Architecture logicielle

L'**architecture logicielle** décrit d'une manière symbolique et schématique les différents éléments d'un ou de plusieurs systèmes informatiques, leurs interrelations et leurs interactions. Contrairement aux spécifications produites par l'analyse fonctionnelle, le modèle d'architecture, produit lors de la phase de conception, ne décrit pas ce que doit réaliser un système informatique mais plutôt comment il doit être conçu de manière à répondre aux spécifications. L'analyse décrit le « quoi faire » alors que l'architecture décrit le « comment le faire ».

1 Contexte et motivation

La phase de conception logicielle est l'équivalent, en informatique, à la phase de conception en ingénierie traditionnelle (mécanique, civile ou électrique) ; cette phase consiste à réaliser entièrement le produit sous une forme abstraite avant la production effective. Par contre, la nature immatérielle du logiciel (modelé dans l'information et non dans la matière), rend la frontière entre l'architecture et le produit beaucoup plus floue que dans l'ingénierie traditionnelle. L'utilisation d'outils CASE (Computer-aided software engineering) ou bien la production de l'architecture à partir du code lui-même et de la documentation système permettent de mettre en évidence le lien étroit entre l'architecture et le produit.

L'architecture logicielle constitue le plus gros livrable d'un processus logiciel après le produit (le logiciel lui-même). En effet, la phase de conception devrait consommer autour de 40 %[1] de l'effort total de développement et devrait être supérieure ou égale, en effort, à la phase de codage mais il peut être moindre. L'effort dépend grandement du type de logiciel développé, de l'expertise de l'équipe de développement, du taux de réutilisation et du processus logiciel.

Les deux objectifs principaux de toute architecture logicielle sont la réduction des coûts et l'augmentation de la qualité du logiciel ; la réduction des coûts est principalement réalisée par la réutilisation de composants logiciels et par la diminution du temps de maintenance (correction d'erreurs et adaptation du logiciel). La qualité, par contre, se trouve distribuée à travers plusieurs critères ; la norme ISO 9126 est un exemple d'un tel ensemble de critères.

1.1 Critères de qualité logicielle

Article détaillé : qualité logicielle.

L'interopérabilité extrinsèque exprime la capacité du logiciel à communiquer et à utiliser les ressources d'autres logiciels comme, par exemple, les documents créés par une certaine application.

L'interopérabilité intrinsèque exprime le degré de cohérence entre le fonctionnement des commandes et des modules à l'intérieur d'un système ou d'un logiciel.

La portabilité exprime la possibilité de compiler le code source et/ou d'exécuter le logiciel sur des plates-formes (machines, systèmes d'exploitation, environnements) différentes.

La compatibilité exprime la possibilité, pour un logiciel, de fonctionner correctement dans un environnement ancien (compatibilité descendante) ou plus récent (compatibilité ascendante).

La validité exprime la conformité des fonctionnalités du logiciel avec celles décrites dans le cahier des charges.

La vérifiabilité exprime la simplicité de vérification de la validité.

L'intégrité exprime la faculté du logiciel à protéger ses fonctions et ses données d'accès non autorisés.

La fiabilité exprime la faculté du logiciel à gérer correctement ses propres erreurs de fonctionnement en cours d'exécution.

La maintenabilité exprime la simplicité de correction et de modification du logiciel, et même, parfois, la possibilité de modification du logiciel en cours d'exécution.

La réutilisabilité exprime la capacité de concevoir le logiciel avec des composants déjà conçus tout en permettant la réutilisation simple de ses propres composants pour le développement d'autres logiciels.

L'extensibilité exprime la possibilité d'étendre simplement les fonctionnalités d'un logiciel sans compromettre son intégrité et sa fiabilité.

L'efficacité exprime la capacité du logiciel à exploiter au mieux les ressources offertes par la ou les machines où le logiciel sera implanté.

L'autonomie exprime la capacité de contrôle de son exécution, de ses données et de ses communications.

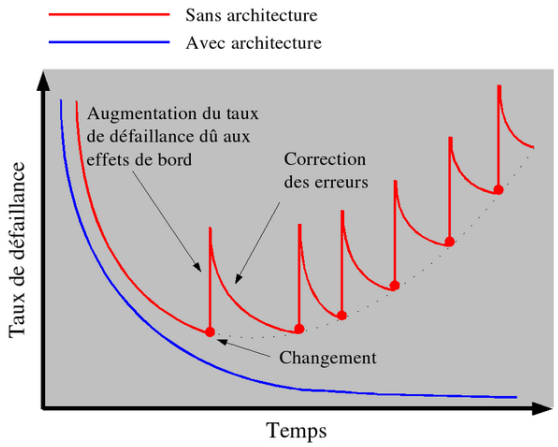
La transparence exprime la capacité pour un logiciel de

masquer à l'utilisateur (humain ou machine) les détails inutiles à l'utilisation de ses fonctionnalités.

