

437 rue des Apothicaires
34000 Montpellier

Domaine de Voluceau
Rocquencourt - BP 105
78153 Le Chesnay Cedex

Mémoire de fin d'études

« En quoi les architectures logicielles et intergicielles
peuvent-elles simplifier le processus de développement et
d'évolution des applications multiplateformes ? »

Benjamin BILLET

Sommaire

1 - INTRODUCTION	1
2 - LE MULTIPLATEFORME, POURQUOI FAIRE ?	12
2.1 - DE L'HETEROGENEITE DES ENVIRONNEMENTS ACTUELS	12
2.2 - LA NECESSITE DE S'OUVRIRE, D'ETRE COMPATIBLE, EXHAUSTIF ET INTEROPERABLE	15
2.2.1 - <i>Qu'est-ce qu'une plateforme, la compatibilité, l'interopérabilité ?</i>	15
2.2.2 - <i>La nécessité d'une stratégie d'ouverture</i>	18
2.2.3 - <i>Pourquoi cibler plusieurs plateformes à l'heure actuelle ?</i>	19
2.3 - LA COMPLEXITE ORGANISATIONNELLE ET TECHNIQUE INDUITE PAR LE MULTIPLATEFORME PAR RAPPORT AUX PROJETS CLASSIQUES	21
2.3.1 - <i>La nécessité de raisonner de manière multidimensionnelle</i>	22
2.3.2 - <i>Des difficultés pour gérer le développement et l'évolution</i>	24
2.3.3 - <i>La démultiplication de la charge et des coûts</i>	27
2.4 - AU FINAL, QUELS ENJEUX POUR LE MULTIPLATEFORME ?	28
3 - COMMENT SIMPLIFIER L'APPROCHE MULTIPLATEFORME ?	30
3.1 - PAS DE SOLUTION EVIDENTE, MAIS UN EVENTAIL DE POSSIBILITES	30
3.2 - AU SEIN MEME DU PROCESSUS DE DEVELOPPEMENT, DES METHODES EXISTENT	30
3.2.1 - <i>Exploiter un code source de référence, une solution fastidieuse</i>	30
3.2.2 - <i>Un mécanisme historique : le préprocessing</i>	33
3.2.3 - <i>Les composants portables, une méthode très appréciée</i>	35
3.2.4 - <i>La conversion automatique de code, un processus complexe et insatisfaisant</i>	36
3.2.5 - <i>Mais ces approches s'intéressent à des aspects locaux</i>	38
3.3 - L'ORGANISATION DU PROJET MULTIPLATEFORME APPORTE ELLE AUSSI DES REPONSES	38
3.3.1 - <i>En parallélisant les projets</i>	38
3.3.2 - <i>Ou en utilisant un projet pilote</i>	40
3.4 - LES ARCHITECTURES LOGICIELLES ET INTERGICIELLES, AU SERVICE DE LA TECHNIQUE ET DE L'ORGANISATION, POUR SIMPLIFIER A LA FOIS LE PROCESSUS DE DEVELOPPEMENT, MAIS AUSSI LE PROCESSUS D'EVOLUTION	40
3.4.1 - <i>La granularité et la réflexion au service du découpage intelligent</i>	41
3.4.2 - <i>Abstraire les spécificités techniques au travers des aspects</i>	45
3.4.3 - <i>Aller plus loin dans l'approche objet au travers des composants distribués</i>	48
3.4.4 - <i>Adopter une approche en couches réparties</i>	51
3.4.5 - <i>Raisonner par services, pour orchestrer et factoriser</i>	54
3.4.6 - <i>Virtualiser, pour que les systèmes s'adaptent à tout</i>	58
3.4.7 - <i>Placer les modèles au cœur de la fabrication logicielle</i>	61
4 - UN CAS PRATIQUE : LA TELEPHONIE MOBILE	73

4.1 - LA TELEPHONIE MOBILE, UN UNIVERS MULTIPLATEFORME PAR EXCELLENCE.....	73
4.1.1 - <i>Un certain nombre de plateformes mobiles</i>	73
4.1.2 - <i>Un marché inégal et des disparités importantes entre les constructeurs</i>	74
4.1.3 - <i>Une effervescence aux enjeux importants</i>	77
4.2 - UN MIDDLEWARE POUR RATIONALISER LA COMMUNICATION ENTRE LES PLATEFORMES ET LE DEVELOPPEMENT D'APPLICATION DE HAUT NIVEAU	79
4.2.1 - <i>L'INRIA, l'équipe ARLES et la startup Ambientic</i>	79
4.2.2 - <i>iBICOOP, un middleware pour unifier les échanges entre terminaux mobiles</i>	80
4.3 - LA SIMPLIFICATION DU PROCESSUS DE DEVELOPPEMENT ET D'EVOLUTION : APPORTS CONCRETS ET REFLEXIONS FUTURES....	87
4.3.1 - <i>En quoi iBICOOP permet-il de simplifier le processus de développement et d'évolution ?</i>	87
4.3.2 - <i>Exemple d'exploitation d'iBICOOP : le partage de photo multiplateforme</i>	90
4.3.3 - <i>iBICOOP, lui aussi victime des difficultés inhérentes au processus de développement des environnements multiplateformes</i>	93
4.3.4 - <i>Les limitations et les perspectives envisageables</i>	98
5 - CONCLUSION	100
6 - BIBLIOGRAPHIE	108
7 - GLOSSAIRE	113
8 - TABLES	116
8.1 - TABLE DES MATIERES	116
8.2 - TABLE DES ILLUSTRATIONS	118

Je remercie, dans l'ordre alphabétique, Emil ANDRIESCU, Bachir MOUSSA TARI BAKO, Pierre-Guillaume RAVERDY, Roberto SPEICYS-CARDOSO et toutes les autres personnes avec qui j'ai directement travaillé pendant ces six mois, pour l'excellent accueil et le soutien dont ils ont pu faire preuve au cours de ma mission.

Je tiens aussi à remercier Valérie ISSARNY pour avoir été l'une des rares intervenantes de l'INRIA à répondre à mes messages pendant ma recherche de stage.

Je remercie enfin l'ensemble de l'équipe ARLES, pour sa bonne ambiance générale et les présentations du jeudi matin. Mention spéciale à Animesh PATHAK pour ses petits cours d'algorithmie 😊.

1 - Introduction

En informatique, un système est un ensemble d'éléments obéissant à certaines règles et concourant à assurer une fonction au travers d'interactions plus ou moins riches et dynamiques. Cette notion fondamentale s'établit à tous les niveaux d'abstraction, c'est-à-dire du plus faible, l'électronique pure, au plus élevé, l'urbanisation des systèmes d'information. Par nature, les systèmes sont hétérogènes, car ils sous-entendent que les éléments qui les composent possèdent des caractéristiques qui leur sont propres. Ces éléments disposent d'un rôle au sein du système et de mécanismes qui leur permettent d'assurer ce rôle. Concrètement, là où plusieurs éléments peuvent avoir un même rôle (ou un rôle similaire), leurs mécanismes peuvent différer fondamentalement. Par exemple, les cartes de crédit sont des éléments à part entière du système de paiement, mais toutes ne fonctionnent pas de la même façon et n'offrent pas les mêmes services, bien qu'elles aient toutes pour rôle de concourir à la procédure de paiement. Aussi, depuis les balbutiements de l'informatique dans les années cinquante, cette hétérogénéité technique est restée une contrainte forte. Qu'il s'agisse des premiers superordinateurs ou de l'informatique nomade telle que nous la connaissons aujourd'hui, la forte complexité a conduit les producteurs de technologie sur des chemins divergents, ce qui a eu pour effet d'induire des difficultés liées aux différences physiques et/ou logicielles propres à telle ou telle solution technique.

Ces divergences sont d'origines multiples et non exclusives. Tout d'abord, elles peuvent être purement techniques, c'est-à-dire découler d'une réponse technologique vis-à-vis d'un critère spécifique comme, par exemple, la recherche de la robustesse, de la fiabilité ou de la performance. Ce critère, que l'on appelle parfois « aspect », va influencer sur la structure et les mécanismes de la solution et provoquer une incompatibilité avec les autres solutions qui proposent des fonctionnalités similaires, mais basées sur des critères différents. De fait, chaque solution peut être vue comme une suite d'avantages et d'inconvénients spécifiques qui vont influencer sur le choix final, bien que chaque solution soit en mesure de remplir le même rôle. Par exemple, toutes les unités de stockage permettent de stocker des données, mais certaines sont plus adaptées pour la lecture où pour l'écriture, certaines sont plus robustes, certaines sont plus rapides, etc.

Dans d'autres cas, ces disparités peuvent être d'ordre stratégique. Une entreprise peut produire des systèmes fermés pour protéger son marché ou exploiter une stratégie de gamme amenant à proposer plusieurs solutions en fonctions des besoins de ses clients et du niveau d'enfermement que l'entreprise veut (ou peut) leur imposer. De fait, les applications doivent tenir compte des spécificités de chaque gamme et anticiper le meilleur comme le moins bon en jonglant avec la notion de « fonctionnalité supportée », c'est-à-dire l'ensemble commun de fonctionnalités que fournit n'importe quel élément de la gamme. Par exemple, toutes les voitures peuvent rouler, mais toutes les voitures ne sont pas équipées d'un GPS. De fait, il est nécessaire soit d'omettre l'existence du GPS pour rester parfaitement générique, soit de prévoir un mécanisme pour tenir compte de sa présence ou de son absence.

Parfois, il arrive que ces différences aient aussi une composante culturelle. Celle-ci peut être liée à un contexte global, comme c'est le cas pour les unités de mesure (normes anglo-saxonnes et normes internationales), ou spécifique au domaine de travail. Par exemple l'informatique possède une culture riche et parfois contre-intuitive, avec ses us et coutumes bien particuliers avec lesquels il faut aussi savoir composer avec ceux-ci. Cette culture peut donc conduire les concepteurs à travailler de plusieurs façons différentes, sur la base de critères arbitraires (« c'est ce qui se fait », « on a toujours fait comme ça », « c'est éprouvé ») ou logiques (« c'est le plus adapté à... », « l'expérience démontre que... »). D'autres paramètres, comme le contexte juridique et le contexte sociologique, peuvent influencer sur la composante culturelle et amener les individus à emprunter des chemins radicalement différents pour parvenir au même but.

Enfin, le choix des solutions techniques au sein d'un système est fortement influencé par des contraintes externes (économiques, temporelles, etc.), comme ce fut le cas dans le monde de la vidéo dans les années 70 où l'on assista à la lutte entre plusieurs standards de vidéocassettes. Dans ce cas-ci, le système privilégié sera celui qui apportera le meilleur lot d'avantages/inconvénients par rapport à un budget ou un délai, la notion de multiplateforme étant généralement reléguée à l'état de « bonus ».

Ces singularités de point de vue, induites par les paramètres complexes et variés que nous venons d'évoquer, conduisent donc à une multitude de technologies hétéroclites,

fondamentalement différentes et bien souvent concurrentes. Celles-ci, combinant leurs lots d'avantages et d'inconvénients, sont exploitées en fonction de besoins spécifiques, ou disparaissent avec le temps. En informatique, on peut citer, entres nombreux autres, quelques exemples de technologies différentes ayant pourtant un rôle identique :

- **les architectures de processeur** (Intel x86, Power PC, etc.).
- **les systèmes d'exploitation** (Windows, Unix, etc.).
- **les langages de programmation.**
- **les formats de fichier** (OpenXML et OpenDocument^a, par exemple).

Dans l'univers de la fabrication logicielle, la prise en compte de ces spécificités lors d'un processus de développement est appelée « développement multiplateforme ». Multiplateforme fait ici référence à l'aspect « plusieurs plateformes », où le terme plateforme est apprécié dans son sens le plus général, c'est-à-dire en tant que base de travail. Ainsi, une plateforme peut représenter n'importe quel système, qu'il s'agisse de matériel (architecture des ordinateurs, architecture de processeur), de système d'exploitation ou d'outils quelconque (système de gestion de base de données, langage de programmation, serveur, etc.). En un sens, si tout est système alors tout est plateforme. De fait, l'aspect multiplateforme consiste à prendre en charge les spécificités de plusieurs plateformes de façon à produire des systèmes capables de fonctionner à partir de deux ou plusieurs plateformes. Par exemple, une application compatible avec plusieurs sources de données peut être considérée comme issue d'un processus multiplateforme. Partiellement, toutefois, puisque l'accès aux sources de données est un élément parmi tant d'autres au sein du système. Plus nombreux seront les éléments du système à devoir tendre vers une approche multiplateforme, plus grandes seront les difficultés qui en découleront, la complexité maximale étant atteinte lorsque le système tout entier doit être multiplateforme.

^a OpenXML et OpenDocument sont deux formats de données pour les applications bureautiques.

A cause des incompatibilités à la fois matérielles et logicielles entre les éléments d'un système, le développement d'applications multiplateformes peut donc devenir un véritable casse-tête, avec son lot d'écueils rédhibitoires. Plusieurs questions méritent d'être posées lorsque l'on aborde la question d'un système multiplateforme. Tout d'abord, comment assurer un processus de développement cohérent sur plusieurs plateformes ? Et, surtout, comment poursuivre ce développement pour l'ensemble des évolutions futures de l'application ? En effet, ce n'est pas tout de porter un logiciel et de l'adapter aux spécificités inhérentes à une plateforme, encore faut-il le faire évoluer par la suite sans que les écarts entre les versions deviennent trop importants en termes de fonctionnalité. De plus, même si les deux applications produites sont identiques, il est important de réfléchir à la prise en charge des anomalies spécifiques qui pourraient survenir sur une plateforme, mais pas une autre.

La complexité du problème est due, avant toute chose, à une complexité technique. Réécrire des applications pour des plateformes radicalement différentes va nécessiter, en plus des compétences techniques adaptées à chaque système cible, une grande part de réflexion et d'adaptation, du début à la fin du processus de conception et de production logicielle. En effet, les plateformes disposent généralement de leur philosophie propre et certaines fonctionnalités spécifiques ne peuvent pas être retrouvées ailleurs (on peut citer certains mécanismes de communication qui diffèrent radicalement entre le monde Unix^a et le monde NT^b). Ceci est d'autant plus vrai lorsque l'on aborde des problématiques de bas niveau^c d'abstraction où les subtilités inhérentes aux mécanismes du système doivent être maîtrisées pour telle ou telle raison. Finalement, cela revient à rechercher une abstraction sur des mécanismes où l'aspect concret est primordial, comme c'est notamment le cas des

^a Système d'exploitation multitâche et multi-utilisateur ayant donné naissance à toute une famille de systèmes, notamment GNU/Linux, BSD, AIX et Mac OS X.

^b Windows NT désigne une famille de systèmes d'exploitation multitâche et multi-utilisateur (Windows XP, Windows 7, Windows 2003 Server, etc.) créée par Microsoft.

^c Le terme « bas niveau » fait référence aux mécanismes proches du matériel (bas niveau d'abstraction). Par opposition, le terme « haut niveau » fait référence aux mécanismes abstraits ne tenant pas compte des détails inhérents au fonctionnement des machines (haut niveau d'abstraction).

infrastructures de développement 3D, qui sont nécessairement proches du matériel et qui emploient des conventions spécifiques, nombreuses et complexes.

Du fait de cette complexité technique, le coût et la durée du développement se retrouvent accrus, tout comme la taille potentielle des équipes de travail. De fait, ces coûts ont longtemps été mis en perspective avec l'occupation du marché par telle ou telle technologie. « Est-ce rentable de porter une application sur telle ou telle plateforme physique ou logicielle ? » est une question qui revient souvent lors des études préliminaires (phase d'inception). Bien entendu, il n'y a pas de réponse toute faite, mais il est certain que le choix d'un portage pour tel ou tel système ne peut être fait sans considérer un rapport entre les difficultés (technique, budgétaire, etc.) et les bénéfices (part de marché, visibilité, etc.) potentiels.

L'autre complexité majeure est organisationnelle, c'est-à-dire corrélée à la gestion de projet. Par rapport au processus de développement classique et au suivi du cycle de vie d'un produit unique, nous avons ici une contrainte « dimensionnelle » à respecter, c'est-à-dire qu'à un instant donné il existera plusieurs versions du produit, différentes d'un point de vue technique, mais identique d'un point de vue client. Nous pourrions comparer notre produit à une population d'organismes vivants, scindée en petits groupes soumis à des contraintes environnementales différentes. En fonction de leur environnement, les groupes évolueraient différemment, bien qu'issus d'une base identique, et parviendraient à former de nouvelles populations aux caractéristiques spécifiques. La principale difficulté, si l'on considère le cas idéal où chaque déclinaison de notre produit est une population totalement indépendante, réside donc dans l'assurance d'une cohérence tout au long des phases clés du processus de développement. En effet, la complexité du multiplateforme s'inscrit dans toutes les phases de conduite de projet :

- **Conception** : « Comment tenir compte des spécificités et possibilités de chaque plateforme ? »
- **Développement** : « Comment implémenter la fonctionnalité à partir des spécificités de la plateforme ? »

- **Test** : « Comment assurer une cohérence et une généricité dans mon processus de test ? »
- **Maintenance** : « Comment assurer le suivi et le dépannage commun des applications produites ? »
- **Evolution** : « Comment assurer la prise en charge des nouvelles fonctionnalités au sein de chaque plateforme ? »

Avec le temps et l'expérience, le monde de l'informatique commence à apporter des réponses pour simplifier le développement des applications au travers de plusieurs plateformes différentes. Actuellement, une grande part des solutions proposées se penchent sur les problèmes techniques inhérents au développement multiplateforme (« comment s'assurer que la technologie employée sur la plateforme *A* pourra être réemployée sur la plateforme *B* »), ce qui passe par une rationalisation des systèmes au travers de normes et de standards^a internationaux. Au fil du temps, le paysage technique se réduit considérablement, de par la disparition de certains acteurs ou de certaines technologies, mais aussi par l'apparition et l'adoption de spécifications et de méthodologies (les « bonnes pratiques ») communes et éprouvées. Cependant, le milieu informatique est une terre fertile où les erreurs du passé sont régulièrement reproduites à des niveaux d'abstraction toujours plus élevés. De fait, si les plateformes et les technologies plus âgées se rationalisent peu à peu, ce n'est pas le cas de leurs homologues plus récents. Ceux-ci, généralement issus de recherches de pointe et d'un processus d'innovation constant, ne sont pas normalisés sur le champ. En effet, les organismes de normalisation doivent observer et analyser les évolutions technologiques avant de pouvoir effectuer leur travail, ce pour garantir une certaine maturité aux normes produites. Ainsi, entre l'apparition d'un ensemble de technologies similaires et leur rationalisation en une norme unique, il peut s'écouler un temps significatif. C'est ce décalage entre rationalisation et bouillonnement technologique qui va poser problème pour le développement multiplateforme car il va être nécessaire, du moins en pendant un certain temps, de composer avec un large ensemble de technologies.

^a Une norme se basant généralement sur le meilleur de chaque standard.

En parallèle de l'évolution des techniques de développement multiplateforme, les entreprises ont manifesté un intérêt croissant vis-à-vis de l'industrialisation du développement informatique. Les premiers balbutiements sur ce sujet remontent à la fin des années 60^a et ont suscité un engouement plutôt tardif, une fois passée l'étape de l'informatisation hâtive et désordonnée des années 70. Concrètement, industrialiser le développement consiste à appliquer au domaine informatique des techniques industrielles éprouvées dans le monde de la productique. En effet, le concept d'usine logicielle, où l'organisation est découpée en tâches fortement granulaires, à l'image une chaîne de production, et où les tâches répétitives (compilation, tests unitaires, déploiement, etc.) sont automatisées au maximum, s'inspire du fonctionnement d'une usine réelle pour assurer une maîtrise et un suivi optimal du processus de développement.

De fait, au niveau des directions des systèmes d'information, l'industrialisation du développement est un enjeu majeur car, en partant du principe qu'il est possible d'appliquer à l'informatique tout ou partie des méthodes industrielles, il serait envisageable de maîtriser parfaitement les délais, les coûts et les anomalies associés au processus de développement comme c'est aujourd'hui possible dans d'autres secteurs. Au-delà des aspects organisationnels certains, si l'industrialisation peut aider à une meilleure maîtrise des projets classiques alors elle peut concourir à une meilleure maîtrise des projets multiplateformes.

Cependant, l'industrialisation parfaite et totale de l'informatique est une chimère car, contrairement à ce que voudrait nous faire croire l'analogie avec l'industrialisation automobile, l'informatique n'est pas un domaine de production de masse. Le processus de production de voitures est efficace car il ne fait qu'effectuer des copies d'un modèle d'origine, ce qui rend une totale automatisation tout à fait envisageable. En informatique, le logiciel n'est produit (écriture de code) qu'une seule fois et sa reproduction ne pose aucune difficulté étant donné son caractère immatériel (répliquer une suite d'octets est une opération négligeable). Pour rester dans l'analogie automobile, cela voudrait dire que ce n'est pas le processus de reproduction de voiture qui est automatisé, mais le processus de

^a Avec notamment un compte-rendu de conférence : « Mass Produced Software Components », By M.D. Mc ILROY.

conception de la voiture initiale. Partant de là, automatiser et gérer finement un processus de réflexion devient nettement plus complexe puisqu'il implique des composantes liées à la pensée humaine.

Aussi, bien que l'on ait accompli de grands progrès en termes d'industrialisation, le développement informatique comprend toujours une notion artisanale car la transformation d'un concept donné en code informatique va être influencée par un certain nombre de facteurs dont les effets vont être difficiles à atténuer.

Des facteurs humains, tout d'abord, car il existe un très grand nombre de manières différentes pour implémenter un concept, même si celui-ci a été restreint. Chaque être humain étant influencé par son vécu et son expérience, cela impacte sur ses choix en termes de développement, le conduisant à préférer telle ou telle option pour telle ou telle raison.

Des facteurs liés à l'organisation, ensuite, car chaque chef de projet, chaque manager et chaque direction des systèmes d'informations possèdent une culture, des méthodes et des manières de procéder qui lui sont propres.

Des facteurs spécifiques, aussi, souvent liés à l'environnement contextuel d'un projet (forte concurrence, urgence, incertitude technique, budget limité, etc.) et influant sur les paramètres de qualité, de coût et de délai.

Enfin, il existe des facteurs marketing liés aux évolutions technologiques. Contrairement à ce que l'on pourrait croire, l'informatique est un milieu sujet aux effets de modes^a et les technologies populaires d'aujourd'hui ne seront pas les technologies populaires de demain. De fait, certains choix peuvent amener un projet à s'enfermer au sein d'une plateforme, complexifiant d'autant plus les phases d'évolutions futures.

A l'heure actuelle, l'industrialisation se manifeste au travers de deux axes majeurs : l'axe technique et l'axe méthodologique. L'axe technique s'intéresse à tout ce qui peut rationaliser le développement, c'est-à-dire la simplification, l'automatisation, la réutilisabilité

^a Quelques exemples : C++, Java, .NET, UML, JEE, XML, PHP, Linux, les terminaux passifs puis les clients lourds à nouveau puis les clients web, l'informatique "nomade", le B2B, les EAI puis les ESB, les SGBDOO, les flux RSS puis Atom, le social networking, etc.

et le respect des standards, entre autres. Généralement, cet axe pousse à son plus haut niveau le concept de composants réutilisables : le framework. Il s'agit d'une base logicielle offrant un cadre technique et structurel imposant une philosophie au développeur, tout en lui apportant une vision simplifiée des mécanismes couramment exploités dans un projet informatique. Cela permet aux développeurs de se concentrer plus sereinement sur la partie métier en fournissant les abstractions nécessaires pour que ne plus avoir à réinventer la roue.

A un niveau plus élevé, l'axe méthodologique se focalise sur la gestion de projet. Il se base sur des normes qualité (ISO9000, par exemple) et des référentiels de bonne pratique (CMMi, par exemple) pour définir les méthodes de travail les plus adaptées en fonction des projets, des ressources, des contraintes et des objectifs.

Globalement, donc, l'industrialisation et la rationalisation du développement concourent à résoudre certaines difficultés liées à la programmation multiplateforme, mais se heurtent à l'évolution rapide des technologies, tel que je l'ai évoqué précédemment. Concrètement, ce qui diverge aujourd'hui entre les systèmes ce ne sont plus forcément les langages, mais les frameworks applicatifs qui sont greffés dessus. Par exemple, même si Java est standardisé, il existe énormément de frameworks différents, ce qui nuit à l'idée initiale de « cadre imposé ». De plus, avec l'explosion de la téléphonie mobile nouvelle génération et le caractère lucratif de ce marché^a (ce qui sous-entend donc la variété et le nombre de ses acteurs), les difficultés liées au développement multiplateforme resurgissent comme au premier jour, notamment car le monde « mobile » n'offre pas les mêmes fonctionnalités que le monde « desktop »^b en terme de matériel (et donc de possibilités).

J'ai été amené au cours de mon stage à prendre en charge la migration d'une application complexe destinée à assurer l'interopérabilité entre les téléphones mobiles et il m'a paru

^a En novembre 2010, le Gartner Group annonçait que les smartphones représentaient 19.3 % des ventes de téléphone mobile, le double de l'année précédente.

