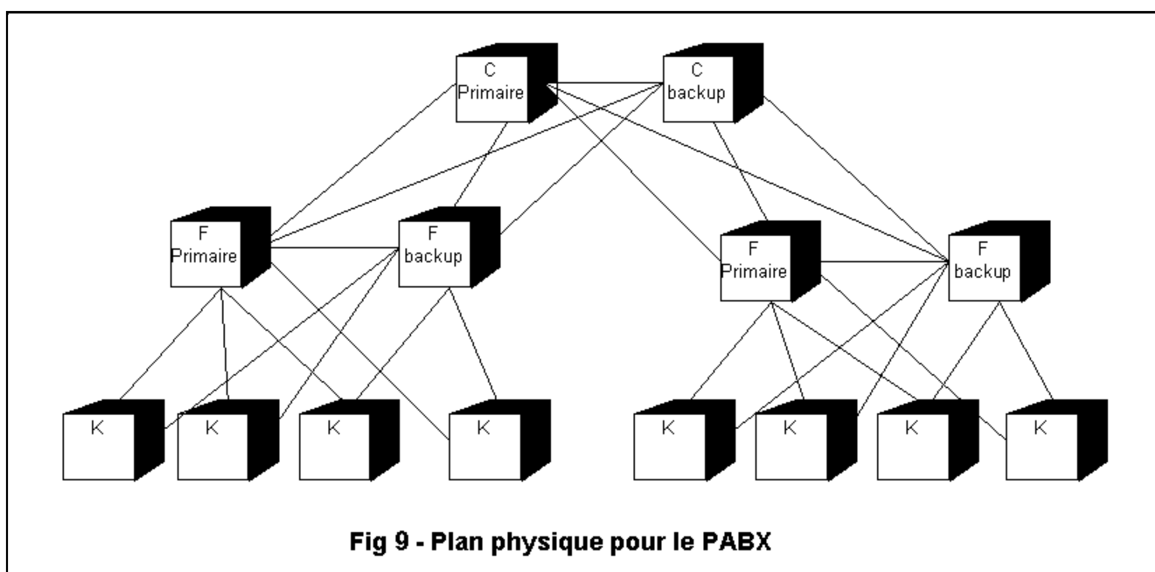

Fig 9 - Plan physique pour le PABX

La figure 9 montre une configuration possible pour un gros PABX, alors que les figures 10 et 11 montrent l'application de l'architecture des processus sur deux architectures physiques différentes, correspondant à un petit et un gros PABX. C, F et K sont trois types d'ordinateurs de capacité différente, supportant trois exécutables différents.

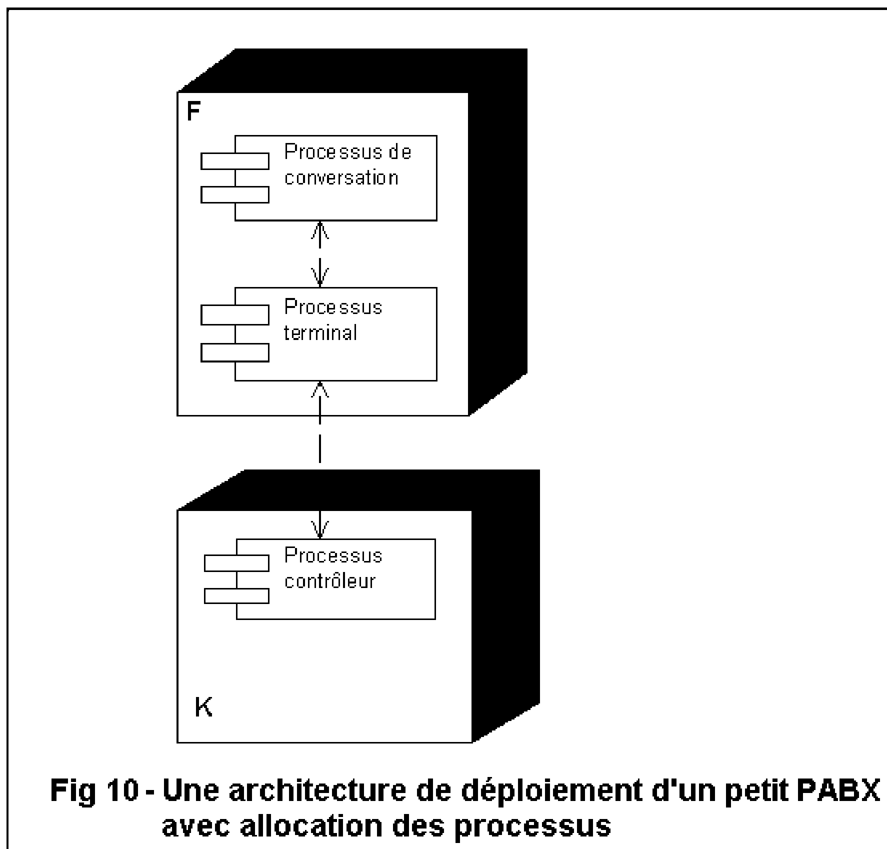

Fig 10 - Une architecture de déploiement d'un petit PABX avec allocation des processus

