Architecture logicielle : une expérimentation industrielle avec Dassault Systèmes

Rémy Sanlaville

▶ **To cite this version:**

Rémy Sanlaville. Architecture logicielle : une expérimentation industrielle avec Dassault Systèmes. Génie logiciel [cs.SE]. Université Joseph-Fourier - Grenoble I, 2002. Français. NNT : . tel-00004589

HAL Id: tel-00004589
https://theses.hal.science/tel-00004589

Submitted on 8 Feb 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Joseph Fourier - Grenoble I

Thèse

pour obtenir le grade de

Docteur de l'Université Joseph Fourier

(arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

Discipline : Informatique

présentée et soutenue publiquement par

Rémy SANLAVILLE

le 3 Mai 2002

Architecture Logicielle : une expérimentation industrielle avec Dassault Systèmes

Composition du jury :

Président de jury :	F. Ouabdesselam	LSR-IMAG, Université de Grenoble I
Rapporteurs :	M. Riveill	I3S , Université de Nice - Sophia Antipolis
	M. Lemoine	CERT ONERA Toulouse
Examinateurs :	J.-F. Doué	Dassault Systèmes
	J. Estublier	LSR-IMAG, Université de Grenoble I
	B. Latchague	Dassault Systèmes
Directeur de thèse :	Y. Ledru	LSR-IMAG, Université de Grenoble I

Thèse préparée au sein du laboratoire Logiciels, Systèmes et Réseaux, IMAG
et de la société Dassault Systèmes.

Ce travail de thèse ne serait bien entendu pas le même sans la rencontre, l'apport et le soutien de nombreuses personnes (je m'excuse pour ceux que je n'ai pu citer ou que j'ai oubliés).

Je tiens à remercier en premier lieu les différents membres du jury qui m'ont fait l'honneur de participer à ma soutenance de thèse. Je souhaite mentionner F. Ouabdesselam pour avoir présidé ma soutenance de thèse ainsi que mes deux rapporteurs M. Lemoine et M. Riveill pour avoir accepté de lire et d'évaluer mon travail au travers de ce document de thèse. Je pense aussi à l'invitation de M. Lemoine à l'ONERA Toulouse, lors de laquelle nous avons discuté de mes travaux.

Je suis très reconnaissant aussi envers les différentes personnes (Jean-Christophe Janodet, Christophe Saint-Marcel, Frédéric Duclos, Philippe Morat, Anne-Marie Sanlaville...) qui ont passé un temps important à relire mon document de thèse et qui m'ont aidé à l'améliorer.

Je souhaite ensuite citer mon directeur de thèse Yves Ledru et mes différents responsables d'équipe de Dassault Systèmes Philippe Zimny, Jean-François Doué et Frédéric Bacquet qui se sont succédés :

- Yves, nous avons commencé à travailler ensemble lors de mon DEA. J'ai eu la chance de pouvoir continuer ma thèse sous ta direction. Tu m'as toujours conseillé justement, poussé à publier mes travaux et en même temps laissé mener mes recherches comme je le souhaitais. Tu m'a permis d'encore mieux appréhender le milieu de la recherche de par tes connaissances et tes responsabilités. J'ai pu profiter de ton savoir et de ton soutien aux moments importants de ma thèse. Que demander de mieux à un directeur de thèse ? Pour tout cela et pour ton amitié je souhaite vivement te remercier. J'espère que nous pourrons continuer à travailler ensemble dans d'autres contextes.

- Jean-François, nous nous sommes rencontrés dès mon arrivée à Dassault Systèmes. Tu étais chargé de suivre mon travail et tu l'as fait avec rigueur du début jusqu'à la fin. Tu as toujours été là pour donner un avis constructif et pour me soutenir devant les différents problèmes que j'ai pu rencontrer. Un lien d'amitié s'est créé au fil du temps et je te suis reconnaissant pour ton aide.

  Frédéric, tu nous as rejoint en fin de parcours, mais ton soutien a été tout aussi important dans ce moment charnière de fin de thèse. J'ai beaucoup apprécié ton enthousiasme envers mes travaux. Vous (Jean-François, toi-même et bien d'autres) avez toujours eu le souci de faire connaître et valoriser mon travail au sein de Dassault Systèmes.

  Philippe, de par tes responsabilités qui t'ont menées rapidement vers d'autres occupations, notre collaboration n'a été que temporaire. Néanmoins ton accueil à été très chaleureux (ce qui est très important lorsqu'on arrive dans un nouveau lieu) et nous avons eu l'occasion ensuite de nous croiser de temps à autre.

Je voudrais continuer en nommant la société Dassault Systèmes, le laboratoire LSR et l'ensemble des personnes (amis, collègues...) qui m'ont beaucoup apporté. Je vais les regrouper par rapport à mes trois principaux lieux de vie géographique au cours de cette thèse :

- Je commencerai par Suresnes (Paris) où siège la maison mère de Dassault Systèmes. Je souhaite remercier la société Dassault Systèmes et ses responsables pour m'avoir permis de réaliser une recherche appliquée sur leur logiciel CATIA V5. Je leur suis reconnaissant pour m'avoir permis de découvrir le cœur de leur architecture V5 et pour avoir accepté que je divulgue de nombreuses informations au travers de mes publications et de ce document de thèse.

  Je tiens à faire part de ma gratitude envers l'ensemble du personnel de Dassault Systèmes. Je pense particulièrement à Jean-Jacques Auffret qui était le responsable pour cette société de la collaboration entre Dassault Systèmes et le laboratoire LSR, et avec qui j'ai beaucoup discuté pendant ces trois années. Je pense aussi aux différents membres de mon équipe avec qui j'ai passé de très bon moments, dont Christian Ronce pour la patience qu'il a eu pour répondre à mes nombreuses questions sur l'Object Modeler. Je ne peux oublier les très nombreux ingénieurs et responsables de Dassault Systèmes qui ont accepté très simplement de me recevoir afin de passer du temps à m'écouter et à m'expliquer leur travail. Je ne peux malheureusement pas tous les nommer tellement le liste est longue mais je tiens à citer Jacques Bacry, Michael Bonin, Christophe Foucart, Daniel Galton, Serge Grapinet, Francis Klein, Laurent Lalère, Emmanuel Langlois, François-Xavier Menard, Philippe Stoesser, Patrice Verdure... Je tiens aussi à remercier les secrétaires et les personnes de la DRH pour l'aide qu'elles m'ont apportée.

  Je souhaite remercier mes amis de Paris (Piotr et Marie-Aude, Bruno et Laurent, Karen et Jean-Michel, Emmanuelle...) pour les moments passés ensemble.

- Je continuerai par Grenoble où se trouve mon laboratoire de recherche et où j'ai passé la majorité de mon temps. Je tiens à citer en premier Jacky Estublier qui est le responsable pour le laboratoire de la collaboration entre Dassault Systèmes et le laboratoire LSR ainsi que le responsable de l'équipe Adèle dont je fais partie. Il m'a apporté beaucoup et m'a permis de réaliser ma thèse dans de très bonnes conditions. Outre l'aspect matériel dont j'ai pu disposer, il m'a été possible de rencontrer et d'inviter des chercheurs reconnus pour leurs travaux. Je remercie Jorge Villalobos pour la connaissance qu'il m'a apportée au niveau de l'orienté objet et pour le travail réalisé autour de l'OMVT. Je tiens à mentionner Jean-Marie Favre pour nos longues discussions, pour ses nombreuses idées et pour tout ce qu'il n'a pas été possible de réaliser. Je pense aussi à mes deux stagiaires pour le travail effectué avec convivialité. J'ai été aussi très heureux de faire partie de cette équipe pour la bonne ambiance qui y règne. Il est agréable de voir que des étudiants et des chercheurs de

différents pays (Algérie, Colombie, Etats-Unis, Espagne, France, Mexique, Syrie, Vietnam...) puissent vivre en parfaite harmonie. J'ai beaucoup appris de leur culture et apprécié nos longs débats (n'est-ce pas Humberto ?). Je remercie aussi tous les membres (chercheurs, secrétaires, administrateurs réseaux, étudiants...) du laboratoire LSR pour ce qu'ils m'ont apporté et les moments que j'ai partagé avec eux lors de mon DEA et de ma thèse.

Je ne peux oublier tous mes amis Grenoblois qui m'ont soutenu et aidé pendant ces trois années. Je pense à mes co-locataires (Seb, Ludo) et dans le désordre à Isabel, Victor, Claire, Mélanie, Stéphanie, Christophe, Sophie, David, JC, Marlène, Fred, Karine, Fred, Humberto, Gaby, ... Je pense aussi à tous les bons moments passés avec les membres de la maison de jeux et aux choristes du C.U.G.

- Je finierai par Irigny (Lyon) mon village natal où j'ai vécu des moments formidables et où j'ai encore des activités. Je pense à tous mes amis du groupe partage et aux nombreux projets qui ont été menés. Je pense particulièrement à Céline et Emmanuel avec qui j'ai parcouru pendant trois étés le chemin de Saint Jacques de Compostelle et à toutes les rencontres simples et riches que nous avons vécues. Je pense à l'école de musique et mon professeur de piano qui m'ont permis de vivre ma passion pour la musique.

Enfin je tiens à citer tous les membres de ma famille qui me sont chers. J'ai toujours pu compter sur eux et ils ont été présents dans les moments difficiles et les moments de joie. Je remercie mes parents pour tout ce qu'ils m'ont apporté ainsi que mon grand-père, mes frères, mes belles-sœurs et ma sœur pour leur soutien. Je pense aussi à ceux que j'ai perdu. Je terminerai par la nouvelle génération montante avec Manon, Perrine, Thomas et Yoan. Leur innocence et leur joie de vivre m'ont beaucoup aidé.

Résumé

L'architecture logicielle est un domaine récent du génie logiciel qui a reçu une attention particulière ces dix dernières années. Les éditeurs de logiciels ont pris conscience qu'une architecture est un facteur critique dans la réussite du développement et facilite la maintenance et l'évolution du logiciel. Elle contribue à la maîtrise des grands logiciels.

L'architecture logicielle permet d'améliorer ces aspects grâce à l'étude des structures de haut niveau du logiciel. De nombreuses avancées ont été proposées au niveau de la formalisation par des Langages de Description d'Architecture (ADLs : Architecture Description Languages), du raisonnement et de l'analyse au niveau architectural. Bien que la communauté scientifique ait réalisé des progrès significatifs, les résultats restent essentiellement académiques. Les retombées de ces recherches ont du mal à pénétrer le milieu industriel.

Ce rapport de thèse relate notre expérience avec Dassault Systèmes : répondre aux besoins de Dassault Systèmes pour le développement de ses logiciels en utilisant une approche basée sur l'architecture logicielle. Dassault Systèmes est le leader mondial de la Conception Assistée par Ordinateur (CAO) avec son logiciel phare CATIA V5 ($\sim$5 MLoc).

Pour atteindre cet objectif, nous avons parcouru les principales approches pour la description d'une architecture logicielle et tenté de les appliquer dans notre contexte industriel. Nous expliquerons les difficultés que nous avons rencontrées pour les utiliser et montrerons pourquoi ces approches ne sont pas adaptées pour la maintenance et l'évolution d'un logiciel tel que CATIA V5. Nous décrirons notre démarche basée sur l'analyse des besoins architecturaux des différents acteurs de Dassault Systèmes qui a permis de fournir des solutions concrètes et exploitables. Enfin, nous expliciterons ces besoins architecturaux et présenterons les divers prototypes que nous avons développés pour y répondre.

Mots Clés : Architecture Logicielle, Structures Architecturales, ADLs, ANSI/IEEE Std P1471-2000, Maintenance et Evolution d'une Architecture Logicielle, Environnement Architectural, CATIA V5, Dassault Systèmes.

Abstract

Over the past decade software architecture has received increasing attention as an important subfield of software engineering. Practitioners have come to realize that getting an architecture right is a critical success factor for system design, development, maintenance and evolution.

The software architecture domain allows to improve these aspects thanks to the study of the high level structures of a system. A lot of academic work has permitted to improve formalisation (through Architecture Description Languages (ADLs)), reasoning and analysis at the architectural level. Although software architecture is on a much more solid footing than a decade ago, it is not yet established as a discipline that is taught and practiced universally across the software industry.

This PhD report relates our industrial experiment with Dassault Systèmes : to answer Dassault Systèmes' needs by using a software architecture approach. This company is the world leader in Computer Aided Design products thanks to their leading software named CATIA V5 (~5 Mloc).

To achieve this task, we have studied the main research proposals for the description of a software architecture and tried to apply them to our industrial context. We will explain the difficulties encountered and show why we have concluded that these proposals are not suited for the maintenance and the evolution of a system like CATIA V5. We will propose an efficient solution based on the analysis of the architectural needs of the different Dassault Systèmes' stakeholders. After describing these architectural needs, we will conclude with a presentation of the various tools we have developed to meet these requirements.

Keywords : Software Architecture, Architectural Structures, ADLs, ANSI/IEEE Std 1471-2000, Maintenance and Evolution of a Software Architecture, Architectural Environnement, CATIA V5, Dassault Systèmes.

Table des matières

Chapitre 1

Introduction

Le domaine de l'architecture logicielle a été un thème de recherche académique important au cours de ces dix dernières années. Les recherches ont permis d'établir les bases pour l'étude de ce domaine. De nombreuses avancées ont été proposées notamment au niveau des langages de description d'architecture (ADLs : Architecture Description Languages). Ces recherches ont amélioré la formalisation de l'architecture logicielle d'un système par la description du comportement de ce dernier. Ces descriptions architecturales ont été couplées avec divers outils d'analyse afin de vérifier des propriétés de couplage et de cohésion et de détecter d'éventuels interblocages.

Bien que la communauté scientifique ait réalisé des progrès significatifs, les résultats restent essentiellement académiques. Les retombées de ces recherches ont du mal à pénétrer le milieu industriel et de nombreuses questions restent d'actualité : Comment pouvons-nous définir précisement ce qu'est l'architecture logicielle ? Quels liens existe-t-il avec le code source et les méthodes de conception ? Comment assurer qu'une architecture logicielle correspond aux besoins initiaux ? Comment retrouver l'architecture logicielle d'une application existante ? Quels sont les besoins des industriels ?... Autant de questions qui restent aujourd'hui sans véritablement de réponse.

Cette thèse qui s'inscrit dans le cadre de la collaboration entre le laboratoire LSR (Logiciels, Systèmes, Réseaux) et la société Dassault Systèmes pour l'étude de l'architecture logicielle de CATIA V5 ($\sim$5 MLoc), a permis de confronter les résultats académiques à une réalité industrielle. A travers cette expérience industrielle, nous n'avons pas la prétention de fournir des réponses à toutes ces interrogations mais plutôt d'apporter une pierre à l'édifice pour la compréhension de l'architecture logicielle et des pratiques associées. Tout au long de ce rapport de thèse, nous tenterons de comprendre pourquoi les industriels ne prennent pas en compte les avancées académiques sur l'architecture logicielle alors que ce domaine semble pouvoir fournir des réponses à leurs besoins pour la conception, la maintenance et l'évolution de leurs logiciels.

Un petit historique

De la crise du logiciel au génie logiciel

A la fin des années 60, lors d'une conférence sur le thème "la crise du logiciel" est apparu le terme "génie logiciel" [NR69]. Cette crise du logiciel provenait directement de l'avènement des ordinateurs de troisième génération. Ces nouvelles machines rendaient possible des applications jusqu'alors jamais réalisées.

Les mécanismes utilisés pour les petits systèmes voyaient leur limite sur ces grosses applications. En effet, les gros projets étaient en retard de quelques années, le coût de production était bien supérieur à l'estimation d'origine, les systèmes étaient peu fiables, difficiles à maintenir et avec de faibles performances. De nouvelles techniques et méthodes pour concevoir et maîtriser ces nouveaux logiciels devenaient indispensables. Il était alors nécessaire de passer d'une démarche artisanale à une discipline d'ingénieur. Il fallait créer l'ingénierie pour le logiciel.

De nombreux thèmes de recherche se sont alors développés et ont participé à l'élaboration de ce nouveau domaine de l'informatique : la spécification et l'analyse des besoins, la conception et la modélisation, les langages de programmation et les environnements de développement, la validation et le test, l'ingénierie inverse et la maintenance, la gestion de projet... [Som92, Fin00].

Les premiers fondements pour la conception des logiciels

Durant les années 70 et 80, différents travaux se sont concentrés sur l'étude du développement, de la maintenance et de l'évolution des logiciels. Ces recherches ont montré l'intérêt de s'abstraire du code source et de dissocier les aspects de conception des aspects de programmation. Ces différents résultats se sont succédés et ont contribué à mieux concevoir les logiciels en les structurant mieux :

- L'étude de la structure d'un logiciel a commencé en 1968 quand E. Dijkstra montra qu'il était avantageux de s'intéresser à la structure d'un programme plutôt qu'à sa programmation [Dij68]. Il créa et utilisa en premier la structure en couches pour écrire un système d'exploitation. Il souligna les bénéfices de cette structure pour le développement et la maintenance.

- D. Parnas, quant à lui, préconisait l'utilisation de la modularisation comme une décomposition de haut niveau du système pour améliorer la flexibilité et la compréhension [Par72].

- Un nouvel apport fût réalisé grâce à F. DeRemer et H. Kron en opposant "Programming-in-the-large versus Programming-in-the-small" [DK76]. Les langages de l'époque ne permettaient que de rester au niveau du code ("Programming-in-the-small") et ne permettaient pas un raisonnement à un niveau plus élevé ("Programming-in-the-large"). Ceci ne permettait pas la

réalisation des recommandations faites par les recherches sur la programmation structurée (trouver des solutions lisibles, compréhensibles et modifiables). Les auteurs argumentaient que la création des modules de programme et la connexion de ceux-ci sont deux efforts de conception différents. Afin d'aider l'effort de connexion, ils inventèrent le premier "Module Interconnection Language" (MIL) nommé MIL75. Ensuite d'autres MILs et des systèmes utilisant ces MILs sont apparus [PDN86]. Nous pouvons d'ailleurs remarquer que les concepteurs du langage de programmation Modula et ensuite ADA se sont inspirés de ces travaux pour la notion de paquetage [Tic92].

- Il y a eu dans les années 75/85 un gros travail sur les modules qui a débouché sur les types abstraits de données (Abstract Data Type : ADT). La conception d'un type de données passe par une description formelle de chacune de ses opérations. Plusieurs langages dont CLU [GL86] ont été construits pour faciliter la gestion de ces types abstraits de données.

- Au début des années 80, de nombreuses recherches ont été réalisées par l'université de Carnegie Mellon pour pouvoir faire communiquer des systèmes écrits séparément dans des langages éventuellement différents. Pour pouvoir réaliser cela, les chercheurs ont créé un langage sophistiqué appelé IDL (Interface Description Language) [NS87]. Les modèles de composants récents (COM/DCOM, CORBA, .NET) incorporent leur propre IDL.

- Pendant ce temps, de nombreuses méthodes de conception et techniques de spécification ont émergé [RW80, Ber81, BGMS96] :

  - Au niveau des méthodes, les plus connues sont les méthodes SADT (System Analysis and Design Technique) [RKES77], SD (Structured Design) [YC79] pour les méthodes fonctionnelles et les méthodes OOA (Object-Oriented Analysis) [CY91], OMT (Object Modeling Technique) [RBP+91] et celle de Grady Booch [Boo91] pour les méthodes objets. Dans les années 90, un effort de standardisation va permettre de regrouper les notations de ces différentes méthodes en une seule nommée UML pour Unified Modeling Language [UML97].

  - Au niveau de la spécification, il y a eu une évolution de l'énoncé informel (langage naturel) aux techniques semi-formelles ou graphiques (modèle entité-relation...) pour arriver aux spécifications formelles (Types abstraits algébriques, VDM, Z, B...) [Gau94, JLNP96, Lam00]. L'avantage des spécifications formelles est qu'elles reposent sur des sémantiques bien définies permettant ainsi de vérifier des propriétés et de lever toute ambiguïté.

L'ensemble de ces travaux a abouti à une amélioration et une meilleure maîtrise de la conception d'un logiciel. Toutes ces avancées ont permis de réaliser des systèmes complexes atteignant des centaines de milliers de lignes de code. Mais il était aussi possible de faire le constat que ces avancées n'ont pas permis de résoudre toutes les difficultés : combien de projets ne finissaient pas dans les temps impartis et dont le coût de production était largement supérieur à celui prévu initialement ? [Wie94]

Des besoins et des logiciels de plus en plus complexes

Parallèlement aux avancées réalisées au niveau du génie logiciel, le matériel et les technologies ont considérablement été améliorés. Avec l'arrivée de ces nouvelles machines plus puissantes et moins onéreuses, il était possible d'imaginer des applications encore plus importantes en taille et en complexité afin de répondre aux besoins sans cesse grandissant des utilisateurs.

Les stations de travail étaient devenues de plus en plus un outil de travail indispensable et les ordinateurs personnels faisaient leur apparition. Les sociétés ont commencé à s'équiper massivement de ces nouvelles machines pour automatiser au mieux les différentes tâches quotidiennes. Les applications de gestion et les applications assistées par ordinateur (CAO/CFAO, PAO...) se sont fortement développées. Les premiers réseaux d'entreprise sont apparus ouvrant la porte au travail collaboratif et aux applications distribuées.

Ces nouveaux besoins ainsi que l'augmentation de la taille et de la complexité des logiciels ont demandé de nouvelles technologies et méthodologies pour concevoir de tels logiciels avec un temps et un coût de production toujours moindres. Les recherches en génie logiciel étaient donc toujours d'actualité et de nouvelles approches sont apparues dans les années 90 : les systèmes distribués, les modèles de composants, la création d'un standard de modélisation à travers UML, l'architecture logicielle...

Aujourd'hui, les gros logiciels représentent des millions de lignes de code ($\sim$30 millions pour Windows XP de Microsoft, $\sim$5 millions pour CATIA V5 de Dassault Systèmes...) avec des développements fortement concurrents ($\sim$1 000 ingénieurs pour Dassault Systèmes). Réaliser et faire évoluer de tels logiciels reste encore un défi.

L'avénement du domaine de l'architecture logicielle

Les concepteurs de logiciels se sont peu à peu rendus compte du rôle déterminant que joue l'architecture logicielle dans la réussite du développement, de la maintenance et de l'évolution de leur système. La conception d'une bonne architecture logicielle peut amener à un produit qui répond aux besoins des clients et qui peut être modifié facilement pour rajouter une nouvelle fonctionnalité, alors qu'une architecture inappropriée peut avoir des conséquences désastreuses jusqu'à l'arrêt du projet [Gar00].

De nombreux travaux ont ainsi fait apparaître la notion d'architecture logicielle comme un champ explicite du génie logiciel. Bien que ce domaine tire parti des travaux des années 70 et 80 décrits précédemment, il se veut encore plus formel et plus orienté sur l'architecture des systèmes que sur le niveau du code.

Les recherches effectuées depuis ces dix dernières années ont permis de réaliser de nombreuses avancées dans ce domaine. L'architecture logicielle est maintenant beaucoup plus visible et est devenue une activité de conception explicite dans le processus de développement. Il n'est plus rare de voir apparaître sur les cartes de visite et les curriculum vitæ la mention d'architecte logiciel. La majorité des conférences en génie logiciel ont des workshops ou des sessions dédiés à l'architecture logicielle.

Les travaux effectués durant ces années ont permis de réaliser trois avancées majeures : le développement de langages de description d'architecture et d'outils associés [MT00, Gar00], l'émergence du thème sur l'architecture de ligne de produits (Product Line Architecture) [Don00, JRL00] ainsi que la formalisation d'architectures logicielles standard et la définition d'un vocabulaire commun [GS93, EHP+96] et la promulgation de ce domaine à travers des livres [SG96, BCK98, HNS00] et des cours dans les universités.

Néanmois, malgré le fait que l'architecture logicielle s'approche maintenant plus d'une science que d'un artisanat, elle reste encore immature. Les progrès réalisés par la communauté scientifique restent essentiellement académiques et les retombées de ces recherches ont du mal à pénétrer le milieu industriel [Gar00].

La collaboration entre Dassault Systèmes et le laboratoire LSR

En 1998, la société Dassault Systèmes et le laboratoire Logiciels, Systèmes, Réseaux (LSR) de l'université de Grenoble ont entrepris une collaboration qui avait pour vocation de partager les connaissances théoriques et techniques respectives :

- D'une part, Dassault Systèmes souhaitait acquérir des connaissances dans divers domaines du génie logiciel et faire évoluer ses méthodes et ses outils pour ses développements présents et futurs.
- D'autre part, le LSR désirait développer des recherches académiques et appliquées en s'appuyant sur des cas concrets grâce à un éditeur de logiciels majeur capable de valider les solutions proposées.

La taille et la complexité des logiciels de Dassault Systèmes étant très élevées, il est difficile de contrôler leur développement, leur maintenance et leur évolution. Les recherches entreprises dans le cadre de cette collaboration devaient proposer des solutions concrètes pour ces besoins. Pour cela, les travaux se sont essentiellement concentrés autour du domaine de l'architecture logicielle pour le logiciel CATIA V5 de Dassault Systèmes. Trois thématiques de recherche ont été définies :

1. Architecture Logicielle : Aspects Fonctionnels
2. Architecture Logicielle : Aspects Non Fonctionnels
3. Evolution d'une Architecture Logicielle

Place de la thèse dans la collaboration

Cette thèse s'inscrit dans le cadre de cette collaboration où nous nous sommes principalement préoccupés de la première thématique sur les aspects fonctionnels de l'architecture logicielle de CATIA V5. Ceci implique entre autres que les directions de travail prises pour cette thèse ont été en relation avec les autres sujets afin d'avoir une vision cohérente pour la collaboration.

Le but de nos recherches était de réaliser des descriptions architecturales de la structure statique de CATIA V5 afin d'améliorer le processus de développement : aider la communication entre les divers intervenants, faciliter la spécification et le développement, réaliser des analyses architecturales... Pour prendre en compte l'existant de CATIA V5, ces descriptions architecturales devaient être extraites automatiquement du code source. Nous avons donc beaucoup interagi avec les personnes qui travaillaient sur le thème de l'évolution de l'architecture logicielle pour les aspects d'ingénierie inverse. Un certain nombre d'outils ont dû être développés afin d'atteindre ces objectifs.

Contributions de cette thèse

Cette collaboration avec Dassault Systèmes sur des aspects d'architecture logicielle autour de CATIA V5 est une expérience industrielle assez unique de par la taille (∼5 MLoc) et le développement fortement concurrent (∼1 000 ingénieurs) de ce logiciel. La gestion d'un développement de cette ampleur amène forcément à une réflexion architecturale pour maîtriser une telle complexité. Au cours de nos travaux, nous avons été confrontés à une multitude de besoins architecturaux des acteurs de Dassault Systèmes[1] ce qui nous a donné la possibilité d'explorer diverses pistes de recherche.

Nos travaux permettent d'apporter des contributions autant au niveau académique qu'au niveau industriel pour la société Dassault Systèmes :

- **Etat de l'art** : un important travail d'état de l'art a été fourni afin de connaître et d'essayer d'appliquer les contributions de ce domaine. Nous présenterons les principaux travaux en lien avec nos recherches et nous les comparerons à nos résultats (cf. chapitres 4, 5 et 7).
- **Conceptualisation de l'architecture V5** : nous avons fourni une vision externe de l'architecture logicielle de CATIA V5 en modélisant cette architecture par trois structures architecturales. Nous décrirons chacune de ces structures en détaillant leur rôle spécifique ainsi que leurs constructions architecturales (cf. section 6.1). Afin d'affiner la compréhension de l'architecture logicielle de

[1] Les acteurs de Dassault Systèmes correspondent aux différents ingénieurs de Dassault Systèmes qui sont impliqués dans le processus de développement de CATIA V5 (cf. section 6.2.2) : programmeurs, product managers (ou chefs de produit), release managers, packaging managers...

CATIA V5, nous avons aussi pris en compte dans notre conceptualisation l'environnement de développement en modélisant les différents acteurs et leurs intérêts respectifs par rapport à ces structures architecturales (cf. section 6.2).

- **Proposition de solutions concrètes** : nous avons développé un certain nombre de prototypes pour pouvoir produire des descriptions architecturales de CATIA V5 et les exploiter. Nous présenterons ces outils pour la visualisation des concepts architecturaux de CATIA V5, la vérification de règles de sémantique statique pour les composants Object Modeler de CATIA V5 et la simulation d'une modification pour l'évolution de l'architecture de CATIA V5 (cf. chapitre 7). L'ensemble de nos prototypes prennent en compte le code source complet de CATIA V5 grâce à un analyseur automatique qui nous garantit le lien entre nos descriptions architecturales et le niveau du code. Nous avons aussi fourni aux divers acteurs de l'entreprise un langage graphique des éléments architecturaux de CATIA V5 qu'ils sont appelés à manipuler.

- **Capitalisation de notre expérience industrielle** : un effort particulier a été réalisé pour capitaliser et retranscrire au mieux notre expérience. Au cours de ce rapport de thèse, nous montrerons les difficultés que nous avons rencontrées pour appliquer les résultats académiques dans notre contexte (cf. section 5.2.1). Nous décrirons comment nous avons procédé pour fournir des solutions concrètes aux besoins architecturaux de Dassault Systèmes pour CATIA V5 (cf. section 5.3). Nous illustrerons nos propos par de nombreux exemples de la pratique quotidienne et des besoins des différents acteurs de Dassault Systèmes (cf. chapitre 7).

Bien que nos travaux soient fortement liés à notre contexte avec CATIA V5, nous avons porté une attention particulière pour nous abstraire au mieux de celui-ci afin de proposer des solutions réutilisables dans d'autres contextes. Nos propositions correspondent à la technologie de composants de Dassault Systèmes mais nous croyons qu'elles peuvent être adaptées à des technologies standard (COM, CORBA et CCM, EJB, .NET).

Plan de la thèse

Ce rapport de thèse est divisé en trois grandes parties :

- **Description du contexte de la thèse** : pour bien comprendre nos travaux, il était important de bien situer le contexte de nos recherches. Cette partie présente donc d'une part Dassault Systèmes et son logiciel phare CATIA et d'autre part ce que signifie le terme architecture logicielle. Concernant la présentation du logiciel CATIA, nous insisterons sur les besoins en génie logiciel de Dassault Systèmes qui l'ont poussé à définir une architecture logicielle particulière pour la version 5 de ce logiciel. Au niveau du terme architecture logicielle, nous reprendrons les principales définitions qui ont été formulées par la communauté académique et proposerons notre point de vue qui servira tout au long de ce rapport.

- **Explication et modélisation de l'architecture V5** : nos travaux se sont concentrés sur le logiciel CATIA V5 qui repose sur l'architecture V5 de Dassault Systèmes. Pour étudier l'architecture de CATIA V5, il a fallu la comprendre et la représenter. Comme notre collaboration nous imposait de prendre en compte l'existant du code source ($\sim$5 MLoc), les approches existantes pour la description d'une architecture logicielle ne pouvaient pas être utilisées. Nous avons donc défini notre propre approche basée sur une analyse des besoins des différents acteurs du processus de développement. Cette deuxième partie fait la synthèse des différentes techniques pour décrire une architecture logicielle et récapitule les différentes raisons qui nous ont amenés à utiliser notre propre approche pour réaliser automatiquement les descriptions architecturales de CATIA V5 à partir de son code source. Nous présenterons aussi de façon détaillée l'architecture V5 de Dassault Systèmes à travers la modélisation que nous en avons faite par l'intermédiaire de trois structures architecturales. L'explication des différents concepts de l'architecture V5 sera utile pour la troisième partie.

- **Présentation de nos résultats et de nos expérimentations** : cette architecture V5 propose des mécanismes efficaces et flexibles pour le développement concurrent de CATIA V5. Mais les bonnes propriétés de cette architecture impliquent aussi des difficultés pour pouvoir la maîtriser. Cette troisième partie présente les besoins architecturaux des différents acteurs de Dassault Systèmes dans le processus de développement de CATIA V5 ainsi que les solutions que nous avons proposées pour faciliter l'utilisation de cette architecture V5. Nous présenterons les différents prototypes que nous avons développés et utilisés pour fournir des réponses concrètes aux besoins des acteurs de Dassault Systèmes. Nous présenterons ensuite notre point de vue qui est de disposer d'un environnement architectural intégré au processus de développement. Nous nous abstrairons de notre contexte avec Dassault Systèmes pour proposer un environnement dédié à l'architecture logicielle. Nous finirons ce rapport de thèse en ouvrant le débat sur diverses réflexions autour de l'architecture logicielle tirées de notre expérience avec Dassault Systèmes.

Notre discours se succèdant tout au long de ces différentes parties, nous invitons le lecteur à suivre cet ordre.

Première partie

Contexte de la thèse : Dassault Systèmes et l'Architecture Logicielle

Pour comprendre les travaux qui ont été réalisés dans cette thèse, il est important de s'imprégner de notre contexte lié à Dassault Systèmes et au domaine de l'architecture logicielle.

Cette première partie est découpée en trois chapitres :

- Le chapitre 2 fournit une présentation globale de Dassault Systèmes et de son logiciel phare CATIA. Nous nous attacherons plus particulièrement sur la version 5 de ce logiciel puisque nos travaux ont porté sur cette version.
- Le chapitre 3 récapitule les principaux besoins qui ont amené Dassault Systèmes à concevoir l'architecture V5 qui a permis le développement de CATIA V5.
- Le chapitre 4 est une introduction au domaine de l'architecture logicielle. Nous décrirons comment les différents travaux de la communauté scientifique définissent l'architecture logicielle avant de proposer notre propre vision.

Cette partie est importante dans le sens où elle permet de décrire le contexte dans lequel nous avons évolué ainsi que notre vision de l'architecture logicielle.

Chapitre 2

Dassault Systèmes et son logiciel phare CATIA

2.1 Présentation de Dassault Systèmes

Dassault Systèmes [DS] est un leader mondial sur le marché des logiciels 3D permettant aux industries manufacturières de gérer le cycle de vie de leurs produits. Ses logiciels de gestion du cycle de vie du produit (PLM : Product LifeCycle Management) ont pour fonction de créer et simuler l'intégralité de la "vie" d'un produit, depuis sa conception initiale jusqu'à la fabrication du produit final. Les outils développés par Dassault Systèmes permettent de concevoir aussi bien de simples objets de la vie quotidienne telle qu'une montre ou un appareil électroménager, que des systèmes infiniment plus complexes comme un paquebot, un avion, une voiture ou une usine.

Dassault Systèmes structure ses activités et commercialise ses produits selon deux types de marchés PLM :

- **Le marché axé sur les procédés** dont les applications logicielles permettent la création, la production et la maintenance des produits. Sur ce marché, Dassault Systèmes propose CATIA (analyse et conception de produits), DELMIA (définition, contrôle et simulation du processus de production), ENOVIA et SmarTeam (gestion de produits et collaboration).
- **Le marché axé sur la conception** qui se concentre principalement sur les besoins en conception des composants de produits, pris individuellement. Sur ce marché, c'est la solution SolidWorks qui est proposée.

Afin de répondre aux besoins de ses clients, Dassault Systèmes a défini une architecture spécifique appelée Version 5 ou V5, dont le lancement est intervenu fin 1999 avec CATIA V5. Le développement de l'architecture V5 a commencé au milieu des années 90 et a suivi un processus incrémental. De nouveaux concepts et caractéristiques ont été rajoutés au fur et à mesure que les besoins se faisaient sentir. Après plusieurs années d'utilisation et d'évolution, cette architecture V5 s'est stabilisée et est optimisée.

Cette architecture V5 constitue la base de développement pour la plupart des logiciels de Dassault Systèmes. Ainsi CATIA, ENOVIA et DELMIA sont basés sur l'architecture V5.

2.2 Présentation de CATIA V5

CATIA [CAT] est devenu la référence du marché des logiciels de CAO/FAO avec près de 19 000 clients et de 180 000 postes de travail installés [DS00]. Toutes les branches industrielles sont représentées et plus particulièrment les secteurs de l'aéronautique (AIRBUS, BOEING...), de l'automobile (BMW, DAIMLER CHRYSLER, RENAULT, TOYOTA...) et du ferroviaire (ALSTOM Transport...).

Au milieu des années 90, Dassault Systèmes a entrepris une refonte complète de son logiciel en développant l'architecture V5 basée sur les dernières technologies de l'époque. Il a ainsi été décidé de passer d'une conception procédurale (Fortran, CATIA V4) à une conception orientée objet (C++, CATIA V5). Cet énorme effort de redéveloppement a abouti à la fin de l'année 1999 au lancement commercial de CATIA V5.

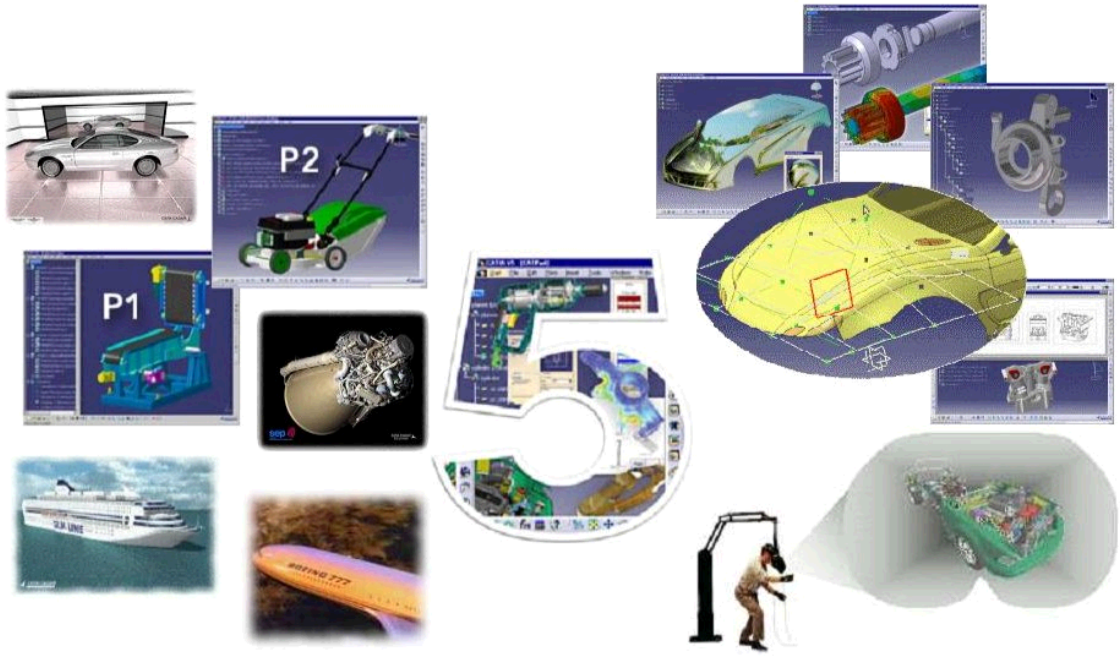


FIG. 2.1 – Une vue d'ensemble de CATIA V5

Cette nouvelle version de CATIA couvre un large éventail des besoins industriels (cf. figure 2.1) :

- **Fonctionnalités puissantes** : permettent à un utilisateur de concevoir, analyser, simuler ainsi que de réaliser du rendu réaliste sur ses produits.

- **Gamme de produits (P1-P2-P3)** : fournit des solutions adaptées aux besoins des petites et moyennes entreprises ainsi que de grosses sociétés industrielles dans tous les domaines : biens de consommations, électrique et électronique, automobile, aéronautique, ferroviaire, construction navale...
- **Interopérabilité** : avec la plupart des technologies et standards industriels tels que STEP, XML, Java, CORBA et COM.
- **Indépendant de la plate-forme** : le code source n'est pas dépendant d'une plate-forme particulière et CATIA V5 peut être exécuté sur 7 plate-formes Unix ou Windows différentes.
- **Architecture ouverte** : des développeurs de logiciels extérieurs à Dassault Systèmes peuvent implanter des fonctionnalités complémentaires et les intégrer dans CATIA V5 pour leur domaine d'activité.

Chapitre 3

Besoins de Dassault Systèmes en génie logiciel

Pour bien comprendre une architecture, il est nécessaire de connaître les motivations sous-jacentes. Nous allons donc présenter les principaux besoins[1] qui ont amené Dassault Systèmes à développer l'architecture V5 (que nous présenterons en détail dans la partie suivante).

Nous discuterons en premier lieu des besoins liés au développement interne à Dassault Systèmes (cf. section 3.1). Ensuite nous parcourrons les besoins liés à ceux des clients et partenaires de Dassault Systèmes (cf. section 3.2). Enfin nous expliquerons pourquoi il n'a pas été possible pour Dassault Systèmes de réutiliser une architecture existante (cf. section 3.3).

3.1 Les besoins internes

Dassault Systèmes est l'un des plus importants éditeurs de logiciels en Europe. Pour s'en convaincre, il suffit de présenter quelques chiffres significatifs :

- **Près de 5 millions de lignes de code** : CATIA V5 comptabilise près de 5 millions de lignes de code correspondant à plus de 50 000 classes C++. Les ingénieurs de Dassault Systèmes ont une vaste expérience des composants logiciels puiqu'ils ont construit et utilisé plus de 8 000 composants. Les différentes versions de CATIA représentent plus de 1 000 000 de fichiers.
- **1 000 ingénieurs** : Dassault Systèmes gère un fort développement concurrent où plus de 1 000 ingénieurs développent en parallèle pour améliorer les fonctionnalités de ses produits.
- **Release[2] tous les 4 mois** : afin de rester compétitif, Dassault Systèmes développe une release tous les 4 mois. Ceci correspond à un temps de cycle extrêmement court. C'est un logiciel qui évolue donc très rapidement.

[1] Nous ne décrirons ici que ceux qui concernent directement cette thèse. Il en existe d'autres comme la qualité (objets de test), la performance... que nous n'avons pas particulièrement étudiés.

[2] Dassault Systèmes utilise le terme release pour indiquer une sous-version. Par exemple, CATIA V5R8 est la version 5 release 8 de CATIA.

Dassault Systèmes a dû relever de nombreux défis pour pouvoir maîtriser une telle complexité. Cette maîtrise est le fruit de nombreuses années d'expérience qui ont permis d'acquérir d'importantes connaissances en génie logiciel.

3.1.1 Gestion de Configuration

La gestion de configuration comprend à la fois la gestion du développement concurrent et la gestion de versions.

Un important développement concurrent
Faire cohabiter plus de mille ingénieurs sur une même gamme de produits n'est pas une chose aisée. Le faire avec des contraintes de productivité extrêmement exigeantes est d'autant plus difficile. En général, à partir d'un certain seuil, productivité et développement concurrent ne vont pas de pair [Bro95].

Gestion de plusieurs versions
Bien que CATIA V5 soit de plus en plus utilisé et doive à terme remplacer la version 4, Dassault Systèmes n'a pas pour autant arrêté la maintenance et le développement de cette dernière. De ce fait, il existe plusieurs releases pour les versions 4 et 5 à maintenir et à faire évoluer conjointement.

Pour répondre à ces besoins, Dassault Systèmes dispose d'outils et de technologies spécifiques. Par exemple, les ingénieurs de Dassault Systèmes ont adapté l'outil de gestion de configuration Adèle [EC94] pour leur processus de développement. De plus, pour faciliter le travail concurrent, il faut disposer de technologies qui permettent de travailler sur de petits modules indépendants. L'architecture V5 a été conçue dans ce but et offre des mécanismes qui permettent à plusieurs ingénieurs de travailler simultanément sur un même composant avec un minimum d'impact.

3.1.2 Optimisation du "Build"

Compiler et construire un système aussi grand que CATIA conduit à des temps de compilation importants (plusieurs jours). Il n'est pas envisageable pour Dassault Systèmes que ses ingénieurs passent la majorité de leur temps à attendre la fin de leurs compilations. De plus, lors de leurs développements, les ingénieurs de Dassault Systèmes peuvent utiliser simultanément plusieurs langages supportés par l'architecture V5. Trouver le bon enchaînement de compilation pour ces différents langages est une tâche qui peut être difficile et longue.

Dassault Systèmes a donc mis en place un processus qui construit entièrement CATIA V5 une fois par semaine et réalise des compilations incrémentales locales à chaque équipe de développement le reste du temps. En plus de ce processus, d'autres mécanismes ont été définis pour limiter au strict minimum la compilation. Ces mécanismes sont essentiellement de deux natures :

- **Un compilateur pour l'architecture V5** : Dassault Systèmes a défini son propre compilateur qui permet de faire cohabiter les différents langages supportés par l'architecture V5. Ce compilateur de haut niveau retrouve et utilise les compilateurs correspondant aux langages utilisés en prenant en compte la plate-forme cible. De plus, il calcule automatiquement le graphe de dépendance pour ne recompiler que les fichiers nécessaires. Ainsi, il optimise au mieux le temps de compilation.
- **Minimiser les liens de dépendance de compilation** : grâce à un mécanisme d'extension particulier, un ingénieur peut étendre un composant sans pour autant dépendre de ce dernier lors de la compilation.

3.2 Les besoins externes

La diversité des secteurs d'activités des clients de Dassault Systèmes demande en plus de fonctionnalités communes, un certain nombre de fonctionnalités orientées métier. Ainsi, CATIA permet de concevoir et d'analyser tout type de pièce (mécanique, plastique, électronique...) pour les différents secteurs de l'industrie (automobile, aéronautique, biens de consommation...).

Les ingénieurs de Dassault Systèmes ont donc décidé de concevoir CATIA comme une infrastructure évolutive sur laquelle il est possible d'intégrer des produits de façon à rajouter des caractéristiques et des services spécifiques à un domaine particulier. Actuellement plusieurs centaines de produits existent pour différents domaines d'applications comme la tôlerie, la fabrication de pièces plastiques... Ces produits peuvent être développés par Dassault Systèmes, par des partenaires (sociétés ayant un contrat spécifique avec Dassault Systèmes pour développer et vendre des produits pour CATIA) ou par le client (cf. figure 3.1). Pour que les clients et les partenaires puissent réaliser et intégrer leurs produits à CATIA, Dassault Systèmes fournit un kit de développement nommé CAA pour Component Applications Architecture.

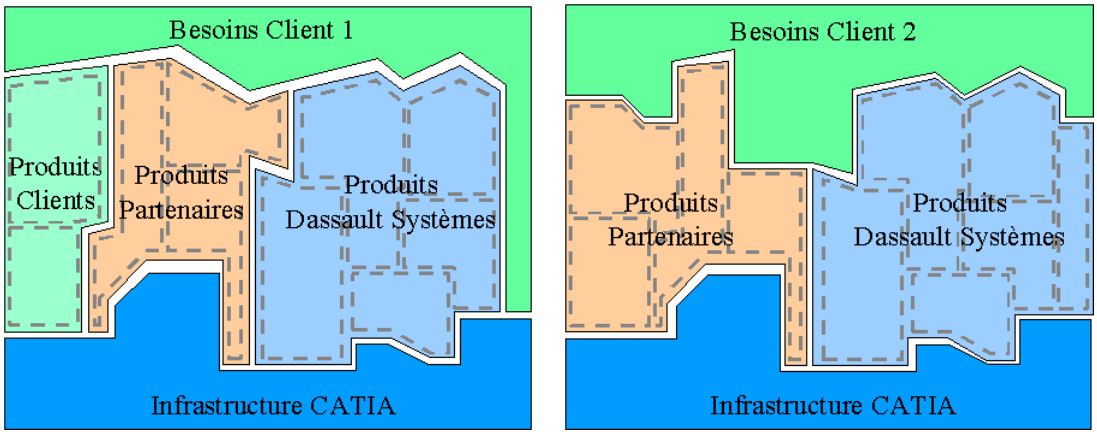


FIG. 3.1 – Différentes configurations de CATIA

CATIA est donc un système de CAO/CFAO qui peut être adapté finement pour répondre aux besoins des clients. CATIA définit en réalité une ligne de produits.

Il existe effectivement chez les clients de Dassault Systèmes un grand nombre de configurations différentes de CATIA. Bien que le thème architecture de ligne de produits (Product Line Architecture) ait pris de plus en plus d'importance ces dernières années [Don00, JRL00], notre recherche n'a pas eu pour objectif de se focaliser sur ce problème. Cette diversité de configurations est possible grâce à une conception ouverte de l'architecture V5 qui permet aux clients et partenaires de Dassault Systèmes d'intégrer simplement leurs propres fonctionnalités. Un des mécanismes associé à cette architecture permet à un développeur de rajouter ses fonctionnalités à des composants de Dassault Systèmes sans qu'il ait besoin de posséder le code source de ceux-ci et sans avoir à recompiler entièrement l'application. Ce mécanisme, nommé extension, est utilisé intensivement en interne ce qui est un très bon critère pour juger de la qualité, de la souplesse et de la fiabilité de ce mécanisme. A notre connaissance ce mécanisme n'existe pas dans d'autres technologies de composants et correspond aux meilleures techniques développées à ce jour pour l'adaptation des logiciels [DES01].

3.3 Pourquoi avoir développé une architecture particulière ?

Pour pouvoir développer un logiciel tel que CATIA V5, Dassault Systèmes a rapidement compris que C++ en tant que tel ne suffisait pas. Ce langage a été choisi pour sa maturité industrielle mais il pose certains problèmes lorsqu'il est utilisé pour de grands logiciels :

- **L'extensibilité** : comme d'autres langages orienté objet, C++ offre des mécanismes d'extensibilité (comme par exemple l'héritage) mais ils sont généralement limités et demandent souvent la disponibilité du code source de l'entité étendue. Pour des raisons de propriété industrielle, Dassault Systèmes ne souhaite évidemment pas fournir son code source à ses clients. Dassault Systèmes a besoin d'une technologie qui permette à un tiers d'étendre et adapter finement CATIA sans avoir besoin de recompiler ou d'accéder au code source.

- **La compilation** : En C++, un changement même mineur dans le code demande souvent un nombre de recompilations significatif. Compiler et construire un système aussi grand que CATIA demande un temps très important (plusieurs jours). Dans un tel contexte, il est nécessaire de limiter au strict minimum les dépendances de compilations. Un processus optimisé ainsi que des mécanismes associés sont nécessaires pour limiter l'impact des modifications sur le logiciel.

- **Travail collaboratif** : Le seul moyen offert par C++ pour travailler sur une même classe est de partager son fichier associé. Même avec des outils de gestion de configuration, cela pose des problèmes dans un contexte de fort développement concurrent (notamment lors de la fusion). Il est préférable de

posséder une notion de composant qui peut être définie par plusieurs fichiers. Cela permet à différents développeurs de travailler sur un même composant avec des fichiers distincts.

Un constat évident s'imposait : C++, seul, ne répond pas aux besoins de Dassault Systèmes. Les ingénieurs de Dassault Systèmes ont donc développé une couche additionnelle à C++ qu'ils ont nommée l'Object Modeler. Cette surcouche contraint la façon d'utiliser C++ et ajoute des mécanismes liés à la notion de composant. Elle correspond au modèle de composants de Dassault Systèmes qui est une partie importante de l'architecture V5.

Une question légitime que nous pouvons nous poser est de savoir pourquoi Dassault Systèmes a créé son propre modèle de composants plutôt que de réutiliser un standard comme COM [Box98, COM], CORBA [Cor] ou EJB [Mon00, EJB]. La réponse contient les éléments suivants :

- **Pour des raisons historiques** : lorsque le besoin de posséder une architecture spécifique s'est fait sentir (dans le milieu des années 90), il n'existait aucune technologie qui permettait de répondre aux exigences de Dassault Systèmes. Ainsi, Dassault Systèmes est l'un des pionniers dans ce domaine. Dassault Systèmes n'a pas seulement défini et créé un modèle de composants mais il l'a aussi utilisé avec succès dans un large développement et sur plusieurs années.

- **Pour des raisons de fonctionnalités** : les modèles de composants existants ne répondent pas entièrement aux besoins de Dassault Systèmes. Entre autres, ils ne supportent pas ou peu de mécanismes pour l'extensibilité qui est considérée comme crucial dans le contexte de CATIA.

- **Du point de vue technique** : l'application CATIA V5 doit pouvoir s'exécuter aussi bien sur les plate-formes Unix et Windows. Actuellement, à part le modèle EJB qui ne satisfait pas les deux critères précédents, il n'existe pas de tel modèle de composants compatible avec des exigences industrielles.

- **Au niveau stratégique** : Dassault Systèmes ne souhaite aucunement dépendre d'une autre société pour l'évolution et la maintenance de son modèle de composants, car son utilisation est vitale pour son processus de développement et pour son futur.

3.4 En résumé

Nous venons de décrire les principaux besoins de Dassault Systèmes liés au développement de CATIA V5. Ces besoins ont été classifiés en deux groupes :

- **Les besoins internes** qui résultent du processus de développement de Dassault Systèmes.

- **Les besoins externes** qui sont motivés par les exigences de ses clients.

Ces besoins ont amené Dassault Systèmes à développer une architecture particulière, nommée l'architecture V5, qui sera décrite en détail dans la partie suivante. C'est au fur et à mesure de l'étude de cette architecture V5 et au fil des discussions avec les ingénieurs de Dassault Systèmes que nous avons pu retrouver ces besoins initiaux. Ce travail d'ingénierie inverse sur ces besoins était nécessaire pour comprendre l'intérêt de l'architecture V5 et nous a aidé lors de la conceptualisation de cette architecture.

L'utilisation de cette architecture V5 pour le développement de CATIA V5 a engendré d'autres types de besoins pour la gestion (conception, maintenance et évolution) du code source. Nous présenterons ces nouveaux besoins dans la troisième partie.

Chapitre 4

L'Architecture Logicielle

Notre travail a consisté à proposer des solutions pour améliorer le processus de développement en utilisant une approche basée sur l'architecture logicielle. Avant d'exposer nos travaux, il est nécessaire de présenter ce que nous entendons par le terme architecture logicielle.

Qu'est ce que l'architecture logicielle ? Cette question fait l'objet d'un vaste débat qui est loin d'être terminé. Bien que de nombreuses avancées aient été réalisées dans ce domaine, la communauté scientifique ne se targue pas d'avoir atteint une maturité suffisante. Ce domaine est encore jeune et de nombreuses pistes restent à explorer. De plus, comme le souligne Bosch, il y a une différence entre la perception académique et la pratique industrielle [Bos00].

Néanmoins, la communauté scientifique (principalement académique) a fourni de nombreuses propositions intéressantes pour l'étude de l'architecture logicielle. Nous commencerons donc par regarder ces propositions (cf. section 4.1). Ensuite nous donnerons notre point de vue et décrirons ce qui nous a été utile pour nos recherches (cf. section 4.2).

4.1 L'architecture logicielle vue par la communauté scientifique

Actuellement, il n'existe pas de définition de l'architecture logicielle qui soit acceptée par tous. Bien que de nombreuses définitions aient été proposées (cf. la page web dédiée à ce sujet du Software Engineering Institute [SEI]), aucune ne s'est vraiment imposée. Cependant les chercheurs sont d'accord pour dire qu'une architecture logicielle décrit les structures de haut niveau d'un système. Il est possible de classifier ces définitions selon deux grandes catégories :

- La première correspond à une vue de l'architecture logicielle qui comprend les structures architecturales du système et des aspects extérieurs à ces structures comme les motivations sur le choix de l'architecture adoptée plutôt qu'une autre (la rapidité du système, le coût de production...) ou encore les besoins

des différents acteurs (client, concepteur, programmeur...) [PW92, GACB95, GP95, SG96, EHP+96].

- La seconde a une vue plus restreinte et ne prend en compte que les structures architecturales du système. Les aspects extérieurs ne sont pas considérés comme faisant partie intégrante de l'architecture logicielle du système [GS93, Jon94, BCK98].

Nous allons maintenant parcourir les principales références de ce domaine en osant une critique de celles-ci par rapport à notre expérience. Pour refléter l'évolution des propositions, nous présenterons ces références dans l'ordre chronologique de publication.

4.1.1 Perry et Wolf 1992 [PW92]

"Software Architecture = {Elements, Form, Rationale}" [PW92]

Historiquement parlant, cet article de Perry et Wolf est important car il est considéré comme l'article fondateur de l'architecture logicielle en tant que domaine à part entière du génie logiciel. Il fournit les bases pour l'étude de l'architecture logicielle.

Perry et Wolf se sont basés sur le domaine de l'architecture des bâtiments pour définir leur vision de l'architecture logicielle. Par analogie à l'architecture des bâtiments, ils ont proposé les concepts suivants :

- **Un modèle pour l'architecture logicielle** qui est défini par un triplet :

  Software Architecture = {Elements, Form, Rationale}

  Les éléments architecturaux peuvent être de trois natures : les éléments de données, les éléments de transformation des données et les éléments de connexion entre les éléments précédents. **La forme architecturale** correspond à la fois à des propriétés et à des relations. Les propriétés définissent des contraintes sur les éléments pris individuellement. Les relations définissent des contraintes sur la disposition des éléments entre eux. Ces contraintes sont quantifiées par des poids d'importance qui permettent de distinguer ce qui est important de ce qui ne l'est pas. **Les critères de choix** capturent les motivations de l'architecte pour le choix d'une architecture plutôt qu'une autre. Ces motivations peuvent être de nature économique (temps...), technique (performance...), liées à la conception (évolution...)...

- **La notion de style architectural** qui dénote une représentation abstraite regroupant des caractéristiques et des décisions de conception communes pour des architectures similaires. Les styles architecturaux ne sont pas des architectures à part entière mais capitalisent des aspects clés qui définissent un groupe d'architectures.

- **Des vues architecturales** qui permettent de représenter différents aspects d'une même architecture. Ces vues sont représentées de manière séparée mais sont interdépendantes. Les auteurs proposent trois vues qui correspondent aux trois types d'éléments du modèle : la vue des données, la vue de transformation et la vue de connexion.
- **L'analyse de l'architecture logicielle**. En plus de fournir une documentation précise et claire, les spécifications ont pour objectif premier de fournir des analyses automatiques sur ces documents afin de faire apparaître divers problèmes. Les auteurs listent deux catégories d'analyse : les analyses de cohésion et les analyses de couplage.

Malgré le fait que cet article soit le premier à être consacré entièrement à l'architecture logicielle, les auteurs ont su proposer de bonnes fondations. En ce qui nous concerne, nous avons utilisé plusieurs concepts qui sont énoncés dans cet article.

Nous adhérons au concept de vues multiples d'une même architecture. Ce concept a l'avantage de pouvoir traiter séparemment différents besoins correspondant à divers acteurs. Par contre nous n'avons pas retenu les mêmes vues proposées par les auteurs. Nous pousserons cette idée même un peu plus loin en montrant qu'en plus de la notion de vues multiples, il existe la notion de structures multiples pour l'architecture V5. Nous verrons ensuite que ces vues multiples peuvent être structurées par ce que la norme ANSI/IEEE Std 1471-2000 appelle des points de vue [Gro00].

Nous pensons, comme les auteurs l'indiquent, que les descriptions architecturales ne servent pas uniquement pour faire de la documentation mais qu'il est intéressant de les exploiter pour faire des analyses automatiques. Notre vision est de disposer d'un environnement architectural qui est intégré au processus de développement. Ceci sera décrit dans la partie III de cette thèse.

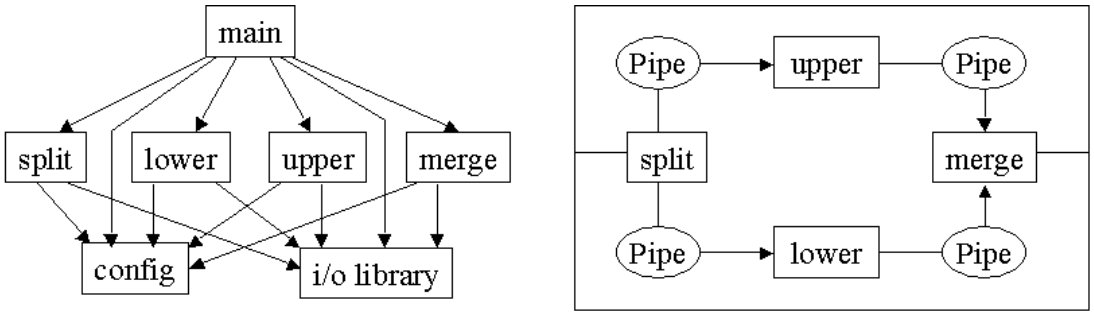
Notre modèle conceptuel de l'architecture logicielle n'est pas formalisé comme le triplet proposé mais nous pouvons retrouver certaines idées. Nos divergences proviennent essentiellement de deux points. Le premier est que les auteurs ont une vue orientée conception initiale alors que dans notre cas l'architecture V5 existait déjà. Le deuxième est lié au fait que nous prenons en compte plusieurs structures qui ont chacune leurs propres éléments architecturaux.

4.1.2 Garlan and Perry 1995 [GP95]

> *"The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time."* [GP95]

Cette définition est tirée de l'éditorial du numéro spécial consacré à l'architecture logicielle du journal IEEE Transactions on Software Engineering. C'est le premier numéro spécial qui est apparu sur ce domaine. Il fournissait à l'époque une bonne synthèse des différents travaux et tendances sur ce domaine.

Cette définition a été formulée en 1994 par un groupe de discussion au Software Engineering Institute de l'université de Carnegie Mellon. Garlan and Perry pensaient que c'était un bon point de départ et étaient conscients que cette définition ne traite qu'une partie de l'architecture logicielle. Ils savaient qu'ils manquaient encore de recul pour bien comprendre ce qu'est l'architecture logicielle. Ils ont donc préféré présenter leur vision de l'architecture logicielle en se basant sur les travaux existants. Nous nous attarderons donc plutôt sur la vision qu'ils proposent que sur cette définition.



(a) Une description définition/utilisation (b) Une description architecturale

FIG. 4.1 – Description définition/utilisation vs Description architecturale

Il est intéressant de reprendre quelques spécificités de l'architecture logicielle décrites par les auteurs :

- **Axée sur l'organisation des structures de haut niveau du système** : lors du développement et de la maintenance d'un gros système, de nouveaux centres d'intérêt apparaissent et deviennent plus essentiels que le choix d'une structure de données ou d'un algorithme. Tandis que les méthodes traditionnelles se sont essentiellement portées sur ces problèmes de structures de données et d'algorithmique, l'architecture logicielle tente de fournir un cadre formel, des notations et des outils pour l'étude de l'organisation des structures de haut niveau d'un système. Nous reviendrons sur ce point qui est bien illustré dans le livre de Shaw et Garlan (cf. section 4.1.4).

- **Des descriptions à base de composants et connecteurs** : la description de la décomposition d'un logiciel en petites parties a longtemps été basée sur les MILs (Module Interconnection Languages) [PDN86] et IDLs (Interface Definition Languages) [Mor87] en termes de définition/utilisation de services (cf. figure 4.1(a)). Dans ce modèle, chaque module définit un ensemble de services qui sont disponibles pour les autres modules et utilise des services fournis par les autres modules. Bien que très utile, ce modèle a l'inconvénient de ne pas faire la distinction entre *implémentation* et *interaction* entre les modules. L'*implémentation* décrit comment un module est construit à partir des services des autres modules alors que l'*interaction* exprime les relations architecturales comme les protocoles de la communication entre les modules. Une description

architecturale, quant à elle, modularise un système sous forme d'une configuration de composants et de connecteurs (cf. figure 4.1(b)). Ce nouveau modèle se focalise sur l'interaction en identifiant des entités de connexion appelées connecteurs et en décrivant de manière précise les protocoles de communications entre les composants. L'article de Allen et Garlan [AG94], intitulé *Beyond Definition/Use : Architectural Interconnection*, explique très bien cette distinction.

- **Méthodes de conception vs Architecture Logicielle** : bien que les méthodes de conception et l'architecture logicielle tentent de réduire la distance entre les besoins et l'implémentation, elles diffèrent sur la manière et suivent une certaine évolution (cf. figure 4.2). Sans l'utilisation de méthodes de conception ou de la discipline de l'architecture logicielle, l'implémentation est laissée totalement au gré du développeur (cf. figure 4.2 (a)). Les méthodes de conception améliorent cette situation en fournissant des liens clairs entre certains besoins et leur implémentation (cf. figure 4.2 (b)). L'architecture logicielle permet de gagner un degré de liberté en s'abstrayant au mieux de l'implémentation. Elle fournit un cadre de développement pour l'implémentation et est concernée par les compromis entre différents choix architecturaux pour leurs capacités à résoudre certains types de problèmes (cf. figure 4.2 (c)).

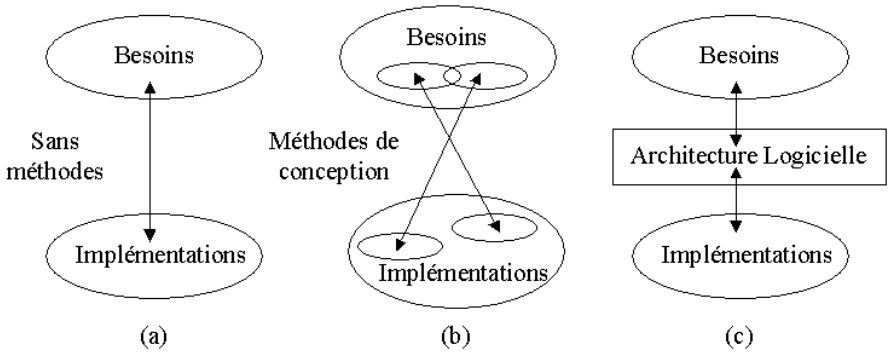


FIG. 4.2 – Méthodes de conception et architecture logicielle

Nous trouvons que ces distinctions résument assez bien le fait que l'architecture logicielle ne se place pas au même niveau que l'algorithmique et les structures de données et tente de résoudre des problèmes qui lui sont propres. Par contre, nous pensons que les auteurs ont restreint la portée des méthodes de conceptions. Nous estimons que les méthodes de conceptions, tout comme l'architecture logicielle, essaient à la fois de cloisonner les besoins et de garder leur trace dans l'implémentation. La différence provient plutôt du niveau d'abstraction et des types de besoins qui ne sont pas les mêmes. Au niveau de l'architecture logicielle, il reste encore à bien identifier ce que sont ces besoins et à fournir des méthodes pour les résoudre.

4.1.3 Gacek, Abd-Allah, Clark and Boehm 1995 [GACB95]

"A software system architecture comprises :

- *A collection of software and system components, connections and constraints.*
- *A collection of system stakeholders' need statements.*
- *A rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements.*

" [GACB95]

L'idée principale de leur article est qu'ils considèrent que les définitions précédentes ([PW92, GS93, Jon94]) se focalisent essentiellement sur les représentations architecturales. Ils critiquent le fait que ces définitions ne sont pas assez précises pour déterminer si un diagramme donné est ou n'est pas une architecture. Pour cela, ils préconisent de prendre en compte les critères de choix d'une architecture au même titre que la structure d'une architecture. Ces critères de choix doivent assurer que les composants, connexions et contraintes définissent un système qui satisfait les besoins des différents acteurs.

Nous sommes d'accord sur le fait que les besoins des différents acteurs et les critères de choix doivent être pris en compte par une approche architecturale. C'est d'ailleurs pour cette raison que nous avons rencontré un grand nombre d'acteurs de Dassault Systèmes, ayant des responsabilités différentes, pour étudier l'architecture V5. Par contre, nous sommes plutôt de l'avis de Bass et al. [BCK98] pour dire que ces aspects (besoins des acteurs et critères de choix) ne sont pas une partie intégrante de l'architecture d'un système (cf. section 4.1.5). En effet, si nous récupérons le code source d'une application, il est plus ou moins possible de retrouver sa (ses) structure(s) architecturale(s) mais ce n'est pas le cas pour ces aspects (besoins des acteurs et critères de choix). Le fait de disposer ou non de ces informations (besoins des acteurs et critères de choix) ne nous empêche pas de modifier l'architecture du logiciel.

Les auteurs présentent dans cet article une vision de l'architecture logicielle centrée sur le processus du cycle de vie. Ils proposent que l'architecture logicielle puisse servir de référence tout au long du cycle de vie (cf. figure 4.3). Pour cela elle doit pouvoir satisfaire l'ensemble des besoins des différents acteurs. Il est donc nécessaire que les descriptions architecturales fournissent des vues multiples pour ces divers besoins.

Nous retrouvons ici deux idées auxquelles nous adhérons. La première est le lien entre l'architecture d'un système et le cycle de vie. Cette idée est très intéressante et n'a pas été à notre avis suffisamment exploitée. Nous ne l'avons pas retrouvée dans d'autres travaux. Nous pensons même que l'architecture d'un système pourrait

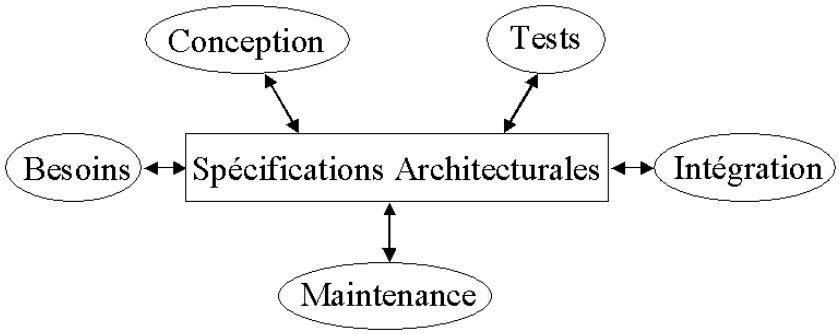


FIG. 4.3 – Description architecturale et cycle de vie

évoluer tout au long du cycle de vie et ne pas rester statique (cf. section 9.6). Ceci permettrait entre autres de faciliter la gestion de l'évolution au niveau de la conception sans pour autant affecter les performances du logiciel à l'exécution. Une de nos propositions est aussi de pouvoir disposer d'un environnement dédié à l'architecture tout au long du processus de développement (cf. chapitre 8).

La deuxième idée est le besoin de multi-vues pour répondre aux besoins des différents acteurs. C'est aussi l'approche que nous avons adoptée et nos prototypes ont été développés dans le but de fournir des vues multiples.

4.1.4 Shaw and Garlan 1996 [SG96]

> *"The architecture of a software system defines that system in terms of computational components and interactions among those components. Components are such things as clients and servers, databases, filters, and layers in a hierarchical system. Interactions among components at this level of design can be simple and familiar, such as procedure call and shared variable access. But they can also be complex and semantically rich, such as client-server protocols, database-accessing protocols, asynchronous event multicast, and piped streams.*
> *In addition to specifying the structure and topology of the system, the architecture shows the correspondence between the system requirements and elements of the constructed system, thereby providing some rationale for the design decisions. At the architecture level, revelant system-level issues typically include properties such as capacity, throughput, consistency, and component compatibility"* [SG96]

Cette définition est tirée du livre de Shaw et Garlan qui fut le premier livre consacré à l'architecture logicielle. Ce livre correspond à une vision très académique et fournit une bonne synthèse des travaux qui ont été publiés jusqu'alors.

Les auteurs définissent l'architecture logicielle de manière assez simple comme un ensemble de composants, de connecteurs et une configuration de ceux-ci guidée par des critères de choix. Nous pouvons constater que cette définition n'apporte pas vraiment d'éléments nouveaux par rapport aux précédentes. Les auteurs se sont plutôt attelés à donner une vision plus formelle de la pratique de l'architecture

logicielle. Cette description est néanmoins limitée par les connaissances de l'époque. Le domaine de l'architecture logicielle était à ses débuts et laissait place à de nombreuses recherches. Par exemple, le livre n'indique pas la notion de structures multiples d'un système. De même, les besoins des différents acteurs et l'influence réciproque de l'organisation d'une société sur l'architecture ne sont pas mentionnés.

Par contre, nous estimons que les auteurs ont bien su identifier sur quoi porte l'étude de l'architecture logicielle.

> *"As the size and complexity of software systems increase, the design and specification of overall system structure become more significant issues than the choice of algorithms and data structures of computation. Structural issues include the organization of a system as a composition of components; global control structure; the protocols for communication, synchronization, and data access; the assignment of functionality to design elements; the composition of design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives. This is the software architecture level of design."* [SG96]

Nous pensons effectivement que l'architecture logicielle a pour objectif de répondre à de nouveaux problèmes qui ne sont pas liés aux algorithmes et aux structures de données. Elle correspond à l'étude de nouvelles structures définies au dessus du code. Ces structures sont importantes car elles jouent un rôle décisif dans la réussite ou non d'un gros projet. Si la modification d'un algorithme ou d'une structure de données n'a pas trop d'impact sur le développement de l'application, ce n'est généralement pas le cas pour ces structures. Elles forment en quelque sorte les fondations du code source. Ces structures sont en général peu significatives sur les petits programmes et tendent à le devenir de plus en plus au fur et à mesure de l'importance de la taille et de la complexité du logiciel.

Notre contexte était donc tout approprié puisque CATIA V5 fait près de 5 millions de lignes de code. Nous avons ainsi pu clairement identifier un certain nombre de structures pour l'architecture V5. Ces structures seront détaillées dans la partie suivante.