^b Le terme « desktop » fait référence au bureau de l'ordinateur. L'informatique « desktop » désigne l'ensemble des applications destinées à être installées en dur sur les machines, en opposition aux applications « légères » s'exécutant le plus souvent dans un navigateur web.

nécessaire de réfléchir aux méthodes envisageables permettant de simplifier le processus de développement et d'évolution des applications multiplateformes. Cette question appelle, bien entendu, un grand nombre de réponses, qu'il s'agisse de solutions techniques destinées à structurer physiquement un projet multiplateforme, de solutions organisationnelles destinées à conduire un projet multiplateforme ou de solutions conceptuelles destinées à modéliser un projet multiplateforme. De fait, je me concentrerai essentiellement sur les méthodes conceptuelles et architecturales qui peuvent y apporter une réponse satisfaisante. Aussi, la problématique que je vais traiter tout au long de ce document est la suivante :

**« En quoi les architectures logicielles et intergicielles peuvent-elles simplifier
le processus de développement et d'évolution des applications
multiplateformes »**

Pour répondre de façon exhaustive à cette problématique, je vais tout d'abord présenter le développement multiplateforme de manière théorique et pratique, en décrivant le contexte actuel qui peut aujourd'hui amener une entreprise à réfléchir au multiplateforme. J'aborderai par ailleurs les enjeux qui y sont associés pour des entreprises.

Ensuite, je m'attacherai à décrire plusieurs niveaux de réflexion que l'on peut envisager pour apporter des réponses aux difficultés du développement multiplateforme. J'exposerai, pour l'exemple, quelques méthodes purement techniques et organisationnelles ainsi que les avantages et les limites associées à la réflexion propre à ces niveaux.

Puis, je développerai ma réflexion sur les architectures logicielles et intergicielles en exposant les méthodes et techniques qui y sont liées, ainsi que la façon dont elles peuvent servir le développement multiplateforme dans une optique de simplification. Leurs avantages et limites respectives seront alors exposés en détail.

Enfin, je traiterais plus spécifiquement de la téléphonie mobile en tant qu'univers multiplateforme à forte croissance et proposerais un cas pratique basé sur ma propre expérience. J'ai, en effet, effectué ma mission à l'Institut National de Recherche en Informatique et Automatique (INRIA) de Rocquencourt, au sein de l'équipe de recherche ARLES (ARchitecture Logicielle Et Systèmes distribués) dont les travaux portent, d'une part, sur les problématiques diverses liées aux systèmes logiciels distribués (comme, par exemple, la découverte, l'accès, la sécurité, la fiabilité, la composition et l'interopérabilité) et, d'autre part, sur la définition de nouvelles infrastructures logicielles (middlewares) pour les réseaux hybrides (réseaux sans fil, réseaux de capteurs, etc.).

Tout au long de ses activités de recherche, l'équipe ARLES a mis au point une technologie permettant à des téléphones mobiles de communiquer, d'échanger et de collaborer autour d'une infrastructure unifiée. Cette technologie, baptisée iBICOOP, est aujourd'hui transférée au sein d'une entreprise, Ambientic, pour servir au développement d'applications commerciales (actuellement dans le milieu de l'évènementiel). C'est dans ce contexte que j'ai été amené à prendre en charge la migration du middleware iBICOOP sur la toute récente plateforme Windows Phone 7 et à réfléchir aux difficultés posées par le développement multiplateforme. Il m'a paru important de profiter de l'expérience précédemment acquise par l'équipe car celle-ci a déjà dû faire face à une première écriture de l'application, tout d'abord compatible Android, Java Standard Edition et Java Mobile Edition, puis à la réécriture de celle-ci vers une plateforme fondamentalement différente, l'iPhone.

Aussi, je décrirais les caractéristiques et le contexte actuel de la téléphonie mobile pour ensuite mettre en lumière les problèmes forts que l'on peut y rencontrer en termes de développement multiplateforme. Puis, autour d'iBICOOP, je décrirais les différentes solutions et méthodes employées pour mener à bien le projet multiplateforme, sur la base de l'expérience de l'équipe, mais aussi par rapport à mes propres expérimentations théoriques et pratiques autour de Windows Phone 7.

2 - Le multiplateforme, pourquoi faire ?

A l'heure actuelle, quand la technologie Java est devenue la réponse facile à tous les problèmes de compatibilité entre systèmes, on peut effectivement s'interroger sur la pertinence des problématiques multiplateformes. Toutefois, nous allons voir comment le contexte technique actuel et à venir remet au goût du jour des problèmes que l'on considérait comme appartenant au passé.

2.1 - De l'hétérogénéité des environnements actuels

Contrairement aux années soixante, où l'informatique était réservée à une population réduite, nous connaissons depuis de nombreuses années une forte croissance du nombre de machines capables de calculer et de communiquer. Il y eut, tout d'abord, la diffusion massive des ordinateurs personnels (fixes, puis portables) puis l'expansion du marché de la téléphonie mobile avec l'ébullition que l'on connaît aujourd'hui autour des smartphones.

Parallèlement à cela, nous assistons à une forte démocratisation des périphériques embarqués et miniaturisés, toujours capables de plus de puissance et de connectivité : téléviseurs, caméras réseau, mini-serveurs, serveurs vidéos, unités de stockage réseau et même des cafetières. On compte au final plus d'un milliard de PC en activité dans le monde et au moins autant de périphériques embarqués. Toujours d'après la même source, ces valeurs devraient doubler d'ici 2013 (Gartner Group, 2008).

Cette progression rapide et l'apparition de périphériques embarqués de plus en plus performants amènent l'informatique à évoluer de manière globale (voir Fig. 1) vers de nouveaux schémas, notamment l'intelligence ambiante, c'est-à-dire la possibilité, pour un ensemble de petits périphériques, de communiquer entre eux de manière naturelle et transparente pour l'utilisateur, ce pour concourir à des objectifs particuliers comme, par exemple, la gestion d'une maison dans un cadre domotique.

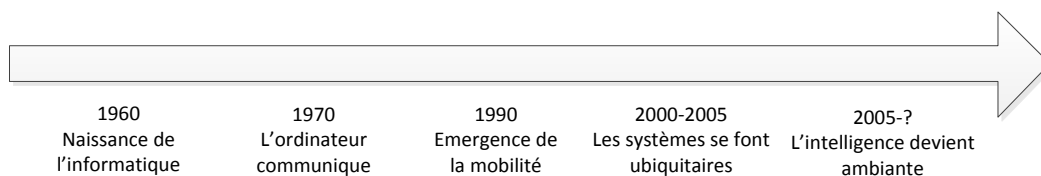


Fig. 1 – Evolution des ordinateurs vers des réseaux de plus en plus pervasifs, c'est-à-dire des réseaux d'ordinateurs capables de se découvrir et de communiquer entre eux sans intervention humaine.

L'accroissement massif du nombre d'ordinateurs sous-entend, ne serait-ce que par un simple jeu de probabilités, un accroissement proportionnel des plateformes incapables de communiquer entre elles. Ceci est d'autant plus flagrant dans le monde de l'embarqué où les ressources sont fortement limitées et où, de par le fait, les aspects de performance et d'efficacité vont être privilégiés à ceux de généricité et de portabilité.

Concrètement, donc, si le nombre de machines augmente, alors le nombre de plateformes potentielles augmente lui aussi, complexifiant d'autant les interactions entre les différents dispositifs qui doivent être non seulement capable de communiquer entre eux au travers de protocoles multiples, mais aussi proposer des applications similaires, compatibles et interopérables avec celles disponibles sur les autres systèmes. C'est typiquement le cas de la téléphonie mobile, où l'on va retrouver une même application d'un téléphone à un autre, voire même des équivalents sur ordinateur personnel.

Plus mitigé, le monde de l'entreprise est aujourd'hui partagé entre deux points de vue différents : centraliser ou disperser le système d'information.

Le premier prône la réunification des applications en un système d'information unique, à l'image d'une colonne vertébrale sur laquelle viendrait se greffer l'ensemble de l'entreprise. Ainsi, le système d'information central (souvent composé d'un mainframe^a) va servir de point d'accès à de nombreux terminaux n'étant plus réduits qu'à des fonctions d'entrées/sorties (affichage, saisie, etc.). Cette centralisation peut se faire au travers des progiciels de gestion intégrés^b (Enterprise Resource Planning) ou encore au moyen des infrastructures de virtualisation (virtualisation de poste, virtualisation d'application).

^a Ordinateur de très grande puissance de traitement, aussi appelé « ordinateur central ».

^b D'après le Grand Dictionnaire Terminologique, un progiciel de gestion intégrée est un logiciel qui permet de gérer l'ensemble des processus opérationnels d'une entreprise, en intégrant l'ensemble des fonctions de cette dernière comme la gestion des ressources humaines, la gestion comptable, financière, mais aussi la vente, la distribution, l'approvisionnement, le commerce électronique.

Le second privilégie les systèmes agiles et fortement décomposés, déployés au travers d'une multitude de systèmes bien distincts capables de communiquer entre eux. La gestion et la communication sont alors assurées par des politiques d'urbanisation qui garantissent la collaboration des systèmes entre eux, notamment au travers de mécanismes de communication spécifiques, comme les bus d'entreprises (ESB), l'orchestration de services ou le cloud computing qui, même s'il est vu par le client comme un grand système virtuel, s'appuie sur une structure fortement décentralisée.

Qui plus est, on assiste à un engouement pour les technologies JEE et .NET ainsi que pour les nombreux frameworks indépendants qui y sont associés. Cet accroissement du nombre d'outils destinés à améliorer la gestion (revue automatique, intégration continue, etc.) et à simplifier le processus de développement (frameworks, serveurs d'applications, etc.) finit par provoquer un effet inverse : les applications deviennent dépendantes de ces outils et ne peuvent plus être manipulées avec des outils équivalents.

Au final, les ressources exploitées sont diverses et variées et doivent continuer à communiquer entre elles (et même établir des communications inter-entreprises). Ces ressources, qu'il s'agisse de matériel ou de logiciel, définissent l'orientation d'un système : ils figent les concepts au travers de technologies difficilement interchangeables. C'est bien là tout le paradoxe, les technologies sensées nous apporter plus de simplicité ou de connectivité nous forcent à orienter les systèmes d'information vers une direction particulière. Ceux-ci finissent alors par se figer sans qu'un retour en arrière ne soit possible.

2.2 - La nécessité de s'ouvrir, d'être compatible, exhaustif et interopérable

Notre société actuelle est sans cesse plus friande de connectivité, tout en restant dans un contexte individualiste qui amène les technologies à se diversifier. De fait, les producteurs de matériels et de logiciels sont confrontés à un problème récurrent : comment concilier diversité et communication ?

2.2.1 - Qu'est-ce qu'une plateforme, la compatibilité, l'interopérabilité ?

La plateforme, une base de travail

En informatique, une plateforme peut être définie comme une base de travail, une solution technique. La plateforme est un support pour écrire, utiliser, développer ou lire un ensemble d'éléments logiciels. Par ailleurs, cette notion est souvent rapprochée de la notion d'environnement, c'est-à-dire l'ensemble des entités matérielles et logicielles qui vont permettre à d'autres programmes de s'exécuter.

La notion de plateforme n'est pas atomique, une plateforme pouvant être composée de plusieurs autres plateformes. Par exemple, lorsque l'on parle de « plateforme web », on peut tout aussi bien désigner le serveur web, le système d'exploitation qui l'exécute ou le matériel qui compose le serveur physique.

En cela, le terme plateforme est une sorte de mot fourre-tout pour désigner un système ou une infrastructure, comme pourrait l'être le terme « module » pour désigner un élément quelconque d'une architecture logicielle.

Une plateforme répond à un ensemble de besoins et désigne généralement un système complexe, critique ou vaste. De fait, les solutions apportées peuvent différer selon celui ou ceux qui l'apportent et plusieurs plateformes différentes peuvent donc répondre à des problèmes de manière totalement différente. Par exemple, il n'existe pas de système plus complexe et plus vaste qu'un système d'exploitation, aussi ceux-ci sont rarement compatibles entre eux alors que leur rôle est identique : transformer le matériel en élégantes abstractions (Tanenbaum, 2008).

De part ces définitions, toute solution technique peut être considérée comme une plateforme, selon le point de vue. On peut citer les exemples suivants :

- **une plateforme matérielle**, c'est-à-dire un ensemble de composants destinés à rendre un service (le processeur, le module de communication, l'ordinateur, etc.).
- **une plateforme d'exécution**, c'est-à-dire un système destiné à exécuter des programmes (système d'exploitation, machine virtuelle).
- **un langage**, c'est-à-dire une manière de donner des instructions à la machine pour réaliser nos programmes.
- **un composant logiciel**, c'est-à-dire un ensemble d'éléments logiciels destinés à accomplir certaines tâches (bibliothèque, kit de développement, interface de programmation, framework, etc.). Ce composant logiciel peut être distribué, c'est-à-dire transformé en une application exécutable exposant dynamiquement des fonctionnalités (bibliothèque dynamique, tiers, service, middleware, serveur, etc.).
- **un format**, c'est-à-dire une manière de représenter et/ou de stocker des données de nature diverses (format de fichier, format d'échange, protocole de communication, etc.).

On remarquera, par ailleurs, que ces différentes plateformes n'ont pas le même impact sur le projet. Par exemple, une plateforme matérielle ou une plateforme d'exécution va avoir une incidence sur la totalité du projet tandis qu'un composant logiciel peut ne concerner qu'une partie du même projet (un système de gestion de base de données, par exemple).

La compatibilité

Deux systèmes *A* et *B* sont dits compatibles s'ils sont capables de communiquer ensemble, ce qui revient à dire que *A* et *B* ont connaissance des interfaces qui leur permettent d'échanger des informations.

Il ne s'agit pas forcément de communication réseau, mais simplement d'échanges de données. Par exemple, on peut dire que la suite bureautique Microsoft Office est compatible avec les formats de fichier OpenDocument exploités par la suite bureautique OpenOffice.org car elle est capable de les traiter^a.

^a Même si la qualité de ce traitement est, dans les faits, très largement discutable.



Fig. 2 – La compatibilité (le système 1 est compatible avec le système 2 et le système 2 est compatible avec le système 3)

Le standard et la norme

Un système *A* est dit « standard » ou « standard de fait » lorsque sa domination est telle que les autres acteurs du domaine font en sorte d’être forcément compatibles avec lui. Il s’agit simplement de la compatibilité, mais vis-à-vis d’un élément dominant. Par exemple, le format de fichier d’image GIF

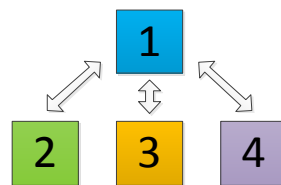


Fig. 3 – Le standard (le système 1 est un standard, et les autres sont compatibles avec lui)

Une norme est un nouveau système issu de la maturation du domaine, c’est-à-dire composé du meilleur de chaque standard. La norme a pour but d’unifier le domaine en nivelant par le haut, là où le standard s’est imposé naturellement sans forcément être le meilleur. Généralement on voit apparaître un ensemble de standards puis, au bout d’un certain temps, un ensemble de normes.

L’interopérabilité

Deux systèmes *A* et *B* sont dits interopérables s’ils sont capables de communiquer ensemble au travers d’un médium standardisé ou normalisé. Contrairement à la compatibilité où chaque système a connaissance des autres, l’interopérabilité impose seulement à chaque système d’avoir connaissance du médium.

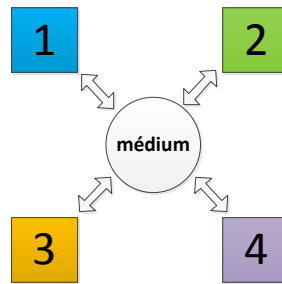


Fig. 4 – L'interopérabilité

2.2.2 - La nécessité d'une stratégie d'ouverture

Aujourd'hui, même si l'interopérabilité est un mot à la mode, il n'en est pas moins issu d'une saine volonté de simplification des échanges entre des logiciels de plus en plus variés. En effet, la fermeture des standards et la restriction à des plateformes spécifiques ont montré leurs limites dans le contexte de diversification et de granularisation des systèmes actuels. Même si aujourd'hui les unités clientes tendent à se vider de leur substance dans le monde de l'entreprise (regain d'intérêt pour les mainframes avec la baisse des coûts, les clients légers, la virtualisation d'applications, etc.) leur nature, elle, tend à se diversifier (téléphone portable, netbook, poste fixe, navigateur). C'est pour cela qu'aujourd'hui, les concepteurs de plateformes, qu'elles soient destinées au monde de l'entreprise ou au grand public, mettent en place ou préparent la mise en place d'une stratégie d'ouverture.

D'après la législation française, « on entend par standard ouvert tout protocole de communication, d'interconnexion ou d'échange et tout format de données interopérables et dont les spécifications techniques sont publiques et sans restriction d'accès ni de mise en œuvre » (Fr. Gov., 2004). Cette définition englobe donc à la fois des protocoles, des formats de fichier, des formats d'échange, des modèles, des représentations de données, etc.

Aussi la stratégie d'ouverture s'axe autour de ces standards ouverts. Cependant leur multiplication tend à avoir un effet collatéral plutôt négatif : la multiplication des standards ouverts sous-entend parfois de faire un choix quant aux standards ouverts supportés par une application. Encore une fois le problème est reporté à des couches d'abstraction supérieures, mais demeure toujours le même. De fait, les standards ouverts deviennent autant de plateformes spécifiques qu'il est nécessaire de prendre en charge.

2.2.3 - Pourquoi cibler plusieurs plateformes à l'heure actuelle ?

Nous avons vu précédemment que les plateformes tendaient à se diversifier avec l'apparition des environnements ubiquitaires. A l'avenir, l'individu ne sera plus associé à une ou deux machines uniques, mais à un ensemble de petits périphériques relativement peu coûteux, mais suffisamment puissant pour communiquer et disposer de comportements complexes (ce que l'on appellera le « personal cloud »). Mark Weiser, le père de l'informatique ubiquitaire, en vient même à prédire^a que l'informatique que nous connaissons à l'heure actuelle disparaîtra au profit d'espaces informatisés, c'est-à-dire d'une virtualisation de l'ensemble des périphériques qui composent notre quotidien.

Ces systèmes, bien qu'interopérables, disposeront de fonctionnements bien particuliers, comme il est déjà possible de le voir avec les smartphones et leurs systèmes d'exploitation généralement peu compatibles entre eux. Cette diversification des périphériques impactera même les entreprises qui centralisent leur patrimoine applicatif car même si les terminaux sont vidés de leur substance, leurs natures évolueront dans des directions tellement nombreuses qu'elles finiront forcément par sortir du cadre général établi par cette centralisation (petits écrans, faible connectivité, ressources réduites, etc.), ce qui nécessitera de décliner les applications centralisées en plusieurs versions, adaptées à des types de terminaux prédéfinis. C'est une pratique déjà courante, notamment pour les applications web, celles-ci étant souvent déployées en plusieurs versions : ordinateur, téléphone mobile, etc.). On trouve même des applications s'adaptant directement à la famille de navigateur, prenant en compte l'ensemble de ses spécificités techniques (bugs connus, accélération matérielle, mode texte, etc.).

Aussi, dans ces environnements de plus en plus distribués, les concepteurs d'applications devront considérer toujours plus de plateformes différentes pour pouvoir conserver une certaine exhaustivité. Toujours dans le cadre de la téléphonie mobile, comment justifier la présence d'une application sur un type de téléphone bien particulier et pas sur les autres ? Autant il est possible de cibler une grande part de marché en se restreignant aux ordinateurs personnels équipés de Windows (3/4 du marché grand public), autant il devient plus difficile

^a Même si, dans le domaine informatique, tout le monde a tendance à prédire tout et n'importe quoi et que, chaque semaine, on nous annonce la fin de l'informatique telle que nous la connaissons.

de se contenter d'un seul système d'exploitation mobile cible étant donné le partage actuel du marché. L'explosion du nombre de plateformes fait apparaître le lien entre exhaustivité et visibilité : pour être représentatif, le logiciel doit être présent partout, fonctionner sur plusieurs plateformes (et si possible avec le même jeu de fonctionnalité), être capable de communiquer avec l'ensemble de ses concurrents, etc.

Il n'est plus possible aujourd'hui de considérer une plateforme unique comme cible de développement alors que les besoins en ouverture se font sentir même dans les systèmes réputés comme historiquement très fermés. C'est pour cela que le nombre de projets multiplateforme augmente significativement aujourd'hui alors qu'il était en baisse depuis de nombreuses années.

2.3 - La complexité organisationnelle et technique induite par le multiplateforme par rapport aux projets classiques

Avant de considérer les projets multiplateformes, il paraît intéressant de se pencher sur les projets classiques pour mieux cerner les particularités et similitudes respectives. En effet, il ne faut pas oublier qu'un projet classique, que nous pourrions appeler projet monoplatforme, est un projet multiplateforme particulier où le nombre de plateformes est égal à un.

Bien qu'il existe de nombreuses méthodologies de projet, il s'agit généralement d'une adaptation d'un modèle historique, le cycle en V (Fig. 5). Celui-ci est très souvent intégré au sein d'un processus itératif, ce qui revient à découper le projet en une succession de petits sous-projets suivant leur propre cycle en V et aboutissant à un résultat partiel, mais exploitable (une release).

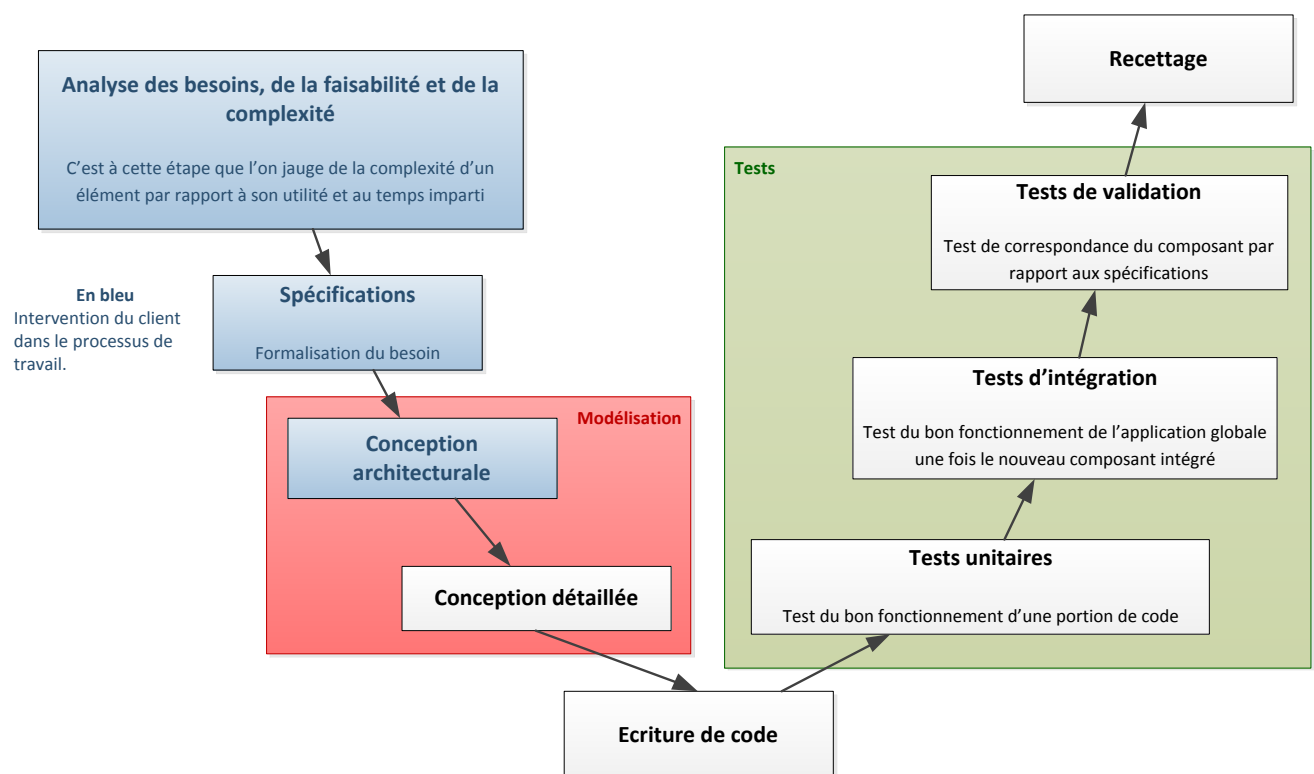


Fig. 5 – Le cycle en V

Cette suite de petits cycles s'inscrit dans un contexte plus vaste impliquant un ensemble d'étapes composant un cycle de vie global :

- **l'avant-vie**, comprenant les études de faisabilité, la phase d'inception, le cahier des charges et les spécifications globales.
- **le développement et les tests**, tels que nous les avons évoqués plus haut, avec les successions d'itérations.
- **la vie**, c'est-à-dire la mise en production et la maintenance applicative.
- **l'après-vie**, qui représente l'évolution future du logiciel, si celle-ci existe.

Les projets multiplateformes suivent ces schémas, mais induisent une complexité supplémentaire due à certaines particularités que je vais exposer dans les chapitres qui vont suivre.

2.3.1 - La nécessité de raisonner de manière multidimensionnelle

Les projets classiques se contentent de suivre leurs cycles de développement en tentant de parvenir à maîtriser à la fois les coûts, la qualité et les délais. Il s'agit d'une progression linéaire, ou d'une suite de progressions linéaires parallélisées mais toujours rigoureusement distinctes (une tâche t n'est jamais réalisée deux fois).