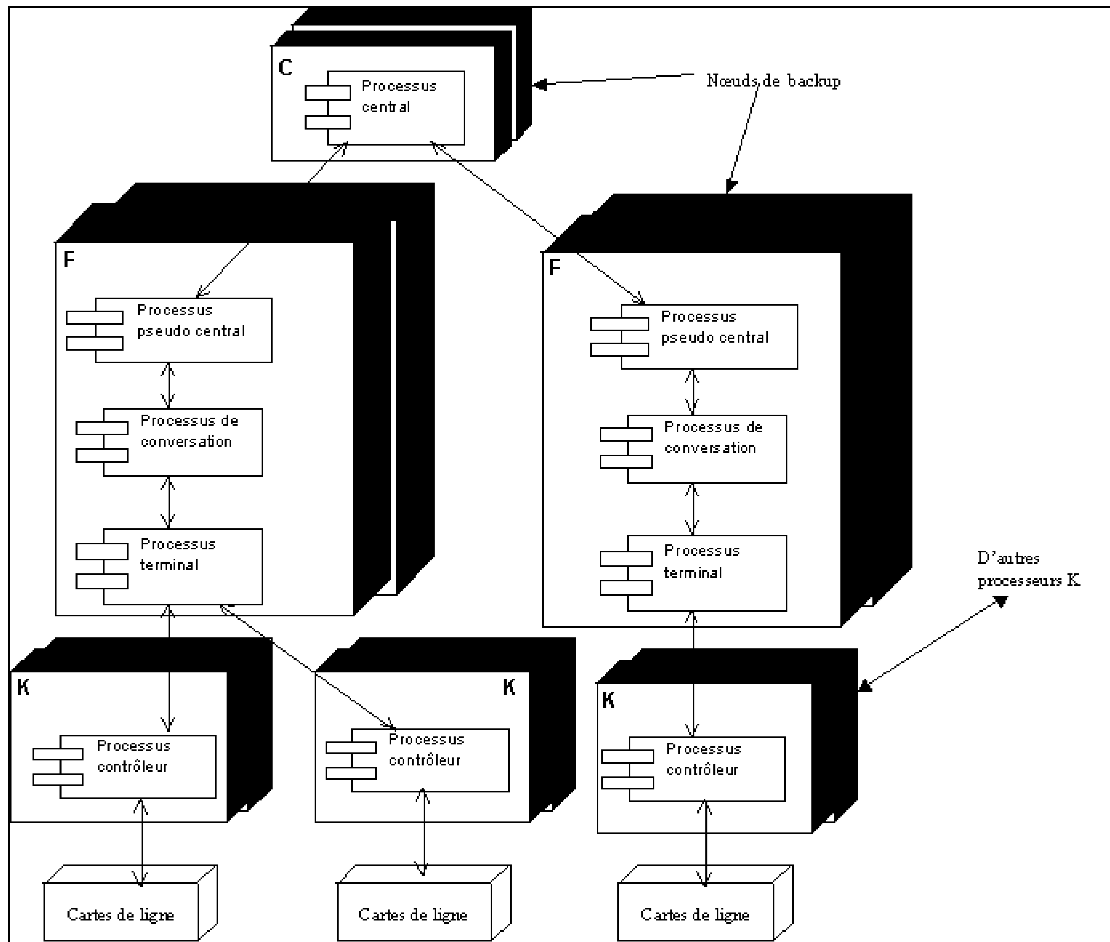

Fig. 11 – Plan physique pour un gros PABX montrant l'allocation des processus

Vue des Cas d'Utilisation

Tout réunir

On montre le travail collaboratif des éléments des quatre vues par l'utilisation d'un petit ensemble de scénarios importants – des *cas d'utilisations* – pour lesquels on va décrire les actions correspondantes (des suites d'interactions entre objets et entre processus) comme décrit par Rubin et Goldberg[6].

D'une certaine manière les cas d'utilisation sont une abstraction des exigences les plus importantes. Leur conception est exprimée par des diagrammes de cas d'utilisation.