4.1.5 Bass, Clements et Kazman 1998 [BCK98]

> *"The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them"* [BCK98]

Ce livre fut le deuxième, après celui de Garlan et Shaw, consacré à l'architecture logicielle. Il a la caractéristique de ne pas se placer principalement dans un contexte académique mais plutôt dans une pratique industrielle.

Nous allons prendre le temps de décortiquer cette définition car de nombreuses idées sont sous-entendues et elle amène à se poser de bonnes questions. Cette définition est celle qui se rapproche le plus de la nôtre.

Les auteurs indiquent que l'architecture logicielle n'est qu'une abstraction de la réalité qui se focalise plus particulièrement sur l'interaction entre les composants. Nous sommes d'accord avec eux sur le fait qu'une description architecturale est une abstraction, sinon il suffirait de prendre le système en entier (comme ils le soulignent). Par contre, nous différons sur la nature de cette abstraction. Ils se positionnent au niveau de l'interaction entre les composants. De notre côté, nous préférons laisser le libre arbitre du niveau d'abstraction selon les besoins. Par exemple, dans notre travail nous avons eu besoin de prendre en compte les aspects architecturaux pour la création des composants afin de réaliser certaines analyses.

Les auteurs insistent sur le fait qu'un système peut comprendre plusieurs structures et qu'aucune d'entre elles ne peut être considérée comme celle définissant l'architecture de ce système. De plus, ils notent que de la même manière qu'il existe plusieurs structures, il existe aussi plusieurs types de composants (modules, processus), plusieurs types d'interactions entre les composants (subdivision, synchronisation) et différents contextes (développement, exécution). Nous partageons ce point de vue qui correspond à notre façon de modéliser l'architecture V5.

Il est important de noter que les notions de multi-structures et de multi-vues sont différentes. Une vue est liée à une structure et il est possible de réaliser de multiples vues pour une même structure. Les vues multiples permettent de montrer sous des angles différents une même structure. Une structure par contre correspond à une configuration particulière de différents éléments. Cette distinction est importante car beaucoup de travaux prennent en compte les vues multiples mais peu proposent des structures multiples. Les recherches en architecture logicielle se sont principalement portées sur la structure fonctionnelle (à travers la spécification du comportement et de l'interaction entre les composants fonctionnels) qui est présente dans la majorité des systèmes. D'autres structures (build...) ne prennent un sens qu'à partir d'une certaine taille des applications. Nous verrons que dans le cadre de l'architecture de CATIA V5, nous avons pu identifier plusieurs structures et plusieurs vues sur ces structures.

Est-ce que tous les systèmes ont une architecture ? Les auteurs pensent que oui puisque chaque système peut être vu comme une décomposition de composants intéragissant entre eux. Dans le cas le plus trivial, le système peut n'être qu'un composant. En ce qui nous concerne, nous pensons plutôt le contraire. Nous ne considérons pas, par exemple, qu'un composant isolé définit une structure et donc une architecture.

Les auteurs font remarquer que l'architecture qui est associée au système peut ne pas être connue (les architectes initiaux sont partis, nous ne possédons que le code source...). Cette observation est pertinente et révèle deux points importants :

- Le premier est la différence entre l'architecture concrète d'un système et une description architecturale de cette dernière. L'architecture concrète d'un système correspond à l'architecture définie dans le code source. Elle correspond à la réalité. Une description architecturale est une abstraction du code

source qui tente de décrire au mieux cette architecture concrète. Elle sert pour la conception, la compréhension et les analyses architecturales.

- Le deuxième est le fait qu'une architecture existe au sein du code indépendamment de ses descriptions associées. Par conséquence, une description architecturale peut ne pas ou ne plus être en conformité par rapport à l'architecture qu'elle décrit. Cette remarque est essentielle et a de fortes implications dans un contexte industriel comme le nôtre. Elle souligne une contrainte forte : les descriptions architecturales doivent toujours être cohérentes par rapport au code pour être utiles. Cette contrainte nous a amenés à utiliser une approche montante en réalisant nos descriptions architecturales par ingénierie inverse directement à partir du code source. CATIA V5 évoluant très rapidement (il sort une release tous les quatre mois) il n'est pas envisageable de maintenir manuellement les descriptions architecturales correspondantes.

Les auteurs notent que le comportement de chaque composant fait partie de l'architecture. Le comportement d'un composant permet de définir comment un autre composant peut interagir avec lui. Ils jugent ainsi que les diagrammes informels sous forme de boîtes et de lignes ne sont pas des architectures puisqu'ils ne décrivent pas concrètement le comportement des composants.

Nous trouvons que sur ce point les auteurs n'ont pas été suffisamment précis. Faut-il nécessairement utiliser des langages formels pour la description du comportement (comme le langage de description d'architecture Wright [AG97] qui utilise CSP [Hoa78]) ? Est-ce que la description des interfaces que propose un composant est suffisante ? Les auteurs ne le précisent pas et ils utilisent des diagrammes informels annotés pour leurs exemples dans leur livre. Nous pensons effectivement qu'un des grands apports de l'architecture logicielle a été de remonter au premier plan les aspects de comportement et d'interactions entre les composants. Par contre la précision de la description de ces aspects varie suivant les besoins comme nous le préciserons plus tard. Dans notre contexte, nous devions étudier les analyses d'impact au niveau architectural (cf. section 7.3) et la description des liens de dépendance en guise de comportement a été suffisante.

Les auteurs signalent très justement qu'une architecture peut très bien être bonne ou mauvaise sans pour autant empêcher le système de fonctionner. Bien entendu, il vaut tout de même mieux que l'architecture réponde au mieux aux besoins des concepteurs et acteurs. Il est donc intéressant de pouvoir évaluer une architecture par rapport à ces besoins. Les auteurs ont une grande expérience dans ce domaine puisqu'ils ont élaboré deux méthodes pour analyser une architecture logicielle. La première, appelée SAAM (Software Architecture Analysis Method), utilise des scénarios correspondants aux attentes des acteurs comme un banc d'essai pour comparer les différentes architectures candidates [KABC96]. La deuxième, appelée ATAM (Architecture Tradeoff Analysis Method) qui est une suite de la première méthode, permet de prendre en compte plusieurs attributs de qualité et a pour objectif de trouver et analyser les compromis dans une architecture logicielle [KKB+98, KBK+99]. De notre côté, nous ne nous sommes pas intéressés à cet aspect car ces méthodes sont orientées vers la création d'une architecture alors que

dans notre cas l'architecture V5 existait déjà. Par contre, ceci nous conforte dans l'idée d'un environnement dédié à l'architecture logicielle tout au long du cycle de vie.

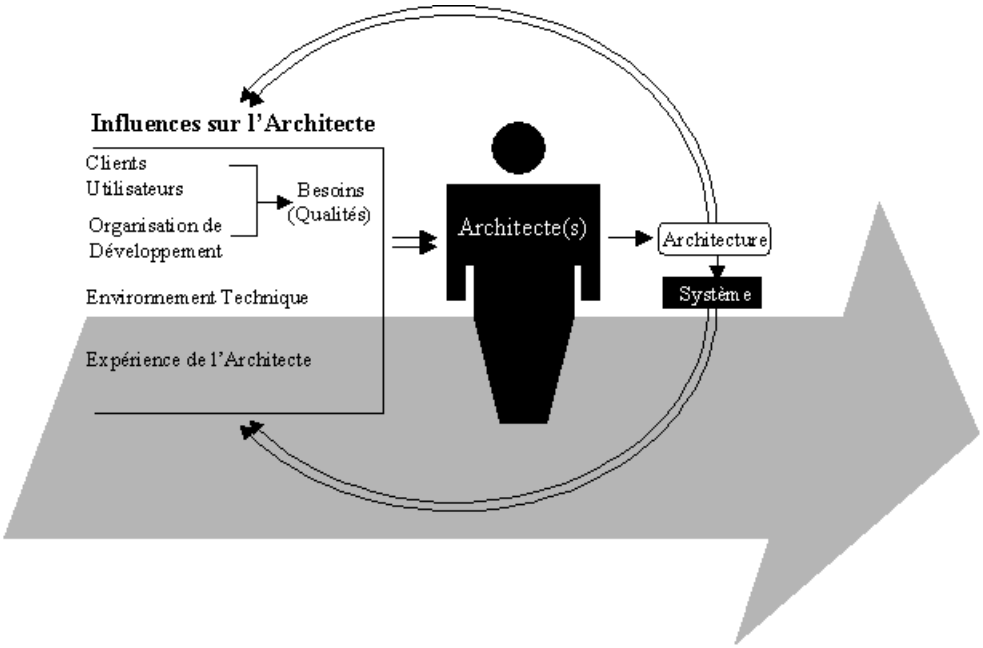


FIG. 4.4 – Architecture Business Cycle

Si leur définition se restreint à des aspects structurels, leur vision de l'architecture logicielle est beaucoup plus ample. Leur livre véhicule beaucoup d'autres idées intéressantes comme leur "Architecture Business Cycle" (cf. figure 4.4). Ce cycle décrit le fait que pour réaliser une architecture, l'architecte est influencé par différents facteurs (les acteurs, l'organisation de développement, son expérience...). De même l'architecture peut à son tour influencer ces différents facteurs, ce qui crée un cycle. Comme nous le décrirons dans la partie suivante, nous avons pu identifier ce cycle dans notre contexte avec Dassault Systèmes. De plus, nous avons pu prendre conscience de l'importance de ces facteurs pour l'étude de l'architecture V5.

Les auteurs n'intègrent pas ces idées directement dans leur définition car ils considèrent que la structure d'un système peut être retrouvée à partir du code mais pas ces facteurs. Ce qui est tout à fait exact. Les besoins des acteurs et les motivations du choix d'une architecture sont des aspects importants à prendre en compte mais ils ne font pas partie de l'architecture d'un système. Ils considèrent à juste titre que cela fait plutôt partie du processus qui accompagne l'architecture du système. Il est possible de faire l'analogie avec l'architecture des bâtiments pour illustrer ce point. Par exemple, nous pouvons voir la structure d'une maison sans pour autant connaître les motivations sous-jacentes.

Nous pensons que ce livre est une bonne référence pour l'étude de l'architecture logicielle. Nous avons retrouvé et utilisé plusieurs idées décrites dans ce livre lors de notre étude de l'architecture V5.

4.1.6 Garlan 2000 [Gar00]

> *"While there are numerous definitions of software architecture, at the core of all of them is the notion that the architecture of a system describes its gross structure. This structure illuminates the top level design decisions, including things such as how the system is composed of interacting parts, where are the main pathways of interaction, and what are the key properties of the parts. Additionally, an architectural description includes sufficient information to allow high-level analysis and critical appraisal".* [Gar00]

A l'occasion du passage au troisième millénaire, une initiative à été prise pour faire le point sur l'ensemble des domaines du génie logiciel. Cette entreprise a abouti à la publication d'un livre, intitulé "le futur du génie logiciel" [Fin00], où des chercheurs influents ont écrit un article correspondant à une vue d'ensemble et au futur de leur domaine. Cette tâche est revenue à David Garlan pour l'architecture logicielle, qui est considéré comme le représentant de cette discipline dans la communauté scientifique.

Comme nous pouvons le remarquer, il n'a pas souhaité donner une nouvelle définition mais a préféré faire un constat. L'ensemble des définitions qui ont été formulées ont au centre de chacune la notion que l'architecture d'un système décrit les éléments essentiels de sa structure. C'est effectivement une bonne synthèse[1] qui correspond à une vue globale de l'architecture logicielle. Il est intéressant de noter que Garlan a pris la précaution d'écrire que cette structure met en évidence des décisions de conception du plus haut niveau. Il signifie ainsi que ces décisions de conception ne font pas partie de cette structure. Ceci rejoint le point de vue de Bass et al. qui est aussi le nôtre. Enfin, nous retrouvons l'idée qu'une description architecturale sert entre autres à réaliser des analyses et des estimations de haut niveau.

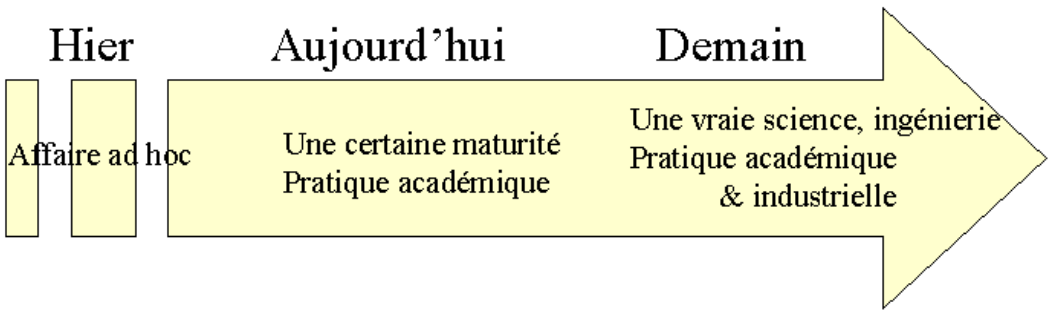


FIG. 4.5 – Evolution de la pratique architecturale

[1] A part éventuellement le fait qu'il ne mentionne pas qu'une architecture peut avoir plusieurs structures.

Dans cet article Garlan fournit une très bonne vision de l'évolution de l'architecture logicielle. Son discours est structuré de manière chronologique et peut être résumé par la figure 4.5 :

- **Hier** : L'architecture logicielle était une affaire ad hoc qui ne répondait pas à une science précise. Les descriptions étaient basées sur des diagrammes informels (boîtes et lignes) qui étaient rarement maintenus une fois le système construit. Les choix architecturaux ne suivaient pas spécialement de méthode. Typiquement, on adaptait une conception antérieure même si elle n'était pas forcément appropriée. La connaissance d'une bonne pratique architecturale ne pouvait se faire que sur ses propres expériences et ne pouvait pas être transmise à d'autres.
- **Aujourd'hui** : Beaucoup de choses ont évolué. Même si nous ne pouvons pas dire que ce domaine est complètement structuré, il est considéré comme une activité de conception explicite dans le processus de développement. Les sociétés ont pris conscience qu'une architecture est un facteur critique dans la réussite d'un développement. De nombreux travaux ont permis de formaliser la pratique architecturale et de disposer d'un vocabulaire commun. L'architecture logicielle commence à être enseignée dans les universités.
- **Demain** : Bien que la connaissance de l'architecture logicielle soit beaucoup plus mature, elle n'a pas encore atteint un niveau suffisant. Elle est encore peu utilisée à travers l'industrie du logiciel. Mais cela commence à venir petit à petit et il faut du temps pour qu'une nouvelle approche se propage. Nous pouvons espérer que les connaissances vont continuer à évoluer pour faire place à une vraie ingénierie de ce domaine. Cependant, il faut s'attendre à de nouveaux défis. La rapide évolution et utilisation d'internet et des systèmes embarqués font place à de nouvelles techniques de conception et de programmation. Cela a un impact non négligeable sur la pratique architecturale.

4.1.7 La norme ANSI/IEEE Std 1471-2000 [Gro00]

> *"architecture : The fundamental organization of a system embodied in its components, their relationships to each other, and the environment, and the principles guiding its design and evolution"* [Gro00]

Cette norme IEEE 1471 a pour objectif de fournir des recommandations pour la description architecturale des gros systèmes sous forme d'un standard. Elle a été définie par un groupe de travail sur l'architecture qui a été constitué par l'IEEE (Institute of Electrical and Electronics Engineers). Leur définition n'apporte rien de nouveau par rapport aux précédentes. Par contre, il est plus intéressant de se tourner vers leur modèle conceptuel pour la description architecturale (cf. figure 4.6[2]).

Un système est lié à un environnement qui l'influence. Il faut rajouter, comme le soulignent Bass et al., que le système peut aussi influencer cet environnement.

[2] Ce schéma est en anglais car il est emprunté à la norme.

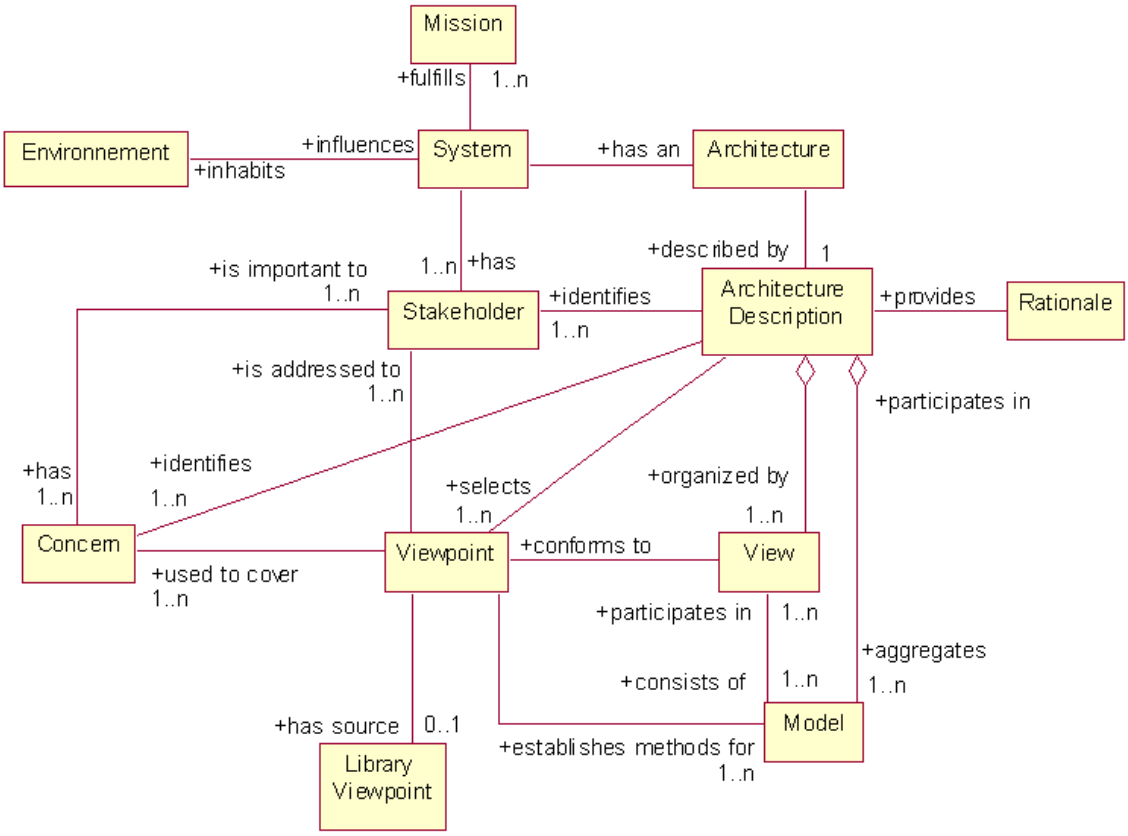


FIG. 4.6 – Modèle conceptuel pour une description architecturale

Nous avons pu noter cette influence mutuelle lors de notre étude de l'architecture V5 à Dassault Systèmes. Notre travail ne s'étant pas focalisé sur ce point, nous fournirons quelques explications concrètes sans pour autant être exhaustif dans la partie suivante.

Un système doit remplir un certain nombre de missions dans son environnement. Par mission, les auteurs entendent une fonctionnalité du système qui est destinée à un ou plusieurs acteurs et qui doit répondre à un ensemble d'objectifs. De ce côté, nous avons regardé principalement les objectifs auxquels devait répondre l'architecture V5 au niveau du génie logiciel et très peu au niveau de la CAO. Ces objectifs au niveau génie logiciel ont été abordés dans le chapitre précédent.

La position de cette norme est la même que celle de Bass et al. concernant la relation entre un système et une architecture. Pour les auteurs, chaque système a une architecture dans le sens de cette recommandation. Comme nous le décrirons plus tard, notre position est plus mitigée.

Cette architecture peut être représentée par une description architecturale. Les auteurs ont eu la bonne idée de séparer l'architecture de sa description architecturale. Par contre, nous pouvons remarquer une erreur dans le modèle conceptuel. En effet, ce modèle indique qu'une architecture ne peut être représentée que par une seule description architecturale. Ce qui est inexact simplement par le fait que si nous demandons à plusieurs architectes de faire une description architecturale d'un même système, il y a de fortes chances qu'il y ait autant de réponses différentes que d'architectes. Nous nous sommes fait confirmer cette erreur par les auteurs qui pensent effectivement qu'il peut exister plusieurs descriptions architecturales pour un même système. En fait, ils placent la description architecturale au centre de leur modèle et ils voulaient plutôt dire qu'une description architecturale décrit une architecture. Dans la partie II, nous présenterons en détail notre description de l'architecture V5.

Pour réaliser un système, différents acteurs (clients, architectes, chef de projets...) sont impliqués. Nous avons pu en répertorier un certain nombre dans l'organisation de Dassault Systèmes (cf. section 6.2). Dans le modèle conceptuel, la description architecturale doit indiquer l'ensemble des acteurs qui sont concernés par la description. Chacun de ces acteurs ont des tâches et donc des intérêts différents par rapport au système et à son architecture. Ces intérêts doivent donc se retrouver dans la description architecturale pour qu'elle puisse être utile. C'est ce que nous avons tenté de faire. Le niveau et le détail de nos descriptions architecturales ont toujours été motivés par rapport aux besoins des acteurs de Dassault Systèmes. Le modèle conceptuel utilise le terme identifier ("identifies") entre la description architecturale ("Architectural Description") et les acteurs ("Stakeholder") ainsi qu'entre la description architecturale et les besoins ("Concern"). Il faut prendre ici le terme identifier dans le sens lister, énumérer...

La description architecturale peut contenir plusieurs modèles sous-jacents. Elle peut être représentée par plusieurs vues qui correspondent à un certain nombre

de ces modèles. Dans la norme 1471, chaque vue doit être conforme à un point de vue ("Viewpoint"). Comme ils aiment à le présenter, une vue est à un point de vue ce qu'un programme est à un langage de programmation. Le point de vue peut être considéré comme le meta modèle de plusieurs vues. Chaque point de vue est défini pour couvrir des intérêts correspondant à un certain nombre d'acteurs. L'intérêt des points de vue est qu'ils ne sont pas forcément liés à une architecture et qu'ils peuvent être réutilisés dans d'autres contextes. C'est d'ailleurs la raison de l'existence d'une librairie de points de vue. De bons exemples de point de vue, même si les auteurs les appellent des vues, sont fournis par les travaux de Kruchten [Kru95] et Hofmeister et al. [HNS00]. Dans notre étude, nous avons pu identifier un certain nombre de points de vue pour l'architecture V5. Nous avons eu le temps d'en étudier trois en détail qui seront présentés dans la partie suivante.

Cette norme ne définit rien concernant la cohérence entre les différentes vues ou entre les modèles. En fait, les auteurs en sont conscients et c'est un sujet qui n'est pas simple à standardiser. Ils préfèrent (au moins dans cette première version) laisser ce problème au soin des architectes.

Nous avons pu constater que notre travail rentre bien dans le cadre de ce modèle et que dans une certaine mesure notre description architecturale de CATIA V5 est une instanciation de cette norme. A part quelques points de détails, nous considérons que les auteurs ont bien su identifier les concepts nécessaires pour réaliser une description architecturale et les modéliser. Nous pouvons regretter le fait qu'il n'existe pas d'exemple disponible mais les auteurs y pensent. Il ne manque plus maintenant qu'à éprouver cette norme sur plusieurs cas concrets afin de l'affiner.

4.2 Notre vision de l'architecture logicielle

Comme nous venons de le voir, il n'y a pas vraiment de consensus pour définir ce qu'est exactement l'architecture logicielle. A première vue cela peut paraître déstabilisant. Après tout, comment pouvons-nous travailler sur un concept sans avoir une définition commune ? En fait, en pratique cela n'est pas trop pénalisant. D'abord parce que même s'il n'existe pas de définition commune, les chercheurs sont plus ou moins d'accord sur les grandes lignes. Ensuite, comme le soulignent Bass et al., ceci n'est pas spécifique au domaine de l'architecture et ce manque de consensus ne nous empêche pas de travailler sur des concepts. Pour s'en convaincre, il suffit de prendre pour exemple le domaine de l'orienté objet. Qui est capable de fournir une définition acceptée par tous ? C'est pourtant une discipline qui a plus de vingt ans et de nombreux résultats ont été exploités. La majorité des développements industriels se font maintenant avec une modélisation et un langage orienté objet.

De notre côté, nous avons adopté une approche plutôt pragmatique. Nous n'avons pas cherché à proposer une définition de l'architecture logicielle qui serait acceptée par tous. Nous avons préféré nous attacher à comprendre comment ce domaine pouvait apporter des réponses à certains besoins de Dassault Systèmes.

Pour cela nous avons consacré beaucoup de temps pour comprendre le contexte de Dassault Systèmes ; c'est-à-dire ses acteurs et leurs besoins ainsi que des aspects techniques liés à l'existant de l'architecture V5. Nous avons ensuite adapté des résultats académiques et proposé d'autres solutions adaptées à notre contexte.

Comme nous avons tenté de le montrer dans la section précédente, nous avons utilisé de nombreux aspects proposés par la communauté scientifique pour nos travaux. Notre vision actuelle de l'architecture logicielle est fortement liée à nos travaux avec Dassault Systèmes et nous en sommes conscients. Nous avons essentiellement étudié les aspects statiques et peu les aspects dynamiques de l'architecture de CATIA V5. Il faudrait aussi d'autres collaborations avec d'autres industriels pour affiner notre vision. Néanmoins, elle n'est pas strictement liée à notre contexte et peut donner des éléments de réponse pour d'autres recherches. Nous pouvons la résumer ainsi.

(Pour faciliter la compréhension nous prendrons à plusieurs reprises l'analogie avec l'architecture des bâtiments.)

4.2.1 A quoi tente de répondre l'architecture logicielle ?

L'architecture logicielle cherche à faciliter et à organiser l'implémentation, l'exécution, l'exploitation et la maintenance d'un logiciel afin de répondre aux besoins des différents acteurs. La figure de Garlan et Perry (cf. figure 4.2) montre très bien la position de l'architecture logicielle entre les besoins et les implémentations. Elle ne se focalise pas sur des problèmes liés à l'algorithmique et aux structures de données mais à de nouvelles structures qui sont au dessus du code [GP95, SG96].

Ceci est la même chose pour les bâtiments. L'architecte ne se soucie guère de savoir comment sont fabriqués le béton, la charpente, les fils électriques, la tuyauterie... Il s'attache plutôt à connaître leurs propriétés et comment les disposer les uns pas rapport aux autres afin de réaliser un cadre de vie répondant aux exigences de ses clients.

4.2.2 Qu'est-ce qu'une architecture logicielle ?

La réponse que nous allons tenter de donner correspond à ce que nous comprenons actuellement grâce à ce travail de thèse.

> *Une architecture logicielle d'un système logiciel est un ensemble de structures de haut niveau, interdépendantes, dans lesquelles il existe une organisation intentionnelle. Une structure est un graphe de composants (les nœuds) et de connecteurs (les arcs).*

De cette définition découle le fait que le domaine de l'architecture logicielle a pour objectif l'étude de ces structures haut niveau et leur organisation.

Notre définition est proche de la position de Bass et al. [BCK98] mais diffère sur quelques points que nous allons détailler ci-après. Nous avons en commun de ne retenir que les aspects structuraux. Les autres aspects (acteurs, besoins, critères de choix...) sont des aspects extérieurs nécessaires pour l'ingénierie (création, maintenance, étude...) de l'architecture d'un logiciel. Par contre, ils ne la caractérisent pas.

Prenons le temps de revenir un peu sur cette définition pour expliciter tout ce qu'elle sous-entend.

[...] un système logiciel [...]

Tout d'abord nous avons utilisé le terme système logiciel plutôt que programme ou logiciel pour souligner le fait que nous ne prenons pas uniquement en compte le code source mais l'ensemble des données qui participent à la création du logiciel ainsi que les ressources utilisées par celui-ci. Ces données et ressources peuvent être en plus du code source, des librairies, des fichiers textuels, des machines, un réseau...

[...] un ensemble de structures [...]

Nous indiquons ensuite qu'une architecture est une ensemble de structures. A la différence de nombreux travaux qui ne prennent en compte que la structure fonctionnelle d'un système, nous sommes de l'avis de Bass et al. [BCK98] qu'une architecture logicielle d'un système possède différentes structures architecturales. Le nombre de ces structures n'est pas fixé et dépend du système logiciel.

Comme cela est précisé dans la définition, une structure est un graphe de composants et de connecteurs. Un composant représente une entité qui met à disposition un certain nombre de services. Un connecteur désigne une entité qui permet de mettre en relation ou connecter au moins deux composants. Ces composants et connecteurs peuvent prendre différentes formes suivant le type de la structure considérée. Par exemple, pour la structure fonctionnelle, un composant peut être une ou plusieurs classes implémentant des interfaces et un connecteur peut être un appel de méthode ou une entité plus complexe et plus riche sémantiquement telle qu'un protocole client/serveur. Pour la structure physique, un composant peut être un module ou un package java et un connecteur peut être un lien de dépendance de compilation entre deux modules.

Dans le cas d'un bâtiment, il est possible d'identifier plusieurs structures telles que le gros œuvre (les murs, les cloisons et la toiture), le réseau électrique, le réseau pour l'arrivée et l'évacuation de l'eau, le réseau pour le gaz...

[...] structures de haut niveau [...]

Ces structures sont de haut niveau car l'architecture est une abstraction du système logiciel. L'architecture permet de s'élever du code source pour avoir une

vision globale. Le terme abstraction représente ici deux réalités :

1. **Un filtre** : l'architecture joue comme un filtre car elle isole du code les éléments liés à la structure de ceux qui ne le sont pas. Ce filtre permet de se focaliser uniquement sur les informations utiles pour les acteurs au niveau de la structure. Nous simplifions ainsi la réalité pour pouvoir mieux la maîtriser.
2. **Une conceptualisation** : l'architecture manipule des concepts abstraits qui ne se retrouvent pas de manière localisée dans le code. La notion de connecteur est un bon représentant. Par exemple, l'implémentation d'un protocole de type Client/Serveur est répandue dans différentes classes. Il n'est généralement pas possible de retrouver ce genre d'informations par des outils d'ingénierie inverse automatique.

[...] structures interdépendantes [...]

Ces structures sont interdépendantes car une structure influence et/ou est influencée par plusieurs autres. L'étude de ces interdépendances est importante si nous souhaitons comprendre ce qui se passe et faire des analyses d'impact.

Si nous prenons l'analogie avec le bâtiment, une maison ou un immeuble n'est pas seulement un ensemble de parpaings, tuiles, vitres, fils électriques, joints, ciments... mis n'importe comment. Lors de la construction d'un immeuble, l'architecte, lorsqu'il conçoit le réseau électrique par exemple, est obligé de prendre en compte les autres structures. Il ne peut pas faire passer les fils électriques n'importe où. Il est obligé de prendre en compte les autres éléments présents. Les différentes structures interfèrent les unes avec les autres.

[...] structures dans lesquelles il existe une organisation intentionnelle [...]

Il existe une certaine organisation dans chaque structure. Ce point est important car il nous différencie de la plupart des autres définitions. Nous pensons qu'une architecture correspond à un ensemble de structures dont chacune d'entre elles suit une certaine organisation qui a été souhaitée pour diverses raisons et qui sont connues ou non. Elles peuvent ne plus être connues parce que les concepteurs initiaux ne sont plus là, parce qu'il n'existe pas de documentations... Le fait de connaître l'organisation d'une structure permet de comprendre la fonctionnalité de chaque entité dans la structure. Nous avons alors suffisamment de repères pour comprendre et modifier cette structure.

Cette notion d'organisation est importante car elle révèle le fait que tout système logiciel n'a pas forcément une architecture ou que cette architecture s'est dégradée au point de devenir inexistante ou non pertinente. Le processus de dégradation d'une architecture d'un système est très bien expliqué dans l'article de Parnas [Par94]. L'auteur indique que les modifications d'un logiciel sont souvent effectuées par des personnes qui ne comprennent pas la conception originale, ce qui cause la dégradation de son architecture. Parfois ces dommages ne sont pas importants mais ils sont

la plupart du temps sérieux. Après plusieurs modifications, plus personne (ni les concepteurs, ni les développeurs qui ont effectué les modifications) ne comprend l'architecture résultante.

Bien sûr le fait de savoir si tout système à une architecture ou non dépend grandement de la définition d'une architecture. Cependant certaines personnes affirment qu'une architecture est une propriété intrinsèque d'un système. C'est-à-dire que si un système existe alors son architecture existe. Il est vrai que même si nous ne connaissons pas l'architecture d'un système ou qu'elle ne peut être décrite par personne, cela ne veut pas dire qu'elle n'existe pas. Mais rien ne nous dit non plus qu'elle existe. Nous ne disposons pas actuellement des connaissances nécessaires pour montrer qu'un système a forcément une architecture. C'est plutôt une prise de position. Nous ne pensons pas que l'architecture est une notion qui relate un fait (comme la masse pour un objet).

4.2.3 Architecture Logicielle et Descriptions Architecturales

Il est important de noter que nous n'avons pas parlé de description architecturale dans notre définition. Ceci est volontaire et exprime la différence qui existe entre l'architecture logicielle d'un système et les représentations que nous pouvons en faire. Sur ce point, nous sommes tout à fait d'accord avec Bass et al. et la norme ANSI/IEEE Std 1471-2000. Il existe une architecture concrète liée au système et des descriptions architecturales qui montrent cette architecture sous différents angles suivant les besoins des acteurs. Cette séparation entre cette architecture concrète et ces descriptions architecturales est importante puisqu'un système fonctionne indépendamment de ces descriptions. Il faut donc veiller à ce qu'elles soient cohérentes avec l'architecture pour être utiles.

En continuant l'analogie avec le bâtiment, ces descriptions architecturales correspondent aux différents plans pour un immeuble. De plus pour souligner l'indépendance entre une architecture et ses représentations architecturales, nous pouvons noter que certaines constructions ont été réalisées sans plans mais qu'elles possèdent tout de même une architecture.

Nous reviendrons de manière plus précise sur ce point dans le chapitre suivant.

4.2.4 L'ingénierie pour l'architecture logicielle

Il est intéressant de regarder l'architecture logicielle par rapport au cycle de vie du logiciel. Il est possible de distinguer trois grandes phases. La première correspond à la conception initiale de l'architecture. La deuxième consiste au développement proprement dit du logiciel par rapport à cette architecture. La troisième est liée à sa maintenance et à son évolution.

Ce qui caractérise la première phase est qu'il n'existe pas d'implémentation de l'architecture et du logiciel. Cette phase consiste à élaborer une architecture qui va permettre de répondre aux exigences des différents acteurs. Comment construire une bonne architecture par rapport à ces besoins ? Il est actuellement difficile de donner une réponse universelle. Nous manquons encore de méthodologie et l'expérience des architectes joue encore un rôle primordial. Néanmoins plusieurs propositions ont été avancées pour améliorer la situation. Plusieurs styles architecturaux avec leurs avantages et inconvénients ont été décrits et répertoriés [BMR+96, SG96, SSRB00]. Dans certains domaines comme l'aéronautique, il existe des architectures spécifiques qui ont été définies et éprouvées. L'approche catalysis [DW98] tente de fournir une première méthodologie dédiée à l'architecture logicielle.

L'architecture qui est conçue sert dans ce cas de manière prospective et permet la communication entre les différents acteurs. Elle n'existe que par l'intermédiaire de descriptions architecturales qui permettent de capter l'intention des concepteurs. Beaucoup de travaux se sont attachés à définir des langages de description d'architecture [Gar00, MT00] pour définir de telles descriptions.

Un des intérêts de ces langages est qu'il est possible de faire des analyses avant même que le développement soit commencé. Il est connu que plus une erreur est découverte tard dans le processus de développement plus il est difficile et coûteux de la corriger. Wright [AG97] permet de réaliser des descriptions comportementales pour faire des analyses d'interblocage ("deadlock") et de cohérence. Rapide [LKA+95] est un langage de prototypage pour simuler l'exécution afin de représenter des propriétés de causalité et de synchronisation.

Une autre préoccupation est de vérifier que l'architecture correspond bien aux attentes des acteurs. Pour cela, plusieurs méthodes ont été proposées pour évaluer une architecture. Les travaux de Constantine et Yourdon permettent d'évaluer la qualité de la structure d'une architecture logicielle en introduisant la notion de cohésion et de couplage [YC79]. Ils ont identifié sept niveaux de cohésion et sept niveaux de couplage. Comme cette méthode est basée sur une approche fonctionnelle, des recherches ont été réalisées pour prendre en compte l'approche orienté objet [CK94, Som92]. Deux autres méthodes ont été proposées pour évaluer une architecture grâce à des attributs de qualité. Ces attributs de qualité sont de deux types : ceux qui sont observables à l'exécution (comportement attendu, performances...) et ceux qui ne le sont pas (le coût et le temps de développement, la facilité d'évolution du système...). La première méthode nommée SAAM (Software Architecture Analysis Method) [KBAW94, KABC96] permet d'estimer une architecture en s'appuyant sur des scénarios utilisateurs. Il est ainsi possible d'évaluer différentes architectures candidates et de faire ressortir la plus appropriée ainsi que les avantages et inconvénients de chacune. La deuxième méthode, appelée ATAM (Architecture Tradeoff Analysis Method) [KKB+98, KBK+99], est une amélioration de la première. A la différence de SAAM, elle prend en compte plusieurs attributs à la fois et permet de localiser et d'analyser des compromis dans une architecture. Les auteurs ont bien su identifier les différentes étapes pour analyser une architecture mais l'évaluation est faite de façon informelle et reste à notre avis fort dépendante de l'expérience des architectes.

La majorité des travaux dans ce domaine se sont attachés à cette première phase de conception. Ces descriptions architecturales servent par la suite de référence pour le développement du logiciel.

La deuxième phase consiste à implémenter l'architecture qui a été adoptée. Elle est caractérisée par le fait que le code source existe mais que le produit n'est pas encore disponible chez le client et que l'architecture est connue puisque c'est elle qui guide le développement. Il faut trouver la ou les technologies qui vont permettre de réaliser concrètement l'architecture. Dans tous les cas, il faut former ses développeurs sur la partie de l'architecture et les technologies dont ils auront besoin. Cela pourra être un langage de programmation, un modèle de composants, la mise en place d'un réseau, des mécanismes particuliers pour la sécurité ou pour la gestion des licences... Il est possible de fournir aux ingénieurs un certain nombre d'outils pour faciliter le développement.

Une fois que le développement a commencé, un nouveau besoin essentiel apparaît. Il faut s'assurer que l'implémentation soit conforme aux spécifications tout au long du développement. Il existe deux manières pour vérifier cette contrainte. La première est une approche descendante où le code est généré automatiquement à partir de spécification de haut niveau. Cette solution n'est pas toujours utilisée ou de manière partielle car il faut pouvoir générer du code performant ce qui n'est généralement pas le cas. La deuxième est une approche montante où une ou plusieurs descriptions architecturales sont générées par ingénierie inverse. Il suffit ensuite de les comparer aux descriptions architecturales correspondant aux intentions des architectes. Cette solution a aussi ses limitations car il n'est pas toujours possible de remonter au niveau d'abstraction souhaité.

La troisième phase qui est vouée à la maintenance et à l'évolution du logiciel est caractérisée par le fait qu'il existe une version stable du logiciel qui a été déployé chez des clients et que l'architecture peut être connue ou non. Le fait que le logiciel existe aussi chez les clients est un facteur qu'il faut prendre en compte car si ces clients peuvent développer à partir du code source public, il devient très difficile de faire des modifications sur cette partie de code. Certaines évolutions ne seront donc plus possibles.

L'architecture peut ne plus être connue car elle s'est dégradée au cours du temps [Par94] ou parce qu'il n'existe plus de descriptions architecturales et que les concepteurs initiaux sont partis. Dans cette nouvelle phase, l'important est de pouvoir faire évoluer l'architecture en ajoutant et améliorant des propriétés tout en gardant les bonnes. Les besoins sont essentiellement de deux ordres : pouvoir intégrer de nouvelles fonctionnalités et restructurer l'architecture.

Les phases deux et trois restent assez proches et certains besoins peuvent être communs (même s'ils sont plus appropriés à une phase particulièrement). Peu de travaux se sont attachés à ces phases et particulièrement aux liens entre les descriptions architecturales et le code. Une de nos contributions a été de se focaliser sur la troisième phase puisque l'architecture V5 existait déjà.

4.3 En résumé

Dans ce chapitre, nous avons cherché à décrire notre vision de l'architecture logicielle par rapport à notre expérience industrielle avec Dassault Systèmes.

Nous avons, en premier lieu, repris les principales publications qui tentent de définir ce qu'est l'architecture logicielle. Nous nous sommes efforcés de ne pas simplement retranscrire les idées de ces publications mais aussi de faire une relecture par rapport à nos travaux et résultats.

Nous avons ensuite donné notre vision en séparant ce qu'est l'architecture logicielle de l'ingénierie pour ce domaine. Nous avons notamment fourni notre définition de l'architecture logicielle. Cette définition n'a pas pour objectif d'être acceptée par tout le monde mais plutôt de formaliser notre compréhension actuelle de l'architecture logicielle et ainsi faciliter la lecture de ce manuscrit.

Deuxième partie

Conceptualisation de l'Architecture V5

Afin de pouvoir travailler sur une architecture d'un logiciel, il est nécessaire de pouvoir la représenter. Pour cela il convient de connaître le modèle inhérent et de trouver un ou plusieurs formalismes, compréhensibles par tous les acteurs concernés, capables de décrire cette architecture. Ces formalismes pourront autant servir pour faciliter la communication interne (en réunion par exemple) que pour être exploités à travers des outils informatiques (visualisations, analyses...).

Le but de cette deuxième partie est de vous présenter l'architecture V5 et la démarche que nous avons suivie pour la décrire. Cette démarche n'est pas propre à notre contexte et peut être réutilisée dans d'autres situations. Cette partie est organisée de la manière suivante :

- Le chapitre 5 présente en premier lieu les différentes approches pour la description d'une architecture logicielle. Nous expliquerons les difficultés que nous avons rencontrées pour utiliser certaines de ces approches. Nous décrirons ensuite l'approche que nous avons adoptée pour pouvoir décrire l'architecture de CATIA V5.

- Le chapitre 6 a pour objectif de vous initier à l'architecture V5 ainsi qu'à l'organisation de développement de Dassault Systèmes. Ceci est essentiel pour comprendre nos travaux et l'intérêt de nos prototypes présentés dans la troisième partie.

Chapitre 5

Description d'une Architecture Logicielle

Les objectifs de la collaboration avec Dassault Systèmes portaient sur l'étude de l'architecture V5 afin de fournir des solutions pour améliorer le développement de CATIA V5. Lorsque nous souhaitons travailler sur une architecture logicielle, il faut en premier lieu disposer d'une description de cette architecture. Selon ses besoins, cette description peut être plus ou moins précise. Par exemple, si elle sert comme base de réflexion pour une réunion, une description semi-formelle pourra être suffisante. Dans l'optique d'analyses automatiques, il faudra une description plus formelle avec un ou plusieurs formalismes permettant de décrire les informations nécessaires pour ces analyses.

Afin de répondre à ce besoin de description architecturale, de nombreux travaux ont été proposés par la communauté scientifique. Au lieu de réinventer la roue, une bonne pratique est de regarder dans un premier temps ce qui existe et de l'évaluer vis-à-vis de ses propres besoins. Ainsi ce chapitre commence par un état de l'art sur les principales propositions (cf. section 5.1). Nous présenterons ensuite notre approche qui a permis de réaliser les descriptions architecturales qui nous intéressaient pour répondre aux besoins de Dassault Systèmes (cf. section 5.2). Nous expliquerons les difficultés que nous avons rencontrées pour utiliser les approches existantes et notamment pourquoi les langages de description d'architecture (ADLs : Architecture Description Languages) ont eu si peu de répercussions sur l'industrie.

5.1 Etat de l'art pour la description d'une architecture logicielle

Les travaux sur les formalismes de description d'architectures logicielles ont fait l'état de nombreuses recherches et ce principalement au début du domaine de l'architecture logicielle dans les années 90 [GP95, SG96].

Cette section présente un rapide état de l'art de ces formalismes en parcourant les principales propositions. Nous commencerons par les ADLs qui représentent une

part importante des travaux sur l'architecture logicielle. Ensuite, nous décrirons une approche qui prilivégie la description d'une architecture en prenant en compte différentes structures. Enfin, nous terminerons par le langage de modélisation UML. Nous nous poserons la question de savoir si UML est un bon candidat pour la description d'une architecture logicielle.

5.1.1 Travaux sur les ADLs

Une vue d'ensemble

Les travaux sur les ADLs ont été un axe de recherche important et de nombreuses propositions ont rapidement emmergé [Gar00, MT00]. Chaque ADL s'est spécialisé dans un thème particulier fournissant des fonctionnalités complémentaires pour la description et l'analyse architecturale : Aesop pour les styles architecturaux [GAO94], le style C2 pour les systèmes distribués et évolutifs (initialement pour les systèmes possédant une interface graphique) [TMA+96, MRT99], Darwin pour les systèmes d'envoi de messages distribués [NKM96], Rapide concernant la simulation [LKA+95, LV95], Wright pour les descriptions comportementales [AG97]...

La prolifération de ces ADLs a amené à réaliser des classifications et comparaisons entre eux [Ves93, Cle96]. Medvidoc et Taylor ont même été jusqu'à fournir un framework pour pouvoir comparer les différents ADLs présents et futurs [MT00].

Plusieurs travaux ont essayé de définir un méta-langage pour profiter des avantages de ces différents ADLs :

- ACME [GMW97, GMW00] fournit une représentation intermédiaire exploitable par les outils architecturaux associés aux différents ADLs. Ce langage fixe un vocabulaire pour les éléments communs aux différents ADLs liés à la structure (composants, connecteurs, systèmes, ports, rôles...) et laisse la possiblité de décrire les spécificités d'un ADL, comme une description comportementale pour Wright, par des propriétés sur ces éléments.
- ADML [PSSC99] qui est basé à la fois sur ACME et sur XML [XML00]. Bien que l'approche de ACME soit intéressante, son utilisation a été étonnament faible. Les auteurs d'ADML ont voulu définir une nouvelle version de ACME en s'appuyant sur le standard XML qui est de plus en plus reconnu et utilisé.
- AML [Wil99] qui a pour but de spécifier de manière précise les sémantiques des ADLs. Il est basé sur l'utilisation de trois constructions (éléments, types et relations) et sur un langage de contraintes exprimées par des prédicats en logique temporelle sur ceux-ci. Par rapport à ACME, AML a l'avantage de prendre en compte les aspects dynamiques.

L'apport des ADLs

L'architecture d'un système joue un rôle déterminant pour le succès de ce système. La conception d'une bonne architecture peut amener à un produit qui

répond aux besoins des clients et qui peut être modifié facilement pour rajouter une nouvelle fonctionnalité, alors qu'une architecture inappropriée peut avoir des conséquences désastreuses jusqu'à l'arrêt du projet. Malgré cela, la pratique architecturale a été longtemps une affaire ad hoc et informelle. Ainsi de nombreuses incohérences et ambiguïtés subsistaient et n'étaient trouvées que tardivement dans le cycle de développement. L'architecte ne possédait pas non plus d'outils pour l'aider dans sa tâche de conception. Les ADLs tentent de prendre en compte ces problèmes en décrivant l'architecture d'un système à partir de notations formelles et de contraintes à vérifier exploitables par des outils automatiques.

Les principaux concepts des ADLs

Généralement les ADLs permettent de décrire une architecture par une configuration à base de composants et de connecteurs.

- **Les composants** : un composant est une entité fournissant des fonctionnalités (principalement des unités de calcul et de données). Il possède un certain nombre d'interfaces (souvent appelées des ports). Chacune d'entre elles décrit un sous-ensemble du comportement du composant et du comportement attendu du système dans lequel il va intéragir. Une interface d'un composant joue en quelque sorte le rôle d'un filtre sur le comportement global du composant. Une description de ce dernier est aussi fournie qui permet entre autres de décrire comment les différentes interfaces du composant interagissent ensemble.

- **Les connecteurs** : un connecteur est une entité définissant l'interaction (protocole de communication) entre plusieurs composants. Comme pour un composant, un connecteur possède un certain nombre d'interfaces (souvent appelées rôles) et une description globale de son comportement. Un des apports des ADLs a été de remonter le concept de connecteur au même niveau que celui des composants. Il a ainsi été possible de formaliser de manière précise un ensemble de protocoles bien connus comme les pipelines, le protocole client-serveur... Une taxonomie des différents connecteurs a été proposée [MMP00].

- **Une configuration** : ces composants et connecteurs sont considérés comme des types qui peuvent être instanciés et assemblés à partir de leurs interfaces (ports et rôles) pour former une configuration particulière. Un langage de contraintes est souvent disponible afin de formuler certaines règles de composition à vérifier.

Une particularité des ADLs est de s'être focalisée sur la description formelle du comportement de l'application à travers ses composants et connecteurs. Pour réaliser ces descriptions formelles, les ADLs se sont basés sur différents formalismes : CHAM [BL93, IW95], CSP [AG97, Hoa78], LOTOS [BB89, HL97], poset (partially ordered event set) [LKA+95, Pra86], StateChart de Harel [Har87, NKM96]... Les ADLs disposent généralement d'outils associés qui exploitent ces descriptions formelles pour réaliser des analyses architecturales.

Un exemple de description en Wright

L'annexe B fournit un exemple complet de description d'une architecture logicielle pour la création d'un index avec l'ADL Wright.

Dans cet exemple, la configuration du système comporte trois instances de composants différents (un composant qui filtre le texte pour renvoyer des couples (mot, numéro de ligne), un composant qui se charge du tri alphabétique de ces couples et un composant qui gère les éventuels doublons) et deux instances du même type de connecteur Pipe (pour gérer les canaux de communication entre les composants).

La description du comportement de ces types de composants et de ce connecteur est réalisée en CSP. Par exemple la description du type d'interface DataInput indique que l'interface (port ou rôle) doit lire l'ensemble des données qui arrivent et s'arrêter au signal *end-of-data*.

Une explication détaillée de l'exemple, de la sémantique de CSP et des différentes vérifications que permet Wright est disponible dans mon rapport de DEA [San97].

Ce qu'il en est aujourd'hui

Malgré l'abondance des ADLs existants, très peu ont influé sur la pratique industrielle. Les seuls que nous connaissons, sont dans des domaines bien précis : ROOM[1] pour le domaine des télécommunications [SGW94] développé par la société ObjecTime (et racheté depuis peu par Rational) et MetaH pour l'avionique [Ves98] développé par Honeywell.

Actuellement, il n'existe plus beaucoup de travaux sur les ADLs et les principaux concepts novateurs ont été définis au début des recherches. Il ne reste véritablement plus que les travaux de l'équipe de l'université de Californie avec leur style C2 et leur environnement ArchStudio [MOT+00]. Même Rapide, qui a été longtemps un des ADLs les plus prometteurs, n'a plus évolué depuis plusieurs années[2].

5.1.2 La norme ANSI/IEEE Std 1471-2000 [Gro00]

La grande majorité des travaux académiques sur l'architecture logicielle, étudie ce que nous appellons dans cette thèse l'architecture logique. Elle décrit la décomposition fonctionnelle du logiciel sous forme d'une configuration de composants et de connecteurs conceptuels. Bien que ces travaux préconisent l'utilisation de multiples vues, comme les vues données et processus pour Perry et Wolf [PW92], ces vues sont toutes rattachées à la structure logique.

[1] Bass et al. [BCK98] présentent ROOM comme un ADL mais cela reste contestable.

[2] En fait, l'histoire veut que les auteurs aient monté une société qui a fait faillite dernièrement.

Récemment, plusieurs travaux, réalisés essentiellement par et/ou pour des industriels, ont mis en évidence la nécessité de prendre en compte d'autres structures architecturales [Kru95, HNS00, GS00]. Ces travaux ont montré que l'architecture logique à elle seule ne suffisait pas à décrire les systèmes étudiés. La description d'une architecture est réalisée par 4 structures (conceptuelle, module, code et exécution) dans les travaux de Hofmeister et al. pour Siemens [HNS00], "4+1" (logique, processus, développement, physique et scénarios) dans les travaux de Kruchten pour Rational [Kru95]. Par contre le nombre et la nature de ces structures ne sont pas toujours similaires.

Cette approche de description d'une architecture logicielle par l'intermédiaire de plusieurs structures semble être de plus en plus reconnue et encouragée. Dernièrement, le groupe de réflexion de l'IEEE sur l'architecture logicielle a formulé des recommandations pour réaliser des descriptions architecturales. Ces recommandations tournent autour d'un modèle conceptuel[3] qui préconise l'utilisation de plusieurs structures (cf. figure 4.6). Les travaux précédents rentrent dans le cadre de ce modèle conceptuel. Ces recommandations ont fait l'objet d'une normalisation sous le nom de ANSI/IEEE Std 1471-2000 [Gro00]. Bien que récente, cette norme semble prendre de plus en plus d'importance [KSK+01, TOG].

Nous pouvons noter que même si Kruchten, Hofmeister et al. utilisent le terme de vue, ils décrivent bien des vues pour des structures différentes. Dans la norme ANSI/IEEE Std 1471-2000, leurs vues correspondent en fait à des points de vue ("viewpoints"). Un point de vue est une sorte de patron ou modèle qui spécifie les conventions et notations pour construire différentes vues. Ils couvrent certains besoins bien identifiés de différents acteurs.

Les "4+1" vues de Kruchten [Kru95]

Kruchten a été le premier à décrire une approche pour spécifier une architecture logicielle en utilisant différentes structures. Pour l'auteur, la description de l'architecture se fait par l'intermédiaire de quatre vues (logique, physique, processus et développement) plus une qui est redondante avec les précédentes et qui permet d'illustrer sur des exemples de scénarios la coopération entre les différentes vues. La figure 5.1 présente la décomposition entre ces "4+1" vues.

Pour chacune de ces vues, l'auteur décrit le rôle et les acteurs concernés. Elles sont toutes accompagnées d'une notation qui permet au concepteur de décrire ses propres vues (cf. annexe C).

- **La vue logique ou décomposition orienté objet** : elle s'attache à décrire les besoins fonctionnels (quels sont les services que le système doit fournir aux utilisateurs). Elle est destinée aux ingénieurs qui sont chargés de la modélisation du logiciel en collaboration avec les utilisateurs et les spécialistes du domaine. Elle se rattache à une modélisation orienté objet puisqu'elle

[3] La présentation de ce modèle a été réalisée à la section 4.1.7.

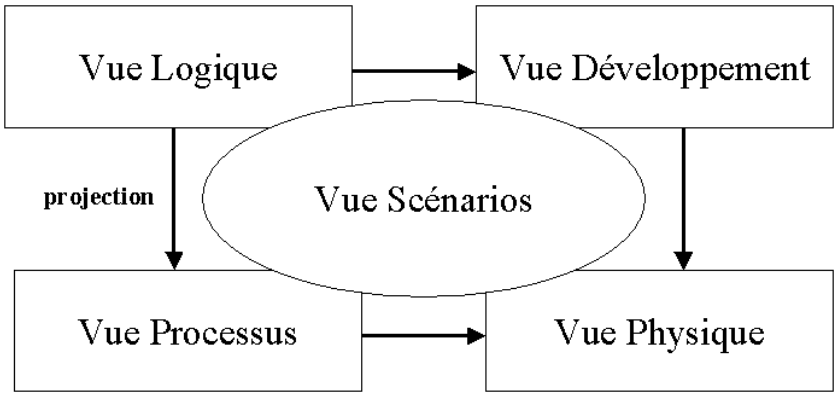


FIG. 5.1 – Le modèle "4+1" vues de Kruchten

définit les abstractions en termes de classes, objets et leurs relations (association, agrégation, utilisation, spécialisation, instanciation). Cette vue s'organise autour d'un diagramme de classes utilisant la notation de Booch [Boo94]. D'autres informations peuvent venir compléter le diagramme de classes. Par exemple, le regroupement de classes ayant des fonctionnalités communes, ou la description du comportement interne d'un objet par l'utilisation d'un diagramme d'états-transitions.

- **La vue processus ou décomposition en processus** : cette vue prend en charge des aspects non fonctionnels comme la performance et la disponibilité. Elle aborde des problèmes liés à la distribution et la synchronisation ainsi que ceux liés aux mécanismes de réplication nécessaires à la tolérance aux fautes. Cette vue est utilisée par les ingénieurs chargés de regrouper et d'affecter les abstractions de la vue logique dans des processus ou tâches[4]. Pour faciliter la description, il est possible d'utiliser différents niveaux d'abstraction. Au niveau d'abstraction le plus élevé, les entités de cette vue sont des processus définis comme des ensembles de tâches pouvant s'exécuter de manière autonome, être répliqués, arrêtés ou relancés. Le résultat est un réseau logique de processus communicants qui devront être liés aux entités machines de la vue physique. C'est la notation proposée par Booch pour décrire les communications entre les tâches Ada qui a été retenue pour cette vue.

- **La vue développement ou décomposition en sous-systèmes** : cette vue se focalise sur l'organisation du code en différents modules et leurs dépendances. Le logiciel peut être découpé en sous-systèmes qui peuvent être développés par un ou plusieurs développeurs. La description peut utiliser une décomposition hiérarchique en couches. Chaque couche décrivant les services disponibles pour la couche supérieure. Cette vue sert aussi de référence pour planifier l'échéancier et la répartition du développement entre les différentes équipes. Elle est plus particulièrement destinée au chef de projet. Une nouvelle fois, c'est une variante de la notation de Booch appliquée au niveau architectural qui est utilisée.

[4]threads en anglais.

- **La vue physique ou correspondance entre le logiciel ("software") et le matériel ("hardware")** : comme pour la vue processus, cette vue prend en compte des aspects non fonctionnels de performance, de tolérance aux fautes, passage à l'échelle, disponibilité. Par contre, nous nous plaçons au niveau matériel où les différentes entités sont des machines placées sur un réseau. Les liens entre les nœuds du réseau modélisent les différents types de communications (communication permanente ou non, lien uni ou bi-directionnel, taille de la bande passante...). Cette vue s'adresse aux ingénieurs système et réseau qui spécifient comment les différents processus du logiciel doivent être répartis sur le réseau. C'est une notation informelle qui est utilisée avec des boîtes et différents types de liens (lignes continues ou non, avec des flèches aux extrémités ou non, avec différentes épaisseurs...). L'auteur n'indique pas si cette notation correspond à une syntaxe graphique existante.

- **La vue scénarios ou regroupement des 4 vues en une seule** : cette vue sert à illustrer des exemples de coopération entre les quatre vues précédentes. Elle décrit des séquences d'interactions entre objets et processus par des diagrammes d'interaction entre objets introduits par Rubin et Goldberg [RG92]. Cette vue est redondante par rapport aux précédentes (d'où le "+1"). Elle ne sert pas principalement à la description de l'architecture du logiciel mais plutôt à illustrer son fonctionnement à partir de scénarios pour guider la conception de l'architecture. Le deuxième intérêt de cette vue est de pouvoir servir de base pour le conception des tests de validation de l'architecture une fois que la conception est terminée. Ces tests de validation pourront être effectués autant sur le papier que sur un premier prototype. La notation est similaire à celle de la vue logique pour les entités et celle de la vue processus pour les liens entre ces entités.

Ces différentes vues ne sont pas indépendantes et il existe des relations entre elles (cf. figure 5.1). Ces relations sont définies par des projections des entités d'une vue sur celles d'une autre vue. Elles sont exprimés sous forme de correspondance entre les nœuds des différentes vues.

Les 4 vues de Hofmeister et al. [HNS00]

Les travaux de Hofmeister, Nord et Soni sont un deuxième exemple significatif d'utilisation de différentes structures pour la description d'une architecture logicielle. Les auteurs ont établi leur approche à partir de l'étude de plusieurs systèmes développés par leur société Siemens. Ces systèmes proviennent de domaines variés comme l'image et le traitement du signal, le temps réel, l'instrumentation et le contrôle, la communication... La taille de ces systèmes analysés vont de moins de 100 000 lignes de code pour le plus petit à plus d'un million de lignes de code pour les plus gros.

A partir de leur expertise sur ces systèmes, les auteurs se sont aperçus que différentes structures architecturales cohabitaient par le fait qu'elles ont un couplage faible et couvrent des besoins différents. Les auteurs ont pu décrire l'architecture de

ces différents systèmes par quatre vues architecturales différentes : vue conceptuelle, vue module, vue exécution et vue code (cf. figure 5.2). Bien que ces structures soient différentes de celles de Kruchten par leur nom, certaines d'entre elles sont assez similaires.

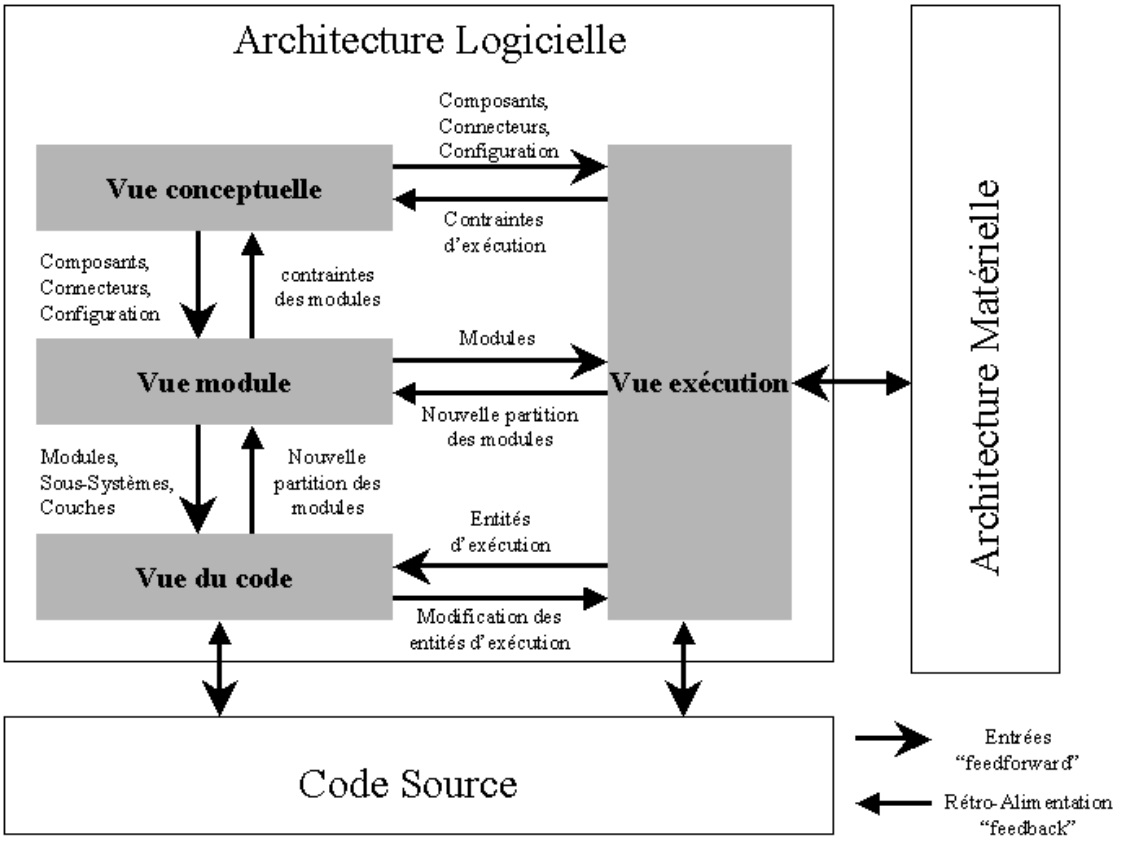


FIG. 5.2 – Les quatre vues d'une architecture logicielle pour Hofmeister et al.

Pour chacune des ces vues, les auteurs fournissent un méta-modèle décrivant les différentes entités et relations disponibles pour réaliser ses propres vues (cf. annexe D). Ils ont priviligé l'utilisation du standard UML pour leur notation [HNS99].

- **La vue conceptuelle** : cette vue est celle qui est la plus proche du domaine d'application car c'est celle qui est la moins contrainte par les plateformes logicielles et matérielles. Avec cette vue, il est possible de modéliser le produit en termes d'une collection de composants et de connecteurs conceptuels. Elle se rapproche de la vue logique de Kruchten. En fait, elle est comparable aux ADLs standards.

- **La vue module** : le principal objectif de cette vue est de prendre en compte les besoins liés à l'implémentation du système et à l'organisation de développement. A la différence de la vue conceptuelle, cette vue permet de décrire explicitement comment les différents éléments interagissent avec la plateforme logicielle (comme les services du système d'exploitation, les APIs

d'une librairie spécialisée...). Cette vue n'est pas simplement un raffinement de la vue conceptuelle. Elle décrit aussi une nouvelle structure à base de sous-systèmes, de modules et de couches hiérarchiques. Elle définit aussi la répartition des éléments conceptuels par rapport aux différents modules. Il est ainsi possible de structurer différemment l'application en regroupant les éléments conceptuels par domaine (module pour l'interface graphique, module de calcul, module pour le réseau...). Cette nouvelle structure facilite la gestion de la complexité du système et la répartition du travail collaboratif. Dans les systèmes de grande taille, ces deux structures (conceptuelle et module) sont généralement différentes. Cette vue se rapproche de la vue développement de Kruchten. Elle utilise les travaux sur les MILs (Module Interconnection Language) [DK76, PDN86] et les techniques de paquetages Ada et Java.

- **La vue du code** : cette vue modélise l'organisation du code source sous forme de fichiers sources (.h, .cpp, .java...), librairies (dlls...), fichiers binaires (.o, .class...), exécutables (.exe...) et de fichiers de configuration pour le déploiement. Généralement il existe différentes versions pour ces fichiers. Elle a donc aussi pour objectif de décrire les décisions relatives à la gestion de configuration. Le principal but de cette vue est de faciliter la construction, l'intégration, l'installation et le test du système tout en respectant l'intégrité des trois autres vues. Cette vue n'est pas présente dans les travaux de Kruchten.

- **La vue exécution** : cette vue prend en compte les aspects dynamiques du logiciel. Elle décrit le système en terme d'éléments exécutables de la plateforme (processus, tâches, sockets, DLLs...), de ressources de la plateforme (mémoire, espace d'adressage, processeur(s)...), de mécanismes de communication entre ces éléments exécutables (RPC : Remote Procedure Call, DCOM : Distributed Component Object Model, IPC : InterProcess Communication...). Elle permet de décrire des informations liées à la distribution comme l'attribution des fonctionnalités du système sur les éléments exécutables, les types de communication entre ceux-ci et l'allocation des ressources physiques pour ceux-ci. Les aspects non fonctionnels comme la location, la migration, la réplication de ces instances exécutables sont aussi pris en compte. Cette vue se rapproche des vues processus et physique de Kruchten.

Même si ces différentes vues architecturales ont un couplage faible, elles ne sont pas disjointes. Certaines entités et relations peuvent se retrouver dans plusieurs vues. Par exemple, il existe une relation de correspondance (de un pour un) entre les entités d'exécution dans la vue d'exécution et les fichiers dans la vue code. Comme la figure 5.2 le suggère, il existe aussi d'autres liens de dépendance entre elles. Certaines décisions de conception pour une vue peuvent affecter et remettre en cause la conception des autres vues.

Comparatif des vues de Kruchten et de Hofmeister et al.

Nous proposons maintenant un comparatif des vues que nous venons de décrire. Bien qu'il soit difficile de comparer précisément ces différentes vues, il est possible de les situer globalement par rapport aux données qu'elles manipulent ainsi qu'à leur rôle respectif. La figure 5.3 présente ce récapitulatif.

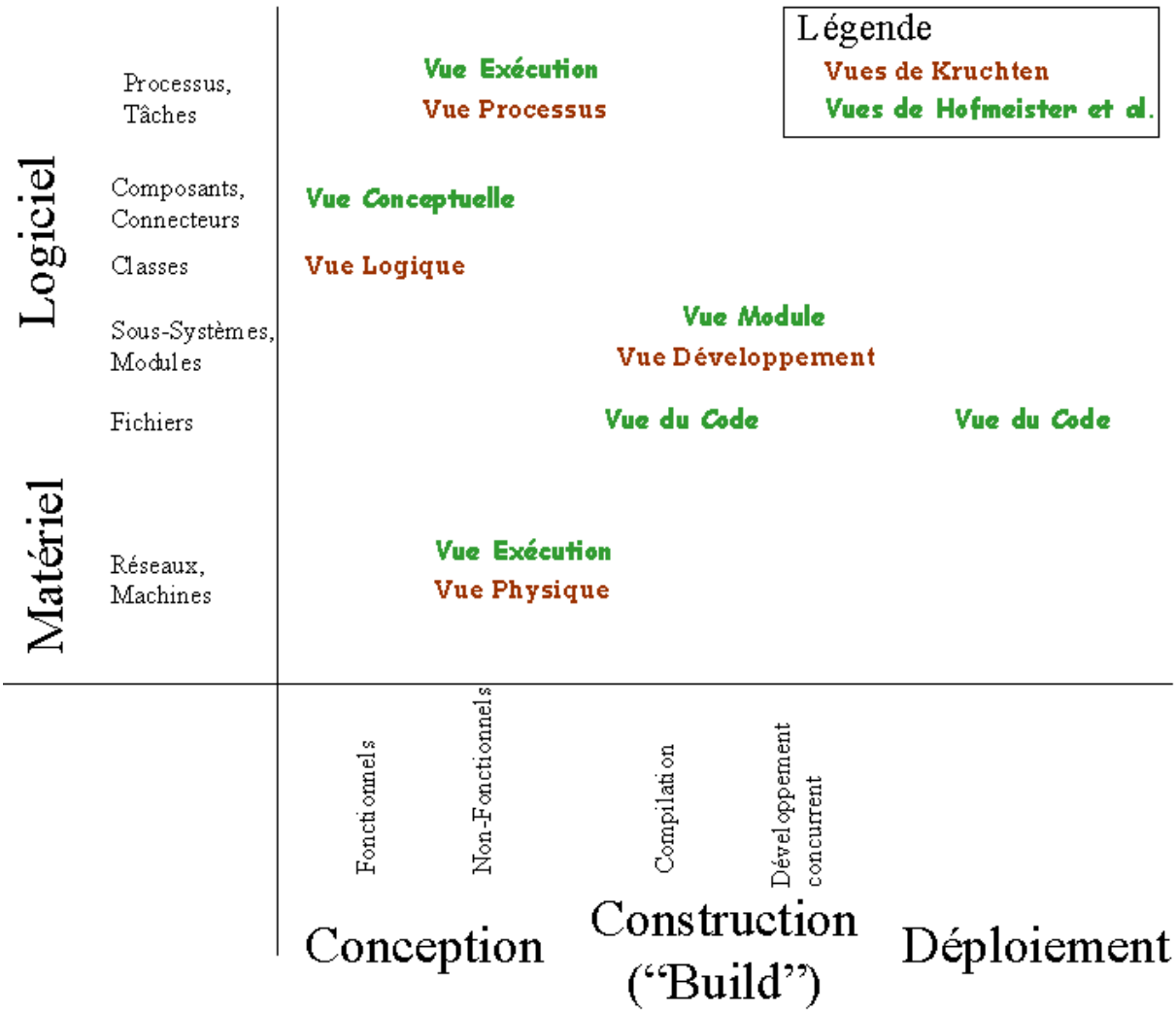


FIG. 5.3 – Comparaison des vues de Kruchten et de Hofmeister et al.

Nous pouvons noter que dans la globalité ces différentes vues restent assez proches à part la vue du code que Kruchten ne propose pas. Par contre, Hofmeister et al. regroupent les vues processus et physique dans une seule qui est la vue exécution. Comme cette vue exécution mélange à la fois des aspects matériel et logiciel, nous pensons comme Kruchten qu'il vaut mieux séparer ces aspects.

5.1.3 Et UML dans tout ça ?

Une question qui vient tout naturellement est de se demander pourquoi ne pas utiliser UML[RJB99] pour la description d'une architecture logicielle ? En effet UML

est devenu un standard de modélisation objet adopté par la majorité des industriels. De plus, il offre la possibilité de prendre en compte différents aspects et structures du logiciel en offrant plusieurs diagrammes de spécification.

Un petit historique

UML est né de par la volonté de Rumbaugh, Booch et Jacobson (personnes influentes dans le domaine des méthodologies objets) d'unifier leurs travaux au sein d'une méthode unique. Ils ont été rapidement rejoints par un consortium de plusieurs grandes entreprises pour définir la version 1.0. Les auteurs d'UML insistent sur le fait que la notation UML est un langage de modélisation objet et non pas une méthode objet. Cette notation UML est une fusion d'un ensemble de notations dont les principales sont celles de Booch [Boo91], OMT [RBP+91] et OOSE [JCJO92].