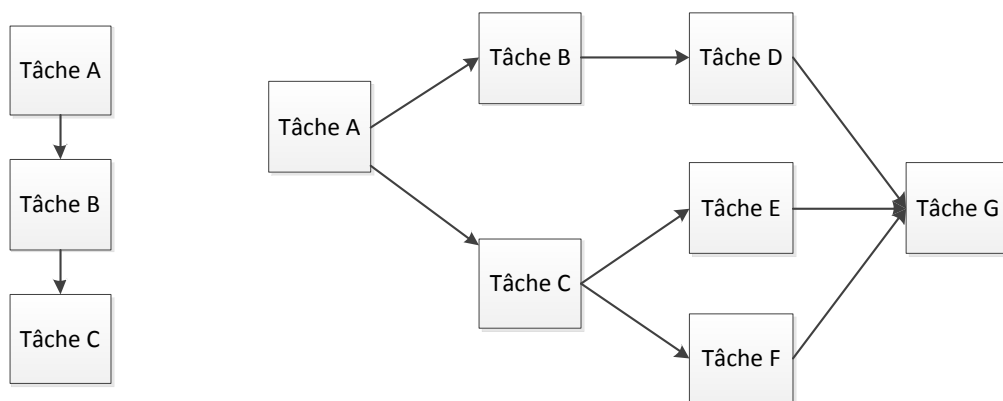


Fig. 6 – Une progression linéaire (à gauche) et un ensemble de progressions linéaires parallélisées (à droite)

Dans un projet classique, le processus de développement n'est destiné qu'à des plateformes connues et uniques (une seule plateforme d'exécution, un seul framework, un ensemble fixe de bibliothèques, etc.). Chaque tâche dans le projet est plus ou moins distincte des autres et peut être associée à une charge plus ou moins évidente (mais néanmoins quantifiable au travers de nombreuses méthodes qu'il ne m'appartient pas de détailler ici), elle-même pouvant être répartie entre les développeurs.

Dans le cas du multiplateforme, notre projet va être réparti sur plusieurs plateformes similaires (plusieurs librairies pour une même chose, plusieurs plateformes d'exécutions, etc.) ce qui revient à gérer virtuellement autant de projets que ce qu'il y a de plateforme cible. Chacun de ces projets est indépendant par nature, mais doit s'inscrire dans une gestion de projet unifié puisqu'il ne s'agit que d'un seul projet d'un point de vue métier. Ainsi, nous avons affaire à un aspect multidimensionnel qui n'existait pas dans notre projet classique. Le projet multiplateforme s'identifie à un ensemble de projets dont les tâches et les représentations métiers sont identiques, mais où les tâches plus techniques peuvent être fondamentalement différentes. Ainsi, la charge (ou tout autre paramètre) associée à une tâche t_1 destinée à la plateforme c_1 est différente de la charge associée à une tâche t'_1 destinée à la plateforme c_2 . Pire encore, t_1 peut ne pas avoir d'équivalence sur la plateforme c_2 ou, au contraire, avoir plusieurs équivalences (tâche découpée différemment). Ainsi, nos différents projets ne représentent pas forcément la même charge totale, sans oublier que ceux-ci ne vont peut-être pas disposer des mêmes ressources humaines et financières.

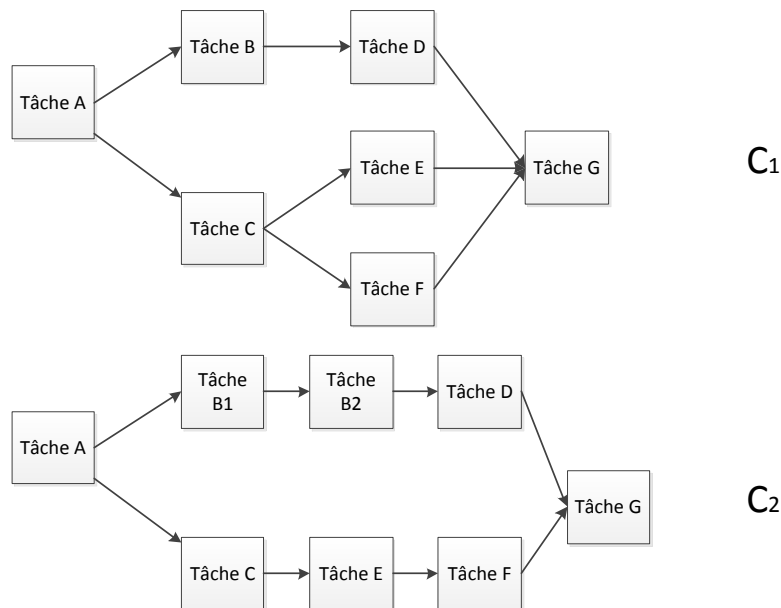


Fig. 7 – Exemple de double découpage pour un projet multiplateforme à deux plateformes

Ainsi, avec le multiplateforme, la taille des équipes risque d'augmenter car le besoin en compétences techniques est accru (nécessité de disposer ou d'acquérir des connaissances sur l'ensemble des technologies). De même, le besoin en organisation (gestion de projet,

suivi, outils de support, etc.) va augmenter lui aussi, puisqu'il existe désormais un plus grand nombre de projets à gérer.

2.3.2 - Des difficultés pour gérer le développement et l'évolution

L'aspect multidimensionnel, principal problème posé par le multiplateforme, fait apparaître de nouvelles difficultés dans la plupart des étapes du cycle de vie de l'application.

Les spécifications

Tout d'abord, les spécifications doivent tenir compte des spécificités inhérentes à chaque plateforme, c'est-à-dire l'ensemble des fonctionnalités supportées respectivement par chaque plateforme. En effet, l'ensemble de fonctionnalités F_1 supportées par la plateforme c_1 n'est pas forcément le même que l'ensemble de fonctionnalités F_2 supportées par la plateforme c_2 . Ces deux ensembles peuvent être identiques ($F_1 = F_2$), partiellement compatibles ($F_1 \cap F_2 \neq \emptyset$) ou incompatibles ($F_1 \cap F_2 = \emptyset$). De plus, une fonctionnalité de F_1 peut disposer d'un équivalent dans F_2 sans que leurs fonctionnements soient identiques (par exemple, deux plateformes peuvent supporter le copier-coller, mais pas de la même façon). Au final, la phase de spécification va tenter de déterminer si les ensembles de fonctionnalités de chaque plateforme du projet sont équivalents ou partiellement compatibles, ce pour s'assurer que chaque plateforme est en mesure de supporter l'ensemble des fonctionnalités nécessaires à l'accomplissement du projet. Par exemple, il est inutile de songer à développer un serveur de fichier sur une plateforme sans support de stockage.

La conception

Tout comme la phase de spécification, les phases de conception doivent tenir compte des fonctionnalités supportées par les différentes plateformes (c'est-à-dire les fonctionnalités qui disposent d'au moins une équivalence sur chaque plateforme). Les spécifications vont retenir quelles fonctionnalités seront mises en œuvre tandis que la conception va permettre de gérer les équivalences entre fonctionnalités. En effet, même si les fonctionnalités d'une plateforme c_1 possèdent des équivalences sur la plateforme c_2 , les différences probables dans leur fonctionnement (entrées/sorties, traitements, états, etc.) vont devoir être gérée. Par exemple, c'est la conception qui va permettre de définir une architecture adaptée pour

la gestion de plusieurs types de bases de données. Par la suite, lors de l'exécution, l'application pourra ainsi gérer des bases de données spécifiques à chaque plateforme.

C'est tout particulièrement dans cette phase que l'on va abstraire les spécificités inhérentes à chaque plateforme. On va pouvoir, au travers de l'architecture, tenir compte des multiples façons de faire la même chose. Cependant, il peut y avoir une limite à cela car, dans le pire des cas, l'abstraction ne permettra de représenter que les fonctionnalités communes à chaque plateforme, c'est-à-dire les fonctionnalités qui possèdent une équivalence sur chaque plateforme. On peut toutefois envisager des versions du projet qui supporteraient ou ne supporteraient pas certaines fonctionnalités en fonction des spécificités des plateformes. Cela ne peut fonctionner, cependant, que pour des fonctionnalités non indispensables à la bonne marche du projet ; on peut effectivement se passer d'une fonctionnalité d'agenda sur un téléphone, mais pas d'une fonctionnalité permettant d'envoyer de passer des appels.

On voit donc apparaître, avant même la phase de production logicielle, des disparités entre les plateformes.

La production

L'écriture de code est la phase la plus impactée par le problème multiplateforme. En effet, il faut tout d'abord considérer que pour chaque plateforme il est nécessaire de disposer des compétences associées pour mener à bien une tâche de développement sous peine d'accroître significativement les risques liés à la découverte technologique (problèmes découlant d'un manque de connaissances techniques vis-à-vis de la plateforme ciblée).

Ensuite, le processus d'écriture de code va consister à transformer les spécifications et les modèles issus de la conception en code source, tout en tenant compte, donc, des subtilités de chaque plateforme. Cependant, d'une plateforme à une autre, la tâche de développement ne représente pas forcément la même complexité et la même charge (dans le meilleur des cas, celles-ci seront identiques, mais ce cas idéal n'est que très rarement rencontré en pratique). De fait, les indicateurs de progression doivent être spécifiques à chaque plateforme, ce qui complexifie le suivi des différents projets.

Les tests associés au processus de développement sont rarement identiques d'une plateforme à une autre. En effet, comme les tâches et les fonctionnalités varient (bien que celles-ci restent équivalentes), les tests ne peuvent pas être transformés tels quels. Bien souvent, on utilise un framework de test bien particulier qu'il sera difficile de retrouver ailleurs et il faudra tenir compte des prérequis nécessaires à l'exécution du test. Ceux-ci peuvent, effectivement, varier en fonction des technologies utilisées (on peut avoir besoin d'initialiser certaines choses sur une plateforme et pas sur une autre, par exemple), tout particulièrement lorsque l'on travaille dans un contexte distribué.

La complexité augmente d'autant si l'infrastructure de test doit permettre leur exécution automatique, comme c'est le cas avec l'intégration continue^a. Malheureusement, même le système d'intégration continue est lié à des contraintes multiplateformes pour pouvoir fonctionner et, de plus, il n'est pas rare qu'il se limite aux tests associés à un langage spécifique.

Les tests unitaires et les tests d'intégration vont permettre de mettre en évidence des bugs logiciels. Ces bugs logiciels sont détectés au fur et à mesure du développement du projet et sont reportés quelque part en attendant leur correction ultérieure. Dans le cadre multiplateforme, l'aspect multidimensionnel vient complexifier cela car les anomalies qui vont apparaître seront forcément liées à une plateforme spécifique. Il est donc nécessaire de prendre en charge des processus de gestion des erreurs pour chaque plateforme.

Plus grave, les tests de validation vont mettre en lumière d'éventuelles erreurs de conception ou de spécification et il sera alors nécessaire de mettre à jour la totalité des sous-projets. La communication devient donc d'autant plus complexe.

La maintenance et l'évolution

Produire une application pour plusieurs plateformes est une tâche fastidieuse, mais qui peut malgré tout être menée à bien manuellement. La conception et les spécifications desquelles elle est issue vont diriger les différents sous-projets associés aux plateformes. De fait, chaque sous-projet parviendra, moyennant une certaine rigueur et un effort non

^a L'intégration continue consiste à vérifier automatiquement et périodiquement que les nouvelles modifications sur le code source ne provoquent pas de régression, c'est-à-dire que l'ensemble des tests unitaires déjà écrit passe toujours après réexécution.

négligeable, à un résultat exploitable (bien que probablement assez peu cohérent). Des difficultés importantes vont alors survenir avec la maintenance corrective. En effet, comme lors des phases de tests, des bugs vont être détectés par les utilisateurs et seront reportés à leur tour. De fait, la maintenance doit continuer à prendre en charge les processus distincts de gestion des erreurs et de leurs corrections pour ne pas laisser pourrir une version particulière du logiciel.

Avec le temps, l'application va être amenée à évoluer, pour des besoins nouveaux ou pour des considérations mercatiques. On va alors se retrouver avec un nouveau cycle de difficultés liées aux phases de développement.

Bien entendu, en cas de budget ou de délais réduits, les efforts vont se concentrer sur les plateformes les plus exploitées et un décalage important va apparaître entre les différentes versions du logiciel, ce qui n'est pas souhaitable dans l'absolu car il peut arriver que les applications sur chaque plateforme ne soient plus compatibles entre elles, ce qui est le pire résultat imaginable.

2.3.3 - La démultiplication de la charge et des coûts

Au final, nous devons retenir un certain nombre de difficultés liées au multiplateforme. Les principales peuvent se traduire ainsi :

- le besoin en compétence et en organisation est accru proportionnellement au nombre de plateforme. De fait, la taille des équipes de travail augmente elle aussi.
- l'organisation est plus complexe, car il faut gérer plusieurs projets parallèles qui, au final, font la même chose d'un point de vue client.
- en terme de code source, plusieurs choses sont réécrites plusieurs fois, à quelques différences près.
- les processus de tests, de gestion des anomalies et de maintenance sont multipliés par le nombre de plateformes.
- les applications peuvent présenter un fort décalage fonctionnel d'une plateforme à une autre, ce décalage ayant tendance à s'accroître avec le temps.

Ces différents effets indésirables ont un impact important sur la charge de travail, les délais et les coûts, à cause de la réécriture permanente et de l'organisation dispersée. Les facteurs

d'échec sont fortement majorés par la complexité organisationnelle, en partie due à la taille plus importante des équipes.

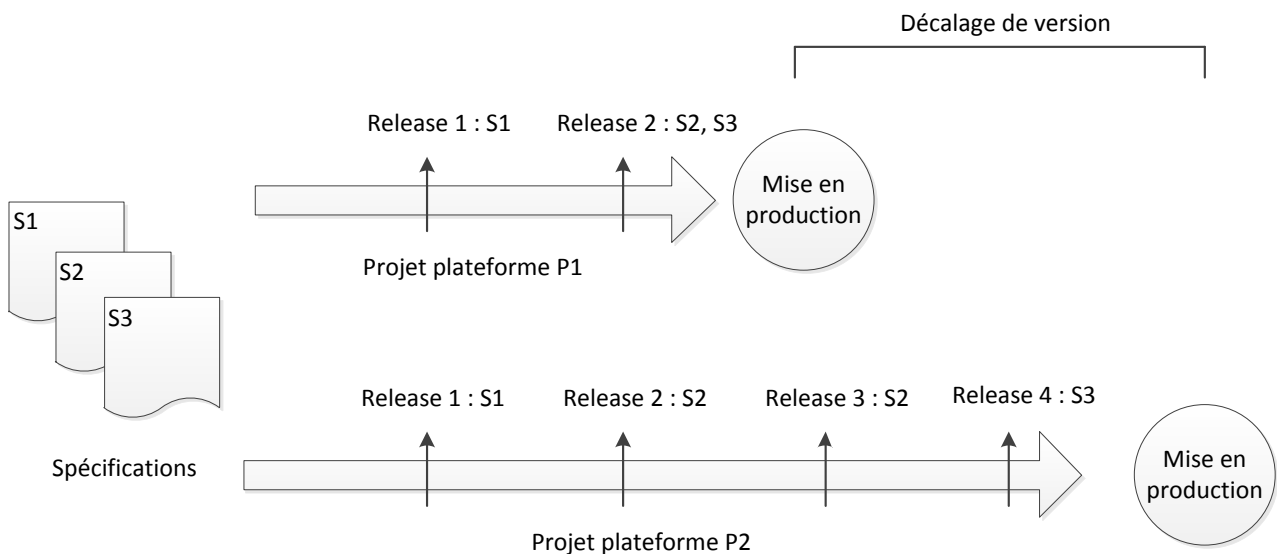


Fig. 8 – Illustration du décalage entre deux versions

Le décalage temporel (version en avance sur une autre) et fonctionnel (versions ne supportant pas le même jeu de fonctionnalités) qui peut survenir entre les versions ainsi que les multiples processus de gestion et d'organisation (tests, détection et prise en charge des erreurs, évolution du logiciel) accroissent eux aussi la difficulté à maintenir et à faire évoluer les applications sur toutes les plateformes simultanément.

2.4 - Au final, quels enjeux pour le multiplateforme ?

Nous avons vu que le développement d'application multiplateforme pose des difficultés supplémentaires à tous les niveaux d'un cycle de développement standard. Aussi, il paraît judicieux de s'interroger sur l'intérêt de porter une application sur telle ou telle plateforme en fonction de ce que l'on veut cibler.

Malgré les difficultés, plusieurs raisons peuvent conduire à envisager un développement multiplateforme, qu'il s'agisse d'une stratégie d'insertion, d'une nécessité environnementale, d'une recherche d'exhaustivité ou d'un choix éthique. Aussi, pour répondre à ces besoins, le multiplateforme doit être pris en charge au travers de plusieurs aspects :

- **la partie conception et développement**, en proposant des langages et des architectures uniques et portables, exploitables sur plusieurs types de plateforme et capables de prendre en charge les aspects génériques des applications.
- **la partie outil**, en proposant des outils pour accompagner le cycle de développement en ce qui concerne, par exemple, la gestion des versions, le report de dysfonctionnements, la revue de code ou les tests automatiques.
- **la partie organisation**, en proposant des processus et des outils de planification et de gestion des différentes versions d'une même application, tout en garantissant la cohérence des différentes versions (fonctionnalités et comportements identiques, mesure de la stabilité, etc.).

La prise en charge du multiplateforme au travers d'outils adaptés va permettre d'abstraire l'aspect multidimensionnel. De fait, les risques de dérives vont être réduits ainsi que les surcoûts liés aux ressources humaines supplémentaires.

C'est cet objectif qui définit les enjeux majeurs de ma réflexion sur le multiplateforme. Comment permettre au processus multiplateforme de production logicielle de s'accomplir de la même façon que sur un projet classique (monoplateforme) ? En répondant à cette question ou du moins en extrayant des techniques et des procédés capables de réduire le nombre de dimensions possibles dans un projet multiplateforme, tous les problèmes liés à cette pluridimensionnalité disparaissent ou, du moins, s'estompent.

3 - Comment simplifier l'approche multiplateforme ?

Nous avons pu voir jusqu'à présent quelles étaient les difficultés posées par un contexte multiplateforme en termes de développement et d'organisation. Toutefois, les enjeux associés méritent que l'on s'intéresse à des méthodes pour minimiser ces difficultés et accroître l'efficacité de la production logicielle selon trois grands angles de réflexion :

- **Organisation** : comment organiser et gérer le développement multiplateforme ?
- **Conception** : comment l'architecture peut-elle permettre une meilleure réutilisabilité et une simplification du processus de développement multiplateforme ?
- **Technique** : quelles sont les solutions techniques pour gérer des ensembles de codes sources destinés à des plateformes différentes ?

3.1 - Pas de solution évidente, mais un éventail de possibilités

Les trois différents points de vue exposés précédemment s'efforcent d'apporter des réponses qui leur sont spécifiques. Bien entendu, trois façons d'aborder le problème sous-entendent des avantages et des inconvénients respectifs. En effet, le multiplateforme concerne l'ensemble des acteurs de la production logicielle, qu'il s'agisse de producteurs (développeurs, testeurs, architectes, analystes) ou de managers (chefs de projet), et chacun entrevoit des solutions propres à ses perspectives. De fait, il n'existe pas une réponse unique pour pallier aux difficultés du multiplateforme, mais plusieurs, à sélectionner en tout ou partie en fonction de ce qui est voulu.

3.2 - Au sein même du processus de développement, des méthodes existent

Nous allons commencer par détailler les solutions communes envisagées par le point de vue technique, du plus simple vers le plus complexe. L'objectif de ce point de vue est d'apporter des réponses qui s'intègrent dans les considérations techniques propres au développement logiciel.

3.2.1 - Exploiter un code source de référence, une solution fastidieuse

L'approche naïve du développement multiplateforme consiste à concentrer les efforts de développement et d'organisation sur une plateforme de référence que nous noterons c_r . Depuis les spécifications du projet, une première version du code source de référence va être rédigée. Une fois testé et validé, ce code va être réécrit manuellement (ou

automatiquement, comme nous l'envisagerons par la suite) pour l'ensemble des autres plateformes par les programmeurs compétents.

De fait, lorsque les spécifications vont être modifiées, c'est tout le processus de génération qui est réitéré : modification du code source de référence, test, validation et production des codes sources.

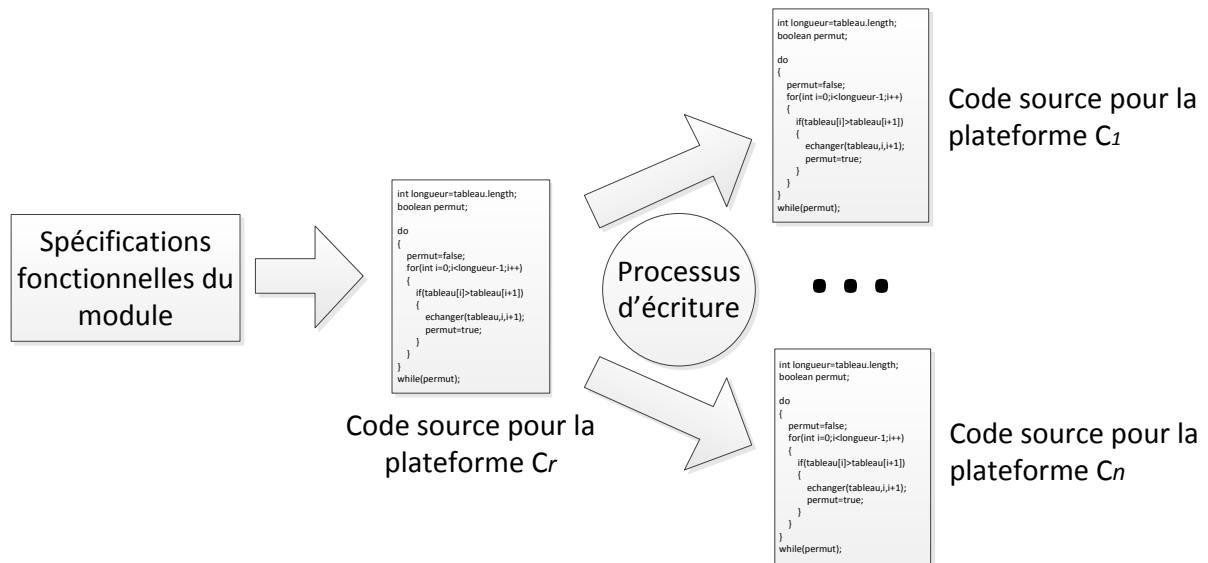


Fig. 9 – Processus simplifié de transformation d'un code source de référence

Cette approche a pour avantage d'être simple à comprendre et à appliquer. Elle supprime l'aspect multidimensionnel en concentrant l'effort de développement sur la plateforme c_r et en effectuant, une fois le code source de c_r produit, de simples transformations vers les autres plateformes. Malgré tout, ce processus se révèle fastidieux et rapidement limité.

Tout d'abord, le processus induit des retards sur les plateformes autres que c_r puisque le travail de portage ne peut pas être effectué sur ces plateformes tant que le code associé à c_r n'est pas validé (manuellement, au travers de tests classiques, ou formellement au travers d'une validation automatique de son modèle). Selon la quantité de code source à réécrire, le décalage peut devenir significatif, notamment si le projet n'a pas été suffisamment découpé dès le départ.

Ensuite, il est rare qu'un code source soit totalement autonome, celui-ci faisant généralement appel à des composants externes (bibliothèques, frameworks, kits de développement, etc.) qui sont généralement des boîtes noires^a pour le programmeur. Ces composants, qui sont eux-mêmes des plateformes, peuvent être compatibles avec c_r , mais il n'est pas garanti qu'ils soient déclinés pour l'ensemble des autres plateformes. C'est alors au programmeur qu'incombe la tâche de trouver d'autres composants ayant des fonctionnalités similaires ou, dans le pire des cas, d'écrire les siens. De fait, la réécriture du code source ne représente pas la même charge selon la plateforme où l'on se situe, comme nous l'avons évoqué précédemment.

Enfin, ce processus ne permet pas de prendre en charge les bogues spécifiques à chaque plateforme. En effet, toutes les erreurs de conception (problèmes de spécifications) ou les erreurs d'implémentation du code source de référence vont se répercuter sur les codes sources associés aux autres plateformes sans que l'inverse soit vrai (voir Fig. 10). Ainsi, des défauts spécifiques peuvent apparaître sur les plateformes autres que c_r et ceux-ci doivent être pris en charge séparément.

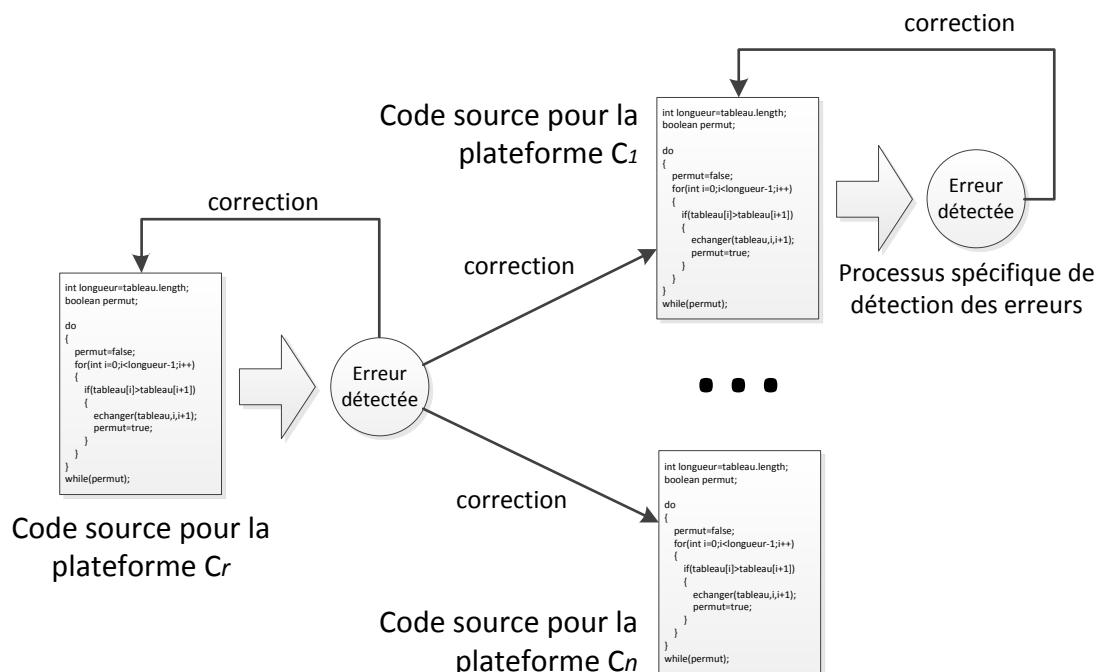


Fig. 10 – Répercussion des erreurs

^a Composant dont on connaît le rôle, les entrées et les sorties, mais pas le fonctionnement interne.