Cette vue est redondante par rapport aux autres (d'où le "+1"), mais elle a deux objectifs principaux :

- comme guide, comme on le verra plus loin, pour découvrir les éléments de l'architecture lors de la conception architecturale,
- afin de jouer un rôle de validation et d'illustration une fois la conception de l'architecture terminée ; sur le papier, mais aussi comme point de départ pour réaliser les tests du prototype architectural.

Notation pour les Cas d'Utilisation

La notation utilisée pour les Cas d'Utilisation est toujours UML. De nouveau l'outil utilisé sera Rose de Rational.

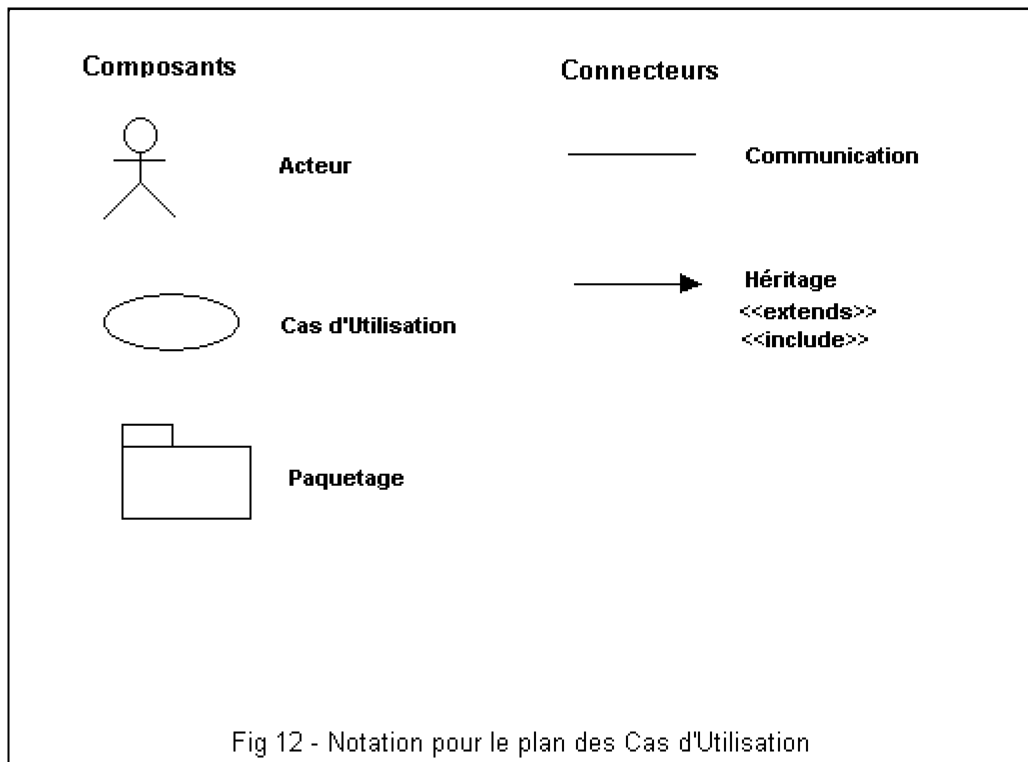

Fig 12 - Notation pour le plan des Cas d'Utilisation

Exemple d'un Cas d'Utilisation

La figure 13 montre une partie d'un Cas d'Utilisation pour un petit PABX, correspondant à l'établissement d'un appel local entre deux correspondants.

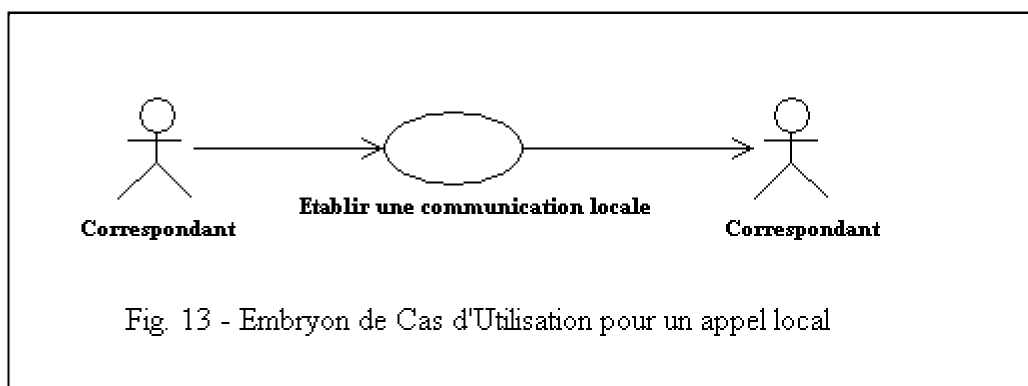

Fig. 13 - Embryon de Cas d'Utilisation pour un appel local

Correspondance Entre les Vues

Les différentes vues ne sont pas totalement orthogonales et indépendantes. Les éléments d'une vue sont connectés aux éléments des autres vues selon certaines règles et heuristiques de conception.