La composabilité exprime la capacité pour un logiciel de combiner des informations provenant de sources différentes.

La convivialité décrit la facilité d'apprentissage et d'utilisation du logiciel par les usagers.

1.2 Diminution de la dégradation du logiciel



Graphique de dégradation du logiciel en fonction de l'architecture. *Inspiré de : Pressman R. S., Software Engineering : A Practitioner's Approach, Fifth Edition. McGraw-Hill. Chapitre 1, p. 8, 2001.*

Une architecture faible ou absente peut entraîner de graves problèmes lors de la maintenance du logiciel. En effet, toute modification d'un logiciel mal architecturé peut déstabiliser la structure de celui-ci et entraîner, à la longue, une dégradation (principe d'entropie du logiciel). L'architecte informatique devrait donc concevoir, systématiquement, une architecture maintenable et extensible.

Dans les processus itératifs comme UP (Unified Process), la gestion des changements est primordiale. En effet, il est implicitement considéré que les besoins des utilisateurs du système peuvent changer et que l'environnement du système peut changer. L'architecte informatique a donc la responsabilité de prévoir le pire et de concevoir l'architecture en conséquence ; la plus maintenable possible et la plus extensible possible.

Bien des logiciels ont été créés sans architecture par plusieurs générations de développeurs ayant chacune usé d'une imagination débordante pour réussir à maintenir l'intégrité du système. Une telle absence d'architecture peut être qualifiée *d'architecture organique*[Note 1]. En effet, un développeur confronté à une telle architecture a plus l'impression de travailler avec un organisme vivant qu'avec un produit industriel. Il en résulte que la complexité du logiciel fait en sorte que celui-ci est extrêmement difficile à comprendre et à modifier. À la limite, modifier une partie du système est plus proche, en complexité, de la transplantation cardiaque que du changement de carburateur.

1.3 Développement pour et par la réutilisation

logo universel du recyclage

La réutilisation de composants logiciels est l'activité permettant de réaliser les économies les plus substantielles[2], encore faut-il posséder des composants à réutiliser. De plus, la réutilisation de composants nécessite de créer une architecture logicielle permettant une intégration harmonieuse de ces composants[3]. Le développement par la réutilisation logicielle impose donc un cycle de production-réutilisation perpétuel et une architecture logicielle normalisée.

Une réutilisation bien orchestrée nécessite la création et le maintien d'une bibliothèque logicielle et un changement de focus ; créer une application revient à créer les composants de bibliothèque nécessaires puis à construire l'application à l'aide de ces composants. Une telle bibliothèque, facilitant le développement d'application est un framework (cadriciel) d'entreprise et son architecture, ainsi que sa documentation sont les pierres angulaires de la réutilisation logicielle en entreprise.

Le rôle de l'architecte informatique se déplace donc vers celui de bibliothécaire. L'architecte informatique doit explorer la bibliothèque pour trouver les composants logiciels appropriés puis créer les composants manquants, les documenter et les intégrer à la bibliothèque. Dans une grande entreprise, ce rôle de bibliothécaire est rempli par l'architecte informatique en chef qui est responsable du développement harmonieux de la bibliothèque et de la conservation de l'intégrité de son architecture.

2 L'architecture une question de point de vue

La description d'un système complexe comme un logiciel informatique peut être faite selon plusieurs points de vue différents mais chacun obéit à la formule de Perry et Wolf[4] : **Architecture = Elements + Formes + Motivations**. Selon le niveau de granularité, les éléments peuvent varier en tailles (lignes de code, procédures ou fonctions, modules ou classes, paquetages ou couches, applications ou systèmes informatiques), ils peuvent varier en raffinement (ébauche, solution à améliorer ou solution finale) et en abstraction (idées ou concepts, classes ou objets, composants logiciels). Les éléments peuvent également posséder une *temporalité* (une existence limitée dans le temps) et une *localisation* (une existence limitée dans l'espace).

Si les éléments sont, en général, représentés par des rectangles ou des ovales, les formes sont quant à elles constituées, la plupart du temps, d'éléments reliés par des droites ou des flèches. La sémantique des liens détermine la majeure partie de la sémantique du diagramme et l'aspect du système qui y est décrit.

2.1 Principaux types de liens

La dépendance fonctionnelle, signifie que l'élément source nécessite l'élément de destination pour réaliser ses fonctionnalités.

Le flot de contrôle, signifie que l'élément de destination prendra le contrôle de l'exécution après la terminaison de l'élément source.

La transition d'état, signifie que le système passera de l'état source à l'état de destination.

Le changement d'activité, signifie que le système réalisera l'activité de destination après l'activité source.

Le flot de données, signifie que l'information s'écoule de l'élément source vers l'élément de destination.

Le lien de communication, signifie que deux éléments échangent de l'information.

La composition, signifie que l'élément source est composé d'une ou de plusieurs données du type de l'élément de destination.

L'héritage (généralisation), signifie que l'élément source possède l'ensemble des données et des comportements de l'élément de destination.