3.2.2 - Un mécanisme historique : le preprocessing

Le premier dispositif utilisé dans le cadre du développement multiplateforme est la compilation conditionnelle, c'est-à-dire la compilation de telle ou telle portion de code en fonction d'instructions conditionnelles fournies par le programmeur. Cela se traduit par la capacité du programmeur à définir des blocs de code spécifiques à chaque plateforme, ces blocs de code étant sélectionnés ou ignorés par le compilateur lors de la transformation du code source.

C'est ce que permet de faire, entres autres, l'opération de preprocessing. D'une manière générale, ce terme désigne une opération destinée à préparer des données en vue de leur utilisation par d'autres mécanismes. Il s'agit concrètement d'un prétraitement automatique réalisé par un élément spécifique : le préprocesseur.

Au-delà de cette définition très générale, on désigne souvent par preprocessing l'opération de transformation du code source précédant la phase de compilation ou d'interprétation.

On distingue deux grandes familles de préprocesseurs : les préprocesseurs syntaxiques et les préprocesseurs lexicaux. Les premiers permettent d'effectuer des transformations sur la syntaxe d'un langage en fonction de règles spécifiées par le programmeur et répondent à des besoins qui ne concernent pas le développement multiplateforme, aussi je n'épiloguerai pas sur leurs rôles. Les préprocesseurs lexicaux, au contraire, apportent toutes les fonctionnalités nécessaires pour répondre aux besoins de compilation conditionnelle. Très simples, ces préprocesseurs ne font que transformer du texte à partir de directives intégrées au code source. Concrètement, le préprocesseur lexical va pouvoir insérer des morceaux de code source selon les directives qui lui sont fournies (inclusion d'un fichier source externe, recopie de la valeur d'une constante, etc.) ce qui va permettre au programmeur de définir des portions de code qui ne seront compilées que si certaines conditions sont réunies.

Par exemple, le processus de transformation d'un code source écrit en langage C vers un code machine exécutable comprend une phase de preprocessing (Fig. 11 et Fig. 12) lexical.

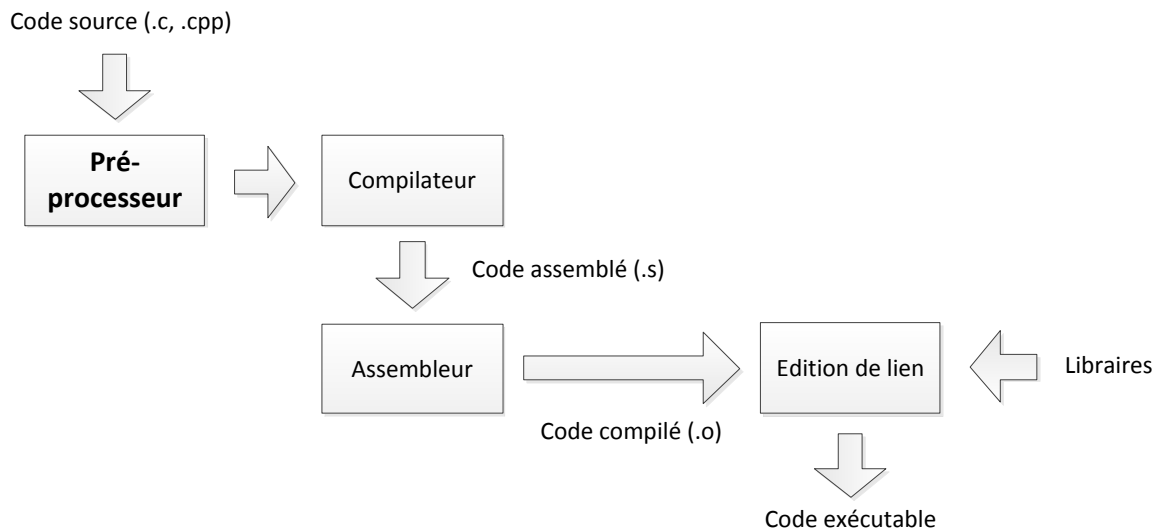


Fig. 11 – Les étapes de production d'un code exécutable par le compilateur Gnu Compiler Collection (GCC), à partir d'un code source écrit en C

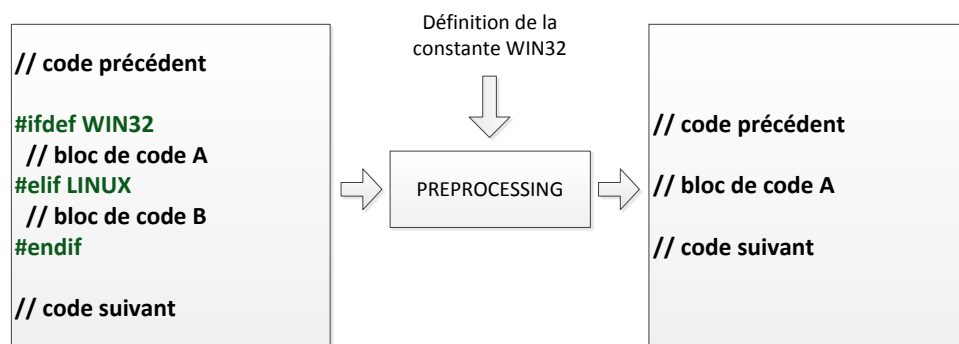


Fig. 12 – Le mécanisme de preprocessing en C (exemple)

De par les propriétés de compilation conditionnelle du préprocesseur lexical, c'est-à-dire sa capacité à injecter ou à sélectionner du code source avant la compilation en fonction de conditions extérieures, le programmeur peut écrire un code source compilable sur plusieurs plateformes, à partir du moment où le langage de base est supporté sur ces plateformes.

Cette méthode, très simple, permet donc d'effectuer des opérations de compilation conditionnelle, mais présente un certain nombre d'inconvénients. Tout d'abord, l'utilisation des préprocesseurs a un impact sur le code et sa lisibilité. Celui-ci est alourdi par les instructions spécifiques et se retrouve surchargé par les détails techniques des différentes plateformes (voir Fig. 12).

Ensuite, un préprocesseur se limite généralement à un seul langage bien que l'on puisse imaginer un préprocesseur générique capable d'effectuer les transformations de code sur plusieurs langages (voir Fig. 13 ci-dessous).

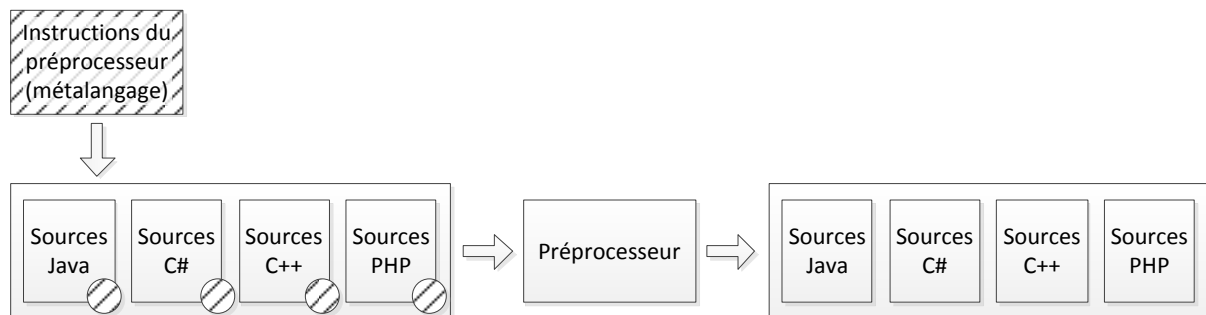


Fig. 13 – Préprocesseur générique capable de manipuler des instructions intégrées dans le code source de plusieurs langages

Aussi, le mécanisme de preprocessing tend à disparaître aujourd'hui au profit de l'introspection, un mécanisme plus puissant que nous verrons plus loin.

3.2.3 - Les composants portables, une méthode très appréciée

Nous avons vu précédemment que les problèmes de réécriture de code étaient liés en partie à l'utilisation de composants externes. En effet, il n'est pas possible de garantir qu'un composant externe *C* développé pour une plateforme *A* va fonctionner sur une plateforme *B*. De même, il n'est même pas possible de garantir qu'un composant *C'* équivalent à *C* soit disponible pour la plateforme *B*.

Cette particularité est due au fait que les composants eux-mêmes sont des plateformes, c'est-à-dire des solutions techniques spécifiques.

Pour tenter de résoudre ce problème, il existe des composants dits « portables » dont la réécriture a déjà été effectuée par leur auteur pour un ensemble de plateformes définies. On dit alors que le composant est supporté^a par ces plateformes. Concrètement, l'effort de gestion des aspects multiplateformes et des équivalences entre les fonctionnalités offertes par celles-ci a simplement été déchargé sur le fabricant de la plateforme. Ainsi, ce n'est plus le développeur ou le concepteur de l'application qui doit gérer l'ensemble des alternatives

^a Par exemple, le kit de développement Qt, destiné à la fabrication d'interfaces graphiques, est supporté par les plateformes d'exécutions Windows, Linux et Mac OS (entre autres).

(ainsi que la gestion des erreurs et des évolutions qui y sont associées), mais bel et bien les développeurs et les concepteurs du composant.

Ainsi, le composant portable a un avantage non négligeable : il permet d'abstraire les spécificités liées aux plateformes pour lequel il a été porté.

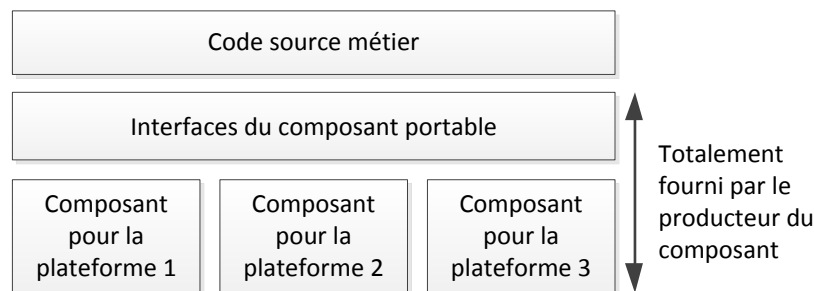


Fig. 14 – Une première forme d'abstraction des spécificités liées aux plateformes

Cette première forme d'abstraction se heurte toutefois à des contraintes proches de celles du préprocesseur. Les composants portables sont souvent limités à un seul langage, bien qu'il existe des composants disponibles au travers de plusieurs langages différents^a.

3.2.4 - La conversion automatique de code, un processus complexe et insatisfaisant

J'ai exposé précédemment une méthode consistant à développer l'application (ou ses constituants) pour une ou plusieurs plateformes définies et à la réécrire ensuite pour les autres plateformes. On pourrait alors imaginer une automatisation de cette opération.

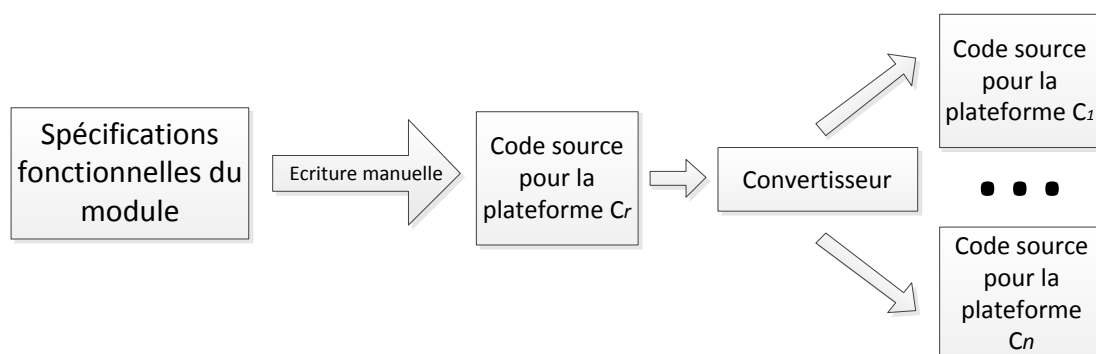


Fig. 15 – Transformation automatique du code source

^a Par exemple, le composant xerces, destiné à lire et à écrire des fichiers XML, est disponible en C, C++ et Java. De même, la librairie cryptographique BouncyCastle est disponible en Java et en C#.

Cependant, à l'heure actuelle les méthodes permettant d'extraire des modèles à partir d'un code source sont fort peu efficaces. Il est aujourd'hui facile de transformer un alphabet et une syntaxe en une autre par un simple mécanisme de substitution, autant les choses deviennent plus complexes lorsqu'il s'agit de transformer des enchaînements complexes d'opérations en enchaînements différents. Qui plus est, un tel convertisseur se heurterait inévitablement à la problématique des équivalences de composants ainsi qu'à leur nombre important.

Concrètement, une fonctionnalité f est réalisée au moyen d'une suite d'instructions I . S'il est relativement facile d'imaginer I à partir de f , le problème inverse est plus difficile, même pour un humain. Etant donné que le code source n'expose que les instructions I , les convertisseurs peinent à définir des équivalences complexes impliquant des suites d'instructions.

Par exemple, imaginons un composant de chiffrement C_1 destiné à la plateforme P_1 et un composant de chiffrement C_2 destiné à la plateforme P_2 qui se manipulent de la façon suivante :

	C_1	C_2
I	Appel d'une méthode d'initialisation	Appel d'une méthode pour chiffrer
	Appel d'une méthode pour définir les paramètres	
	Appel d'une méthode pour chiffrer	
	Appel d'une méthode pour finaliser le chiffrement	

Nous constatons que le composant C_1 est bien plus complexe à utiliser que le composant C_2 . Bien entendu, il n'est pas obligatoire de réaliser les différentes étapes d'utilisation de C_1 dans l'ordre ; on peut envisager d'initialiser C_1 à un endroit de l'application puis de chiffrer des données beaucoup plus tard à un autre endroit de l'application. Toute la complexité résidera donc à faire le lien entre les instructions appartenant à C_1 et celles appartenant à C_2 . A noter que passer de C_2 à C_1 est trivial (transformation d'un élément en plusieurs) et que seul l'inverse pose problème.

3.2.5 - Mais ces approches s'intéressent à des aspects locaux

Les différentes méthodes que je viens d'exposer à titre informatif ont toutes pour inconvénients d'être fortement locales, c'est-à-dire concentrées sur des aspects très proche de la technique et donc assez peu généralisable (limitation à un langage, équivalence de composants, compilation conditionnelle, etc.). Cela pose des difficultés car, sans un minimum d'abstraction, l'approche multiplateforme s'embarrasse de détails inutiles liés aux spécificités concrètes de chaque plateforme.

Il est important de considérer le fait suivant : il n'est pas possible de tenir compte de toutes les spécificités d'une plateforme. Il faut extraire l'essence de la plateforme vis-à-vis de la fonctionnalité recherchée, sans tenir compte des éléments annexes que celle-ci peut fournir (par exemple, une voiture sert à se déplacer et l'on peut se passer de la direction assistée).

L'approche locale, même si elle est simple à appréhender et à mettre en œuvre, montre rapidement ses limites, nécessitant bien souvent une grande part d'intervention humaine et une grande énergie juste pour maintenir la cohérence du processus (suivi des erreurs, problème de décalage, etc.), au détriment de son efficacité.

3.3 - L'organisation du projet multiplateforme apporte elle aussi des réponses

Nous venons de voir quelques solutions proches du code source permettant de gérer avec plus ou moins d'efficacité la complexité du développement multiplateforme. Il existe, de la même façon, des approches beaucoup plus globales, basées sur l'organisation des projets multiplateformes. Ce point de vue organisationnel, lui, cherche à résorber l'aspect multiplateforme au travers d'une gestion de celui-ci au niveau du processus complet de production logicielle.

3.3.1 - En parallélisant les projets

Le processus de parallélisation des projets est proche de la technique à base d'un code source de référence. En effet, ce processus se base lui aussi sur une référence unique : les spécifications et la conception.

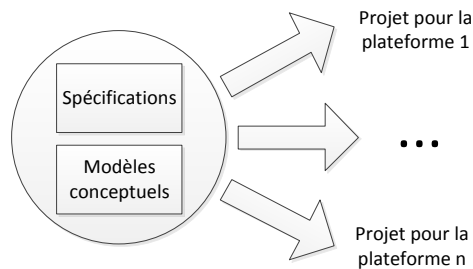


Fig. 16 – Parallélisation des projets

Cette fois-ci, on considère chaque plateforme impactant globalement le projet (système d’exploitation, architecture matérielle, langage, etc.) et on considère le développement sur cette plateforme comme étant un projet unique disposant de son propre cycle de vie, de sa propre équipe et de ses propres ressources. On pourrait comparer ça à l’implantation géographique des filiales d’une grosse entreprise^a où les décisions prises au niveau de la maison mère impactent l’ensemble des sociétés filles, elles-mêmes indépendantes les unes des autres.

Comme précédemment, cette façon de procéder casse la complexité liée à l’aspect multidimensionnel. Ce sont les spécifications et, éventuellement, les modèles conceptuels qui vont guider le développement vers un but commun.

Bien entendu, si cette manière de procéder résout ce problème en particulier, elle accroît globalement les coûts car rien n’est réutilisé d’une équipe à une autre. En effet, seule une parfaite autonomie va permettre de garantir la disparition de l’aspect multidimensionnel ; si un seul bloc de code est partagé d’un projet à un autre alors celui-ci hérite des propriétés multidimensionnelles du multiplateforme et la séparation entre les projets s’en retrouve irrémédiablement brisée.

Le problème du décalage entre versions n’est pas non plus résolu par cette façon de procéder. Au contraire, il est fortement amplifié car les processus sont totalement autonomes les uns par rapport aux autres, ce qui peut entraîner des dérives. C’est pour cela que chaque projet doit disposer de sa propre organisation, mais qu’il doit aussi exister une organisation centralisée qui va permettre d’avoir une vision globale. Ainsi, les dérives pourront être corrigées au cas par cas.

^a Google France est une entité totalement autonome vis-à-vis de Google Allemagne, par exemple.

3.3.2 - Ou en utilisant un projet pilote

Une autre méthode, issue de la précédente, consiste à utiliser un projet comme pilote au lieu des spécifications.

Contrairement à la méthode précédente, le projet pilote va avoir une fonction de prototype. L'expérience capitalisée sur celui-ci va permettre de mettre en évidence des problèmes beaucoup plus tôt. Qui plus est, les projets dépendants du projet pilote suivent des processus beaucoup moins distincts qu'avec une parallélisation totale. De fait, il est possible d'envisager de partager du code entre les différentes plateformes, moyennant quelques adaptations automatiques (notamment une transformation de la syntaxe) et manuelles (la recherche des équivalences entre les plateformes).

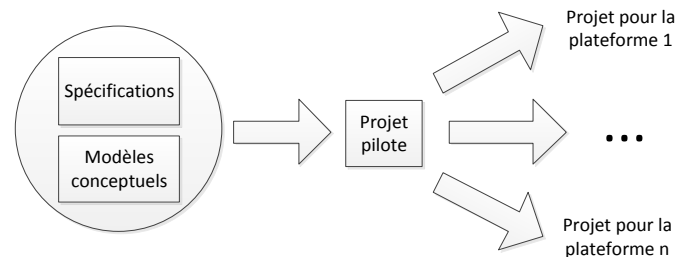


Fig. 17 – Utilisation d'un projet pilote

Il s'agit, ni plus ni moins, d'une extension de la technique à base d'un code source de référence, le code source de référence devenant, ici, un projet complet.

Les défauts de cette approche sont les mêmes que ceux de la précédente, si ce n'est que le décalage entre versions est réduit à une dimension temporelle (retard de version) plutôt qu'à une dimension fonctionnelle (fonctionnalités accidentellement différentes).

3.4 - Les architectures logicielles et intergicielles, au service de la technique et de l'organisation, pour simplifier à la fois le processus de développement, mais aussi le processus d'évolution

Les approches techniques et organisationnelles apportent des réponses fortement locales (très proches de la technique) et fortement globales (très proches de la gestion de projet). Toutefois, nous avons vu que ces deux extrêmes ont leurs limites et nécessitent une approche intermédiaire destinée à tenir compte à la fois des aspects techniques et des aspects organisationnels. Cette approche intermédiaire s'intéresse particulièrement à la

conception des applications (architecture logicielle) ainsi qu'à leurs interactions (architecture intergicielle, ou urbanisation). En effet, la conception fait abstraction de la notion de plateforme et s'efforce de décrire une représentation des programmes et des systèmes plutôt que les programmes et les systèmes eux-mêmes. Cette modélisation va dans le sens de notre réflexion sur le multiplateforme, en offrant un dénominateur commun à toutes les plateformes.

3.4.1 - La granularité et la réflexion au service du découpage intelligent

Le découpage d'une application va permettre de distinguer deux ensembles d'éléments fondamentaux :

- **ce qui est générique**, c'est-à-dire facilement transposable directement d'une plateforme à une autre moyennant quelques adaptations mineures et aisément automatisables (la transformation de la syntaxe d'un langage, par exemple). Plus généralement, il s'agit de ce qui est conceptuellement identique lors de la considération d'un ensemble de plateformes.
- **ce qui est spécifique à la plateforme**, c'est-à-dire les mécanismes et les composants propres à la plateforme que l'on ne pourra pas retrouver, ou seulement partiellement, sur les autres plateformes. Par exemple, la notion de cercle est commune à toutes les librairies graphiques, mais chacune d'entre elles dispose de ses propres méthodes pour le dessiner.

Cette distinction est fondamentale et va permettre de focaliser l'aspect technique uniquement sur les éléments spécifiques, les éléments génériques restants communs à l'ensemble des plateformes. Ainsi, idéalement, les éléments génériques peuvent manipuler les éléments spécifiques, sans que l'inverse soit possible.



Au final le seul couplage se situe dans une couche conceptuelle intermédiaire : celle qui permet d'assurer la communication entre généricité et spécificité : l'interface.

Ces trois éléments nous amènent à la notion de factorisation. Une grande part de nos difficultés sur le multiplateforme proviennent des couches spécifiques qui viennent « polluer » l'ensemble de nos applications. Concrètement donc, la factorisation consiste à

isoler ce qui est générique de ce qui est spécifique grâce à la barrière des interfaces. Cette isolation va pouvoir être faite à deux niveaux :

- **au niveau de l'architecture logicielle**, c'est-à-dire au niveau de la construction de l'application. La séparation générique/spécifique va se faire au sein même de l'application (ce qui limite à un langage unique). Malgré l'isolation, les interfaces et les éléments génériques resteront couplés à certaines plateformes, comme le langage.
- **au niveau de l'architecture intergicielle**, en isolant le spécifique et en lui permettant de communiquer avec le générique au travers d'interface faiblement couplée, comme le réseau par exemple.

L'importance de la granularité

La granularité, au sein d'une application, représente la finesse des différents modules, c'est-à-dire leur capacité à ne traiter qu'un ensemble d'opérations au périmètre fortement délimité (grain). Par exemple, un composant qui peut transformer un nombre en code binaire est un composant à forte granularité. A contrario, un composant qui peut transformer un nombre en code binaire, hexadécimal et octal est un composant à faible granularité.

Bien entendu, la notion de granularité dépend de l'observateur : plus le niveau d'abstraction est élevé, plus la granularité paraît forte et plus le niveau d'abstraction est faible, plus la granularité paraît faible.

Concrètement, les éléments spécifiques d'une application multiplateforme doivent avoir une granularité forte pour pouvoir être portés plus facilement. En effet, étant donné que les plateformes ne proposent pas les mêmes fonctionnalités, il peut être judicieux de ne porter que les éléments nécessaires plutôt que de devoir chercher des équivalents massifs. Ainsi, si certains éléments spécifiques sont communs à deux plateformes, l'effort de portage sera réduit (voir Fig. 18).

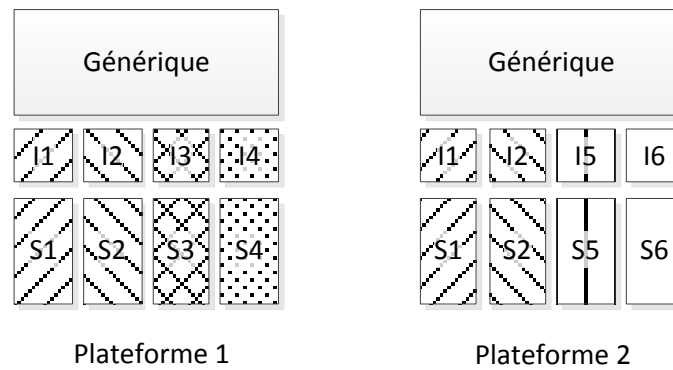


Fig. 18 – Deux applications avec deux parties spécifiques (S1 et S2) identiques

La réflexion, pour factoriser dynamiquement

D'un point de vue structurel, il est préférable que la communication s'opère du générique vers le spécifique. Dans l'idéal c'est la partie générique qui crée, manipule et interroge la partie spécifique et jamais l'inverse. Néanmoins, au sein d'une application, les liens entre les éléments sont généralement forts, même lorsque l'on va utiliser des interfaces. En effet, les applications se limitent à des unités restreintes (l'exécutable, le composant statique ou dynamique, le paquetage, le script, etc.) et il n'est pas possible pour les éléments de cette unité de manipuler d'autres éléments qui n'existent pas encore.