Une rapide présentation d'UML

UML se concentre sur la description d'un système[5] plutôt que sur le processus de développement[6]. Pour décrire sa propre modélisation, la notation UML met à disposition neufs diagrammes :

- **Les diagrammes de cas d'utilisation** qui permettent de définir l'ensemble des fonctionnalités du système du point de vue de l'utilisateur.
- **Les diagrammes de classes** qui modélisent la structure statique du logiciel en termes de classes et de relations entre elles.
- **Les diagrammes d'états-transitions** permettent de décrire le comportement d'une classe sous forme d'automates d'états finis.
- **Les diagrammes d'activités** qui sont une variante des diagrammes d'états-transitions et qui permettent de représenter le comportement interne d'une méthode ou le déroulement d'un cas d'utilisation.
- **Les diagrammes d'objets** montrent la structure statique du logiciel par un ensemble d'instances des différentes classes des diagrammes de classes.
- **Les diagrammes de collaboration** montrent les interactions entre les objets par une représentation spatiale. Ces diagrammes sont une extension des diagrammes d'objets.
- **Les diagrammes de séquence** qui, comme les diagrammes de collaboration, montrent les relations entre les objets mais en privilégiant cette fois l'aspect temporel.

[5] A l'origine, la notation UML était utilisée pour la description de systèmes logiciels. Actuellement, elle est aussi adoptée pour décrire d'autres types de systèmes.

[6] Un processus de développement basé sur UML, nommé RUP pour Rational Unified Process, a été défini par Rational [JBR99].

- **Les diagrammes de composants** qui décrivent les éléments physiques (modules, tâches, sous-programmes) constituant le système ainsi que leurs relations. Les composants peuvent être organisés en paquetages, qui définissent des sous-systèmes.

- **Les diagrammes de déploiement** qui décrivent la disposition physique du matériel qui compose le système et la répartition des composants sur ce matériel.

La notation UML a été conçue pour être générique, extensible et configurable par l'utilisateur. Ceci a pour avantage de pouvoir modéliser un grand nombre de domaines d'activités.

Il est intéressant de souligner la similarité entre certains diagrammes d'UML et les vues des travaux de Kruchten : les diagrammes de cas d'utilisation pour la vue scénarios, les diagrammes de classes pour la vue logique, les diagrammes de composants pour les vues processus et développement, et les diagrammes de déploiement pour la vue physique.

UML et l'architecture logicielle

Qu'en est-il pour l'architecture logicielle ? Cette notation, qui a été conçue à l'origine pour une modélisation objet, permet-elle de réaliser des descriptions architecturales semblables aux ADLs ?

Plusieurs travaux ont montré que c'était effectivement possible. Certains auteurs se sont appuyés sur la notation UML pour décrire leur ADL [EM99, RMRR98]. Les travaux de Hofmeister et al., présentés précédemment, préconisent aussi la notation UML pour décrire leur quatre vues architecturales [HNS99].

La technique utilisée à travers les diverses approches est souvent similaire. Les auteurs utilisent les diagrammes proposés par UML pour spécifier les différents aspects et structures d'une architecture : les diagrammes de classes permettent de représenter les aspects structurels (composants, connecteurs, modules, processus...), les diagrammes d'objets décrivent les instances de ces entités, les diagrammes d'états-transitions servent pour les descriptions comportementales, les diagrammes de composants peuvent être utilisés pour les fichiers (fichiers sources, librairies, exécutables...) , les diagrammes de déploiement sont plus appropriés pour les aspects matériels (machines, réseaux...). Les paquetages UML offrent la possibilité de regrouper plusieurs entités et de réaliser des descriptions hiérarchiques et modulaires. Le langage de contraintes OCL [WK98, WK99] associé à UML permet d'exprimer les contraintes architecturales à vérifier.

Par contre, même si la notation UML permet de réaliser des descriptions architecturales, elle n'est pas forcément la plus appropriée et la plus pratique. Un certain nombre de lacunes et de difficultés à utiliser cette notation ont été identifiées [GK00, HNS99, MR99] :

- **Description de concepts architecturaux** : UML ne fournit pas de description explicite pour les concepts architecturaux comme par exemple les notions de connecteurs ou de ports. Le lien entre ces concepts architecturaux et leur modélisation doit être défini et maintenu implicitement par l'architecte. Pour pouvoir représenter ces concepts, il faut utiliser massivement les mécanismes d'extension comme les stéréotypes.

- **Manque de sémantique** : l'utilisation intensive de mécanismes d'extension comme les stéréotypes pose des problèmes sémantiques. En effet, utiliser une même notation pour des concepts relativement différents porte à confusion. Ceci ne facilite pas non plus l'exploitation de la description par différents outils.

- **Représentation des différentes structures** : les différents diagrammes disponibles dans la notation UML répondent à une modélisation objet. Par contre, ils ne sont pas toujours adaptés pour décrire les différentes structures d'une architecture logicielle (comme celles de Kruchten ou de Hofmeister et al.). Il arrive ainsi souvent que l'on doive utiliser les diagrammes de classes pour plusieurs structures en utilisant les mécanismes d'extension pour les différencier.

- **Représentation graphique** : il devient rapidement difficile de gérer la représentation graphique pour afficher l'ensemble des informations. L'utilisation intensive des stéréotypes dégrade la vue d'ensemble et le repérage des différentes entités. De plus, il n'est pas toujours aisé de retrouver la correspondance d'éléments entre différentes vues.

Les différentes expérimentations montrent clairement que bien qu'il soit possible d'utiliser UML pour représenter une architecture logicielle, cette notation n'est pas la mieux adaptée. Certains chercheurs ont ainsi tenté d'étendre le méta-modèle d'UML afin de faciliter la description architecturale. L'idée est de ne plus utiliser les mécanismes d'extension pour décrire les concepts architecturaux mais de les rajouter directement au méta-modèle. L'exemple le plus représentatif est l'approche Catalysis [DW98]. Le principal inconvénient de cette technique est que du même coup nous ne suivons plus exactement le standard UML et qu'il devient difficile d'utiliser les outils associés. Cette approche a d'un côté l'avantage de fournir des solutions concrètes et adaptées à son domaine d'activité mais a de l'autre côté l'inconvénient d'être propriétaire et de voir apparaître de multiples modèles s'il n'y a pas un effort de standardisation.

Nous pouvons tout de même noter que la version 2.0 d'UML semble fournir quelques réponses pour ces aspects. Par exemple, UML 2.0 devrait prendre en compte la notion de composant logique (dans le sens des composants COM, des composants EJB...). Nous ne pouvons malheureusement pas en dire plus pour l'instant car la spécification de cette version 2.0 n'est pas encore disponible mais il est intéressant de suivre l'évolution de ce standard qui semble vouloir intégrer des concepts pour la gestion d'une architecture logicielle.

5.2 Notre approche

Afin de décrire l'architecture logicielle de CATIA V5, nous avons d'abord tenté d'utiliser les principales approches décrites précédemment (section 5.2.1). Nous avons été confrontés à plusieurs difficultés qui nous ont amenés à définir notre propre stratégie (section 5.3).

5.2.1 Evaluation des différentes approches de description d'une architecture logicielle

Un ADL pour l'architecture V5 ?

Puisque nous voulions réaliser une description de l'architecture logicielle de CATIA V5, nous nous sommes tout naturellement tournés vers les ADLs. Nous avons regardé s'il n'était pas possible d'utiliser un ou plusieurs ADLs existants.

Notre expérience avec Dassault Systèmes a permis de relever quelques caractéritiques des ADLs qui ne sont pas compatibles avec des contraintes industrielles. Nous avons été confrontés à plusieurs problèmes qui nous ont permis de comprendre pourquoi les ADLs n'arrivent pas à influencer la pratique quoditienne des industriels [LSE00, SLEFew] :

- **Des concepts architecturaux spécifiques** : les ADLs introduisent des concepts abstraits tels que les connecteurs, les ports... Ces concepts abstraits ne correspondent pas forcément à des concepts au niveau du code (interfaces, classes...). C'est une démarche acceptable lors d'un premier développement, mais plus difficile quand le code existe déjà. La correspondance est parfois loin d'être évidente et un concept architectural peut être réparti dans plusieurs fichiers de programmation. Du coup, certains de ces concepts architecturaux ne sont pas toujours compris ou même reconnus par les ingénieurs de Dassault Systèmes. La principale raison est qu'ils ne voient pas le lien avec ce qu'ils font et utilisent ; d'autant que les mécanismes de l'Object Modeler privilégient l'assemblage de composants par construction plutôt que par connexion[7].
- **Prendre en compte l'existant** : les ADLs ne fournissent aucun support pour gérer les systèmes existants. Ceci est difficilement acceptable par Dassault Systèmes puisqu'actuellement CATIA V5 comporte près de 5 millions de lignes de code. Il n'est pas non plus concevable de vouloir reconstruire la description architecturale à la main. En pratique, cela demande de disposer d'un outil d'ingénierie inverse qui reconstruise cette description à partir du code source.

[7] L'assemblage de composants se fait par construction lorsque le modèle de composants offre essentiellement des mécanismes pour la conception des composants et peu pour leurs connexions. C'est par exemple le cas avec le modèle de composants COM ou l'Object Modeler qui fournissent des instructions de programmation au-dessus de C++ comme la délégation, l'agrégation, l'extension... A l'inverse, l'assemblage de composants se fait par connexion lorsque le modèle de composants favorise la réalisation d'applications par la connexion des divers composants. C'est par exemple le cas avec le modèle de composants des JavaBeans où les composants peuvent être connectés visuellement.

Mais, il s'avère que cette reconstruction est loin d'être évidente et il n'est pas toujours possible de synthétiser des concepts comme les connecteurs à partir du code source. Du coup, les ADLs ne permettent de travailler que dans le cadre d'une conception initiale.

- **Lien constant avec le code source** : du fait des temps de cycle très courts entre deux releases (4 mois), le code source de CATIA V5 évolue continuellement. Il ne suffit donc pas de réussir à réaliser une description architecturale à un moment donné. Il faut aussi garder constamment le lien entre cette description architecturale et le code source pour qu'elle ne devienne pas obsolète rapidement. La plupart des ADLs ne répondent pas à ce besoin puisqu'ils ne prennent pas en compte ce lien entre une description architecturale et le code source (ni en génération de code, ni en ingénierie inverse) et s'adaptent mal aux phases de maintenance.

- **Besoins de représentations multi-vues** : les travaux sur les ADLs ne s'intéressent essentiellement qu'à la vue logique de l'architecture. Pourtant, dans le cas de systèmes complexes tels que CATIA V5, ceci n'est pas suffisant parce qu'il existe d'autres structures architecturales qui jouent un rôle important dans le processus de développement [BCK98, Gro00].

- **Besoins d'analyses architecturales autres que comportementales** : les ADLs et outils associés se focalisent plus particulièrement sur les analyses comportementales (Wright pour la vérification [AG97], Rapide pour la simulation [LKA+95]). Bien que très utiles pour les systèmes critiques, elles ne constituent généralement pas un intérêt majeur pour les autres systèmes. Ces outils ne sont alors intéressants que dans la mesure où ils ne coûtent rien, c'est-à-dire qu'ils sont automatiques et rapides. Pour leur part, les ingénieurs de Dassault Systèmes sont beaucoup plus intéressés par des outils d'analyse d'impact pour gérer le développement et l'évolution de CATIA V5. Dans ce cas les liens de dépendance sont essentiels et suffisants pour pouvoir réaliser de tels calculs.

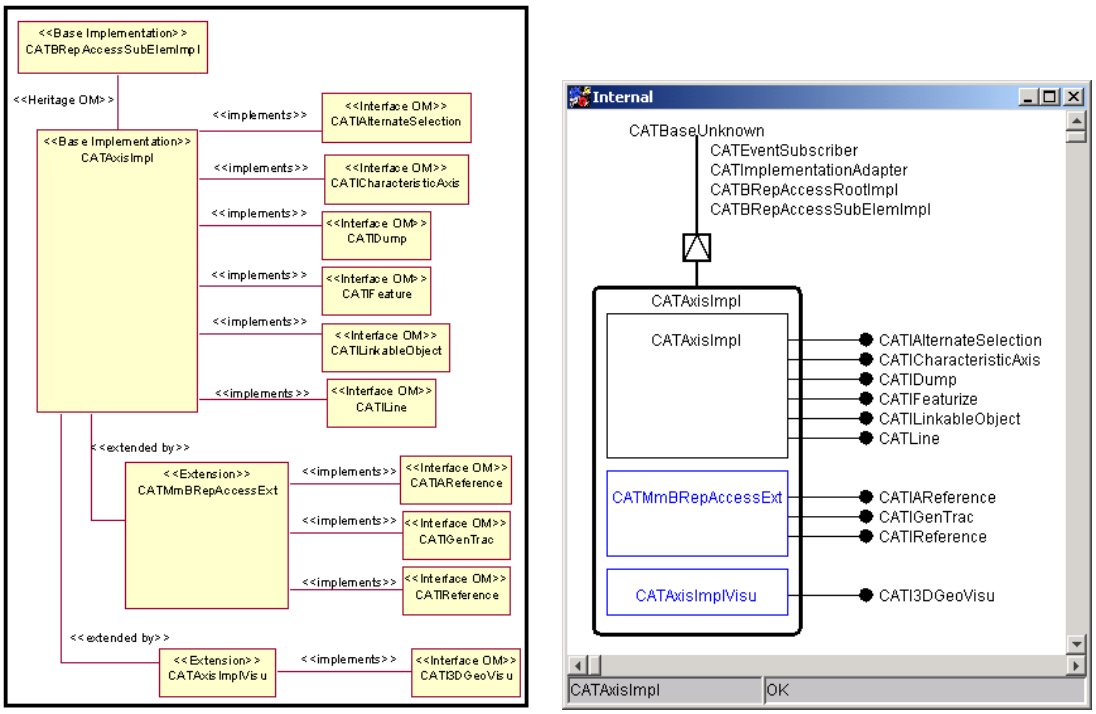
Notre conclusion est qu'aujourd'hui, les communautés scientifiques et techniques n'ont pas d'ADL prêt à l'emploi pour les systèmes existants et qui continuent à évoluer [San00]. Il est donc facile de comprendre pourquoi les ADLs ont reçu si peu d'attention de la part des industriels.

Les descriptions et analyses architecturales que nous souhaitons réaliser ne correspondent pas à ce que la communauté académique appelle communément un ADL. Les principaux besoins de Dassault Systèmes ne se situent pas au niveau d'analyses comportementales mais plutôt à des aides à la compréhension et à des analyses d'impact. De plus, Dassault Systèmes ne se situe pas dans un développement initial mais doit gérer la maintenance et l'évolution d'un logiciel comportant près de 5 millions de lignes de code. Il n'est donc pas acceptable pour Dassault Systèmes de ne pas prendre en compte l'existant. Dès lors, il a fallu trouver le bon niveau de description pour répondre à ces besoins.

Utilisation du standard UML pour décrire l'architecture de CATIA V5 ?

Comme UML fait partie de la culture d'entreprise de Dassault Systèmes, nous avons tenté d'utiliser la notation UML pour représenter nos vues architecturales à travers l'outil Rational Rose [Ros].

La figure 5.4 montre un exemple de description d'un composant Object Modeler avec la notation UML. Comme les concepts de l'Object Modeler n'ont pas de représentation explicite dans UML, nous avons dû recourir au mécanisme d'extension appelé les stéréotypes. Ce mécanisme permet de spécialiser une entité d'UML en lui attribuant un nom pour former un nouveau stéréotype. Nous nous en sommes servi aussi bien pour les entités que pour les relations. Par exemple, nous avons créé des stéréotypes pour distinguer les différents types d'interfaces existantes dans l'architecture V5 (les interfaces Object Modeler, les interfaces C++, les interfaces IDL...). De même pour les relations, nous avons utilisé les stéréotypes pour différencier les types d'héritage existants (héritage Object Modeler, héritage C++).



Description UML avec l'outil Rational Rose

Description avec notre syntaxe graphique et notre outil

FIG. 5.4 – Description du composant Object Modeler CATAxisImpl

Lors de nos expérimentations avec la notation UML, nous avons été confrontés principalement à deux types de problèmes :

- **La non adéquation du formalisme** : sur un exemple aussi simple que celui de la figure 5.4, nous pouvons constater que même avec peu d'éléments, il y a

rapidement un encombrement, ce qui ne facilite pas la lecture. L'architecture de CATIA V5 possède des composants Object Modeler qui sont beaucoup plus complexes et qui peuvent atteindre plusieurs dizaines d'interfaces. La description en UML de ces composants n'est plus visualisable et exploitable par les ingénieurs. A titre de comparaison, nous avons fourni dans la même figure, la représentation du même composant Object Modeler avec cette fois notre syntaxe graphique (cf. section 6.1 et annexe E). Vous conviendrez sûrement comme nous qu'elle est plus lisible et intuitive.
De plus, nous ne représentons ici qu'un seul composant Object Modeler. Sachant qu'actuellement il existe environ 8 000 composants Object Modeler (simplement pour la structure logique), il faut disposer de mécanismes de visualisation évolués comme des représentations en graphe ou en arbre. Là non plus, la notation UML n'est pas appropriée pour réaliser ce type de visualisation.

- **La non disponibilité d'outils** : en plus de ces problèmes liés à la notation UML, nous en avons rencontré d'autres liés au manque d'outils possédant des fonctionnalités évoluées. Les outils existants n'offrent pas ou peu de fonctionnalités liées à la disposition automatique, à la navigation entre les éléments, à la recherche d'informations, à la vérification des contraintes OCL... Par exemple, le placement des éléments de la figure 5.4 a du être réalisé à la main pour une question de lisibilité car il n'existe actuellement pas d'outils qui permettent de le faire automatiquement. Les outils actuels ne permettent pas non plus de répondre à certains besoins des ingénieurs de Dassault Systèmes. Par exemple, à partir du composant Object Modeler CATAxisImpl de la figure 5.4, ils souhaitent pouvoir afficher la spécification de l'un des composants pères (comme CATBRepAccessSubElemImpl) par un simple clic souris sur le nom du composant cible ; ou encore connaître l'ensemble des implémentations pour l'interface Object Modeler CATLine... Ce manque d'outillage est indirectement un frein à l'utilisation de la notation d'UML.

Comme pour les travaux qui ont utilisé UML pour réaliser leurs descriptions architecturales, nos expérimentations ont confirmé les difficultés et lacunes décrites dans la section 5.1.3 pour utiliser cette notation[8].

Autant nous admettons volontiers qu'UML est un standard à suivre pour la modélisation objet, autant nous pensons qu'il n'est pas encore adapté pour décrire des architectures logicielles dans un contexte industriel. Par contre, il est important de pouvoir l'exploiter autant que possible et espérer que les futures spécifications d'UML prendront en compte ces limitations. Tous ces travaux qui ont tenté d'utiliser UML pour leur description architecturale peuvent servir de base pour intégrer un meilleur support pour la modélisation architecturale dans les spécifications futures d'UML (cf. préface de Cris Kobryn pour le livre Applied Software Architecture [HNS00]). Des conférences qui vont dans cette direction commencent à apparaître [KSK+01].

[8]Nos expérimentations ont été réalisées avant la connaissance de ces travaux qui ont été publiés à peu près au même moment ou même après.

Une instanciation du modèle conceptuel de l'IEEE ?

Lors de notre étude de l'architecture de CATIA V5, nous nous sommes rapidement rendus compte qu'il existait plusieurs structures pour cette architecture. Les ingénieurs de Dassault Systèmes utilisent la notion de composant pour des contextes différents et il n'existe pas de définition commune. Un composant est une unité fonctionnelle pour les développeurs alors que c'est une unité de compilation pour les responsables de la construction du logiciel ("Release Manager") ou encore une unité de déploiement pour les responsables des produits. En fait, comme dans le cas de Kruchten et de Hofmeister et al., les composants et les connecteurs correspondent à des réalités différentes suivant les structures architecturales.

Une autre caractéristique importante de notre contexte est la taille du logiciel, le grand nombre d'entités à manipuler et la rapide évolution du système. Avec de telles contraintes, il n'est pas envisageable de décrire et analyser à la main les différents composants de CATIA V5. Il faut donc automatiser les différentes tâches de description et d'analyse. Pour cela, il faut définir un méta-modèle qui a l'avantage de formaliser clairement les concepts architecturaux. Ce méta-modèle sert ensuite de référence pour sauvegarder les données architecturales et pour les manipuler et les exploiter à travers différents outils [FDE+01]. Ceci correspond aussi au modèle conceptuel de l'IEEE pour qui une description architecturale comprend un ensemble de modèles et un ensemble de vues conformes à ces derniers.

Comme pour l'organisme de standardisation de l'IEEE, nous avons aussi ressenti le besoin de prendre en compte les facteurs liés au processus de développement dans lequel se place le système logiciel. Nous avons répertorié les différents acteurs ainsi que leurs besoins architecturaux et le lien avec les différentes structures architecturales que nous avons étudiés (cf. sections 5.3.1 et 6.2 et chapitre 7).

Nous en avons conclu que le modèle conceptuel de l'IEEE correspondait bien à nos besoins pour décrire l'architecture de CATIA V5. Comme pour Kruchten et pour Hofmeister et al., nos travaux sont une instanciation de ce modèle. Par contre les structures qui ressortaient de l'architecture de CATIA V5 n'étaient pas tout à fait les mêmes.

Lors de cette thèse nous avons étudié plus particulièrement trois structures de l'architecture V5 qui seront présentées de manière détaillée dans le chapitre suivant :

- **La structure logique** : Elle décrit la décomposition fonctionnelle du logiciel sous forme d'une configuration de composants et de connecteurs conceptuels. Cette architecture correspond à la vue conceptuelle dans les travaux de Hofmeister et al. [HNS00] ou encore la vue logique dans les travaux de Kruchten [Kru95].
- **La structure physique** : Elle décrit la structure physique (système de fichiers) pour l'implémentation du logiciel. Cette architecture est orientée pour la compilation et sert de base pour le déploiement. Elle correspond à la vue

architecturale du code pour les travaux de Hofmeister et al. et à la vue de développement pour les travaux de Kruchten.

- **La structure produit ou de packaging** : Elle s'occupe de la gestion pour le déploiement et pour les licences sous forme de produits hiérarchiques. Elle ne correspond à aucune autre vue architecturale dans d'autres travaux. Cela s'explique par le fait que CATIA n'est pas un produit à part entière mais plutôt une infrastructure sur laquelle il est possible de rajouter un certain nombre de produits pour répondre aux besoins des clients (cf. figure 3.1). Ceci permet aux clients de Dassault Systèmes de spécialiser CATIA à leur métier. Une autre explication provient du fait que cette structure permet aussi de répondre à des besoins de vente ("marketing"). Cet aspect économique n'a pas été abordé dans les travaux précédents.

Bien que nous ayons étudié principalement ces trois structures, nous savons qu'il en existe d'autres dans l'architecture V5. A la différence de Kruchten et de Hofmeister et al., nous sommes plutôt d'avis de ne pas fixer le nombre de ces structures car cela dépend beaucoup du contexte. Nous pensons aussi qu'il est intéressant de répertorier ces différentes structures qui peuvent être réutilisées dans d'autres projets.

5.3 Notre stratégie pour décrire l'architecture de CATIA V5

Même si nous n'avons pas pu réutiliser ces techniques de description architecturale telles quelles, ces différentes expérimentations nous ont permis d'élaborer une stratégie pour décrire l'architecture de CATIA V5. Les difficultés rencontrées dans l'utilisation de certaines d'entres elles comme les ADLs ou la notation UML, nous ont poussés à comprendre pourquoi nous ne pouvions pas les appliquer dans le contexte de Dassault Systèmes et à trouver des solutions pour y remédier. Pour cela, nous avons cherché à comprendre la signification et l'intérêt d'une description architecturale.

Pour notre part, une description architecturale n'a de sens que par rapport à des besoins bien identifiés. Nous plaçons les besoins des différents acteurs au centre de notre approche. Une description architecturale est définie par rapport aux besoins qu'elle doit prendre en compte et non l'inverse. Notre approche n'a rien d'original mais demande une certaine méthodologie qui n'est généralement pas fournie avec les différentes techniques de description architecturale.

Concrètement, notre approche se résume en quatre points qui peuvent suivre un cycle de vie en spirale (ce que nous avons d'ailleurs fait) :

1. La première étape est d'identifier les besoins des différents acteurs (cf. section 5.3.1).

2. Cette analyse des besoins permet de choisir le bon niveau d'abstraction (cf. section 5.3.2).
3. Parallèlement à ces deux premières étapes, il est important de bien comprendre l'architecture du système. Cette étude doit permettre d'identifier les concepts architecturaux pertinents et de réaliser une modélisation de l'architecture du système. Elle doit prendre en compte les besoins des acteurs définis dans la première étape et vérifier que le niveau d'abstraction choisi dans la deuxième étape est techniquement réalisable (cf. section 5.3.3).
4. Enfin, il ne reste plus qu'à réaliser concrètement les vues architecturales et les outils associés pour répondre aux besoins initiaux (cf. section 5.3.4).

Une partie importante de cette thèse a été consacrée à comprendre les besoins architecturaux des acteurs de Dassault Systèmes ainsi que la compréhension et la modélisation de l'architecture V5.

5.3.1 Les besoins architecturaux des acteurs de Dassault Systèmes

Pour bien comprendre les besoins architecturaux des acteurs de Dassault Systèmes, nous avons passé beaucoup de temps à discuter avec eux et à analyser leurs méthodes de travail pour identifier les problèmes et les besoins associés qu'ils rencontraient. Nous avons en outre pu identifier plusieurs types d'acteurs par rapport à leurs objectifs et leurs intérêts sur l'architecture V5. Le chapitre suivant les présentera plus en détail.

Le domaine d'activité de Dassault Systèmes fait que ses principaux besoins architecturaux ne se situent pas dans la construction d'un logiciel sûr[9]. Bien que CATIA nécessite une grande qualité (n'oublions pas que la plupart des avions, voitures... sont réalisés avec CATIA) pour que les pièces conçues correspondent bien au souhait du concepteur et aux normes en vigueur, ce n'est pas pour autant un logiciel critique (dans le sens où si CATIA doit s'arrêter à cause d'une erreur quelconque, il n'y a pas de risques majeurs).

Les ingénieurs de Dassault Systèmes doivent faire face à d'autres problèmes qui sont plus importants pour eux. La qualité du logiciel est un facteur important (de nombreux tests sont réalisés par des tests unitaires et des testeurs) mais ce n'est pas le seul. Les performances en temps d'exécution ainsi que la facilité de développement et d'évolution du logiciel[10] sont des critères aussi essentiels que la qualité du logiciel. La taille et la complexité de CATIA V5 posent de nombreux problèmes difficiles à résoudre pour prendre en compte ces derniers critères. Les mécanismes offerts par l'architecture V5 (architecture ouverte) et le développement

[9] C'est-à-dire que, d'après les facteurs de qualité définis par Meyer [Mey97], la robustesse n'est pas une qualité essentielle pour le logiciel CATIA à comparer des qualités de validité et d'extensibilité.

[10] N'oublions pas que Dassault Systèmes produit une nouvelle release tous les 4 mois.

fortement concurrent (plus de mille ingénieurs) font qu'il n'est pas évident de connaître l'architecture de CATIA V5 et de savoir quels sont les impacts suite à une modification. Ces informations ne peuvent pas être calculées à la main et il faut donc disposer d'outils automatiques [San00].

Les besoins architecturaux des acteurs de Dassault Systèmes se situent actuellement plus au niveau d'une aide au développement liée à l'utilisation de l'architecture V5 et à l'analyse d'impact. Nous sommes attachés à fournir des solutions à ces besoins. Des exemples concrets sur ces besoins seront fournis dans la troisième partie de ce rapport de thèse.

5.3.2 Une description abstraite pour qui ? Pour quoi ?

Nos expériences avec les ADLs ont montré que le niveau de decription ne correspondait pas aux besoins de Dassault Systèmes. Il a donc fallu trouver le bon niveau d'abstraction. La figure 5.5 présente un aperçu des différents langages d'abstraction (diagramme de classes d'UML [BRJ99, RJB99], MILs [DK76, PDN86], CDLs (CCM [OMG, Rui]), ADLs [Gar00, MT00], Langages Formels [Lam00]), leurs objectifs ainsi que le type de logiciel auquel ils sont destinés. Ces langages d'abstraction sont utilisés pour décrire différents types de concepts (code, module, composant...) et couvrent différents types d'informations de haut niveau (structures, dépendances, comportement, fonctionnalités). Selon les informations que nous possédons, il est possible de réaliser différentes analyses et de développer différents outils couvrant une grande palette de besoins. D'un côté, plus l'information est riche sémantiquement, plus les analyses seront précises et élaborées. Mais il va de pair que ces informations sont les plus difficiles à extraire et à décrire et qu'il n'est pas toujours possible de le faire automatiquement. D'un autre côté, des analyses simples peuvent fournir des bénéfices utiles et suffisants (cf. partie III).

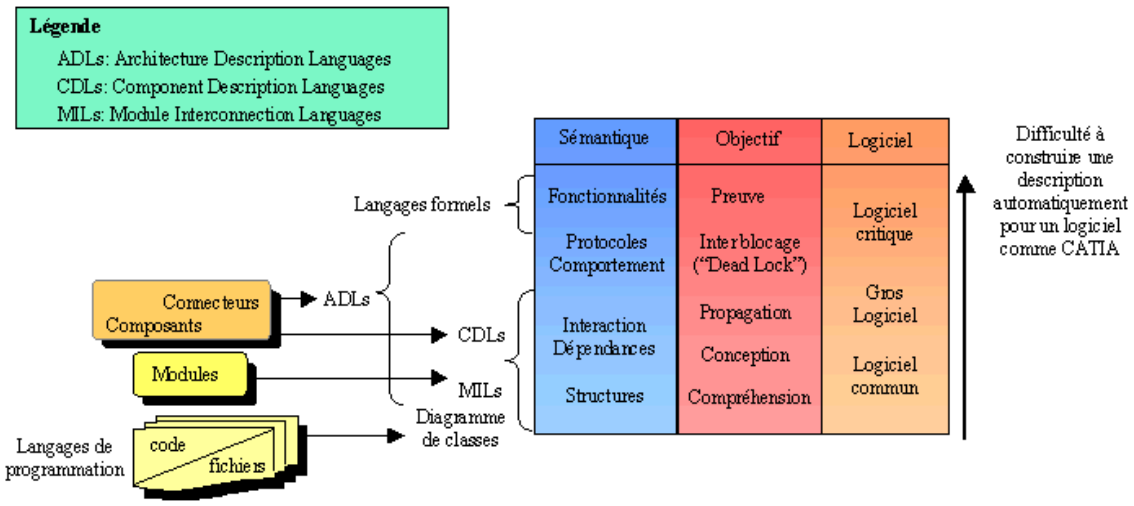


FIG. 5.5 – Aperçu des langages d'abstraction

Le contexte de Dassault Systèmes nous imposait d'avoir un lien constant entre les descriptions architecturales et le code source. Cela demande donc de disposer d'outils automatiques pour l'extraction des informations architecturales nécessaires à partir du code source. Notre niveau d'abstraction était donc limité mais heureusement suffisant pour prendre en compte les besoins de Dassault Systèmes qui nous intéressaient (cf. figure 5.6).

5.3.3 Compréhension et modélisation de l'architecture V5

Cette étude plus technique est essentielle car, comme le souligne Bass et al. [BCK98] et Hofmeister et al. [HNS00], l'architecture logicielle et le fonctionnement de la société sont très liées et s'influencent mutuellement. Il est donc important de réaliser cette étape parallèlement aux précédentes car elle permet de mieux comprendre les besoins des acteurs. De plus cela permet d'être conscient des limitations techniques auxquelles nous risquons d'être confrontés lors de l'abstraction. L'influence mutuelle entre l'architecture logicielle et la culture d'entreprise sera mise en évidence au chapitre suivant dans le cas de Dassault Systèmes.

Pour bien comprendre l'architecture V5, nous avons discuté longuement avec les ingénieurs de Dassault Systèmes et développé quelques exemples utilisant les différents types de construction disponibles dans cette architecture. Nous avons cherché à identifier les concepts architecturaux pertinents en distinguant les aspects conceptuels des aspects de programmation. Suite à cette compréhension, nous avons commencé à modéliser l'architecture V5 et plus particulièrement l'Object Modeler. Pour cela, nous avons utilisé la notation UML et réalisé une modélisation à trois niveaux : le niveau conceptuel pour décrire les concepts de l'architecture V5, le niveau réalisation pour indiquer des informations liées à l'implémentation et le niveau code pour faire le lien avec les instructions des fichiers C++ et textuels. Ceci a permis de bien clarifier les idées et de faire le lien entre le code source et un niveau conceptuel.

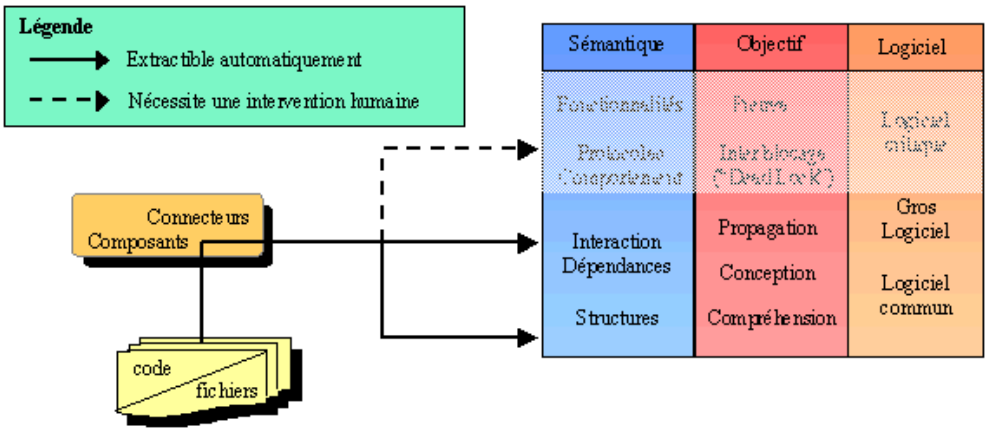


FIG. 5.6 – Le niveau d'abstraction dont nous avions besoin

Comme le montre la figure 5.6, nous avons pu extraire automatiquement du code source les informations statiques des composants et connecteurs des structures architecturales que nous avons étudiées. A l'image des travaux de Kruchten [Kru95], nos connecteurs sont de simples liens de dépendance qui nous permettent de calculer les analyses souhaitées. Les concepts et les méta-modèles des trois structures architecturales seront présentés en détail dans le chapitre suivant.

Actuellement, si nous voulions réaliser des analyses plus poussées, il faudrait compléter les méta-modèles en conséquence et extraire interactivement du code les informations manquantes.

5.3.4 Réalisation des vues architecturales

La réalisation concrète des différentes vues architecturales s'est faite en deux temps :

1. **Définition d'une syntaxe graphique** : pour décrire les instances des concepts de l'architecture V5, nous avons défini notre propre syntaxe graphique (nous avons vu dans la section 5.2.1 que la notation UML n'était pas très appropriée). Ce choix s'est révélé judicieux car la syntaxe graphique qui a été définie a permis de représenter simplement des informations complexes (cf. figure 5.4). Cela a aussi facilité le bon accueil de nos prototypes par les ingénieurs de Dassault Systèmes. Cette syntaxe graphique est basée sur trois notations : la notation UML puisqu'il est le standard de la modélisation objet et est utilisé chez Dassault Systèmes, la notation utilisée par Microsoft pour représenter les composants COM car l'Object Modeler a des similitudes avec ce modèle de composants et que cette notation a l'avantage d'être pratique, connue et largement répandue ; et les notations trouvées dans les documents de Dassault Systèmes car elles sont connues et utilisées par les ingénieurs de Dassault Systèmes et ses clients. Nous avons donc tiré parti de la culture d'entreprise ce qui a permis d'exploiter cette syntaxe graphique naturellement dès sa présentation. La conception de cette syntaxe graphique devait aussi répondre à d'autres contraintes : faciliter la communication "humaine", simple à apprendre et à utiliser (80% des conceptions se documentent avec 20% des constructions), exploitable par des outils informatiques ou non (tableau, papier...)... Cette syntaxe graphique sera présentée dans la section 6.1 et reprise dans l'annexe E.
2. **Développement d'un ensemble de prototypes** : A ce stade nous avions tout ce dont nous avions besoin pour développer des prototypes. Les outils que nous avons réalisés sont naturellement orientés pour répondre aux besoins architecturaux de Dassault Systèmes que nous voulions couvrir. Nous possédons actuellement un outil pour la visualisation et la navigation entre les différentes entités de l'architecture de CATIA V5 ; un outil pour la visualisation et l'exploration de cette architecture et un outil de simulation d'analyse d'impact. Le développement de ces prototypes devait, en plus de répondre à des problèmes concrets, prendre en considération des contraintes techniques (liaison constante avec le code source) et des contraintes humaines (facilité

d'utilisation et de compréhension). Ces différents prototypes seront présentés dans la troisième partie.

5.3.5 Pour conclure

Nous ne préconisons pas de suivre à la lettre nos résultats (les structures architecturales que nous avons retenues ne correspondront pas à tous les contextes). Au contraire, nous pensons qu'il n'existe pas de solution toute faite mais plutôt une approche globale à adapter suivant son contexte. Ceci découle du fait que ce sont les besoins des acteurs qui motivent le choix des descriptions architecturales à produire. De plus l'architecture logicielle et l'organisation de la société sont très liées et s'influencent mutuellement. Il est donc important de bien étudier ces deux aspects afin de proposer des solutions adaptées aux besoins des acteurs de la société.

Le contexte dans lequel s'inscrivent les recherches doit aussi être pris en compte. En ce qui nous concerne, nous étions confrontés à un gros logiciel existant en perpétuelle évolution. Mais ce contexte n'est pas propre à Dassault Systèmes, pour cela il suffit d'observer la longévité des logiciels et le nombre de systèmes existants sans description architecturale. Les principaux travaux publiés dans le domaine de l'architecture logicielle se sont au contraire attachés à décrire des solutions pour la conception initiale d'un logiciel. Par cette thèse, nous nous sommes attachés à fournir une expérience industrielle pour la description architecturale d'un logiciel en perpétuelle évolution. Il reste encore de nombreuses idées à explorer et à formaliser concernant l'architecture logicielle et l'évolution des logiciels (cf. chapitre 9).

Bien que nous ne prônions pas de solution toute faite, nous pouvons émettre certains conseils : comprendre les besoins des acteurs et le fonctionnement de la société, trouver le bon niveau de description, ne pas sous estimer l'intérêt des descriptions et analyses simples et choisir une syntaxe graphique facile à apprendre et à utiliser.

5.4 En résumé

Dans ce chapitre, nous avons tenté de mettre en évidence les points suivants :

- **Etat de l'art sur les différentes approches pour la description d'une architecture**. Nous avons décrit les principales approches qui ont été proposées par la communauté scientifique dans le domaine de l'architecture logicielle : les ADLs, les travaux qui prennent en compte plusieurs structures architecturales et l'utilisation du standard UML.
- **Utilisation de ces approches dans notre contexte avec Dassault Systèmes**. Nous avons exposé les difficultés que nous avons rencontrées lors de nos expérimentations pour utiliser certaines de ces approches.
- **Présentation de notre approche pour décrire l'architecture de CATIA V5**. Ces expérimentations ont permis de définir notre propre approche basée sur les besoins des différents acteurs.

Notre expérience nous pousse à croire qu'il n'existe pas vraiment de solution clé en main mais plutôt une démarche générale basée sur l'analyse des besoins. Néanmoins, il est important de pouvoir capitaliser au mieux les différentes expériences comme Kruchten, Hofmeister et al. et nos travaux ont cherché à le faire. En effet, s'il n'est généralement pas possible de réutiliser l'ensemble des résultats d'une expérience, il est probable qu'une partie puisse être utile pour son contexte. Par exemple, un architecte peut reprendre la vue processus de Kruchten et/ou la vue code de Hofmeister et al. pour répondre aux besoins de ses acteurs. C'est d'ailleurs ce que préconise la norme ANSI/IEEE Std P1471-2000 où il est possible de capitaliser les différentes structures architecturales dans une librairie de point de vue.

Il est intéressant de souligner qu'une originalité de nos travaux est d'être parti directement du code source pour extraire automatiquement nos descriptions architecturales.

Chapitre 6

L'Architecture V5 et l'organisation de développement associée

Afin de répondre à ses besoins en termes de génie logiciel (cf. chapitre 3), Dassault Systèmes, lors de la refonte de son logiciel CATIA pour le passage à la version 5, a défini une architecture spécifique appelée l'architecture V5. Son développement a suivi un processus incrémental et de nouveaux concepts ont été rajoutés au fur et à mesure des besoins. Certains besoins propres à Dassault Systèmes ont permis de faire ressortir et dissocier des concepts qui ne sont pas disponibles dans d'autres architectures. Comme nous allons le décrire dans ce chapitre, cette architecture est originale sur plusieurs points (extension Object Modeler, lien entre les différentes structures architecturales...).

Pour pouvoir décrire l'architecture de CATIA V5, une partie de notre travail a été de comprendre et de conceptualiser cette architecture V5. Cela nous a amené à étudier l'organisation de développement et identifier les différents acteurs de Dassault Systèmes. En effet comme le précisent Bass et al. dans leur cycle "Architecture Business Cycle" [BCK98], l'organisation de la société influe et est influencée par l'architecture.

Ce chapitre a pour but de présenter l'architecture V5 et l'organisation de développement de Dassault Systèmes. La section 6.1 présente la modélisation de l'architecture V5 que nous avons réalisée. La section 6.2 décrit l'organisation de Dassault Systèmes ainsi que ses différents acteurs. Cette dernière section met en évidence l'influence mutuelle entre l'architecture V5 et l'organisation de développement de Dassault Systèmes.

6.1 Modélisation de l'architecture V5

Comme nous l'avons déjà exposé dans le chapitre précédent, l'architecture V5 possède plusieurs structures architecturales qui intéragissent. Dans cette thèse, nous avons pu en étudier trois : l'architecture logique (cf. section 6.1.1), l'architecture physique (cf. section 6.1.2) et l'architecture produit (cf. section 6.1.3).

Cette section décrit de manière détaillée ces trois structures architecturales ainsi que leurs interactions. Pour chacune de ces structures architecturales, nous présenterons les besoins qu'elle satisfait et son méta-modèle.

La réalisation de ces méta-modèles a représenté une part importante de notre travail. Ils permettent de formaliser les concepts importants de l'architecture V5. Les méta-modèles que nous présentons ci-après sont volontairement simplifiés pour faciliter la lecture et la compréhension[1].

6.1.1 L'architecture logique

Présentation générale

L'architecture logique se concentre sur la conception des fonctionnalités du logiciel. Cette structure se matérialise par une configuration de composants et de connecteurs fonctionnels. Elle se rapproche de la vue conceptuelle des travaux de Hofmeister et al. et de la vue logique des travaux de Kruchten.

L'architecture V5 est organisée en couches, appelées modeleurs, à partir du langage de programmation C++ (cf. figure 6.1). Ces modeleurs offrent des services de haut niveau couvrant des besoins spécifiques allant du génie logiciel (Object Modeler) à des besoins métiers (Applications Modelers) en passant par la CAO/CFAO (Feature Modeler). Pour des questions de gains de performance, une couche n peut directement passer à une couche $n-2$ sans passer par la couche $n-1$.

Au cours de cette thèse, nous nous sommes particulièrement intéressés à l'Object Modeler car cette couche correspond au modèle de composants pour l'architecture logique de Dassault Systèmes. C'est elle qui définit les bases de l'architecture logique.

Présentation du méta-modèle de l'Object Modeler

L'Object Modeler a été principalement conçu pour répondre au besoin de Dassault Systèmes de posséder une architecture ouverte qui permet à ses clients d'étendre simplement CATIA V5 avec un minimum d'impact (cf. section 3.2). L'Object Modeler permet aussi de répondre à des besoins internes liés au développement fortement concurrent, à la facilité d'évolution et à des mécanismes liés à leur domaine d'activité.

L'Object Modeler fournit une large palette de concepts et de mécanismes pour l'implémentation de la partie fonctionnelle du logiciel à travers des composants [EFS02]. Ces concepts et mécanismes permettent de créer et d'assembler des composants de différentes manières. La figure 6.2 présente le méta-modèle pour l'Object Modeler.

[1] Une grande partie des figures de cette section sont en anglais car nous avons repris la terminologie utilisée à Dassault Systèmes.

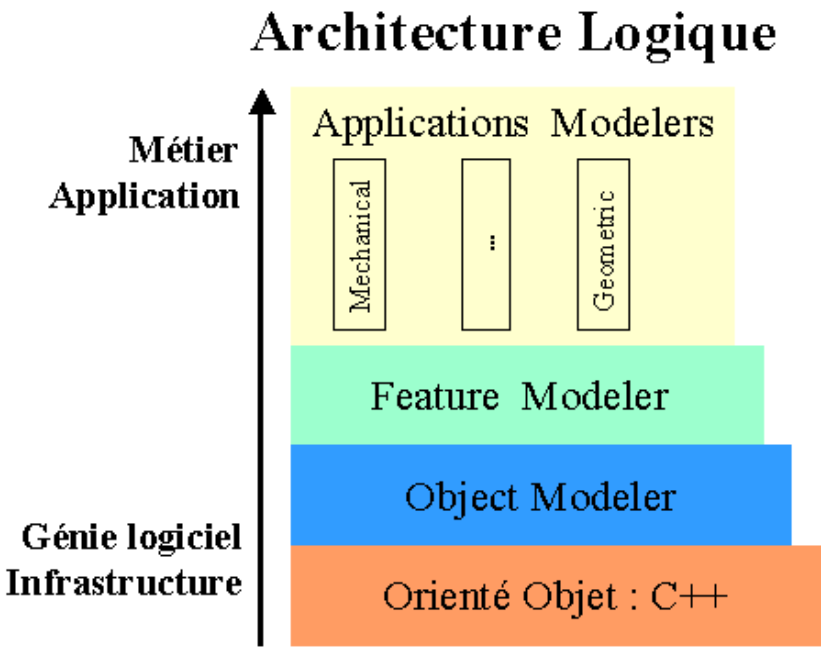


FIG. 6.1 – Les modeleurs de l'architecture logique

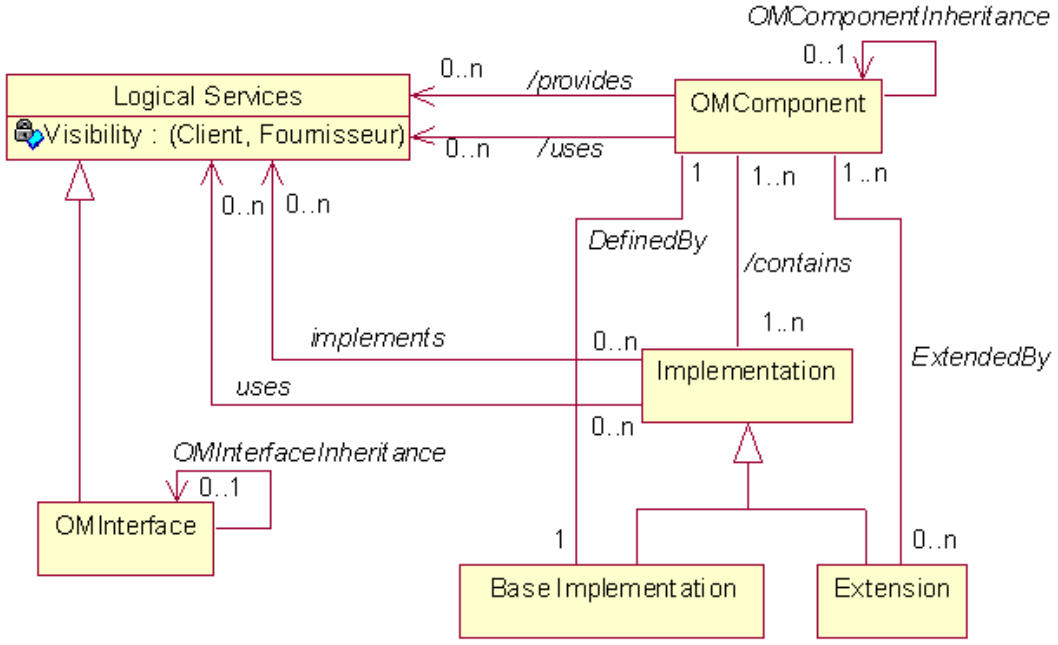
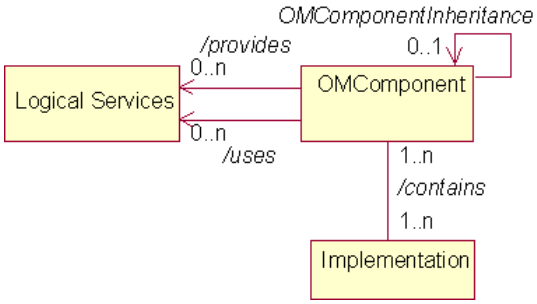
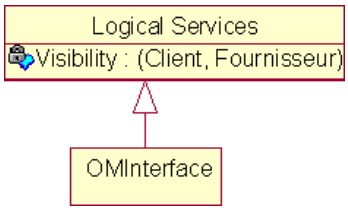


FIG. 6.2 – Le méta-modèle de l'Object Modeler

Composant Object Modeler : un composant Object Modeler est une entité qui est caractérisée par un nom et qui permet de regrouper un ensemble de codes élémentaires appelés implémentations. Un composant Object Modeler fournit et utilise un ensemble de services logiques par l'intermédiaire de ses implémentations et des implémentations de ses composants Object Modeler pères.

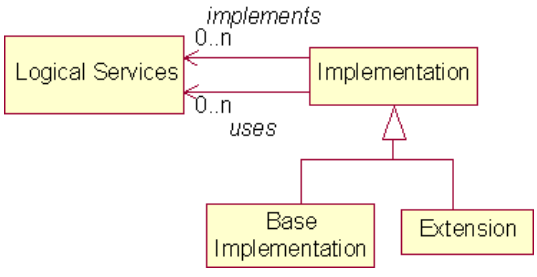


Services Logiques : un service logique est une entité qui spécifie et regroupe un certain nombre de fonctionnalités de l'application. Ces services peuvent prendre différentes formes comme des interfaces Object Modeler ou encore des événements[2]. Ce concept de service logique est similaire au concept de port dans CCM [Rui]. Comme les partenaires et certains clients peuvent développer des applications propres, Dassault Systèmes met à disposition les spécifications de certains de ces services logiques. Une propriété de visibilité permet de distinguer les services logiques qui ne sont accessibles que par Dassault Systèmes de ceux qui le sont aussi par ses clients lors du développement. L'attribut de visibilité permet ainsi de distinguer les services logiques qui ne doivent plus être modifiés par Dassault Systèmes. Dans la réalité, ce filtre sur les services logiques a plusieurs niveaux indiquant le degré de liberté de modification par Dassault Systèmes.



Interfaces Object Modeler : une interface Object Modeler est un type particulier de service logique. Elle joue le rôle d'intermédiaire entre le client et le composant Object Modeler et permet de retrouver la bonne implémentation lorsque le client demande une interface Object Modeler au composant Object Modeler. Un mécanisme appelé QueryInterface est associé à ce concept d'interface Object Modeler. Il est similaire à celui de COM [Box98] et permet de demander dynamiquement à un composant une référence sur une interface qu'il possède. C'est un mécanisme de navigation entre les interfaces du composant. Le concept d'interface Object Modeler est très utile puisqu'il permet de garantir l'indépendance du code, par le fait qu'il isole le client des modifications internes au composant Object Modeler.

Implémentation : une implémentation a pour objectif de réaliser un certain nombre de services logiques. Pour cela, une implémentation peut utiliser les services d'autres composants en utilisant le mécanisme de QueryInterface.



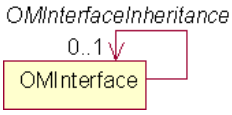
[2]Par souci de simplification, nous ne présentons ici que les interfaces Object Modeler.

Actuellement, il existe deux types d'implémentations :

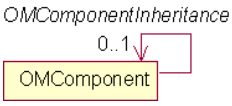
- **Les Bases Implémentations** : elles ont un rôle particulier, puisque ce sont elles qui définissent l'identité du composant, i.e. le nom du composant Object Modeler. C'est à partir de ce nom que les implémentations et les objets C++ accéderont au composant Object Modeler.
- **Les Extensions** : une extension permet d'étendre un ou plusieurs composants Object Modeler pour leur rajouter du comportement. Une caractéristique particulière est que les extensions connaissent le ou les composants qu'elles étendent mais ne se connaissent pas entre elles.

Héritages Object Modeler : Il existe deux concepts d'héritage Object Modeler qui diffèrent de l'héritage orienté objet :

- **L'héritage Object Modeler d'interface** : cet héritage indique que si une implémentation *impl1* implémente une interface Object Modeler, elle doit aussi implémenter les interfaces Object Modeler mères de cette interface. L'implémentation possèdera l'ensemble de ces interfaces Object Modeler (l'interface fille et les interfaces mères). Par conséquence, si un client demande par le mécanisme du QueryInterface une interface Object Modeler mère de cette implémentation *impl1*, une référence sur *impl1* lui sera retournée.



- **L'héritage Object Modeler de composant** : si un composant Object Modeler C1 hérite au sens héritage Object Modeler de composant d'un composant Object Modeler C2, cela signifie que C1 récupère l'ensemble des services logiques fournis et implémentés par le composant C2 et par transitivité par l'ensemble des composants pères de C2. Par conséquence, si un client demande par le mécanisme du QueryInterface une interface Object Modeler fournie par un de ses composants père, le composant Object Modeler C1 lui retournera une référence sur la bonne implémentation.



Nous n'avons présenté que les concepts les plus importants et utiles pour ce rapport de thèse. Mais en réalité, il en existe beaucoup d'autres comme l'adhésion conditionnelle, la délégation...

La création de l'Object Modeler a suivi un processus incrémental et ces mécanismes sont apparus au fur et à mesure des besoins. Même avec peu de concepts, il est difficile d'intégrer un nouveau concept suite à l'effet combinatoire des interactions entre ces mécanismes et par le fait que les concepts initiaux n'ont pas forcément été conçus pour intéragir avec le dernier. Par exemple, nous n'avons donné qu'une explication simplifiée des héritages Object Modeler. En réalité, ces concepts sont beaucoup plus complexes : l'utilisation simultanée de plusieurs mécanismes engendre de nombreux cas particuliers. De plus ces héritages Object Modeler cohabitent avec l'héritage C++. L'accumulation de ces facteurs fait que la sémantique de ces héritages Object Modeler est très difficile à définir. Le développement fortement concurrent et la taille de l'application font que l'utilisation

de ces héritages Object Modeler est difficile à maîtriser et est à l'origine d'une grande partie des erreurs. Le chapitre suivant montrera la nécessité de posséder un outil automatique pour diagnostiquer ce type d'erreur.

Une particularité de l'Object Modeler est que l'assemblage des entités Object Modeler se fait plus par construction que par connexion. De nombreux mécanismes sont fournis pour faciliter la construction et l'évolution d'un composant comme les héritages Object Modeler ou l'extension alors que la connexion entre deux composants se fait principalement par le mécanisme du QueryInterface. Si un client souhaite rajouter une nouvelle fonctionnalité, il va utiliser le mécanisme d'extension plutôt que de créer un nouveau composant et de le connecter aux autres composants. Ceci permet de rajouter de nouvelles fonctionnalités sans perturber l'architecture de CATIA V5. Cette particularité est à l'origine des difficultés que nous avons rencontrées pour identifier une notion de connecteur. A l'image de Kruchten, nos connecteurs représentent des liens de dépendance reflétant ces mécanismes de construction.

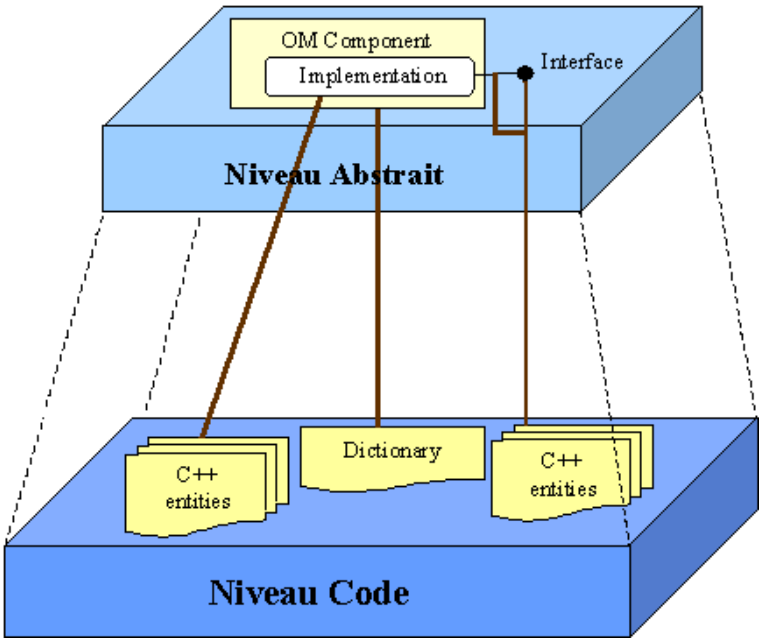


FIG. 6.3 – Lien entre le niveau abstrait et le niveau code pour l'Object Modeler

Lien entre le méta-modèle et le niveau code

Du point de vue réalisation, tous les concepts fournis par l'Object Modeler sont implémentés par des entités C++ et des fichiers textuels (cf. figure 6.3). Par exemple, les interfaces et les implémentations sont représentées par des classes C++. En fait, le niveau réalisation est assez complexe du fait que la correspondance n'est pas une bijection : plusieurs fichiers C++ et textuels sont souvent nécessaires pour réaliser une seule entité Object Modeler.

De plus, pour des raisons de performance et des considérations de nature

non-fonctionnelle, un composant peut être implémenté de différentes manières. Dassault Systèmes a conçu et testé une large palette de techniques permettant de construire des composants efficaces. Du même coup, le développement et la maintenance de ces composants demandent une expérience certaine.

Afin de garder le contrôle sur le logiciel résultant, les concepts Object Modeler sont transcrits en C++ par des patrons ou des conventions de nommage. Cette approche est très similaire à celle utilisée par les JavaBeans de Sun [Eng97, Sun]. Afin de faciliter la création d'entités Object Modeler, des macros sont fournies pour le code répétitif et certains fichiers sont générés automatiquement.

Il est intéressant de souligner que la notion d'extension est plutôt simple d'un point de vue abstrait mais que la technologie sous-jacente est très puissante et offre de nombreux avantages. Attention, cette notion n'est pas à confondre avec l'agrégation COM [Box98] qui lui ressemble d'un point de vue abstrait mais qui est loin d'être aussi performante d'un point de vue technique. L'extension permet à quiconque (autant aux ingénieurs de Dassault Systèmes qu'à ses clients) de rajouter des fonctionnalités sans avoir besoin du code source.

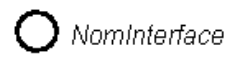
Des informations complémentaires sont fournies dans des fichiers textuels appelés dictionnaires, contenant des tuples "composant - interface - dll". Ces fichiers permettent de localiser et lire de façon incrémentale les librairies nécessaires pendant l'exécution ce qui permet un gain de performance important.

Syntaxe graphique associée

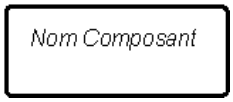
La syntaxe graphique que nous avons définie et exploitée pour décrire les entités de l'Object Modeler est basée sur trois notations :

- La notation UML puisqu'il est le standard de la modélisation objet et est utilisée chez Dassault Systèmes,
- La notation de Microsoft pour représenter les composants COM car l'Object Modeler a des similitudes avec ce modèle de composants et que cette notation a l'avantage d'être pratique, connue et largement répandue,
- Les notations trouvées dans les documents de Dassault Systèmes car elles sont connues et utilisées par les ingénieurs de Dassault Systèmes et ses clients.

Une interface Object Modeler est représentée par un cercle avec le nom de l'interface à côté. Cette représentation est similaire à la notation COM.



Un composant Object Modeler est représenté par un rectangle avec des coins arrondis. Le nom du composant est affiché à l'intérieur du composant.



Une implémentation Object Modeler (base implémentation ou extension) est représentée par un rectangle contenant le nom de l'implémentation. Généralement, il est possible de distinguer une base implémentation d'une extension selon le contexte. Par exemple, dans la vue interne d'un composant Object Modeler (cf. figure 6.5), la base implémentation est toujours l'implémentation qui est décrite en premier et elle a le même nom que le composant Object Modeler. De plus, nous utilisons des couleurs différentes pour faciliter la distinction : pour les bases implémentations nous utilisons la couleur noire qui est la même que pour les composants Object Modeler alors que les extensions sont dessinées avec la couleur bleue. Si le contexte ne permet pas de faire la distinction, il est possible de rajouter un stéréotype (comme pour la notation UML) devant le nom de l'implémentation : <<Base Implémentation>> ou <<Extension>>.

L'héritage Object Modeler de composant est représenté par un triangle à l'intérieur d'un carré. Le triangle correspond à la notation UML pour l'héritage objet et le carré symbolise le composant Object Modeler.

L'héritage Object Modeler d'interface est représenté par un triangle à l'intérieur d'un cercle. Le triangle correspond à la notation UML pour l'héritage objet et le cercle symbolise l'interface Object Modeler.

L'implémentation d'une interface Object Modeler par une entité Object Modeler (implémentation Object Modeler ou composant Object Modeler) est représentée par un trait continu qui va relier l'interface Object Modeler à l'entité Object Modeler qui l'implémente.

La figure 6.4 récapitule la syntaxe graphique pour décrire les entités de l'Object Modeler.

Dans notre syntaxe graphique, nous n'avons pas pris en compte les interfaces Object Modeler qui sont utilisées par un composant Object Modeler ou par une implémentation Object Modeler. Ceci provient du fait que nous ne disposions pas de ces données car il était difficile techniquement de les récupérer automatiquement à partir du code source. Néanmoins, il est possible de réutiliser la notation proposée pour le modèle de composants CCM [Rui] pour représenter ces informations. Dans la notation de CCM, une interface requise par un composant est représentée par un demi cercle symbolisant un réceptacle pour cette interface.

Utilisation de la syntaxe graphique

Dans la figure 6.5, nous présentons un exemple d'utilisation de cette syntaxe graphique avec notre prototype OMVT (cf. section 7.1). Dans cet exemple, nous décrivons le composant Object Modeler CATAxisImpl par deux vues différentes : la vue interne qui permet de visualiser comment est construit le composant Object

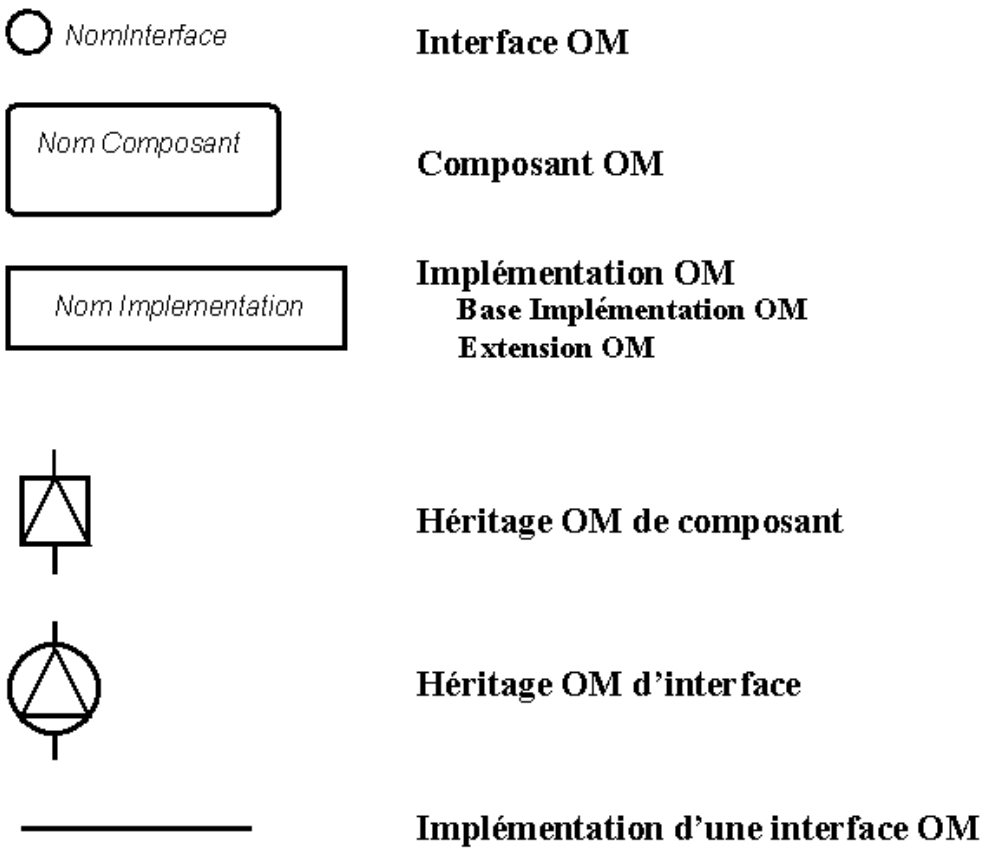


FIG. 6.4 – Syntaxe graphique pour l'Object Modeler

Modeler et la vue externe qui affiche l'ensemble des interfaces Object Modeler fournies par le composant Object Modeler.

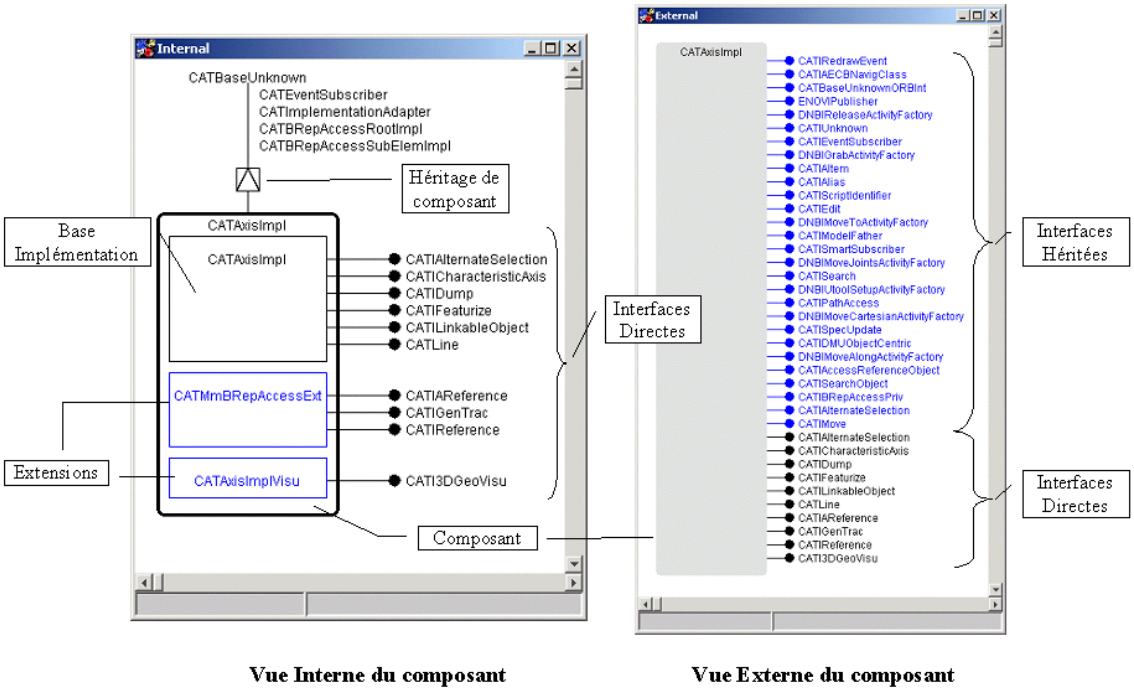


FIG. 6.5 – Vues interne et externe pour le composant Object Modeler CATAxisImpl

Dans la vue interne nous représentons le composant Object Modeler avec les différentes implémentations qui le constituent. Ces différentes implémentations sont affichées à l'intérieur du composant Object Modeler ne laissant dépasser de ce dernier que les interfaces Object Modeler qu'ils fournissent. La base implémentation Object Modeler associée au composant Object Modeler est décrit en premier suivi des extensions Object Modeler de ce dernier. Dans cet exemple, nous avons spécifié un héritage Object Modeler de composant où l'ensemble des super composants Object Modeler sont indiqués par ordre d'ascendance.

Dans la vue externe nous représentons le composant Object Modeler comme une boîte noire avec l'ensemble des interfaces Object Modeler qu'il propose. Dans cette vue, il est possible de distinguer les interfaces Object Modeler qui sont héritées des super composants Object Modeler (interfaces qui sont dessinées en bleu et qui sont listées en premier à partir du haut du composant) des interfaces Object Modeler proposées par le composant Object Modeler (interfaces qui sont dessinées en noir et qui sont décrites à la suite des interfaces héritées).

6.1.2 L'architecture physique

Présentation du méta-modèle

L'architecture physique se concentre sur les besoins de Dassault Systèmes pour la construction de CATIA V5 (cf. section 3.1.2). Elle permet aussi de répondre à des besoins liés au développement concurrent en structurant l'implémentation du logiciel par rapport à l'organisation de développement. Elle se rapproche de la vue code des travaux de Hofmeister et al. et de la vue développement des travaux de Kruchten.

L'architecture physique décrit le système de fichiers et les dépendances de compilation liées à l'implémentation du logiciel. Ces informations sont exploitées par le compilateur de haut niveau associé à l'architecture V5 pour optimiser et automatiser au mieux la construction du logiciel. La figure 6.6 présente le méta-modèle pour l'architecture physique.

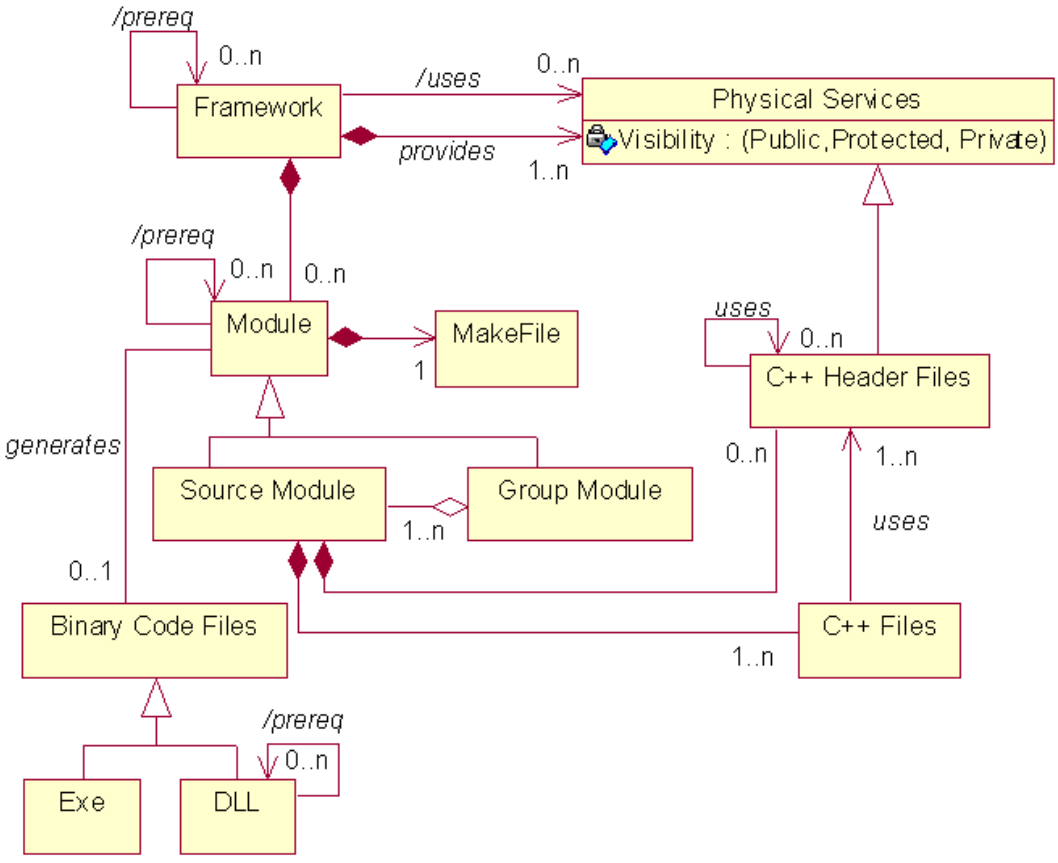
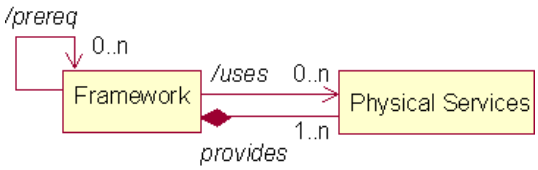
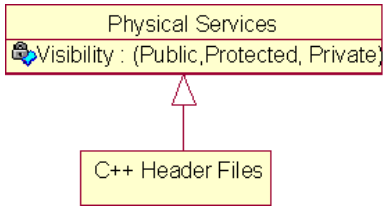


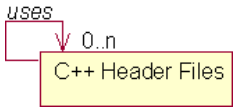
FIG. 6.6 – Le méta-modèle de l'architecture physique



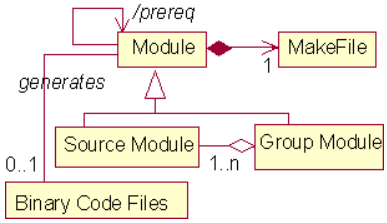
Les Frameworks : un framework est un composant physique qui permet de regrouper des entités Object Modeler délivrant des services similaires. Par exemple, les entités Object Modeler (interfaces, extensions) de visualisation sont regroupées dans un même framework. Cela permet de structurer l'implémentation du logiciel non plus par des composants Object Modeler mais par des domaines d'activités. D'ailleurs, il est intéressant de noter que ces frameworks reflètent l'organisation des équipes de développement de Dassault Systèmes puisque chaque équipe est spécialisée dans un domaine particulier. Chaque framework est géré par une équipe et une seule. Un framework fournit et utilise un ensemble de services physiques. Pour chaque framework, les relations de dépendance de compilation avec les autres frameworks (caractérisées par la relation "prereq" entre les frameworks dans le méta-modèle) sont repertoriées dans ce qui est appelé la carte d'identité du framework. Ces informations sont nécessaires pour le bon déroulement du processus de construction du logiciel.