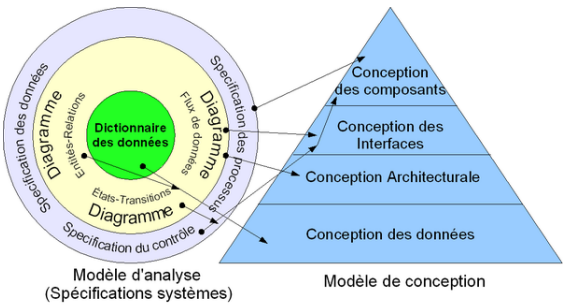
L'envoi de message, signifie que l'élément source envoie un message à l'élément de destination.

2.2 Les modèles d'architecture

Indépendamment de la forme que prend un diagramme d'architecture, celui-ci ne représente toujours qu'un point de vue sur le système considéré, le plus important étant les motivations. En effet, à quoi sert de produire un diagramme s'il est inutile (pas utilisé) ou si les raisons des choix architecturaux sont vagues et non-explicités. Pour éviter de formuler les motivations pour chaque diagramme, l'architecte informatique produira les différents diagrammes en fonction d'un *modèle de conception* et réutilisera des patrons de conception éprouvés.

Un modèle de conception (ou d'architecture) est composé d'un ensemble de points de vue, chacun étant composé d'un ensemble de différentes sortes de diagrammes. Il propose également des moyens pour lier les différentes vues et diagrammes les uns aux autres de manière à naviguer aisément, il s'agit des mécanismes de traçabilité architecturale. La traçabilité doit également s'étendre aux spécifications systèmes et même jusqu'aux besoins que ces spécifications comblent. La devise des trois pères fondateurs d'UML est « *Centré sur l'architecture, piloté par les cas d'utilisation et au développement itératif et incrémentiel* ». Cette devise indique clairement qu'aucune décision architecturale ne doit être prise sans que celle-ci ne soit dirigée (pilotée) par la satisfaction des spécifications systèmes (cas d'utilisation).

2.2.1 Le modèle conventionnel



Modèles d'analyse et d'architecture centrées sur les données avec traçabilité. *D'après : Pressman R. S., Software Engineering : A Practitioner's Approach, Fifth Edition. McGraw-Hill. Chapitre 13, p. 337, 2001.*

Ce diagramme décrit, à gauche, les spécifications systèmes qui sont également représentées par des diagrammes (Entités-Relations, Flux de données, États-Transitions). Et à droite, nous avons les différentes activités de conception prenant comme intrants les livrables de la phase d'analyse. Nous voyons que l'architecture logicielle traditionnelle nécessiterait de produire au moins quatre vues distinctes : une architecture des données (conception des données), une architecture fonctionnelle et/ou modulaire (conception architecturale), une autre architecture fonctionnelle et/ou modulaire pour les inter-

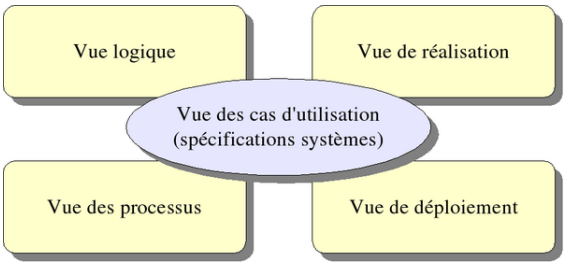
faces utilisateurs (conception des interfaces) et une architecture détaillée (ordinogrammes, états-transitions) des différents modules (conception des composants).

La pyramide exprime que chaque couche est bâtie sur la précédente. En effet, les composants réalisant les fonctionnalités du logiciel doivent manipuler des éléments de données qui doivent donc être préalablement décrits. De même, les composants réalisant les interfaces utilisateurs doivent utiliser les fonctionnalités du logiciel préalablement décrites. Et finalement, la création de l'architecture détaillée de chacun des composants du logiciel nécessite, évidemment, que ceux-ci soient préalablement inventés.

Ce modèle d'architecture impose une séparation claire entre les données, les traitements et la présentation.

Modèle d'analyse ou modèle d'architecture ? Puisque l'analyse produit également des diagrammes, il est naturel de se questionner, en effet, quand se termine l'analyse et quand commence l'architecture ? La réponse à cette question est fort simple : les éléments des diagrammes d'analyse correspondent à des éléments visibles et compréhensibles par les utilisateurs du système, alors que les éléments des diagrammes d'architectures ne correspondent à aucune réalité tangible pour ceux-ci.

2.2.2 Le modèle des 4 + 1 vues



Modèle d'architecture de Philippe Kruchten (modèle des 4 + 1 vues) *D'après : Muller P-A, Gaertner N., Modélisation objet avec UML, 2em édition. Eyrolles, p. 202, 2003.*

Le modèle de Kruchten[5] dit modèle des 4 + 1 vues est celui adopté dans l'Unified Process[6]. Ici encore, le modèle d'analyse, baptisé vue des cas d'utilisation, constitue le lien et motive la création de tous les diagrammes d'architecture.