Ainsi, au sein de l'unité, les éléments chargés de l'instanciation des parties spécifiques appartiennent eux-mêmes à la partie spécifique car ils ont connaissance de la nature concrète de ce qu'ils fournissent aux couches supérieures (par exemple, une fabrique^a est un composant spécifique car il est en charge de fournir une implémentation parmi celles dont il a connaissance).

Il peut être intéressant, dans le cadre du multiplateforme, de disposer d'une unité applicative servant de dénominateur commun, c'est-à-dire implémentant l'ensemble de la partie générique de l'application finale. Ce dénominateur commun sera ensuite intégré à chaque projet dépendant des plateformes, l'intérêt étant pour le dénominateur commun de traiter avec les éléments de la partie spécifique, sans pour autant sortir de sa propre unité.

^a La fabrique (ou factory method) est un patron de conception créationnel permettant d'instancier des objets dont le type est dérivé d'un type abstrait. La classe exacte de l'objet produit n'est donc pas exactement connue par l'objet qui exploite la fabrique.

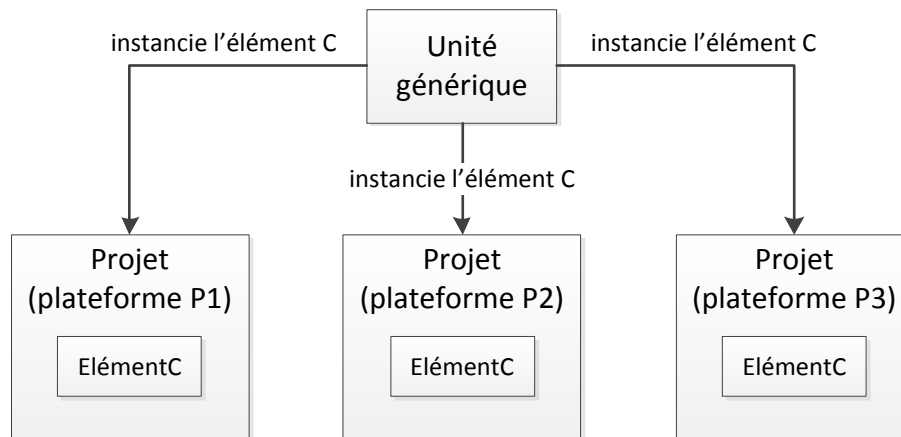


Fig. 19 – Instanciation d'un élément par l'unité générique

La réflexion est un mécanisme permettant à un programme de manipuler ses structures internes de haut niveau (ses objets, par exemple) durant son exécution. On distinguera, par ailleurs, deux types de réflexion :

- **la réflexion structurelle**, qui consiste à abstraire le code d'un programme (permettant dès lors de manipuler ce programme pendant son exécution, pour des opérations de débogage par exemple) et tous les types accessibles par ce programme (permettant au programme de modifier ses objets, d'instancier des classes et d'invoquer des méthodes de manière dynamique).
- **la réflexion comportementale**, qui va permettre au programme de prendre connaissance de la façon dont il est exécuté par l'évaluateur de programme et de modifier le comportement de celui-ci. Ainsi, le programme peut, par exemple, se restructurer de façon à s'adapter à son environnement.

C'est la réflexion comportementale qui va permettre à l'unité générique de manipuler directement des éléments au sein d'autres unités exécutables prévues pour cela (bibliothèques dynamiques, paquets, etc.). Ainsi, fonction de l'unité exécutable à laquelle elle sera liée, l'unité générique va manipuler des éléments dont l'implémentation sera différente, sans que leur existence ne soit obligatoire lors de sa compilation. De fait, l'unité générique manipule correctement l'ensemble des unités spécifiques au sein de chaque projet sans contrevenir à la communication ascendante.

Au final, la réflexion comportementale rend possible la description de mécanismes abstraits tout en manipulant des éléments concrets qui n'existent pas encore

3.4.2 - Abstraire les spécificités techniques au travers des aspects

Idéalement, une application doit répondre à des propriétés de forte cohésion et de faible couplage. La forte cohésion consiste à restreindre les fonctionnalités d'un élément à son domaine strict (par exemple, une classe s'occupant à la fois de préoccupation métier, de persistance et de communication réseau est faiblement cohésive), car un élément qui effectue trop de tâches devient rapidement difficile à comprendre et à maintenir, perd en réutilisabilité et se trouve fragilisé par les changements internes ou externes. Le faible couplage consiste à minimiser les dépendances entre modules pour réduire l'impact des changements ultérieurs et favoriser la réutilisabilité.

On pourrait croire que ces deux notions sont incompatibles, mais celles-ci interviennent toutes deux à des niveaux différents (voir Fig. 20), le faible couplage intervenant au niveau de modules dont la structure interne est fortement cohésive et fortement couplée. Cette notion prend toute son importance dans le cadre du multiplateforme, car le faible couplage va permettre d'accroître la granularité et donc de simplifier l'écriture des applications multiplateformes.

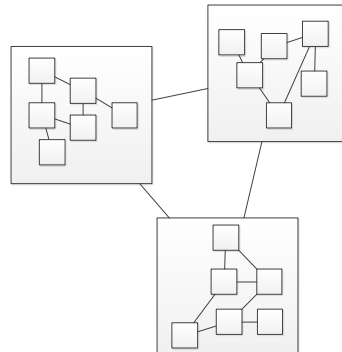


Fig. 20 – Illustration du faible couplage et de la forte cohésion

Cependant, il faut considérer qu'une application est constituée de deux grandes familles d'exigences: les exigences fonctionnelles et les exigences non fonctionnelles. Les exigences fonctionnelles se préoccupent du comportement du système, de ses fonctions et de ses services d'un point de vue métier. Les exigences non fonctionnelles vont plutôt s'intéresser aux contraintes imposées pour la satisfaction des exigences fonctionnelles (performance, sécurité, sûreté, convivialité, concurrence, etc.). On peut aussi parler de fonctions supports.

Les exigences fonctionnelles se prêtent bien à une représentation modulaire de par leur verticalité (propriété à être regroupée hiérarchiquement en blocs bien distincts), mais pas une grande partie des exigences non fonctionnelles du fait d'une certaine horizontalité, c'est-à-dire à la portée difficilement délimitable, ce qui favorise un fort couplage à tous les niveaux de l'application. Par exemple, la gestion des droits (Fig. 21) peut être représentée comme un module unique, mais celle-ci va être couplée avec les différents constituants internes des autres modules, impliquant un fort couplage général.

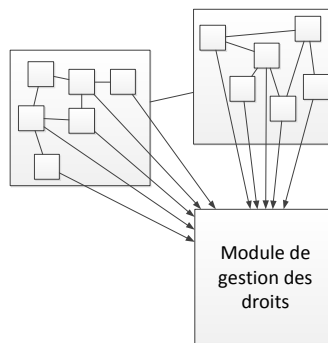


Fig. 21 – Exemple d'exigence non fonctionnelle transversale

Aussi, la programmation par aspect est un paradigme^a de programmation apparu au milieu des années 1990 au Xerox Palo Alto Research Center (travaux de Gregor Kiczales et de son équipe) pour pallier aux difficultés de la programmation objet et de la programmation impérative (les deux paradigmes de programmation majeurs employés par l'industrie informatique) à traiter ces préoccupations transverses. Ainsi, la programmation orientée aspect va chercher à séparer l'implémentation de toutes les exigences, qu'elles soient fonctionnelles ou non, d'un logiciel. Chaque problématique est alors écrite séparément et intégrée aux autres pour former le système final, au moyen de règles définies par le programmeur. Grâce à cela, les exigences qui affectaient de multiples classes vont pouvoir être encapsulées dans des modules parfaitement distincts et limités, accroissant de fait leur réutilisabilité.

Le concept de préoccupation tel que défini par Gregor Kiczales et ses collègues peut être implémenté sous deux formes :

^a Un paradigme de programmation est une philosophie de programmation décrivant la manière dont les solutions aux problèmes informatiques doivent être formulées et abordées.

- **un composant**, si la préoccupation peut être clairement encapsulée dans un objet, ou un module. Il s'agit là des implémentations des exigences fonctionnelles d'un système.
- **un aspect**, s'il n'est pas possible d'encapsuler la préoccupation dans un composant. Il s'agit souvent des exigences non fonctionnelles d'un système.

En ce sens, la programmation orientée aspect ne se substitue pas au paradigme procédural ou au paradigme objet, mais va plutôt étendre ceux-ci avec le concept d'aspect en plus de celui de composant. Ainsi, les aspects influent directement sur les composants sans que ceux-ci en aient conscience, ce qui garantit que les préoccupations transverses ne contamineront plus les composants de manière directe.

Par exemple, imaginons un logiciel qui se synchronise sur un serveur de fichier. Dans une approche objet classique, les composants qui manipulent les fichiers locaux s'occuperaient des vérifications régulières auprès du serveur et du rapatriement des modifications éventuelles. Dans une approche par aspect, les composants qui manipulent les fichiers locaux n'auraient pas conscience d'être synchronisés. Ce serait directement le module de synchronisation qui injecterait ses capacités de synchronisation dans les composants qui manipulent les fichiers.

En termes de conception et de développement, cela implique une approche légèrement différente (voir Fig. 22). L'analyse et la modélisation du système vont, comme avant, mettre en évidence les composants verticaux et leurs interactions, mais aussi les différents aspects associés aux préoccupations transverses. Puis, composants et aspects seront écrits séparément par le programmeur, les aspects encapsulant à la fois des fonctionnalités, mais aussi des règles d'intégrations qui vont permettre à cet aspect d'être injecté au sein des composants. Enfin, aspects et composants vont être intégrés entre eux au moyen d'un mécanisme appelé tisseur d'aspect. Le tisseur va s'occuper de l'injection des fonctionnalités propres aux aspects directement dans les composants et composer le système final à partir des règles définies par le programmeur. Cette opération peut être effectuée statiquement (le tisseur d'aspect étant un préprocesseur) ou dynamiquement pendant l'exécution.

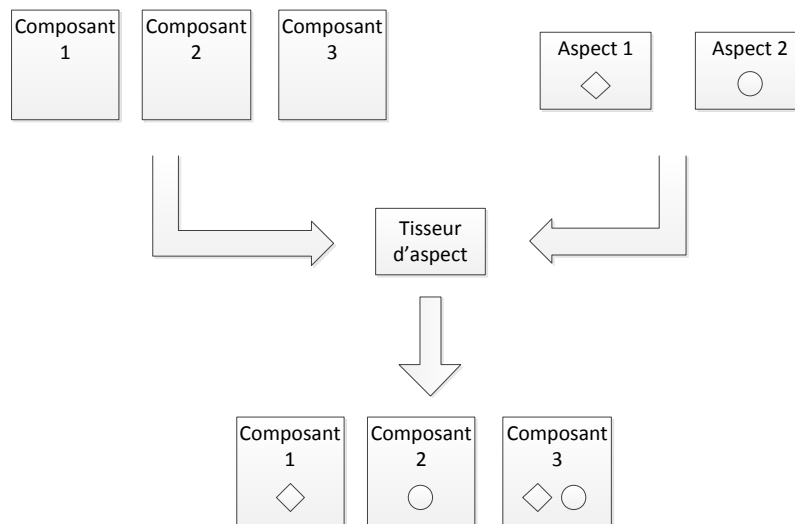


Fig. 22 – Principe général de la programmation orientée aspect

L'injection de ces traitements spécifiques se fait au niveau de ce que l'approche aspect nomme « points de jonction » (join point). Il s'agit d'évènements spécifiques tels que l'invocation d'une méthode, la modification d'un attribut ou l'instanciation d'une classe. C'est autour (avant ou après, par exemple) de ces points de jonction que l'on va pouvoir insérer notre « code d'aspect » (advice), formant ainsi un « point de coupe » (pointcut), c'est-à-dire un point de jonction destiné à recevoir du code d'aspect.

Au final, un « aspect » est un composant qui va encapsuler plusieurs codes d'aspect et les points de jonction qui leur sont associés.

L'avantage fondamental de la programmation orientée aspect réside dans l'encapsulation des préoccupations non fonctionnelles au sein de ces composants particuliers que sont les aspects. Le découpage est plus fin et le mécanisme de tissage d'aspect, s'il est statique, peut être extérieur au langage, comme nous l'avons vu précédemment pour le preprocessing. Au final, en écrivant des aspects spécifiques aux différentes plateformes cibles, nous pouvons abstraire les spécificités de celles-ci et les injecter directement dans le code avant sa compilation.

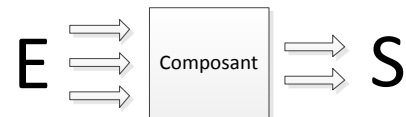
3.4.3 - Aller plus loin dans l'approche objet au travers des composants distribués

La factorisation des éléments de l'application est un des points clés de la simplification du processus multiplateforme. Néanmoins, les méthodes de factorisation que nous avons vues

jusqu'à présent portent sur l'architecture interne des applications en tant qu'entité exécutable par un processeur.

Cette façon de procéder implique cependant que l'on se restreigne à un langage spécifique et aux composants compatibles avec celui-ci. En effet, si je désire modifier un composant de mon application, je dois modifier le code source de mon programme, ce qui implique ensuite de retester l'ensemble de mon application pour m'assurer que les modifications n'ont pas eu d'effet néfaste sur l'ensemble des éléments qui fonctionnaient déjà. De plus, si l'application est écrite dans un langage compilé, il faudra la recompiler à nouveau, sauf si celle-ci utilise l'introspection pour charger dynamiquement des classes au travers de plusieurs composants.

Si l'on se place à un niveau encore plus élevé, un composant est une entité logicielle dont on connaît les interfaces, c'est-à-dire les entrées et les sorties. On peut



en connaître le fonctionnement interne (le composant est une boîte blanche) ou non (le composant est une boîte noire) sans que cela n'ait d'impact sur son utilisation.

Si l'on généralise, on peut considérer que le composant est un logiciel à part entière disposant d'interfaces de communication. C'est ce logiciel que nous appellerons « composant distribué ».

La communication entre composants distribués peut se faire d'une multitude de façon dont les plus communes sont :

- le réseau (pair à pair)
- les communications inter-processus, ou IPC (signaux, échanges de messages, mémoire partagée)
- les bus logiciels spécifiques (CORBA, D-Bus, etc.)
- les services web

Ces méthodes de communication sont dites « lâches », c'est-à-dire que la communication entre deux composants *A* et *B* est établie au moyen d'un médium permettant de modifier *B* sans avoir à modifier *A*, et vice versa. Par ailleurs, la notion de couplage lâche est liée à la

notion d'interopérabilité, car les médias de communication ouverts sont souvent considérés comme lâches.

Le découpage en objets distribués répond donc à notre besoin de factorisation en associant un composant logiciel à son langage, sa plateforme d'exécution et aux autres composants non distribués qui le constituent. Le composant logiciel distribué est donc toujours dépendant de ses aspects spécifiques, mais ceux-ci sont totalement encapsulés avec le composant. Bien entendu, cette encapsulation peut être plus ou moins étendue en fonction de la nature plus ou moins lâche du couplage (voir Fig. 23 et Fig. 24)

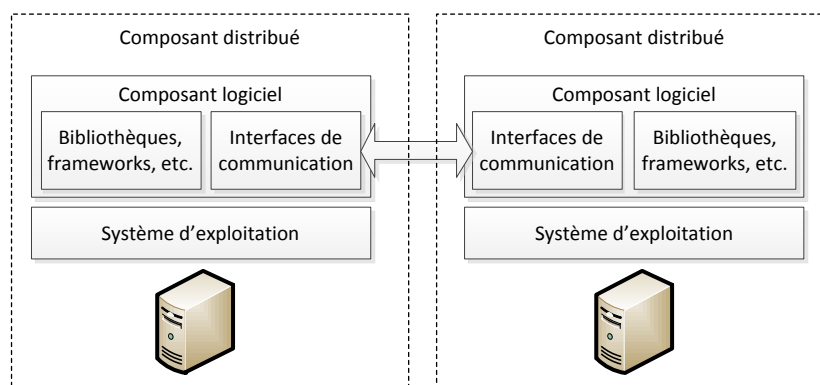


Fig. 23 – Exemple de composant distribué avec communication intermachines (réseau) : couplage très lâche

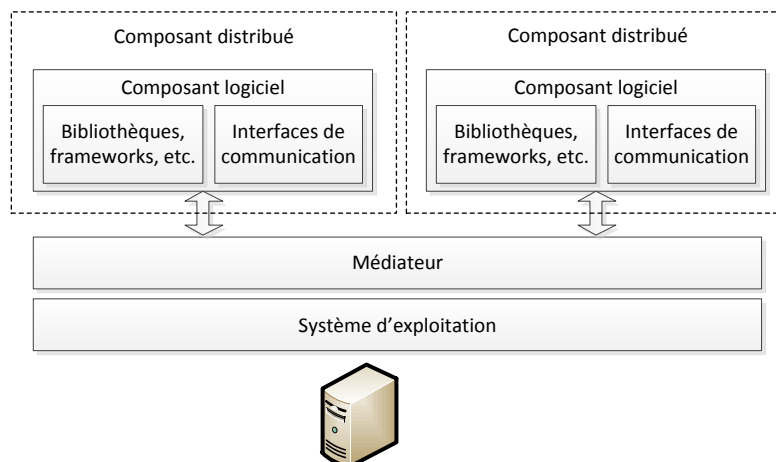


Fig. 24 – Exemple de composant distribué avec médiateur (bus logiciel, communication inter-processus, etc.) : couplage plus important du au système d'exploitation commun

Tout comme les composants classiques, les composants distribués composent une architecture. Celle-ci n'est toutefois plus dite « logicielle », mais « interlogicielle » du fait de sa

nature dispersée au travers de plusieurs logiciels exécutés sur des processus ou des processeurs différents.

Ces architectures sont donc soumises aux mêmes inconvénients que les architectures à base de composants classiques, notamment en ce qui concerne les préoccupations transversales qui nuisent au faible couplage général de l'application.

Cependant, ces problèmes de couplages sont exacerbés par le médium de communication, généralement plus lourd (réseau, IPC, etc.) qu'une simple invocation de méthode, et sont plus difficiles à résoudre par une approche par aspects. En effet, la programmation par aspects nécessite de pouvoir injecter des traitements de manière statique (préprocessing) ou dynamique (pendant l'exécution) dans les composants ce qui nécessite, dans le cas des composants distribués, de définir des interfaces spécifiques pour l'injection de traitements ou de prévoir des architectures intergicielles complexes pour que les composants soient en mesure de connaître, dès leur instanciation, les règles des aspects qui les concernent. De plus, l'injection en elle-même sera dépendante de la lourdeur du médium de communication, encore une fois.

3.4.4 - Adopter une approche en couches réparties

La répartition des composants permet de réaliser des systèmes dans lesquels le couplage entre deux éléments A et B est lâche, c'est-à-dire que B peut être remplacé par un équivalent B' sans que le fonctionnement de A ne soit perturbé, et vice versa.

Concrètement, dans une architecture objet classique, les objets interagissent entre eux et forment ce que l'on appelle des couches, c'est-à-dire des paquetages de composants possédant une forte cohésion et une sémantique proche.

Les informations véhiculées au sein du programme circulent à travers ces couches qui les transforment et les transmettent à d'autres couches. Ainsi, l'architecture en couche s'établit comme un ensemble de strates où, dans l'idéal, chaque strate n'a connaissance que de deux autres strates : celle du dessous et celle du dessus. Bien souvent, la hauteur des couches définit leur genericité, celle-ci diminuant au fur et à mesure que l'on se rapproche des couches basses.

Ainsi, comme les composants peuvent être répartis, les couches peuvent l'être de la même façon, qu'elles soient elles-mêmes composées de composants distribués ou non. Les applications sont alors composées de couches réparties qui communiquent au travers de couplages lâches, ce qui permet de garantir le caractère interchangeable de chaque couche. A noter que, bien souvent, une couche répartie sera appelée « tiers » et qu'une application dont l'architecture est composée de couches réparties sera dite « n-tiers » ou « multitiers ».

L'architecture 2-tiers

Cette architecture fait intervenir deux couches réparties pouvant prendre en charge plusieurs aspects d'une application. L'architecture 2-tiers la plus connue est l'architecture client-serveur où un consommateur de ressource (le client) va demander un service à une autre application (le serveur). On distingue, d'après le Gartner Group, plusieurs types d'architecture client-serveur. Celles-ci sont présentées sur le schéma suivant :

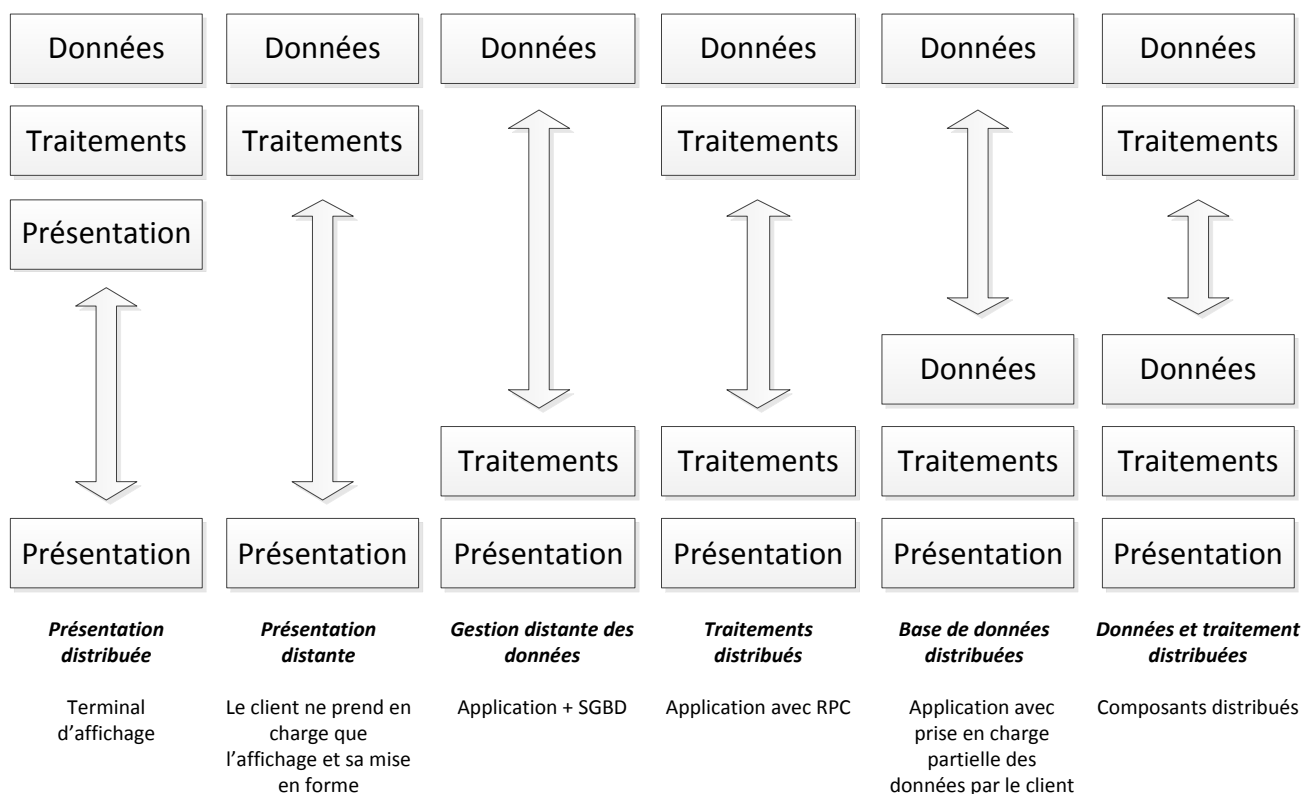


Fig. 25 – Classification des types d'architectures client-serveur par le Gartner Group

L'architecture 3-tiers

L'architecture 2-tiers que nous venons de voir précédemment présente l'inconvénient majeur de coupler deux niveaux applicatifs^a. L'architecture 3-tiers vient résoudre ce problème en mettant en place trois couches réparties prenant chacune en charge un niveau applicatif :

- **la couche de présentation**, qui va servir d'interface homme-machine pour l'utilisateur (affichage de données et récupération des entrées utilisateurs).
- **la couche métier**, qui définit la logique de l'application et qui est prise en charge par un calculateur intermédiaire.
- **la couche d'accès aux données**, qui gère le stockage persistant des données et assure une centralisation des données.