De la vue logique à celle des processus

On identifie un certain nombre de caractéristiques importantes des classes de l'architecture logique :

- L'autonomie : les objets sont-ils actifs, passifs, protégés ?
  - un objet *actif* prend l'initiative d'invoquer ses propres opérations ou celles d'autres objets. Il a aussi un contrôle total sur l'invocation de ses opérations par d'autres objets.
  - un objet *passif* n'invoque aucune opération spontanément et n'a aucun contrôle sur l'invocation de ses propres opérations par d'autres objets.

- Un objet *protégé* n'invoque spontanément aucune opération, mais il a un certain contrôle sur l'invocation de ses propres opérations par d'autres objets.

- La persistance : les objets sont-ils transitoires, permanents ?
- La subordination : l'existence ou la persistance d'un objet dépend-elle d'un autre objet ?
- La distribution : L'état ou les opérations d'un objet sont-elles accessibles par plusieurs nœuds de l'architecture de déploiement ? Par plusieurs processus de l'architecture des processus ?

Dans la vue logique d'architecture, on considère chaque objet comme actif, et comme potentiellement concourant, c'est à dire comme agissant « en parallèle » avec d'autres objets, et nous ne prêtons pas plus d'attention au degré de simultanéité nécessaire afin d'obtenir cet effet. Ceci, étant donné que l'architecture logique ne prend en compte que l'aspect fonctionnel des exigences.
Toutefois, lorsque l'on commence à définir l'architecture des processus, implémenter chaque objet avec son propre thread de contrôle (par exemple, son propre processus Unix ou sa propre tâche Ada) n'est pas très pratique dans l'état actuel de la technologie, compte tenu du coût énorme que cela impliquerait. De plus, si les objets sont concourants, il est nécessaire de recourir à une certaine forme d'arbitrage pour invoquer leurs opérations.
Autrement dit, plusieurs threads de contrôle sont nécessaires pour diverses raisons :

- Afin de réagir rapidement à certains types de stimuli externes, en incluant les évènements dépendant du temps
- Afin de profiter de la présence de plusieurs CPUs dans un nœud, ou de plusieurs nœuds dans un système distribué
- Afin d'augmenter l'utilisation de la CPU, en allouant la CPU à d'autres activités pendant qu'un thread de contrôle est suspendu dans l'attente de l'accomplissement d'une autre activité (par exemple, l'accès à un dispositif externe, ou l'accès à un autre objet actif).
- Afin de traiter en priorité certaines activités (et éventuellement augmenter les temps de réponse)
- Afin d'assurer la capacité à évoluer du système (avec des processus supplémentaires partageant la charge)
- Afin de séparer les intérêts des différents domaines du logiciel
- Afin d'obtenir une meilleure disponibilité du système (avec des processus de sauvegarde)

On utilise simultanément deux stratégies afin de déterminer la « bonne » quantité de concourance et de définir l'ensemble des processus nécessaires. En gardant à l'esprit l'ensemble des architectures cibles physiques potentielles, on peut procéder soit :

- **Intérieur-Extérieur :**
  En débutant par l'architecture logique : définir des tâches agents qui multiplexent un même thread de contrôle à travers plusieurs objets actifs d'une classe ; les objets dont la persistance ou la vie est subordonnée à un objet actif sont aussi exécutés dans ce même agent ; plusieurs classes qui ont besoin d'être exécutées exclusivement les unes par rapport aux autres ou qui ne nécessitent que peu de traitement, partagent un seul agent. On effectue ce regroupement jusqu'à ce que l'on atteigne un nombre de processus raisonnable permettant la distribution et l'utilisation des ressources physiques.
- **Extérieur-Intérieur**
  En débutant par l'architecture physique : identifier les stimuli (requêtes) externes au système, définir les processus du client qui permettront de traiter les stimuli et les

processus des serveurs qui ne font que fournir des services sans les démarrer ; utiliser les contraintes d'intégrité des données et de sérialisation du problème pour définir le bon nombre de serveurs, et allouer des objets aux agents du client ou des serveurs ; identifier quels objets doivent être distribués.