La vue des cas d'utilisation La vue des cas d'utilisation est un modèle d'analyse formalisé par Ivar Jacobson. Un cas d'utilisation est défini comme un ensemble de scénarios d'utilisation, chaque scénario représentant une séquence d'interaction des utilisateurs (acteurs) avec le système.

L'intérêt des cas d'utilisation est de piloter l'analyse par les exigences des utilisateurs. Ceux-ci se sentent concernés car ils peuvent facilement comprendre les cas d'utilisation qui les concernent. Cette méthode permet donc d'aider à formaliser les véritables besoins et attentes des utilisateurs ; leurs critiques et commentaires étant les briques de la spécification du système.

L'ensemble des cas d'utilisation du logiciel en cours de spécification est représenté par un diagramme de cas d'utilisation, chacun des scénarios de celui-ci étant décrit par un ou plusieurs diagrammes dynamiques : diagrammes d'activités, de séquence, diagrammes de communication ou d'états-transitions.

La vue logique La vue logique constitue la principale description architecturale d'un système informatique et beaucoup de petits projets se contentent de cette seule vue. Cette vue décrit, de façon statique et dynamique, le système en termes d'objets et de classes. La vue logique permet d'identifier les différents éléments et mécanismes du système à réaliser. Elle permet de décomposer le système en abstractions et constitue le cœur de la réutilisation. En effet, l'architecte informatique récupérera un maximum de composants des différentes bibliothèques et cadriciels (framework) à sa disposition. Une recherche active de composants libres et/ou commerciaux pourra également être envisagée.

La vue logique est représentée, principalement, par des diagrammes statiques de classes et d'objets enrichis de descriptions dynamiques : diagrammes d'activités, de séquence, diagrammes de communication ou d'états-transitions.

La vue des processus La vue des processus décrit les interactions entre les différents processus, threads (fils d'exécution) ou tâches, elle permet également d'exprimer la synchronisation et l'allocation des objets. Cette vue permet avant tout de vérifier le respect des contraintes de fiabilité, d'efficacité et de performances des systèmes multitâches.

Les diagrammes utilisés dans la vue des processus sont exclusivement dynamiques : diagrammes d'activités, de séquence, diagrammes de communication ou d'états-transitions.

La vue de réalisation La vue de réalisation permet de visualiser l'organisation des composants (bibliothèque dynamique et statique, code source...) dans l'environnement de développement. Elle permet aux développeurs de se retrouver dans le capharnaüm que peut être un projet de développement informatique. Cette vue permet également de gérer la configuration (auteurs, versions...).

Les seuls diagrammes de cette vue sont les diagrammes de composants.

La vue de déploiement La vue de déploiement représente le système dans son environnement d'exécution. Elle traite des contraintes géographiques (distribution des processeurs dans l'espace), des contraintes de bandes passantes, du temps de réponse et des performances du système ainsi que de la tolérance aux fautes et aux pannes. Cette vue est fort utile pour l'installation et la maintenance régulière du système.

Les diagrammes de cette vue sont les diagrammes de composants et les diagrammes de déploiement.

3 Les styles architecturaux

L'architecture logicielle, tout comme l'architecture traditionnelle, peut se catégoriser en styles[7]. En effet, malgré les millions de systèmes informatiques construits de par le monde au cours des cinquante dernières années, tous se classent parmi un nombre extrêmement restreint de styles architecturaux[8]. De plus, un système informatique peut utiliser plusieurs styles selon le niveau de granularité ou l'aspect du système décrit. Nous ferons remarquer que, comme en architecture traditionnelle, c'est souvent par le mélange d'anciens styles que les nouveaux apparaissent.

3.1 Architecture en appels et retours

Cette architecture est basée sur le raffinement graduel proposé[9] par Niklaus Wirth. Cette approche, également appelée décomposition fonctionnelle, consiste à découper une fonctionnalité en sous-fonctionnalités qui sont également divisées en sous sous-fonctionnalités et ainsi de suite ; la devise diviser pour régner est souvent utilisée pour décrire cette démarche.

Si à l'origine cette architecture était fondée sur l'utilisation de fonctions, le passage à une méthode modulaire ou objet est toute naturelle ; la fonctionnalité d'un module ou d'un objet est réalisée par des sous-modules ou des sous-objets baptisés travailleurs (worker). Le terme hiérarchie de contrôle est alors utilisé pour décrire l'extension de cette architecture au paradigme modulaire ou objet. Une forme dérivée de cette architecture est l'architecture distribuée où les fonctions, modules ou classes se retrouvent répartis sur un réseau.