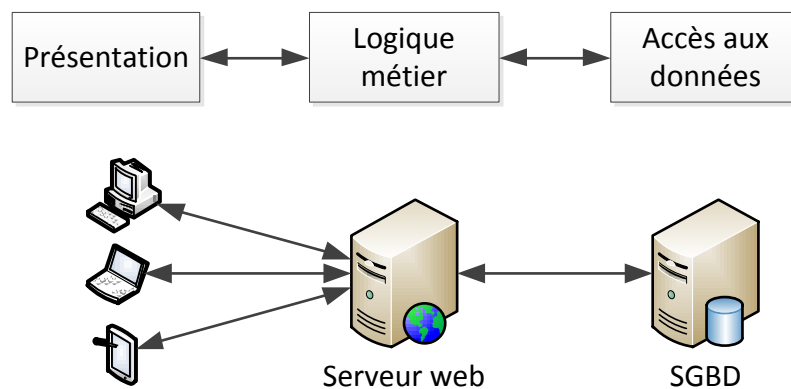


Fig. 26 – Architecture 3-tiers générique (en haut) et exemple courant d'implémentation

Grâce à cette nouvelle découpe, la logique métier est factorisée en une couche unique, permettant un accès depuis plusieurs plateformes distantes. De plus, les spécificités liées aux données sont factorisées dans le tiers d'accès aux données, celui-ci devant se conformer, dans le cas idéal, à des interfaces génériques connues de la logique métier.

^a On considère, en effet, qu'une application, distribuée ou non, est composée de trois niveaux :

- la présentation (interaction avec l'utilisateur, saisie, affichage).
- les traitements (les opérations à réaliser par l'application), ceux-ci pouvant être locaux, c'est-à-dire dédiés à la gestion de la présentation, et globaux, c'est-à-dire constituant l'application elle-même.
- les données ainsi que leur gestion (stockage, accès, etc.)

Ainsi, chaque couche spécifique (tiers de présentation et tiers d'accès aux données) peut être modifiée ou remplacée sans perturber le fonctionnement de ses collaboratrices.

Pourquoi accroître le nombre de tiers ?

L'augmentation du nombre de tiers, avec un minimum de deux, a pour principal avantage de permettre la factorisation d'une partie de l'application. Une part des tiers devient alors notre ensemble générique, c'est-à-dire réutilisable quel que soit la plateforme qui y accède, tandis que le reste compose notre ensemble spécifique, c'est-à-dire l'ensemble d'éléments dépendants des plateformes.

Bien que l'architecture 3-tiers permette de prendre en charge les trois niveaux applicatifs, le nombre de tiers peut encore être augmenté. En effet, le tiers de logique métier et le tiers d'accès aux données peuvent nécessiter un découpage bien plus fin (augmentation de la granularité) qui va pouvoir être réalisé par l'ajout de nouvelles couches. Ainsi, les niveaux applicatifs traitements et données seront eux-mêmes répartis en sous-tiers, ce qui au final se traduit par :

- **une meilleure répartition de la logique applicative**, pour améliorer la réutilisabilité et la flexibilité globale grâce à des tiers indépendants et fortement spécialisés, exploitables au sein de plusieurs applications. Les performances et la tolérance aux pannes sont aussi accrues par la réplication des couches, un client pouvant changer de couche si celle qu'il utilise est indisponible ou surchargée.
- **une meilleure répartition de la gestion des données**, en permettant aux couches de logique métier de se connecter à des sources de données multiples ou de répartir la gestion des données en cas de volume important (bases de données distribuées).

3.4.5 - Raisonner par services, pour orchestrer et factoriser

Les architectures à base de composants distribués et les architectures en couche que nous avons vues précédemment étendent les concepts de couches et de composants logiciels en forçant les éléments à se doter de descriptions claires et figées de leurs interfaces de communication (traitements disponibles et entrées/sorties associées).

Au travers de ces architectures, les éléments distribués tendent à accomplir des tâches aux champs de plus en plus restreints (spécialisation) et, de fait, tendent à devenir de plus en plus petits (granularisation).

Cette spécialisation a amené les constituants des infrastructures distribuées à se rapprocher de plus en plus du concept de service, c'est-à-dire d'un travail effectué au bénéfice d'autres entités. Cette nouvelle façon d'aborder les architectures distribuées s'inspire d'une forme particulière des architectures en flots de données^a et a donné naissance à l'architecture orientée service aux alentours des années 2000.

L'architecture orientée service (ou SOA pour Service Oriented Architecture), standardisée par l'OASIS^b, définit une infrastructure pour organiser et utiliser des fonctionnalités distribuées. On remarque que cette approche se distingue déjà des précédentes d'un point de vue philosophique : ce ne sont plus des éléments techniques (composants, couches, etc.) qui sont distribués, mais bien des fonctionnalités, c'est-à-dire des réponses à des besoins indépendamment de leur aspect technique.

Philosophiquement encore, l'architecture orientée service se base sur trois concepts clés (OASIS, 2006) :

- **la visibilité**, c'est-à-dire la capacité pour demandeurs et producteurs de fonctionnalités de se voir mutuellement. Ceci se traduit par des mécanismes de description et de découverte de ces fonctionnalités.
- **l'interaction**, c'est-à-dire la capacité pour les demandeurs de besoin à utiliser les fonctionnalités. Ceci se traduit par la mise en œuvre de mécanismes de communication et d'utilisation distante de ces fonctionnalités.
- **l'effet**, c'est-à-dire le résultat d'une interaction, qu'il s'agisse d'un résultat réel (retourné à l'appelant) ou d'un effet sur l'état global du système. A noter qu'il s'agit d'un effet utile pour le demandeur vis-à-vis du métier de celui-ci.

Là où l'architecture par composants distribués et l'architecture multitiers définissaient respectivement le composant et la couche comme entité atomique (indivisible),

^a Architecture dans laquelle les composants sont reliés entre eux par des flux de données.

^b Organization for the Advancement of Structured Information Standards. Il s'agit d'un consortium mondial à but non lucratif qui pilote le développement, la convergence et l'adoption des standards ouverts pour les architectures de systèmes d'information.

l'architecture orientée service pose le service comme étant le plus petit élément lui appartenant. Un service va pouvoir effectuer, décrire et proposer une fonctionnalité à un demandeur, ce qui fait de lui un mécanisme au travers duquel des besoins et des fonctionnalités sont associés (OASIS, 2006).

Les services sont caractérisés par leur description (*service description*). Cette description va permettre d'exposer les informations suivantes aux différents consommateurs de services :

- le service existe et est accessible (*service reachability*)
- le service est en mesure d'effectuer un ensemble de tâches (*service functionality*) dont les noms, les entrées et les sorties sont clairement définies (*service interface*)
- le service est soumis à un ensemble de politiques (*service policy*) pouvant définir des contraintes telles que le protocole utilisé, l'utilisation du chiffrement, etc.
- le service sera conforme ou non (dans une certaine mesure, implicite ou explicite) aux politiques propres au consommateur. On dit qu'un contrat (*service contract*) est passé entre le consommateur et le service.

Le principe de visibilité évoqué est assuré par un annuaire de services qui va pouvoir découvrir et enregistrer des services et leurs descriptions. Cet annuaire, qui est lui-même exposé sous la forme de service, permettra par la suite aux consommateurs de rechercher des services spécifiques selon les critères disponibles dans les descriptions de service.

Le principe d'interaction est assuré d'une part par les descriptions fournies par l'annuaire de service et d'autre part par le médium utilisé pour la communication avec les services. Ces médias sont les mêmes que pour les couches et les composants distribués et sont généralement des couplages lâches (réseau, communication interprocessus, etc.) bien que cela ne soit pas imposé. En effet, les différents constituants de l'architecture orientée service peuvent être implémentés de plusieurs façons, selon des approches de plus ou moins haut niveau. Par exemple, les services peuvent être implémentés sous forme de services web, de démons ou de composants distribués connectés à un bus de services.

Il est même possible d'implémenter une architecture orientée service au sein d'une application unique, au travers d'un framework dédié à cela. Cependant, une telle approche

diminue fortement l'aspect « lâche » du couplage, bien qu'il apporte des fonctionnalités intéressantes de découverte et d'orchestration.

Enfin, l'effet de l'invocation d'un service est lié à certaines de ses caractéristiques. Dans l'idéal, un service doit être :

- **autosuffisant du point de vue de l'appelant**, c'est-à-dire disposer de ses propres ressources pour fonctionner sans que l'appelant n'ait à fournir de paramètres techniques (persistance, format, encodage, etc.).
- **sans état**, c'est-à-dire ne pas être perturbé par ses précédentes exécutions (le service est une instance unique).
- **atomique**, c'est-à-dire qu'à la fin de son invocation le service est rendu ou non (pas de service partiellement rendu).
- **opaque**, c'est-à-dire que son implémentation est inconnue du consommateur de service, à l'exception des fonctionnalités offertes (service interface) et des informations requises par le consommateur pour déterminer si un service correspond bien à son besoin (service policy).
- **exploitable au travers d'un couplage lâche**, bien que ce dernier point soit sujet à caution puisque l'architecture orientée service peut être mise en place au sein d'une application monolithique.

De fait, le service va retourner un résultat et/ou modifier l'état global du système, mais jamais son propre état. De plus, la propriété d'atomicité garantit que le service a été rendu et que l'on ne se trouve pas dans un état instable tant au niveau local (données manipulées par le service) qu'au niveau global (état global du système).

Il est important de noter qu'un service peut faire appel à d'autres services pour parvenir à la fin de son exécution. Il est en effet possible pour les services d'interagir entre eux ou d'être associé dans un flux de données au travers de ce que l'on appelle l'orchestration de service. Le chef d'orchestre a connaissance des services et possède une description du flux de données au travers des différents services. Il va ainsi superviser la circulation de ce flux et l'exploitation des différents services concernés, jusqu'à ce que l'objectif soit rempli. Le chef d'orchestre est lui aussi exposé en tant que service, de façon à ce qu'un flot de service (business process) puisse être exploité en tant que service.

L'orchestration a pour avantage de pouvoir composer avec des services existants et d'assurer un cheminement entre services. Ce cheminement étant décrit, il peut être modifié simplement au niveau du chef d'orchestre, on parlera alors de business process management (BPM). On parle de chorégraphie lorsque le chef d'orchestre est absent et que les différents services sont en mesure de s'auto-organiser pour atteindre un but et, par la suite, de s'adapter aux changements structurels pouvant survenir au cours du temps (dysfonctionnement d'un service, apparition d'un service aux politiques plus intéressantes, etc.).

L'architecture orientée service offre des avantages à plusieurs niveaux. Tout d'abord, sur le plan organisationnel, elle permet d'augmenter la flexibilité grâce aux principes de bases des services, tels qu'exposés précédemment. La forte standardisation des implémentations de cette architecture contribue, par ailleurs, à permettre un déploiement simple et autonome des services, réduisant les phénomènes de dépendances. De même l'intégration standard et la réutilisation des services garantissent une excellente modularité et une forte réutilisabilité. De plus, là encore, la standardisation forte et le couplage lâche vont réduire l'effort de développement lors d'une migration ou d'une extension du système.

Au final donc, les services disposent des mêmes propriétés intéressantes que les composants et les couches distribuées et permettent la factorisation des différentes parties d'un système. Couplé à une infrastructure intergicielle (à base de services web par exemple), ils rendent accessible des fonctionnalités entières, à partir du moment où la plateforme qui les exploite est en mesure de les invoquer.

3.4.6 - Virtualiser, pour que les systèmes s'adaptent à tout

Les composants, les couches et les services distribués ont tous trois la capacité d'être totalement autonomes, c'est-à-dire encapsulant à la fois les fonctionnalités qu'elles sont censées assurer et un environnement d'exécution complet (voir Fig. 23, page 50). Cet environnement d'exécution comprend un groupe de ressources physiques (mémoire, processeur, unités de stockage, interfaces réseau, etc.), un système d'exploitation et des éléments de plus haut niveau tels qu'un serveur d'application, etc.

Ce bloc autonome, exploitable au travers du faible couplage, est une structure parfaitement factorisée, accessible depuis n'importe quel autre élément composant le système d'information, à condition que celui-ci soit capable de traiter avec le médium de communication idoine. Cependant, les éléments distribués sont destinés, tôt ou tard, à migrer et à évoluer, voire même à changer de plateforme d'exécution. Des problèmes vont alors apparaître puisque ces éléments sont fortement couplés à l'infrastructure qui les exécute.

La virtualisation apporte une réponse à ces problèmes en inversant la logique de migration : « si l'élément distribué ne s'adapte pas à la nouvelle plateforme, alors la nouvelle plateforme doit s'adapter à lui ». On considère alors que notre ensemble de ressources physiques peut être fragmenté et alloué à des machines virtuelles isolées les unes des autres, l'opération de virtualisation désignant l'abstraction d'un ensemble de ressources physiques (ensemble d'unités de stockage agrégées en un stockage unique, ensemble de processeurs agrégés en un processeur unique, etc.).

Au final, les éléments distribués sont associés à une infrastructure virtuelle d'exécution (système d'exploitation virtualisé et logiciels nécessaires à l'exécution), permettant de les traiter comme de grandes structures transposables, dupliquables et redéployables.

On distinguera trois grandes approches (voir Fig. 27) permettant de virtualiser un système d'exploitation.

La première, appelée virtualisation totale, consiste à amener le système d'exploitation virtualisé à croire qu'il communique directement avec le matériel alors que celui-ci n'est qu'une couche logicielle émulant le matériel.

La seconde consiste à injecter, dans le système d'exploitation virtualisé, une couche logicielle (interface paravirtualisée) lui permettant de communiquer directement avec les couches d'abstraction matérielle du système d'exploitation hôte. On parle alors de paravirtualisation.

Enfin, la troisième approche, l'hypervision, consiste à permettre au système d'exploitation virtualisé de communiquer avec le matériel soit directement au travers de fonctionnalités de virtualisation offertes nativement par le matériel, soit indirectement au travers du système de virtualisation (hyperviseur) qui fera office de système d'exploitation hôte.

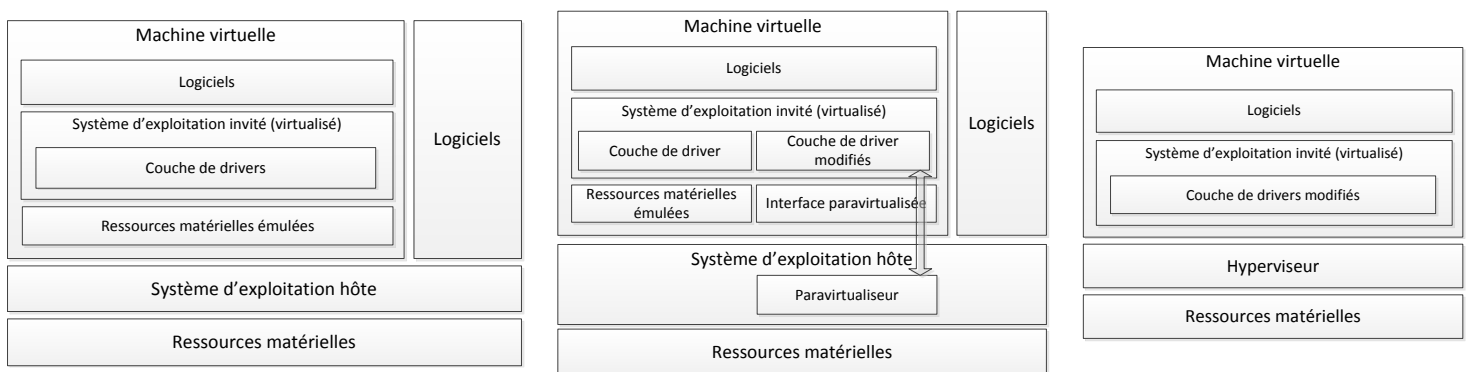


Fig. 27 – Architecture d'un virtualiseur total (à gauche), d'un paravirtualiseur (au centre) et d'un hyperviseur (à droite).

La différence majeure entre les trois approches consiste en une quantité différente de couches à traverser pour qu'une instruction du système virtualisé soit exécutée par le matériel.

Dans le cadre de la virtualisation totale, les instructions traversent les différentes couches d'abstraction du système d'exploitation virtualisé, la couche de ressources matérielles émulées qui va les interpréter et les retranscrire sur le système d'exploitation hôte où elles traverseront là encore les différentes couches d'abstraction avant d'atteindre le matériel.

Dans la technique de paravirtualisation, le nombre de couches diminue. En effet, le système d'exploitation hôte est modifié pour pouvoir communiquer directement avec le système d'exploitation invité, de manière à réduire le nombre de couches traversées pour accroître les performances globales.

Enfin, en ce qui concerne l'hypervision, le système d'exploitation hôte est totalement supprimé et remplacé par l'hyperviseur, un système d'exploitation minimaliste dédié à la virtualisation. Le système d'exploitation invité communique alors directement avec l'hyperviseur et le nombre de couches traversées est alors encore réduit.

Concrètement, la virtualisation permet de coupler finement les éléments distribués et leur environnement d'exécution en les enfermant dans un paquetage portable de haut niveau : la machine virtuelle. Ainsi, si le système destiné à accueillir l'élément distribué n'est pas compatible avec celui-ci, la virtualisation va permettre d'abstraire les spécificités matérielles pour assurer malgré tout son fonctionnement.

Cependant, les machines virtuelles sont des entités énormes (système d'exploitation et logiciels) et peu granulaires. Elles sont délicates à manipuler du fait de leur lourdeur et ne peuvent constituer qu'une solution de dernier recours en ce qui concerne le multiplateforme.

3.4.7 - Placer les modèles au cœur de la fabrication logicielle

Les approches conceptuelles que nous avons considérées jusqu'à présent se basent sur le concept de factorisation : factorisation de code, factorisation de fonctionnalité et factorisation de plateforme. Les applications sont alors constituées de parties génériques (le composant, la machine virtuelle, le service, le tiers, etc.) qui dépendent concrètement d'une plateforme, mais qui sont capables d'être exploitées au travers d'interfaces indépendantes, et de parties spécifiques qui sont directement associées à une plateforme.

Toutefois, ces notions de générique/spécifique vont évoluer en fonction du point de vue de l'observateur (voir Fig. 28). Par exemple, imaginons deux arbres. Vu de loin, ils peuvent avoir une forme proche qui va nous amener à penser qu'ils sont identiques. Cependant, si l'on se rapproche d'eux (à quelques mètres), on va constater que leurs formes sont légèrement différentes, mais que la forme de leurs feuilles est identique. En s'approchant encore, les feuilles vont nous paraître différentes, mais pas les cellules qui les constituent, et ainsi de suite. A chaque point de vue, on trouvera des éléments qui nous paraissent identiques (généricité, abstractions) et d'autres éléments qui nous paraissent différents (spécificité, concrétisations).

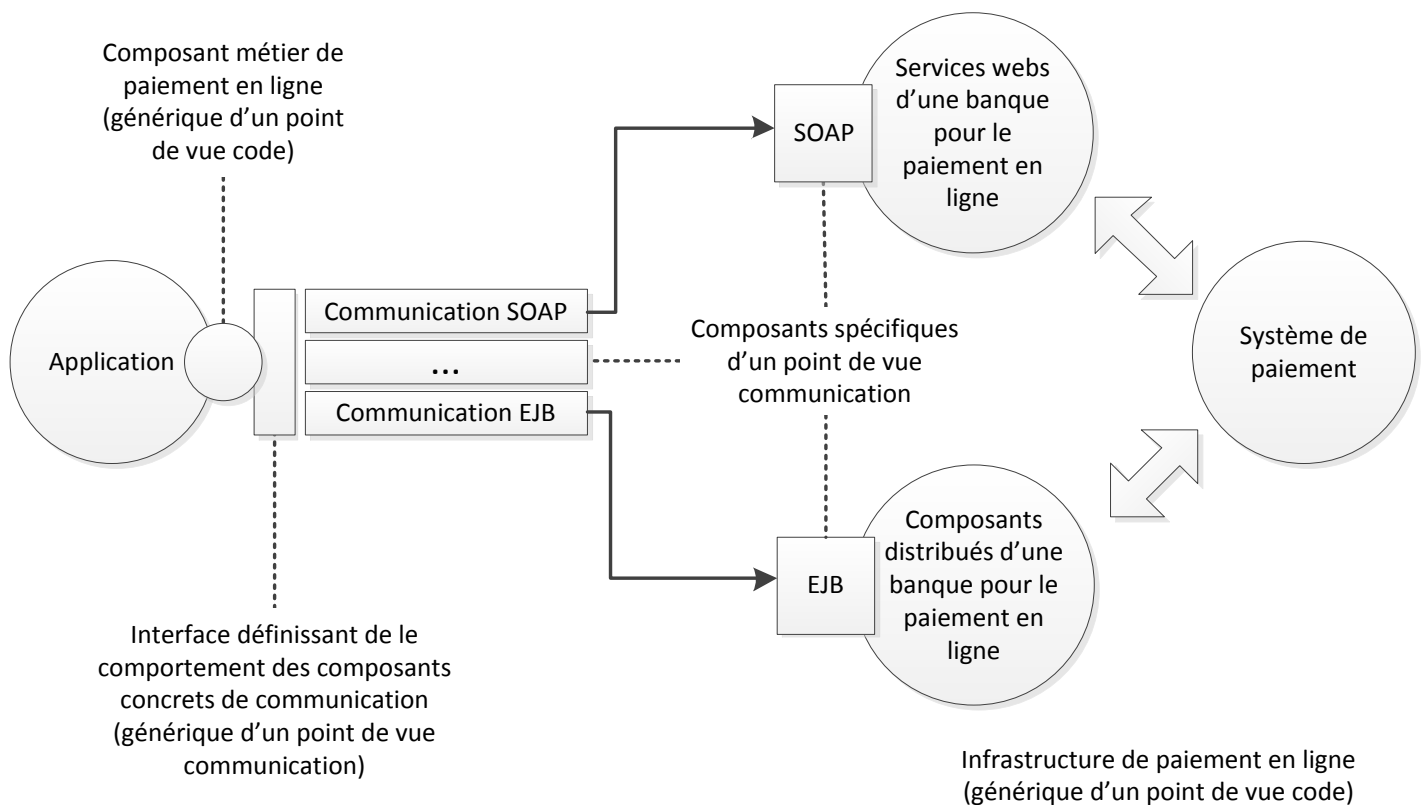


Fig. 28 – Illustration de l'importance que peut avoir le point de vue de l'observateur sur la définition des principes de spécificité/généricité

Il existe cependant une forme de factorisation reposant exclusivement sur des parties génériques : le modèle.

Un modèle est avant tout une représentation du monde réel, issu de la compréhension ou de la caractérisation de celui-ci par un individu. Il s'agit d'une vision schématique et abstraite d'un ensemble d'éléments que l'on veut décrire. En d'autres termes, tout comme un mot n'est pas ce qu'il désigne, un modèle n'est pas exactement le monde réel, mais une représentation volontairement simplifiée et incomplète de celui-ci.

De nos jours, la modélisation fait partie intégrante du processus de développement logiciel et sert à plusieurs choses, dont notamment :

- **clarifier, comprendre et représenter**, c'est-à-dire offrir une vision d'ensemble d'un système existant ou futur pour mettre en évidence des difficultés ou des défauts.
- **communiquer**, c'est-à-dire échanger autour d'une représentation unifiée de ce qui a été compris. Au travers de notations plus ou moins formalisées, graphiques ou non,

les individus vont pouvoir décrire ce qu'ils ont compris d'un problème et discuter autour d'une représentation dont le sens est le même quel que soit celui qui en prend connaissance.

Ici, les niveaux d'abstraction vont avoir toute leur importance car il sera possible d'adapter la complexité et le point de vue du modèle à son interlocuteur. Un client n'aura pas besoin de connaître les modèles proches du niveau physique, mais aura tout à gagner à prendre connaissance du modèle global pour pouvoir communiquer autour de celui-ci.

- **favoriser la compréhension extérieure**, c'est-à-dire permettre à de nouveaux intervenants de comprendre à leur tour ce qui existe, quel que soit leur niveau (nouveau prestataire, nouveau développeur, nouvel investisseur, etc.).

Cependant, la modélisation représente un effort important et, surtout, constant. Même si un modèle ne peut pas être faux par définition (on peut seulement parler de décalage entre un modèle et ce qu'il est censé représenter), celui-ci est constamment raffiné, corrigé et revu pour rester cohérent vis-à-vis de ce qu'il représente, notamment si ce qu'il représente évolue. C'est tout à fait le cas dans le processus de développement puisque l'on doit modéliser ce qui existe, mais aussi ce qui n'existe pas encore.

Au final, donc, les modèles sont présents dans tout projet informatique, mais sont rarement maintenus au-delà des phases de conception. Le système physique évolue en fonction des difficultés rencontrées, mais les modèles sont rarement mis à jour. De plus en plus de gens considèrent que la modélisation est une redondance par rapport à l'écriture du code et que les logiciels sont en fait écrits deux fois : une première fois lors de la production des documents d'analyse et une seconde fois lors de la production du code source.

Dans le cadre du multiplateforme, le modèle représente finalement une abstraction salutaire car il est par nature indépendant des considérations techniques et donc des plateformes. Ainsi, nous disposons d'une factorisation « parfaite » puisqu'à chaque modèle correspond au moins une implémentation concrète, qu'elle soit réalisée manuellement ou non.

En effet, on peut envisager de transformer le modèle, qui est notre partie générique, en code source spécifique basé sur des implémentations connues et validées. C'est ce que propose l'ingénierie dirigée par les modèles, une forme d'ingénierie générative qui consiste à produire tout ou partie d'une application informatique à partir de ses modèles.

Concepts de base : le modèle, le métamodèle et le métamétamodèle

Nous avons vu précédemment que la modélisation est une technique permettant de représenter des systèmes dont les objectifs consistent à analyser les systèmes existants, à concevoir les nouveaux systèmes et à fournir un support de communication unique entre les différents acteurs du processus de production logicielle.