Les services physiques : un service physique met à disposition un certain nombre de fichiers accessibles par les frameworks. Ces services peuvent prendre différentes formes mais sont essentiellement des en-têtes C++. Ces services physiques ont trois niveaux de visibilité : les services privés qui ne sont disponibles que par le framework en question, les services protégés qui ne sont accessibles que par les frameworks de Dassault Systèmes et les services publics qui sont utilisables par tout le monde (Dassault Systèmes, les partenaires et les clients).

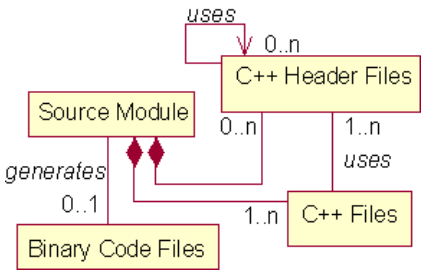


Les en-têtes C++ : un en-tête C++ est un fichier textuel qui contient la spécification fonctionnelle d'une interface C++ ou d'une interface Object Modeler. Il existe des liens de dépendance entre ces en-têtes C++ par le fait qu'une interface peut avoir besoin d'une autre interface lors de la compilation (par exemple lors d'un héritage ou de la déclaration d'une interface Object Modeler).

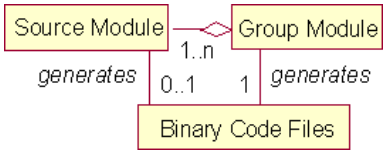


Les modules : pour gérer le grand nombre de fichiers et le travail au sein d'une équipe, un framework possède un certain nombre de modules. Un module possède un fichier MakeFile nécessaire pour la création du logiciel et de son éventuel fichier binaire. Il existe deux types de modules :

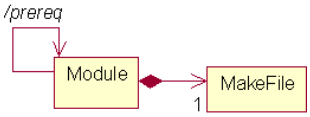
- **Les modules de code source** : ces modules sont dans un certain sens similaire au framework mais avec un granularité plus petite. Ils peuvent déclarer des interfaces C++ qui ne sont visibles qu'à l'intérieur du module. Leur principal rôle est de regrouper des fichiers C++, ce qui permet d'appliquer certaines opérations sur cet ensemble de fichiers C++ particuliers. Un module de code source peut générer un fichier binaire mais cela n'est pas obligatoire.



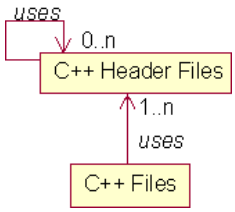
- **Les modules de groupe** : Un module de groupe rassemble des modules de code source de son framework. Ces modules de groupe ont été ajoutés pour faciliter la maintenance des fichiers binaires. Par exemple, ils permettent de réduire le nombre de DLLs et facilitent leur renommage. A la différence d'un module de source, un module de groupe génère forcément un fichier binaire.



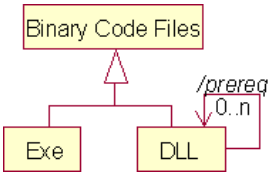
Le MakeFile : Un makefile est un fichier utilisé lors de la création du logiciel. Il déclare des informations liées aux options de compilation, aux options pour le pré-processeur, aux liens de dépendance avec d'autres modules pour l'édition de liens. Dans le cas des modules de groupe, il définit aussi le type du fichier binaire à générer.



Les fichiers C++ : un fichier C++ est un fichier textuel dont le code source est une classe C++ ou une implémentation Object Modeler. De même que pour les en-têtes C++, il existe des liens de dépendance entre les fichiers C++ et les en-têtes C++ par le fait qu'une classe C++ peut avoir besoin d'un en-tête C++ pour compiler (par exemple lors d'un héritage ou encore de l'implémentation d'une interface Object Modeler).



Les fichiers binaires : Les fichiers binaires sont utilisés lors de l'exécution de CATIA V5. Ils sont générés lors de la construction du logiciel. Il existe deux types de fichiers binaires : les fichiers exécutables et les librairies dynamiques (DLLs).



Lien entre le méta-modèle et le niveau code

D'un point de vue réalisation, un framework est un arbre de fichiers et de répertoires qui suivent des conventions de nommage et des patrons de conception de façon à ajouter de la sémantique (cf. figure 6.7). Par exemple, la visibilité des services physiques est définie par le nom du répertoire (PublicInterfaces, ProtectedInterfaces ou PrivateInterfaces) dans lequel ils sont situés.

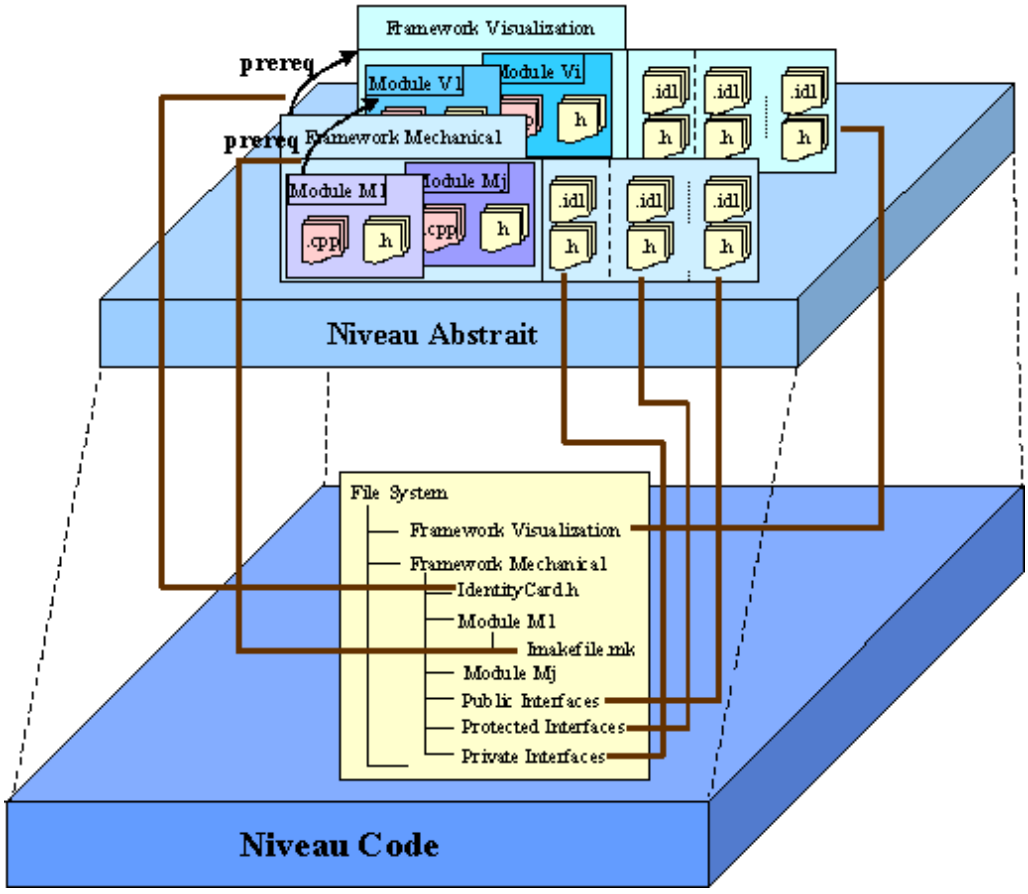


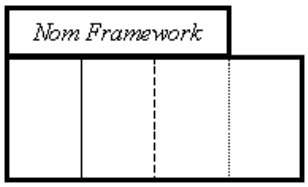
FIG. 6.7 – Lien entre le niveau abstrait et le niveau code pour l'architecture physique

La carte d'identité du framework est un fichier textuel ("IdentityCard.h") qui joue plus ou moins le rôle d'interface pour le framework du fait qu'elle contient les caractéritiques du framework et les liens de dépendance de compilation avec les autres frameworks (la relation "prereq" dans la figure 6.7).

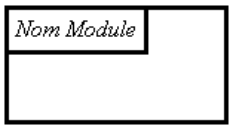
Le Makefile qui définit des informations de compilation pour un module est un fichier spéficique appelé "Imakefile.mk" qui se trouve dans le répertoire qui représente ce module. Ces fichiers ainsi que la carte d'identité du framework sont exploités pour le compilateur haut-niveau associé à l'architecture V5 lors de la création du logiciel.

Syntaxe graphique associée

La syntaxe graphique associée à l'architecture physique est basée sur la notation UML et sur les notations trouvées dans les documents de Dassault Systèmes.



Un framework est représenté par un dossier où le nom du framework est inscrit dans l'onglet en haut à gauche. Cette notation est similaire à la notation UML pour les paquetages. A l'intérieur du rectangle, il est possible de distinguer quatre parties qui définissent le degré de visibilité des données contenues dans chaque partie. Le degré de visibilité est défini par la position des parties dans le rectangle. Plus la partie est à gauche moins elle est visible et inversement plus elle est à droite et plus elle est visible de l'extérieur du framework. La partie la plus à gauche permet de représenter les modules dont leurs fichiers ne sont visibles qu'à l'intérieur de chaque module. La partie suivante regroupe les interfaces privées au framework qui ne sont visibles qu'à l'intérieur du framework. La troisième partie contient les interfaces protégées qui ne sont visibles que par les frameworks de Dassault Systèmes. Enfin la partie la plus à droite représente les interfaces publiques qui sont visibles par tout le monde (Dassault Systèmes et ses clients).



Un module est aussi représenté par un dossier mais à la différence du framework, l'onglet qui permet d'inscrire le nom du module est placé à l'intérieur du dossier.

Les fichiers sont représentés par un rectangle. Pour distinguer les différents types de fichier, nous inscrivons le nom de l'extension à l'intérieur de ce rectangle : .h pour les en-têtes C++, .cpp pour les fichiers C++... Le nom du fichier peut aussi être mentionné mais cela n'est pas obligatoire.

La relation de dépendance de compilation ("prereq") est  représentée par une flèche entre les deux frameworks impliqués. Si un framework *Fw1* dépend d'un autre framework *Fw2* alors la flèche partira du framework *Fw1* pour pointer sur le framework *Fw2*.

La figure 6.8 récapitule la syntaxe graphique pour décrire les entités de l'architecture physique.

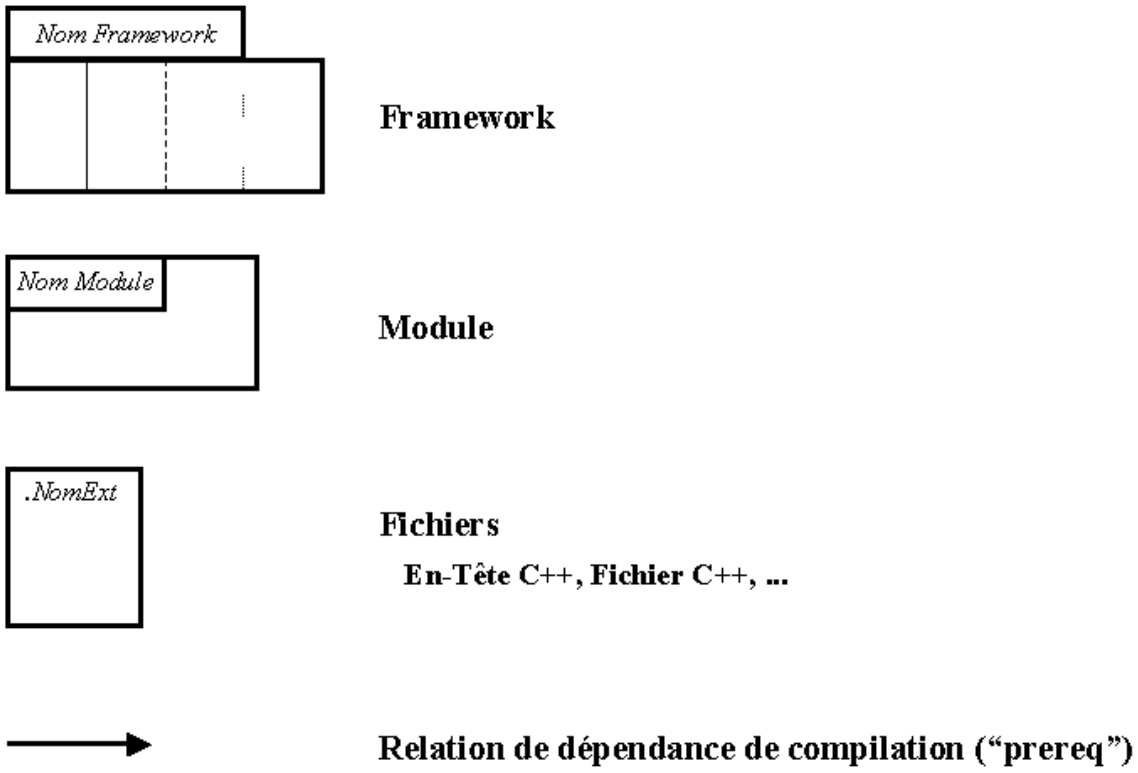


FIG. 6.8 – Syntaxe graphique pour l'architecture physique

Utilisation de la syntaxe graphique

La syntaxe graphique associée à l'architecture physique a été principalement exploitée dans nos documents et très peu dans nos prototypes. Cela s'explique par le fait que le prototype OMVT ait été développé initialement pour l'architecture logique (cf. section 7.1) et que l'environnement GSEE ait été conçu pour être générique ce qui ne nous permettait pas d'utiliser une syntaxe graphique particulière (cf. section 7.2).

Néanmoins nous avons pu intégrer dans l'OMVT quelques vues pour l'architecture physique sous forme d'arbres grâce au composant Swing JTree de java. Dans ces vues, nous avons spécialisé le composant Swing JTree pour pouvoir afficher des icônes semblables à notre syntaxe graphique pour les différentes entités de l'architecture physique.

6.1.3 L'architecture produit ou de packaging

Présentation du méta-modèle

L'architecture produit se concentre sur les besoins de Dassault Systèmes pour la gestion des configurations des différentes applications (cf. section 3.2). Elle s'occupe en outre de la gestion des licences et permet de définir la stratégie commerciale de Dassault Systèmes. Cette structure architecturale est particulière à Dassault Systèmes et nous ne l'avons pas retrouvée dans d'autres travaux. Ceci s'explique par le fait que d'une part CATIA V5 n'est pas un logiciel à part entière mais plutôt une infrastructure qu'il est possible de spécialiser pour son domaine d'activité en ajoutant un certain nombre de produits orientés métier. D'autre part cette structure architecturale reflète des besoins économiques qui n'ont généralement pas été étudiés par les précédents travaux.

L'architecture produit a pour objectif de créer des supports comme un cédérome qui contiennent l'ensemble des fichiers nécessaires pour l'installation de CATIA V5 et des différents produits sur une machine cliente. Ces produits sont définis de manière hiérarchique à partir des frameworks. La figure 6.9 présente le méta-modèle pour l'architecture produit.

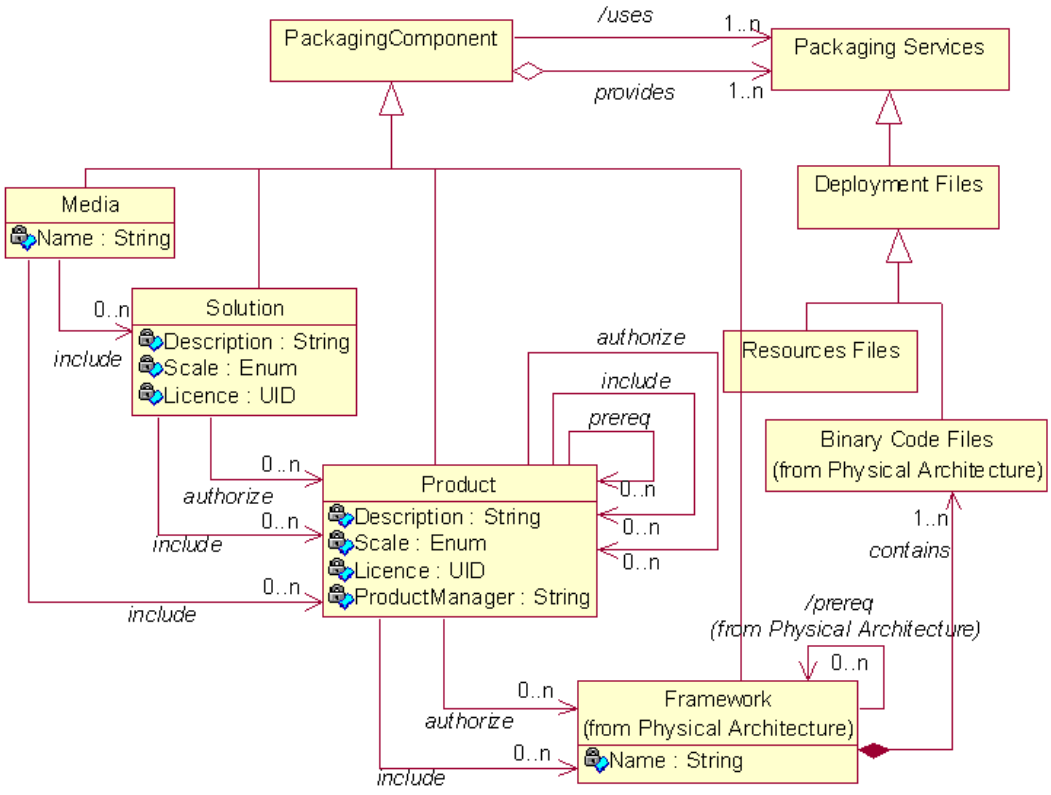
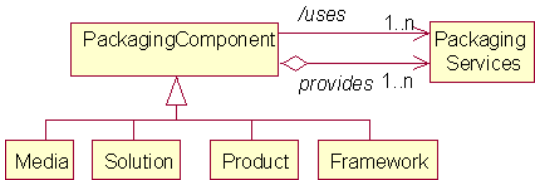
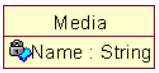


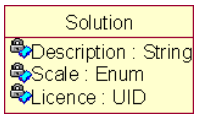
FIG. 6.9 – Le méta-modèle de l'architecture produit



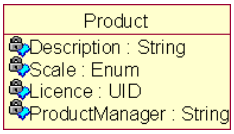
Les composants de packaging : un composant de packaging peut prendre plusieurs formes et suit une organisation hiérarchique allant du média au framework. Pour créer un composant de packaging, il est possible de se baser sur d'autres composants de packaging de niveau inférieur (et de même niveau dans le cas des produits). Il offre et utilise des services de packaging qui résultent de cette construction. Un composant de packaging nécessite une licence pour pouvoir être utilisé. Il existe actuellement quatre types de composants de packaging :



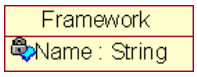
- **Les médias** : un média correspond au support (par exemple un cédérome) utilisé pour l'installation de produits et/ou de solutions sur une des sept plate-formes Windows et Unix disponible pour CATIA V5.



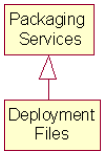
- **Les solutions** : une solution est un regroupement de produits pour fournir une suite d'applications pour un domaine particulier. Elle est définie par un nom et par les produits qu'elle utilise. Elle fournit des informations relatives à la gestion des licences et au niveau de gamme[3] auquel elle est affectée.



- **Les produits** : un produit est un regroupement de frameworks et éventuellement de produits pour former un produit commercial. C'est donc plus ou moins un rassemblement de fonctionnalités qui sont jugés suffisantes pour pouvoir être vendues. Il est défini par un nom et par les produits et frameworks qu'il utilise. Il déclare des informations relatives à la gestion des licences et au niveau de gamme auquel il est affecté. Un produit est géré par un chef de produit qui est responsable de l'implémentation des frameworks qui sont directement concernés par le produit.

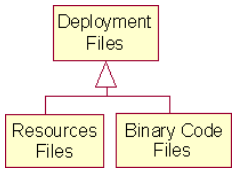


- **Les frameworks** : les frameworks jouent le rôle de la brique de base pour la création des différents produits commerciaux. Un framework ne peut pas être vendu séparément. Les frameworks ont un statut particulier car ils font le lien entre l'architecture physique et l'architecture produit et se retrouvent dans ces deux structures architecturales. Le framework est vu dans cette architecture produit non plus comme une entité de structuration et de compilation mais comme une entité avec un potentiel fonctionnel par l'intermédiaire des fichiers binaires qu'il possède.



Les services de packaging : Un service de packaging met à disposition des données nécessaires pour le bon déroulement de l'exécution du logiciel. Ils correspondent essentiellement aux fichiers de déploiement.

[3] cf. gamme de produits (P1-P2-P3) dans la section 2.2. Le niveau de gamme est représenté par la propriété Scale dans le méta-modèle.



Les fichiers de déploiement : Les fichiers de déploiement sont les fichiers qui seront copiés sur la machine cliente lors de l'installation du logiciel. Nous avons représenté deux types de fichiers de déploiement : les fichiers binaires (présentés dans l'architecture physique) et les fichiers ressources (des fichiers images pour les icônes, des fichiers textuels pour la gestion des différentes langues...).

Il existe trois types de liens de dépendance entre les composants de packaging qui sont utiles lors du déploiement et pour la gestion des licences :

- **Les liens d'inclusion ("include")** : ce type de lien indique que si un composant de packaging inclut un autre composant de packaging alors ce dernier devra être installé physiquement sur la machine dès que le premier le sera. Ceci est nécessaire car un composant de packaging a en général besoin de services de packaging d'autres composants de packaging pour fonctionner. Ce lien indique aussi que le client ne pourra accéder qu'aux fonctionnalités offertes par le composant de packaging acheté et pas à celles des composants de packaging inclus.

- **Les liens d'autorisation ("authorize")** : ce lien sert pour la gestion des licences. Ce type de lien implique en même temps un lien de type inclusion. A la différence du lien d'inclusion, le client aura le droit d'utiliser toutes les fonctionnalités fournies par les composants de packaging autorisés.

- **Les liens de prérequis ("prereq")** : ce type de lien signifie que le produit requis doit être déjà présent sur la machine du client. A la différence des liens de type inclusion, le composant de packaging prérequis ne sera pas installé en même temps que le produit acheté. Cela implique entre autres que le client doit préalablement acheter et installer les produits prérequis. Il est important de souligner qu'un autre lien aussi nommé "prérequis" existe dans l'architecture physique mais que sa nature est différente. Dans l'architecture physique, un lien de prérequis reflète une dépendance de compilation alors que dans l'architecture produit c'est un lien de dépendance de déploiement.

Il est intéressant de noter que le choix des liens reflète la politique commerciale de Dassault Systèmes. Par exemple, prenons le cas où un produit A a besoin d'un produit B pour fonctionner et que le client souhaite utiliser l'ensemble des fontionnalités des deux produits. Dassault Systèmes définira un lien d'inclusion entre le produit A et le produit B pour que le client achète les deux produits. Par contre, si Dassault Systèmes accepte que le client n'achète que le produit A, ce sera un lien d'autorisation entre le produit A et le produit B qui sera choisi.

Lien entre le méta-modèle et le niveau code

D'un point de vue réalisation un média est un support physique qui contient l'ensemble des fichiers nécessaires pour l'installation de CATIA V5 sur les postes des clients. Les solutions et les produits sont représentés par des répertoires qui contiennent chacun un fichier textuel. Comme pour le cas des frameworks, ce fichier

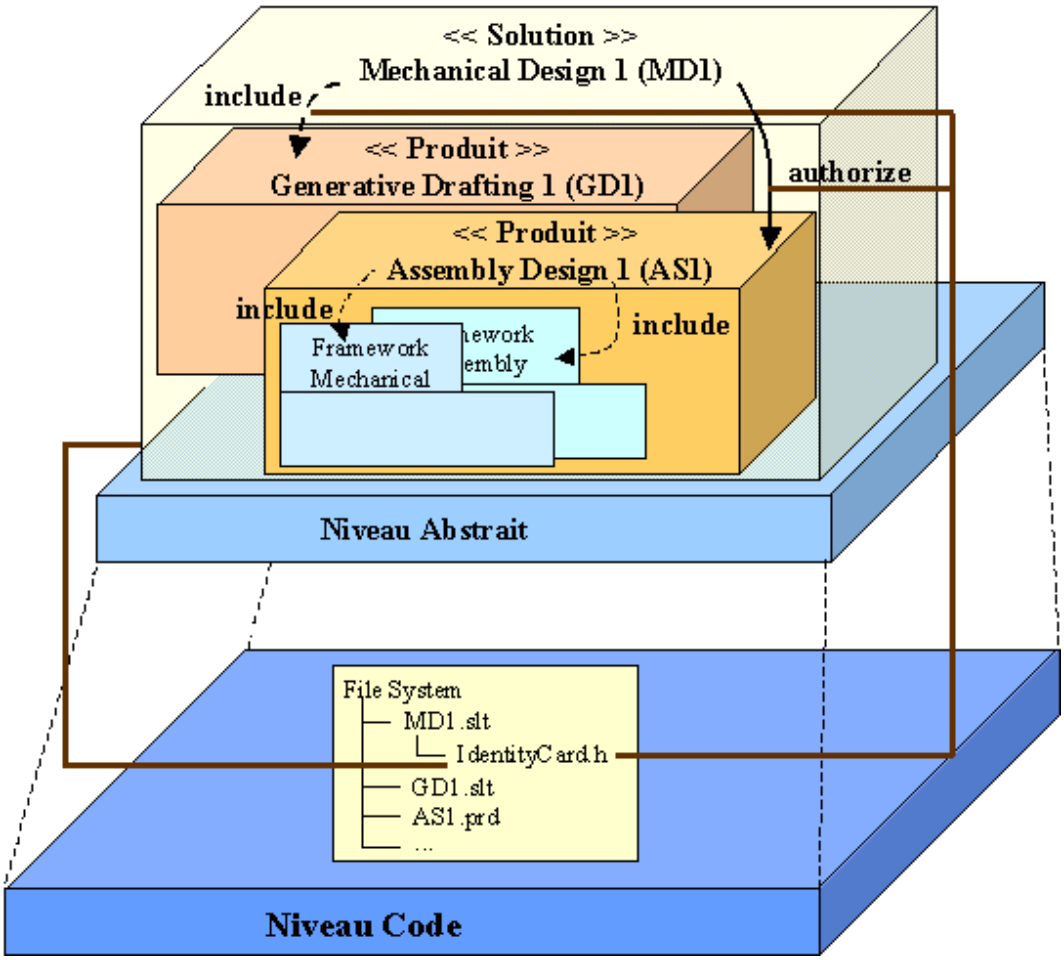


FIG. 6.10 – Lien entre le niveau abstrait et le niveau code pour l'architecture produit

est appelé la carte d'identité de la solution ou du produit. Il joue le rôle d'une interface pour la solution ou le produit car il contient ses caractéristiques (nom, gamme de produit, licence, liens de dépendance).

Par exemple, dans la figure 6.10, nous avons représenté la solution Mechanical Design 1 (MD1)[4] ainsi que deux des produits dont il dépend : Generative Drafting (GD1) et Assembly Design (AS1). Le lien de dépendance entre la solution MD1 et le produit GD1 est de type inclusion alors que c'est un lien de type autorisation pour le produit AS1. L'ensemble de ces informations sont définies dans la carte d'identité de la solution qui se trouve au niveau du code dans le fichier IdentityCard.h du répertoire MD1.slt. Les informations pour le produit AS1 qui inclut les frameworks Mechanical et Assembly sont définies de même dans la carte d'identité de ce produit qui se trouve au niveau du code dans le fichier IdentityCard.h du répertoire AS1.prd.

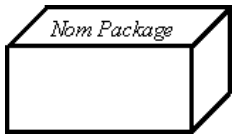
Les informations de ces cartes d'identités sont exploitées par un outil automatique. Il vérifie en premier lieu la cohérence de ces informations et génère ensuite la création des paquetages pour construire les médias.

Syntaxe graphique associée

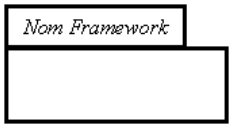
La syntaxe graphique associée à l'architecture produit a été inspirée par les notations trouvées dans les documents de Dassault Systèmes.

Un média est représenté par un disque pour symboliser le support de stockage du cédérome. Le nom du média est inscrit à l'intérieur du disque.

Une solution ou un produit est représenté par un parallélépipède rectangle symbolisant l'emballage d'un produit. Si le contexte le nécessite, il est possible de rajouter un stéréotype devant le nom pour indiquer le type du paquetage : <<Solution>> ou <<Produit>>.



Pour **le framework**, nous réutilisons la même notation que celle que nous avons définie pour l'architecture physique. Par contre pour l'architecture produit, il n'est généralement pas utile d'indiquer les quatre types de visibilité du framework.



Pour **les fichiers**, nous reprenons aussi la même notation que celle que nous avons définie pour l'architecture physique. La distinction entre les différents types de fichiers se fait toujours par le nom de l'extension : .jpg pour une image jpeg, .exe pour un exécutable...

[4]Le chiffre 1 à la fin du nom correspond à la gamme de produit.

La relation d'autorisation est représentée par une flèche à trait ⟶ continu entre les deux composants de packaging impliqués. Si un composant de packaging *Pck1* dépend d'un autre composant de packaging *Pck2* alors la flèche partira du composant de packaging *Pck1* pour pointer sur le composant de packaging *Pck2*.

La relation d'autorisation est représentée par une flèche à trait ⇢ pointillé régulier entre les deux composants de packaging impliqués. Si un composant de packaging *Pck1* dépend d'un autre composant de packaging *Pck2* alors la flèche partira du composant de packaging *Pck1* pour pointer sur le composant de packaging *Pck2*.

La relation de prérequis est représenté par une flèche à trait ⇢ pointillé irrégulier entre les deux composants de packaging impliqués. Si un composant de packaging *Pck1* dépend d'un autre composant de packaging *Pck2* alors la flèche partira du composant de packaging *Pck1* pour pointer sur le composant de packaging *Pck2*.

La figure 6.8 récapitule la syntaxe graphique pour décrire les entités de l'architecture produit.

Utilisation de la syntaxe graphique

Pour les mêmes raisons que pour l'architecture physique, la syntaxe graphique associée à l'architecture produit a été principalement exploitée dans nos documents et très peu dans nos prototypes.

Comme pour l'architecture physique, nous avons défini une vue sous forme d'arbre pour afficher les entités de l'architecture produit dans le prototype de l'OMVT. Nous avons de nouveau spécialisé le composant Swing JTree pour pouvoir afficher des icônes semblables à notre syntaxe graphique associée à l'architecture produit.

Dans l'environnement GSEE, nous avons utilisé des couleurs différentes pour différencier les trois types de relations de dépendance entre les composants de packaging.

6.1.4 Liens entre ces architectures

Nous venons de présenter, de façon séparée, les trois structures architecturales que nous avons étudiées. Mais ces structures architecturales ne sont pas indépendantes et pour bien comprendre l'architecture V5, il faut aussi s'intéresser à leur cohabitation.

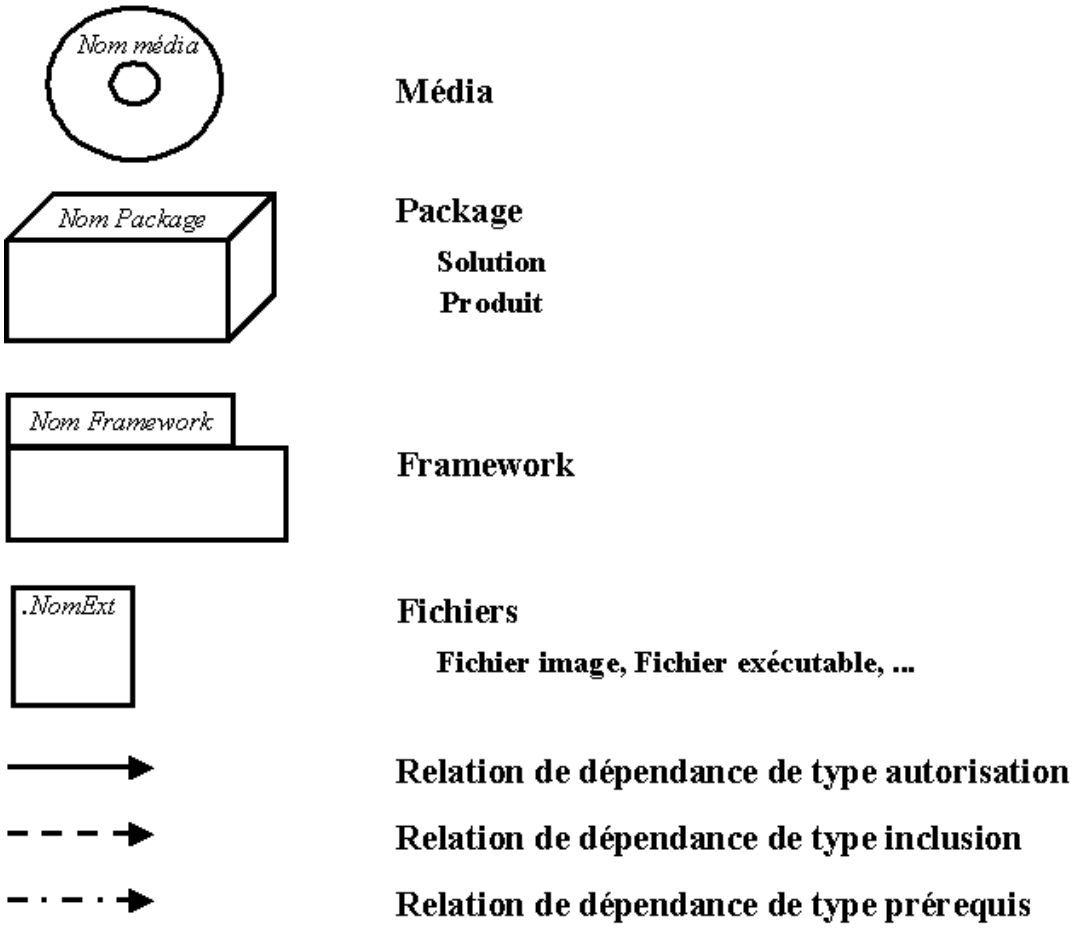


FIG. 6.11 – Syntaxe graphique pour l'architecture produit

Lien entre les architectures logique et physique

La figure 6.12 schématise le parallèle entre ces deux architectures. L'architecture logique regroupe les différents services fonctionnels que peut fournir un composant. Par exemple, le composant OM 1 possède à la fois des services de visualisation, de mécanique et certaines fonctionnalités propres à un client (ici Boeing). Inversement, l'architecture physique se concentre sur les domaines d'activités en regroupant l'ensemble des services d'un domaine particulier dans un même framework. Par exemple, le framework de visualisation contient les entités Object Modeler (interfaces, extensions et bases implémentations) liées au domaine de visualisation. Le framework mécanique sert à définir l'interface IMeca1 implémentée par l'extension 1.

Nous pouvons noter qu'une extension peut étendre plusieurs composants Object Modeler. Ceci permet de créer une nouvelle fonctionnalité par une équipe spécialisée et de faire profiter cette fonctionnalité à des composants Object Modeler maintenus par d'autres équipes.

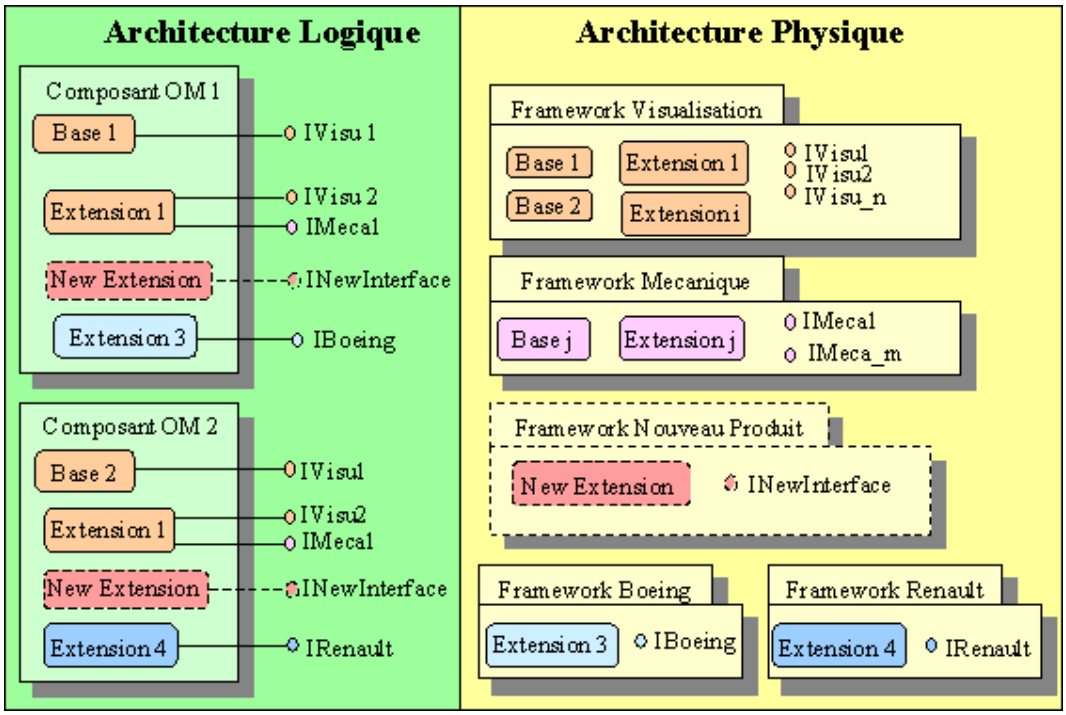


FIG. 6.12 – Parallèle entre l'architecture logique et l'architecture physique

La figure 6.13 permet de faire ressortir distinctement les structures et les relations de ces deux architectures. Dans cette figure, les différentes entités (base implémentation, extensions, interfaces) du composant Object Modeler sont dispersées dans plusieurs frameworks. Grâce au mécanisme d'extension, le partenaire peut rajouter ses fonctionnalités au composant Object Modeler sans modifier les entités Object Modeler et les frameworks de Dassault Systèmes. Il lui faut simplement créer son propre framework avec son extension et indiquer que cette extension étend le composant Object Modeler en question.

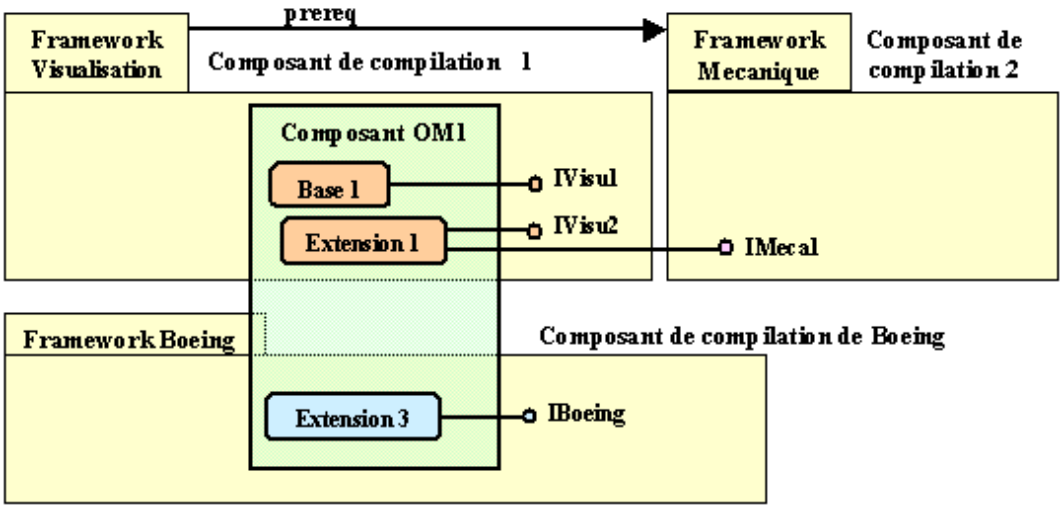


FIG. 6.13 – Lien entre l'architecture logique et l'architecture physique

Il est intéressant de remarquer que les liens de dépendance de construction entre ces entités Object Modeler (implémentation d'une interface, extension d'un composant) n'impliquent pas forcément des liens de dépendance de compilation entre les frameworks. Si le fait d'implémenter une interface Object Modeler crée une dépendance de compilation entre le framework de l'implémentation et le framework de l'interface Object Modeler, cela n'en est rien pour le cas de l'extension d'un composant Object Modeler. Par exemple, dans la figure 6.13, le lien de dépendance entre le framework Visualisation et le framework Mécanique provient du fait que l'extension 1 du framework Visualisation implémente l'interface IMeca1 du framework Mécanique. Par contre, l'extension du composant OM1 (défini par la base 1 du framework Visualisation) par l'extension 3 du framework Boeing ne crée pas de dépendance entre le framework Boeing et le framework Visualisation. Cette bonne propriété permet à Dassault Systèmes de faciliter le travail concurrent et de créer une architecture ouverte où il est possible de rajouter ses fonctionnalités avec un minimum d'impact de compilation. Ce mécanisme d'extension destiné initialement à ses clients est aussi utilisé en interne.

Le concept d'extension est réellement un point fort de l'architecture V5. Il offre de nouvelles possibilités pour l'adaptation d'un logiciel et constitue une innovation par rapport aux modèles de composants existants (même en prenant en compte les plus récents tels que CCM (CORBA 3.0) de l'OMG [Rui] ou le framework .NET de microsoft [MSN]). Utiliser intensivement en interne des mécanismes fournis à ses clients montre l'efficacité et l'intérêt de ceux-ci.

Lien entre les architectures physique et produit

Concernant le lien entre l'architecture physique et l'architecture produit, la figure 6.14 illustre l'interaction entre ces deux architectures. Comme nous l'avons dit

précedemment, un composant de packaging est un regroupement de frameworks. Dans notre exemple, nous avons trois produits (deux produits de Dassault Systèmes et un produit d'un partenaire de Dassault Systèmes) contenant chacun plusieurs frameworks ainsi que certaines relations de dépendance entre les frameworks et entre les produits[5].

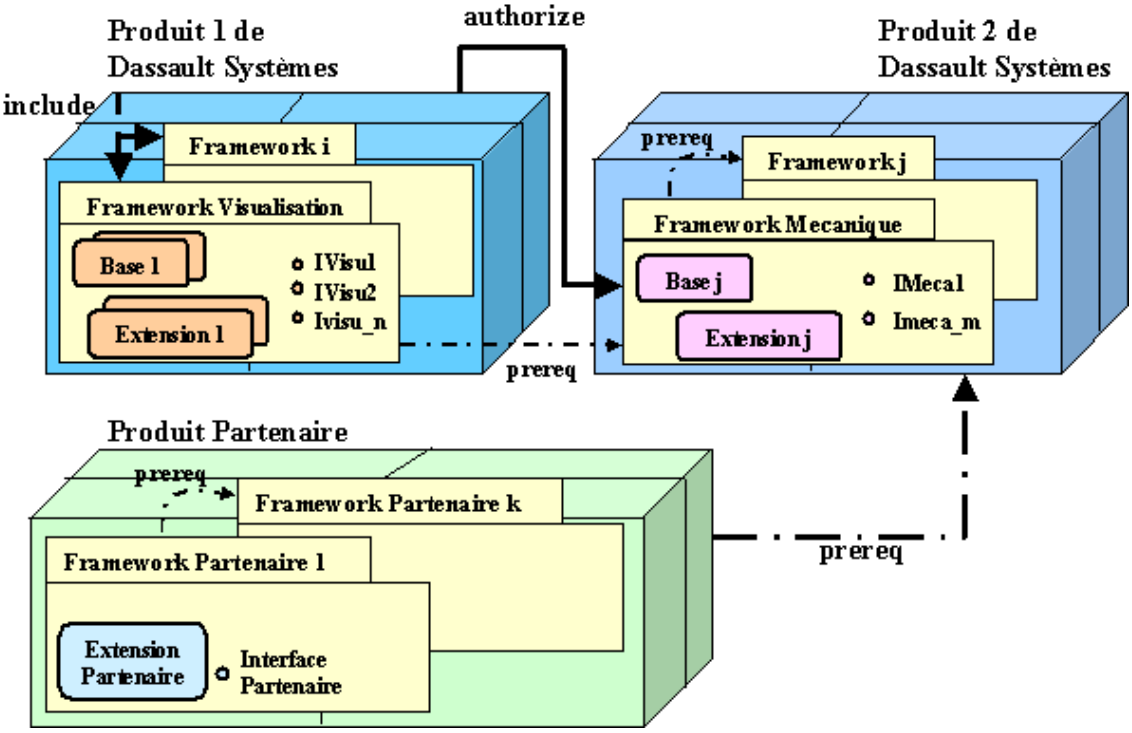


FIG. 6.14 – Lien entre l'architecture physique et l'architecture produit

Les relations de dépendance entre les composants de packaging ont deux origines :

- **Liens de fonctionnalité** : lors de la création d'un composant de packaging, le responsable de ce composant va utiliser d'autres composants de packaging pour définir les fonctionnalités de ce composant. Il crée ainsi des relations de dépendance entre ce composant et les autres. C'est ce que nous pouvons voir dans la figure pour le produit 1 de Dassault Systèmes avec le framework Visualisation et le framework i.

- **Liens de programmation** : ces liens qui ne sont pas spécialement souhaités par le responsable du composant de packaging sont de deux natures :

  - **Les liens de compilation ou "build-time"** : ces liens correspondent aux liens de dépendance de compilation entre les frameworks. Lorsqu'un framework (Fw1) d'un produit (Prd1) a un lien de dépendance de compilation vers un autre framework (Fw2) d'un autre produit (Prd2), cela génère un lien de dépendance entre le produit Prd1 et le framework (Fw2). C'est ce qui est illustré dans la figure avec le framework Visualisation du

[5]Pour une question de lisibilité nous n'avons pas représenté toutes les relations de dépendance.

produit 1 de Dassault Systèmes et le framework Mécanique du produit 2 de Dassault Systèmes.

- **Les liens d'exécution ou "run-time"** : comme nous pouvons le constater dans la figure, il existe aussi des liens de dépendance qui ne sont pas dûs aux dépendances de compilation entre les frameworks. C'est le cas avec le produit du partenaire vers le produit 2 de Dassault Systèmes. Ces liens reflètent le fait qu'un composant de packaging peut avoir besoin d'une ressource (un fichier image pour une icône, un fichier textuel pour un message d'erreur...) d'un autre composant de packaging lors de l'exécution du logiciel.

La description de ces liens est primordiale pour la construction des médias. Par exemple, si un lien de programmation n'est pas décrit, une erreur se produira à l'exécution lorsque l'utilisateur demandera une fonctionnalité ou une ressource qui n'est pas accessible à cause de ce lien manquant. Il est donc important de s'assurer que tous les liens sont bien décrits. Concernant les liens de compilation, il n'y a pas de problème pour les connaître par le fait que le compilateur indique une erreur s'il manque un de ces liens. Le point délicat se situe au niveau des liens d'exécution puisqu'il n'existe pas d'outil qui vérifie que tous ces liens sont indiqués. Actuellement, seuls les nombreux tests de qualités et la connaissance des ingénieurs de Dassault Systèmes permettent de s'assurer de la complétude de ces liens.

Concernant le type des relations de dépendance entre les produits, il en existe trois : les relations d'inclusion, les relations d'autorisation et les relations de prérequis (cf. section 6.1.3). Dans notre exemple, nous avons choisi arbitrairement les instances de ces relations mais nous aurions pu en choisir d'autres. Ce choix dépend de la volonté du responsable du produit et suit essentiellement une logique financière (cf. section 6.1.3).

Lien entre les architectures logique et produit

Ce lien entre ces deux architectures montre la flexibilité et l'ouverture de l'architecture V5 pour créer des configurations différentes de CATIA V5 selon les sites. Pour expliquer cela, nous nous appuierons sur la figure 6.15. Dans cet exemple, il existe un composant Object Modeler C1 et cinq produits (trois de Dassault Systèmes : P1,P2 et P3 ; un de BOEING : PB ; et un de RENAULT : PR). Ces produits contiennent chacun différentes entitités Object Modeler pour le composant Object Modeler C1. Par exemple, le produit P1 contient la base implémentation B1 et l'interface I1. Nous avons aussi représenté trois sites : celui de Dassault Systèmes, celui de BOEING et celui de RENAULT. Sur le site de Dassault Systèmes, il y a les trois produits (P1, P2 et P3) que les ingénieurs de Dassault Systèmes ont développés. Sur le site de BOEING, il y a les produits P1 et P2 de Dassault Systèmes que la société BOEING a acheté ainsi que le produit PB que les ingénieurs de BOEING ont développé. Sur le site de RENAULT, nous trouvons les produits P1 et P3 de Dassault Systèmes que la société RENAULT a acheté ainsi que le produit PR que les ingénieurs de RENAULT ont développé. Ainsi ces trois

sites ont des configurations différentes puisqu'ils ne possèdent pas exactement les mêmes produits. Dans notre exemple, les sociétés BOEING et RENAULT n'ont pas acheté et développé les mêmes produits du fait qu'ils ne souhaitent que les produits dont ils ont besoin pour leur métier (respectivement aéronautique et automobile).

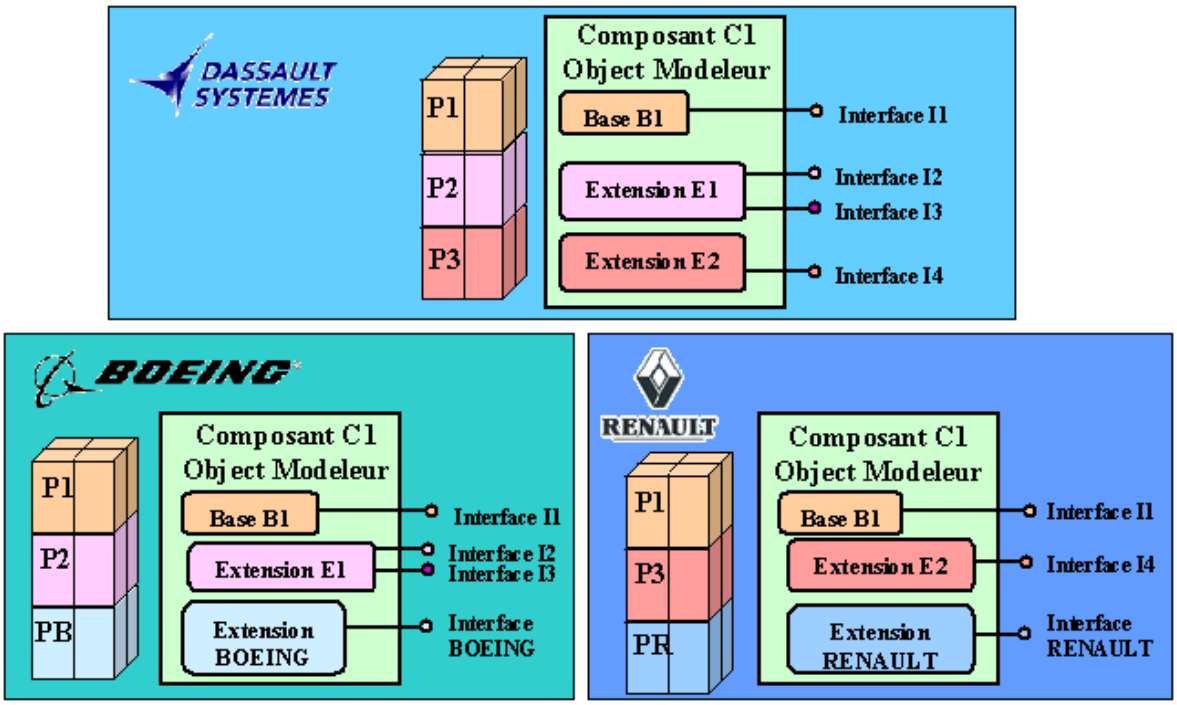


FIG. 6.15 – Lien entre l'architecture logique et l'architecture produit

Il est intéressant de remarquer que les différentes configurations de produits influent sur la structure de l'architecture logique. En effet, nous pouvons constater que dans notre exemple les entités Object Modeler diffèrent selon les sites. Le composant Object Modeler C1 offre les services logiques des produits présents sur le site. Par exemple, sur le site de BOEING, le composant Object Modeler C1 possède les interfaces I1, I2, I3 et Interface BOEING alors que sur le site RENAULT ce même composant Object Modeler C1 possède les interfaces I1, I4 et Interface RENAULT. Ceci permet à un client de personnaliser CATIA V5 en faisant "son marché" dans l'offre des produits de Dassault Systèmes, des produits des partenaires de Dassault Systèmes et de ses propres produits.

Lien entre les architectures logique, physique et produit

Il est intéressant de noter que même si l'architecture produit influence l'architecture logique, il n'existe pas de liens directs de compilation entre ces deux architectures (cf. figure 6.15). Les ingénieurs de Dassault Systèmes souhaitaient que ces deux architectures soient indépendantes au niveau de la compilation pour que lorsqu'un client achète un nouveau produit, il n'ait pas à recompiler toute l'application. Ils ont réussi à atteindre cet objectif en intercalant l'architecture physique entre les deux. Du coup l'impact n'est plus direct mais indirect : l'architecture

logique impacte l'architecture physique qui à son tour peut impacter l'architecture produit (cf. figure 6.16).

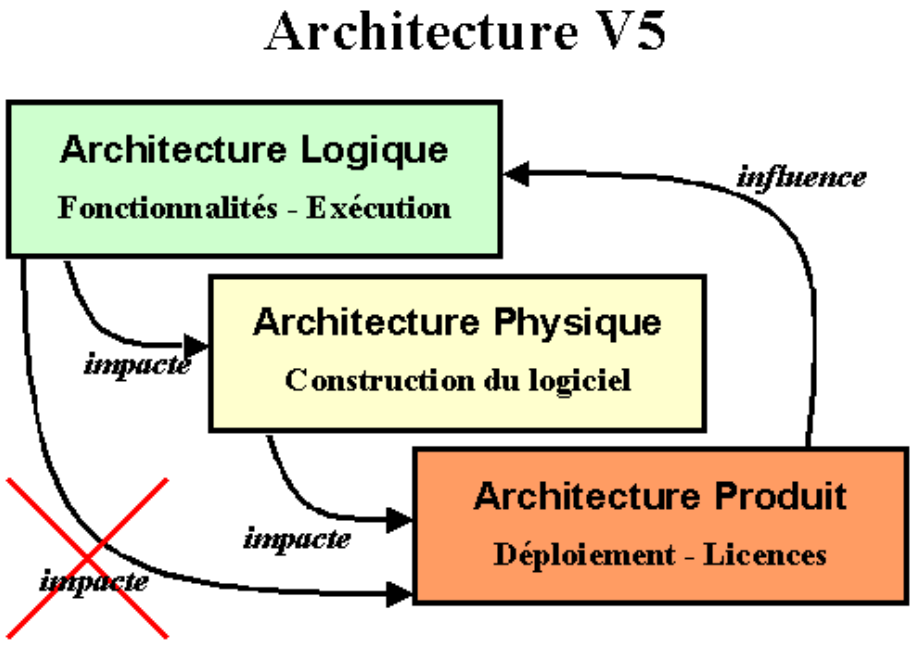


FIG. 6.16 – Liens entre les architectures logique, physique et produit

Ce système a l'avantage de rajouter un degré de liberté qui permet à un client de choisir les fonctionnalités de ses composants suivant les produits qu'il achète sans pour autant créer de dépendances de compilation et donc de devoir recompiler l'application. L'idée initiale des frameworks est de les voir comme des briques de base qu'il est possible d'assembler pour former une configuration particulière. Les ingénieurs de Dassault Systèmes, d'une certaine manière, ont réussi à créer un système qui s'approche du jeu de Lego. Mais cette réussite à un coût : ce système est assez complexe et n'est pas simple à utiliser et à maîtriser. Une modification d'une instance dans une architecture peut impacter cette même architecture ou une autre architecture qui, par transitivité, peut en impacter une autre.

Voilà par exemple, une requête d'un chef de produit à un ingénieur spécialisé dans l'architecture produit : "Je te sollicite pour te demander quels seraient les impacts au niveau des produits pour la prochaine release si j'ajoutais un lien de dépendance sur le framework InfInterfaces dans la carte d'identité du produit CATIAModelEditor". Ce chef de produit pose cette question car il doit avoir besoin d'un en-tête C++ du framework InfInterfaces suite à un besoin fonctionnel. Ceci implique donc un lien de compilation entre un de ses frameworks et le framework InfInterfaces. Par transitivité, cela crée un lien de dépendance entre son produit CATIAModelEditor et le framework InfInterfaces. De même cette dépendance risque de provoquer d'autres dépendances par rapport aux composants de packaging qui utilisent son produit CATIAModelEditor. Il souhaite donc connaître cet impact pour décider s'il a intérêt ou non à utiliser les services du framework InfInterfaces.

Comparatif de nos structures architecturales aves les vues de Kruchten et de Hofmeister et al.

Nous pouvons maintenant comparer nos travaux avec ceux de Kruchten et de Hofmeister et al. Pour cela nous reprenons la figure 5.3 qui fournit un comparatif des différentes vues architecturales de Kruchten et de Hofmeister et al. pour y intégrer nos trois structures architecturales (cf. figure 6.17).

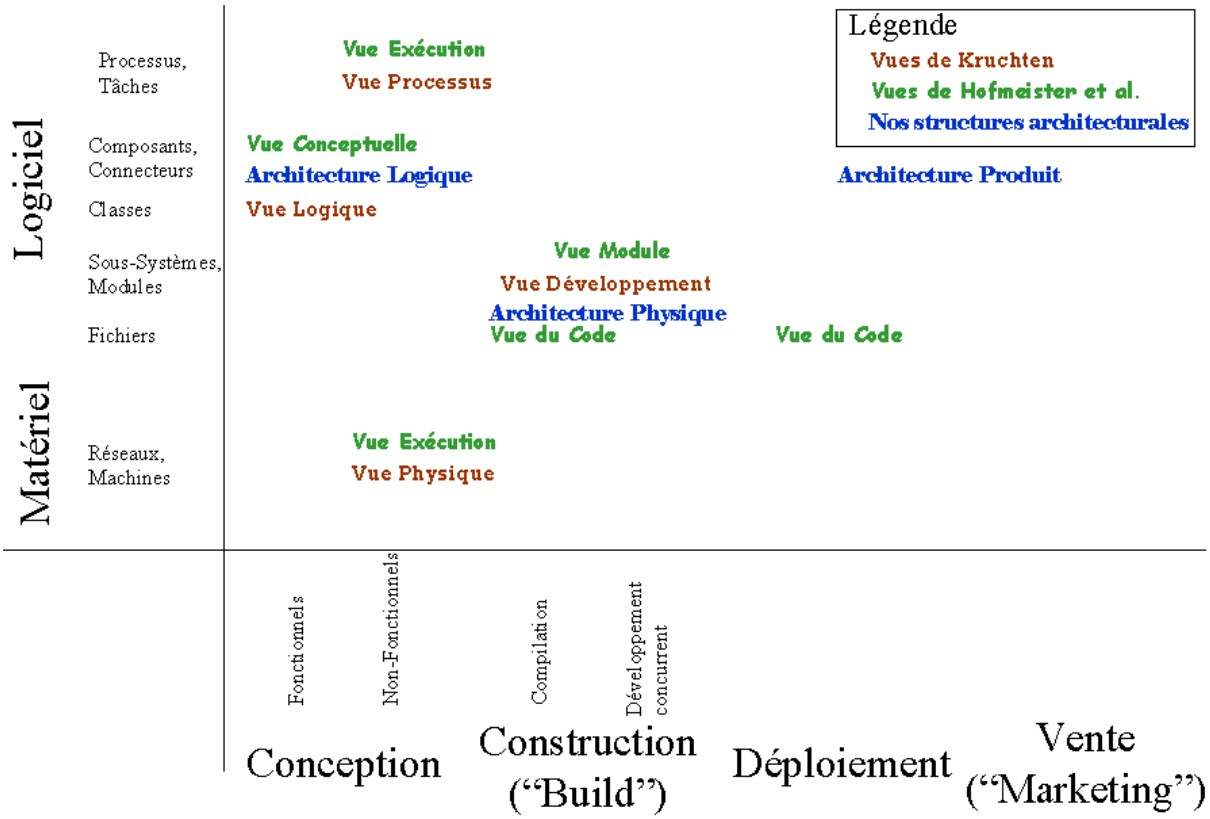


FIG. 6.17 – Comparaison des vues de Kruchten et de Hofmeister et al. et de nos structures architecturales

Nous pouvons voir que nos structures architecturales reprennent les aspects fonctionnels, de construction et de déploiement du logiciel des vues de Kruchten et Hofmeister et al. Par contre les aspects matériel et d'exécution (processus et tâches) ont été omis. Cela ne veut pas pour autant dire que ces aspects n'existent pas dans le cas de CATIA V5. Ceci reflète plutôt que nos recherches se sont limitées aux aspects fonctionnel et statique. Comme nous l'avons signalé, nous n'avons eu le temps d'étudier que trois structures architecturales et nous savons qu'il en existe d'autres. De plus, un autre projet dans la collaboration entre le laboratoire LSR et Dassault Systèmes avait pour objectif d'étudier les aspects non-fonctionnels au niveau des composants que nous n'avons pas pris en compte. Enfin, notre architecture produit permet d'ajouter des aspects de vente ("marketing") qui n'existent pas dans les travaux de Kruchten et Hofmeister et al.

6.2 Une organisation spécifique

Afin de bien comprendre une architecture logicielle, il ne suffit pas de s'intéresser uniquement aux concepts et aux aspects techniques de cette dernière mais il faut aussi prendre en compte ce que nous pourrons appeler l'environnement de développement qui regroupe l'organisation de la société, la culture d'entreprise, le processus de développement, les facteurs humains... Les travaux de Hofmeister et al. [HNS00] et Bass et al. [BCK98] soulignent en effet que ces aspects et l'architecture logicielle ne sont pas déconnectés et s'influencent mutuellement. Pour Hofmeister et al., l'organisation est l'un des trois facteurs qui affectent la conception de l'architecture et qui doit être pris en compte lors de l'analyse globale. Pour Bass et al., les différents acteurs (développeurs, décideurs, clients...) ont chacun des intérêts qui leur sont propres et souhaitent que l'architecture les prenne en compte et les optimise. D'un autre côté, l'architecture influence l'organisation et le processus de développement par le fait qu'elle permet de définir et structurer les différentes équipes, que la réussite ou l'échec d'une architecture accroît l'expérience de l'architecte... Cette influence mutuelle crée un cycle que Bass et al. appellent l'*Architecture Business Cycle (ABC)* (cf. figure 4.4).

Lors de la création d'une architecture logicielle, l'organisation et les différents acteurs sont des données importantes à prendre en compte. Par exemple, le simple fait de choisir une technologie voire un langage de programmation est une décision qui ne peut être dissociée de ces données. Il faut prendre en compte les connaissances des ingénieurs et leur adaptabilité à évoluer rapidement. Dans le cas de Dassault Systèmes, lors du passage de la version 4 à la version 5 de CATIA, le langage de programmation est passé d'un langage procédural (Fortran) à un langage orienté objet (C++). Lorsqu'une société a des centaines de développeurs qu'il faut former cela peut prendre plusieurs années et coûter très cher.

Lors de nos travaux nous avons pu observer cette influence réciproque entre l'architecture de CATIA V5 et l'environnement de développement de Dassault Systèmes. La section 6.2.1 illustre de manière générale cette influence dans le contexte de Dassault Systèmes. La section 6.2.2 décrit plus précisément différents acteurs de Dassault Systèmes pour l'architecture V5 ainsi que leurs intérêts respectifs sur cette architecture.

6.2.1 L'environnement de développement de Dassault Systèmes et l'architecture logicielle de CATIA V5

Afin de développer et maintenir CATIA V5, Dassault Systèmes a défini son propre processus de développement avec une organisation hiérarchique divisée en équipes. Il est intéressant de noter les similitudes entre cette organisation et les structures architecturales que nous avons étudiées. Les équipes sont généralement affectées à des tâches spécifiques pour une structure architecturale particulière. Par exemple, une équipe particulière se charge des besoins liés à l'architecture produit alors que d'autres doivent réaliser une fonctionnalité particulière de CATIA V5

comme la gestion de pièces mécanique pour l'architecture logique. De plus, pour cette architecture logique les équipes sont rattachées à l'un des modeleurs de cette architecture.

Le passage de la version 4 à la version 5 de CATIA s'est accompagné d'une refonte complète de l'architecture. Un nouveau langage de programmation et de nouvelles technologies sont utilisés pour le développement. Ceci a permis d'offrir de nouvelles fonctionnalités et services aux clients de Dassault Systèmes. Par exemple, depuis la version 5, CATIA est aussi disponible sur la plate-forme Windows. Mais cela a demandé aussi un important effort de formation pour que les ingénieurs de Dassault Systèmes soient opérationnels. Nous avons pu contater la volonté de Dassault Systèmes de former ses ingénieurs par la création de formations internes pour l'apprentissage de l'architecture V5 et d'autres externes pour apprendre et maîtriser le langage C++, les technologies liées à Windows (COM, OLE, VB...)... L'architecture de CATIA V5 a ainsi influencé l'organisation de Dassault Systèmes.

D'un autre côté, le savoir-faire de Dassault Systèmes et les connaissances de certaines équipes spécialisées dans un domaine de la CAO se retrouvent à travers les produits et frameworks. Les équipes ont la gestion d'un certain nombre de frameworks et réalisent d'une certaine manière une partition pour l'architecture physique. L'architecture de CATIA V5 est donc aussi influencée par l'organisation de Dassault Systèmes.

Un certain équilibre entre l'architecture logicielle de CATIA V5 et l'environnement de développement de Dassault Systèmes a fini par trouver ses marques au fil des années.

6.2.2 Différents acteurs pour l'architecture de CATIA V5

Dans les petits projets, les ingénieurs portent souvent différentes "casquettes" et font différents types de travaux. Dans les gros projets, ces différents métiers ressortent et chaque métier peut correspondre à une équipe. Les ingénieurs se spécialisent dans une tâche précise du développement. Le processus de développement doit ainsi prendre en compte le travail coopératif pour être efficace.

Au fur et à mesure de nos rencontres avec les ingénieurs de Dassault Systèmes pour comprendre l'architecture de CATIA V5, nous nous sommes aperçus qu'il existait effectivement différents acteurs ayant des intérêts propres [SFL01] (cf. figure 6.18). Ces acteurs ont pu être identifiés par le fait qu'ils ont chacun une connaissance et une perception différente de l'architecture V5. Ces différentes vues de l'architecture V5 reflètent les multiples métiers et objectifs des acteurs.

Afin de comprendre les intérêts spécifiques de chaque acteur sur l'architecture V5, deux critères ont pu être identifiés [SFL01] :

1. **Filtre sur le modèle** : selon le rôle de l'acteur, certaines entités du modèle l'intéresseront plus particulièrement. Par exemple, un programmeur sera plus

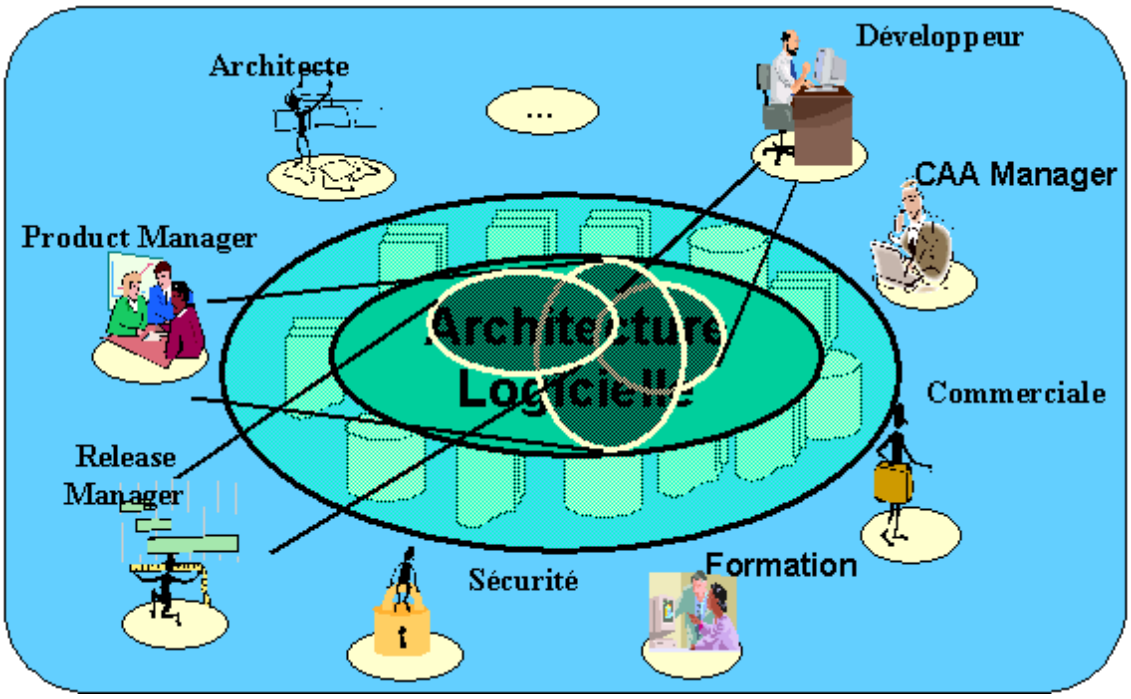


FIG. 6.18 – Architecture V5 et Acteurs de Dassault Systèmes

intéressé par l'architecture logique et les entités Object Modeler alors qu'un release manager sera concerné par l'architecture physique et les problèmes de compilation.

2. **Filtre sur les instances** : suivant la position de l'équipe dans l'organisation, certaines instances seront plus intéressantes pour les acteurs de cette équipe. Par exemple, un programmeur qui implémente des services basiques est plus intéressé par ses propres composants et les composants qui les utilisent alors qu'un programmeur qui implémente un produit est plus intéressé par ses propres composants et ceux qu'il utilise. De plus, certains acteurs comme l'architecte préfèrent obtenir des vues globales plutôt que locales.

Comme nous le verrons dans la partie suivante, nos outils prennent en compte ces aspects de sélection d'entités du modèle et des instances. Ceci permet à l'utilisateur de retrouver rapidement les informations qui lui sont utiles.

Nous présentons ci-dessous quelques exemples d'acteurs :

Programmeur	
Rôle	Développe une partie de l'application.
Intérêts	Technologies de Programmation (Langages de Programmation, Modèles de Composants...).
Filtre sur le modèle	Architectures logique et physique.
Filtre sur les instances	Vues locales sur ses instances, les instances qu'il utilise et/ou les instances qui utilisent ses instances.

Un programmeur est chargé de développer et maintenir soit une partie de l'infrastructure V5 soit des fonctionnalités d'un produit. Il utilise le langage C++ ainsi que les mécanismes de l'Object Modeler pour réaliser son développement. Il gère ses fichiers sources dans les frameworks et modules de son équipe. Lorsqu'il crée une dépendance avec un framework géré par une autre équipe, il prévient son chef de produit pour rajouter ce lien de compilation dans la carte d'identité du framework. Il est donc intéressé par les architectures logique et physique.