3.2 Architecture en couches

La conception de logiciels nécessite de recourir à des bibliothèques. Une bibliothèque très spécialisée utilise des bibliothèques moins spécialisées qui elles-mêmes utilisent des bibliothèques génériques. De plus, comme nous l'avons déjà mentionné, le développement efficace de composants réutilisables nécessite de créer une bibliothèque logicielle ; l'architecture en couches est la conséquence inéluctable d'une telle approche. En effet, les nouveaux composants utilisent les anciens et ainsi de suite, la bibliothèque tend donc à devenir une sorte d'empilement de composants. La division en couches consiste alors à regrouper les composants possédant une grande cohésion (sémantiques semblables) de manière à créer un empilement de paquetages de composants ; tous les composants des couches supérieures dépendants fonctionnellement des composants des couches inférieures.

3.3 Architecture centrée sur les données

Dans cette architecture, un composant central (SGBD, Datawarehouse, Blackboard) est responsable de la gestion des données (conservation, ajout, retrait, mise à jour, synchronisation, ...) . Les composants périphériques, baptisés clients, utilisent le composant central, baptisé serveur de données, qui se comporte, en général, de façon passive (SGBD, Datawarehouse). Un serveur passif ne fait qu'obéir aveuglément aux ordres alors qu'un serveur actif (Blackboard) peut notifier un client si un changement aux données qui le concerne se produit.

Cette architecture sépare clairement les données (serveurs) des traitements et de la présentation (clients) et permet ainsi une très grande intégrabilité, en effet, des clients peuvent être ajoutés sans affecter les autres clients. Par contre, tous les clients sont dépendants de l'architecture des données qui doit rester stable et qui est donc peu extensible. Ce style nécessite donc un investissement très important dans l'architecture des données. Les datawarehouses et les bases de données fédérées sont des extensions de cette architecture.

3.4 Architecture en flot de données

Cette architecture est composée de plusieurs composants logiciels reliés entre eux par des flux de données. L'information circule dans le réseau et est transformée par les différents composants qu'elle traverse. Lorsque les composants se distribuent sur une seule ligne et qu'ils ne font que passer l'information transformée à leur voisin, on parle alors d'architecture par lot (batch). Si les composants sont répartis sur un réseau informatique et qu'ils réalisent des transformations et des synthèses intelligentes de l'information, on parle alors d'architecture de médiation. Les architectures orientées évènements font également partie de cette catégorie.

3.5 Architecture orientée objets

Les composants du système (objets) intègrent des données et les opérations de traitement de ces données. La communication et la coordination entre les objets sont réalisées par un mécanisme de passage de messages. Cette architecture est souvent décrite par les trois piliers : encapsulation, héritage et polymorphisme.

L'encapsulation concerne l'architecture détaillée de chaque objet, les données étant protégées d'accès direct par une couche d'interface. De plus, les sous-fonctions, inutiles pour utiliser l'objet, sont masquées à l'utilisateur de l'objet. L'héritage permet d'éviter la redondance de code et facilite l'extensibilité du logiciel, les fonctionnalités communes à plusieurs classes d'objets étant regroupées dans un ancêtre commun. Le polymorphisme permet d'utiliser des objets différents (possédant des comportements distincts) de manière identique, cette possibilité est réalisée par la définition d'interfaces à implémenter (classes abstraites).

3.6 Architecture orientée agents

Article détaillé : Système multi-agents.

L'architecture orientée agents correspond à un paradigme où l'objet, de composant passif, devient un composant projectif :

En effet, dans la conception objet, l'objet est essentiellement un composant passif, offrant des services, et utilisant d'autres objets pour réaliser ses fonctionnalités ; l'architecture objet n'est donc qu'une extension de l'architecture en appels et retours, le programme peut être écrit de manière à demeurer déterministe et prédictible.

L'agent logiciel, par contre, utilise de manière relativement autonome, avec une capacité d'exécution propre, les autres agents pour réaliser ses objectifs : il établit des dialogues avec les autres agents, il négocie et échange de l'information, décide à chaque instant avec quels agents communiquer en fonction de ses besoins immédiats et des disponibilités des autres agents.

4 Historique

4.1 1960 à 1970

L'origine de la notion d'architecture logicielle remonte à la fin des années 1960 avec l'invention de la programmation structurée. Un programme informatique était alors conceptualisé comme une suite d'étapes (flux de contrôle) représentée par les premiers diagrammes d'architecture, les organigrammes (ordinogrammes). Au début des années 1970, avec le développement de la programmation modulaire, les programmes informatiques furent considérés comme des ensembles de composants (les modules) échangeant de l'information. Les diagrammes de flux de données furent alors utilisés pour représenter ce type d'architecture.