Un modèle est issu d'un processus intellectuel durant lequel un observateur va sélectionner les aspects jugés pertinents d'un système en fonctions d'objectifs précis. Il s'agit d'un point de vue de l'observateur parmi tant d'autres, choisi en fonction de ce qui doit être montré, comme si cet observateur avait appliqué un filtre, une réduction sur le système, à l'image d'un dessinateur qui projette le monde réel sur un support en deux dimensions.

Le simple fait de comprendre un système est déjà un travail de modélisation, même si celui-ci reste informel. Néanmoins, pour qu'un modèle puisse être transformé de manière automatique ou semi-automatique, il faut que la machine soit en mesure de le manipuler. Celui-ci doit donc être exprimé formellement dans un langage dont la définition sera appelée métamodèle, conformément à la définition suivante (OMG, 2010) :

"A metamodel is a model used to model modeling itself."

Un métamodèle est donc un modèle représentant un langage de description de modèle. On confond souvent le langage et son métamodèle car la différence entre les deux concepts est assez subtile, cependant le métamodèle peut être considéré comme la grammaire du langage (les éléments disponibles et leur logique d'assemblage). Par exemple, UML est un langage graphique permettant de décrire des modèles et UML est lui-même doté d'un métamodèle qui définit l'ensemble des éléments de construction disponibles, leurs manières d'interagir et leurs significations (sémantique) compréhensibles de la même façon par tous ceux qui en ont connaissance.



65

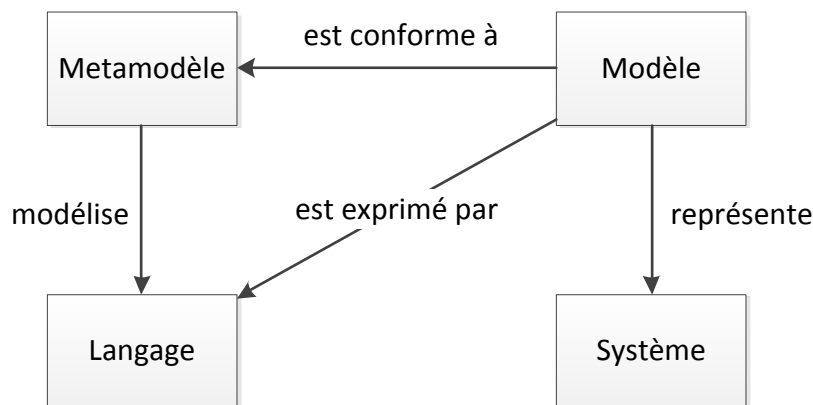


Figure 30 – Le système, le modèle, le langage et le métamodèle

Formellement, un métamodèle peut être découpé en deux parties distinctes :

- **la terminologie**, c'est-à-dire l'ensemble des concepts à disposition, leurs propriétés et leurs relations formalisées de manière générique pour être réutilisées par une multitude de domaines (on peut imaginer un métamodèle dédié aux modèles représentant le fonctionnement des universités, par exemple). On parlera parfois de Domain Specific Language^a (DSL).
- **les assertions**, c'est-à-dire les règles supplémentaires destinées à contraindre les éléments décrits par la terminologie.

Les métamodèles posent toutefois un problème singulier. En effet, tout comme le modèle avant lui, le métamodèle n'est pas soumis à une syntaxe spécifique. Il est possible d'en définir un ou plusieurs pour une multitude de domaines, ce qui pourrait conduire à un accroissement chaotique de leur nombre. C'est pour cela qu'est apparu le concept de métamétamodèle, c'est-à-dire un métamodèle permettant de décrire des métamodèles. Etant donné que le métamétamodèle est un métamodèle, il possède la particularité de pouvoir se décrire lui-même (métacircularité).

Au final, tout ceci s'inscrit dans une architecture quatre niveaux, M0 à M4, où M0 représente les systèmes concrets, M1 les modèles, M2 les métamodèles et M3 le seul et

^a Domain Specific Language par opposition à General Purpose Language, c'est-à-dire un langage de type généraliste. Un langage généraliste est conçu pour donner des instructions à la machine (Java et C++ par exemple), ce qui permet de pouvoir créer des programmes pour n'importe quel domaine. A contrario, le langage dédié est conçu pour résoudre des problèmes spécifiques à un domaine défini (par exemple, le langage *csound* est dédié à la production sonore).

unique métamétamodèle. A partir de cela, on peut considérer qu'un élément du niveau M_n est une instance d'un élément du niveau M_{n+1} (avec $0 \leq n \leq 2$).

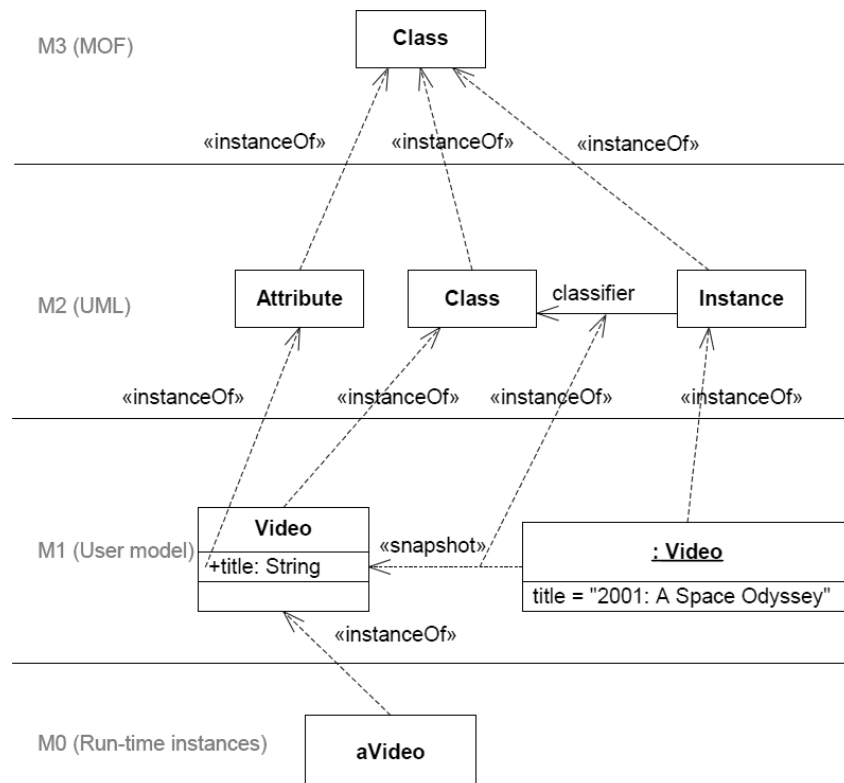


Fig. 31 – Exemple d'application de la hiérarchie à quatre niveaux (OMG, 2011)

Le MOF

Le Méta-Object Facility est un métamétamodèle adopté en 1997 par l'Object Management Group. Il est issu des travaux sur la partie réflexive d'UML (c'est-à-dire la capacité pour UML de se décrire lui-même), ce qui explique les similitudes entre le formalisme du MOF et celui d'UML bien que les deux spécifications évoluent de manière indépendante aujourd'hui.

Typiquement, un métamodèle basé sur le MOF est un paquetage qui contient un ensemble de classes reliées entre elles par des associations et potentiellement regroupées en sous-paquetages. Une classe peut embarquer des attributs et des opérations. Le tout peut être précisé au moyen de contraintes.

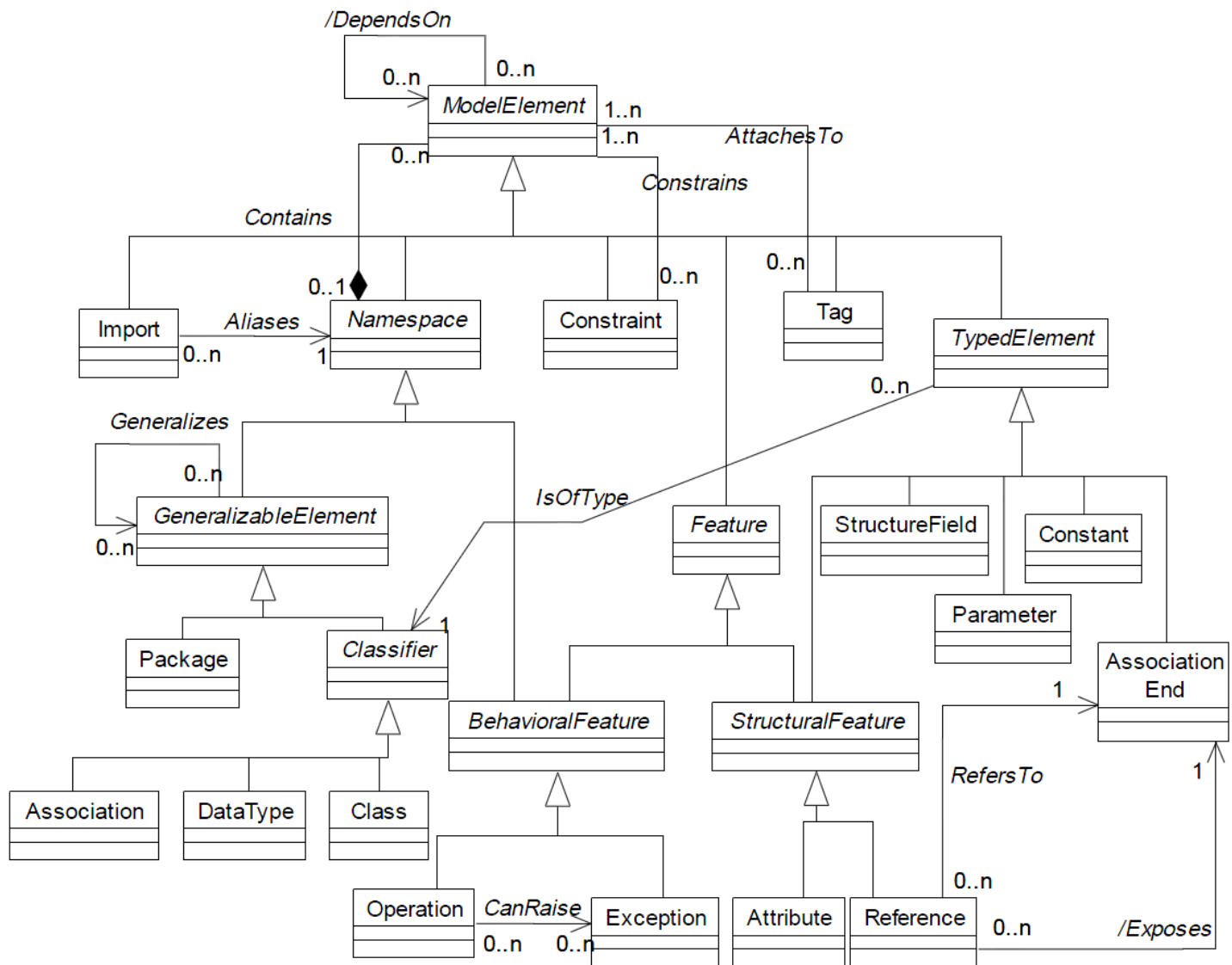


Fig. 32 – Extrait du MOF (OMG, 2010)

L'approche MDA (Model Driven Architecture)

L'architecture pilotée par les modèles a pour objectif la construction de modèles indépendants des détails techniques des plateformes d'exécutions, et la transformation de ceux-ci en modèles code source. La principale problématique revient à transformer les modèles du niveau M1 en systèmes techniques du niveau M0, en exploitant les métamodèles du niveau M2 formalisés au travers du métamétamodèle unique du niveau M3.

L'architecture pilotée par les modèles définit un processus bien particulier pour réaliser cette transformation. L'entrée dans le cycle de développement s'effectue au travers d'un modèle d'exigence appelé CIM (Computation Independant Model). Concrètement, ces

modèles peuvent être comparés à des diagrammes de cas d'utilisation et n'auront que peu d'incidence sur la génération de code proprement dite. Il s'agit plutôt d'un modèle contractuel destiné à être validé par le client, bien qu'un lien puisse être tracé entre le CIM et l'ensemble des autres modèles, jusqu'au code source final.

Une fois le CIM validé, on va pouvoir créer les modèles indépendants de la plateforme (PIM, ou Platform Independent Model), parfois appelés « modèles d'analyse et de conception abstraite ». Ces modèles se concentrent sur une logique métier sans que les considérations techniques soient abordées, non seulement pour l'implémentation, mais aussi pour le déploiement. Il s'agit là d'un effort de conception et de modélisation qui va permettre produire les parties génériques de notre application. Les implémentations concrètes n'arriveront que plus tard dans le cycle de développement, pour assurer une meilleure séparation des univers génériques/spécifiques.

A noter qu'il est possible de définir les PIM au travers de nombreux langages spécifiques. On pourra utiliser UML mais aussi de nombreux DSL, généralement de petites tailles et adaptés à des modélisations spécifiques (par exemple, le langage CWM est dédié aux bases de données).

Une fois les PIMs correctement définis (ceux-ci doivent être suffisamment précis pour pouvoir effectuer une génération de code), les aspects de génération de code proprement dit pourront être abordés au travers des PSMs (Platform Specific Model) ou « modèle de code » ou « modèle de conception concrète ». Il s'agit cette fois de définir un modèle respectant le PIM, mais lié à une plateforme d'exécution. Ce sont ces nouveaux modèles qui pourront être transformés en code source par une machine puisque les détails d'implémentation liés à chaque plateforme auront été définis.

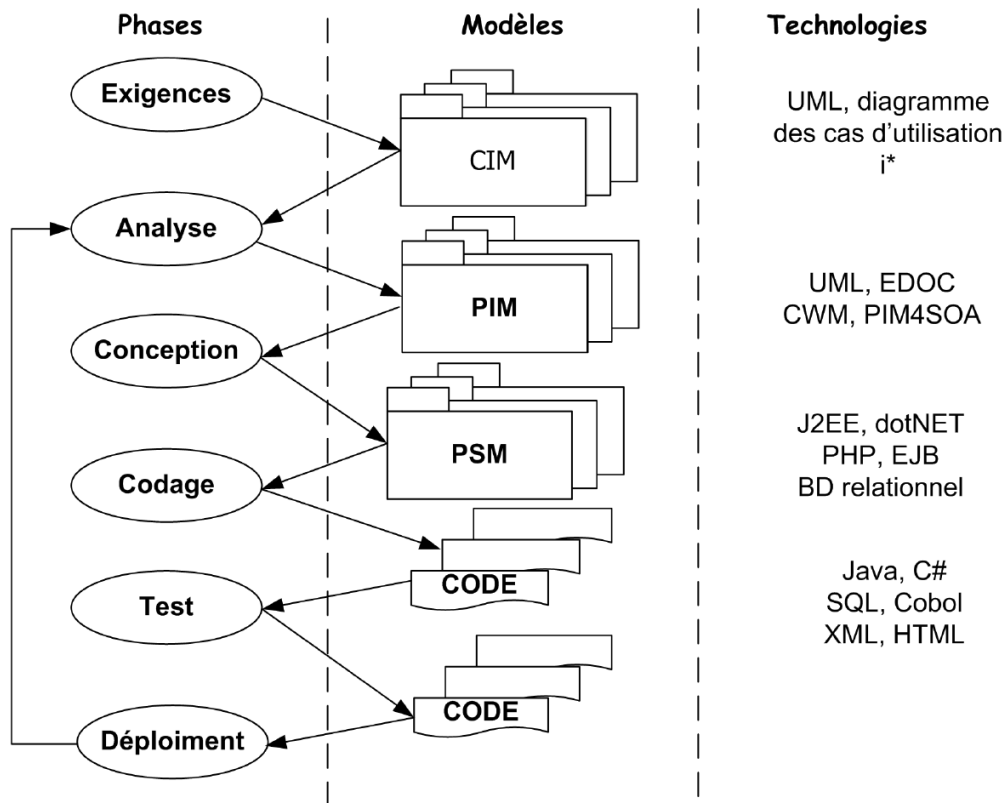


Fig. 33 – Processus de développement selon l'approche MDA (Hammoudi, 2010)

Les métalangages

L'UML (Unified Modeling Language) est le langage de modélisation préconisée par l'OMG, bien que celui-ci ne permette pas, à l'origine, de représenter un programme complet de manière suffisamment précise pour que l'opération de transformation du modèle en code soit triviale. En effet, comme les langages de programmation sont généralement peu abstraits puisqu'ils ont pour vocation de transmettre des instructions aux machines, le modèle doit être extrêmement précis (par exemple, le modèle doit définir d'une manière ou d'une autre des corps de méthodes). Aussi, UML a été étendu avec des métalangages, c'est-à-dire des pseudos-langages permettant de décrire certaines choses.

Le principal avantage de ces métalangages réside dans leur simplicité de manipulation, car ceux-ci se révèlent nettement plus flexibles que l'habituelle notation graphique d'UML. Bien entendu, les deux notations (graphique et textuelle) restent possibles car ces métalangages peuvent être décrits sous forme de métamodèles.

On distinguera, par exemple, le langage OCL (Object Constraint Language) qui permet de décrire des contraintes à l'image de ce que l'on pourrait rencontrer dans une programmation par contrats (préconditions, postconditions, invariants, etc.).

De même, il existe un métamodèle AS (Action Semantics) permettant de décrire des constructions d'objet, des affectations de variables, des suppressions, des boucles, des tests, etc. Cependant, le métamodèle AS ne possède pas encore de syntaxe textuelle comme c'est le cas pour OCL.

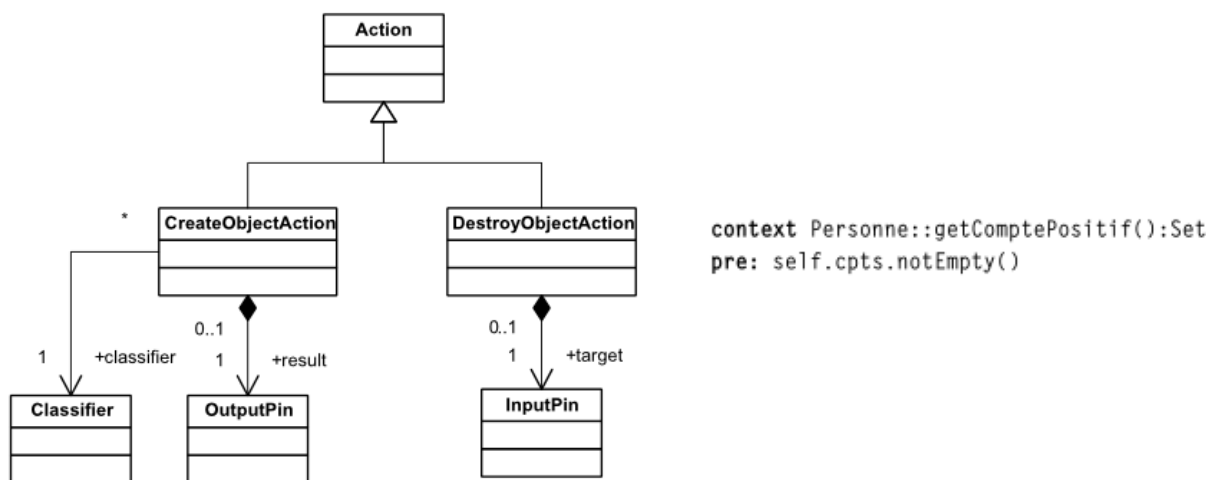


Fig. 34 – Extrait du métamodèle AS représentant la construction et la destruction d'objet (à gauche) et exemple de contrainte OCL (à droite)

Pourquoi modéliser ?

Les avantages d'une approche dirigée par les modèles sont plutôt évidents. En effet, la phase de génération de code nous permet de nous affranchir définitivement de la notion de plateforme, si tant est que le modèle soit suffisamment abstrait pour cela et que les plateformes de destination soient prises en charge par le générateur. Ainsi, le modèle devient notre abstraction unique déclinable pour n'importe quel système, n'importe quelle plateforme d'exécution, n'importe quel environnement de support.

Au-delà de cela, il faut noter que les phases de modélisation sont consommatrices de temps et doivent être réalisées quel que soit le cas, ingénierie dirigée par les modèles ou non. De fait, la possibilité de générer directement tout ou partie d'une application à partir des modèles est un gain de temps considérable dans la phase de production logicielle, mais aussi

dans la phase de test car, pour peu que le modèle soit valide, la génération de code ne peut pas faire d'erreur en elle-même si tant est qu'elle soit fiable.

De la même façon, l'évolution et la maintenance sont facilitées puisqu'il n'est pas nécessaire de réécrire du code lorsque le modèle est amélioré ou corrigé. Cela rejoint en partie le principe de source unique transformée pour chaque plateforme cible, tel que décrit précédemment (p. 30 et p. 40).

Enfin, l'approche dirigée par les modèles favorise la capitalisation des connaissances. Comme les modèles sont abstraits, ils sont réutilisables et adaptables par nature, ce qui en fait des candidats idéaux pour la constitution d'une bibliothèque de modèles pouvant être intégrés plus tard, en tout ou partie, à des modèles plus vastes.

Cependant, il s'agit là d'avantages théoriques. Les approches dirigées par les modèles ne sont pas suffisamment standardisées à l'heure actuelle (par exemple, une partie de MDA n'a pas été achevée par l'OMG).

De plus, la construction et la transformation de modèle ne sont pas des opérations simples, surtout en sachant que le niveau de précision requis est élevé pour que la génération soit efficace. Par conséquent, l'effort de développement est en partie reporté sur l'effort de modélisation.

4 - Un cas pratique : la téléphonie mobile

Pour illustrer les propos tenus tout au long de mon mémoire, j'ai choisi l'univers de la téléphonie mobile comme toile de fond car celui-ci, de par son développement rapide et relativement récent, remet au goût du jour l'ensemble des difficultés liées au développement multiplateforme.

4.1 - La téléphonie mobile, un univers multiplateforme par excellence

La téléphonie mobile présente certains traits caractéristiques des domaines multiplateformes. Tout d'abord, les plateformes d'exécutions (système d'exploitation) et les architectures sont nombreuses et se développent anarchiquement. En effet, le marché est en forte croissance et sa pénétration mondiale est très importante, ce qui favorise l'ébullition technologique et la fabrication à l'emporte-pièce de terminaux mobiles par une multitude de producteurs de matériel et de logiciel dont l'objectif est de conserver son marché plutôt que d'assurer une saine pérennité aux technologies produites.

De fait, les terminaux ont des philosophies de fonctionnement différentes, des capacités différentes et des cibles (consommateurs) différentes qui font de chaque modèle de téléphone un modèle presque unique dans l'univers des modèles disponibles.

Enfin, l'ébullition et l'aspect lucratif du marché de la téléphonie nécessitent, pour les producteurs d'applications, de devoir prendre en compte l'ensemble de ces spécificités pour proposer des applications dont la visibilité sera maximale et qui seront, de surcroît, en mesure de communiquer, d'une part entre elles et d'autre part avec l'univers applicatif qui existe déjà.

4.1.1 - Un certain nombre de plateformes mobiles

Le paysage des téléphones mobiles a toujours été diversifié, mais c'est véritablement l'explosion du marché des smartphones qui a offert des possibilités de plus en plus intéressantes en termes de connectivité et de capacité de calcul. A l'heure actuelle, un téléphone vendu sur quatre (Gartner, 2011) est un smartphone, ceux-ci ayant connu une croissance de 74% entre le deuxième trimestre 2010 et le deuxième trimestre 2011.

Au-delà des spécificités matérielles, on distinguera un certain nombre de systèmes d'exploitation pour téléphones mobiles (liste non exhaustive) :

- BlackBerry OS (RIM)

- Android (Google)
- iOS (Apple)
- Windows Phone et Windows Mobile (Microsoft)
- Symbian (Nokia)
- WebOS (HP)
- téléphones à base Java Mobile Edition (tous constructeurs confondus)

Chaque système d'exploitation mobile est généralement accompagné par une couche logicielle ajoutée par le constructeur du téléphone physique. De même, chaque système d'exploitation est décliné en multiples versions, celles-ci n'étant pas toujours mises à jour sur les téléphones déjà en circulation.

De fait, on assiste à une multiplication des plateformes d'exécutions potentielles qui ne fournissent pas les mêmes fonctionnalités logicielles, les mêmes langages et les mêmes environnements de travail.

4.1.2 - Un marché inégal et des disparités importantes entre les constructeurs

La répartition du marché entre les fournisseurs de systèmes d'exploitation est assez inégale comme le montrent la table et la figure ci-dessous.

Système d'exploitation smartphone	Part de marché 2011
Android	39.5%
BlackBerry	14.9%
iOS	15.7%
Symbian	20.9%
Windows Phone 7/Windows Mobile	5.5%
Autres	3.5%
Total	100.0%

Tableau 1 – Répartition des différents systèmes d'exploitation pour smartphones dans le monde (IDC, 2011)

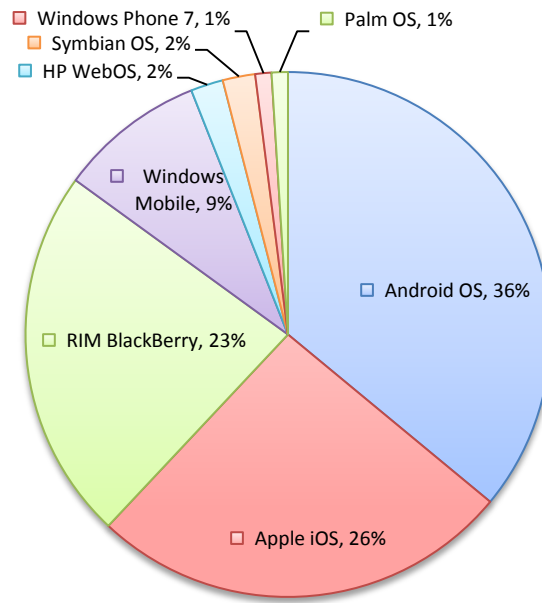


Fig. 35 – Part de marché des smartphones aux Etats-Unis (Nielsen, 2011)

Dans la suite de mon propos, je me restreindrais à l'ensemble composé des systèmes d'exploitation Android, iOS et BlackBerry, étant donné qu'ils représentent 85% du marché, ainsi qu'au Windows Phone 7 (dont la faible répartition peut actuellement s'expliquer par sa récente apparition) et aux téléphones portables ancienne génération^a. Cette restriction est justifiée par la disparition de WebOS et le déclin de Symbian OS ainsi que par la liste des plateformes prises en compte lors des travaux effectués en stage.