Le résultat obtenu est une correspondance entre les classes (et leurs objets) et un ensemble de tâches et de processus de l'architecture des processus. Il y a, habituellement, une tâche agent pour une classe active, avec toutefois quelques variations : plusieurs agents pour une classe donnée afin d'accroître le débit, ou plusieurs classes appliquées à un seul agent, car leurs opérations sont rarement invoquées, ou afin de garantir leur exécution séquentielle.

Notez qu'il ne s'agit pas là d'un processus linéaire et déterministe menant à une architecture des processus optimale ; trouver un *compromis* acceptable demandera quelques itérations. Il y a de nombreuses autres manières de procéder, comme l'ont montré par exemple Birman et al. ou Witt et al.[7]. La méthode précise d'élaboration de cette correspondance ne fait pas partie des objectifs de cet article, mais on peut l'illustrer à travers un petit exemple.

La figure 14 montre comment un petit groupe de classes d'un hypothétique système de contrôle de trafic aérien peut être appliqué sur des processus.

La *classe de vol* est appliquée sur un ensemble d'*agents de vol* : il y a plusieurs vols à traiter, un grand nombre de stimuli externes, le temps de réponse est critique, la charge doit être distribuée sur plusieurs CPUs. De plus, les aspects liés à la persistance et à la distribution du traitement du vol sont confiés à un *serveur de vol*, qui est dupliqué pour des raisons de disponibilité.

Un *profil* ou une *autorisation* de vol sont toujours subordonnés à un vol, et bien qu'il y ait des classes complexes, elles partagent les processus de la classe de vol. Les vols sont distribués sur plusieurs autres processus, surtout en ce qui concerne les interfaces d'affichages et les interfaces externes.

Une *classe de sectorisation*, qui établit un quadrillage de l'espace aérien afin de déterminer la juridiction des contrôleurs pendant les vols, à cause de ses contraintes de fiabilité, ne peut être traitée que par un seul agent, mais elle peut partager le processus serveur de vol : les mises à jour sont peu fréquentes.

Les *lieux* et *l'espace aérien* ainsi que d'autres informations aéronautiques statiques sont des objets protégés, partagés entre plusieurs classes, rarement mis à jour ; elles sont appliquées sur leur propre serveur, puis transmis aux autres processus.