- 1964, création de Simula-I
- 1967, création de Simula-67

4.2 1970 à 1980

C'est au cours de la décennie 1970–80 que les grands principes architecturaux furent élaborés. L'on distingua l'architecture système décrivant les relations et interactions de l'ensemble des composants logiciels de l'architecture détaillée décrivant l'architecture individuelle de chacun des composants. L'on sépara l'architecture statique décrivant les interrelations temporellement invariables (dépendances fonctionnelles, flux de contrôle, flux de données) de l'architecture dynamique, décrivant les interactions et l'évolution du système dans le temps (diagrammes d'activité, de séquence, d'états, réseaux de Petri, etc.). C'est également au cours de cette décennie que furent élaborés les principes directeurs de l'architecture logicielle contemporaine : masquage de l'information, indépendance fonctionnelle, forte cohésion et couplage faible. Les principaux styles architecturaux virent également le jour : architecture en appels et retours (hiérarchie de contrôle), architecture centrée sur les données, architecture en flot de données, architecture en couches et architecture orientée objets.

Événements importants

- 1970, E. F. Codd publie : "*A Relational Model of Data for Large Shared Data Banks*", ACM, Vol. 13, n° 6, pp. 377-387.
- 1971, N. Wirth crée le langage Pascal. "*The Programming Language Pascal*, *Acta Informatica*, n° 1, pp. 35-63. La même année, il publie "*Program Development by Stepwise Refinement*", Comm. ACM, vol. 14, n° 4, pp. 221-227.
- 1972, O. Dahl, E. Dijkstra et C. Hoare publient : "*Structured Programming*", Academic Press.
- 1973, J.B. Dennis publie : *Modularity*, In 4dvanced Course in Software Engineering, F. Bauer, Ed., Springer- Verlag, Berlin.
- 1974, IBM définit le langage SEQUEL (*Structured English Query Language*) et l'implémente sur le prototype SEQUEL-XRM.
- 1975, M. A. Jackson. Publie : "*Principles of Program Design*", Academic Press.
- 1976-77, IBM révise SEQUEL appelée SEQUEL/2 fut définie et le nom changé en SQL.
- 1979, *Relational Software* introduit son produit *Oracle V2* comme système de gestion de bases de données relationnelles. Cette version implémente un langage SQL de base (requête et jointure).
- 1979, E. Yourdon et L.L. Constantine publient : "*Structured Design : Fundamentals of a Discipline of Computer Program and Systems Design*", Prentice Hall.

4.3 1980 à 1990

La décennie 1980-90 fut celle du développement de l'architecture orientée objet. Ce type d'architecture introduisit trois nouveaux types de composants logiciels : l'objet, la classe et la méta-classe ; ainsi que des relations ontologiques entre ces composants : est un (héritage), est composé de (composition), etc. La relation d'héritage est une innovation majeure permettant la réutilisation de code et facilitant son adaptation à d'autres contextes d'utilisation.

Au niveau industriel, par contre, cette décennie est sans conteste celle de l'architecture à trois couches centrée sur les données (3-tiers). Ce type d'architecture logicielle, séparant l'architecture des programmes de l'architecture des données, s'oppose ainsi complètement au principe de forte cohésion prôné par l'architecture objet. L'accent est mis sur l'architecture des données ; les diagrammes de ce type d'architecture sont les modèles conceptuels de données et les schémas entités relations. Le développement des systèmes de gestion de bases de données relationnelles, des protocoles multibases ainsi que leurs normalisations (standardisations) constituent les fondations technologiques de ce choix architectural.

Événements importants

- 1983, Grady Booch publie : "*Software Engineering with Ada*", Benjamin Cummings.
- 1983, *Relational Software* devient *Oracle Corporation* et introduit *Oracle V3* (support des transactions).
- 1983-1985, Bjarne Stroustrup développe le C++ au Bell Laboratory.
- 1984, U. Dayal and H. Hwang publient : "*View definition and generalization for database integration in MULTIBASE : A system for heterogeneous distributed databases*", IEEE Trans, Software Engineering, SE-10, No. 6, 628-644.
- 1986, SQL a été adopté par l'institut de normalisation américaine (ANSI), puis comme norme internationale par l'ISO (ISO/CEI 9075).
- 1987, Ivar Jacobson fonde *Objectory Systems* pour vendre sa méthode de développement : *ObjectOry*
- 1990, A.P. Sheth and J.A. Larson publient : "*Federated Database Systems and Managing Distributed, Heterogeneous, and Autonomous Databases*". ACM Computing Surveys.

4.4 1990 à 2000

Au début de la décennie 1990-2000 on dénombrait un très grand nombre de représentations architecturales distinctes. En 1995, UML (Unified Modeling Language) devint la norme internationale de représentation de l'architecture logicielle. Le développement orienté objet se répand dans l'industrie, les systèmes de gestion de bases de données sont maintenant perçus comme une façon commode d'assurer la persistance des objets. Les systèmes de gestion de base de données objet-relationnels et objets font leurs apparitions. On voit également apparaître les architectures distribuées ; les programmes informatiques ne sont plus simplement perçus comme devant offrir des services à des êtres humains mais également à d'autres programmes. L'arrivée des réseaux ouverts, en particulier Internet, change complètement le paysage architectural. L'architecture à trois couches centrée sur les données (3-tiers) est maintenant réalisée par le triplet serveur de base de données, serveur d'application web et navigateur web.