Si le programmeur gère des services de l'infrastructure V5, ses modifications pourront avoir des impacts sur ceux qui utilisent ses services. Il tente donc de minimiser au mieux ces éventuels impacts. Il est donc intéressé par ses instances et les instances qui les utilisent. Au contraire si le programmeur implémente des fonctionnalités d'un produit, ses modifications n'auront pas d'impact sur le reste du développement à DS. Par contre, il est plus sensible aux modifications des services qu'il utilise. Il est donc intéressé par ses instances et les instances qu'il exploite.

Architecte	
Rôle	Définit les architectures appropriées et vérifie leur réalisation.
Intérêts	Les différentes architectures avec leurs liens de dépendance
Filtre sur le modèle	Architectures logique, physique et produit.
Filtre sur les instances	Vues globales sur l'ensemble des instances.

Un architecte a pour objectif de gérer les différents besoins des acteurs et de fournir les meilleures solutions pour l'ensemble de ces besoins. Il ne doit donc pas privilégier le besoin d'un acteur particulier mais faire des compromis pour le bien commun. Il s'intéresse donc à l'ensemble des structures architecturales et doit avoir une vue globale des instances.

Product Manager	
Rôle	Définit les fonctionnalités de son produit. Gère ses développeurs pour réaliser ces fonctionnalités.
Intérêts	Technologies de Programmation (Langages de Programmation, Modèles de Composants...), Technologies de Packaging (Librairies dynamiques...), Besoins des Clients...
Filtre sur le modèle	Architectures logique, physique et produit.
Filtre sur les instances	Vues locales sur ses instances et sur les instances de ses programmeurs.

Un product manager (ou chef de produit) est responsable d'un produit et doit s'assurer du bon déroulement de son développement. Il a sous sa charge une équipe de développement comprenant plusieurs programmeurs. Il doit avoir de bonnes connaissances générales sur les différentes technologies pour pouvoir correctement estimer la faisabilité des nouvelles fonctionnalités à implémenter dans

le temps imparti (une release tous les 4 mois). Il est directement responsable des frameworks attachés à son produit. C'est lui qui doit gérer les cartes d'identités de ses frameworks et de la carte d'identité de son produit. Il est donc intéressé par les instances de ses programmeurs pour les architectures logique et physique et ses propres instances pour l'architecture produit.

Release Manager	
Rôle	Vérifie et corrige les erreurs de compilation dûes au développement concurrent. Exécute les objets de tests
Intérêts	Dépendance de Compilation (Langages de Programmation, Modèles de Composants...), Technologies de Packaging (Librairies dynamiques...).
Filtre sur le modèle	Architecture physique.
Filtre sur les instances	Vues locales sur les instances des programmeurs de ses équipes.

Le release manager se charge d'une partie de la construction de l'application. Il est affilié à plusieurs équipes de développement. Il doit faire en sorte que les implémentations de ses équipes se compilent bien ensemble. Il exécute aussi les objets de tests réalisés par ses équipes pour vérifier la qualité de l'application. Il s'intéresse donc à l'architecture physique et aux instances des programmeurs de ses équipes.

Packaging Manager	
Rôle	Définit les différentes configurations de produits, les licences et construit les médias (cédéromes...).
Intérêts	Dépendance d'exécution ("Run-time") : Fichiers Ressources (Images, Fichiers Textuels...), Technologies de Packaging (Librairies dynamiques...) ; Technologies de Licences, Prix...
Filtre sur le modèle	Architecture produit.
Filtre sur les instances	Vues locales de ses instances.

Le packaging manager a la responsabilité de la création des médias, de la gestion des licences des produits et du prix de ces produits. Il doit s'assurer qu'il ne manque rien sur le média qui pourrait compromettre la bonne exécution de CATIA V5. C'est à lui que revient la tâche de vérifier que tous les liens d'exécution sont présents (cf. section 6.1.4). Une personne particulière a pour charge de définir le prix de l'ensemble des produits. Cette personne a une responsabilité importante puisque c'est elle qui établit la stratégie commerciale de Dassault Systèmes. Le packaging manager s'intéresse aux instances de l'architecture produit qui sont sous sa responsabilité.

La table 6.1 résume la correspondance entre les architectures et les intérêts des acteurs de Dassault Systèmes présentés dans cette thèse.

Acteurs DS / Architecture V5	Programmeurs	Product Manager	Release Manager	Packaging Manager	Architectes	...
Architecture Logique	Vues locales de ses instances	Vues locales de ses programmeurs			Vues globales	
Architecture Physique	Vues locales de ses instances	Vues locales de ses programmeurs	Vues locales de ses programmeurs		Vues globales	
Architecture Produit		Vues locales de son produit		Vues locales de ses produits	Vues globales	
...						

TAB. 6.1 – Intérêts des acteurs de Dassault Systèmes sur l'architectures V5

6.3 En résumé

Pour bien comprendre une architecture logicielle, il est important de connaître autant les aspects techniques de celle-ci que l'environnement de développement de la société qui l'utilise.

Ce chapitre avait pour but de se focaliser sur ces deux points pour notre contexte :

- **Présentation de l'architecture V5** : d'une part, une explication détaillée de l'architecture V5 a été fournie par l'intermédiaire d'une modélisation de ses trois principales structures architecturales : l'architecture logique, l'architecture physique et l'architecture produit. Nous nous sommes efforcés de décrire le rôle et l'intérêt de chacune de ses structures architecturales ainsi que leurs méta-modèle et concepts.

- **Présentation de l'environnement de développement de Dassault Systèmes** : d'autre part, nous nous sommes aussi intéressés à l'environnement de développement de Dassault Systèmes. Plusieurs acteurs de Dassault Systèmes ont été mis en évidence par leurs rôle et intérêts concernant l'architecture V5 [SFL01].

Ce travail de synthèse et de modélisation a représenté un effort important car il n'existait pas de tel document. Nous avons apporté une nouvelle vision de l'architecture V5 grâce à notre décomposition en différentes structures architecturales et le lien de l'architecture logicielle de CATIA V5 avec les différents acteurs de Dassault Systèmes.

Notre expérience avec Dassault Systèmes confirme l'intérêt de prendre en compte différentes structures architecturales ainsi que l'environnement de développement décrit principalement dans quatre travaux précédents : Les "4+1" vues de Kruchten [Kru95], les vues de Hofmeister et al. [HNS00], la norme ANSI/IEEE Std P1471-2000 [Gro00] et les travaux de Bass et al. [BCK98].

La modélisation de l'architecture V5 a pu montrer le besoin de disposer d'autres structures architecturales que celles qui avaient été proposées par Kruchten et par Hofmeister et al. comme par exemple l'architecture produit. Nous pensons qu'il est intéressant de pouvoir capitaliser au mieux ces diverses structures architecturales qui permettent de répondre à des besoins différents et qui pourront éventuellement être réutilisées dans d'autres contextes.

L'étude approfondie de l'architecture V5 ansi que de l'environnement de développement nous a permis de bien comprendre les besoins architecturaux des acteurs de Dassault Systèmes pour le développement de CATIA V5. Afin de répondre à certains de ces besoins, nous avons conçu plusieurs prototypes. La partie suivante a pour objectif de vous présenter ces besoins et ces prototypes.

Troisième partie

Expérimentations et Résultats

Dans une phase de conception, une description architecturale est très utile pour la communication et la compréhension commune des différents acteurs et sert de référence pour le développement d'un logiciel. Elle permet aussi de réaliser différentes analyses afin de détecter des erreurs de conception avant même le début du développement. C'est dans cette optique que les différentes recherches dans le domaine de l'architecture logicielle, notamment au niveau des ADLs, ont été effectuées.

Par contre dans une phase de maintenance ou d'évolution, les vues architecturales utiles ne sont pas uniquement des vues de conception mais aussi des vues qui reflètent les structures architecturales qui sont dans le code source. De ce côté, il existe peu de recherche et notre contexte nous a amenés à travailler sur ces aspects.

Le chapitre 7 présente nos prototypes et expérimentations pour la maintenance et l'évolution de CATIA V5. Nous tâcherons d'expliquer au mieux l'intérêt qu'apporte chacun de ces prototypes pour répondre aux besoins des différents acteurs de Dassault Systèmes.

Fort de notre expérience, le chapitre 8 fait abstraction du contexte de Dassault Systèmes pour présenter ce que pourrait être un atelier de génie logiciel dédié à l'architecture logicielle. Nous présenterons un exemple complet avec un tel environnement pour la gestion de sites web.

Nous terminerons ce rapport de thèse avec le chapitre 9 où nous aborderons diverses réflexions autour de l'architecture logicielle. Nous exposerons notre point de vue à partir des leçons tirées de notre collaboration avec Dassault Systèmes.

Chapitre 7

Présentation de nos prototypes

L'architecture V5 a été utilisée avec succès pour construire de gros logiciels (CATIA, DELMIA, ENOVIA). Le lancement commercial de CATIA V5 a prouvé sa fiabilité. L'utilisation de différentes structures architecturales lors du processus de développement offre des propriétés intéressantes. Il est possible de structurer le logiciel selon différents points de vue : décomposition fonctionnelle du logiciel centrée sur les objets du monde réel (pièces mécaniques...) pour l'architecture logique, partitionnement du logiciel en frameworks pour gérer le développement concurrent au niveau de l'architecture physique, regroupement de services communs sur ces objets (analyses, visualisation...) en produits commerciaux pour l'architecture produit.

Néanmoins la flexibilité de cette architecture V5 a un coût. Elle n'est pas simple à maîtriser et demande des ingénieurs expérimentés et qualifiés pour l'utiliser. Le développement fortement concurrent et le volume de code à gérer augmentent cette complexité dans le cas de CATIA V5. L'architecture V5, tout comme le modèle COM de Microsoft, n'est pas simple à comprendre et à utiliser. Il est difficile d'avoir une vision claire de cette architecture juste en restant au niveau du code source.

Notre travail a consisté à apporter une compréhension externe à Dassault Systèmes de l'architecture V5. Notre approche a été de clarifier la vision de l'architecture V5 en s'abstrayant du code source et en identifiant différentes structures architecturales (cf. chapitre 6). Nous avons en même temps répertorié les besoins architecturaux des différents acteurs de Dassault Systèmes pour l'utilisation de l'architecture V5 et développé plusieurs prototypes pour y répondre. Ces prototypes sont basés sur les descriptions architecturales que nous avons réussies à extraire automatiquement du code source. La diversité des besoins des acteurs a permis d'explorer diverses directions : la visualisation et la navigation entre les structures architecturales de CATIA V5 (cf. sections 7.1 et 7.2), l'utilisation de diverses métriques architecturales (cf. section 7.2), la vérification de certaines règles de sémantique statique (cf. section 7.1), l'exploration de l'architecture logicielle de CATIA V5 (cf. section 7.2) et le calcul d'analyses d'impact au niveau du processus de développement (cf. section 7.3).

Notre collaboration avec Dassault Systèmes nous a permis de confronter et de mettre en pratique nos résultats théoriques par l'intermédiaire de ces prototypes qui sont basés sur l'ensemble du code source de CATIA V5. Ces prototypes ont pu confirmer l'intérêt d'utiliser une approche architecturale pour améliorer le processus de développement et la maintenance de CATIA V5.

Au niveau de l'extraction des données architecturales de CATIA V5, nous nous sommes appuyés sur un outil d'ingénierie inverse développé par les ingénieurs de Dassault Systèmes. Cet outil permet d'extraire à partir du code source différentes informations dans un fichier qui sont ensuite exploitées par des outils d'analyse dans le processus de développement de Dassault Systèmes. Comme ce fichier contient suffisamment de données architecturales pour nos expérimentations, nous en avons fait le point de départ de nos outils ce qui nous évitait de nous préoccuper de ces problèmes d'extraction. A partir de ce fichier, nous appliquons différents outils d'analyse que nous avons développés afin de créer une base de données ObjectStore [ODE] et vérifier certaines propriétés. L'ensemble de nos prototypes utilisent cette base de données.

7.1 L'Object Modeler Vizualization Tool (OMVT)

L'OMVT a été le premier prototype que nous avons développé. Comme son nom l'indique, il était initialement destiné à l'Object Modeler mais au fur et à mesure de nos investigations sur l'architecture V5, nous avons rajouté des fonctionnalités liées aux architectures physique et produit. Néanmoins, il reste principalement centré sur l'Object Modeler.

L'idée de ce prototype n'a pas été instantanée mais correspond plutôt à une longue réflexion qui a germé au fur et à mesure de notre compréhension de l'Object Modeler et de nos rencontres avec les acteurs de Dassault Systèmes. La compréhension et la modélisation de l'architecture V5 (cf. Chapitre 6) ainsi que l'analyse des besoins (cf. section 7.1.1) ont été un processus long mais essentiel pour bien cerner la problématique de Dassault Systèmes. Nous avons réalisé de nombreuses expérimentations avant de nous lancer dans le développement de ce prototype (cf. section 7.1.2). Ce prototype a été un véritable enjeu pour nous tant au niveau de sa réalisation que de l'intérêt de nos recherches pour Dassault Systèmes (cf. section 7.1.3).

7.1.1 Analyse des besoins

Au fur et à mesure de notre compréhension de l'architecture V5 (particulièrement de l'Object Modeler) et par les divers entretiens avec les différents acteurs de Dassault Systèmes, nous avons pris conscience de l'importance de cette architecture dans le processus de développement. Nous avons aussi pu comprendre les difficultés rencontrées par les acteurs de Dassault Systèmes pour utiliser cette architecture

V5. La taille et la complexité du logiciel CATIA V5 et le développement fortement concurrent font qu'il est difficile de maîtriser l'ensemble sans outil approprié. Bien que Dassault Systèmes ait développé ou adapté un grand nombre d'outils pour leur processus de développement, la grande majorité est destinée au niveau C++ et peu d'entre eux se placent au niveau architectural. Nous avons donc voulu fournir un outil au niveau architectural pour aider les acteurs de Dassault Systèmes dans leurs tâches quotidiennes.

L'OMVT est un outil consacré essentiellement à l'Object Modeler pour l'architecture logique. L'idée initiale était de pouvoir représenter avec notre syntaxe graphique l'ensemble des composants Object Modeler de CATIA V5. Pour bien comprendre l'intérêt, il faut se mettre dans la peau d'un ingénieur de Dassault Systèmes. Voici les principales difficultés rencontrées par les ingénieurs lorsqu'ils développent avec l'Object Modeler :

- **Besoin de vues conceptuelles** : les développeurs utilisent des mécanismes de bas niveau (conventions de nommages, macro...) pour décrire leurs entités Object Modeler. Plusieurs de ces entitiés sont basées sur une même macro mais avec des paramètres différents. Du coup, la distinction de ces entités au niveau du code source n'est pas évidente et il est difficile de connaître leur sémantique et leur importance au niveau architectural. Par exemple, il n'est pas possible de connaître l'effet d'un héritage Object Modeler de composant sans outil adapté. De plus, cette macro, qui permet de déclarer une classe en une entité Object Modeler, est plus perçue comme une instruction de compilation que comme une instruction qui a une répercussion sur l'architecture logique. Les acteurs ont besoin de vues conceptuelles qui distinguent clairement les entités Object Modeler du code source et qui permettent de visualiser les conséquences sur l'architecture logique.

- **Besoin d'une description centralisée** : L'utilisation des extensions et le développement fortement concurrent font qu'il n'est pas possible de connaître la spécification complète d'un composant Object Modeler sans un outil approprié. La spécification du composant se trouve éparpillée dans différents frameworks et équipes différentes. De plus, si un ingénieur utilise un héritage Object Modeler, la spécification du composant ne peut être connue qu'après un calcul lié à la sémantique de l'héritage Object Modeler en question.

  Le fait de disposer d'une description centralisée est intéressant à la fois pour les développeurs pour détecter certaines contructions qui ne fonctionnent pas (comme le fait d'implémenter deux fois une même interface) et pour les clients pour connaître l'ensemble des interfaces du composant. La contrainte de ne pas implémenter une interface plusieurs fois n'est pas un problème trivial dans le contexte d'un développement fortement concurrent et où la spécification du composant est répartie dans plusieurs équipes.

- **Besoin de fonctionnalités de recherche** : la taille de CATIA V5 (près de 5 millions de lignes de code) pose des problèmes de traitement du volume de code. Il est difficile de rechercher manuellement des informations dans l'ensemble de

ces données. De plus le résultat de ces recherches peut être rapidement obsolète du fait de l'évolution constante du code source. Comme ceci correspond à une tâche quotidienne, des outils de recherche d'informations ont été développés. Néanmoins ces outils sont plus appropriés au niveau code source qu'au niveau architectural. Par exemple, il est difficile pour un développeur qui maintient une interface de connaître l'ensemble des implémentations existantes pour cette interface. Cette information est utile car elle donne une idée de l'impact s'il souhaite modifier la spécification de son interface.

L'ensemble de ces besoins traduisent une même idée : l'Object Modeler, qui est une surcouche à C++ pour pallier certains manques, décrit une nouvelle structure qui nécessite un (ou plusieurs) outil(s) approprié(s) pour la manipuler et la maîtriser. Cette structure architecturale doit être prise en compte au même titre que la structure liée au langage de programmation.

7.1.2 Etude de l'existant

Initialement, nous ne souhaitions pas développer notre propre outil par manque de temps et de connaissances (la visualisation de gros volumes de données est un problème complexe qui est un thème de recherche à lui seul). Avec l'aide d'une étudiante de 3^e année d'ingénieur ENSIMAG, nous avons donc réalisé une étude de l'existant [Ang00].

Comme CATIA V5 représente une grande quantité de données et que nous voulions représenter les dépendances entre ces données, nous nous sommes tournés vers des outils de représentation de graphe. Nous en avons étudié cinq[1] : Dotty [Dot], DaVinci [DaV], GraphPanel [GrP], VGJ [VGJ], GEF [GEF].

Cette étude devait répondre à une liste de critères bien définis :

- **Le format du fichier d'entrée des données** : Ce format devait être suffisamment simple pour pouvoir générer automatiquement des fichiers de données à partir d'informations extraites du code source de CATIA V5.
- **L'extensibilité de l'outil** : nous souhaitions pouvoir d'une part paramétrer facilement l'outil pour nos besoins et d'autre part pouvoir le faire communiquer avec des programmes externes.
- **Les vues disponibles dans l'outil** : comme nous voulions réaliser plusieurs vues pour les différentes structures de l'architecture de CATIA V5, nous devions nous assurer qu'il était possible de les créer.
- **Les fonctionnalités de navigation** : nous souhaitions pouvoir naviguer entre les différentes vues architecturales par de simples cliques souris. Par exemple, si nous avons une vue d'un composant Object Modeler et que l'utilisateur sélectionne une de ses interfaces par un double clique alors il faut afficher la vue architecturale correspondant à cette interface.

[1] Pour une liste plus complète vous pouvez vous référer à la page web de Sander [San].

- **La représentation des entités** : comme nous avions défini une syntaxe graphique, nous désirions que l'outil puisse la représenter au mieux.

Cette étude a montré qu'aucun de ces outils ne répondaient de manière satisfaisante à nos besoins. Néanmoins nous avons été intéressé par Dotty pour la représentation de gros volume de données. Nous avons donc procédé en deux étapes :

1. Nous avons tout d'abord développé l'OMVT qui propose un grand nombre de vues locales des entités Object Modeler avec des fonctionnalités de navigation et d'analyse de ces entités. Nous proposons aussi quelques vues globales mais sous forme de tableau ou d'arbre (grâce aux composants swing JTable et JTree de java) ce qui nous évitait de nous préoccuper des problèmes de placement.

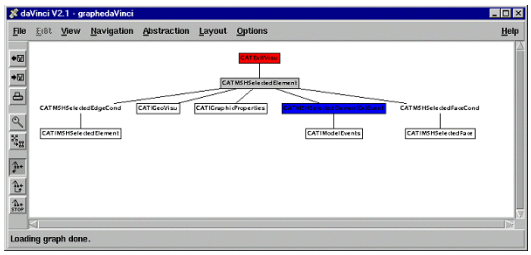
2. Ensuite, nous avons utilisé l'outil GSEE [Fav01], développé par Jean-Marie Favre (un collègue de notre équipe de recherche), pour des visualisations globales de l'architecture de CATIA V5 (cf. section 7.2). GSEE utilise Grappa qui est une implémentation en java de Dotty. Tout comme Dotty, Grappa se base sur l'outil Dot pour la gestion de la visualisation de gros graphes de dépendance.

La figure 7.1 de la page suivante fournit un aperçu de nos tests sur les différents outils que nous avons étudiés. Elle permet d'apprécier l'intérêt de l'OMVT simplement au niveau de la représentation de la syntaxe graphique. Nous avons aussi réalisé d'autres tests peu concluants avec l'outil Rational Rose [Ros] dédié à UML (cf. section 5.2.1).

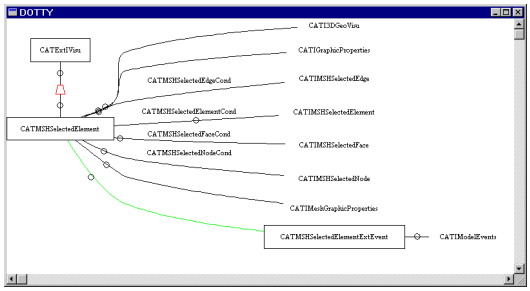
La table ci-dessous montre un récapitulatif de notre étude. Pour plus de précision, nous vous invitons à vous référer au rapport de stage [Ang00].

	Format Fichier	**Extensibilité**	**Vues**	**Navigation**	**Présentation**
Dotty	Texte simple	API	Multi-graphes, Multi-vues	–	Agréable
DaVinci	Texte	Menus, Icônes, API	–	Géométrique, Structurale	Agréable
GraphPanel	PostScript	–	–	–	Agréable
VGJ	GML	–	–	–	Pauvre
GEF	Librairie Java	Classes Java	Multi-vues	–	Agréable

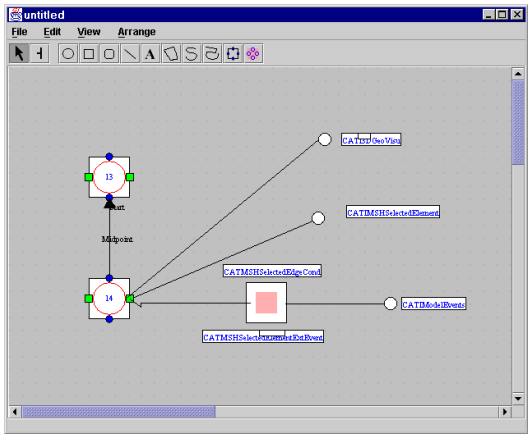
TAB. 7.1 – Etude de l'existant pour les outils de visualisation de graphe



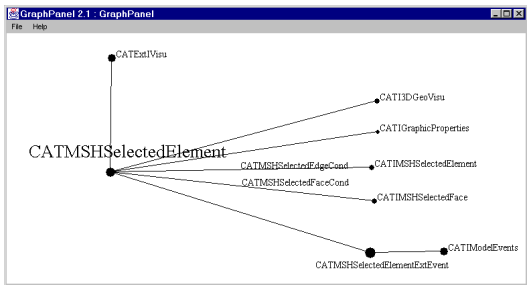
(a) DaVinci



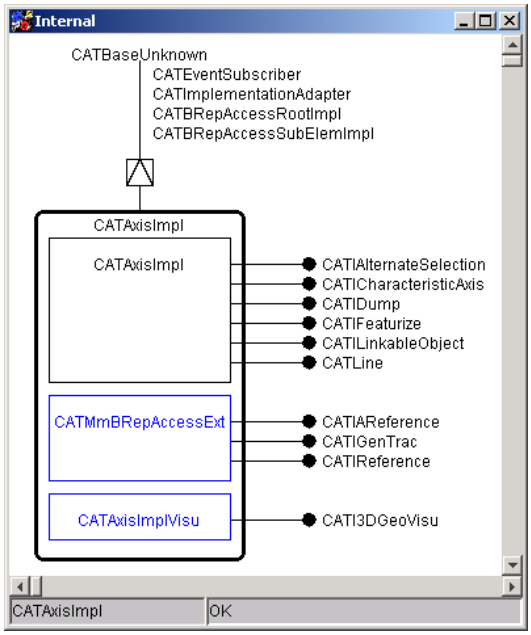
(b) Dotty



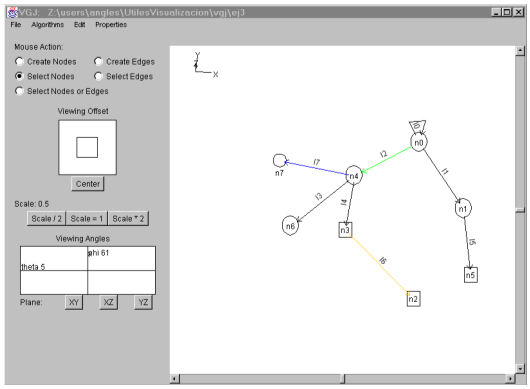
(c) GEF



(d) GraphPanel



(e) OMVT



(f) VGJ

FIG. 7.1 – Comparaison des outils de visualisation de graphe par la représentation d'un composant Object Modeler

7.1.3 Présentation de l'OMVT

L'OMVT a été développé dans le but de répondre aux besoins des acteurs de Dassault Systèmes au niveau architectural et plus particulièrement de l'Object Modeler pour l'architecture logique. Ce développement a fait l'objet d'un effort collectif qui a permis de réaliser un prototype fonctionnel, stable et performant. Cet outil offre diverses fonctionnalités de visualisation, de navigation, de recherche et d'analyse. L'utilisateur peut choisir parmi plusieurs vues locales et globales prédéfinies pour les différentes entités architecturales. La différence entre les vues locales et les vues globales est que les premières sont destinées à une instance particulière de l'architecture V5 alors que les dernières présentent un ensemble de données comme les arbres d'héritages. L'outil permet aussi de réaliser des analyses pour vérifier certaines règles de sémantique statique.

Tout au long de la présentation de l'outil, nous illustrerons nos propros par des exemples qui font référence au même composant Object Modeler CATAxisImpl. Nous insisterons sur les fonctionnalités de navigation qui sont un point fort de l'outil.

Les vues locales

Une vue locale représente une entité particulière de l'architecture V5 avec une visualisation prédéfinie. Pour chaque entité disponible dans l'outil (interface Object Modeler, extension Object Modeler, framework...), il existe plusieurs vues qui lui sont associées. Cela permet de représenter une même entité avec des points de vues différents pour les divers besoins des acteurs de Dassault Systèmes.

Le choix d'une vue se fait en trois étapes (cf. figure 7.2) :

1. En premier lieu, l'utilisateur choisit, parmi une liste qui lui est proposé, le type de l'entité qu'il souhaite visualiser (base implémentation Object Modeler, composant Object Modeler...).
2. Ensuite, une deuxième liste propose l'ensemble des vues qui sont disponibles pour le type de l'entité sélectionnée (vue interne, vue externe, vue physique...).
3. Enfin, l'utilisateur peut indiquer l'instance qui l'intéresse.

Nous pouvons remarquer que les deux premières étapes correspondent à un filtre sur le modèle alors que la troisième étape est un filtre sur les instances. Ceci reflète les deux critères que nous avions identifiés pour caractériser les intérêts spécifiques de chaque acteur (cf. section 6.2).

L'OMVT possède actuellement 22 vues qui sont principalement destinées à l'architecture logique. Les premières vues qui ont été réalisées, permettent de représenter les composants Object Modeler. Par exemple, la figure 7.3 montre deux vues pour le composant Object Modeler CATAxisImpl. La vue interne permet de visualiser comment est construit le composant et est utile pour le(s) développeur(s) du composant. La vue externe, quant à elle, affiche le composant comme une boîte noire avec l'ensemble des interfaces qu'il expose. Cette vue permet aux utilisateurs de ce composant de connaître ses fonctionnalités et aux développeurs

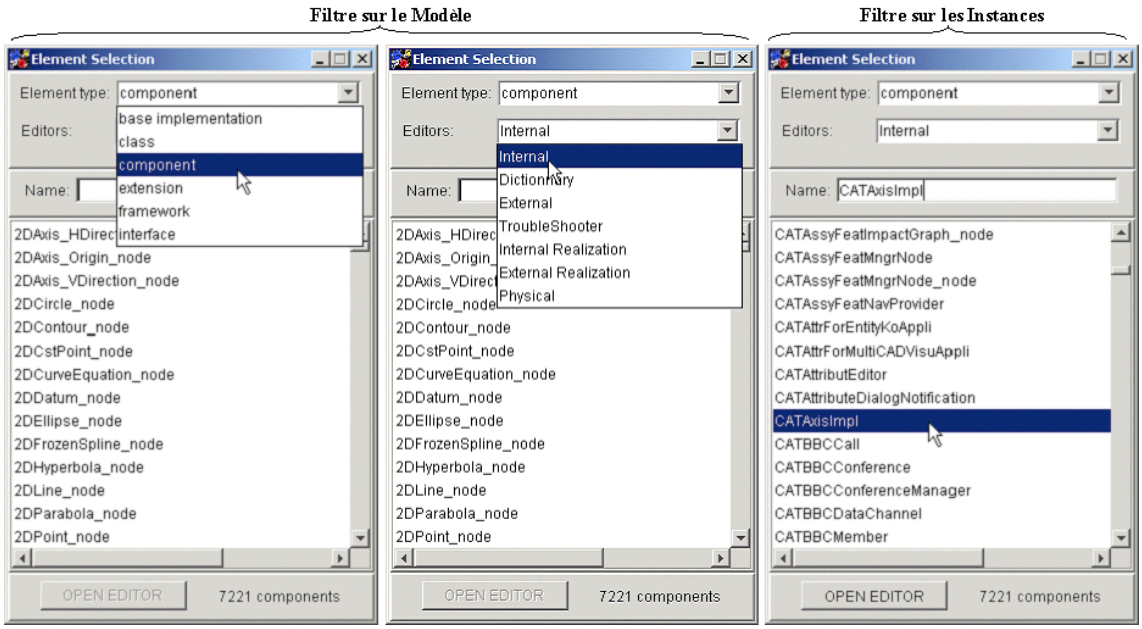


FIG. 7.2 – Sélection d'une vue dans l'OMVT

de ce composant de mieux comprendre la sémantique de l'héritage Object Modeler de composant. Les interfaces héritées sont représentées en bleu dans la vue externe.

Les vues logiques sont des représentations abstraites du code source qui devaient auparavant être réalisées mentalement par les ingénieurs. Les vues représentées dans la figure 7.3 sont plutôt simples (ce qui fait leur intêret) mais pas simplistes. Pour en être convaincu, il est important de noter qu'elles représentent une partie du code source de CATIA V5 répartie dans plusieurs fichiers C++ et développée par différents ingénieurs.

La vue de l'architecture physique de la figure 7.4 permet de visualiser l'ensemble des fichiers C++ regroupés par framework qui sont nécessaires pour réaliser le composant CATAxisImpl. Cette vue est importante pour les ingénieurs parce qu'elle représente les concepts qu'ils manipulent quotidiennement. Elle permet de faire le lien entre leurs implémentations et comment ces implémentations sont interprétées par l'architecture V5. La génération automatique de ces vues est très appréciable car il est très difficile et long de les réaliser manuellement à cause du développement concurrent et de la taille du code source.

Le passage entre les différentes vues se fait au travers de menus contextuels. Il est aussi possible de générer de nouvelles vues en sélectionnant directement les entités souhaitées. Par exemple, si l'utilisateur sélectionne une interface d'un composant par un double-clique, la vue par défaut pour les interfaces Object Modeler va s'afficher avec les données de l'interface en question. Ces fonctionnalités de navigation permettent aux acteurs de Dassault Systèmes de rechercher des informations et d'explorer les différentes entités de CATIA V5 par étapes successives et sans perdre le

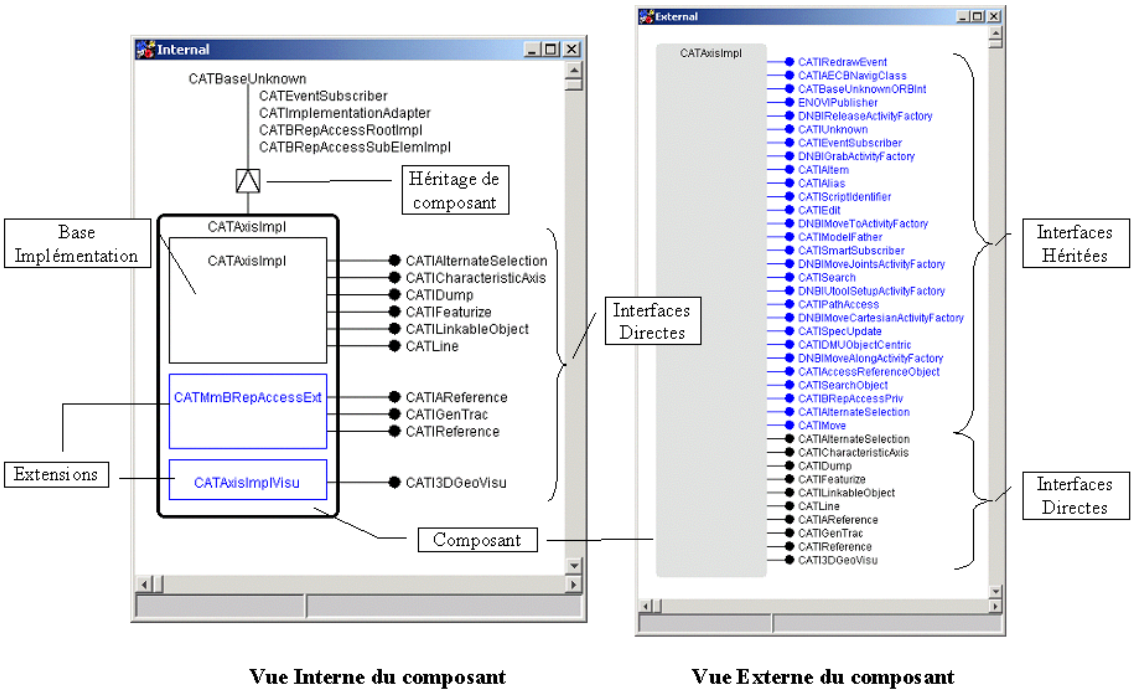


FIG. 7.3 – Vues Interne et Externe de l'Architecture Logique pour le composant CATAxisImpl

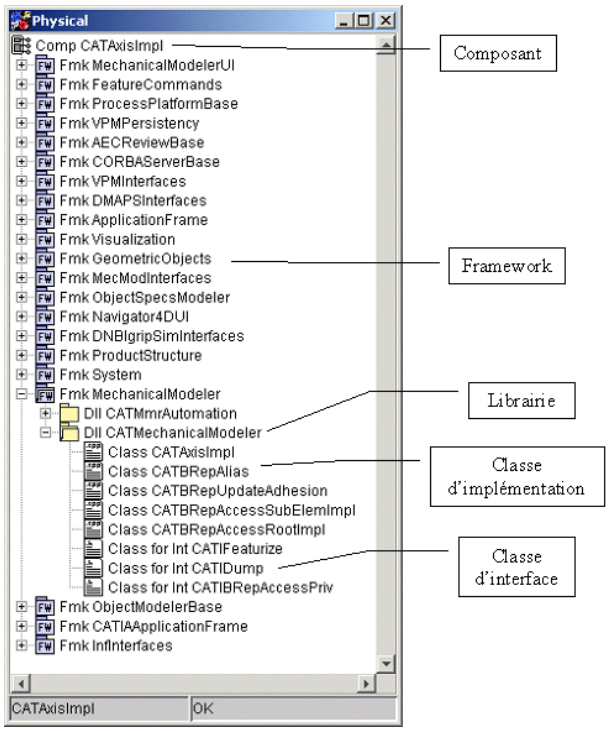


FIG. 7.4 – Une vue de l'Architecture Physique pour le composant CATAxisImpl

fil conducteur. La figure 7.5 présente un scénario où l'utilisateur part du composant CATAxisImpl. Il demande l'affichage du menu contextuel de l'interface CATLine de ce composant par un clique droit sur cette interface. Il choisit ensuite la vue d'implémentation pour connaître l'ensemble des implémentations disponibles pour cette interface. Dans cette nouvelle vue qu'il va explorer, il va demander d'afficher la spécification du composant CATCGMObject en double cliquant dessus.

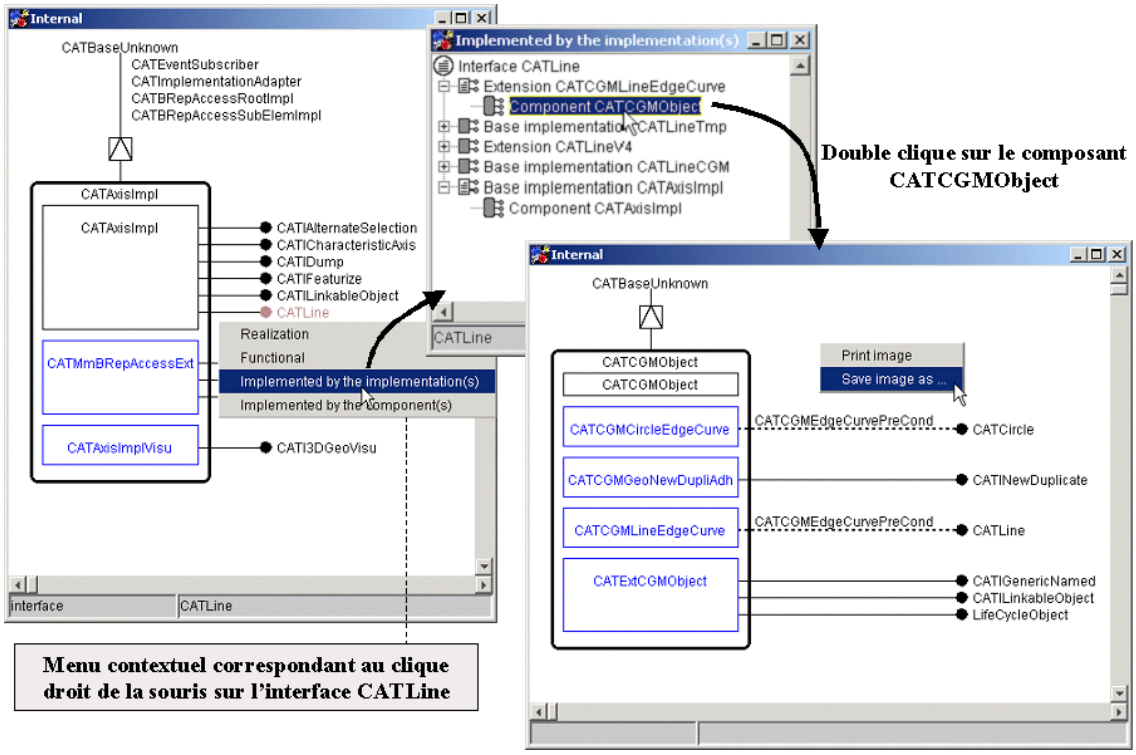


FIG. 7.5 – Un scénario de navigation avec l'OMVT

Les analyses

D'une certaine manière, l'Object Modeler peut être considéré comme un langage au dessus de C++ qui possède ses propres règles de sémantique statique. Par exemple, une de ces règles est qu'un composant ne peut pas implémenter plus d'une fois une même interface. Les ingénieurs de Dassault Systèmes ont repertorié et documenté l'ensemble des constructions à proscrire sous le nom de TroubleShooter. Néanmoins, la violation de ces règles n'est pas simple à éviter pour les raisons suivantes :

- De nombreuses règles impliquent un grand nombre de détails techniques qui ne sont ni simples à retenir, ni faciles à représenter mentalement. De plus, un nouveau développeur peut aisément écrire un morceau de code qui transgresse ces règles.

- Les ingénieurs ne possèdent pas les vues appropriées qui permettent de comprendre facilement le problème. Il ne faut pas oublier que les informations en

question sont codées à partir de macros et autres caractéristiques bas niveaux réparties dans différents fichiers.

- Le développement fortement concurrent fait qu'un composant est souvent développé par différents ingénieurs. Il est donc difficile de connaître la spécification complète du composant et les éventuelles évolutions de celui-ci.

- L'Object Modeler offre des mécanismes qui peuvent avoir un impact sur une partie importante du code source. Une modification d'un composant peut influencer par les liens d'héritage Object Modeler des centaines voire des milliers d'autres composants. Il n'est donc pas simple de s'assurer que sa modification n'enfreigne pas une de ces règles.

Dans de telles conditions, il n'est pas surprenant de voir apparaître des incohérences. Dans le processus de développement de Dassault Systèmes, ces incohérences sont détectées par un outil d'analyse pour le TroubleShooter toutes les semaines.

Il est intéressant de noter que ces incohérences peuvent avoir des conséquences plus ou moins graves, mais qu'elles n'empêchent pas la compilation du code. Ces incohérences provoquent deux types d'erreurs : des erreurs à l'exécution ou des erreurs de conception. Les erreurs à l'exécution ne sont pas toujours immédiatement détectées car elles ne sont générées que dans certains cas d'utilisation bien précis et ne provoquent pas toujours un arrêt du logiciel. Les erreurs de conception, quant à elles, peuvent ne pas influencer les fonctionnalités destinées à l'utilisateur mais peuvent nuire dans certains cas à l'évolution de l'architecture logicielle de CATIA V5. Les ingénieurs peuvent donc continuer à travailler sans que ces erreurs ne soient complètement corrigées.

Nous avons réimplémenté le moteur d'analyse du TroubleShooter dans l'OMVT et intégré des vues spécifiques pour représenter simplement les résultats de ces analyses. Un ingénieur peut ainsi facilement visualiser les erreurs de ses composants Object Modeler et comprendre rapidement l'origine de ces erreurs. La figure 7.6 illustre cela par un scénario typique. Après avoir demandé la vue TroubleShooter pour le composant CATAxisImpl, l'utilisateur peut apercevoir directement sur le composant un certain nombre d'avertissements. Dans cet exemple, l'utilisateur est intéressé par l'avertissement de multi-adhésion pour l'interface CATIAAlternate-Selection (i.e. que l'interface est implémentée plus d'une fois par le composant en question). Pour avoir des explications sur cette multi-adhésion, il lui suffit juste de double-cliquer sur l'icône qui se trouve à côté de l'interface. A ce moment, la vue de multi-adhésion apparaît et fournit un diagnostic clair et simple du problème. Grâce aux fonctionnalités de navigation de notre outil, l'utilisateur pourra retrouver les fichiers C++ correspondants afin de corriger le problème.

Il est important de noter que ces vues permettent un gain de temps important. Sans ces vues, l'ingénieur doit retrouver manuellement les fichiers des entités Object Modeler incriminés ce qui n'est pas toujours évident à cause de l'héritage Object Modeler et au développement concurrent.

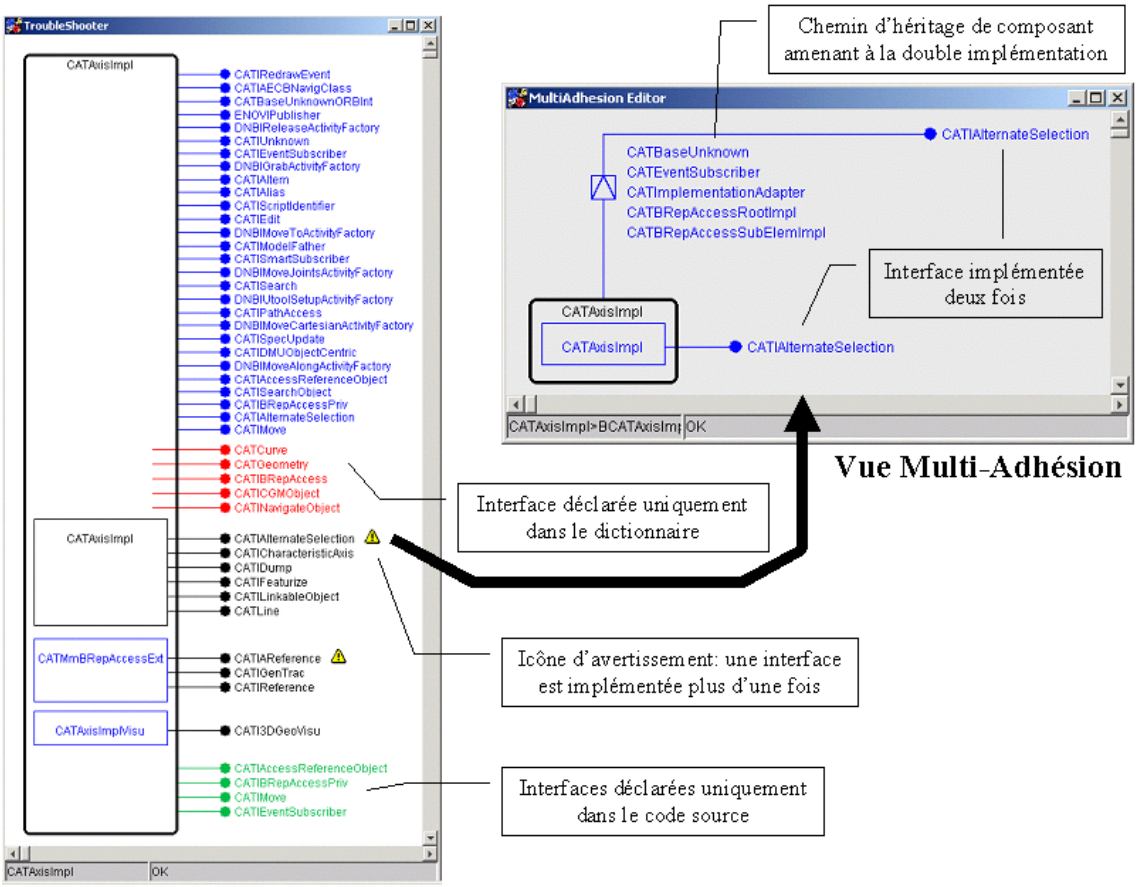


FIG. 7.6 – Vues TroubleShooter de l'Architecture Logique pour le composant CATAxisImpl

Les vues globales

A la différence des vues locales, qui sont destinées à une instance particulière, les vues globales (appelées browsers dans l'OMVT) permettent de visualiser un ensemble d'instances prédéfinies. Ces vues globales sont accompagnés de simples métriques qui permettent de réaliser certaines interprétations. Il est par exemple possible de retrouver le composant qui implémente le plus grand nombre d'interfaces ou la hauteur d'un arbre d'héritage.

L'OMVT possède actuellement cinq vues globales :

1. **La vue des composants Object Modeler** : cette vue est une table répertoriant l'ensemble des composants Object Modeler avec quelques métriques prédéfinies comme le nombre d'extensions, le nombre d'interfaces directes, le nombre d'interfaces héritées. Cette table peut être transférée sous Excel pour afficher des graphiques et réaliser des statistiques. Il est ainsi possible de voir la répartition du nombre d'interfaces par composant. Cette vue est liée à l'architecture logique.
2. **La vue des héritages** : cette vue permet de visualiser les trois arbres d'héritage existants dans l'architecture logique : héritage Object Modeler d'interface, héritage Object Modeler de composant et héritage C++ (cf. ci-après).
3. **La vue du TroubleShooter** : au même titre que la vue des composants Object Modeler, cette vue affiche un table sur l'ensemble des composants Object Modeler. Par contre, les métriques diffèrent et portent sur des données liées aux erreurs du TroubleShooter. C'est une nouvelle vue pour l'architecture logique.
4. **La vue des frameworks** : cette vue est un arbre regroupant l'ensemble des frameworks de CATIA V5. Il correspond à une structure hiérarchique allant du framework aux fichiers C++ en passant par les librairies dynamiques. Il correspond à une vue de l'architecture physique.
5. **La vue des produits** : cette vue est un arbre visualisant la structure hiérarchique des produits (media - solution - produit - framework). C'est une vue de l'architecture produit.

La figure 7.7 présente la vue globale pour la gestion des différents héritages pour l'architecture logique. L'utilisateur peut choisir, par l'intermédiaire du menu, un des trois types de lien d'héritage : l'héritage Object Modeler d'interface, l'héritage Object Modeler de composant (celui qui a été sélectionné dans la figure) et l'héritage C++. Pour chaque nœud de l'arbre, trois métriques sont affichées qui permettent de connaître la répartition de ces nœuds dans l'arbre : le poids et son pourcentage (la proportion du sous-arbre formé à partir du nœud en question par rapport à l'arbre complet), la hauteur (la hauteur de ce sous-arbre) et le nombre de fils (le nombre de fils du premier niveau de ce sous-arbre). Une fonction de recherche est également fournie afin de retrouver rapidement une entité particulière. Dans la figure, l'utilisateur a recherché le composant CATAxisImpl. Il peut ainsi se rendre compte que ce composant dépend de plusieurs autres super-composants et qu'aucun composant ne dépend de lui (en ce qui concerne la relation d'héritage Object Modeler de composant) puisqu'il n'a pas de fils. Ces informations sont intéressantes car cela signifie

que si un développeur souhaite réaliser une modification sur ce composant, il ne risque pas d'affecter de composants fils pour cette relation d'héritage. Par contre, ce composant pourra l'être si un de ses composants pères est modifié.

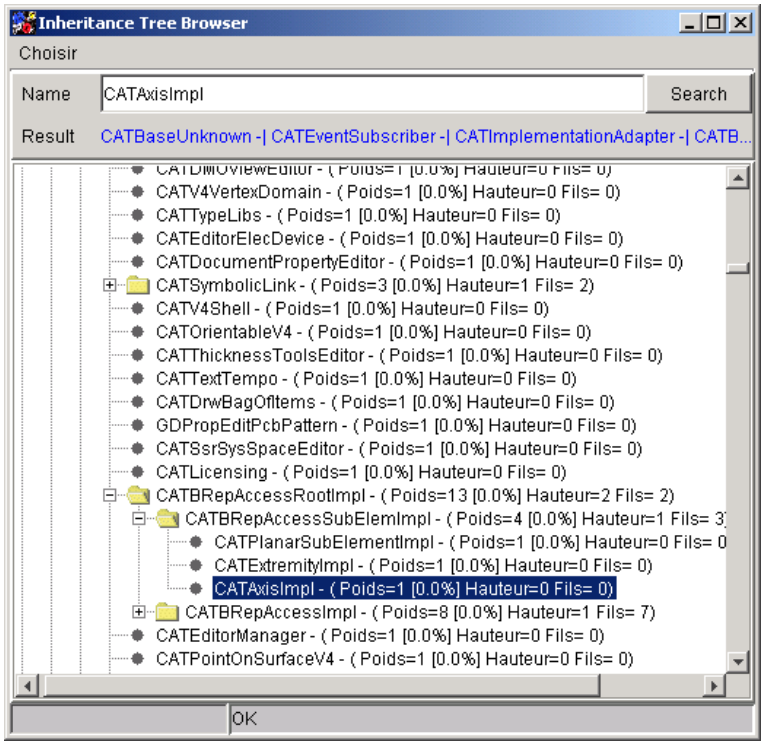


FIG. 7.7 – Vue globale des liens d'heritages

La réalisation de l'OMVT

Le processus de développement qui a accompagné ce prototype a suivi un cycle de vie en spirale qui a l'avantage de faciliter la maîtrise du développement en ajoutant de nouvelles fonctionnalités au fur et à mesure des besoins. Nous sommes actuellement à la version 1.9.1. La bonne connaissance des besoins architecturaux de Dassault Systèmes a permis de bien définir le cahier des charges du prototype avec ses aspects fonctionnels et non fonctionnels. Avec l'aide précieuse d'un collègue (Jorge Villalobos), nous avons défini et développé une architecture logicielle évolutive qui a permis le développement concurrent de six personnes et l'ajout rapide de nouvelles fonctionnalités. Il faut par exemple, un à deux jours pour développer une nouvelle vue et quelques minutes pour l'intégrer dans le prototype.

L'architecture de l'OMVT est techniquement difficile à expliquer et est basée sur de nombreux patrons de conception. La figure 7.8 présente une vue schématique de notre architecture logique. Elle est construite sur la base du patron de conception MVC (Modèle-Vue-Contrôleur) [BMR+96] et l'utilisation de nombreux autres patrons de conception [GHJV95] (fabrique abstraite, observateur, commande, médiateur, pont, prototype). Le patron de conception MVC a été instancié sur deux niveaux : le premier correspond à la structure principale de l'application et

le second est utilisé pour les vues du premier niveau. Les différents éléments de l'architecture logique ont été pensés comme des composants dans le sens où ils peuvent être ajoutés, enlevés et remplacés facilement.

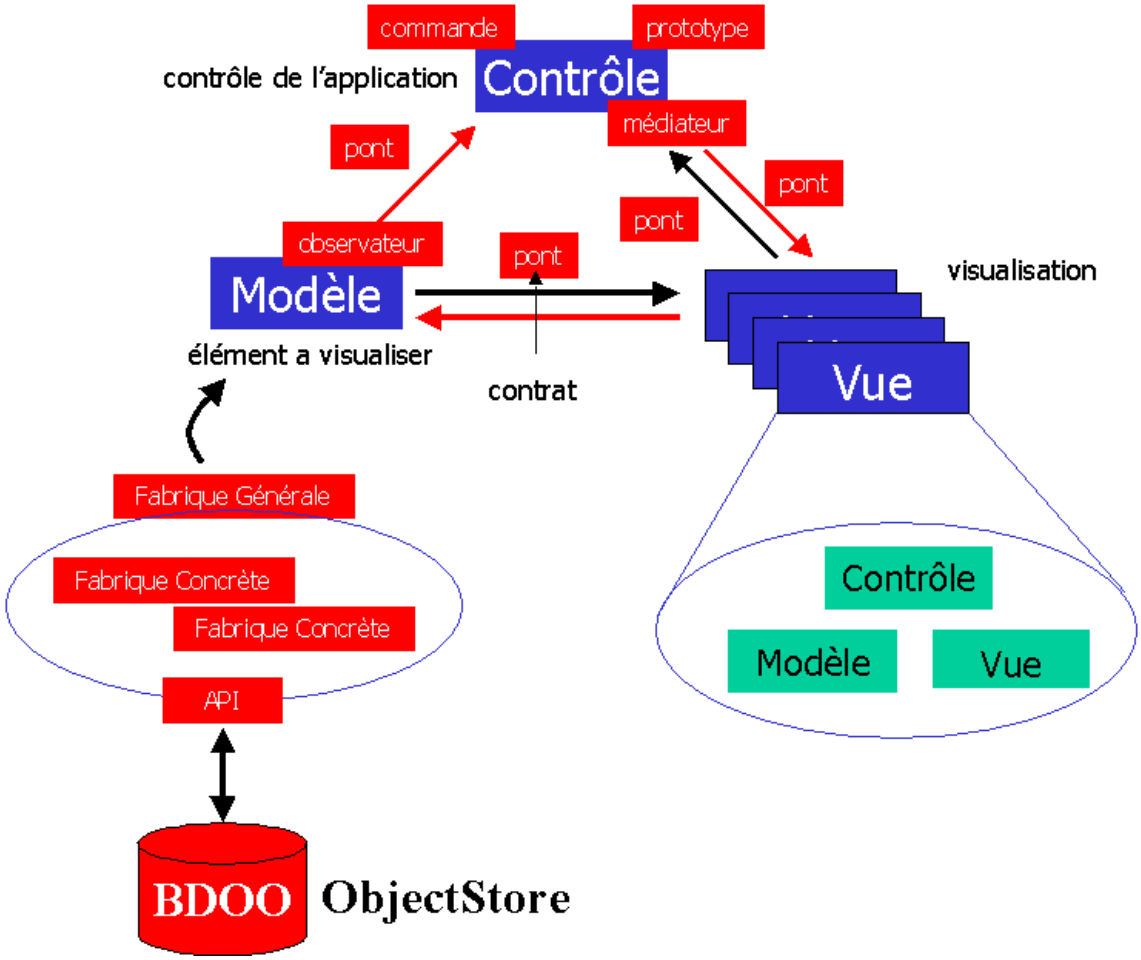


FIG. 7.8 – L'architecture logique de l'OMVT

Sans rentrer dans les détails, l'architecture logique peut être expliquée par les trois entités du patron de conception MVC :

- **Le modèle** : le modèle de l'application correspondant à l'ensemble des données que l'utilisateur souhaite visualiser. Ces informations sont construites dynamiquement au fur et à mesure des requêtes de celui-ci. La création d'un modèle se fait par l'intermédiaire du patron de conception fabrique abstraite qui extrait les informations de notre base de données orienté objet ObjectStore. L'utilisation de ce patron de conception nous permet de rendre l'application indépendante de cette base de données. Il est possible de changer simplement la version de notre base de données. Une API (Application Program Interface) a été construite au dessus de la base de données pour factoriser les requêtes courantes et faciliter son évolution et sa maintenance . Une documentation importante sur cette API, générée par l'outil javadoc de SUN, a permis de

l'utiliser simplement [Ang00]. Un système de cache a été mis en place pour gérer le chargement des gros volumes de données. Il améliore de manière significative la performance en terme de rapidité de notre application. La partie modèle dispose aussi du patron de conception observateur pour assurer la mise à jour des vues lors d'une modification du modèle.

- **Les vues** : les vues définissent une représentation concrète du modèle pour l'utilisateur. Elles s'appuient sur la syntaxe graphique que nous avons définie. Une vue est un composant à part entière qui est défini par une architecture MVC de deuxième niveau. Cela nous permet de créer des vues textuelles et graphiques qui utilisent des composants graphiques des packages AWT et Swing de java. L'implémentation de l'ensemble des vues s'est faite de manière concurrente, chaque vue étant sous la responsabilité d'un unique développeur. Pour intégrer une nouvelle vue à l'application, il suffit de la déclarer par l'intermédiaire d'un fichier d'initialisation que l'application parcourt à son lancement. L'application offre aussi une fonctionnalité pour rajouter une vue dynamiquement au cours de l'exécution.

- **Le contrôleur** : le contrôleur de l'application possède la connaissance du système. Il gère le cycle de vie des composants modèles et vues et l'interaction avec l'utilisateur. C'est lui qui permet d'assembler les morceaux et de vérifier la cohérence des actions à réaliser. Il joue le rôle de médiateur entre les différentes vues et administre les fonctionnalités de navigation de l'outil. C'est un composant complexe qui utilise plusieurs patrons de conception : commande, médiateur et prototype [GHJV95].

Ces trois entités sont reliées par des protocoles de communication précis qui définissent une sorte de contrat d'interaction. Ces contrats peuvent être vus comme des connecteurs et sont réalisés par le patron de conception pont [GHJV95].

L'implémentation de cette architecture logique est réalisée à travers un framework (dans le sens de Johnson [Joh92] et non de l'architecture V5) qui contient un ensemble de classes abstraites spécialisables par les développeurs pour réaliser leurs vues. Ce framework permet de réaliser et d'intégrer très rapidement de nouveaux composants dans l'architecture. Grâce au développement concurrent des six programmeurs, nous avons pu réaliser 22 vues locales et 5 vues globales en un peu moins de 2 mois.

Le conception et le développement complet du prototype ont duré à peu près 5 mois et représentent :

- 37 scripts perl et 1 fichier java totalisant 5948 lignes de code et de commentaires pour la création de la base de données,

- 54 fichiers java totalisant 5 465 lignes de code et de commentaires pour l'API,

- 194 fichiers java totalisant 28 328 lignes de code et de commentaires pour l'application.

7.1.4 Evaluation

A la suite du développement de l'OMVT, nous avons réalisé plusieurs démonstrations chez Dassault Systèmes. Un total de 34 acteurs, correspondant aux différents profils existants, ont pu assister à ces présentations et utiliser l'outil. Ils ont pu faire part de leur satisfaction et proposer des améliorations. Ces acteurs ont particulièrement apprécié le fait de pouvoir visualiser pour la première fois des composants Object Modeler et ont émis des commentaires encourageants : "très prometteur pour contrôler la définition d'un composant", "permet de mieux comprendre l'architecture globale et de situer son travail dans cette architecture", "syntaxe graphique très appréciable et intuitive, pourrait être adoptée par l'ensemble de Dassault Systèmes", "fournit à la fois des vues globales et détaillées qui ne sont pas disponibles actuellement sans parcourir de nombreux fichiers"...

Le bon accueil de l'OMVT par les acteurs de Dassault Systèmes peut s'expliquer par une bonne compréhension de leur problématique. L'analyse des besoins a permis de bien modéliser les différents acteurs de Dassault Systèmes ainsi que les problèmes auxquels ils étaient confrontés. Il a ainsi été possible de développer des fonctionnalités qui répondent à des besoins concrets. L'utilisation d'une syntaxe graphique claire et intuitive ainsi que la disponibilité de fonctionnalités de navigation ont contribué au succès de l'outil.

7.2 Generic Software Exploration Environment (GSEE)

Grâce à l'OMVT, nous avons pu répondre à une partie des besoins architecturaux de Dassault Systèmes. Bien que ce prototype ait été conçu pour être évolutif, en pouvant rajouter de nouvelles vues architecturales en quelques jours, il ne permet pas de couvrir l'ensemble des besoins architecturaux des différents acteurs. Il n'est par exemple pas possible de visualiser de gros graphes de données ou d'expérimenter de nouvelles vues en quelques minutes. Pour disposer de ces fonctionnalités, nous avons utilisé un environnement pour l'exploration des logiciels nommé GSEE [Fav01]. Cet outil a été développé par Jean-Marie Favre, un collègue de l'équipe de recherche Adèle.

Dans le cadre de notre collaboration avec Dassault Systèmes, cet environnement nous a été très utile car il nous a permis d'explorer l'architecture V5 et de réaliser de nombreuses expériences à un coût très faible (cf. section 7.2.3). Comme nous avions travaillé essentiellement sur l'architecture logique avec l'OMVT, nous avons profité de GSEE pour réaliser des expérimentations sur les architectures physique et produit. De plus, nous avons pu visualiser des entités et des dépendances liées à différentes architectures dans une même vue. GSEE a permis de répondre à des besoins autres que ceux pris en compte par l'OMVT (cf. section 7.2.1).

7.2.1 Analyse des besoins

Avec l'OMVT, nous avons réalisé un outil qui permet de répondre à des besoins clairement identifiés. Nous avons développé un certain nombre de vues prédéfinies qui suivent une syntaxe graphique spécifique. Ces vues architecturales ne sont pas destinées à une personne particulière mais plutôt à un grand nombre d'acteurs.

Bien que l'OMVT ait été bien accueilli par les différents acteurs, il n'aborde pas certains besoins de Dassault Systèmes :

- **Répondre à un besoin précis et ponctuel** : Par la diversité des travaux existant à Dassault Systèmes, les ingénieurs ont aussi des besoins propres et momentanés. C'est-à-dire, qu'un ingénieur peut avoir besoin d'une vue architecturale qui n'intéresse que lui et qui peut n'être utile qu'une seule fois ou très peu de fois.

- **Vues globales sur l'architecture** : Avec l'OMVT, nous avons fourni principalement des vues locales pour des instances. Mais, il est aussi intéressant de posséder des vues globales pour analyser et vérifier des propriétés sur l'ensemble de l'architecture. Ces vues globales permettent aussi de visualiser des liens de dépendance liés à l'interaction entre les différents composants.

- **Prototypage et expérimentation pour de nouvelles vues** : Comme pour la réalisation d'un logiciel, il est intéressant de pouvoir créer des prototypes de vues architecturales et de les montrer aux acteurs concernés pour vérifier qu'elles correspondent bien à leurs attentes. Une fois que les vues architecturales ont été approuvées par les acteurs, il est possible de passer à l'implémentation concrète de celles-ci. Si la réalisation de ces prototypes de vues architecturales demande un faible effort, cela permet de tester de nouvelles vues qui n'ont pas forcément été demandées et qui peuvent tout de même être utiles.

- **Faciliter le dialogue entre les acteurs** : Le processus de développement pour l'architecture V5 correspond à un certain nombre d'étapes effectuées par des acteurs différents. Du coup, l'interaction entre ces acteurs doit pouvoir être efficace. Pour cela, il faut que chacun puisse comprendre le contexte des autres et le lien avec le sien. Il est donc intéressant de pouvoir réaliser des vues qui mélangent des entités de différentes architectures qui correspondent aux intérêts des différents acteurs.

Bien que nous ayons conçu pour l'OMVT un framework objet qui permette d'intégrer rapidement une nouvelle vue, cela nécessite une certaine connaissance de ce framework et quelques centaines de lignes de code java. Ceci est adapté pour réaliser une vue prédéfinie et qui correspond à un besoin durable et/ou multiple. Par contre, cela ne l'est pas pour explorer un logiciel à la demande pour des raisons de coût de développement (même si celui-ci est de quelques jours).

7.2.2 Environnements pour l'exploration des logiciels

Dans les domaines de la compréhension, la maintenance et l'évolution des logiciels, de nombreuses recherches ont porté sur l'ingénierie inverse et l'exploration des logiciels. Plusieurs prototypes ont été proposés et utilisés dans différents projets. Ces prototypes peuvent être classifiés de la manière suivante :

- **Exploration sur base d'un méta-modèle fixe** : les prototypes de ce type sont prévus pour explorer des logiciels qui sont conformes à un méta-modèle prédéfini. En général, ce méta-modèle correspond à un langage de programmation particulier. Par exemple, le produit industriel imagix [Ima] permet de visualiser en trois dimensions le code source des logiciels écrits en C.

- **Exploration sur base d'un méta-modèle extensible** : comme pour les prototypes précédents, l'exploration est liée à un méta-modèle prédéfini. Par contre, ils sont prévus pour évoluer et l'extension du méta-modèle se fait généralement assez simplement. Nous retrouvons dans ce type de prototype l'environnement FAMOOS [DD99] qui possède un méta-modèle extensible pour l'orienté objet.

- **Exploration indépendante du méta-modèle** : à la différence des précédents, ces prototypes ne sont pas dépendants d'un méta-modèle quelconque. Ils permettent d'explorer n'importe quel logiciel tant que nous leur indiquons le méta-modèle de celui-ci. Ces prototypes sont donc plus génériques avec la contrainte de fournir explicitement les relations que nous voulons visualiser. Dans cette catégorie, nous retrouvons entre autres les outils Rigi [MTO+92, Rig] et PBS [FHK+97, PBS].

Au niveau de cette classification des différents environnements pour l'exploration des logiciels, nous pouvons remarquer une progression en ce qui concerne la généricité des outils. L'environnement GSEE, développé au sein de notre équipe de recherche par Jean-Marie Favre, se distingue de ces autres environnements par le fait qu'il offre en plus un nouveau degré de liberté. L'approche de GSEE est de pouvoir découvrir un logiciel comme un explorateur qui part à l'aventure [Fav02]. GSEE permet non seulement d'explorer les instances d'un logiciel mais aussi son méta-modèle. A la différence des autres environnements, il n'est pas besoin de connaître le méta-modèle sous-jacent de l'application pour pouvoir l'explorer. GSEE se base sur l'introspection[2] de java pour découvrir dynamiquement ce méta-modèle. Il est ainsi possible de visualiser n'importe quelle entité et relation sans avoir à rajouter une seule ligne de code ou à fournir la description de celles-ci. GSEE permet aussi de découvrir ces entités et relations au fur et à mesure de l'exploration du logiciel.

Avant le développement de GSEE, un étudiant de DEA de notre équipe de recherche a réalisé des expérimentations pour l'exploration de l'architecture de

[2]L'introspection objet est un mécanisme qui permet de découvrir à l'exécution des informations sur les classes des objets instanciés

CATIA V5 avec l'outil Rigi [Ngu00]. Il a été possible d'effectuer un certain nombre d'analyses sur les liens de dépendance entre les frameworks et de proposer des mesures de couplage et de cohésion. Par contre le principal inconvénient que nous avons rencontré avec Rigi est qu'à chaque fois que nous voulions visualiser une nouvelle entité ou relation, il fallait la décrire dans le format de fichier de Rigi. Il faut donc connaître ce format de fichier et passer un temps non négligable pour réaliser la description en question.

Dans le cadre de cette thèse, nous avons choisi d'utiliser l'environnement GSEE car il nous fournissait l'ensemble des fonctionnalités nécessaires pour répondre aux besoins décrits dans la section précédente. La propriété de pouvoir générer des vues à la demande par l'intermédiaire d'un langage de requête et sans avoir à écrire une seule ligne de code a été d'un grand apport dans nos expérimentations et a permis un gain de temps conséquent.

7.2.3 Exploration de l'architecture V5 avec GSEE

Dans ce rapport, nous ne présenterons l'environnement GSEE que dans le contexte de Dassault Systèmes et l'architecture V5. Pour une description plus complète de cet environnement et de son utilisation, vous pouvez vous référer à l'article qui lui est consacré [Fav01].

Comme nous l'avons déjà indiqué, GSEE nous a permis de réaliser un grand nombre d'expérimentations. Nous allons maintenant présenter certaines de ces expérimentations qui ont consisté à explorer et améliorer nos connaissances sur les trois structures architecturales que nous avons étudiées (architecture logique, architecture physique et architecture produit) ainsi que sur l'interaction entre ces différentes structures architectures.

Exploration de l'architecture logique

Les premiers essais que nous avons effectués avec GSEE ont porté sur l'architecture logique. Nous en avons profité pour apprendre à manipuler cet environnement et nous avons généré des vues qui ne sont pas disponibles avec l'OMVT.

Nous nous appuierons sur la figure 7.9 pour présenter rapidement GSEE. Le formulaire situé en en-tête de l'environnement permet de spécifier notre requête ainsi que le rendu de la visualisation. Le premier champ (pour le type) et le troisième champ (pour le modèle) servent à définir un filtre sur le modèle de données. Il est ainsi possible de spécifier quels types de données nous souhaitons visualiser et les liens de dépendance entre ces données. Dans l'exemple, nous demandons de prendre en compte les interfaces Object Modeler (champ type) avec la relation d'héritage Object Modeler (champ modèle). Le champ entité est un filtre sur les instances. Dans l'exemple, nous demandons de n'afficher que les interfaces Object Modeler dont le nom commence par CATIA. Nous pouvons noter au passage que nous retrouvons les deux filtres (sur le modèle et sur les instances) qui permettent

de spécifier les intérêts spécifiques de chaque acteur (cf. section 6.2.2). Les deux derniers champs ont pour rôle de décrire le rendu des nœuds du graphe ainsi que des paramètres pour la gestion du placement. Concernant le rendu d'un nœud, il est possible de définir trois métriques simples : la largeur, la hauteur et la couleur du nœud. Dans notre exemple, la hauteur et la largeur représentent le nombre d'implémentation pour cette interface et la couleur spécifie dans quel framework est situé l'interface.

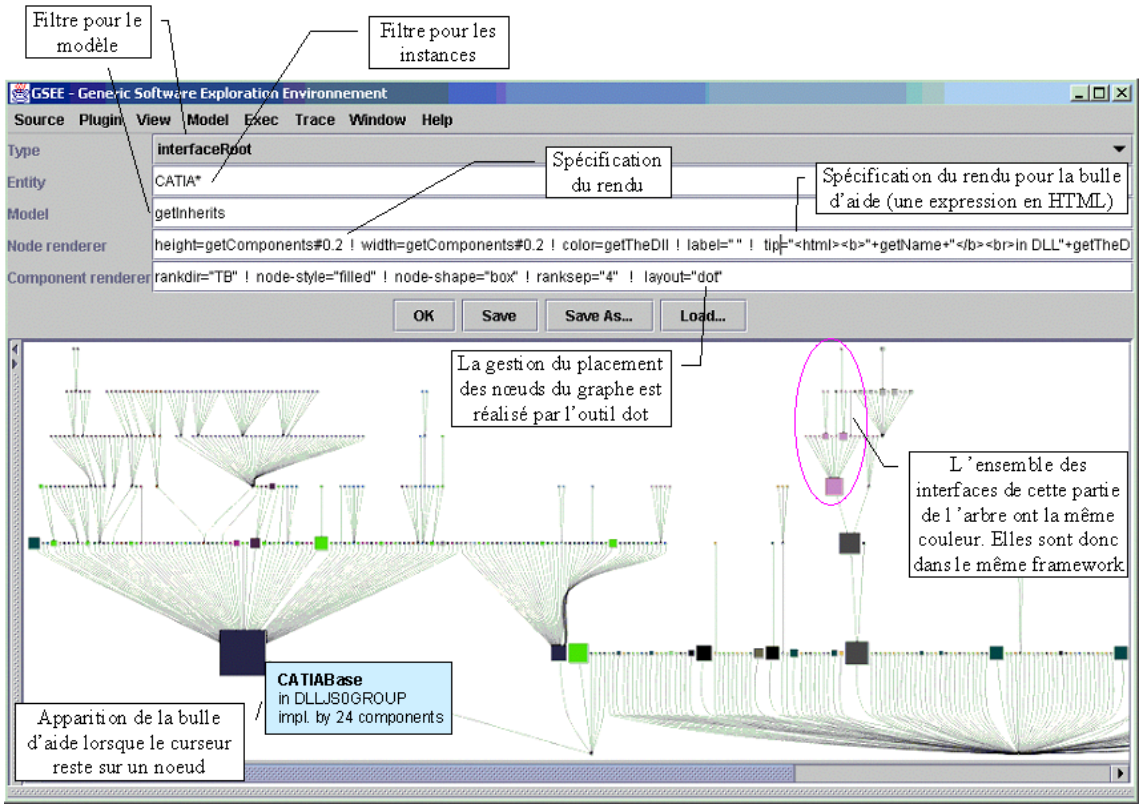


FIG. 7.9 – Arbre d'héritage Object Modeler des interfaces

La connexion de nos données architecturales de CATIA V5 avec GSEE a pu être réalisée simplement par le fait que GSEE est indépendant du modèle de données. En effet, GSEE s'appuie sur l'introspection de java ce qui lui permet de retrouver dynamiquement toutes les informations nécessaires. Notre base de données ObjectStore et notre API pour l'exploiter étant implémentées en java, la connexion avec GSEE n'a pas posé de problème.