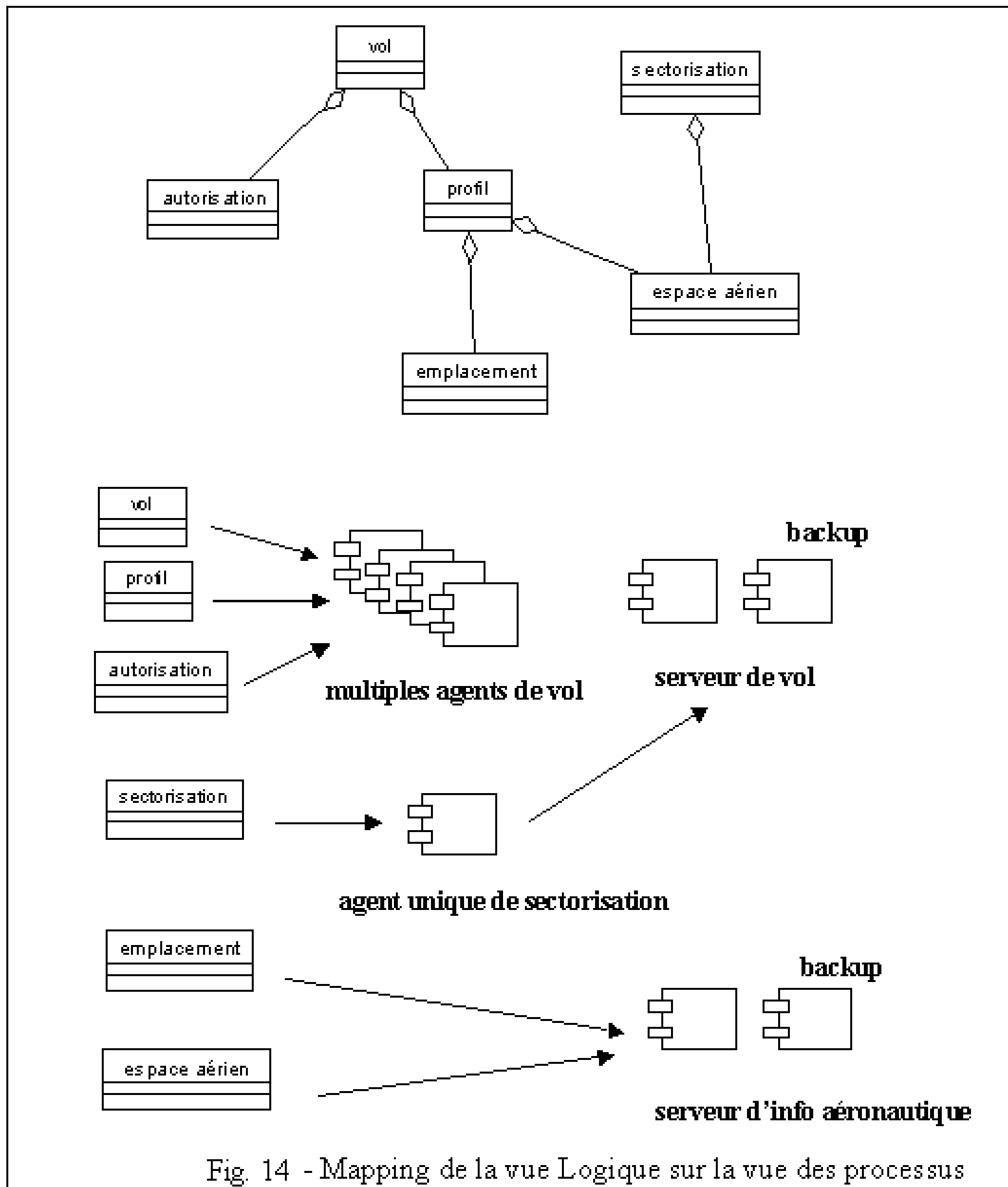

Fig. 14 - Mapping de la vue Logique sur la vue des processus

De la vue logique à la vue d'implémentation

Un classe est habituellement implémentée comme un module, un type dans la partie visible d'un *paquetage* Ada par exemple. Les grosses classes sont décomposées en plusieurs paquetages. Des ensembles de classes intimement liées – des paquetages – sont regroupées en sous-systèmes. Des conditions supplémentaires, telles que l'organisation en équipe, la quantité de lignes de code attendue (habituellement de 5 à 20K SLOC par sous-système), le pourcentage de réutilisation et de factorisation attendu, les principes stricts de traitement par couche (les questions de visibilité), la politique de réalisation et la gestion de configuration, doivent être prises en compte dans la définition des sous-systèmes. Toutefois on obtient habituellement au final une vue qui n'a pas de correspondance unitaire avec la vue logique.

La vue logique et la vue d'implémentation sont très proches, mais elles traitent de problèmes très différents. On a observé que plus le projet est gros, plus la distance entre ces vues est grande. Il en est de même pour la vue de déploiement et la vue des processus : plus le projet est gros, plus la distance entre les vues est grande. Si, par exemple, on compare la figure 3b et la figure 6, il n'y a pas de correspondance une à une entre les paquetages et les couches. Si l'on prend le paquetage « Interfaces externes – Passerelle », son implémentation est répartie

entre plusieurs couches : les protocoles de communication appartiennent à des sous-systèmes dans ou sous la couche 1, les mécanismes généraux de passerelle appartiennent à des sous-systèmes de la couche 2, et les passerelles réelles spécifiques à des sous-systèmes de la couche 5.

De la vue des processus à la vue de déploiement

Les processus et les groupes de processus sont appliqués sur le matériel physique disponible, en différentes configurations pour les tests et le déploiement. Birman décrit des schémas très élaborés à propos de ce type de correspondance dans le projet Isis[5].

Les scénarios font le plus souvent référence à la vue logique, en termes de classes utilisées, et à la vue des processus quand les interactions entre les objets concernent plus d'un «thread » de contrôle.

Adapter le modèle

Toutes les architectures ne nécessitent pas l'intégralité des « 4+1 » vues. Les vues superflues peuvent être supprimées de la description architecturale, comme la vue de déploiement, s'il n'y a qu'un processeur, ou la vue des processus s'il n'y a qu'un processus ou qu'un programme. Pour de très petits systèmes, il est même possible que la vue logique et la vue d'implémentation soient si proches qu'elles ne nécessitent pas de description séparée. Les scénarios eux sont utiles dans tous les cas.

Le processus itératif

Witt et al. proposent 4 phases pour la conception d'une architecture : esquisser, organiser, spécifier et optimiser, divisées en 12 étapes[7]. Ils précisent qu'un retour en arrière peut parfois être nécessaire. Nous pensons que cette approche est trop « linéaire » pour un projet ambitieux et innovateur. On ne sait que trop peu de choses à la fin des 4 phases pour valider l'architecture. Pour notre part, nous préconisons un développement plus itératif, où l'architecture est vraiment prototypée, testée, mesurée, analysée puis raffinée dans les itérations suivantes. En plus de réduire les risques associés à l'architecture, cette approche a d'autres bénéfices pour le projet : formation et familiarisation des équipes avec l'architecture, familiarisation avec les outils, « run-in » des procédures et des outils, etc. (On parle ici de prototypes évolutifs, qui grandissent progressivement pour devenir le système, et non de prototypes exploratoires jetables). Cette approche itérative permet aussi aux exigences d'être affinées, mûries et mieux comprises.