La recherche universitaire sur l'architecture logicielle se concentre davantage sur les problèmes de couplage entre objets et d'interopérabilité syntaxique et sémantique. Le problème du couplage est essentiellement perçu comme un problème de communication entre objets, voire de dialogues entre agents intelligents ; l'architecture orientée agent apparaît. Le principe de réutilisation des composants logiciels est maintenant appliqué à l'architecture. Des façons de faire, principes ou styles architecturaux peuvent être réutilisés ; les patrons de conception apparaissent.

Événements importants

- 1992, Gio Wiederhold publie : "*Mediators in the Architecture of Future Information Systems*", IEEE Computer Magazine
- 1994, La définition moderne d'agent logiciel est accepté par la communauté scientifique : "*Special issue on intelligent services.*", Communication of the ACM.
- 1994, Sun Microsystems lance le langage de programmation pur objet Java.
- 1995, UML voit le jour (OOPSLA'95).
- 1995, Gamma *et al.* publient : "*Design Patterns : Elements of Reusable Object-Oriented Software*", Addison-Wesley.
- 1998, Le W3C recommande le langage à balise XML.

4.5 2000 à 2010

Cette décennie est caractérisée par un retour des bases de données distribuées rendu possible grâce aux technologies XML. Le XML est un ensemble de règles permettant de représenter et de structurer des données, c'est une restriction de SGML. Ces données peuvent être syntaxiquement normalisées et la souplesse de la technologie permet d'exprimer des sémantiques variées. L'architecture

3-tiers traditionnelle peut se retrouver sous la forme de trois couches de médiations de données : gestion de données XML, transformation et fusion de données XML et présentation de données XML. La technologie XML permet la spécification syntaxique (DTD, XSD), la transformation (XSLT), la présentation (XSL) et la spécification sémantique (RDF, RDFS, OWL). Il existe des bibliothèques logicielles permettant de gérer les données XML et la plupart des systèmes de gestion de base de données supportent maintenant XML.

Ce nouveau type d'architecture s'oppose à la vision centralisée des données que l'on retrouve dans une architecture centrée sur les données traditionnelle (comme le « datawarehouse » promu par IBM). Cette forme d'architecture se nomme la *médiation de données*, elle est caractérisée par :

- Un traitement intelligent de l'information (des données supportant un haut niveau d'abstraction et de généralisation).
- Un accès et une intégration de plusieurs sources d'information.
- Une transformation dynamique du flux d'information par des filtres et traducteurs.
- L'existence de répertoires intelligents et de bases d'information comme des catalogues.
- La gestion de l'incertitude reliée aux données absentes ou incomplètes et aux données mal comprises.
- La gestion explicite de l'interopérabilité sémantique grâce à des ontologies générales et de domaines.

Le développement orienté agent (OA) sort progressivement des universités, il existe une multitude d'outils logiciels pour la conception de systèmes basés sur les agents mais la plupart ne sont pas encore destinés à devenir des outils de production. Le langage KQML (Knowledge Query and Manipulation Langage) est un langage de communication inter-agent qui pourrait très bien s'imposer dans un proche avenir. Il n'y aura pas de révolution au niveau des langages de programmation, les différentes fonctionnalités des agents sont implémentées à l'aide de bibliothèques logicielles. Les trois types d'architectures OA qui se dégagent sont : *l'architecture réfléchie*, *l'architecture réactive* et *l'architecture hybride*.

5 Outils

- Azuki - Framework Java ayant pour objectif la separation des préoccupations.
- BoUML- Modeleur UML open source multiplateforme compatible avec la norme UML 2.0 capable d'effectuer de la rétro-ingénierie
- IBM Rational - Le produit des trois amigos fondateurs d'UML
- Silverrun - Un logiciel de modélisation pour la méthode Datarun, version nord-américaine de Merise.
- Open ModelSphere - Modélisation de données conceptuelle et relationnelle, modélisation de processus d'affaires et modélisation UML sous licence GPL.
- Acceleo - Générateur de code Open Source basé sur Eclipse et EMF
- Power Designer - Logiciel de modélisation des méthodes Merise, SGBD, UML, Processus métiers, Architecture d'entreprise.
- DocGen BOUML - Plugin de BOUML permettant la génération des documents d'architecture, d'analyse et de design à partir d'un modèle UML et de la philosophie OpenUP .
- Papyrus - Plug-in Eclipse de modélisation UML2, SysML et MARTE : .
- Obeo Designer - Atelier de modélisation sur-mesure basé sur la plateforme Eclipse et une approche DSL :
- Capella - Un atelier de modélisation des architectures systèmes, matérielles et logicielles.

6 Notes

[1] Au Québec, on note une utilisation fréquente mais erronée du terme “architecture organique” comme étant de l'architecture logicielle, surtout dans les milieux gouvernementaux.