Comme nous pouvons le constater dans la figure 7.9, GSEE permet de visualiser des grandes quantités d'informations. Cette vue représente 529 interfaces sous forme d'un arbre. Grâce aux métriques que nous avons spécifiées, nous apercevons immédiatement les interfaces les plus implémentées, ainsi que leur place dans l'arbre d'héritage. De plus nous pouvons remarquer que les interfaces qui sont entourées par l'éclipse ont la même couleur, ce qui reflète une bonne cohésion pour ce framework (il contient une partie entière de l'arbre d'héritage). Pour information, le temps im-

parti pour spécifier cette requête et générer la vue correspondante est de quelques minutes.

Exploration de l'architecture physique

Comme l'architecture physique est liée à la compilation et à la modularisation, nous nous sommes attachés à représenter les liens de build-time entre les frameworks (cf. section 3.1.2).

La figure 7.10 affiche des liens de dépendance de compilation entre différents frameworks. Dans cet exemple nous nous sommes basés sur le fait qu'il existe un lien de dépendance de compilation entre deux frameworks lorsqu'une extension Object Modeler du framework dépendant implémente une interface Object Modeler de l'autre framework. Le framework MechanicalModeler est le framework initial à partir duquel nous avons commencé le calcul des liens de dépendance de compilation. Pour la représentation du graphe, nous avons défini deux métriques sur les nœuds : la largeur d'un nœud correspond au nombre d'interfaces que possède le framework et la hauteur de ce nœud correspond au nombre d'extensions de ce même framework. Il est ainsi possible de visualiser rapidement les frameworks qui ont pour objectif de définir des interfaces (les nœuds qui sont étirés horizontalement) des frameworks qui regroupent des implémentations (les nœuds qui sont étirés verticalement).

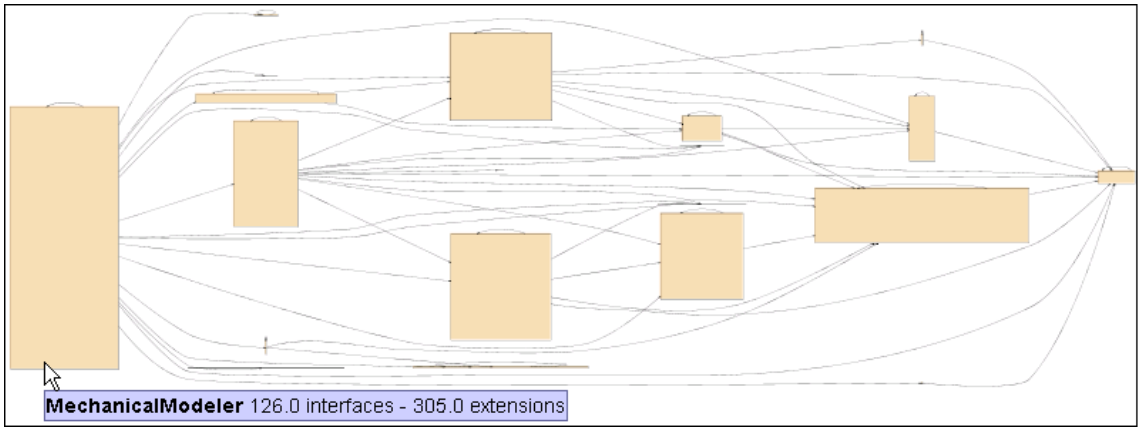


FIG. 7.10 – Liens de dépendance de compilation entre des frameworks

Ce type de figure permet d'expliquer l'origine de certains liens de dépendance de compilation (la relation prereq entre deux frameworks dans l'architecture physique) indiqués dans la carte d'identité des frameworks. Cette figure serait encore plus pertinente s'il était possible de rajouter une nouvelle métrique sur les liens entre les nœuds. Il serait en effet intéressant de pouvoir indiquer le nombre d'occurences pour un lien de dépendance entre deux frameworks. Pour la représentation graphique, plus le nombre d'occurences serait important plus le lien entre les deux nœuds serait épais. Cette information serait utile pour connaître le couplage entre les différents frameworks. Cela donnerait une bonne indication pour savoir s'il est facile ou non

de supprimer une relation de dépendance de compilation entre deux frameworks et aiderait pour le travail de restructuration. Malheureusement, GSEE ne permet actuellement pas de définir de telle métrique et cette fonctionnalité est prévue pour une version ultérieure.

Exploration de l'architecture produit

Au niveau de l'architecture produit, nous avons cherché à représenter la structure hiérarchique des composants de packaging ainsi que les différents types de liens de dépendance entre eux. A la différence de l'OMVT, qui fournit une vue complète de cette structure sour forme d'un arbre grâce au composant Swing JTree, nous pouvons avec GSEE représenter cette structure de manière beaucoup plus exploitable. Nous avons profité de la représentation en graphe qui permet de représenter de gros volumes de données, de la possibilité de visualiser avec des couleurs différentes plusieurs relations et de la disponibilité de métriques qui permettent de faire ressortir des informations pertinentes.

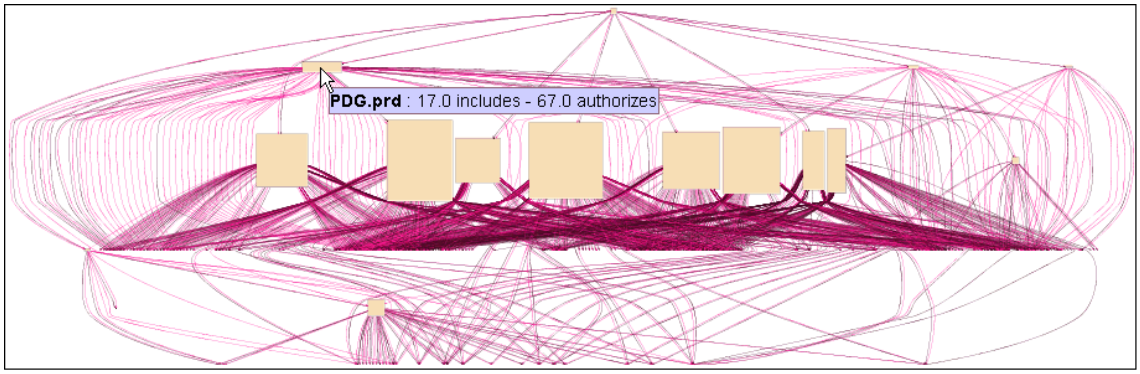


FIG. 7.11 – Liens de dépendance de type inclusion et autorisation entre des composants de packaging

Dans la figure 7.11, nous sommes partis de la solution nommée HD2 (le sommet du graphe) et nous avons parcouru de manière transitive les liens de dépendance de type inclusion (liens en noir) et de type autorisation (lien en rose). Sur chaque composant de packaging (nœud du graphe) nous avons ajouté deux métriques : la largeur du nœud représente le nombre de dépendances de type autorisation et la hauteur correspond au nombre de dépendances de type inclusion. Ainsi les nœuds étirés horizontalement ont plus de liens de type autorisation et inversement les nœuds étirés verticalement ont plus de liens de type inclusion.

Dans la figure, il est possible de distinguer plusieurs niveaux. Le premier (la racine de l'arbre) correspond au nœud initial qui est la solution HD2. Le deuxième représente des produits, comme le produit PDG, qui ont plus de liens de type autorisation. Le troisième niveau représente au contraire les produits qui ont plus de liens de type inclusion. Le quatrième niveau est dédié aux frameworks. Nous avons pu constater que souvent les deux types de liens entre les composants de packaging

existent en même temps. Par exemple, c'est le cas pour la solution HD2 qui a autant de relations de type inclusion que de relations de type autorisation sur les mêmes composants de packaging. Ceci nous a d'abord surpris puisque, comme nous l'avons expliqué dans la section 6.1.3, une relation de type autorisation implique une relation de type inclusion. Après s'être renseigné auprès d'un packaging manager, ceci s'explique par le fait que leur outil qui analyse les cartes d'identités des produits ne tient pas toujours compte de cette implication suite à des évolutions historiques, ce qui devrait être corrigé dans une prochaine release de CATIA V5.

Il est possible aussi de remarquer que les produits regroupent un grand nombre de frameworks. Ils ont donc de fortes chances d'être impactés par l'évolution constante du code source. Ceci explique le fait qu'il existe une équipe entière consacrée à cette structure architecturale pour s'assurer de la cohérence des différents composants de packaging et que les déploiement de ces derniers sur les postes des clients s'effectura sans problèmes.

Exploration de l'interaction des différentes architectures

Comme nous avons pu le souligner dans la partie précédente, les différentes structures architecturales ne sont pas indépendantes. Une modification dans une structure architecturale peut impacter d'autres structures architecturales. Il est donc important de ne pas seulement visualiser ces structures architecturales indépendamment mais aussi de montrer ces liens de dépendance entre elles.

Grâce aux fonctionnalités d'exploration de GSEE, nous avons aussi pu croiser des informations des trois structures architecturales que nous avons étudiées dans une même vue. Par exemple, la figure 7.12 montre l'intérêt d'une telle vue pour un chef de produit. En haut de la figure, nous pouvons visualiser les informations de l'architecture logique : un composant Object Modeler (la racine du graphe) ainsi que les interfaces qu'il implémente. Au milieu de la figure, nous voyons le lien avec l'architecture physique : chaque interface Object Modeler est définie par des fichiers de déclaration qui appartiennent à un framework. En bas de la figure, sont représentées les informations de l'architecture produit : comme nous l'avons décrit, un produit est un regroupement de frameworks. La figure affiche le lien entre le composant Object Modeler CATAxisImpl et les frameworks contenant les interfaces qu'il implémente ainsi que les produits contenant ces frameworks.

Le travail du chef de produit est de fournir pour son produit le maximum de fonctionnalités logiques avec le minimum de liens de dépendance entre les frameworks et entre les produits pour faciliter la maintenance et l'évolution. Ce style de vue est utile au chef de produit car elle lui fournit des informations nécessaires pour faire des choix dans ce sens. Elle est par exemple intéressante pour connaître l'impact, au niveau des architectures physique et produit, de l'implémentation d'une nouvelle interface ou encore de savoir si le fait de ne plus implémenter une interface permet de supprimer des liens de dépendance.

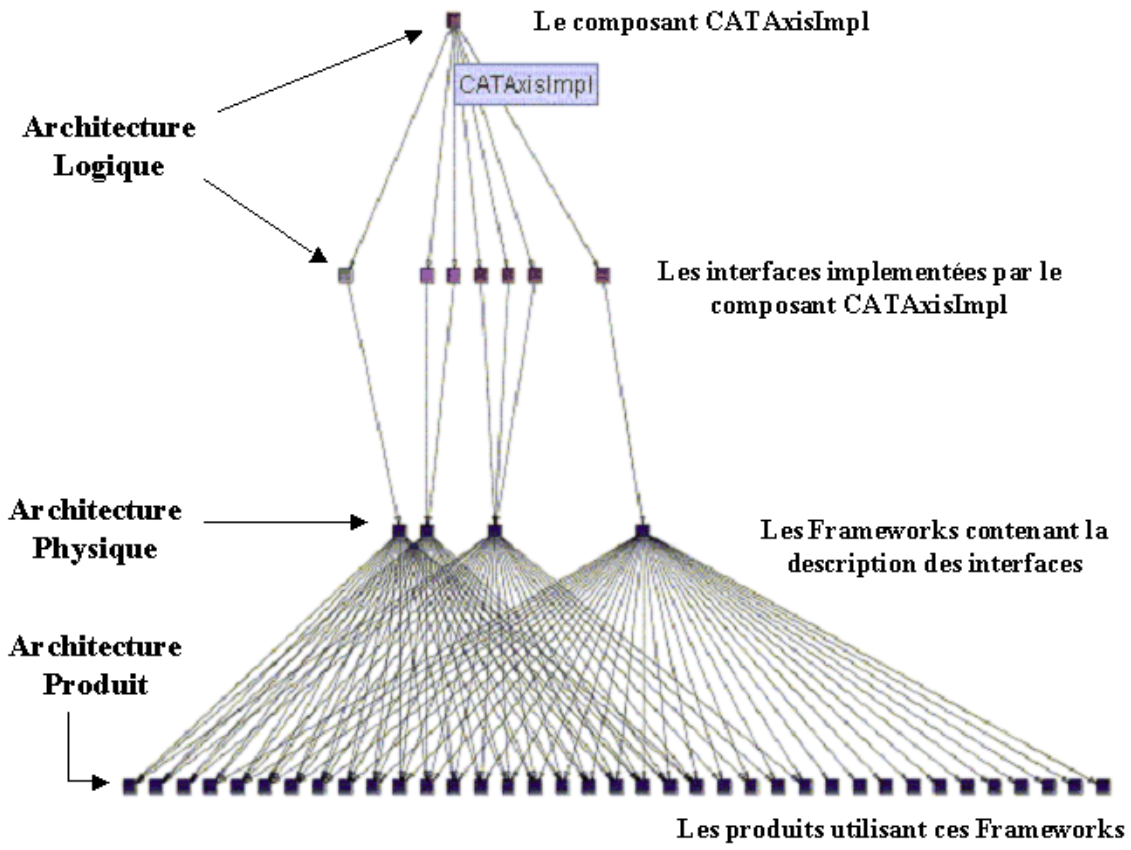


FIG. 7.12 – Représentation d'entités de différentes architectures

Aide à la communication

Une entité peut être utilisée dans différentes structures et être vue différemment selon le point de vue de l'acteur. Par exemple, le framework est un composant de compilation pour les release manager dans l'architecture physique alors que c'est un composant de packaging pour le packaging manager dans l'architecture produit. Ainsi, la communication entre les différents acteurs est importante et nécessaire. GSEE permet de faciliter cette communication en regroupant des intérêts de différents acteurs dans une même vue [SFL01]. Ces vues sont réalisées en utilisant des métriques architecturales utiles (bien que simples) similaires à celles utilisées au niveau du code source pour le projet FAMOOS [DD99]. Par exemple dans la figure 7.9, des intérêts de différents acteurs sont représentés : l'arbre d'héritage des interfaces Object Modeler ainsi que les interfaces fortement utilisées pour les développeurs les chefs de produit et les architectes ; les frameworks pour les release managers, les packaging managers et les architectes. Du coup, un chef de produit peut aisément expliquer à un release manager pourquoi il doit inclure un lien de dépendance de compilation pour un framework particulier. Un architecte peut aussi montrer pourquoi il est difficile de changer un lien dans l'arbre d'héritage.

7.2.4 GSEE et l'OMVT

L'outil GSEE n'a pas pour objectif de remplacer l'OMVT. Il doit plutôt être vu comme un outil complémentaire [SFL01]. En effet, ces deux outils ne sont pas faits pour les mêmes buts et ils ont chacun leurs avantages et inconvénients. Concernant l'OMVT, ses atouts sont sa simplicité d'utilisation et de compréhension grâce à l'utilisation d'une syntaxe graphique claire et précise qui correspond aux habitudes des utilisateurs, à la disponibilité de vues pré-définies et aux fonctionnalités de navigation. Par contre, en ce qui concerne GSEE, ses principaux avantages sont l'indépendance du méta-modèle, la possibilité d'afficher de gros volume de données et la création de nouvelles vues sans avoir à développer une ligne de code. Ainsi il est possible de prototyper rapidement de nouvelles vues architecturales avec GSEE. Si certaines vues correpondent à un besoin récurent, il est possible de les implémenter et de les intégrer dans l'OMVT en utilisant la syntaxe graphique préféfinie.

7.3 Simulation d'analyses d'impact

Notre troisième prototype est un outil de simulation qui prend en entrée un certain nombre de données et qui fournit en sortie le résultat d'un calcul complexe. Par simulation, nous entendons le fait de simuler une action dans le processus de développement comme modifier une entité architecturale et non dans le sens de simuler une exécution de CATIA V5 comme pourrait le faire l'ADL Rapide.

La maintenance et l'évolution d'un logiciel aussi complexe que CATIA est véritablement un défi. Par la taille et le développement concurrent, il est très difficile voire impossible pour un ingénieur de Dassault Systèmes de se rendre compte de l'impact d'une modification sans outil approprié.

Au fur et à mesure de nos travaux, nous nous sommes rendu compte de l'importance de ce type d'outil. Nous avons donc décidé de continuer nos recherches dans cette direction. Nous avons tout d'abord mené plusieurs entretiens pour répertorier les besoins spécifiques en termes d'analyse d'impact (cf. section 7.3.1). Nous avons ensuite réalisé un premier prototype qui nous a permis de bien comprendre les concepts et les fonctionnalités pour ce type d'outil (cf. section 7.3.3).

Même si par manque de temps nous n'avons pu réaliser qu'une première ébauche de ce prototype, cela nous a permis d'identifier des concepts importants pour ce type d'outil et de pouvoir proposer des améliorations à partir des permiers résultats obtenus (cf. section 7.3.4).

7.3.1 Analyse des besoins

Avec l'OMVT et GSEE, nous avons abordé des besoins liés à la représentation et la compréhension des différentes structures architecturales de CATIA V5. Ici, nous nous intéresserons aux besoins liés à l'analyse d'impact pour la maintenance et l'évolution de CATIA V5. Dans un contexte comme celui de Dassault Systèmes où il existe de fortes contraintes de développement (cf. chapitre 3) ce type de besoin est important. Quand nous avons à faire à des logiciels qui font plusieurs millions de lignes de code, il n'est pas possible de connaître l'impact d'une modification sans outil adapté. En général, le développeur livre son code et attend les résultats de la compilation et des tests. Dans le cadre de Dassault Systèmes, ce processus est long car il existe plusieurs niveaux de compilation et de tests dans l'organisation de la société (niveau équipe et niveau département). Du coup, il se peut qu'il faille attendre plusieurs jours avant de connaître l'impact d'une modification.

Suite au choix d'orienter ma thèse sur des aspects d'analyses d'impact, nous avions décidé de répertorier en premier lieu les simulations intéressantes pour Dassault Systèmes. Ensuite nous n'avons implémenté qu'une partie d'entre elles en fonction du temps disponible. Afin de nous concentrer sur des simulations utiles et représentatives pour Dassault Systèmes, nous avons rencontré plusieurs acteurs possédant des profils (et donc des besoins) différents : développeurs, chefs de produit, architectes, packaging managers...

Les acteurs de Dassault Systèmes réalisent un certain nombre d'opérations qui leur sont propres. Les principales opérations qui nous concernent sont des opérations d'édition. Nous les avons regroupées en six grandes catégories : création, renommage, déplacement, découpage, fusion et suppression.

Concernant l'opération de création, il semble qu'elle soit plutôt bien maîtrisée par les différents acteurs. Cela vient du fait que la complexité est largement moindre par le fait que l'entité qui doit être créée n'est pas encore utilisée par d'autres entités. Il est donc beaucoup plus facile de calculer son impact sur l'architecture. Par contre, il existe véritablement des besoins d'analyse d'impact pour les autres opérations.

Pour des questions de lisibilité, nous allons regrouper les différents types de scénarii selon les différents acteurs. Dans le cadre de ce rapport de thèse, nous ne présenterons qu'une partie des acteurs, ce qui est largement suffisant pour se donner une bonne idée des besoins.

Développeur

Opérations :

- *Créer, Renommer, Déplacer, Découper, Fusionner, Supprimer*

Données manipulées :

- *Entités liées à l'architecture logique (interface, composant, extension...)*
- *Entités liées à l'architecture physique (module, Imakefile.mk...)*
- *Fichiers liés à la programmation (.cpp, .h, .java, .idl, .dll...)*
- *Fichiers de ressources (images, messages d'erreurs...)*

Le développeur peut souhaiter éditer différents types d'entités qui lui appartiennent. Ce peut-être un nom de fichier (DLL, fichier C++, fichier IDL...), un nom de répertoire (module), une entité Object Modeler (interface, composant, extension...).

Exemple :
Le développeur veut déplacer une extension d'un module vers un autre module. Dans ce cas, il faut mettre à jour les fichiers Imakefile.mk des deux modules en question ainsi que les modules qui utilisaient cette extension. De plus, si les modules ne sont pas dans le même framework, il va falloir faire de même pour les cartes d'identité.

Chef de Produit

Opérations :

- *Créer, Renommer, Déplacer, Découper, Fusionner, Supprimer*

Données manipulées :

- *Entités liées à l'architecture physique (carte d'identité de ses frameworks)*
- *Entités liées à l'architecture produit (carte d'identité de son produit)*

Bien que le chef de produit puisse jouer le rôle de développeur, il a comme particularité la gestion de son produit. Cela implique qu'il souhaite éditer les cartes d'identité de ses frameworks et de son produit.

Exemple :
Si un développeur a besoin d'une extension qui se trouve dans un framework qui n'est pas indiqué dans la carte d'identité du framework de l'extension, alors son chef de produit doit rajouter ce nouveau lien de dépendance dans la carte d'identité du framework de l'extension. Par transitivité, il va aussi devoir ajouter ce lien de dépendance dans la carte d'identité de son produit et avertir les autres chefs de produit qui utilisent son framework ou produit de mettre à jour leur carte d'identité pour leur produit.

Packaging Manager

Opérations :
- *Créer, Renommer, Déplacer, Découper, Fusionner, Supprimer*

Données manipulées :
- *Entités liées à l'architecture produit*
  - *média, solution, produit, framework*
  - *licences*
  - *prix*
- *Fichiers pour le déploiement (DLL, fichiers de ressource...)*

La fonction d'un packaging manager étant de créer et de déployer des composants de packaging chez les clients, il manipule concrètement les entités de l'architecture de packaging. Il veut pouvoir éditer les cartes d'identité de ses composants de packaging.

La principale difficulté pour un packaging manager est de créer un media où il ne manque aucun fichier nécessaire au bon fonctionnement des applications du média. Il doit gérer les différentes entités liées au déploiement.

Exemples de scénario :
Que se passe-t-il si je rajoute ou j'enlève un produit dans ma solution ?
Si le client souhaite un nouveau produit. Quelle(s) licence(s) dois-je lui fournir ?
Si le client veut un certain nombre de produits, quel est le composant de packaging (média, solution) le plus approprié ?

7.3.2 Etat de l'art

Au niveau de la littérature académique, nous n'avons recensé qu'un seul travail similaire pour l'analyse d'impact au niveau architectural. Il correspond à la thèse de J. Stafford [Sta00] et de son outil Aladdin [Sta98, Sta00, SW98].

Aladdin est un outil d'analyse de dépendance au niveau architectural. Il est basé sur la spécification des chaînes de dépendance entre les différents éléments architecturaux.

L'auteur considère quatre types d'entités dans une architecture logicielle :

- **Les éléments de données** : qui contiennent les données qui sont utilisées par les composants (données de transformation, données de communication...).
- **Les éléments de traitement** : qui expriment le comportement des composants pour la transformation des données.
- **Les éléments de connexion** : qui représentent le comportement des composants pour l'interaction. Ces éléments sont aussi appelés ports.
- **Les éléments d'états** : qui fournissent l'état courant des composants en termes à la fois de données et de traitement.

Ils prennent en compte trois types de chaîne de dépendance pour un composant C :

- **Les chaînes "Affected-by"** : qui consistent en l'ensemble des composants et/ou leurs éléments qui peuvent affecter un élément de C. Ce sont les éléments qui affectent C ("C is affected by").
- **Les chaînes "Affects"** : qui consistent en l'ensemble des composants et/ou leurs éléments qui peuvent être affectés par C. Ce sont les éléments qui sont affectés par C ("C affects").
- **Les chaînes "Related-to"** : qui regroupent l'ensemble des composants et/ou leurs éléments qui peuvent affecter ou être affectés par un élément de C. Cette chaîne est la combinaison de deux précédentes pour les éléments de C.

L'architecture de l'outil Aladdin peut se résumer ainsi (cf. figure 7.13). Aladdin possède une première partie qui se charge de la lecture des spécifications architecturales. Actuellement, l'outil accepte le format de spécification de l'ADL Rapide. Le choix du format de l'ADL Rapide a été motivé par le fait qu'il a l'avantage de pouvoir décrire les relations structurales et comportementales. Néanmoins, il est possible d'utiliser d'autres formats de spécification tant que le parser pour le format souhaité est disponible. Dans une deuxième partie, un module va s'occuper de prendre l'arbre syntaxique abstrait généré par le parseur pour construire une table de dépendance dans un format spécifique. Cette table permet de représenter le graphe de dépendance entre les différents éléments de la spécification architecturale. Une troisième partie se charge de calculer les chaînes de dépendance à partir de cette table pour une requête particulière définie par le client. Une dernière partie est consacrée à l'interaction avec le client en lui offrant des fonctionnalités pour définir ses requêtes soit graphiquement, soit textuellement.

En ce qui nous concerne, nous n'avons pas pu utiliser l'outil Aladdin pour plusieurs raisons dont la principale est liée au passage à l'échelle. Comme nous l'avons expliqué dans la partie précédente, il ne nous était pas possible de décrire automatiquement le comportement de CATIA V5 dans un format tel que Rapide.

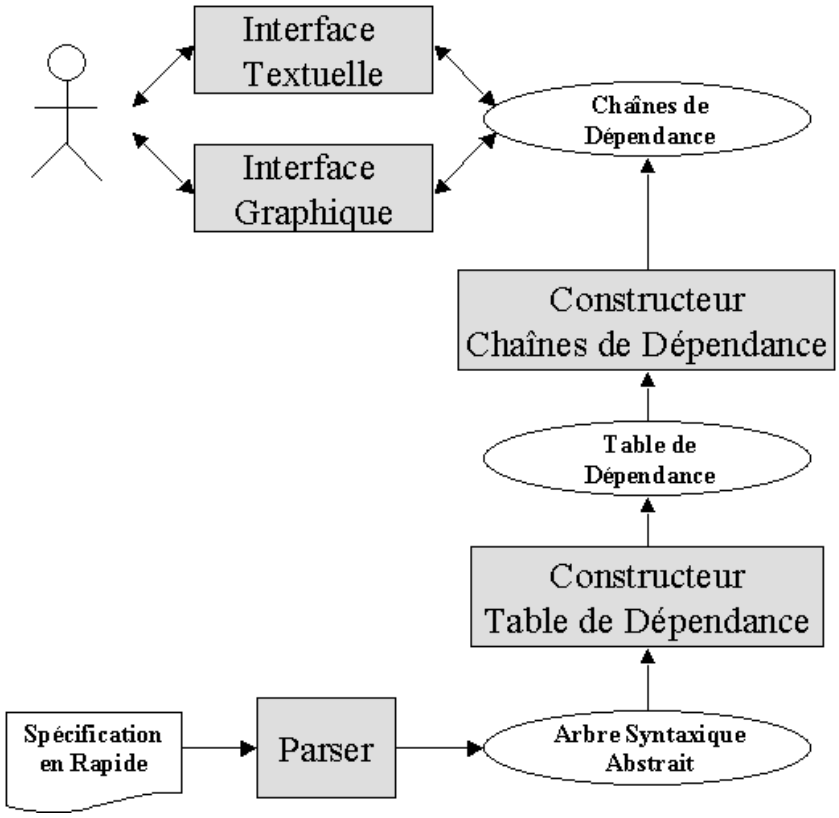


FIG. 7.13 – L'architecture de l'outil Aladdin

Nous aurions pu développer notre propre parser pour produire l'arbre de syntaxe abstraite et l'intégrer à Aladdin. Nous ne l'avons pas fait car il n'est pas réaliste de créer une table de dépendance complète pour un logiciel qui fait près de 5 millions de lignes de code et qui possède des millions de dépendances. Il faut aussi prendre en compte que CATIA V5 évolue très rapidement, ce qui demande de mettre à jour régulièrement cette table.

Nous avons aussi rencontré J. Stafford lors d'une conférence où elle nous a expliqué que l'outil Aladdin n'était plus maintenu et qu'il serait difficile de faire nos modifications à partir du code source. Il était donc plus simple de réaliser nous-même notre propre outil.

7.3.3 Présentation du prototype

Vu la diversité des acteurs et des besoins, nous avons adopté la même démarche que l'OMVT en concevant un framework évolutif plutôt que d'implémenter l'ensemble des besoins un à un. Ce framework fournit les mécanismes de base pour la gestion des analyses d'impact et est suffisamment flexible pour pouvoir intégrer facilement de nouvelles entités, de nouveaux scénarii de modifications, de nouvelles relations de dépendance et de nouvelles métriques.

Ce choix de disposer d'un framework évolutif nous assurait de pouvoir ajouter de nouveaux besoins que nous n'avions pas pris en compte. Comme nous n'avions pas le temps d'implémenter tous les scénarii, nous pouvions ainsi n'en réaliser que quelques-uns. Il était ainsi possible de vérifier que le prototype correspondait bien aux attentes des acteurs et de pouvoir suivre un cycle de vie en spirale.

Lors du développement de cet outil, nous avons identifié plusieurs étapes et concepts :

1. **Le choix de l'entité** : l'acteur doit pouvoir choisir l'entité sur laquelle il souhaite faire sa simulation. Comme nous l'avons déjà indiqué, cette sélection doit pouvoir se faire sur les deux critères que nous avons identifiés, i.e. un filtre sur le modèle et un filtre sur les instances. Actuellement, notre implémentation n'offre la possibilité de ne choisir que parmi les instances des différents composants Object Modeler (cf. figure 7.14).
2. **Le choix du scénario de modification** : l'acteur peut ensuite demander de réaliser une simulation d'analyse d'impact sur l'entité sélectionnée. Il doit pour cela spécifier la modification à simuler. Dans notre prototype, une fenêtre permet de réaliser ce choix (cf. figure 7.15). L'ensemble des scénarii sont regroupés par type d'acteur pour reprendre la classification utilisée pour l'analyse des besoins. Cela permet aussi à l'acteur de retrouver rapidement l'ensemble des actions qu'il a l'habitude de réaliser. Dans notre implémentation, il est possible de rajouter simplement et sans recompilation un nouvel acteur ou une nouvelle action par l'intermédiaire d'un fichier textuel de configuration.
3. **Description de l'action d'édition** : à la suite du choix de l'action à simuler, il faut fournir la nouvelle spécification relative à cette action. Pour simplifier notre implémentation, nous avons décidé de réaliser cette spécification à partir d'un fichier textuel plutôt que de modifier l'entité directement à la souris. L'utilisateur doit créer le fichier déclarant l'édition qu'il souhaite simuler et l'indiquer au prototype (cf. figure 7.15). Pour cela nous avons défini une syntaxe textuelle pour pouvoir décrire les entités de l'architecture V5 et implémenté l'analyseur correspondant.
4. **Le choix des liens de dépendance** : l'architecture V5 possède un grand nombre de structures et de type de liens de dépendance. L'utilisateur peut n'être intéressé que par certaines relations de dépendance particulières. Il doit donc pouvoir choisir ces relations de dépendance sur lesquelles il souhaite effectuer l'analyse d'impact. Notre framework permet de spécifier un ensemble de liens de dépendance pour chaque entité de l'architecture V5. Actuellement, notre prototype n'offre que la relation de dépendance pour l'héritage Object Modeler pour les composants Object Modeler.
5. **Analyse du résultat de la simulation** : Pour pouvoir juger de l'impact d'une modification, nous avons choisi de visualiser les impacts par des métriques prédéfinies. L'outil permet de calculer un certain nombre de métriques, relatives à l'entité choisie, avant et après la modification. L'utilisateur doit pouvoir choisir les métriques qui l'intéressent et qui seront appliquées lors de la simulation. Notre framework permet de spécifier un ensemble de

métriques pour chaque entité de l'architecture V5. Actuellement, notre prototype n'offre qu'un ensemble de métriques pour le TroubleShooter. Dans la figure 7.16, nous pouvons visualiser l'impact de la modification du composant CATViewer par rapport aux métriques de multi-adhésion. Dans cet exemple, il est possible de voir que la modification génère 2 multi-adhésions locales (i.e. au niveau du composant CATViewer) et 9 multi-adhésions globales (i.e. par la relation d'héritage Object Modeler de composant). Cette modification pourra donc provoquer une erreur à l'exécution suite aux 2 multi-adhésions locales et aura un impact au niveau de la conception suite aux 9 multi-adhésions globales. Nous pouvons en conclure que cette modification à des conséquences non négligeable au niveau de l'Object Modeler. Si le développeur juge qu'elle est indispensable, il faudra qu'il prenne en compte les impacts liés à modification.

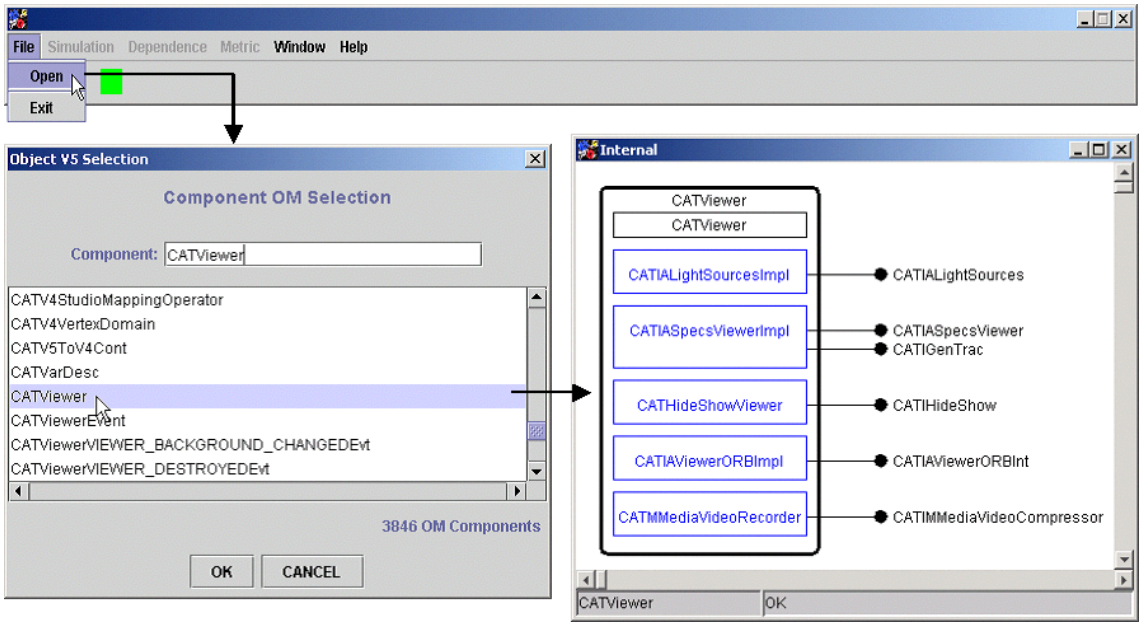


FIG. 7.14 – Choix du composant CATViewer

L'implémentation de notre framework et de notre prototype relève d'aspects techniques de programmation. Nous ne présenterons pas l'architecture de l'outil de façon détaillée mais nous pouvons indiquer que la modélisation est centrée sur les entités de l'architecture V5. Pour chacune de ses entités, il est possible de spécifier des relations de dépendance ainsi que des métriques qui lui sont associées. Plusieurs patrons de conception ont été utilisés. Comme pour l'OMVT, le prototype a été construit sur la base du patron MVC et des fabriques de classes permettent la création des instances du modèle à partir de notre base de données Object Store. Nous avons utilisé le patron observateur pour les mises à jour du modèle et le patron visiteur pour réaliser l'ensemble des calculs pour l'analyse d'impact sur les instances de ce modèle. C'est notamment grâce à ce patron visiteur qu'il est possible de rajouter simplement une nouvelle relation de dépendance ou une nouvelle métrique.

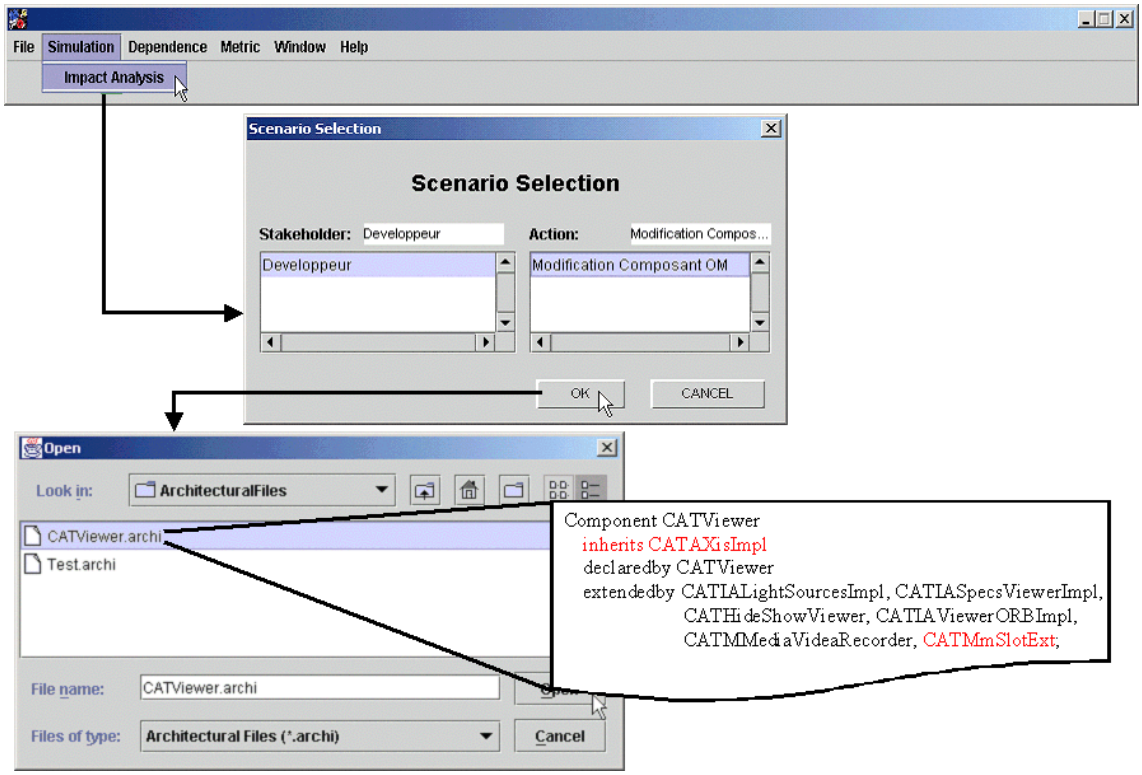


FIG. 7.15 – Choix du scénario de simulation et de la nouvelle spécification

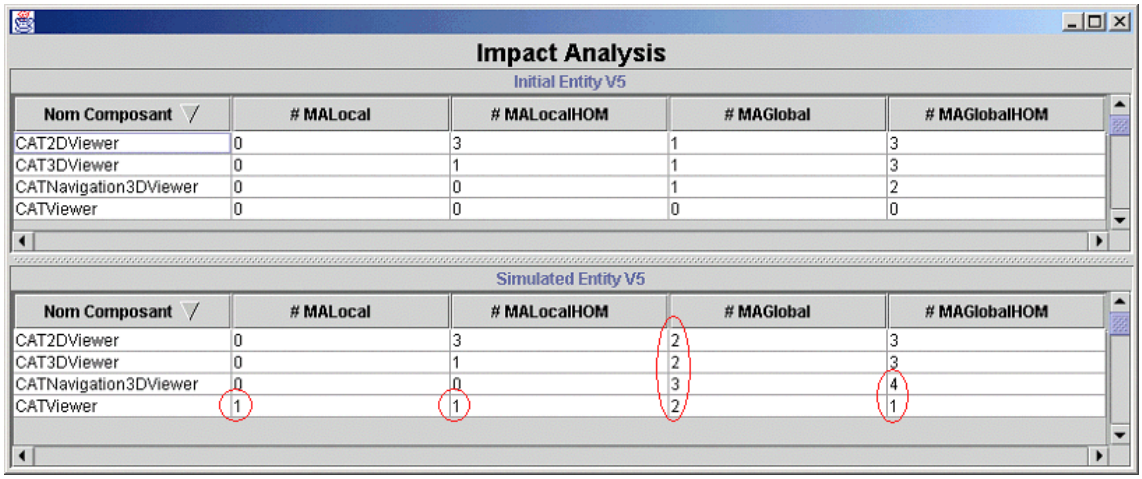


FIG. 7.16 – Résultat de la simulation d'analyse d'impact pour le composant CATViewer

7.3.4 Améliorations et perspectives

Bien que notre prototype soit fonctionnel, il peut encore être grandement amélioré et étendu. Nous pointerons ici trois types d'améliorations : mieux exploiter les ressources actuelles du framework, améliorer et ajouter de nouvelles fonctionnalités pour la gestion de la simulation d'analyse d'impact et étendre les fonctionalités de l'outil vers des aspects de restructuration.

Mieux exploiter les ressources actuelles du framework

Le manque de temps ne nous a pas permis d'exploiter entièrement les fonctionnalités de notre framework. Nous n'avons effectivement implémenté qu'une seule relation de dépendance et une seule métrique alors que le moteur d'analyse du framework permet d'en gérer autant que nous en voulons. Il serait intéressant de créer d'autres liens de dépendance entre les différentes structures architecturales. Nous avions le souhait de pouvoir montrer concrètement l'impact d'une modification de l'architecture logique sur l'architecture physique et transitivement sur l'architecture produit, impact que nous avions identifié et conceptualisé.

Améliorer et ajouter de nouvelles fonctionnalités pour la simulation d'analyse d'impact

A ce niveau, nous pouvons considérer les deux points suivants :

- **Améliorer la visualisation des résultats** : comme nous pouvons le constater dans la figure 7.16, il n'est pas facile d'interpréter les résultats et de se faire rapidement une opinion sur l'impact de la modification. Une première solution est de formater l'affichage afin d'en faciliter la lisibilité et l'interprétation des résultats. Différentes pistes peuvent être explorées : utilisation des couleurs pour indiquer si la modification a été bénéfique ou non (cf. figure 7.17), utiliser un certain nombre d'indicateurs (graduation du vert au rouge), n'afficher en premier lieu que les informations pertinentes... Une deuxième proposition consisterait à développer un module expert capable de fournir un diagnostic sur les résultats du calcul de l'analyse d'impact.

- **Réaliser plusieurs simulations successivement** : actuellement nous n'assurons l'exactitude des résultats que pour une simulation. Nous ne garantissons rien quant aux résultats de plusieurs simulations appliquées successivement. En effet, cela demande une gestion fine du modèle de données que nous n'avons pas eu le temps de mettre en place pour s'assurer que les nouvelles simulations prennent bien en compte le résultat des simulations précédentes. En plus de cela, il faudrait aussi ajouter les fonctionnalités de défaire/refaire ("undo/redo") sur les différentes simulations exécutées. Ceci permettrait à un acteur d'effectuer plusieurs essais pour trouver la meilleure configuration.

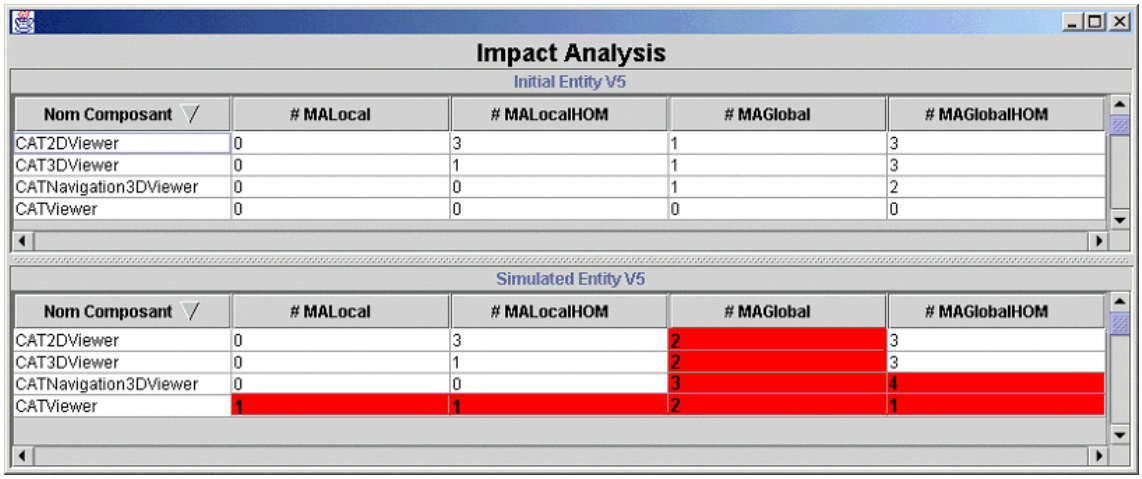


FIG. 7.17 – Utilisation de couleurs pour afficher les résultats d'une simulation d'analyse d'impact

Etendre les fonctionalités de l'outil vers des aspects de restructuration

Il est possible de se baser sur le moteur d'analyse d'impact pour proposer de nouvelles fonctionnalités liées à des aspects de restructuration.

Certains besoins de restructuration ne sont pas simples à définir précisément et le grand nombre de possibilités empêche l'acteur de toutes les tester. Par exemple, il arrive que des architectes ou chefs de produit souhaitent restructurer leurs frameworks. Cette opération est généralement délicate et elle doit être préparée et planifiée longtemps à l'avance. Il est difficile de trouver la meilleure solution (i.e. avec le minimum d'impact) pour découper ou regrouper certains frameworks. Un outil de restructuration serait d'une grande utilité et pourrait tester les diverses possibilités afin de fournir la ou les meilleures solutions. Afin de réduire le nombre de possibilités, il est envisageable de le coupler avec une base de connaissances pour ne tester que les cas référencés comme pertinents.

Ces fontionnalités permettraient de passer d'un outil de simulation à un outil de restructuration où l'acteur ne propose plus une solution à tester mais simplement son besoin. Dans l'état actuel, ceci ne reste encore qu'une proposition puisque nous n'avons réalisé aucune expérience pour vérifier la validité et la faisabilité de cette approche dans le contexte de CATIA V5.

7.4 En résumé

Les deux premières parties de cette thèse ont été axées sur les besoins qui ont motivé la création de l'architecture V5, sur la présentation détaillée de cette architecture et sur notre approche pour pouvoir décrire l'architecture de CATIA V5. Dans ce chapitre, nous avons plutôt axé notre discours sur les besoins des différents acteurs pour l'utilisation de l'architecture V5. Nous avons montré comment nous

avons pu répondre concrétement à certains de ces besoins par l'intermédiaire de trois prototypes :

- **L'OMVT** qui nous a permis de représenter des entitiés de l'architecture V5 et plus spécifiquement de l'Object Modeler. Cet outil a été largement apprécié par l'ensemble des acteurs de Dassault Systèmes car c'était la première fois que les ingénieurs pouvaient voir concrètement une partie de l'architecture V5 par l'intermédiaire des composants Object Modeler. L'utilisation d'une interface graphique intuitive a été un facteur important pour les ingénieurs de Dassault Systèmes. La disponibilité de fonctionnalités de navigation entre les différentes entités a l'avantage de pouvoir faire des recherches rapides en gardant toujours le contexte de la recherche.

- **L'utilisation de GSEE** qui, par sa philosophie pour l'exploration des logiciels sans avoir à définir le méta-modèle, nous a permis de réaliser de nombreuses expériences sans écrire la moindre ligne de code. Nous avons ainsi pu explorer les architectures physique et produit ainsi que l'interaction entre les différentes structures architecturales. La visualisation en graphe par l'intermédiaire de Grappa, nous a permis de réaliser des vues globales qui ne sont pas diponibles avec l'OMVT.

- **L'outil de simulation d'analyse d'impact** qui a pour objectif de répondre à des besoins de maintenance, d'évolution et de restructuration de l'architecture de CATIA V5. Actuellement, notre prototype n'est pas suffisamment fonctionnel mais il nous a permis d'identifier les concepts importants pour ce type d'outil et de réaliser un framework évolutif. Un effort particulier a été réalisé pour répertorier et classifier les besoins des différents acteurs pour l'analyse d'impact.

La conception et l'implémentation de ces prototypes représentent des processus de développement conséquents totalisant environ 30 mois/homme. Pour pouvoir réaliser ces prototypes, nous avons utilisé plusieurs pratiques du génie logiciel : analyse des besoins auprès des acteurs de Dassault Systèmes, utilisation du cycle de vie en spirale, modélisation en UML de nos développements, gestion d'un développement concurrent pour l'OMVT (6 personnes), développement de composants réutilisables, conception d'architectures flexibles et évolutives par l'intermédiaire de frameworks objets, rigueur de programmation par l'utilisation de patrons de conception, documentation importante du code source par l'intermédiaire de l'outil javadoc de SUN, gestion du déploiement de nos outils sur différentes machines...

Ces expériences ont été bénéfiques pour bien comprendre les difficultés et les avantages de concevoir, de mettre en place et d'utiliser une architecture logicielle. Cela nous a permis de ne pas seulement avoir une connaissance théorique de l'architecture logicielle mais aussi pratique. Ceci nous a aidé tout au long de nos travaux aussi bien au niveau académique pour bien comprendre les différentes contributions qu'au niveau de notre collaboration avec Dassault Systèmes pour la compréhension des besoins des acteurs de Dassault Systèmes.

Plusieurs autres expérimentations sur l'architecture de CATIA V5 ont été effectuées au sein de notre équipe de recherche par des étudiants de DEA et de DESS [Cer00, Jam00, Ngu00].

Chapitre 8

Un Atelier de Génie Logiciel pour l'Architecture Logicielle

Dans le chapitre précédent, nous avons vu que l'utilisation de l'architecture V5 engendrait divers besoins architecturaux. Nous pensons que ces types de besoins ne sont pas spécifiques au contexte de Dassault Systèmes et qu'ils sont généraux au domaine de l'architecture logicielle. Le simple fait de créer et/ou d'utiliser une architecture demande diverses fonctionnalités liées à la gestion de cette architecture (cf. section 8.1). Par exemple, avec le langage java, il existe une notion de paquetage qui permet de regrouper et structurer les différentes classes. Cette structure architecturale s'accompagne de nouveaux besoins pour gérer la cohérence et le couplage entre les paquetages ainsi que des problèmes de restructuration comme le renommage d'un paquetage.

Nous avons montré que pour gérer ces besoins architecturaux, les acteurs ont besoin d'outils adaptés. Nous avons présenté trois types d'outils pour la visualisation, l'exploration et la simulation d'analyse d'impact au niveau architectural. Dans ce chapitre, nous allons voir qu'il est possible d'imaginer un environnement complet pour l'architecture logicielle intégrant les outils précédents ainsi que bien d'autres (cf. section 8.2).

Enfin, nous montrerons que ce type d'environnement existe déjà pour la gestion de site web. Pour cela, nous illustrerons nos propos avec pour exemple l'environnement Dreamweaver (cf. section 8.3).

8.1 Elever les outils existants au niveau architectural

> *"Elevating system development from module to the architecture level requires a corresponding elevation in our tools [...] While we have a long history and mature technology for the former, we have just begun to recreate these capabilites at the software architecture level."* [Bal99]

Comme Robert Balzer, nous pensons qu'il faut élever les différents outils existants au niveau architectural. Durant les deux dernières décénnies, de nombreux ateliers de génie logiciel au niveau du code source ont été développés et utilisés pour différents objectifs : outils de développement rapide (JBuilder [JBu], Visual C++ [Vis]...), outils d'analyse (découpe[1] [HR92], insure++ [Ins], Rational Purify [Pur]...), outils de métriques (McCabe [McC], métriques logicielles [FP98]), outils d'ingénierie inverse (FAMOOS [DD99], PBS [PBS], Rigi [Rig]...), outils d'analyse de performance et de tests (Logiscope tools [Log])...

Bien que plusieurs études aient été menées pour remonter ces outils au niveau architectural (découpe à partir d'une description architecturale [Zha98], outils d'analyse de dépendance [SRW98], proposition de métriques pour l'architecture logicielle [Ben98, Zha01]...), nous n'en sommes qu'au début ce qui laisse encore place à de nombreux travaux[2].

8.2 Vers un environnement architectural

Nous pensons qu'il est nécessaire de disposer d'un environnement architectural associé au processus de développement. Cet environnement architectural doit pouvoir disposer de l'ensemble des fonctionnalités fournies par un atelier de génie logiciel traditionnel. Nous proposons de se calquer sur l'architecture utilisée dans les environnements traditionnels pour les langages de programmation (comme par exemple dans FAMOOS [DD99]), mais cette fois avec des données architecturales (cf. figure 8.1).

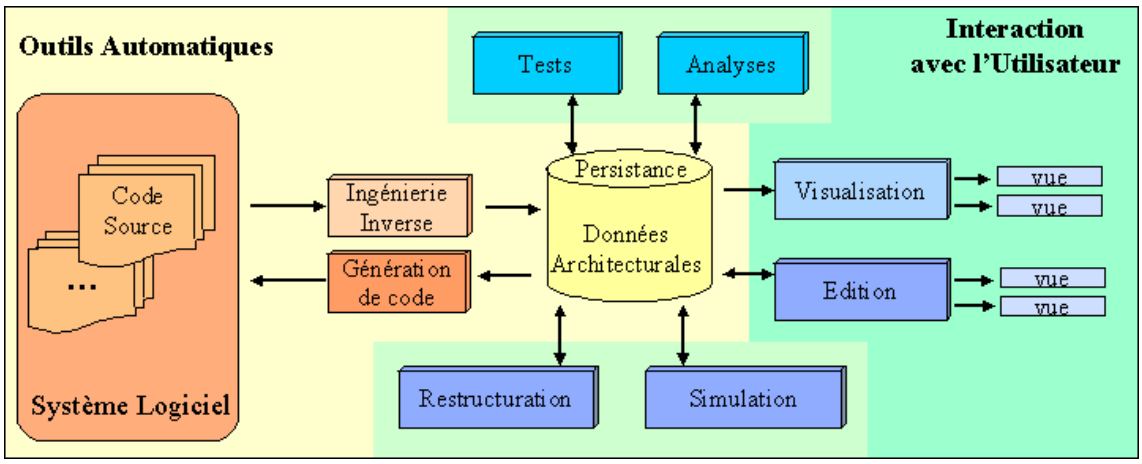


FIG. 8.1 – Environnement pour l'Architecture Logicielle

[1] Slicing en anglais.

[2] "Although software architecture is on a much more solid footing than a decade ago, it is not yet established as a discipline that is taught and practiced universally across the software industry. One reason for this is simply that it takes time for new approaches and perceptions to propagate. Another reason is that the technological basis for architecture design is still immature." David Garlan [Gar00].

Afin de toujours garder le lien avec le code source, il faut disposer d'un outil d'ingénierie inverse qui extrait les informations architecturales directement à partir de celui-ci. Notre collaboration avec Dassault Systèmes a montré qu'il était possible d'extraire automatiquement les structures architecturales et certains liens de dépendance entre les différentes entités architecturales. Pour pouvoir recueillir les autres données architecturales qui sont plus riches sémantiquement, une intervention humaine est alors nécessaire (cf. section 5.3.3).

Au-dessus de la base de données qui gère la persistance des informations architecturales, une grande palette de services et d'outils peuvent être construits :

- **Les outils de visualisation** : ces outils de visualisation, comme l'OMVT, permettent de représenter l'architecture logicielle du système par l'intermédiaire de différentes vues. Ces vues sont définies dans le but de répondre aux besoins des différents acteurs du processus de développement (développeurs, clients, chefs de projet...). Elles peuvent servir de base pour expliquer certains choix, aider à raisonner au niveau architectural ou encore faciliter la compréhension de l'architecture logicielle du système. Ces outils sont importants car ces descriptions architecturales ne sont en général pas simples à réaliser du fait que les informations architecturales peuvent être dispersées dans différents endroits et fichiers. L'ajout de fonctionnalités de navigation entre les différentes vues est un aspect important à ne pas négliger. Cela permet aux acteurs d'exploiter au mieux ces vues.

- **Les outils d'exploration** : ces outils d'exploration, tel que GSEE, rentrent plus ou moins dans la catégorie des outils précédents. Ils sont généralement complémentaires et permettent de générer des vues qui ne sont pas prédéfinies à l'avance. Ils permettent aussi de répondre à des besoins temporaires, de faire rapidement des expérimentations et de découvrir le système par succession de requêtes.

- **Les outils d'édition** : ces outils permettent d'assister le travail des développeurs dans leur tâche quotidienne. L'interface graphique de ces outils offre la possibilité de cacher la complexité du code source par une abstraction de celui-ci et de faciliter le développement grâce à des palettes de composants paramétrables. Par exemple, avec Visual C++, un développeur peut développer et utiliser des composants COM sans connaître les caractéristiques et la syntaxe textuelle de ce modèle de composants. En ce qui concerne l'Object Modeler, les différents acteurs n'auraient plus à connaître les macros pour les différentes entités et constructions de ce dernier. Cette abstraction a en plus l'avantage de pouvoir faire évoluer les différents mécanismes du modèle de composant sans trop d'impact et sans que les développeurs ne s'en aperçoivent. Les outils d'édition peuvent aussi contraindre la programmation en imposant l'utilisation de certains patrons de conception ou de certaines règles syntaxiques.

- **Les outils d'analyse** : alors que les outils de visualisation n'affichent seulement que les informations architecturales extraites du code source de différentes façons, la disponibilité de ces données architecturales offre la possi-

bilité de réaliser diverses analyses. Par exemple, l'OMVT analyse les informations architecturales de CATIA V5 pour vérifier des erreurs de construction du TroubleShooter. Mais beaucoup d'autres analyses peuvent être implémentées : vérification d'un certain nombre de contraintes, analyses de cohérence et de cohésion, calcul de différentes métriques et statistiques, analyses de performances...

- **Les outils de tests** : tout comme pour un langage de programmation, il serait possible de spécifier un ensemble de tests pour vérifier que l'architecture logicielle au niveau du code est bien conforme aux descriptions architecturales réalisées par les architectes lors de la phase de conception. Par exemple, nous pourrions imaginer de concevoir des tests pour vérifier que l'implémentation des composants correspond bien aux descriptions architecturales initiales. Il serait ainsi possible de s'assurer que toutes les interfaces, et uniquement celles qui avaient été définies, sont bien implémentées. Ces outils sont dans une certaine sorte des outils d'analyse.

- **Les outils de simulation** : les outils de simulation sont intéressants dans la mesure où ils permettent de réaliser virtuellement certaines actions. Cela peut aller de la simulation d'une exécution de programme (comme dans Rapide) jusqu'à la simulation de calculs complexes (comme pour les analyses d'impact). Ces fonctionnalités de simulation permettent un gain de temps important puisqu'elles évitent de développer concrètement les différentes solutions pour trouver la meilleure.

- **Les outils de restructuration** : à la différence des outils de simulation où c'est l'utilisateur qui va spécifier et guider la simulation, les outils de restructuration appliquent un certain nombre de règles prédéfinies automatiquement. Ce n'est plus l'utilisateur mais l'outil qui devient l'expert du domaine. L'outil réalise automatiquement les différentes heuristiques et opérations prédéfinies. Ces types d'outils sont similaires aux travaux de Fowler et al. sur la restructuration par factorisation de code [FBB+99].

- **Les outils de génération de code** : l'ensemble des outils précédents travaillent sur une abstraction architecturale à partir des données stockées dans la base de données sans modifier le code source. Bien que certaines modifications architecturales soient difficiles à générer directement en code source, il est possible dans la majorité des cas de maintenir automatiquement le lien entre les deux niveaux. Certains outils, comme ceux pour l'édition et pour la restructuration, n'ont vraiment d'intérêt que s'ils sont couplés avec de tels outils.

Nous venons de décrire plusieurs types d'outils qui pourraient faire partie d'un atelier de génie logiciel au niveau architectural mais cette liste n'a pas pour but d'être exhaustive (par exemple, nous n'avons pas discuté des fonctionnalités pour la recherche d'informations). Nous voulons plutôt donner un aperçu de ce que pourrait être un environnement pour la gestion d'une architecture logicielle. De plus, l'architecture de cet environnement permet d'intégrer simplement d'autres types d'outils autour de la base de données.

8.3 Illustration avec Dreamweaver pour la gestion de sites web

Il est intéressant de regarder d'un peu plus près les outils pour la gestion (création, maintenance et évolution) de sites web. Bien que l'expansion des sites internet soit récente, des environnements complets ont vu le jour pour gérer un site web. Ces environnements offrent de nombreuses fonctionnalités comparables à celles que nous venons de décrire dans la section précédente.

Il nous est apparu intéressant de faire l'analogie avec ce type d'environnement car la gestion d'un site web engendre des besoins architecturaux similaires au développement d'un logiciel et parce qu'il existe sur le marché des produits commerciaux. De plus, il est plus facile d'expliquer et de comprendre ces besoins car la problématique est un peu plus simple et que de plus en plus de personnes ont été confrontées à ces besoins pour la gestion de leur site web. Pour développer notre exemple, nous nous baserons sur l'environnement Dreamweaver de Macromedia [Dre]. Notre choix s'est porté sur cet environnement car c'est celui que nous connaissons le mieux[3], mais la plupart de ces environnements se ressemblent.

Réaliser une page web n'est pas difficile en soi et il possible de l'écrire avec un simple éditeur de texte comme Notepad. Par contre, gérer un site complet qui contient plusieurs dizaines de pages web est autrement plus complexe. Le gérer à la main devient une tâche périlleuse et le besoin de disposer d'outils appropriés s'en fait ressentir. Par exemple, si le responsable du site souhaite renommer le nom d'un fichier qui est référencé (au sens d'hyperlien web) de nombreuses fois, il va falloir éditer et modifier ces liens dans l'ensemble des fichiers concernés. Le faire à la main est une tâche longue et pénible et il est toujours possible d'oublier de mettre à jour certains liens. Utiliser un environnement comme Dreamweaver permet de faire cela automatiquement en quelques secondes tout en nous assurant la complétude.

Nous avons pu remarquer que comme dans le cas d'une architecture logicielle d'un système, il est possible d'identifier plusieurs structures architecturales pour la gestion d'un site web. De par nos connaissances de l'environnement Dreamweaver, nous en avons recensé trois qui correspondent plus ou moins à l'architecture logique, l'architecture physique et l'architecture de publication.

Nous allons maintenant décrire plus en détails ces trois structures architecturales avec pour exemple mon site web disponible à l'adresse `http://www-adele.imag.fr/~sanlavil/`

[3]Notre discours sera basé sur la version 2 puisque c'est cette version que nous connaissons et utilisons. Même si actuellement cet environnement est à la version 4, cela ne gène en rien notre argumentation.

L'architecture logique pour la gestion d'un site web

Cette structure architecturale reflète les entités et liens de dépendance entre les différentes pages web. Elle est liée à la phase de conception des pages web par un langage dédié à internet (le plus courant étant HTML qui devrait être au fur et à mesure remplacé par XML).

Dreamweaver fournit une interface graphique wysiwyg qui permet à quiconque de réaliser sa page sans avoir besoin de connaître la syntaxe textuelle de HTML. Pour les personnes plus expérimentées, Dreamweaver propose un éditeur de texte par défaut (il est aussi possible de choisir son éditeur de texte préféré) pour modifier la page directement à partir du code HTML. La synchronisation entre l'éditeur wysiwyg et l'éditeur de texte se fait en temps réel. Dreamweaver assure donc à tout moment le lien entre le code HTML et sa représentation graphique (cf. figure 8.2).

FIG. 8.2 – Deux vues pour l'architecture logique

Dreamweaver dispose d'une vue de l'architecture logique sous forme d'un arbre représentant le graphe de dépendance des hyperliens des différentes pages du site (cf. partie gauche de la figure 8.3). Cette vue permet de représenter une vue globale de l'architecture logique alors que les vues précédentes sont orientées vers une vue locale d'une page HTML. Nous pouvons ici voir le parallèle avec l'OMVT qui est plutôt dédié aux vues locales des composants Object Modeler et GSEE qui permet de visualiser des vues globales sous forme de graphe.

Diverses analyses sont disponibles pour l'architecture logique. Il est par exemple possible de vérifier les liens brisés (i.e. que lien hypertexte n'est pas ou plus correct)

ou les fichiers orphelins (i.e. que ces fichiers ne sont référencés par aucune autre page web du site et que ce n'est pas la page d'accueil du site). Lorsque l'utilisateur demande cette analyse, une fenêtre de diagnostic liste l'ensemble des problèmes ainsi que les pages HTML incriminées. Grâce aux fonctionnalités de navigation, il est possible d'afficher la page web ainsi que le lieu exact de l'erreur en question. L'utilisateur n'a plus qu'à faire les corrections en conséquence.

L'architecture physique pour la gestion d'un site web

L'architecture physique est liée à la structuration et gestion des fichiers nécessaires pour le site web. La vue architecturale correspondante est un système de fichiers rassemblant l'ensemble des répertoires et fichiers contenu dans le répertoire racine du site web (cf. partie droite de la figure 8.3). A partir de cette vue architecturale, il est possible de réaliser les actions courantes pour la gestion des fichiers et répertoires : créer, renommer, supprimer, ouvrir, acquérir...

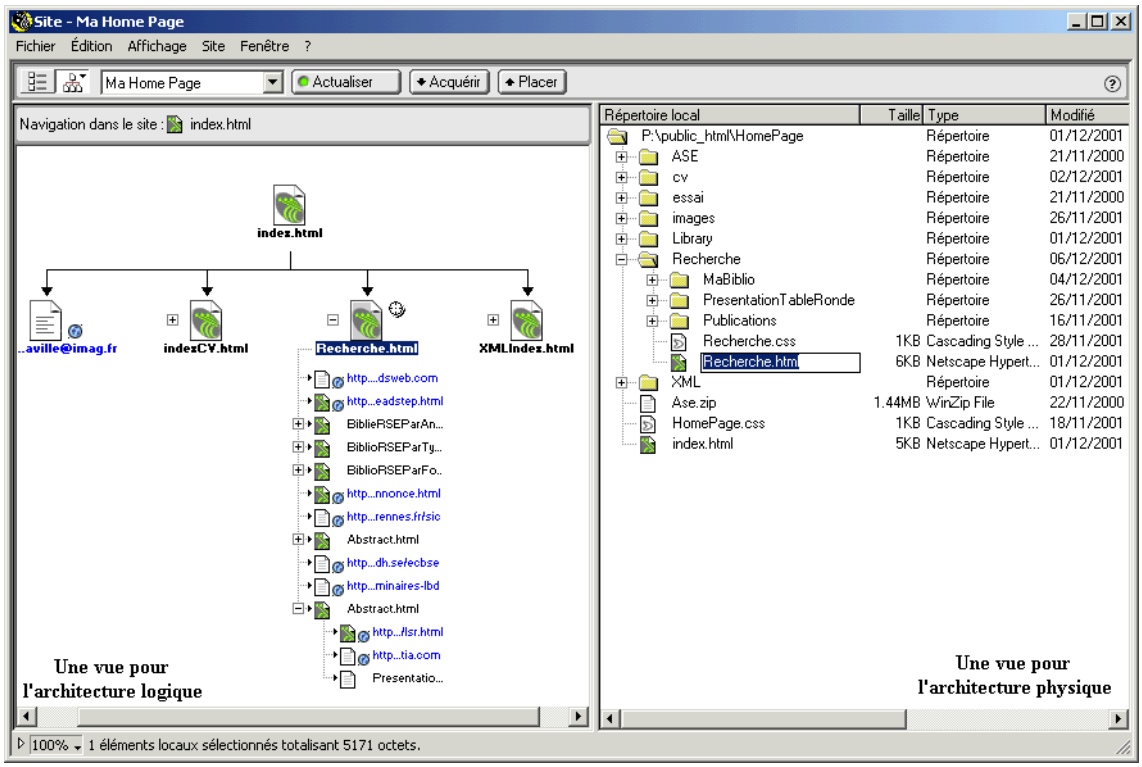


FIG. 8.3 – Deux vues pour l'architecture logique et l'architecture physique

Les actions d'édition précédentes peuvent avoir un impact sur l'architecture logique. Par exemple, si l'utilisateur souhaite renommer un fichier, l'ensemble des pages web du site qui référencent ce fichier doivent être mis à jour sous peine de voir leur lien brisé. Dreamweaver facilite la tâche de l'utilisateur en automatisant le processus. Il calcule l'ensemble des liens de dépendance et demande à l'utilisateur s'il souhaite mettre à jour les fichiers impactés.

L'architecture de publication pour la gestion d'un site web

Cette structure architecturale répond au besoin de disposer de deux types de sites dans la gestion d'un site web :

- **Le site local** : ce site correspond au site de la personne qui est en train de créer ou de maintenir son site (cf. partie droite de la figure 8.4). Sur ce site les données ne sont visibles que par le propriétaire du site local et peuvent ne pas encore être disponibles à travers internet. C'est sur ce site que les fonctionnalités pour l'architecture physique sont offertes.

- **Le site distant** : ce site est celui qui est utilisé par le serveur web (par exemple Apache) pour que le site soit accessible par les internautes à travers leur navigateur web (cf. partie gauche de la figure 8.4).

La distinction entre le concept de site local et le concept de site distant permet de répondre concrètement à différents besoins. Par exemple, cela facilite la gestion concurrente d'un site web par plusieurs acteurs. Chaque acteur a son propre site local qu'il peut faire évoluer indépendamment des autres, le site distant jouant le rôle de base commune (c'est dans une certaine mesure la release commerciale d'un logiciel). Ainsi chaque acteur a la possibilité de réaliser différents tests et de ne publier que les données qui l'intéressent. De plus, il est possible que la structure du site local soit différente de la structure du site distant. Par exemple, un acteur peut réaliser un certain nombre de tests et donc posséder des fichiers et répertoires qu'il ne souhaite pas publier.

Cette séparation entre site local et site distant exprime le besoin de pouvoir gérer deux structures différentes. Une structure pour la conception (site local) et une structure pour la publication (site distant).

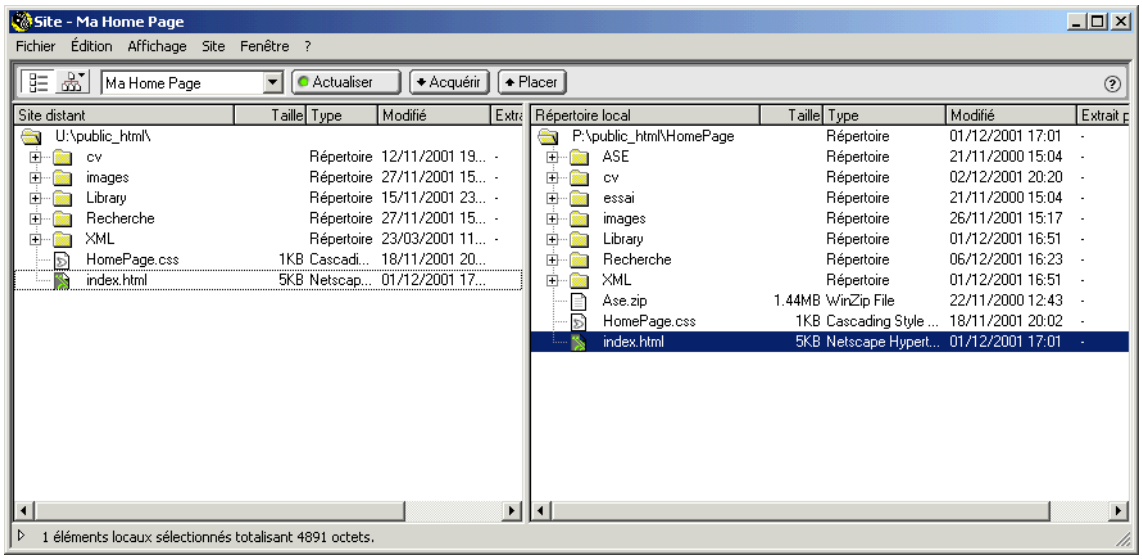


FIG. 8.4 – Une vue pour l'architecture de publication

Mais le fait d'utiliser ces deux types de site a un coût : il faut assurer la cohérence entre ces deux sites. De ce côté, Dreamweaver aide les concepteurs en fournissant des outils de mise à jour de site. Il est possible par un simple clic sur un bouton (après avoir configuré l'accès au serveur via un réseau local ou via FTP) :

- **De placer, ou encore publier des fichiers** : c'est-à-dire de copier des fichiers depuis le site local vers le site distant.
- **D'acquérir des fichiers** : c'est-à-dire de copier des fichiers depuis le site distant vers le site local. C'est l'opération duale de la première. Cette fonction est surtout utile lorsqu'il y a un développement concurrent du site. Dans ce cas, il faut pouvoir mettre à jour son site local par rapport aux dernières modifications réalisées par les autres concepteurs.
- **D'actualiser le site local ou le site distant** : ceci permet de s'assurer que le concepteur utilise bien la dernière version du site. Cette fonctionnalité est utile car à force de faire des modifications, il est possible que le concepteur ne sache plus sur quel site se trouvent les fichiers les plus récents.