Une approche pilotée par les cas d'utilisation

La fonctionnalité la plus critique du système est enregistrée sous la forme de scénarios (ou cas d'utilisations). Par critique on entend : les fonctions qui sont les plus importantes, *la raison d'être*[8] du système, ce qui est le plus fréquemment utilisé, où qui présente un risque technique majeur qui doit être atténué.

Début :

- On choisit, pour une itération, une petite partie des cas d'utilisation, en fonction de leur risque et de leur criticité. Les cas d'utilisation peuvent être synthétisés afin de résumer un certain nombre d'exigences des utilisateurs.
- Une architecture « de paille » est mise en place. Les cas d'utilisation sont alors « retranscrits » afin d'identifier les abstractions majeures (les classes, les mécanismes, les processus, les sous-systèmes) comme indiqué par Rubin et Goldberg [6] – décomposés en séquences de paires (objet, opération).
- Les éléments architecturaux découverts sont alors disposés sur les 4 plans : logique, des processus, d'implémentation et de déploiement

- Cette architecture est alors mise en œuvre, testée, mesurée, et cette analyse peut détecter certains défauts ou d'éventuelles améliorations.
- Les leçons apprises sont enregistrées.

Boucle :

L'itération suivante peut alors débuter par :

- la réévaluation des risques,
- l'élargissement de la palette decas d'utilisation à prendre en compte
- la sélection de quelques cas d'utilisation supplémentaires permettant d'atténuer les risques ou permettant une meilleure couverture de l'architecture

Alors :

- Essai d'exécution de ces cas d'utilisation sur l'architecture précédente
- Découverte des éléments supplémentaires d'architecture, ou parfois changements significatifs de l'architecture nécessaires afin de répondre à ces scénarios
- Mise à jour des 4 plans principaux : logique, des processus, d'implémentation et de déploiement
- revue des cas d'utilisation existant à partir des changements
- mise à jour de l'implémentation (le prototype architectural) afin de satisfaire le nouveau jeu étendu de cas d'utilisation
- Test. Mesure en charge, si possible dans l'environnement cible réel.
- Les cinq plans sont alors revus afin de détecter d'éventuelles simplifications, réutilisations ou mises en commun
- Les lignes directrices de conception sont mises à jour
- Les leçons apprises sont enregistrées

Fin de la boucle

Le prototype architectural initial évolue afin de devenir le système réel. Heureusement, après 2 ou 3 itérations, l'architecture par elle-même se stabilise : aucune nouvelle abstraction majeure n'est découverte, ni aucun nouveau système ou processus, ni aucune nouvelle interface. Le reste est du domaine de la conception logicielle, où, évidemment, le développement peut continuer en utilisant des procédures et des méthodes similaires.
La durée de ces itérations varie considérablement : avec la *taille* du projet à mettre en place, avec *le nombre de personnes* concernées et leur familiarisation avec le domaine et la méthode, avec le *degré d' « innovation »* du système dans cette organisation de développement. La durée d'une itération peut aller de 2-3 semaines pour un petit projet (10 KSLOC) à 6-9 mois pour un gros système de commande ou de contrôle (700 KSLOC).

Documenter l'architecture

La documentation produite au cours de la conception de l'architecture est enregistrée dans deux documents :

- Un *Document d'Architecture Logicielle*, dont l'organisation est très proche des « 4+1 » vues (cf. fig 15 pour un plan typique)
- Un *Guide de Conception Logicielle*, qui enregistre (entre autres choses) les décisions de conception les plus importantes, qui doivent être respectées pour maintenir l'intégrité architecturale du système.

Page de Titre
Historique
Table des Matières

1. Introduction

1.1. Objectif

1.2. Portée

1.3. Définitions, Acronymes et Abréviations

1.4. Références

1.5. Aperçu

2. Représentation de l'Architecture

3. Objectifs et Contraintes de l'Architecture

4. Vue des Cas d'Utilisation

4.1. Réalisations des Cas d'Utilisation

5. Vue Logique

5.1. Aperçu

5.2. Paquetages de Conception Significatifs pour l'Architecture

6. Vue des Processus

7. Vue de Déploiement

8. Vue d'Implémentation

8.1. Aperçu

8.2. Couches

9. Vue de Donnée (optionnelle)

10. Taille et Performance

11. Qualité

Fig. 15 – Plan d'un Document d'Architecture Logicielle

Conclusion

Ce modèle des «4+1 » vues a été utilisé avec succès sur plusieurs gros projets, avec ou sans personnalisation ou ajustement dans la terminologie[4]. Il permet normalement aux différentes parties prenantes de trouver ce qu'elles veulent voir à propos de l'architecture logicielle. L'approche des ingénieurs systèmes se fait par la vue de Déploiement puis la vue des Processus. Les utilisateurs finaux, les clients, les spécialistes en données par la vue Logique. Les chefs de projet, l'équipe de configuration logicielle la voient eux par la vue d'Implémentation.