7 Notes et références

[1] Pressman R. S., Software Engineering : A Practitioner's Approach, Third Édition. McGraw-Hill. Chapitre 4, p. 107, 1992.

[2] Yourdon E., Software Reuse. Application Development Strategies. vol. 1, n0. 6, p. 28-33, juin 1994.

[3] David Garlan, Robert Allen, John Ockerbloom, *Architectural Mismatch : Why Reuse Is So Hard*, IEEE Software, Nov./Dec. 1995

[4] Perry D.E, Wolf A.L., Foundation for the study of Software Architecture. ACM Software Eng. Notes, p. 40-50, octobre 1992

[5] Philippe B. Kruchten, *The 4+1 View Model of Architecture*, IEEE Software, novembre 1995.

[6] Jacobson I., Booch G., Rumbaugh J., The Unified Software Development Process, ISBN 0-201-57169-2

[7] Bass L., Clement P., Kazman R., Software Architecture in Practice, Addison-Wesley, 1998

[8] David Garlan et Mary Shaw, *An Introduction to Software Architecture*, CMU-CS-94-166, School of Computer Science, Carnegie Mellon University, janvier 1994

[9] Wirth N., Program Development by Stepwise Refinement, CACM, vol. 14, no. 4, 1971, pp. 221-227

-  Portail de la programmation informatique

8 Sources, contributeurs et licences du texte et de l'image

8.1 Texte

- **Architecture logicielle** *Source :* https://fr.wikipedia.org/wiki/Architecture_logicielle?oldid=119865465 *Contributeurs :* Michel BUZE, Tieno, Sanao, Urhixidur, JB, Elg, Outs, Romanc19s, RobotQuistnix, MMBot, 16@r, Jlancey, Pautard, Paglop, Adzero, Liquid-aim-bot, FHd, PieRRoBoT, Rhadamante, JnRouvignac, A2, Escarbot, Deep silence, BOT-Superzerocool, MirgolthBot, TomT0m, Xibot, Nono64, Nipou, Slacrampe, TXiKiBoT, Herve1729, Mathias.bollaert, OKBot, Ange Gabriel, Dhatier, DumZiBoT, BOTarate, Alexbot, Mayayu, SooW, HerculeBot, SilvonenBot, ZetudBot, Luckas-bot, Micbot, SimonMalenky, MastiBot, Lomita, RedBot, Nicolapedia, KamikazeBot, ZéroBot, ChuispastonBot, MikeGyver, Marco.savard, Paul.schrepfer, Bertol, Sergelucas, HSeine, AvocatoBot, LouisAlain, Ecosoq, Moun3imy, Addbot, Christophe.gatti et Anonyme : 47

8.2 Images

- **Fichier:Graphique_dégradation_logiciel.png** *Source :* https://upload.wikimedia.org/wikipedia/commons/0/02/Graphique_d%C3%A9gradation_logiciel.png *Licence :* CC-BY-SA-3.0 *Contributeurs :* Oeuvre personnelle, inspirée de : Pressman R. S., Software Engineering : A Practitioner's Approach, Fifth Edition. McGraw-Hill. Chapitre 1, p. 8, 2001. *Artiste d'origine :* ?
- **Fichier:Modèle_architecture_Kruchten.png** *Source :* https://upload.wikimedia.org/wikipedia/commons/4/4f/Mod%C3%A8le_architecture_Kruchten.png *Licence :* CC-BY-SA-3.0 *Contributeurs :* œuvre personnelle, d'après : Muller P-A, Gaertner, N., Modélisation objet avec UML, 2em édition. Eyrolles, p. 202, 2003. *Artiste d'origine :* ?
- **Fichier:Modèles_analyse_et_architecture_centrées_données.png** *Source :* https://upload.wikimedia.org/wikipedia/commons/9/9c/Mod%C3%A8les_analyse_et_architecture_centr%C3%A9es_donn%C3%A9es.png *Licence :* CC-BY-SA-3.0 *Contributeurs :* Oeuvre personnelle, inspirée de Pressman R. S., Software Engineering : A Practitioner's Approach, Fifth Edition. McGraw-Hill. Chapitre 13, p. 337, 2001. *Artiste d'origine :* ?
- **Fichier:Nuvola_apps_kcmsystem.svg** *Source :* https://upload.wikimedia.org/wikipedia/commons/7/7a/Nuvola_apps_kcmsystem.svg *Licence :* LGPL *Contributeurs :* Own work based on Image:Nuvola apps kcmsystem.png by Alphax originally from [1] *Artiste d'origine :* MesserWoland
- **Fichier:Recycle001.svg** *Source :* https://upload.wikimedia.org/wikipedia/commons/4/44/Recycle001.svg *Licence :* Public domain *Contributeurs :* Originally from en.wikipedia ; description page is (was) here *Artiste d'origine :* Users Cbuckley, Jpowell on en.wikipedia

8.3 Licence du contenu

- Creative Commons Attribution-Share Alike 3.0