8.3.1 Conclusion

Nous avons pu voir que certains besoins des concepteurs de site web se placent à un niveau architectural. La gestion d'un site web, même très simple, demande de manipuler des entités et des liens de dépendance de différentes natures. Nous avons montré que Dreamweaver offre la possibilité de gérer trois structures architecturales et fournit un certain nombre de fonctionnalités associées à chacune d'entre elles.

Cette analogie avec les environnements pour la gestion d'un site web est intéressante par le fait que nous pouvons constater que même avec un langage assez simple comme HTML et avec seulement quelques dizaines de fichiers, la gestion d'un site à la main devient difficile et que des outils appropriés sont nécessaires.

Il est facile d'imaginer l'importance de disposer d'un environnement architectural pour gérer le développement d'un logiciel comme CATIA V5 qui contient près de 5 millions de lignes de code C++ et qui possède des milliers de fichiers.

8.4 En résumé

Dans ce chapitre, nous avons tenté de montrer, comme le souligne Robert Balzer [Bal99], la nécessité de posséder un environnement complet dédié à l'architecture logicielle.

Nous avons détaillé ce à quoi pourrait ressembler ce type d'environnement en proposant un certain nombre d'exemples d'outils architecturaux. Ces outils couvrent l'ensemble des phases du cycle de vie du logiciel : conception, maintenance, évolution, tests et déploiement.

Une analogie avec un environnement pour la gestion de sites web a été étudiée pour illustrer notre argumentation. Nous avons pu souligner le fait que même avec un langage aussi simple que HTML et quelques dizaines de fichiers, le besoin de gérer des structures architecturales à travers d'outils appropriés devient indispensable.

Il semble que ce type d'environnement commence à apparaître pour les logiciels, comme avec le produit Small Worlds[4] [Sma], mais il reste encore du chemin à parcourir avant qu'ils ne soient utilisés quotidiennement dans le milieu industriel.

[4]Nous n'avons eu connaissance de ce produit que très recemment (décembre 2001). Il ne nous a donc pas été possible de l'étudier suffisamment pour pouvoir le présenter dans le cadre de cette thèse.

Chapitre 9

Réflexions autour de l'Architecture Logicielle

Notre collaboration avec Dassault Systèmes a été une expérience enrichissante où nous avons été confrontés aux réalités d'un développement important à travers le logiciel CATIA V5. Notre contexte de travail lié à une pratique fortement industrielle nous a souvent demandé de remettre en question nos connaissances académiques et d'aller à l'encontre de plusieurs idées reçues. L'aspect le plus marquant a été de comprendre et de faire face à certaines décisions qui n'ont pas été motivées uniquement par des aspects techniques mais aussi économiques et humains.

Les besoins architecturaux de Dassault Systèmes pour le logiciel CATIA V5 étant importants, de nombreuses directions de recherche s'ouvraient à nous. Comme il n'était pas possible de toutes les explorer, nous avons dû nous concentrer uniquement sur quelques aspects. Néanmoins, même si nous n'avons pas pu pousser nos recherches sur l'ensemble des thèmes architecturaux que nous avons rencontrés, nous avons pu aborder et identifier plusieurs points importants pour la pratique du domaine de l'architecture logicielle.

Nous souhaitons dans ce dernier chapitre, formuler quelques leçons et pistes de réflexion tirées de notre expérience avec Dassault Systèmes qui n'ont pas pu être abordées dans ce rapport de thèse [EFS02]. Nous souhaitons terminer ce rapport de thèse en abordant diverses discussions autour de l'architecture logicielle pour ouvrir ce travail de thèse sur un débat plus large et tenter de mieux comprendre le domaine et la pratique de l'architecture logicielle. Nous souhaitons faire partager notre expérience et espérons que d'autres retours d'expérience pourront conforter ou non nos impressions.

9.1 Etude de l'architecture logicielle par une approche montante ou descendante ?

L'architecture logicielle d'un système étant une abstraction de ce dernier, il est possible de suivre deux approches pour travailler sur ce domaine :

- **Approche descendante ou "top-down"** : soit nous travaillons à partir de spécifications et descriptions architecturales pour aller ensuite vers le code. C'est cette approche qui est utilisée dans la majorité des travaux disponibles dans la littérature. Souvent même, comme dans le cas des ADLs, les recherches ne se placent qu'au niveau de la spécification et il n'existe pas de lien avec le code.

- **Approche montante ou "bottom-up"** : soit nous utilisons une approche inverse qui consiste à partir du code pour remonter aux spécifications architecturales. C'est cette approche que nous avons adoptée. Nous avons fait ce choix car CATIA V5 représente près de 5 millions de ligne de code et qu'il n'était pas envisageable pour Dassault Systèmes de ne pas prendre en compte cet existant.

Le choix que nous avons fait dépend beaucoup de notre contexte. Bien évidemment, chacune de ces deux approches a ses avantages et ses inconvénients et elles apportent des réponses différentes. Mais plus que cela, nous pensons qu'elles sont complémentaires et qu'il faut les utiliser conjointement (cf. figure 9.1).

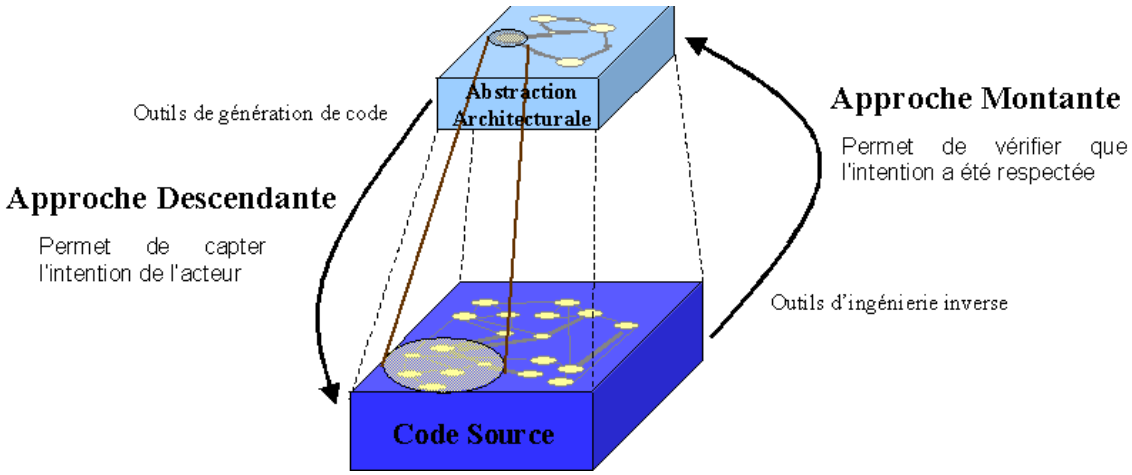


FIG. 9.1 – Complémentarité des approches montante et descendante

L'approche descendante est importante car elle permet de capter l'intention de l'acteur. Les spécifications architecturales expriment la volonté du concepteur pour l'architecture de son système. Elles jouent le rôle de référence pour l'implémentation et la maintenance du système.

Par contre ces spécifications ne sont pas suffisantes par le fait que l'architecture qui se trouve dans le code est généralement différente de l'architecture décrite par les spécifications. L'utilisation de langages de programmation, de mécanismes particuliers... amène souvent à la création de liens de dépendance non souhaités qui compromettent l'intention initiale. Cette architecture du code est très importante puisque c'est elle qui est utilisée par les clients et par les développeurs. Il est donc nécessaire d'utiliser aussi une approche montante pour refléter l'architecture du code, la réalité de ce qui s'exécute. Cela permet de vérifier que l'architecture de

code correspond au mieux à l'intention des architectes et de corriger les éventuelles dérives.

Même si dans cette thèse nous n'avons pris en compte que l'approche montante, nous pensons qu'il faut combiner les deux approches. Cela permet de vérifier la cohérence et de toujours garder le lien entre les intentions des architectes et la réalité du code. Pour l'architecture V5, nous avons pu remarquer que certaines actions avaient été prises par les architectes de Dassault Systèmes pour prendre en compte l'intention des concepteurs. Par exemple, les cartes d'identité des frameworks qui définissent les liens de dépendance de compilation entre les frameworks avaient été créées pour décrire l'intention des chefs de produits pour la gestion des frameworks de leur produit.

L'idéal serait sans doute de n'utiliser qu'une approche descendante, mais pour cela il faudrait posséder des outils de génération de code suffisamment puissants pour ne pas avoir à retoucher le code (comme aujourd'hui nous ne modifions plus le code binaire à la main et n'utilisons plus que des langages de programmation de haut niveau). Mais actuellement, il n'est pas possible de demander à des développeurs de ne pas modifier le code source directement. D'une part, parce que les langages de spécification n'ont généralement pas un pouvoir d'expression suffisant (certains aspects architecturaux devront être exprimés directement dans le code) et d'autre part parce qu'il n'existe pas de tel générateur de code. Il faudrait aussi gérer le fait que le code généré ait à cohabiter avec du code écrit par les développeurs (structures de données, algorithmes...) et s'assurer des problèmes d'optimisation.

9.2 Un langage propre pour l'architecture logicielle ?

Au cours de notre étude sur l'architecture V5, nous nous sommes aperçus que les ingénieurs de Dassault Systèmes avaient créé un langage au dessus de C++ pour l'Object Modeler. Ce langage est actuellement défini à travers l'utilisation de différentes macros intégrées dans le code source C++. A un moment donné, nous avions le souhait de définir un langage spécifique pour pouvoir décrire la structure statique de l'Object Modeler. Ce langage aurait eu pour but de remplacer (et même supprimer) les macros situés dans le code C++. Les descriptions architecturales auraient été localisées dans des fichiers particuliers.

Dans le contexte de l'architecture V5, la création d'un langage architectural propre ainsi que la séparation du langage de programmation aurait eu plusieurs avantages :

- Le langage aurait été clairement défini ce qui aurait permis de l'exploiter plus facilement par des outils.
- L'architecture V5 aurait gagné en flexibilité par le fait de ne plus avoir à

recompiler le code C++ pour une modification au niveau de la structure architecturale.

- Nous aurions pu créer un compilateur spécifique pour l'Object Modeler. Le TroubleShooter aurait été la base de la sémantique statique de ce compilateur puisqu'il permet de vérifier certaines règles de construction statiquement.

- La séparation du langage de programmation de la structure architecturale facilite la construction d'un logiciel par assemblage. Par exemple, dans le cas de l'Object Modeler, il est possible d'imaginer que les développeurs implémentent leurs extensions et qu'un autre acteur ait la charge de spécifier la structure des composants Object Modeler en spécifiant l'ensemble des extensions pour chaque composant.

- Les développeurs n'auraient à connaître et ne disposeraient que des constructions architecturales dont ils ont besoin. Cela éviterait qu'ils n'aient entre les mains des constructions architecturales dont ils n'en n'ont pas l'utilité et qui influencent l'architecture. Cela permettrait de limiter le pouvoir d'action des développeurs uniquement aux besoins de leur métier.

Malheureusement nous n'avons pas eu l'occasion de tenter l'expérience car l'opération était considérée comme trop risquée. Le code de CATIA V5 repose sur l'Object Modeler et il était difficile de pouvoir prédire à l'avance l'impact d'un simple test et de s'assurer que cela n'entrave pas le bon fonctionnement de la société. Arrêter le développement de près de mille ingénieurs, même pendant quelques heures, suite à une erreur de notre part représente un coût financier important (quelques millions de francs).

Néanmoins nous pensons qu'il est bon de pouvoir utiliser un langage précis pour décrire les mécanismes architecturaux. Ce langage peut très bien réutiliser la syntaxe du langage de programmation si celui-ci permet de spécifier l'ensemble des instructions architecturales souhaitées.

Actuellement, nous pouvons contater que, dans les récentes propositions de langage architectural, deux approches sont en train d'émerger :

- Soit les descriptions architecturales sont distinctes du code source comme nous souhaitions le faire pour l'Object Modeler. C'est l'approche qui a été prise par l'OMG avec CCM [Rui].

- Soit les instructions architecturales sont intégrées directement dans la syntaxe du langage de programmation. C'est l'approche que Microsoft a commencé à prendre avec son langage C# [Cor01]. Ce langage a par exemple intégré des instructions pour la gestion des événements.

Ces deux approches ont chacune leurs avantages et inconvénients. Comme nous l'avons décrit ci-dessus, la première approche permet de distinguer le langage de programmation de la structure architecturale, d'être plus flexible. Par contre, elle a

l'inconvénient de devoir gérer plusieurs fichiers et outils ce qui n'est pas le cas pour la deuxième approche.

9.3 Ne pas négliger les descriptions et outils “simples”

Les descriptions architecturales spécifiées à l'aide de langages formels (comme dans les ADLs existants) permettent de réaliser des calculs complexes (comme des analyses d'interblocage). Mais comme nous avons pu le voir dans la section 5.3.2, ce niveau d'abstraction n'est pas toujours adapté et correspond plutôt à la conception de logiciels critiques. De plus cela peut demander un investissement important (engager ou former un personnel compétent dans les langages formels, outils...) qui n'est pas toujours justifié.

En ce qui concerne CATIA V5, certains besoins de Dassault Systèmes se situent au niveau de la compréhension et de l'analyse d'impact. La complexité engendrée par la taille et le fort développement concurrent fait que même des choses simples peuvent rapidement devenir compliquées. Le fait de fournir une vision graphique et centralisée des composants Object Modeler, ce qui n'a rien d'extraordinaire et compliqué, est déjà une aide importante pour les ingénieurs de Dassault Systèmes. Notre prototype de l'OMVT n'a rien de révolutionnaire mais il est intéressant pour les ingénieurs de Dassault Systèmes parce qu'il répond à des besoins quotidiens et simplifie et automatise des tâches répétitives. L'utilisation d'une syntaxe graphique simple et intuitive a joué un rôle important et a été très appréciée par les ingénieurs de Dassault Systèmes.

Nous pensons que dans la majorité des cas, des outils “simples” permettent de répondre à de réels besoins et peuvent être plus importants que des outils beaucoup plus poussés. Les outils comme pour l'analyse d'interblocage sont intéressants s'ils répondent à un vrai besoin ou s'ils sont “gratuits” (i.e. qu'ils sont automatiques et rapides). L'important est de trouver le bon niveau pour la description architecturale ainsi que les outils associés. Pour cela, seule une analyse des besoins permet de trouver ce niveau.

9.4 De l'intérêt des descriptions architecturales locales

Lorsqu'on nous parle d'architecture logicielle, généralement la première idée qui nous vient à l'esprit est une cartographie complète du système logiciel.

C'est aussi ce que nous pensions au début et voulions réaliser avec CATIA V5. Au fur et à mesure de nos travaux, notre vision a un peu changé et évolué :

- Nous nous sommes d'abord aperçus que des vues locales comme pour les vues

pour les entités Object Modeler avaient un sens et correspondaient mieux aux besoins de certains acteurs. Ces acteurs n'ont qu'une vue restreinte de l'architecture et ces vues locales sont plus adaptées. Une vue globale de l'architecture n'apporterait pas une grande aide pour leur travail.

- Nous avons réalisé quelques tests pour tenter de représenter une cartographie de CATIA V5 sous forme d'un graphe de dépendance avec l'aide de l'outil Dotty. Mais un graphe contenant des milliers de nœuds et des milliers de liens de dépendance est difficilement exploitable. Il ressemble plus à un plat de spaghettis qu'à autre chose.

- Nous aurions pu tenter d'améliorer la visualisation de ce graphe avec des techniques de représentation de gros volumes de données [MH96, Ver01]. Mais même si nous avions réussi à réaliser une description complète et exploitable de l'architecture de CATIA V5, nous ne serions pas sûrs que cela serait la vue architecturale la plus utile pour les ingénieurs de Dassault Systèmes. En effet l'architecture V5 a été construite pour faciliter le développement concurrent. La taille du logiciel et la répartition du travail sont telles que personne n'a une idée complète de l'architecture de CATIA V5. Contrairement à ce que nous pouvons imaginer, cela ne pose pas de problème pour maintenir et faire évoluer ce logiciel.

Pour autant les descriptions globales ne sont pas inintéressantes comme nous avons pu le voir avec GSEE (cf. section 7.2.3). De plus, pour certains projets, la cartographie de l'architecture logicielle de leur système peut avoir un sens et être utile. Mais cela n'est pas toujours le cas contrairement à ce que nous pouvions penser initialement.

9.5 Il faut du temps pour réaliser une bonne architecture logicielle

Par bonne architecture logicielle, nous entendons celle qui permet de répondre aux besoins des différents acteurs. Elle doit fournir des concepts et mécanismes performants pour construire des applications efficaces et fiables. Ceci est effectivement important mais ne sert pas à grand chose si ces concepts et mécanismes ne sont pas compris et bien utilisés. Ainsi, elle doit aussi pouvoir être facilement maîtrisée et comprise par les différents acteurs dans leur travail de tous les jours. Ce dernier point est essentiel et sans doute le plus important.

Une bonne architecture doit donc à la fois être performante et facile à utiliser. Comme nous allons le voir ci-dessous, ceci n'est pas toujours compatible et il faut parfois trouver le bon équilibre.

9.5.1 Définition vs Utilisation d'un concept

C'est une erreur de croire que puisqu'il est facile d'expliquer et de réaliser un concept, son utilisation va l'être aussi. Il y a de nombreux contre-exemples comme l'instruction *goto* ou encore les pointeurs dans les langages de programmation. Au niveau des modèles de composants, nous pouvons citer le mécanisme de COM [Box98] pour la gestion mémoire (allocation et libération mémoire des composants) par l'intermédiaire des méthodes *AddRef* et *Release*. Le mécanisme est pourtant simple, il suffit d'incrémenter un compteur (par l'appel de la méthode *AddRef*) lorsque nous souhaitons utiliser le composant et de décrémenter ce même compteur (par l'appel de la méthode *Release*) lorsque nous ne souhaitons plus l'utiliser. Une fois que le compteur atteint la valeur nulle, le composant est libéré de la mémoire. Il est difficile de faire plus simple et pourtant plusieurs gros projets ont montré que ce mécanisme était difficile à maîtriser entraînant des erreurs non triviales à détecter et à corriger (mauvaises allocations ou suppression de composants alors qu'ils étaient toujours utilisés).

Ceci n'est pas dû uniquement à un manque d'expérience. Même après des années d'utilisation, certains concepts restent difficiles à maîtriser. Nous pouvons d'ailleurs constater que ces concepts finissent par être supprimés et remplacés par d'autres au détriment éventuellement de l'efficacité et de la flexibilité. Par exemple, dans plusieurs langages et modèle de composants récents (Java, C#, .Net...), la mémoire est gérée automatiquement par un ramasse miettes ("garbage collector") et la syntaxe du langage n'offre plus d'instructions pour la manipulation des pointeurs.

Néanmoins il est difficile de remplacer un concept si le projet est avancé. Plus un concept est utilisé, plus le coût de la migration sera élevé jusqu'à même compromettre le bon déroulement du projet. Parfois, il est moins risqué de laisser ces concepts et de gérer les problèmes. C'est pour cela que les modifications fondamentales se font lors de la création d'une nouvelle version.

La réalisation d'une nouvelle architecture logicielle est le résultat de l'expérience des architectes et des besoins des clients. Le développement de l'Object Modeler a suivi un processus incrémental. Le concept d'extension a été le point de départ afin de répondre à des problèmes identifiés dans le développement des versions précédentes de CATIA. Par contre les concepts comme la délégation, l'héritage Object Modeler... ont été rajoutés ensuite pour répondre à des problèmes pratiques survenus au fur et à mesure. Parfois, différentes implémentations d'un même concept ont été développées et utilisées simultanément pour gagner en performance et en stabilité.

Concevoir et développer une architecture n'est pas seulement difficile mais peut aussi être risqué. Cela demande de fortes compétences et expériences. Il ne suffit pas de penser à la création des concepts mais aussi à leur facilité d'utilisation.

9.5.2 Architecture logicielle et interactions entre les concepts

Lorsque nous voulons concevoir et faire évoluer une architecture logicielle, il est difficile de la créer et de la maintenir sans engendrer des incompatibilités d'interaction entre les concepts. Nous avons pu répertorier au moins trois raisons pour cela :

- **La difficulté de gérer l'évolution** : le fait qu'un modèle soit défini incrémentalement pose inévitablement un certain nombre de problèmes. Les concepts initiaux n'ont pas forcément été conçus pour interagir avec de nouveaux concepts et la cohabitation peut engendrer des incompatibilités plus ou moins importantes.

- **Une utilisation non prévue ou détournée** : lorsque le concepteur invente un nouveau mécanisme, il le fait généralement dans un cadre et une utilisation bien précise. Il n'est de toute façon pas possible de prévoir tous les cas. Dès que le mécanisme pourra être utilisé par de nombreux développeurs, il est certain qu'au moins un d'entre eux l'utilisera d'une manière détournée ou différente à ce que le concepteur aurait pu imaginer, pour résoudre plus simplement un problème auquel il est confronté. Il est possible de prendre pour exemple l'analogie avec l'arithmétique sur les pointeurs. Cela peut aussi s'expliquer par le fait que les concepts peuvent n'être perçus qu'à leur plus bas niveau. Il n'existe donc pas de distinction entre le concept et son implémentation ce qui engendre généralement des utilisations non prévues. Ces utilisations détournées peuvent éventuellement bien fonctionner mais il est plus probable que ce ne soit pas contrôlé et cela peut engendrer de nouveaux problèmes.

- **Une complexité difficilement maîtrisable** : l'interaction entre les différents concepts crée un nombre de combinaisons qui est exponentiel. Même avec peu de concepts, il en résulte une complexité difficilement maîtrisable qui a pour conséquence de ne pas pouvoir définir précisément la sémantique.

Les ingénieurs de Dassault Systèmes ont effectivement été confrontés à ce genre de problèmes. Par exemple, l'ajout du concept de l'héritage Object Modeler a engendré des conflits d'interactions avec d'autres concepts comme l'extension. Certaines configurations particulières, qui ont été repertoriées et documentées sous le nom d'"anti-patterns", ne fonctionnent pas. Les ingénieurs de Dassault Systèmes ont développé pour cela l'outil TroubleShooter qui permet de détecter automatiquement ces "anti-patterns".

9.5.3 De l'importance de la formation

La pratique montre qu'il n'est pas simple de développer avec un modèle de composants. Cela prend du temps et nécessite une certaine expérience. Il ne suffit pas de connaître les spécifications du modèle de composants, il est plus important d'apprendre comment l'utiliser correctement.

Initialement, pour des raisons de productivité et par le fait que le niveau des ingénieurs de Dassault Systèmes est élevé, l'Object Modeler était expliqué très rapidement et une documentation expliquant les différents concepts un par un était disponible. Il en a résulté que certains ingénieurs, ne maîtrisant pas assez l'utilisation de l'Object Modeler, ont parfois recopié et adapté des macros liées à l'Object Modeler sans toujours en connaître les conséquences. Les architectes de Dassault Systèmes se sont aperçus que cela portait préjudice à l'architecture de CATIA V5 et au bon déroulement du processus de développement. Ils ont donc décidé de créer des formations spécifiques pour que les ingénieurs puissent utiliser correctement l'Object Modeler pour leurs besoins quotidiens.

Nous pouvons remarquer que ce besoin d'enseigner une bonne pratique de l'utilisation des concepts n'est pas propre à Dassault Systèmes. Par exemple, SUN, pour son langage Java, met à disposition une documentation importante où de nombreux exemples concrets sont fournis. Cette compagnie gère même un magazine électronique bi-mensuel où un thème de programmation est discuté en montrant les bonnes et mauvaises façons pour réaliser son implémentation en Java.

9.5.4 Un ou plusieurs modèles de composants ?

Il est commun d'associer à un logiciel un seul modèle de composant. Ce modèle de composants (COM, EJB, Corba...) est d'ailleurs lié à l'architecture logique. Mais depuis, quelques années la notion de multiples structures architecturales est de plus en plus acceptée : les travaux de Kruchten [Kru95], ceux de Hofmeister et al. [HNS00], la norme ANSI/IEEE Std 1471-2000 [Gro00] ainsi que nos travaux. Nous pouvons donc nous demander s'il ne pourrait pas exister plusieurs modèles de composants, un pour chaque structure architecturale.

En ce qui concerne l'architecture V5, nous avons remarqué que c'était le cas. C'est d'ailleurs comme cela que nous avons modélisé les trois structures architecturales que nous avons étudiées. Nous avons fait cela car les différents acteurs ont une notion différente de ce qu'est un composant dans l'architecture V5. Pour les développeurs, un composant correspond à la notion de composant Object Modeler alors que pour un release manager ce sera le concept de framework. Dans nos entretiens, il nous a fallu rapidement clarifier la nature du composant (composant Object Modeler, composant physique...) pour éviter les confusions.

Le fait que l'architecture V5 possède plusieurs modèles de composants n'est sans doute pas propre à cette architecture. Mais il faudrait d'autres retours d'expérience pour confirmer ou non le fait que dans les gros systèmes il existe généralement plusieurs structures architecturales et modèles de composants de natures différentes.

Nous pouvons tout de même constater que dans leur langage Java, les ingénieurs de SUN ont pris conscience de ces aspects et que certains concepts permettent d'identifier différents types de composants. Au niveau logique, nous retrouvons les modèles de composants des JavaBeans et des EJB. Mais nous pouvons aussi

remarquer que les fichiers jar, qui permettent de regrouper plusieurs fichiers pour faciliter le déploiement, peuvent aussi être considérés comme des composants.

Une difficulté pour discuter de ce point provient du fait qu'il n'existe pas de consensus sur ce qu'est un composant [FCD+01]. Les composants sont souvent définis comme des unités architecturales mais selon les personnes les composants sont vus comme des unités de conception, de réutilisation, de développement, de déploiement, de paquetage... qui peuvent prendre la forme de fichiers, de classes, de modules... Cela dépend beaucoup de la communauté (ingénierie inverse, architecture logicielle, orienté objet...) à laquelle nous nous adressons.

9.6 L'architecture logicielle et le cycle de vie

Une autre idée qui est véhiculée lorsque nous parlons d'architecture logicielle est que l'architecture logicielle d'un système reste la même de la conception à l'exécution du programme. La communauté de l'architecture logicielle a beaucoup travaillé sur la phase de conception d'une architecture et considéré pendant longtemps que ce serait la même lors de l'exécution. Pour preuve, les ADLs ne considèrent qu'une seule architecture et ne font les analyses que sur les spécifications de cette architecture de conception. Même l'outil Rapide qui a pour but de simuler le comportement architectural du système logiciel lors de son exécution, le fait sur les descriptions de cette architecture de conception. Cette vision commence à changer depuis l'étude des architectures de lignes de produits où une architecture logicielle peut servir de base pour différentes applications. Dans ce cas, l'architecture abstraite est unique mais générique.

De notre côté nous pensons que l'architecture peut évoluer et donc être différente au cours des diverses phases du cycle de vie (cf. figure 9.2). Cela permet de répondre à des besoins qui ne sont pas toujours compatibles.

Par exemple, le fait de mettre en place des mécanismes architecturaux flexibles pour faciliter la tâche du développeur pour le développement, la maintenance et l'évolution de son code ne rime généralement pas avec des besoins de performance en temps d'exécution et en place mémoire. Pour répondre à ce problème, il est possible d'imaginer une architecture de conception qui est orientée pour la gestion du code source et une autre architecture pour l'exécution qui répond aux problèmes de performance. Cette proposition n'est pas déraisonnable et est techniquement possible en exploitant la phase de compilation. Nous n'avons malheureusement pas eu le temps de faire des tests dans cette direction mais des travaux similaires ont montré l'intérêt et la faisabilité de cette approche au niveau du code [MTC97].

L'architecture qui se trouve chez le client peut très bien être différente de l'architecture qui se trouve chez l'éditeur du logiciel. Par exemple, dans le cas de CATIA V5, il existe de nombreuses configurations différentes chez les clients. L'architecture produit de l'architecture V5 a été justement conçue pour répondre

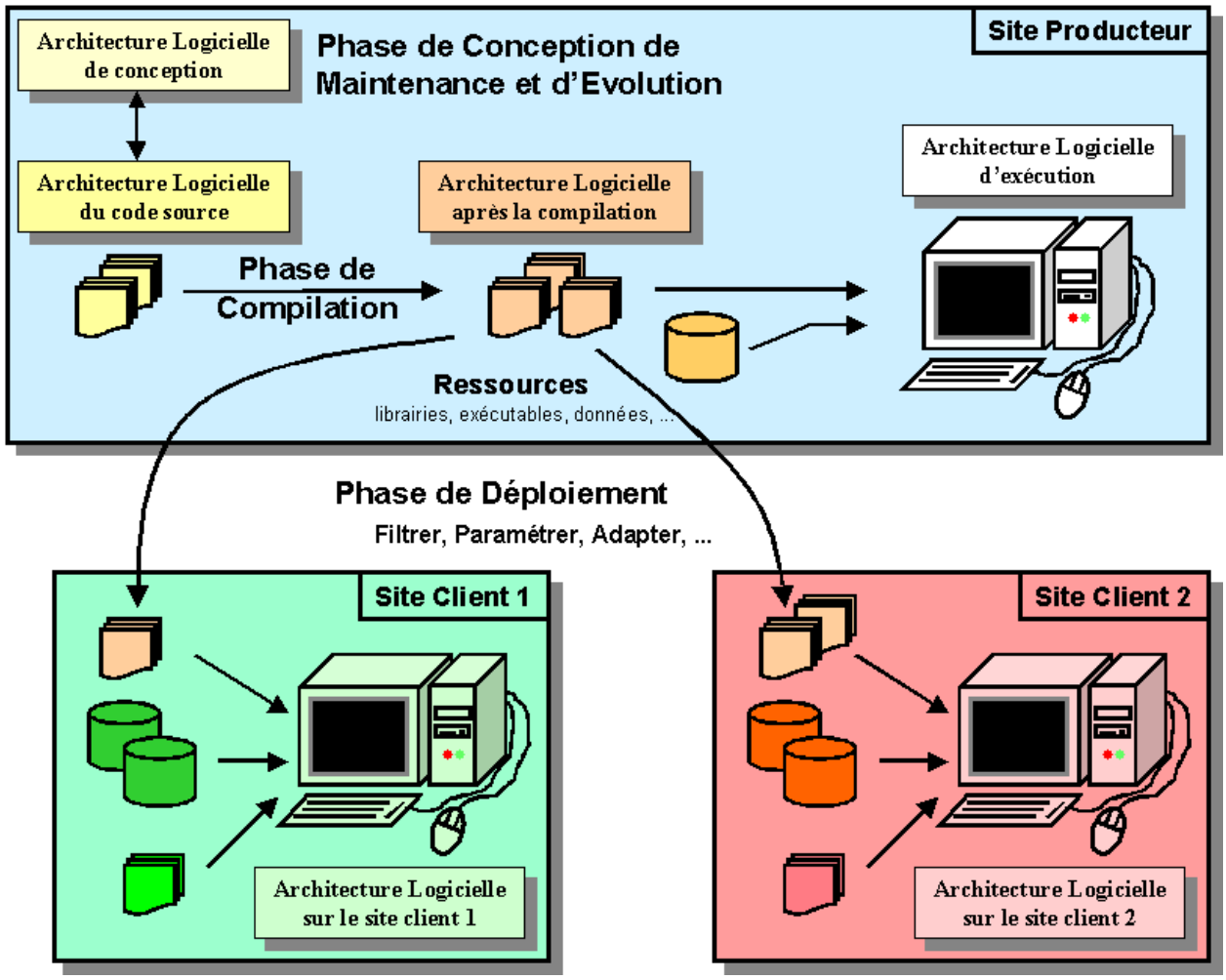


FIG. 9.2 – Architecture Logicielle et Cycle de Vie

à ce besoin de pouvoir adapter CATIA V5 aux spécificités des clients de Dassault Systèmes. Cette fois, c'est grâce à la phase de déploiement que cela est possible. L'utilisation de filtres et la gestion des licences permettent de ne recopier et de ne disposer que des fonctionnalités achetées par les clients.

En généralisant un peu, nous pouvons remarquer qu'à chaque phase correspondent des besoins spécifiques qui ne sont pas toujours compatibles. L'utilisation de différentes architectures appropriées aux besoins de chacune de ces phases est possible grâce à une étape de transformation comme la compilation ou le déploiement.

Le fait de posséder des architectures logicielles différentes suivant les phases du cycle de vie ne correspond pas toujours à un réel besoin et dépend du contexte. Il n'existe d'ailleurs que très peu de logiciels qui ont cette propriété. Mais cela peut aussi provenir du fait que les éditeurs n'ont pas les mécanismes suffisants pour utiliser cette approche. Néanmoins, il est important d'avoir conscience de ce type d'approche qui permet de proposer une solution élégante pour des besoins qui ne sont pas toujours compatibles.

9.7 En résumé

Nous venons d'exposer diverses réflexions autour de l'architecture logicielle. Les différents points abordés reflètent une partie de nos travaux sur l'architecture logicielle de CATIA V5. Notre collaboration avec Dassault Systèmes nous a permis d'acquérir une forte expérience dans ce domaine mais de nombreuses questions subsistent encore. Ces reflexions n'ont pas pour vocation de formuler des vérités mais plutôt d'engendrer des interrogations et d'éveiller la curiosité.

Nous souhaitions terminer ce rapport de thèse par une discussion autour de l'architecture logicielle pour ouvrir le débat sur une vision plus large que celle abordée dans nos travaux. La compréhension de ce qu'est une architecture logicielle ainsi que de sa pratique quotidienne restent une question d'actualité. Nous espérons que d'autres collaborations de ce type pourront avoir lieu pour permettre de formuler des réponses plus précises et de mieux comprendre les besoins industriels en matière d'architecture logicielle.

Quatrième partie

Conclusion et Perspectives

Chapitre 10

Conclusion

Notre collaboration avec Dassault Systèmes pour l'étude de l'architecture logicielle de CATIA V5 a été une expérience riche qui nous a permis d'apprendre et de comprendre de nombreux aspects du génie logiciel et plus particulièrement de l'architecture logicielle. Au cours de nos travaux, nous avons eu l'occasion d'être confontré à une grande variété de besoins architecturaux, ce qui nous a donné la possibilité d'explorer de nombreuses pistes de recherche.

Le conception et le développement de CATIA V5 sont un projet de grande envergure qui a demandé à Dassault Systèmes de relever de nombreux défis (cf. chapitre 3). Les ingénieurs de Dassault Systèmes ont dû mettre en place une architecture logicielle particulière appelée l'architecture V5 (cf. chapitre 6). La réalisation de cette architecture V5 a suivi un processus incrémental qui a duré plusieurs années. Cette architecture logicielle offre des mécanismes souples pour le développement concurrent, l'évolution et la spécialisation de CATIA V5 aux domaines d'activité des clients de Dassault Systèmes. Par contre, elle n'est pas simple à maîtriser et demande des ingénieurs qualifiés.

Actuellement, CATIA V5 représente près de 5 millions de lignes de code et est développé par près de 1 000 ingénieurs. Lorsque l'on atteint ce niveau de complexité, il n'est plus possible de gérer le développement juste en restant au niveau du code. L'objectif de nos recherches a consisté à regarder ce que le domaine de l'architecture logicielle pouvait apporter comme réponse pour améliorer le processus de développement.

Tout au long de cette thèse, nous nous sommes efforcés de suivre une démarche scientifique qui consiste à étudier les travaux existants et à essayer de les appliquer ou de les adapter à son contexte :

- **Etat de l'art** : un important travail d'état de l'art a été effectué afin de connaître les différents travaux et résultats dans le domaine de l'architecture logicielle. Nous avons cherché à comprendre ce qu'est une architecture logicielle à travers les principales recherches qui ont été menées jusqu'alors (cf. chapitre 4). Nous avons aussi repris les différents travaux sur les formalismes

de descriptions d'architectures logicielles, incluant les ADLs, qui ont longtemps été un thème de recherche important (cf. section 5.1).

- **Etude de l'existant** : avant de nous lancer dans le développement de nos prototypes, nous avons toujours regardé si nous ne pouvions pas utiliser ou adapter des outils existants (cf. sections 7.1.2, 7.2.2 et 7.3.2).

Nous avons ainsi pu confronter les principaux résultats académiques de ces dix dernières années à la réalité industrielle de Dassault Systèmes. Nos travaux ont permis de fournir une vision externe de l'architecture V5 et de proposer des solutions architecturales concrètes pour améliorer le processus de développement de CATIA V5.

Notre approche

Notre travail a consisté à étudier CATIA V5 et son processus de développement par une approche architecturale. Afin d'atteindre nos objectifs, nous avons suivi une démarche génie logiciel basée sur l'analyse des besoins.

Analyses des besoins de Dassault Systèmes

Nous avons réalisé de nombreux entretiens avec les différents acteurs de Dassault Systèmes pour pouvoir identifier leurs besoins architecturaux. Nous avons pris conscience de la difficulté de maîtriser l'architecture V5 à travers le processus de développement. Les besoins des acteurs de Dassault Systèmes se situent actuellement plus au niveau d'une aide au développement pour l'utilisation de l'architecture V5 et à l'analyse d'impact (cf. section 5.3.1). La complexité de CATIA V5 liée à la taille de ce logiciel ($\sim$5 MLoc) et le développement fortement concurrent ($\sim$1 000 ingénieurs) ne facilitent pas la maintenance et l'évolution de ce logiciel. Ces aspects ont été très peu étudiés au niveau académique puisque les recherches se sont plutôt axées sur des aspects de conception.

Notre étude sur les ADLs a permis de montrer les difficultés que nous avons rencontrées pour les utiliser dans notre contexte industriel (cf. section 5.2.1). Ils ont été pensés pour une phase de conception et ne sont pas adaptés pour des besoins liés à la maintenance et à l'évolution de logiciel comme CATIA V5. De plus, bien que le niveau d'abstraction des ADLs par la description du comportement d'un système permette de réaliser des calculs complexes (comme des analyses d'interblocage), il n'est pas adapté pour répondre aux besoins d'analyse d'impact des acteurs de Dassault Systèmes (cf. section 5.3.2).

Nous avons donc défini un niveau d'abstraction qui permet à la fois de prendre en compte l'ensemble du code source de CATIA V5 et de définir des descriptions architecturales pour répondre aux besoins de Dassault Systèmes (cf. section 5.3.3).

Compréhension et conceptualisation de l'architecture V5

Pour réaliser ce niveau d'abstraction, il a été nécessaire de comprendre et de modéliser l'architecture V5 pour définir les concepts architecturaux pertinents. La conceptualisation de l'architecture V5 a représenté un travail important d'ingénierie inverse où nous avons toujours cherché à garder le lien entre le niveau du code source et notre niveau d'abstraction. Ceci nous garantissait de pouvoir récupérer les informations architecturales directement à partir du code source.

L'étude approfondie de l'architecture V5 nous a permis de comprendre que cette architecture ne possédait pas une mais plusieurs structures architecturales. Notre conceptualisation de l'architecture V5 est donc basée sur la modélisation de différentes structures architecturales. A ce niveau, nos travaux sont similaires à ceux de Kruchten [Kru95] et ceux de Hofmeister et al. [HNS00]. Par contre nous n'avons pas retenu exactement les mêmes structures architecturales que ces précédents travaux. Nos recherches se sont concentrées sur trois des structures de l'architecture V5 : l'architecture logique, l'architecture physique et l'architecture produit (cf. chapitre 6). Pour chacune d'entre elles, nous avons défini leurs objectifs et identifié les concepts associés. Nous avons aussi répertorié les différents acteurs de Dassault Systèmes dans le processus de développement ainsi que leurs intérêts par rapport à ces trois structures architecturales (cf. chapitre 6).

A ce titre, nos travaux sont une instanciation de la norme ANSI/IEEE Std P1471-2000 [Gro00] qui préconise la modélisation des acteurs ainsi que leurs besoins et la prise en compte de différentes structures architecturales.

Réalisation de divers prototypes

L'ensemble de ces connaissances (besoins architecturaux des acteurs de Dassault Systèmes et modélisation de l'architecture V5) était alors suffisant pour pouvoir réaliser les descriptions architecturales et les diverses analyses associées utiles aux acteurs de Dassault Systèmes [FDE+01].

Afin de prendre en compte le code source existant et de pouvoir exploiter l'ensemble des données architecturales, il était nécessaire de disposer d'outils automatiques [San00]. Pour récupérer les informations architecturales directement à partir du code source, nous avons utilisé un analyseur de Dassault Systèmes. Ces données architecturales ont été stockées dans une base de données sur laquelle nous avons développé nos différents prototypes. Nous avons exploité ces données architecturales dans différentes directions : la visualisation et l'analyse avec l'OMVT (cf. section 7.1), l'exploration des différentes structures architecturales de CATIA V5 avec GSEE (cf. section 7.2) et la simulation d'une modification de l'architecture V5 avec notre prototype pour la simulation d'analyse d'impact (cf. section 7.3).

A travers ces différents développements, nous nous sommes aperçus qu'il était possible d'imaginer un environnement complet dédié à l'architecture logicielle.

Comme Balzer [Bal99], nous pensons qu'il faut élever les différents outils existants au niveau architectural. Nous avons donc proposé notre vision de ce que pourrait être un tel environnement (cf. chapitre 8).

Les contributions de cette thèse

Bien que nos résultats soient fortement liés au contexte de Dassault Systèmes, nous estimons que notre démarche peut être appliquée dans d'autres contextes et pour d'autres technologies standard (COM, CORBA, CCM, EJB, .NET). De plus, le travail réalisé au cours de cette thèse a apporté des contributions autant pour Dassault Systèmes que pour la communauté académique.

Pour la communauté académique

Notre collaboration avec Dassault Systèmes apporte une expérience industrielle assez unique de par la taille et le développement fortement concurrent de CATIA V5.

Nous avons fourni un important travail d'état de l'art qui a permis de récapituler les principales contributions ainsi que les liens entre ces travaux (cf. chapitre 4 et section 5.1). Notre travail a permis d'aborder des aspects de maintenance et d'évolution qui avaient été peu étudiés par la communauté académique. Nous avons pu montrer que dans ce contexte il était difficile de réutiliser les travaux précédents comme dans le cas des ADLs (cf. section 5.2.1).

Nous avons présenté en détail l'architecture logicielle de CATIA V5 qui offre des mécanismes originaux comme l'extension Object Modeler. La modélisation que nous avons réalisée de cette architecture logicielle par trois structures architecturales différentes renforce l'idée de Kruchten [Kru95] et de Hofmeister et al. [HNS00] qu'une architecture logicielle d'un système peut être constituée de plusieurs structures architecturales (cf. section 6.1). La prise en compte de l'environnement de développement en modélisant les acteurs ainsi que leurs besoins correspond aux recommandations de la norme ANSI/IEEE Std P1471-2000 [Gro00] (cf. section 6.2).

Un des aspects importants de cette thèse au niveau académique est l'effort que nous avons porté pour capitaliser notre expérience. Nous avons toujours eu le souci de présenter les besoins architecturaux des différents acteurs de Dassault Systèmes ainsi que notre démarche pour proposer des solutions à ces besoins (cf. section 5.2 et chapitre 7).

Comme nous avons pu le décrire dans le chapitre 4, la communauté scientifique n'a pas trouvé de consensus pour définir ce qu'est l'architecture logicielle et ce domaine manque encore de maturité pour influencer la pratique industrielle. Nous estimons que le fait de bien identifier et bien définir les besoins liés à ce domaine permettront d'apporter des réponses à ces problèmes. Nous espérons que notre expérience ira dans ce sens et qu'elle permettra de faire avancer les connaissances dans ce domaine.

Pour la société Dassault Systèmes

La thèse ayant été financée par une bourse CIFRE, il était important pour nous que nos contributions puissent servir autant à la communauté académique qu'au niveau industriel et particulièrement pour Dassault Systèmes.

De ce côté, nous avons pu montrer l'intérêt d'utiliser une approche architecturale pour améliorer le processus de développement de CATIA V5. L'analyse des besoins des acteurs de Dassault Systèmes a permis de clarifier leurs besoins architecturaux et de proposer des solutions concrètes par le biais de nos prototypes. Notre vision externe a permis de reformuler la compréhension de l'architecture V5 à travers trois de ses structures architecturales.

Les différents acteurs de Dassault Systèmes ont manifesté leur intérêt pour nos résultats et particulièrement pour notre vison externe de l'architecture V5 et pour l'OMVT. Nous avons pu sensibiliser les ingénieurs de Dassault Systèmes aux bénéfices que peut apporter une approche architecturale. Notre objectif n'était pas de fournir des outils prêts à l'emploi mais plutôt de montrer ce qu'il est possible de faire à travers nos expérimentations et nos prototypes pour une éventuelle industrialisation.

Pour moi

En ce qui me concerne, cette thèse correspondait à mon choix de réaliser une recherche appliquée et m'a permis d'atteindre mon objectif : concilier une démarche de recherche académique dans un contexte fortement industriel, tout en proposant des solutions concrètes et exploitables.

Cela n'a évidemment pas été une tâche facile et a demandé différentes compétences : une forte connaissance du domaine de l'architecture logicielle, une capacité de communication et d'écoute lors des divers entretiens avec les différents acteurs de Dassault Systèmes, une facilité d'adaptation pour bien comprendre le contexte de CATIA V5 et de Dassault Systèmes, un esprit de synthèse et de conceptualisation pour exploiter les nombreuses informations liées à l'architecture V5 et aux besoins des acteurs, une expertise technique pour concevoir et développer nos prototypes.

Cette collaboration m'a permis d'acquérir une connaissance et une expérience importante dans le domaine du génie logiciel et plus particulièrement de l'architecture logicielle.

Chapitre 11

Perspectives

La collaboration avec Dassault Systèmes pour l'étude de l'architecture logicielle de CATIA V5, nous a permis d'être confronté à une grande variété de besoins architecturaux. Nos travaux ont apporté des réponses concrètes à certains besoins architecturaux des acteurs de Dassault Systèmes.

Néanmoins il ne nous a pas été possible de tous les étudier et il a bien fallu faire des choix. De nombreuses pistes de recherches restent encore à explorer. Les perspectives de cette thèse peuvent se décliner à différents niveaux.

Concernant le travail effectué

A ce niveau différentes améliorations peuvent être apportées pour poursuivre notre travail :

- **Industrialisation de l'OMVT** : au cours de cette thèse, nous avons montré l'intérêt et les bénéfices de disposer d'un tel outil pour améliorer le développement de CATIA V5. L'ensemble des acteurs de Dassault Systèmes ont approuvé les fonctionalités disponibles dans notre prototype. Dans l'état actuel, notre prototype est suffisamment stable et performant pour pouvoir penser à l'industrialiser et à le déployer chez Dassault Systèmes.

- **Conceptualisation de l'architecture V5** : dans le temps qui nous était imparti, nous n'avons étudié que trois des structures architecturales de l'architecture V5. De plus pour l'architecture logique, nous n'avons pris en compte que l'Object Modeler et laissé de côté les autres modeleurs (cf. section 6.1.1). Nous pourrions donc continuer notre travail de modélisation en intégrant ces modeleurs et les autres structures architecturales de CATIA V5.

- **Gestion de l'évolution** : nos travaux sur l'analyse d'impact pour la modification de l'architecture logicielle de CATIA V5 n'en sont qu'à leur début. Notre prototype est à un stade préliminaire et de nombreuses améliorations peuvent être apportées (cf. section 7.3.4).

- **Environnement architectural** : dans le chapitre 8, nous avons proposé un environnement dédié à l'architecture logicielle. Les recherches que nous avons menées ont permis de réaliser une partie des fonctionnalités d'un tel environnement. Il serait donc intéressant de poursuivre nos travaux pour étudier et intégrer les autres fonctionnalités manquantes.

Questions ouvertes autour de l'architecture logicielle

Dans le dernier chapitre, nous avons exposé diverses réflexions autour de l'architecture logicielle : étude de l'architecture logicielle par une approche montante ou descendante ? , un langage propre pour l'architecture logicielle ? , l'architecture logicielle et le cycle de vie...

Le but de ce chapitre n'était pas d'affirmer des positions fermes mais plutôt d'ouvrir la discussion. Les différents points abordés n'ont pas pu être étudié suffisamment pour avois un avis précis. Il faudrait donc continuer nos recherches sur ces aspects pour pouvoir les confirmer ou les infirmer.

D'autes collaborations industrielles

Bien que nous ayons cherché à nous abstraire au mieux de notre contexte, nous avons conscience que notre expérience industrielle reste liée à notre collaboration avec Dassault Systèmes. Afin d'élargir notre vision de l'architecture logicielle et de se rendre compte de ce qui est spécifique ou non à Dassault Systèmes, il serait intéressant de pouvoir étudier d'autres contextes industriels. Ceci permettrait aussi d'affiner notre connaissance et de pouvoir apporter des réponses aux différentes interrogations du dernier chapitre.

Comme nous avons pu le souligner dans le chapitre 4, il n'existe pas de consensus pour définir ce qu'est l'architecture logicielle. Nous estimons que pour pouvoir apporter des éléments de réponse à cette question, il faut pouvoir bien identifier les besoins architecturaux. Notre expérience avec Dassault Systèmes a été riche à ce niveau mais nous pourrions encore améliorer ces connaissances à travers d'autres collaborations industrielles.

Un CATIA pour le génie logiciel

Notre étude de l'architecture logicielle de CATIA V5, nous a amené à consolider nos connaissances pour le domaine de la CAO/CFAO. Nous avons pu constater que de nombreux besoins et fonctionnalités (visualisation complexe, édition, analyse d'impact...) que nous avons proposés pour un environnement dédié à l'architecture logicielle sont disponibles dans CATIA V5 pour les produits manufacturés.

Comme CATIA est une infrastructure évolutive que nous pouvons spécialiser pour son domaine d'activité, il est possible d'imaginer de vouloir créer un CATIA pour le génie logiciel. Actuellement, les outils pour l'architecture logicielle sont encore loin d'être au niveau de CATIA V5 mais il ne faut pas oublier qu'il a fallu plus de 20 ans à Dassault Systèmes pour arriver au niveau actuel dans le domaine de la conception mécanique.

Nous pensons que dans un futur proche, il sera possible de développer son logiciel comme un constructeur d'automobile conçoit et fabrique sa voiture. Il faudra sûrement encore attendre une quinzaine d'années avant qu'il n'existe véritablement d'environnement pour le génie logiciel et l'architecture logicielle aussi performant que pour la CAO.

Cinquième partie

Annexes

Annexe A

Publications et Activités au cours de ma Thèse

A.1 Mes Publications

Cette section réunit mes diverses publications qui ont été acceptées au cours de ma thèse. Un effort particulier a été réalisé pour transmettre au mieux notre expérience industrielle et nos résultats à la communauté scientifique. Dans le but de faciliter la lecture, nous les avons regroupées par type de publication.

Chapitres de Livre

[EFS2002] Jacky Estublier, Jean-Marie Favre and Rémy Sanlaville. *An Industrial Experience with Dassault Systèmes' Component Model.* Book chapter in Building Reliable Component-Based Systems, I. Crnkovic, M. Larsson editors, Archtech House publishers. To appear, 2002.

Un autre chapitre de livre a été proposé et est en cours de relecture.

[SLEFew] Rémy Sanlaville, Yves Ledru, Jacky Estublier and Jean-Marie Favre. *Architectural Environment for the Evolution of Complex Software.* Chapter proposal for the book 4-Volume books on Software Architectures, PLAs, Components and Enterprise Frameworks, Mohamed Fayad, David Garlan, and Wolfgang Pree editors. In review. http ://www.cse.unl.edu/~fayad/Books/NewBooks/

Revue

[DES2001] Frédéric Duclos, Jacky Estublier and Rémy Sanlaville. *Architectures Ouvertes pour l'Adaptation des Logiciels.* Revue Génie Logiciel, Numéro 58, pp. 19-25, Septembre 2001.

Conférences Internationales avec comité de sélection

[DES2000] Frédéric Duclos, Jacky Estublier and Rémy Sanlaville. *Architectures Ouvertes pour l'Adaptation des Logiciels.* Proceedings of the 13^{th} International Conference on Software and Systems Engineering and their Applications (ICSSEA'2000). Vol. 1, Paris, France, December 2000.

[FC+2001] Jean-Marie Favre, Humberto Cevantes, Frédéric Duclos, Rémy Sanlaville and Jacky Estublier. *Issues in Reengineering the Architecture of Component-Based Software.* SWARM forum (Software Architecture Recovery and Modeling) at WCRE'2001 (Working Conference on Reverse Engineering) Stuttgart, Germany, October 2001.

[FD+2001] Jean-Marie Favre, Frédéric Duclos, Jacky Estublier, Rémy Sanlaville and Jean-Jacques Auffret. *Reverse Engineering a Large Component-based Software Product.* Proceedings of the 5^{th} European Conference on Software Maintenance and Reengineering (CSMR'2001). Lisbon, Portugal, pp. 95-104, March 2001.

[LPS99] Yves Ledru, Marie-Laure Potet and Rémy Sanlaville. *VDM Modules.* In John Fitzgerald and Peter Gorm Larsen, editors, VDM in Practice, pp. 1-12, September 1999.

[LSE2000] Yves Ledru, Rémy Sanlaville and Jacky Estublier. *Defining an Architecture Description Language for Dassault Systèmes.* In B. Balzer and Henk Obbink, editors, Proceedings of the 4^{th} International Software Architecture Workshop (ISAW4). Limerick, Ireland, pp. 115-120, June 2000.

[San2000] Rémy Sanlaville. *Software Architecture : An Industrial Experiment with Dassault Systèmes.* Actes du Symposium Doctoral ASE'2000. In Renaud Marlet and John Penix, editors, Technical Report, Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), Number PI-1353, pp. 25-30, November, 2000.

[SFL2001] Rémy Sanlaville, Jean-Marie Favre and Yves Ledru. *Helping Various Stakeholders to Understand a Very Large Component-Based Software.* Proceedings of the 27^{th} EUROMICRO Conference, pages 104-111, Warsaw, Poland, September, 2001.

A.2 Mes Présentations

Comme la majorité des recherches et des résultats sont restés au niveau académique, le contexte fortement industriel de nos travaux a été un aspect apprécié. A ce titre, j'ai été sollicité à plusieurs reprises pour exposer notre expérience et nos résultats :

- J'ai contribué à une journée dédiée à l'architecture logicielle et organisée par le club de chercheurs français sur les systèmes informatiques de confiance de la S.E.E. (Société des Electriciens et des Electroniciens) en octobre 2001. Durant cette journée, j'ai pu relater notre expérience industrielle avec Dassault Systèmes.

- J'ai été invité à être un membre d'une table ronde sur le thème des composants : "Isn't Software Develoment Today Already Component-based ?". Cette table ronde a eu lieu lors de la conférence internationale Euromicro Workshop on Component-Based Software Engineering qui s'est déroulée à Varsovie en septembre 2001.

- J'ai pu exposer mes recherches dans le cadre des séminaires du DEA Systèmes et Communication de Grenoble en février 2001.

- J'ai effectué une présentation de nos travaux et de nos résultats au laboratoire DTL/ASR de France Telecom R&D de Grenoble en mars 2002.

A.3 Mes Livrables

Dans le cadre de la collaboration entre le laboratoire LSR et Dassault Systèmes, nous devions chaque année fournir un certain nombre de livrables pour vérifier l'avancée de nos travaux. Ces livrables étaient définis au début de l'année par mon responsable Dassault Systèmes, mon directeur de thèse et moi-même. Ceci a permis de structurer et de planifier mon travail sur l'ensemble de ces trois années.

Année 1999

Cette première année de thèse était focalisée sur la compréhension de l'Object Modeler et sur l'élaboration d'un langage de description d'architecture pour ce dernier.

Délivrables 1999

- **Cahier des charges** : ce document avait pour but de décrire l'ensemble des aspects qui devaient être présents dans le langage de description d'architecture adapté à l'Object Modeler.

- **Manuel de référence** : ce document devait apporter une aide aux différents acteurs (développeurs, analystes, architectes...) de Dassault Systèmes pour l'utilisation de ce langage de description d'architecture.

- **Syntaxe graphique** : ce document fournissait la syntaxe graphique associée à ce langage de description d'architecture. Cette syntaxe graphique a été reprise lors du développement de l'OMVT et a participé au bon accueil de cet outil par les ingénieurs de Dassault Systèmes.

Annexes 1999

- **Glossaire** : ce document permettait de définir et de clarifier les concepts utilisés dans nos différents délivrables ainsi que les concepts de l'Object Modeler.
- **Sujet Stage Rosana Angles** : ce document est le sujet de stage que j'avais proposé aux élèves de troisième année de l'école d'ingénieur de l'ENSIMAG de Grenoble pour leur projet de fin d'étude et qui a été retenu par une étudiante (Rosana Angles).
- **Rapport ANRT** : ma thèse étant financée par une bourse CIFRE, je devais fournir chaque année un rapport à l'ANRT (Association Nationale de la Recherche Technique) qui est l'organisme responsable de la gestion et de l'animation des conventions CIFRE, pour le compte du ministère chargé de la Recherche. Ce rapport permet à l'ANRT de vérifier le bon déroulement de la thèse.

Année 2000

Cette deuxième année de thèse a été particulièrement productive et constructive. Elle a été marquée par la réalisation de l'OMVT et par nos diverses publications. C'est aussi au cours de cette année que nous avons découvert que l'Object Modeler n'était pas la seule structure architecturale de CATIA V5. Nous avons donc étudié et modélisé l'architecture V5 à travers trois de ses structures architecturales.

Délivrables 2000

- **Prototype Object Modeler Visualisation Tool** : suite à la compréhension de l'Object Modeler et des besoins architecturaux des acteurs de Dassault Systèmes, nous avons pris conscience de l'importance de disposer d'un outil spécifique. L'élaboration et le développement de ce prototype en quelques mois a été possible grâce au travail collaboratif de six personnes dont Rosana Angles.
- **Soumissions pour des publications** : ce délivrable regroupe l'ensemble de nos publications pour l'année 2000. Nos soumissions ont été acceptées dans quatre conférences internationales : ISAW-4 (le workshop de référence pour l'architecture logicielle), ASE'2000 (au Doctoral Symposium), ICSSEA'2000 et CSMR'2001.
- **Analyses d'impact à DS - Les Besoins** : ce document recense et structure par type d'acteur les différents besoins de Dassault Systèmes pour l'analyse d'impact.

Annexes 2000

- **Rapport sur les présentations de l'OMVT à Dassault Systèmes** : ce rapport fait la synthèse des commentaires (encouragements et améliorations à apporter) des différents acteurs de Dassault Systèmes qui ont pu assister à nos présentations de l'OMVT.

- **Stage Rosana Angles** : cette annexe regroupe le rapport et la présentation pour la soutenance de stage de Rosana Angles. Le travail effectué par Rosana Angles a été d'une grande utilité pour la réalisation de l'OMVT et la collaboration avec Dassault Systèmes.

- **Rapport ANRT** : ce document est le rapport de l'année 2000 pour l'ANRT qui récapitule l'avancée de nos travaux pour cette année.

Année 2001

Cette troisième année de thèse a permis de confirmer nos résultats précédents et de poursuivre notre compréhension de l'architecture V5. Nous avons exploré, grâce à l'environnement GSEE, les trois structures architecturales de l'architecture V5 que nous avions étudiées et développé un premier prototype pour calculer l'impact d'une modification sur l'architecture logicielle de CATIA V5. Nous avons poursuivi notre effort de diffusion de nos résultats à travers différentes publications.

Délivrables 2001

- **Rapport de Thèse** : ce document correspond à ce rapport de thèse. Il a demandé beaucoup de temps et de travail pour faire une synthèse de l'ensemble de nos travaux. Un effort particulier a été réalisé pour situer et comparer nos travaux avec les principales publications sur l'architecture logicielle. Ce rapport récapitule aussi notre modélisation de l'architecture V5 en trois structures architecturales et les prototypes que nous avons développés pour répondre à certains besoins des acteurs de Dassault Systèmes.

- **Prototype pour la simulation d'analyses d'impact** : suite au rapport sur les besoins en terme d'analyse d'impact des acteurs de Dassault Systèmes, j'ai développé un premier prototype pour simuler l'impact d'une modification de l'architecture logicielle de CATIA V5. Bien que nous n'ayons pas eu le temps d'en tirer profit, nous avons pu identifier les concepts importants.

- **Soumissions pour des publications** : ce délivrable regroupe l'ensemble de nos publications pour l'année 2001. Cette année encore, nous avons réussi à publier nos résultats à travers un chapitre de livre, un article dans une revue française et deux articles dans des conférences internationales (ECBSE'2001 et WCRE'2001).

Annexes 2001

- **Présentations** : cette annexe correspond aux deux présentations et à la table ronde que j'ai effectuées au cours de cette année 2001.

- **Stage de Tahia Ben Haj Abdellatif** : cette annexe regroupe le rapport et la présentation pour la soutenance de stage de Tahia Ben Haj Abdellatif. L'objectif de ce stage était d'apporter une connaissance sur les technologies liées à XML pour notre équipe de recherche.

- **Rapport ANRT** : ce document est le dernier rapport pour l'ANRT qui fait l'état de nos travaux pour cette troisième année et la synthèse de l'ensemble du travail réalisé au cours de cette thèse.

A.4 Encadrement de stagiaires

Afin d'accomplir mon travail de thèse, j'ai eu l'occasion d'encadrer deux étudiantes. Je tiens particulièrement à les remercier pour leur convivialité, leur enthousiasme et le travail qu'elles ont produit :

- **Rosana Angles pour son stage de fin d'étude de 3^e année d'ingénieur ENSIMAG** : l'objectif du stage de Rosana était de réaliser une étude de l'existant des outils pour la représentation de gros graphes de données et de participer au développement de l'OMVT. Elle a conçu et développé entièrement l'API qui est disponible pour utiliser notre base de données ObjectStore qui contient les données architecturales de CATIA V5. L'ensemble des outils de notre équipe de recherche est d'ailleurs basé sur cet API pour exploiter ces données architecturales. Elle a ensuite fortement contribué au développement de l'OMVT en réalisant les premiers tests et en prenant en charge un certain nombre de nos éditeurs. Elle a fourni un excellent travail qui a été directement exploité dans le cadre de notre collaboration avec Dassault Systèmes.

- **Tahia Ben Haj Abdellatif pour son stage de TER (Travaux d'Etudes et de Recherche)** : ce stage s'inscrit dans la formation de maîtrise d'informatique de Grenoble et a pour but de faire découvrir le monde de la recherche au stagiaire par l'intermédiaire d'une activité de type recherche à un niveau élémentaire. L'objectif du stage de Tahia était d'apporter une connaissance sur les technologies liées à XML pour notre équipe de recherche. Pour cela, elle a effectué un état de l'art de ces différentes technologies et réalisé diverses expérimentations pour implémenter des parsers SAX et DOM et utiliser XSLT. Ensuite, elle a décrit une DTD pour l'architecture V5 et développé un petit prototype pour manipuler des entités de CATIA V5 à partir de notre base de données.

Annexe B

Exemple d'une spécification en WRIGHT

Pour faciliter la compréhension, nous prendrons un exemple simple pour la création d'un index. Comme le montre la figure B.1, l'architecture logicielle du système est un pipeline.

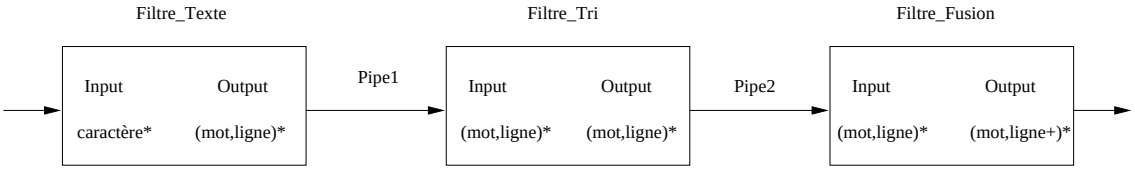


FIG. B.1 – Architecture logicielle en pipeline pour la création d'un index

La spécification complète de cet exemple en Wright est définie par la création d'un style PipeFilter pour la description du connecteur pipe (section B.1) et de la description complète du système qui utilise ce style (section B.2).

Pour de plus amples informations, veuillez vous référer à mon rapport de DEA [San97].

B.1 Le style PipeFilter

Style PipeFilter

Interface Type DataOutput = $(\overline{write!x} \rightarrow DataOutput \sqcap \overline{close} \rightarrow \S)$

Interface Type DataInput = $\overline{read} \rightarrow \; (data?x \rightarrow DataInput$
$\Box$
$end\!-\!of\!-\!data \rightarrow \overline{close} \rightarrow \S)$

Connector Pipe
Role Source = DataOutput
Role Sink = DataInput

Glue = $Open_{\langle\rangle}$

where $Open_{\langle\rangle} = Source.write?x \rightarrow Open_{\langle x\rangle}$

$\Box\ Source.close \rightarrow Closed_{\langle\rangle}$

$Open_s a_{\langle x\rangle} = Sink.read \rightarrow \overline{Sink.data!x} \rightarrow Open_s$

$\Box\ Source.write?y \rightarrow Open_{\langle y\rangle} a_s a_{\langle x\rangle}$

$\Box\ Source.close \rightarrow Closed_s a_{\langle x\rangle}$

$Closed_s a_{\langle x\rangle} = Sink.read \rightarrow \overline{Sink.data!x} \rightarrow Closed_s$

$Closed_{\langle\rangle} = Sink.read \rightarrow \overline{Sink.end-of-data} \rightarrow Sink.close \rightarrow \S$

Constraints

$\forall\ c : Connectors \bullet Type(c) =$ Pipe

$\forall\ c : Components;\ p : Port \mid p \in Ports(c) \bullet Type(p) =$ DataInput $\vee$ $Type(p) =$ DataOutput

End Style

Cette spécification définit notre style PipeFilter. Ce style déclare un connecteur Pipe qui possède deux rôles : un rôle Sink qui lit l'ensemble des données qui arrivent et s'arrête au signal *end-of-data* et un rôle Source pour les renvoyer. Le comportement de ce connecteur est fourni par la Glue qui spécifie que les données reçues par le connecteur doivent être renvoyées dans le même ordre et que le connecteur possède un buffer pour gérer le flux de données. Enfin le style définit deux contraintes qui doivent être vérifiées par l'ensemble des configurations qui utilisent ce style. Ces contraintes stipulent que tous les connecteurs de la configuration doivent être de type Pipe et que tous les composants du système ne doivent posséder que des ports de type DataInput ou DataOutput.

B.2 La configuration du système

Configuration Index

Style PipeFilter

Component Filtre_Texte

Port Input = DataInput

Port Output = DataOutput

Computation =

$\overline{Input.read} \rightarrow$ $((Input.data?c \rightarrow$ (**Computation**

$\sqcap$

$(\overline{Output.write!(w,i)} \rightarrow$ **Computation**)))

$\Box$

$(Input.end-of-data \rightarrow \overline{Input.close} \rightarrow$ $((\overline{Output.close} \rightarrow \S)$

$\sqcap$

$(\overline{Output.write!(w,i)} \rightarrow$
$\overline{Output.close} \rightarrow \S))))$

Component Filtre_Tri

Port Input = DataInput

Port Output = DataOutput

Computation =

let $SendData = (\overline{Output.write!(w,i)} \rightarrow SendData) \sqcap (\overline{Output.close} \rightarrow \S)$

in $\overline{Input.read} \rightarrow$ $((Input.data?(w,i) \rightarrow$ **Computation**)

$\Box$

$(Input.end-of-data \rightarrow \overline{Input.close} \rightarrow SendData))$

Component Filtre_Fusion

Port Input = DataInput

Port Output = DataOutput

Computation =

$\overline{Input.read} \rightarrow$ $((Input.data?(w,i) \rightarrow$ (**Computation**

$\sqcap$

$(\overline{Output.write!(w1,J)} \rightarrow$ **Computation**)))

$\Box$

$(Input.end-of-data \rightarrow \overline{Input.close} \rightarrow$ $((\overline{Output.close} \rightarrow \S)$

$\sqcap$

$(\overline{Output.write!(w,I)} \rightarrow$
$\overline{Output.close} \rightarrow \S))))$

Instances

Texte	:	Filtre_Texte
Tri	:	Filtre_Tri
Fusion	:	Filtre_Fusion
P1,P2	:	Pipe

Attachments	Texte.Output	**as**	P1.Source
	Tri.Input	**as**	P1.Sink
	Tri.Output	**as**	P2.Source
	Fusion.Input	**as**	P2.Sink

End Configuration

Cette spécification décrit la configuration de notre système pour la création d'un index. Nous indiquons en premier lieu les différents types de composants et de connecteurs dont nous allons nous servir. Nous déclarons utiliser le style Pipe-Filter défini précédemment et spécifions trois nouveaux types de composants : le composant Filtre_Texte qui filtre le texte pour renvoyer des couples (mot, numéro de ligne), le composant Filtre_Tri qui se charge du tri alphabétique de ces couples et le composant Filtre_Fusion qui gère les éventuels doublons. Ensuite, nous indiquons les différentes instances de composants et de connecteurs de notre système. Nous déclarons une instance pour chacun de ces trois types de composants et deux instances du connecteur Pipe. Enfin, nous indiquons la topologie de notre système par la connexion des différentes instances à travers leurs ports ou leurs rôles comme cela est indiqué dans la figure B.1.

Annexe C

Les "4+1" Vues de Kruchten

Kruchten a proposé une approche pour modéliser une architecture logicielle à travers différentes structures [Kru95]. Il a identifié et formalisé quatre structures (vue logique, vue physique, vue processus et vue développement) plus une qui est redondante avec les précédentes pour illustrer à partir de scénarios la coopération entre ces différentes structures.

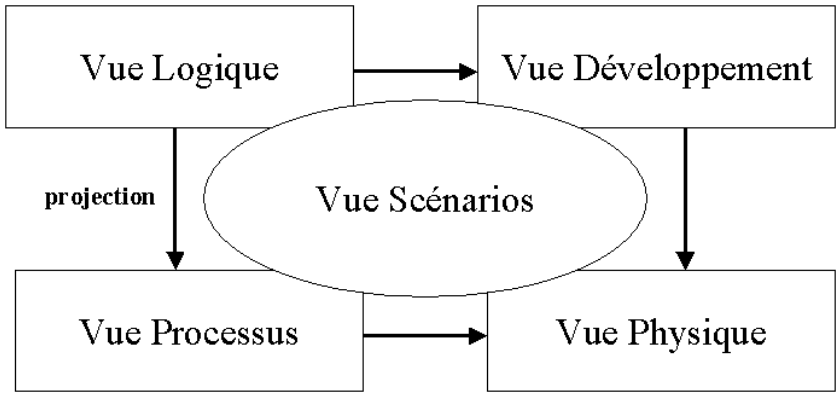


FIG. C.1 – Le modèle "4+1" vues de Kruchten

C.1 La vue logique

La vue logique s'attache à décrire les besoins fonctionnels (quels sont les services que le système doit fournir aux utilisateurs). Elle est destinée aux ingénieurs qui sont chargés de la modélisation du logiciel en collaboration avec les utilisateurs et les spécialistes du domaine. Elle se rattache à une modélisation orienté objet puisqu'elle définit les abstractions en termes de classes, objets et leurs relations (association, agrégation, utilisation, héritage, instanciation). Cette vue correspond à un diagramme de classe utilisant la notation de Booch (cf. figure C.2). D'autres informations peuvent venir compléter le diagramme de classe. Par exemple, le regroupement de classes ayant des fonctionnalités communes, ou la description du comportement interne d'un objet par l'utilisation d'un diagramme d'états-transitions.