D'autres jeux de vues ont été proposés et discutés, chez Rational et ailleurs, par exemple par Meszaros (BNR), Hofmeister, Nord et Soni (Siemens)[9], Emery et Hilliard (Mitre)[10], mais on a remarqué que souvent les autres vues proposées pouvaient facilement être regroupées dans les 4 que l'on a décrit. Par exemple, une vue de Coût et «Schedule » peut se retrouver dans la vue d'Implémentation, une vue de Données dans la vue Logique, une vue d'Exécution dans une combinaison entre la vue de Déploiement et la vue des Processus.

L'IEEE travaille actuellement sur un projet de norme de description d'architecture[11] utilisant le principe d'emploi de vues différentes, chacune s'adressant à des interlocuteurs différents.

Vue	*Logique*	*Processus*	*Implémentation*	*Déploiement*	*Scénarios*
Composants	Classe	Tâche	Module, Sous-système	Nœud	Etape, Scripts
Connecteurs	association, héritage,	Rendez-vous, Message, broadcast, RPC, etc.	dépendance de compilation, clause « with », « include »	Média de communication, LAN, WAN, bus, etc.	
Conteneurs	Paquetage	Processus	Sous-système (bibliothèque)	Sous-système physique	Web
Partie-prenantes	Utilisateur final	Concepteur système, intégrateur	Développeur, gestionnaire (manager)	Concepteur système	Utilisateur final, développeur
Intérêts	Fonctionnalité	Performance, disponibilité, tolérance au panne logicielle, intégrité	Organisation, réutilisation, portabilité, ligne de produit	Capacité à évoluer, performance, disponibilité	Compréhension
Outil	Rose	Rose	Rose	Rose	Rose

Table 1 – Résumé du modèle des « 4+1 » vues

Remerciements de l'auteur

Le modèle des « 4+1 » vues doit son existence à de nombreux collègues de Rational, à Hughes Aircraft du Canada, à Alcatel, et à d'autres. Je voudrais en particulier remercier pour leur contributions Ch. Thompson, A. Bell, M. Devlin, G. Booch, W. Royce, J. Marasco, R. Reitman, V. Ohnjec, et E. Schonberg.

Références

[1] D. Garlan & M. Shaw, "An Introduction to Software Architecture," *Advances in Software*

Engineering and Knowledge Engineering, Vol. 1, World Scientific Publishing Co. (1993).
[2] D. E. Perry & A. L. Wolf, "Foundations for the Study of Software Architecture", *ACM Software Engineering Notes*, 17, 4, Octobre 1992, 40-52.
[3] Ph. Kruchten & Ch. Thompson, "An Object-Oriented, Distributed Architecture for Large Scale Ada Systems", *Proceedings of the TRI-Ada '94 Conference*, Baltimore, Novembre 6-11, 1994, ACM, p. 262-271.
[4] G. Booch & J. Rumbaugh & I. Jacobson, *Guide de l'utilisateur UML, Eyrolles.*
[5] K.P. Birman, et R. Van Renesse, *Reliable Distributed Computing with the Isis Toolkit*, IEEE Computer Society Press, Los Alamitos CA, 1994.
[6] K.Rubin & A.Goldberg, *Object Behavior Analysis*, CACM, 35,9(Sept. 1992) 48-62.
[7] B. I. Witt, F. T. Baker et E. W. Merritt, *Software Architecture an Design-Principles, Models, and Methods*, Van Nostrand Reinhold, New-York (1994) 324p.
[8] NdT : en français dans le texte
[9] C. Hofmeister, R. L. Nord et D. Soni, *Applied software Architecture,* Addison-Wesley, Longman (1999)
[10] D. Garlan (ed.), Proceedings of the First Internal Workshop on Architectures for Software Systems, CMU-CS-TR-95-151, CMU, Pittsburgh, 1995.

[11] IEEE AWG, *Recommended practice for Architectural Description*, IEEE P1471, Draft 4, Octobre 1999.