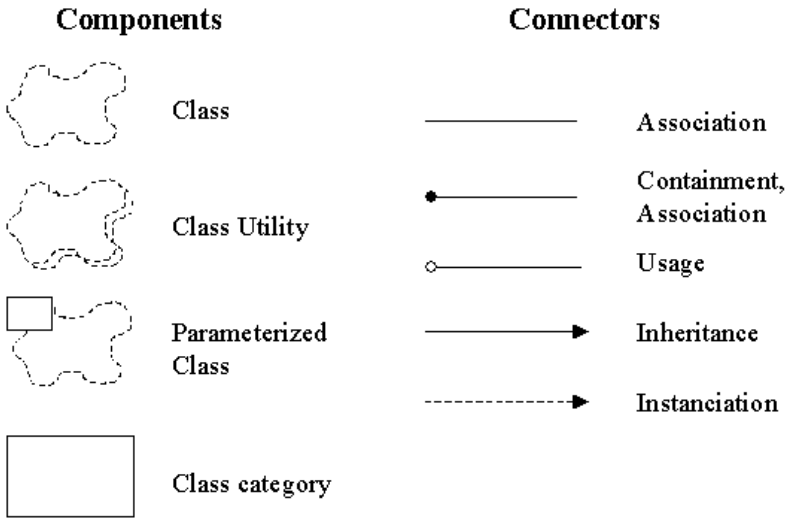


FIG. C.2 – Notation pour la vue logique

C.2 La vue processus

La vue processus prend en charge des aspects non fonctionnels comme la performance et la disponibilité. Elle aborde des problèmes liés à la distribution et la synchronisation ainsi que ceux liés aux mécanismes de réplication nécessaires à la tolérance aux fautes. Cette vue est utilisée par les ingénieurs chargés de regrouper et d'affecter les abstractions de la vue logique dans des processus ou tâches ("threads"). Pour faciliter la description, il est possible d'utiliser différents niveaux d'abstraction. Au niveau d'abstraction le plus élevé, les entités de cette vue sont des processus définis comme des ensembles de tâches pouvant s'exécuter de manière autonome, être répliqués, arrêtés ou relancés. Le résultat est un réseau logique de processus communicants qui devront être liés aux entités machines de la vue physique. C'est la notation proposée par Booch pour décrire les communications entre les tâches Ada qui a été retenue pour cette vue.

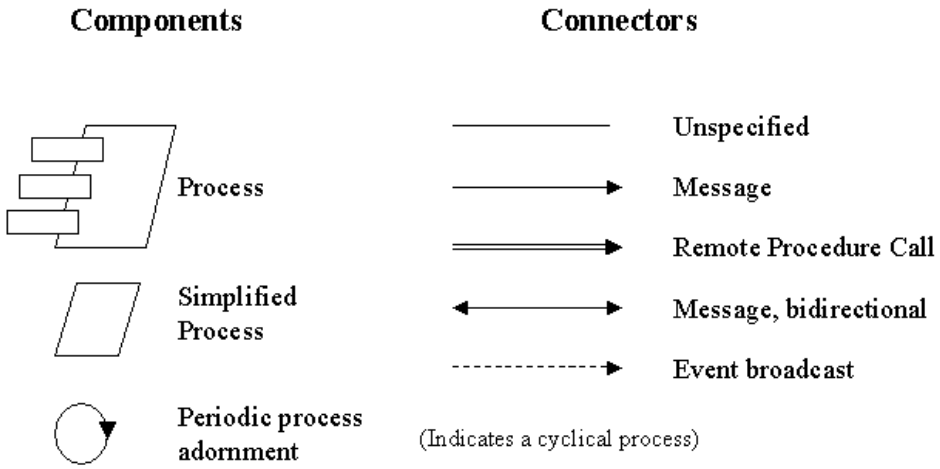


FIG. C.3 – Notation pour la vue processus

C.3 La vue développement

La vue développement se focalise sur l'organisation du code en différents modules et leurs dépendances. Le logiciel peut être découpé en sous-systèmes qui peuvent être développés par un ou plusieurs développeurs. La description peut utiliser une décomposition hiérarchique en couches. Chaque couche décrivant les services disponibles pour la couche supérieure. Cette vue sert aussi de référence pour planifier l'échéancier et la répartition du développement entre les différentes équipes. Elle est plus particulièrement destinée au chef de projet. Une nouvelle fois, c'est une variante de la notation de Booch appliquée au niveau architectural qui est utilisée.

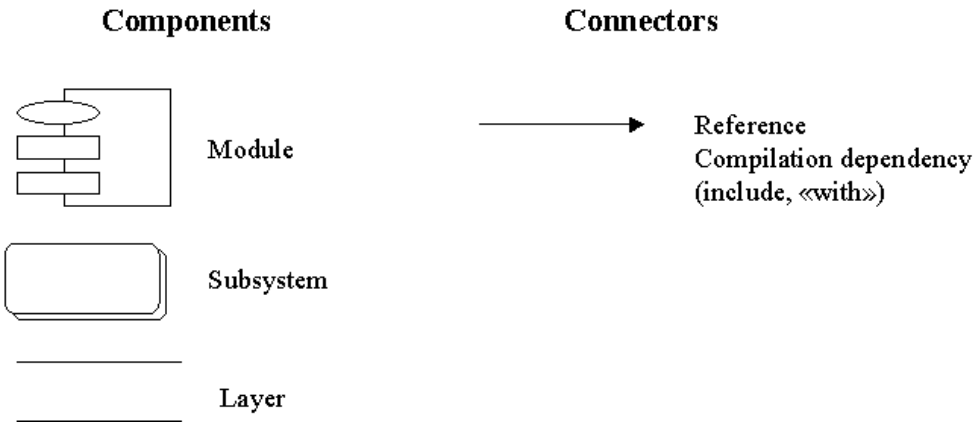


FIG. C.4 – Notation pour la vue de développement

C.4 La vue physique

Comme pour la vue processus, la vue physique prend en compte des aspects non fonctionnels de performance, de tolérance aux fautes, passage à l'échelle, disponibilité. Par contre, nous nous plaçons au niveau matériel où les différentes entités sont des machines placées sur un réseau. Les liens entre les nœuds du réseau modélisent les différents types de communications (communication permanente ou non, lien uni- ou bi-directionnel, taille de la bande passante...). Cette vue s'adresse aux ingénieurs système et réseau qui spécifient comment les différents processus du logiciel doivent être répartis sur le réseau. C'est une notation informelle (cf. figure C.5) qui est utilisée avec des boîtes et différents types de liens (lignes continues ou non, avec des flèches aux extrémités ou non, avec différentes épaisseurs...). L'auteur n'indique pas si cette notation correspond à une syntaxe graphique existante.

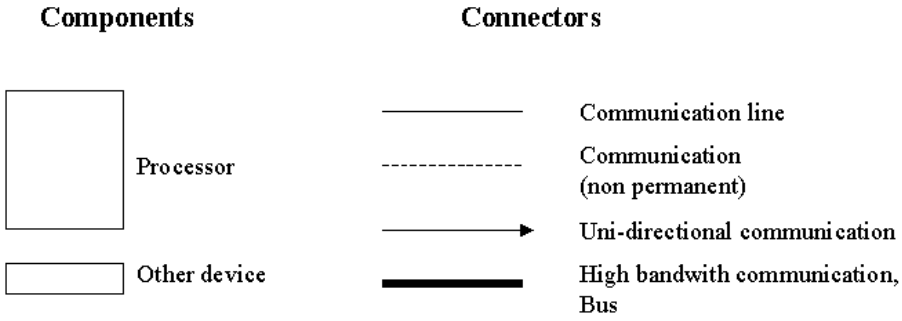


FIG. C.5 – Notation pour la vue physique

C.5 La vue scénarios

La vue scénarios sert à illustrer des exemples de coopération entre les quatre vues précédentes. Elle décrit des séquences d'interactions entre objets et processus par des diagrammes d'interaction entre objets introduits par Rubin et Goldberg [RG92]. Cette vue est redondante par rapport aux précédentes (d'où le "+1"). Elle ne sert pas principalement à la description de l'architecture du logiciel mais plutôt à illustrer son fonctionnement à partir de scénarios pour guider la conception de l'architecture. Le deuxième intérêt de cette vue est de pouvoir servir de base pour le conception des tests de validation de l'architecture une fois que la conception est terminée. Ces tests de validation pourront être effectués autant sur le papier que sur un premier prototype. La notation est similaire à celle de la vue logique pour les entités (cf. figure C.2) et celle de la vue processus pour les liens entre ces entités (cf. figure C.3).

Annexe D

Les 4 vues de Hofmeister et al.

Les travaux de Hofmeister, Nord et Soni ont permis de confirmer l'intérêt des travaux de Kruchten [Kru95] pour la spécification d'une architecture logicielle à travers différentes structures. Par contre, ils n'ont pas repris exactement les mêmes structures architecturales. Pour eux, une architecture logicielle peut être modélisée par quatre structures : vue conceptuelle, vue module, vue du code et vue exécution (cf. figure D.1).

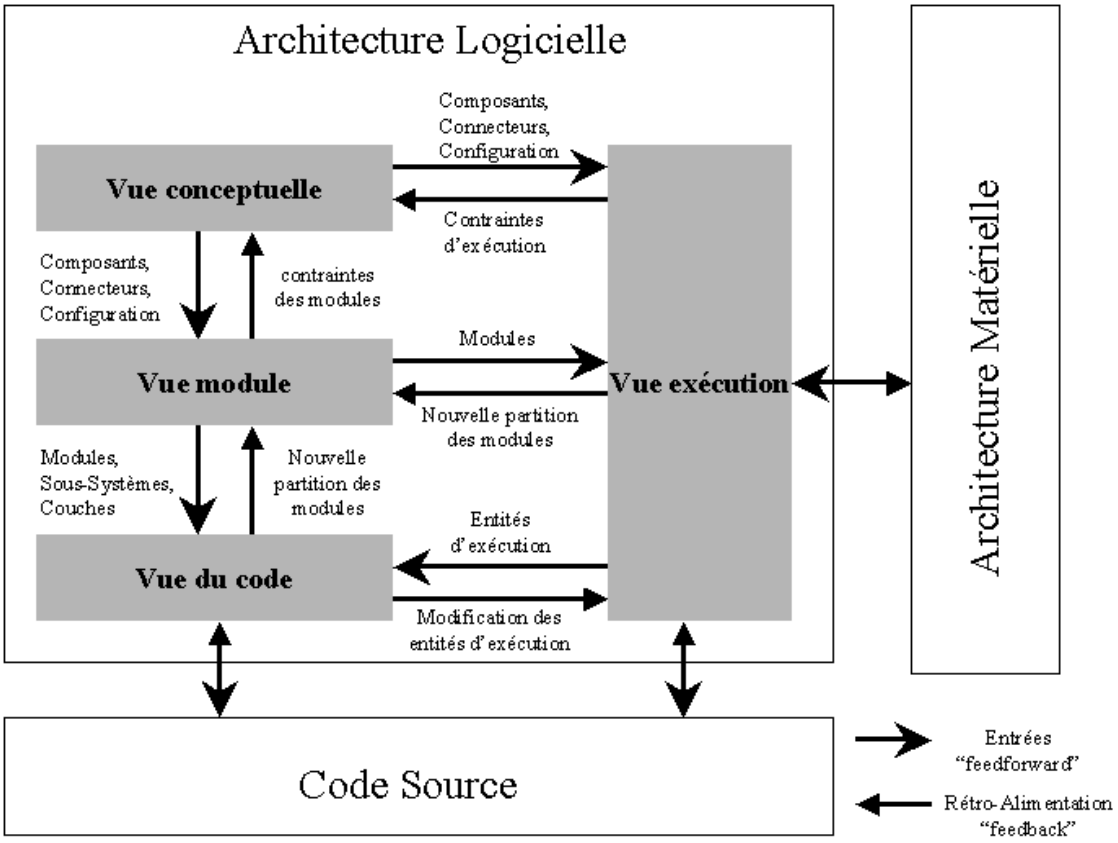


FIG. D.1 – Les quatre vues d'une architecture logicielle pour Hofmeister et al.

D.1 La vue conceptuelle

La vue conceptuelle est celle qui est la plus proche du domaine d'application car c'est celle qui est la moins contrainte par les plateformes logicielles et matérielles. Avec cette vue, il est possible de modéliser le produit en termes d'une collection de composants et de connecteurs conceptuels. En fait, elle est comparable aux ADLs standards.

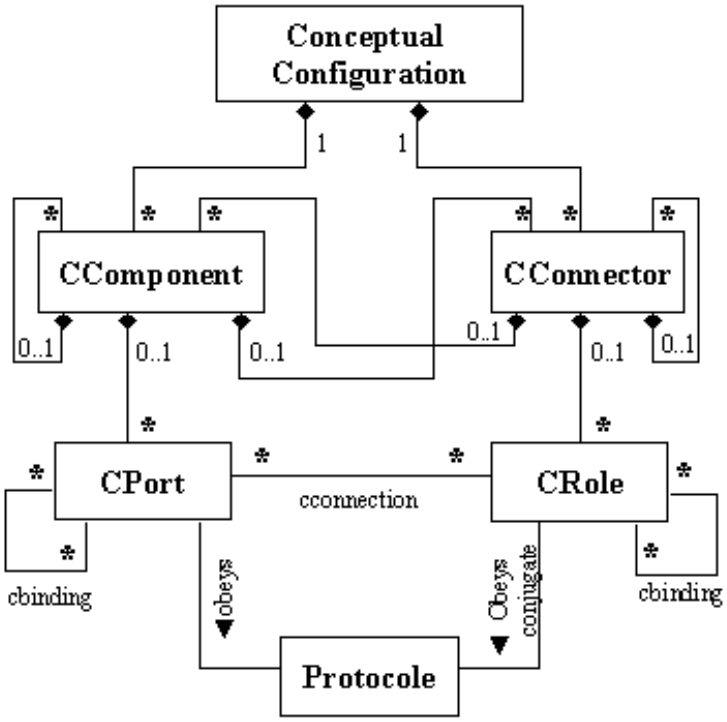


FIG. D.2 – Méta-modèle de la vue conceptuelle

D.2 La vue module

Le principal objectif de cette vue module est de prendre en compte les besoins liés à l'implémentation du système et à l'organisation de développement. A la différence de la vue conceptuelle, cette vue permet de décrire explicitement comment les différents éléments interagissent avec la plateforme logicielle (comme les services du système d'exploitation, les APIs d'une librairie spécialisée...). Cette vue n'est pas simplement un raffinement de la vue conceptuelle. Elle décrit aussi une nouvelle structure à base de sous-systèmes, de modules et de couches hiérarchiques. Elle définit aussi la répartition des éléments conceptuels par rapport aux différents modules. Il est ainsi possible de structurer différemment l'application en regroupant les éléments conceptuels par domaine (module pour l'interface graphique, module de calcul, module pour le réseau...). Cette nouvelle structure facilite la gestion de la complexité du système et la répartition du travail collaboratif. Dans les systèmes de grande taille, ces deux structures (conceptuelle et module) sont généralement

différentes. Cette vue utilise les travaux sur les MILs (Module Interconnection Language) [DK76, PDN86] et les techniques de paquetages Ada et Java.

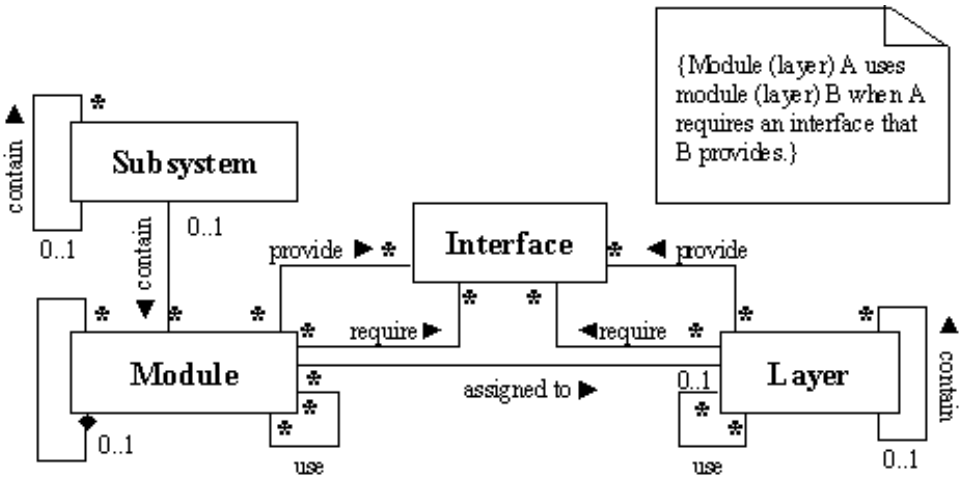


FIG. D.3 – Méta-modèle de la vue module

D.3 La vue du code

La vue du code modélise l'organisation du code source sous forme de fichiers sources (.h, .cpp, .java...), librairies (dlls...), fichiers binaires (.o, .class...), exécutables (.exe...) et de fichiers de configuration pour le déploiement. Généralement il existe différentes versions pour ces fichiers. Elle a donc aussi pour objectif de décrire les décisions relatives à la gestion de configuration. Le principal but de cette vue est de faciliter la construction, l'intégration, l'installation et le test du système tout en respectant l'intégrité des trois autres vues.

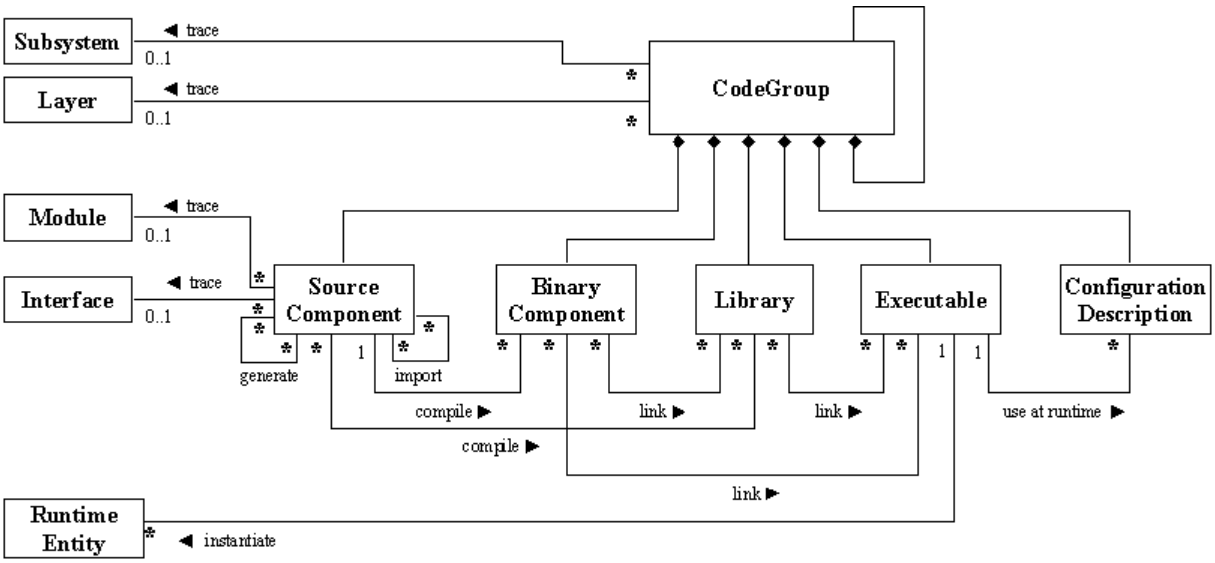


FIG. D.4 – Méta-modèle de la vue code

D.4 La vue exécution

La vue exécution prend en compte les aspects dynamiques du logiciel. Elle décrit le système en terme d'éléments exécutables de la plateforme (processus, tâches, sockets, DLLs...), de ressources de la plateforme (mémoire, espace d'adressage, processeur(s)...), de mécanismes de communication entre ces éléments exécutables (RPC : Remote Procedure Call, DCOM : Distributed Component Object Model, IPC : InterProcess Communication...). Elle permet de décrire des informations liées à la distribution comme l'attribution des fonctionnalités du système sur les éléments exécutables, les types de communication entre ceux-ci et l'allocation des ressources physiques pour ceux-ci. Les aspects non fonctionnels comme la location, la migration, la réplication de ces instances exécutables sont aussi pris en compte.

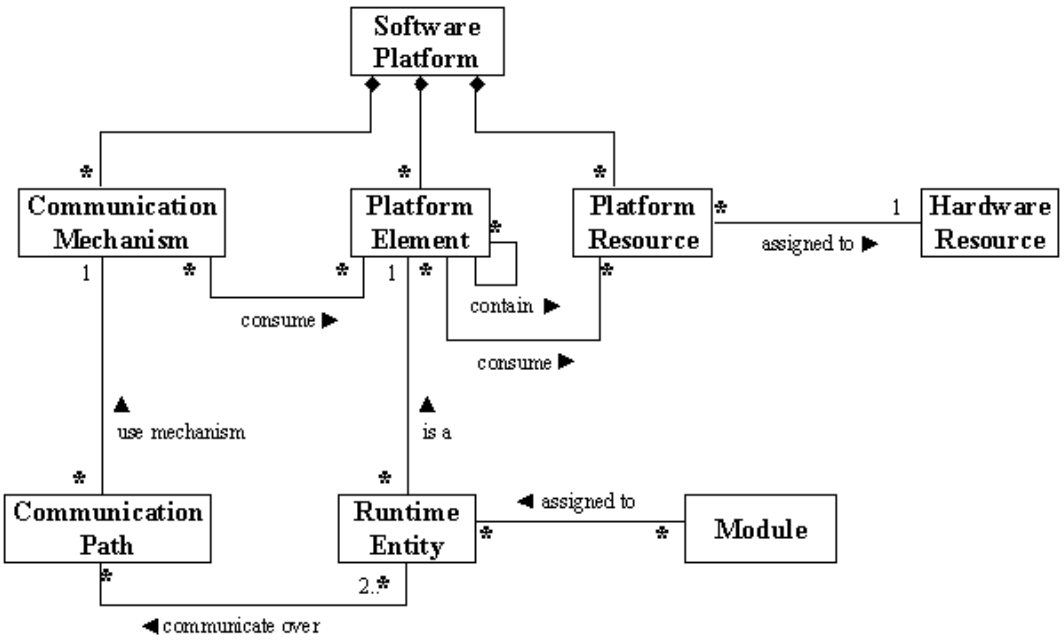


FIG. D.5 – Méta-modèle de la vue exécution

Annexe E

Notre conceptualisation de l'architecture V5

Notre conceptualisation de l'architecture V5 correspond à une instanciation de la norme ANSI/IEEE Std P1471-2000 [Gro00] qui préconise la modélisation des acteurs ainsi que de leurs besoins et la prise en compte de différentes structures architecturales (cf. section 4.1.7). La partie II de ce rapport de thèse présente de manière détaillée notre conceptualisation de l'architecture V5.

Cette annexe reprend notre modélisation de l'architecture V5 ainsi que la syntaxe graphique associée que nous avons définie et utilisée. La réalisation de cette syntaxe graphique a été basée sur la culture d'entreprise de Dassault Systèmes ce qui a permis de l'exploiter naturellement dès sa présentation. Sa conception devait répondre à un certain nombre de contrainte : faciliter la communication "humaine", simple à apprendre et à utiliser (80% des conceptions se documentent avec 20% des constructions), exploitable par des outils informatiques ou non (tableau, papier...)...

Bien que l'architecture V5 possède diverses structures architecturales, le temps qui nous était imparti nous a permis d'en étudier principalement trois : l'architecture logique, l'architecture physique et l'architecture produit ou de packaging (cf. figure E.1).

E.1 L'architecture logique

L'architecture logique se concentre sur la conception des fonctionnalités du logiciel. Cette structure se matérialise par une configuration de composants et de connecteurs fonctionnels. Dans le contexte de Dassault Systèmes, nous nous sommes particulièrement intéressés à l'Object Modeler car il correspond au modèle de composants pour l'architecture logique de Dassault Systèmes. C'est lui qui définit les bases de l'architecture logique.

L'Object Modeler a été principalement conçu pour répondre au besoin de Dassault Systèmes de posséder une architecture ouverte qui permet à ses clients

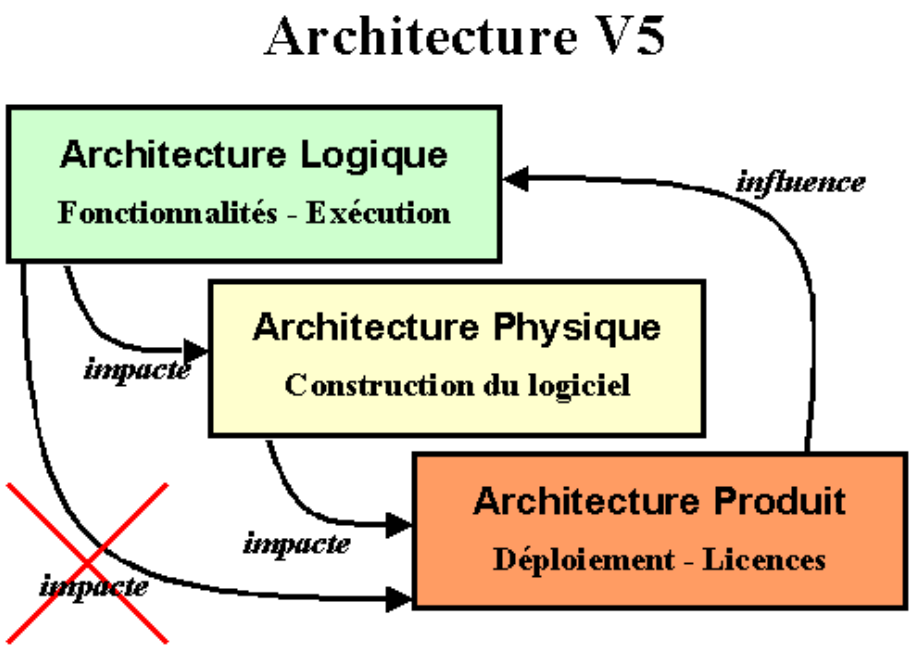


FIG. E.1 – Liens entre les architectures logique, physique et produit

d'étendre simplement CATIA V5 avec un minimum d'impact (cf. section 3.2). L'Object Modeler permet aussi de répondre à des besoins internes liés au développement fortement concurrent, à la facilité d'évolution et à des mécanismes liés à leur domaine d'activité. La figure E.2 présente le méta-modèle pour l'Object Modeler.

La syntaxe graphique associée est basée sur trois notations :

- La notation UML puisqu'il est le standard de la modélisation objet et est utilisée chez Dassault Systèmes,

- La notation de Microsoft pour représenter les composants COM car l'Object Modeler a des similitudes avec ce modèle de composants et que cette notation a l'avantage d'être pratique, connue et largement répandue,

- Les notations trouvées dans les documents de Dassault Systèmes car elles sont connues et utilisées par les ingénieurs de Dassault Systèmes et ses clients.

La figure E.3 récapitule la syntaxe graphique pour décrire les entités de l'Object Modeler.

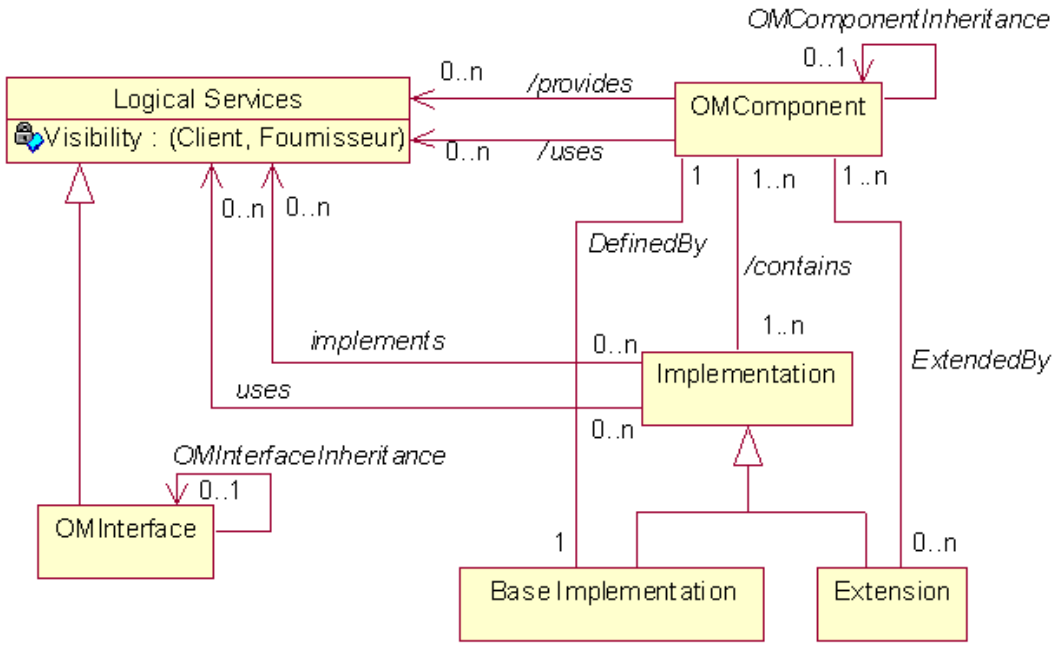


FIG. E.2 – Le méta-modèle de l'Object Modeler

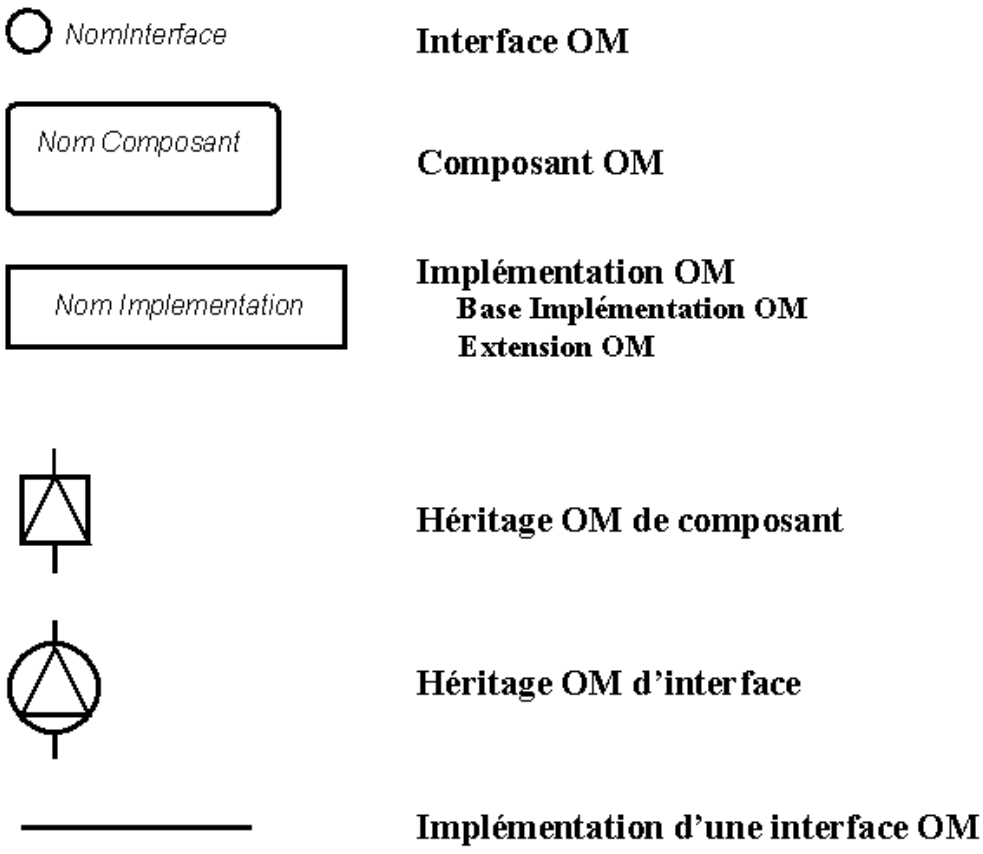


FIG. E.3 – Syntaxe graphique pour l'Object Modeler

E.2 L'architecture physique

L'architecture physique se concentre sur les besoins de Dassault Systèmes pour la construction de CATIA V5 (cf. section 3.1.2). Elle permet aussi de répondre à des besoins liés au développement concurrent en structurant l'implémentation du logiciel par rapport à l'organisation de développement. La figure E.4 présente le méta-modèle pour l'architecture physique.

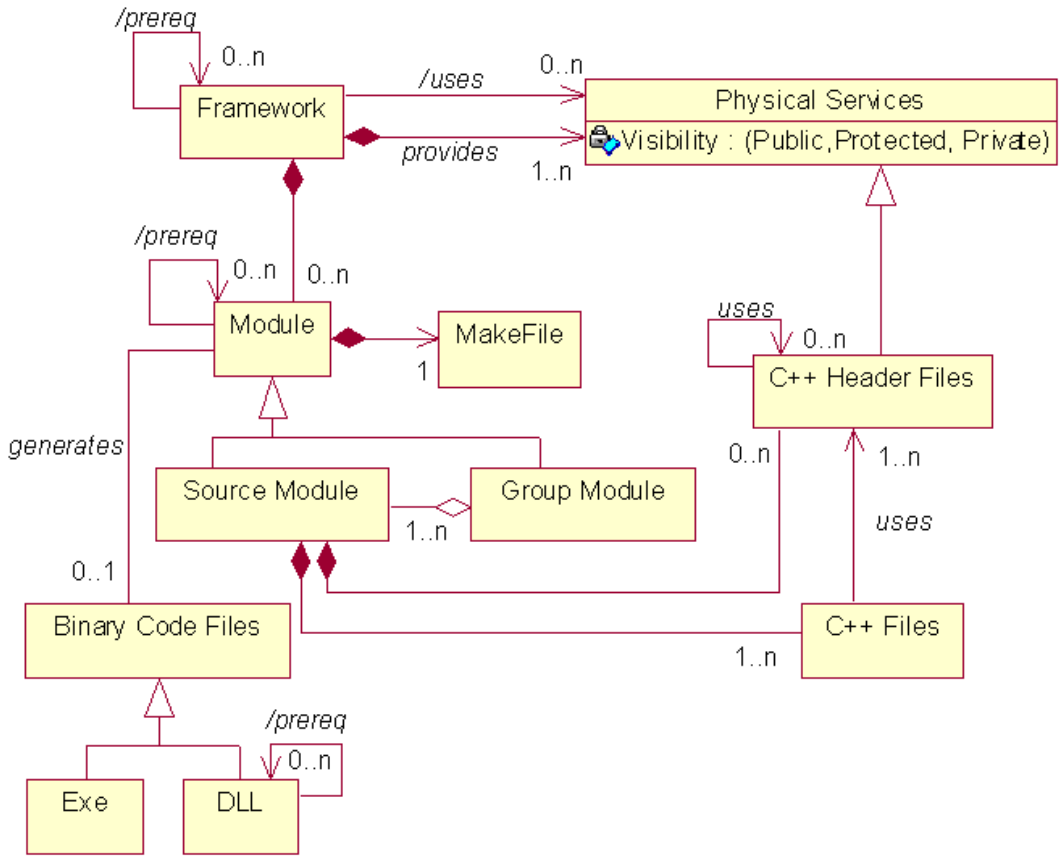


FIG. E.4 – Le méta-modèle de l'architecture physique

La syntaxe graphique associée à l'architecture physique est basée sur la notation UML et sur les notations trouvées dans les documents de Dassault Systèmes (cf. figure E.5).

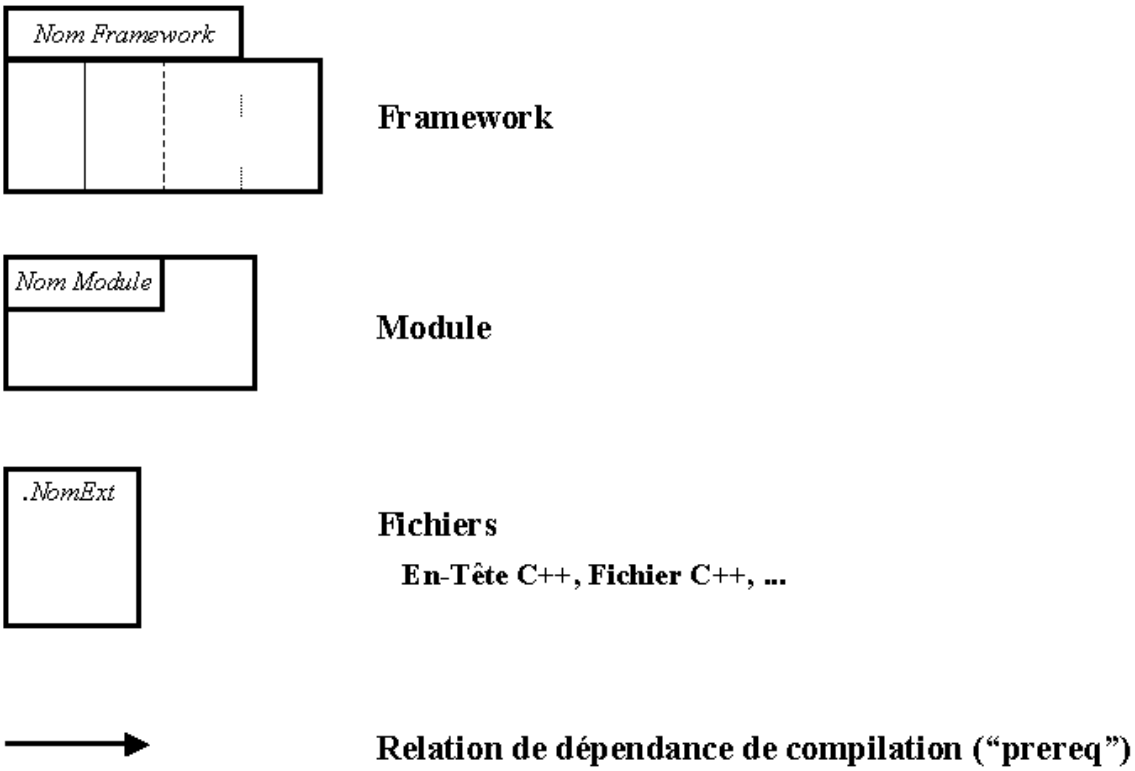


FIG. E.5 – Syntaxe graphique pour l'architecture physique

E.3 L'architecture produit ou de packaging

L'architecture produit se concentre sur les besoins de Dassault Systèmes pour la gestion des configurations des différentes applications (cf. section 3.2). Elle s'occupe en outre de la gestion des licences et permet de définir la stratégie commerciale de Dassault Systèmes. Cette structure architecturale est particulière à Dassault Systèmes et nous ne l'avons pas retrouvée dans d'autres travaux. Ceci s'explique par le fait que d'une part CATIA V5 n'est pas un logiciel à part entière mais plutôt une infrastructure qu'il est possible de spécialiser pour son domaine d'activité en ajoutant un certain nombre de produits orientés métier et d'autre part cette structure architecturale reflète des besoins économiques qui n'ont généralement pas été étudiés par les précédents travaux. La figure E.6 présente le méta-modèle pour l'architecture produit.

La syntaxe graphique associée à l'architecture produit a été inspirée par les notations trouvées dans les documents de Dassault Systèmes (cf. figure E.7).

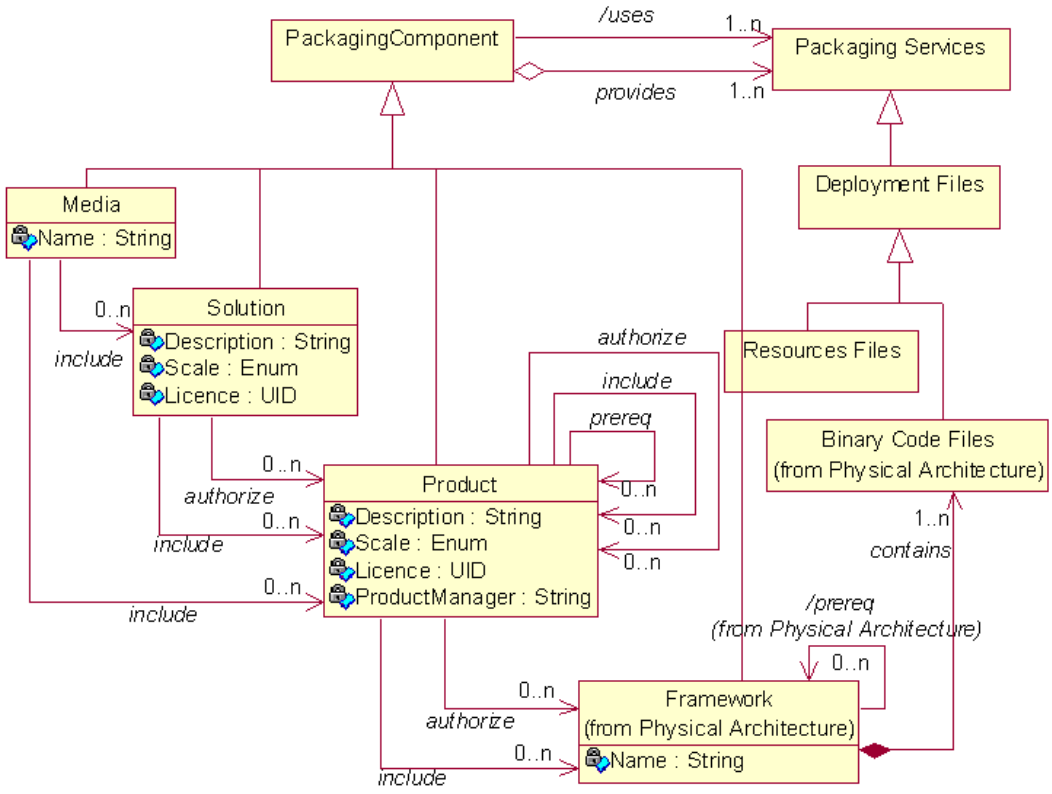


FIG. E.6 – Le méta-modèle de l'architecture produit

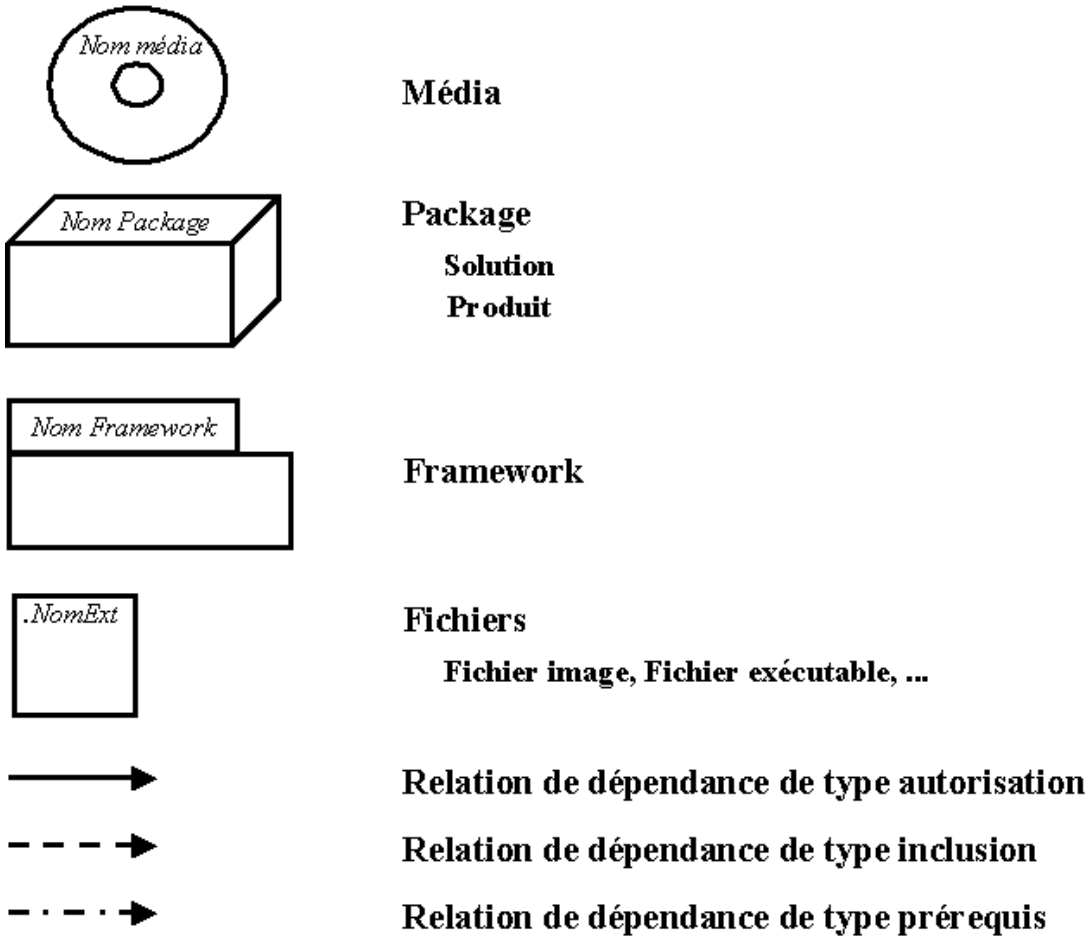


FIG. E.7 – Syntaxe graphique pour l'architecture produit

Acronymes

ADL	**A**rchitecture **D**escription **L**anguage
ADML	**A**rchitecture **D**escription **M**arkup **L**anguage
ANSI	**A**merican **N**ational **S**tandards **I**nstitute
API	**A**pplication **P**rogram **I**nterface
CAA	**C**omponent **A**pplications **A**rchitecture
CAO	**C**onception **A**ssistée par **O**rdinateur
CCM	**C**ORBA **C**omponent **M**odel (CORBA 3)
COM	**C**omponent **O**bject **M**odel
CORBA	**C**ommon **O**bject **R**equest **B**roker **A**rchitecture
DCOM	**D**istributed **C**omponent **O**bject **M**odel
DLL	**D**ynamic **L**ink **L**ibrary
EJB	**E**ntreprise **J**ava **B**eans
FAO	**F**abrication **A**ssistée par **O**rdinateur
HTML	**H**yper**T**ext **M**arkup **L**anguage
IDL	**I**nterface **D**efinition **L**anguage
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
IPC	**I**nter-**P**rocess **C**ommunication
MIL	**M**odule **I**nterconnection **L**anguage
OCL	**O**bject **C**onstraint **L**anguage
OLE	**O**bject **L**inking **E**mbedding
OMG	**O**bject **M**anagement **G**roup
OMT	**O**bject **M**odeling **T**echnique
PLM	**P**roduct **L**ifeCycle **M**anagement. Gestion du cycle de vie d'un produit.
RPC	**R**emote **P**rocedure **C**all
RUP	**R**ational **U**nified **P**rocess
SEI	**S**oftware **E**ngineering **I**nstitute
STEP	**ST**andard for the **E**xchange of **P**roduct model data (ISO 10303)
UML	**U**nified **M**odeling **L**anguage
XML	e**X**tensinble **M**arkup **L**anguage

Bibliographie

[AG94] Robert Allen and David Garlan. Beyond Definition/Use : Architectural Interconnection. In *Proceedings of the ACM Interface Definition Language Workshop*, volume 29(8), pages 35–45. ACM SIGPLAN Notices, August 1994.

[AG97] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3) :213–249, July 1997.

[Ang00] Rosana Angles. Outils pour l'exploration et la définition de l'architecture des logiciels. Stage ingénieur 3^e année, ENSIMAG Grenoble, Avril 2000.

[Bal99] Robert Balzer. Instrumenting, Monitoring, & Debugging Software Architectures, 1999. Available at `http://www.cs.wpi.edu/~cs562/s98/pdf/instrumenting-software-architectures.pdf`.

[BB89] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers North-Holland, 1989.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[Ben98] PerOlof Bengtsson. Towards Maintainability Metrics on Software Architecture : An Adaptation of Object-Oriented Metrics. In *Proceedings of the First Nordic Workshop on Software Architecture (NOSA'98)*, pages 20–21, Ronneby, Sweden, August 1998. Available at `http://www.ipd.bth.se/pob/archive/nosa98.pdf`.

[Ber81] G.D. Bergland. A Guide Tour of Program Design Methodologies. *IEEE Computer*, 14(16) :13–37, October 1981.

[BGMS96] G. Bernot, M-C. Gaudel, B. Marre, and F. Schlienger. *Précis de Génie Logiciel*. Masson, 1996.

[BL93] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by Multiset Transformation. *Communications of the ACM (CACM)*, 36(1) :98–111, January 1993.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture : A System of Patterns*, volume 1. Wiley, 1996.

[Boo91] Grady Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings, Redwood City, 1991.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Object Technology Series. Addison-Wesley, second edition, 1994.

[Bos00] Jan Bosch. *Design and Use of Software Architectures : Adopting and Evolving a Product-Line Approach.* Addison-Wesley, 2000.

[Box98] Don Box. *Essential COM.* Addison-Wesley, January 1998.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Object Technology Series. Addison-Wesley, Reading, Massachusetts, USA, 1999.

[Bro95] Frederick P. Brooks, Jr. *The Mythical Man-Month : Essays on Software Engineering, 20th Anniversary Edition.* Addison-Wesley, Reading, Mass., second edition, 1995.

[CAT] Dassault Systèmes. *CATIA.* `http://www.catia.com/`.

[Cer00] Humberto Cervantes. Analyse de dépendances et Découpe dans un Logiciel de Grande Taille. Rapport de DEA Informatique Systèmes et Communication, Université Joseph Fourier de Grenoble, June 2000.

[CK94] S. R. Chidamber and C. F. Kemerer. A Metric Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6) :476–493, June 1994.

[Cle96] P. Clements. A Survey of Architecture Description Languages. In *Proceedings 8th ACM/IEEE Int. Workshop on Software Specification and Design (IWSSD)*, pages 16–25, Germany, March 1996. IEEE Computer Society Press.

[COM] Microsoft. *COM Specification.* Available at `http://www.microsoft.com/com/resources/comdocs.asp`.

[Cor] Object Management Group. *Corba Home Page.* `http://www.corba.org/`.

[Cor01] Microsoft Corporation. *Microsoft C# Language Specifications.* Microsoft Press, 2001.

[CY91] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Yourdon Press, Englewood Cliffs, New Jersey, USA, second edition, 1991.

[DaV] University of Bremen. *DaVinci.* `http://www.informatik.uni-bremen.de/daVinci/`.

[DD99] Stéphane Ducasse and Serge Demeyer. *The FAMOOS Object Oriented Reengineering Handbook.* University of Bern, 1999. Available at `http://www.iam.unibe.ch/~famoos/handbook/`.

[DES01] Frédéric Duclos, Jacky Estublier, and Rémy Sanlaville. Architectures Ouvertes pour l'Adaptation des Logiciels. *Revue Génie Logiciel*, 58 :19–25, September 2001.

[Dij68] E.W. Dijkstra. The Structure of the T.H.E. Multiprogramming System. *Communications of the ACM*, 11(5) :341–346, May 1968.

[DK76] F. DeRemer and H. Kron. Programming in the Large vs Programming in the Small. *IEEE Transactions on Software Engineering*, 2(2) :80–86, June 1976.

[Don00] Patrick Donohoe, editor. *Software Product Lines : Experience and Research Directions, First Software Product Line Conference*, volume 576. Kluwer Academic Publishers, Boston, August 2000. `http://www.wkap.nl/book.htm/0-7923-7940-3`.

[Dot] AT&T Bell Labs. *Dotty.* `http://seclab.cs.ucdavis.edu/~hoagland/Dot.html`.

[Dre] Macromedia. *Dreamweaver.* `http://www.macromedia.com/fr/software/dreamweaver/`.

[DS] Dassault Systèmes. *Dassault Systèmes Home Page.* `http://www.3ds.com/`.

[DS00] Dassault Systèmes. *Rapport Annuel*, 2000.

[DW98] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML : The Catalysis Approach.* Addison-Wesley Object Technology Series. Addison-Wesley Publishing Company, 1998.

[EC94] Jacky Estublier and Rubby Casallas. The Adele Configuration Manager. In Walter Tichy, editor, *Configuration Management*, pages 99–133. John Wiley and Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, 1994.

[EFS02] Jacky Estublier, Jean-Marie Favre, and Rémy Sanlaville. An Industrial Experience with Dassault Systèmes' Component Model. In I. Crnkovic and M. Larsson, editors, *Builiding Reliable Component-Based Systems.* Archtech House, To appear, 2002.

[EHP+96] Walter J. Ellis, Rich Hilliard, Peter T. Poon, David Rayford, Thomas F. Saunders, Basil Sherlund, and Ronald L. Wade. Toward a Recommended Practice for Architectural Description. In *Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'96)*, Montreal, Quebec, Canada, October 1996.

[EJB] Sun. *Entreprise Java Beans.* `http://java.sun.com/products/ejb/`.

[EM99] Alexander Egyed and Nenad Medvidovic. Extending Architectural Representation in UML with View Integration. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 2–16. Springer, 1999.

[Eng97] Robert Englander. *Java Beans Guide du Programmeur.* O'Reilly, 1st edition, 1997.

[Fav01] Jean-Marie Favre. GSEE : a Generic Software Exploration Environment. In *9th International Workshop on Program Comprehension (IWPC'2001)*, pages 233–244, Toronto, Ontario, Canada, May 2001. IEEE Computer Society.

[Fav02] Jean-Marie Favre. A New Approach to Sofware Exploration : Backpacking with GSEE. In *Accepted to the 6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, Budapest, Hungary, March 2002. IEEE Computer Society.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring : Improving the Design of Existing Code.* Addison-Wesley, June 1999.

[FCD+01] Jean-Marie Favre, Humberto Cevantes, Frédéric Duclos, Rémy Sanlaville, and Jacky Estublier. Issues in Reengineering the Architecture of Component-Based Software. In *SWARM forum (Software Architecture Recovery and Modeling) at WCRE'2001 (Working Conference on Reverse Engineering)*, Stuttgart, Germany, October 2001.

[FDE+01] Jean-Marie Favre, Frédéric Duclos, Jacky Estublier, Rémy Sanlaville, and Jean-Jacques Auffret. Reverse Engineering a Large Component-based Software Product. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pages 95–104, Lisbon, Portugal, March 2001.

[FHK+97] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The Software Bookshelf. *IBM Systems Journal*, 36(4) :564–593, November 1997.

[Fin00] Anthony Finkelstein, editor. *The Future of Software Engineering.* 22nd International Conference on Software Engineering (ICSE'2000), IEEE Computer Society Press / ACM Press, June 2000.

[FP98] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics : A Rigorous and Practical Approach.* PWS Publishers, second edition, 1998.

[GACB95] Cristina Gacek, Ahmed Abd-Allah, Bradford Clark, and Barry Boehm. On the Definition of Software System Architecture. In D. Garlan, editor, *Proceedings of the First International Workshop on Architectures for Software Systems - In Cooperation with the 17th International Conference on Software Engineering*, pages 85–95, Seattle, WA, April 1995.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT'94 : The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 175–188. ACM Press, December 1994.

[Gar00] D. Garlan. Software Architecture : a Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering, 22nd International Conference on Software Engineering (ICSE'2000)*, pages 93–101, Limerick, Ireland, June 2000. IEEE Computer Society Press / ACM Press.

[Gau94] Marie-Claude Gaudel. Formal Specification Techniques. In Bruno Fadini, editor, *Proceedings of the 16th International Conference on Software Engineering*, pages 223–232, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[GEF] University of California. *GEF : Graph Editing Framework.* `http://www.ics.uci.edu/pub/arch/gef/`.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GK00] David Garlan and Andrew J. Kompanek. Reconciling the Needs of Architectural description with Object-Modeling Notations. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Proceedings of the Third International Conference on the Unified Modeling Language*, volume 1939 of *LNCS*, pages 498–512, York, UK, October 2000. Springer.

[GL86] J. Guttag and B. Liskov. *Abstraction and Specification In Program Development.* The MIT Press/Mc Graw-Hill, 1986.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. Acme : An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. Acme : Architectural Description of Component-Based Systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 3, pages 47–67. Cambridge University Press, New York, NY, 2000.

[GP95] David Garlan and Dewayne Perry. Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[Gro00] IEEE Architecture Working Group. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems.* IEEE Std 1471-2000. Institute of Electrical and Electronics Engineers, New York, NY, USA, October 2000.

[GrP] *GraphPanel.* `http://binger.centre.edu/GraphPanel`.

[GS93] David Garlan and Mary Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–40. World Scientific Publishing Company, 1993.

[GS00] David Garlan and Joãn Pedro Sousa. Documenting Software Architectures : Recommendations for Industrial Practice. Technical Report CMU-CS-00-169, Carnegie Mellon University, Pittsburgh, Pennsylvania, October 2000.

[Har87] David Harel. Statecharts : A Visual Formulation for Complex Systems. *Science of Computer Programming*, 8(3) :231–274, June 1987.

[HL97] Maritta Heisel and Nicole Lévy. Using LOTOS Patterns to Characterize Architectural Styles. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97 : Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 818–832. Springer-Verlag, 1997.

[HNS99] C. Hofmeister, R. Nord, and D. Soni. Describing Software Architecture with UML. In P. Donohoe, editor, *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, pages 145–160, San Antonio, Texas, February 1999. Kluwer Academic.

[HNS00] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley, 2000.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8) :666–677, August 1978.

[HR92] Susan Horwitz and Thomas Reps. The Use of Program Dependence Graphs in Software Engineering. In *Proceedings of the 14th International Conference on Software Engineering*, pages 392–411, May 1992.

[Ima] Imagix Corporation. *Imagix*. `http://www.imagix.com/`.

[Ins] Parasoft. *Insure++*. `http://www.parasoft.com/products/insure/index.htm`.

[IW95] Paola Inverardi and Alexander L. Wolf. Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[Jam00] Sonia Jamal. Visualisation de la structure du logiciel CATIA sous-forme de graphes. Rapport de DESS Compétence Complémentaire en informatique, Université Joseph Fourier de Grenoble, June 2000.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Object Technology series. Addison-Wesley, Reading, Massachusetts, USA, 1999.

[JBu] Inprise-Borland. *JBuilder Software*. `http://www.inprise.com/jbuilder/`.

[JCJO92] I. Jacobson, M. Christerson, M. Jonsson, and P. van Övergaard. *OO Software Engineering, A Use Case Driven Approach*. Addison-Wesley, Reading, Massachusets, 1992.

[JLNP96] Paul Jacquet, Yves Ledru, Xavier Nicollin, and Marie-Laure Potet. Catalogue de l'exposition : Logiciels Critiques. Médiathèque de l'IMAG, Grenoble, January 1996.

[Joh92] Ralph E. Johnson. Documenting Frameworks using Patterns. In *Proceedings of the 7th Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA'92)*, pages 63–76, October 1992. Published as ACM SIGPLAN Notices, volume 27, number 10.

[Jon94] A.K. Jones. The Maturing of Software Architecture. Software Engineering Symposium, Software Engineering, Institute Pittsburgh, Pa, August 1994.

[JRL00] Mehdi Jazayeri, Alexander Ran, and Phillip van der Linden. *Software Architecture For Product Families : Putting Research into Practice*. Addison-Wesley, 2000.

[KABC96] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. Scenario-Based Analysis of Software Architecture. *IEEE Software*, 13(6) :47–55, November 1996.

[KBAW94] Rick Kazman, Len Bass, Gregory Abowd, and Mike Webb. SAAM : A Method for Analyzing the Properties of Software Architectures. In

Proceedings of the 16th International Conference on Software Engineering (ICSE'94), pages 81–90, Sorrento, Italy, May 1994. IEEE Computer Society Press / ACM Press.

[KBK+99] Rick Kazman, Mario Barbacci, Mark Klein, S. Jeromy Carrière, and Steven G. Woods. Experience with Performing Architecture Tradeoff Analysis. In *Proceedings of the 21th International Conference on Software Engineering (ICSE'99)*, pages 54–64, New York, May 1999. IEEE Computer Society Press / ACM Press.

[KKB+98] Rick Kazman, Mark Klein, Mario Barbacci, Tom Longsta, Howard Lipson, and Jeromy Carrière. The Architecture Tradeoff Analysis Method. In *Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'98)*. IEEE Computer Society Press, August 1998.

[Kru95] Philippe B. Kruchten. The 4 + 1 view model of architecture. *IEEE Software*, 12(6) :42–50, November 1995.

[KSK+01] Philippe Kruchten, Bran Selic, Wojtek Kozaczynski, Grant Larsen, and Alan Brown. Workshop 16 : Describing Software Architecture with UML. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'2001)*, page 777, Toronto, Canada, May 2001. IEEE Computer Society Press / ACM Press.

[Lam00] Axel van Lamsweerde. Formal Specification : a Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering, 22nd International Conference on Software Engineering (ICSE'2000)*, pages 147–160, Limerick, Ireland, June 2000. IEEE Computer Society Press / ACM Press.

[LKA+95] David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4) :336–355, April 1995.

[Log] Telelogic. *Logiscope Tools.* `http://www.telelogic.com/products/logiscope/`.

[LSE00] Yves Ledru, Rémy Sanlaville, and Jacky Estublier. Defining an Architecture Description Language for Dassault Systèmes. In B. Balzer and Henk Obbink, editors, *Proceedings of the 4th International Software Architecture Workshop (ISAW4)*, pages 115–120, Limerick, Ireland, June 2000.

[LV95] David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9) :717–734, September 1995.

[McC] MCCabe & Associates. *McCabe Tools.* `http://www.mccabe.com/main.htm`.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* The Object-Oriented Series. Prentice-Hall, Englewood Cliffs, New Jersey, USA, second edition, 1997.

[MH96] Spiros Mancoridis and Richard C. Holt. Recovering the Structure of Software Systems Using Tube Interconnection Clustering. In *Proceedings of International Conference on Software Maintenance (ICSM'96)*, pages 23–32, Monterey, CA, November 1996. IEEE Computer Society Press.

[MMP00] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a Taxonomy of Software Connectors. In *Proceedings of the 22th International Conference on Software Engineering (ICSE'2000)*, pages 178–187, Limerick, Ireland, June 2000. IEEE Computer Society Press / ACM Press.

[Mon00] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 2nd edition, 2000.

[Mor87] C. R. Morgan. Introduction to the Special Issue on the Interface Description Language IDL. *SIGPLAN Notices*, 22(11) :1–3, November 1987.

[MOT+00] Nenad Medvidovic, Peyman Oreizy, Richard N. Taylor, Rohit Khare, and Michael Guntersdorfer. An Architecture-Centered Approach to Software Environment Integration. Technical Report UCI-ICS-00-11, Department of Information and Computer Science, University of California, Irvine, March 2000. Available at `ftp://www.ics.uci.edu/pub/arch/papers/TR-UCI-ICS-00-11.pdf`.

[MR99] Nenad Medvidovic and David S. Rosenblum. Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. In P. Donohoe, editor, *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, pages 161–182, San Antonio, Texas, February 1999. Kluwer Academic.

[MRT99] Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 44–53. IEEE Computer Society Press / ACM Press, 1999.

[MSN] Microsoft. *Framework .NET*. `http://msdn.microsoft.com/net/`.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1) :70–93, January 2000.

[MTC97] R. Marlet, S. Thibault, and C. Consel. Mapping Software Architectures to Efficient Implementations via Partial Evaluation. In *Proceedings of the 12th International Conference on Automated Software Engineering (ASE'97)*, pages 183–192, Nevada, USA, November 1997. IEEE Computer Society Press.

[MTO+92] Hausi A. Müller, S. R. Tilley, M. A. Orgun, B. D. Corrie, and N. H. Madhavji. A Reverse Engineering Environment Based on Spatial and Visual Software Interconnection Models. In *Proceedings of the Fifth ACM SIGSOFT International Symposium on Software Development Environment (SIGSOFT'92)*, volume 17(5) of *ACM Software Engineering Notes*, pages 88–98, December 1992.

[Ngu00] Si Triet Nguyen. Environnements d'Exploration de Grands Logiciels. Rapport de DEA Informatique Systèmes et Communication, Université Joseph Fourier de Grenoble, September 2000.

[NKM96] Keng Ng, Jeff Kramer, and Jeff Magee. A CASE Tool for Software Architecture Design. *Journal of Automated Software Engineering (JASE)*, 3(3/4) :261–284, August 1996.

[NR69] P. Naur and B. Randell, editors. *Software Engineering*, NATO Scientific Affairs Division, January 1969.

[NS87] J.R. Nestor and D.L. Stone. IDL : Background and Status, Special Issue on the Interface Description Language IDL. *ACM SIGPLAN Notices*, 22(11) :5–9, November 1987.

[ODE] Object Design. *Object Store.* `http://www.objectdesign.com/htm/object_prod.asp`.

[OMG] Object Management Group. *OMG Home Page.* `http://www.omg.org/`.

[Par72] David Lorge Parnas. On the Criteria for Decomposing Systems into Modules. *Communication of the ACM*, 15(12) :1053–1058, December 1972.

[Par94] David Lorge Parnas. Software Aging. In *Proceedings of 16th International Conference on Software Engineering (ICSE'94)*, pages 279–287, Sorrento, Italy, May 1994. IEEE Computer Society Press / ACM Press.

[PBS] Holt Group, University of Toronto. *PBS Home Page.* `http://swag.uwaterloo.ca/pbs/`.

[PDN86] R. Prieto-Diaz and J. Neighbors. Module Interconnection Languages. *Journal of Systems and Software*, 6(4) :307–334, November 1986.

[Pra86] V. Pratt. Modelling Concurrency with Partial Orders. *International Journal of Parallel Programming*, 15(1) :33–71, February 1986.

[PSSC99] Steve Pruitt, Doug Stuart, Wonhee Sull, and T.W. Cook. The Merit of XML as an Architecture Description Language Meta-Language. In *Working IFIP Conference on Software Architecture (WICSA 1)*, February 1999.

[Pur] Rational. *Purify.* `http://www.rational.com/products/purify_nt/index.jsp`.

[PW92] D.E. Perry and A.L. Wolf. Foundations for the Study of Software Architecture. *Software Engineering Notes, ACM SIGSOFT*, 17(4) :40–52, October 1992.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, New Jersey, USA, 1991.

[RG92] K. S. Rubin and A. Goldberg. Object-Behavior Analysis. *Communications of the ACM*, 35(9) :48–62, September 1992.

[Rig] University of Victoria. *Rigi Home Page.* `http://www.rigi.csc.uvic.ca/`.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Object Technology Series. Addison Wesley Longman, Reading, Massachusetts, USA, 1999.

[RKES77] Douglas T. Ross and Jr. Kenneth E. Schoman. Structured Analysis for Requirements Definition. *IEEE Transactions on Software Engineering*, 3(1) :6–15, January 1977. Special collection on Requirement Analysis.

[RMRR98] Jason E. Robbins, Nenad Medvidovic, David F. Redmiles, and David S. Rosenblum. Integrating Architecture Description Languages with a Standard Design Method. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 209–218. IEEE Computer Society Press / ACM Press, 1998.

[Ros] Rational Software Corporation. *Rational Rose.* `http://www.rational.com/products/rose/index.jsp`.

[Rui] Diego Sevilla Ruiz. The CORBA & CORBA Component Model (CCM) Page. `http://ditec.um.es/~dsevilla/ccm/`.

[RW80] W.E. Riddle and J.C. Wileden. *Tutorial on Software System Design : Description and Analysis.* Computer Society Press, 1980.

[San] Georg Sander. Graph Drawing Tools and Related Work. `http://rw4.cs.uni-sb.de/users/sander/html/gstools.html`.

[San97] Rémy Sanlaville. Description d'architectures logicielles : Utilisation du formalisme WRIGHT pour l'interconnexion de machines abstraites B. Rapport de DEA Informatique Systèmes et Communication, Université Joseph Fourier de Grenoble, June 1997.

[San00] Rémy Sanlaville. Software Architecture : An Industrial Experiment with Dassault Systèmes. In Renaud Marlet and John Penix, editors, *Actes du Symposium Doctoral ASE'2000. Technical Report number PI-1353*, pages 25–30. Institut de Recherche en Informatique et Systemes Aleatoires (IRISA), November 2000. Available at http ://www.irisa.fr/bibli/publi/pi/2000/1353/1353.html.

[SEI] Software Engineering Institute. *How Do You Define Software Architecture ?* `http://www.sei.cmu.edu/architecture/definitions.html`.

[SFL01] Rémy Sanlaville, Jean-Marie Favre, and Yves Ledru. Helping Various Stakeholders to Understand a very large Component-Based Software. In *Proceedings of the 27th EUROMICRO Conference*, pages 104–111, Warsaw, Poland, September 2001.

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

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling.* Wiley, April 1994.

[SLEFew] Rémy Sanlaville, Yves Ledru, Jacky Estublier, and Jean-Marie Favre. Architectural Environment for the Evolution of Complex Software. In *Chapter proposal for the book 4-Volume books on Software Architectures,*

PLAs, Components and Enterprise Frameworks. Mohamed Fayad, David Garlan, and Wolfgang Pree, editors, In review. `http://www.cse.unl.edu/~fayad/Books/NewBooks/`.

[Sma] Information Laboratory. *Small Worlds.* `http://www.thesmallworlds.com/`.

[Som92] I. Sommerville. *Software Engineering.* Addison-Wesley, 4th edition, 1992.

[SRW98] Judith A. Stafford, Debra J. Richardson, and Alexander L. Wolf. Aladdin : A Tool for Architecture-Level Dependence Analysis of Software Systems. Technical Report CU-CS-858-98, University of Colorado, Boulder, Colorado, April 1998.

[SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture : Patterns for Concurrent and Networked Objects*, volume 2. Wiley, 2000.

[Sta98] Judith A. Stafford. Dependence Analysis for Software Architectures. In Steve Easterbrook, editor, *Actes du Symposium Doctoral ASE'1998.*, pages 43–46, Honolulu, Hawaii, October 1998. Available at `http://ase.informatik.uni-essen.de/ase/past/ase98/ASE98DocSymProc.pdf`.

[Sta00] Judith A. Stafford. A Formal, Language-Independent, and Compositional Approach to Control Dependence Analysis. Ph.D. Thesis, University of Colorado, Boulder, Colorado, July 2000. Available at `ftp://ftp.sei.cmu.edu/pub/jas/Stafford-thesis.zip`.

[Sun] Sun. *Java Beans.* `http://java.sun.com/products/javabeans/`.

[SW98] Judith A. Stafford and Alexander L. Wolf. Architecture-Level Dependence Analysis in Support of Software Maintenance. In D. Perry and J. Magee, editors, *Proceedings of the 3rd International Workshop on Software Architecture (ISAW-3)*, ACM SIGSOFT Software Engineering Notes, pages 129–132, Orlando, FL, November 1998.

[Tic92] W. F. Tichy. Programming-in-the-large : Past, Present, and Future. In *Proceedings of the 14th International Conference on Software Engineering (ICSE'92)*, pages 362–367. IEEE Computer Society Press / ACM Press, May 1992.

[TMA+96] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. A Component and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22(6) :390–406, June 1996.

[TOG] The Open Group. *The Open Group Architectural Framework (TOGAF) Version 7 (in review).* `http://www.opengroup.org/public/togaf/v7/cr_invite.htm`.

[UML97] Object Modeling Group. *Unified Modeling Language version 1.0*, January 1997.

[Ver01] Frédéric Vernier. La multimodalité en sortie et son application à la visualisation de grandes quantités d'information. Thèse, Univeristé Joseph Fourier, Grenoble, France, February 2001. Available at `http://iihm.imag.fr/publs/2001/These_Vernier.pdf`.

[Ves93] Steve Vestal. A Cursory Overview and Comparison of Four Architecture Description Languages. Technical report, Honeywell Technology Center, Minneapolis MN, 1993. Available at `http://www.htc.honeywell.com/projects/dssa/ftp/papers/four_adl.ps`.

[Ves98] Steve Vestal. *MetaH User's Manual.* Honeywell Technology Center, Minneapolis MN, 1998. Version 1.27. Available at `http://www.htc.honeywell.com/metah/uguide.pdf`.

[VGJ] University of Auburn. *VGJ : Visualizing Graphs with Java.* `http://www.eng.auburn.edu/department/cse/research/graph_drawing/graph_drawing.html`.

[Vis] Micorsoft. *Visual C++.* `http://msdn.microsoft.com/visualc/`.

[Wie94] L.R. Wiener. *Les avatars du logiciel : leçons à tirer de quelques fiascos informatiques.* Addison-Wesley, 1994.

[Wil99] D. Wile. AML : An Architecture Meta-Language. In Robert J. Hall and Ernst Tyugu, editors, *14th IEEE International Conference on Automated Software Engineering (ASE'99)*, pages 183–190. IEEE Computer Society Press, 1999. Available at `http://www.cs.vassar.edu/faculty/welty/ase/99Wile.pdf`.

[WK98] Jos Warmer and Anneke Kleppe. *The Object Constraint Language : Precise Modeling with UML.* Addison-Wesley, 1998.

[WK99] Jos Warmer and Anneke Kleppe. OCL : The Constraint Language of the UML. *Journal of Object-Oriented Programming*, 12(1) :10–13,28, May 1999.

[XML00] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0*, second edition, October 2000. Available at `http://www.w3.org/TR/2000/REC-xml-20001006`.

[YC79] Edward Yourdon and Larry L. Constantine. *Structured Design : Fundamentals of a Discipline of Computer Program and Systems Design.* Yourdon Press, New York, 1979.

[Zha98] J. Zhao. Applying Slicing Technique to Software Architectures. In *Proceedings of the 4th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'98)*, pages 87–98, Monterey, California, USA, August 1998. IEEE Computer Society Press.

[Zha01] J. Zhao. Metrics for Software Architectures, January 2001. Available at `http://www.fit.ac.jp/~zhao/pub/ps/metrics.ps.gz`.