D'un point de vue technique, ces différentes plateformes ne fonctionnent pas de la même façon et ne disposent pas des mêmes possibilités pour le développeur que ce soit au niveau des langages, des bibliothèques graphiques, des frameworks, etc.

Par exemple, les smartphones BlackBerry et les smartphones Android, ainsi que l'ensemble des téléphones portables à base de Java Mobile Edition^b, imposent un développement à

^a Téléphone portable en opposition à smartphone, c'est-à-dire tous les téléphones mobiles classiques qui ne sont pas commercialement déclinés en tant que smartphones.

^b La technologie Java est déclinée en trois versions : Standard, Entreprise et Mobile. Java Standard Edition (JSE) et Java Enterprise Edition (JEE) sont respectivement destinés à la fabrication d'applications de bureau et à la fabrication d'applications d'entreprises (applications web ou distribuées, par exemple). Java Mobile Edition (JME) est, quant à lui, destiné aux systèmes embarqués et est pourvu d'une version réduite du framework Java ainsi que d'une machine virtuelle Java adaptée à ce type de périphériques.

base du langage Java. Malgré l'utilisation du même langage, les frameworks diffèrent significativement. Java Mobile Edition embarque un framework limité par rapport à la version Standard, BlackBerry y ajoute une couche et un framework propriétaire tandis qu'Android utilise une machine virtuelle Java spécifique (Dalvik) et un framework qui lui est propre.

Les smartphones Apple, équipés du système d'exploitation iOS, nécessitent de développer au moyen du langage Objective C.

Enfin, les smartphones Windows Phone impliquent un développement en C#, basé sur un sous-ensemble du framework Silverlight, initialement prévu pour le développement web. Sous Windows Mobile, le framework était bien plus complet (proche de la version de bureau), mais a été abandonné pour repartir sur une base nouvelle.

Les langages ne sont pas le seul problème, la philosophie des différents frameworks diffère elle aussi, ce que l'on peut observer au travers deux fonctionnalités de base : le stockage et la communication.

Au niveau stockage, les smartphones sont généralement dotés d'une mémoire volatile et d'un stockage physique (généralement une carte SD). La philosophie d'Android est de proposer plusieurs types de stockages, exploitables en fonction des cas et des capacités du téléphone. On distinguera alors un espace de fichier partagé dédié aux préférences des applications, un stockage interne visible uniquement par l'application qui l'a initialisé, un stockage externe (la carte SD), des répertoires publics (pour la musique, les images, les vidéos, etc.) et une base de données embarquée.

A contrario, iOS présente un support de stockage arborescent propre à l'application et composé de répertoires spécifiques persistants ou non. Les applications peuvent accéder indirectement (au travers d'opérations spécifiques) aux données du téléphone comme les images, les vidéos.

Encore plus restrictif, le Windows Phone présente le support de stockage comme un espace virtualisé (isolated storage), persistant et de taille fixe, visible uniquement par l'application qui l'a initialisé, sans aucun partage possible (aucune application ne peut consulter le contenu de l'isolated storage, pas plus que l'utilisateur). L'accès aux informations du téléphone est totalement indirect, s'effectuant au travers d'interfaces graphiques propres au

système d'exploitation (exactement comme des boîtes de dialogue standardisées). A noter que la mise à jour Mango (septembre 2011) apporte une base de données embarquée, directement accessible au travers de la technologie de mapping objet-relationnel LINQ.

En ce qui concerne la communication, les téléphones à base de JME disposent des procédés de communication classiques (sockets unicast, broadcast et multicast), utilisable de manière synchrone ou asynchrone.

Côté Windows Phone, la communication est extrêmement limitée. Le téléphone ne peut manipuler que des connexions HTTP (services web) asynchrones, à raison de 6 connexions simultanées. A noter qu'une mise à jour, Mango (septembre 2011), permet d'utiliser des sockets clients^a asynchrones, limités à une trentaine de connexions simultanées.

De même, l'iOS propose des connexions asynchrones, sans toutefois se restreindre aux services web.

Ces disparités significatives peuvent malheureusement être observées à tous les niveaux et complexifier le processus de développement :

- accès aux informations stockées sur le téléphone (contacts, calendrier, etc.)
- utilisation des fonctionnalités du téléphone (appels, envoi d'email et de SMS, recherche web, etc.)
- support de la 3D
- encodage des caractères
- support du multimédia (lecteur multimédia, protocoles de streaming, formats supportés)
- sécurité (algorithmes de chiffrement, gestion des certificats, etc.)
- interface graphique

4.1.3 - Une effervescence aux enjeux importants

Comme je l'ai exposé plus tôt, le marché de la téléphonie mobile est en pleine expansion, siège d'une lutte entre les habituels acteurs de l'informatique grand public. Cependant, les

^a Ces sockets ne permettent pas au téléphone d'écouter des communications entrantes, elles ne peuvent que contacter une socket serveur. La communication reste donc toujours à l'initiative du téléphone.

smartphones s’immiscent aussi dans le monde de l’entreprise, remplaçant peu à peu les assistants personnels informatisés (PDA).

En considérant une entreprise spécialisée dans le développement d’application pour les terminaux mobiles, un dilemme apparaît immédiatement : « quelle plateforme vais-je privilégier ? ». Malheureusement, les parts de marchés de chaque acteur sont faibles (au maximum quarante pour cent) au vu de ce que l’on connaît dans le monde de l’informatique de bureau (plus de trois quarts des ordinateurs personnels fonctionnent avec une version de Windows). De fait, se concentrer sur une seule plateforme en particulier n’est absolument pas une bonne idée car ce serait se restreindre à une faible part des utilisateurs, dans le meilleur des cas.

L’effervescence de la téléphonie mobile impose donc aux entreprises productrices de logiciels mobiles de tenir compte de plusieurs plateformes pour pouvoir disposer d’une visibilité honorable sur le marché de la téléphonie mobile (70% environ du marché est tenu par iOS, Android et BlackBerry, ce qui représente déjà une certaine quantité de travail). De plus, l’effervescence du marché est telle qu’il est difficile de prédire qui sera un acteur majeur de la téléphonie mobile de demain. Il faut garder à l’esprit qu’il y a encore deux ans, le système Android occupait une part de marché beaucoup plus faible. Qui sait, donc, si demain les Windows Phone n’occuperont pas une place bien plus importante ?

Cet aspect mouvant, couplé au marché fragmenté et à la disparité entre les constructeurs, oblige les entreprises à se concentrer sur plusieurs plateformes. Ces enjeux sont d’autant multipliés par le fait que la téléphonie mobile est un support récent, présentant de formidables apports en termes de communication et de visibilité. Hélas, ces apports ne sont réels que si les applications produites sont exhaustives et se concentrent sur un panel de plateformes, ce qui redonne aux problématiques multiplateformes une nouvelle raison d’être.

4.2 - Un middleware pour rationaliser la communication entre les plateformes et le développement d'application de haut niveau

Nous avons vu tout au long de ce document que le multiplateforme pouvait être simplifié en fournissant des abstractions logicielles et intergicielles. Aussi, le monde de la téléphonie mobile étant un monde fortement distribué (composés de petits périphériques communiquant entre eux ou avec le cloud), l'approche la plus efficace consiste à disposer d'une abstraction globale des fonctionnalités classiques des téléphones (stockage, communication, etc.), exploitable sur une majorité de terminaux.

4.2.1 - L'INRIA, l'équipe ARLES et la startup Ambientic

L'Institut National de Recherche en Informatique et Automatique (INRIA) est un établissement public à caractère scientifique et technologique (EPST) destiné à mettre en réseau les compétences de l'ensemble du dispositif de recherche français, dans le domaine des sciences et technologies de l'information et de la communication (STIC).

L'équipe de recherche ARLES (ARchitecture Logicielle Et Systèmes distribués), dans laquelle j'ai effectué ma mission, se situe dans le centre de Paris-Rocquencourt (l'INRIA comptant huit centres répartis sur toute la France) et s'intéresse à l'étude des systèmes logiciels distribués, plus spécifiquement à la définition de nouvelles architectures logicielles de systèmes, basées sur l'exploitation des technologies réseaux émergentes.

Actuellement, les travaux de l'équipe ARLES relèvent plus précisément de deux axes complémentaires :

- **la définition de langages, méthodes et outils** pour le développement de systèmes logiciels distribués, par composition dynamique de ressources numériques déployées dans le réseau. Il s'agit ici de définir les abstractions associées aux ressources pour en permettre la découverte, l'accès, la composition et l'interopérabilité tout en garantissant des propriétés de qualité (disponibilité, fiabilité, sécurité), d'adaptabilité au contexte ou de respect de la vie privée.
- **la définition de nouvelles infrastructures middleware** pour la réalisation de l'informatique diffuse dans les réseaux hybrides combinant notamment capteurs,

réseaux sans fil ad hoc et à infrastructure. Il est ici essentiel d'étudier des solutions à la gestion de systèmes distribués hétérogènes, qui intègrent les capacités limitées de certaines de leurs ressources, ou encore la possible connectivité via différentes liaisons radio. Nos travaux portent ainsi sur l'étude d'algorithmes et de protocoles distribués pour le partage et la coordination des ressources numériques dans des environnements multi-radios, multi-réseaux et multi-protocoles.

Dans cette optique, l'équipe ARLES a développé, au fil des ans, une expertise technologique qui a été peu à peu capitalisée sous forme de bibliothèques et de technologies diverses. Il y a quelque temps, certains membres de l'équipe ont monté une startup, Ambientic, dont l'objectif réside dans le transfert de ces technologies sous la forme d'un middleware baptisé iBICOOP. Ce transfert effectué, la société Ambientic se destine à la fabrication d'applications commerciales pour les téléphones mobiles, basées sur ce middleware.

4.2.2 - iBICOOP, un middleware pour unifier les échanges entre terminaux mobiles

Les middlewares, abréviation de « middle software », permettent de faire communiquer des applications en fournissant des interfaces de médiation (adaptation, transformation, etc.) permettant l'échange d'informations.

Le middleware, qui peut être une simple couche logicielle, va en quelque sorte servir de « ciment » entre plusieurs applications. Une application très répandue des middlewares consiste à permettre à des logiciels de manipuler de multiples bases de données différentes en fournissant une abstraction des concepts sous-jacents (ex : JDBC).

Classification des middlewares

On distingue quatre grandes familles de middleware, c'est-à-dire quatre grandes façons d'échanger de l'information (Issarny et al., 2011) :

- **l'appel de procédure distante** (remote procedure call), c'est-à-dire l'invocation de traitements distants exposés entre les applications au travers du middleware (Fig. 36).
- **l'évènementiel**, c'est-à-dire la capacité pour des applications de s'enregistrer à des files d'évènements gérées par le middleware, un peu comme sur un forum (Fig. 36).

- **le partage d'espace**, c'est-à-dire la communication au travers d'un flux de données partagé entre plusieurs applications et maintenu par le middleware (Fig. 37).
- **l'échange de message**, c'est-à-dire la capacité pour les applications de communiquer par messages au travers du middleware, un peu comme dans un échange d'email (Fig. 37).

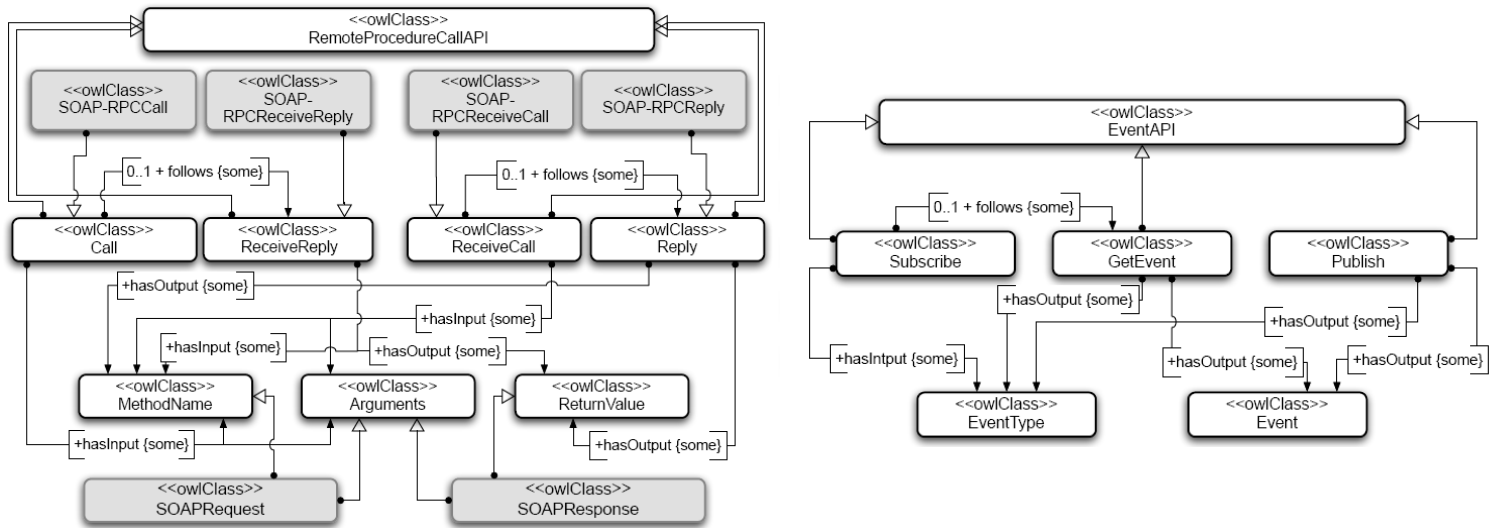


Fig. 36 – Middleware RPC (à gauche) et middleware orienté événements (à droite)

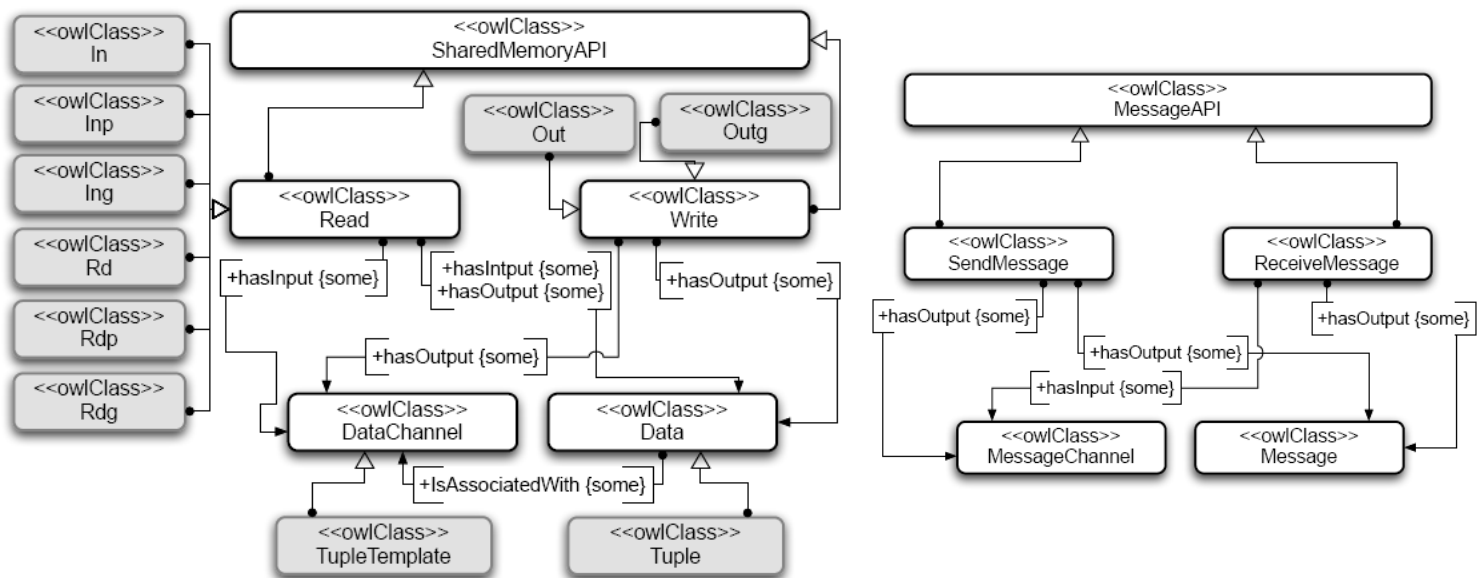


Fig. 37 – Middleware à espace partagé (à gauche) et middleware orienté message (à droite)

Ces différentes philosophies de fonctionnement permettent toutes à des applications de communiquer, avec plus ou moins de limitations en fonction de la nature des données échangées et de leur volume (par exemple, il est préférable d'utiliser un middleware à mémoire partagée pour virtualiser des stockages distribués tandis qu'il est préférable d'utiliser des appels de procédure distante pour distribuer des traitements).

De plus, il s'agit d'une classification stricte, il est rare qu'un middleware appartienne à une famille exclusive, il y a généralement des chevauchements (par exemple, les bus d'entreprises adoptent souvent une approche par message conjointement à une approche par appel de procédure distante).

Présentation du middleware iBICOOP et de son infrastructure

Le middleware iBICOOP a été conçu pour permettre à des périphériques hétérogènes (principalement des téléphones mobiles, bien que les versions Java soient déclinées pour les applications de bureau et les applications embarquées) de communiquer de manière standardisée.

Plus généralement, le middleware iBICOOP offre une abstraction permettant de manipuler les différents aspects d'un téléphone indépendamment de sa nature : communication, sécurité, stockage et gestion des informations personnelles. Grâce à cela, les applications développées par-dessus iBICOOP peuvent être portées très facilement sur tout terminal mobile en mesure d'accueillir le middleware, c'est-à-dire l'ensemble des plateformes suivantes :

- Téléphones et périphériques embarqués à base de Java Mobile Edition et dérivés (RIM/BlackBerry par exemple)
- Téléphones sous iOS
- Téléphones et périphériques embarqués sous Android
- Téléphones sous Windows Phone, périphériques embarqués sous Windows CE
- Ordinateurs équipés d'un système d'exploitation Windows, Linux, Mac, etc. (déclinaison d'iBICOOP sous Java Standard Edition)

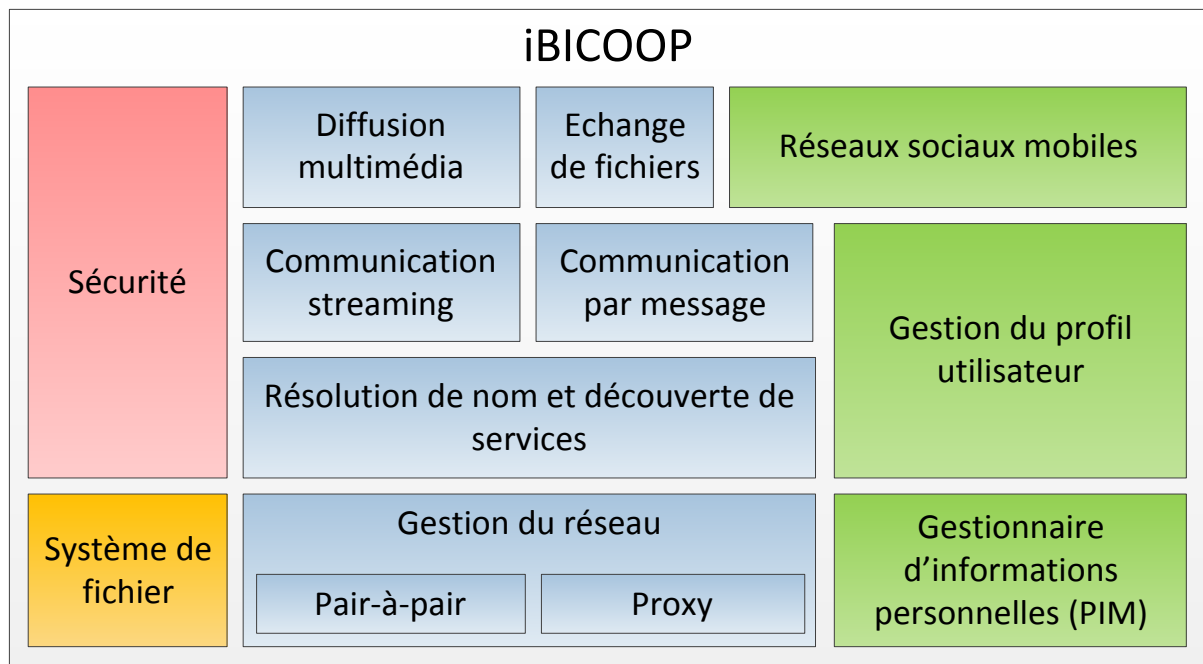


Fig. 38 – Architecture du middleware iBICOOP

Tel que visible sur la figure ci-dessus, le middleware prend en charge un certain nombre d'aspects liés à la manipulation du téléphone. Tout d'abord, la manipulation du stockage (lecture/écriture) est unifiée ainsi que les aspects liés à la sécurité (chiffrement, manipulation de certificat, keystore, hashage et signature). De même, il fournit des abstractions pour la manipulation des informations stockées par l'utilisateur (contacts, agenda, documents, informations personnelles, etc.) et pour leur sécurisation (échanges sécurisés, filtres, partenariats temporaires, traçabilité, synchronisation, authentification, calculabilité de la confiance, etc.).

Enfin, le middleware fournit une infrastructure pour la communication entre les téléphones. Celle-ci comprend plusieurs composants :

- des connexions TCP/IP pair-à-pair (unicast, broadcast et multicast) sur plusieurs types de réseaux (Wifi, 3G, Edge, etc.). Ces connexions sont de type simple^a car certains téléphones ferment automatiquement les connexions après l'envoi d'un message.

^a Les canaux de communication peuvent être catégorisés en trois types : simplex, half-duplex et full-duplex. Un canal simplex ne permet une communication qu'à un seul sens (envoi ou réception), un canal half-duplex autorise une communication à double sens, mais sans que l'envoi et la réception puissent être réalisés simultanément et enfin, un canal full-duplex permet une communication parallèle dans les deux sens.

- des structures spécifiques pour effectuer des communications par envoi de message ou en flot continu (streaming), le tout de manière synchrone ou asynchrone.
- un service de nommage et de résolution pour les différents points d'entrées et de sorties du téléphone sous la forme d'urls spécifiques : *ibiurl://<id du terminal>/<id de l'utilisateur>/<id de l'application>/<id de service>/<chemin personnalisé>*.
- un service de découverte basé à la fois sur une détection des périphériques connectés à un même réseau (découverte broadcast), sur une détection de groupe (découverte multicast), sur une détection par annuaire (découverte par repository) et sur un cache local.
- des services de plus haut niveau pour l'échange de fichier (avec reprise), la diffusion multimédia et la synchronisation de données.

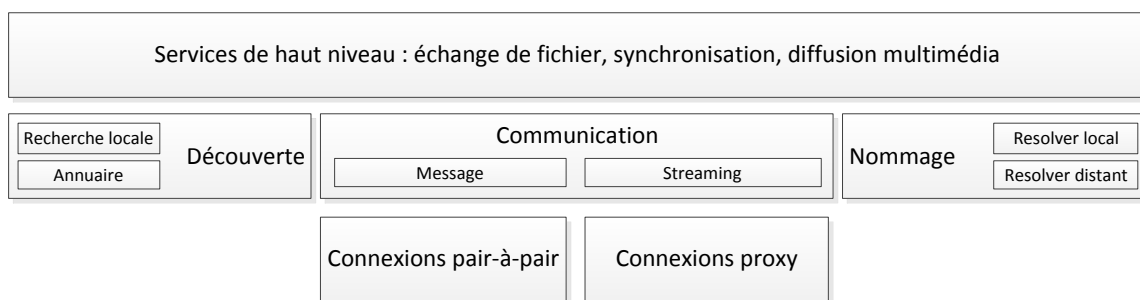


Fig. 39 – Les couches de communication d'iBICOOP

Cette infrastructure de communication peut fonctionner en totale autarcie, grâce aux communications directes entre les téléphones, ou fonctionner au travers d'une infrastructure propre au middleware, le tout étant réalisable au sein d'un réseau ad-hoc ou au travers d'internet.

L'infrastructure qui entoure le middleware est composée d'un serveur de communication (proxy), d'un serveur de ressources (repository) et d'un serveur d'authentification (security). Au final, ces trois serveurs prennent en charge des aspects distincts de la communication.

Le serveur d'authentification (security) permet à un tiers d'enregistrer des identifiants et des certificats qui seront utilisés plus tard pour une authentification sous forme de challenge à résoudre (voir Fig. 40). De même, ce serveur permet aux clients de communiquer de manière sécurisée avec les deux autres serveurs composant l'infrastructure.

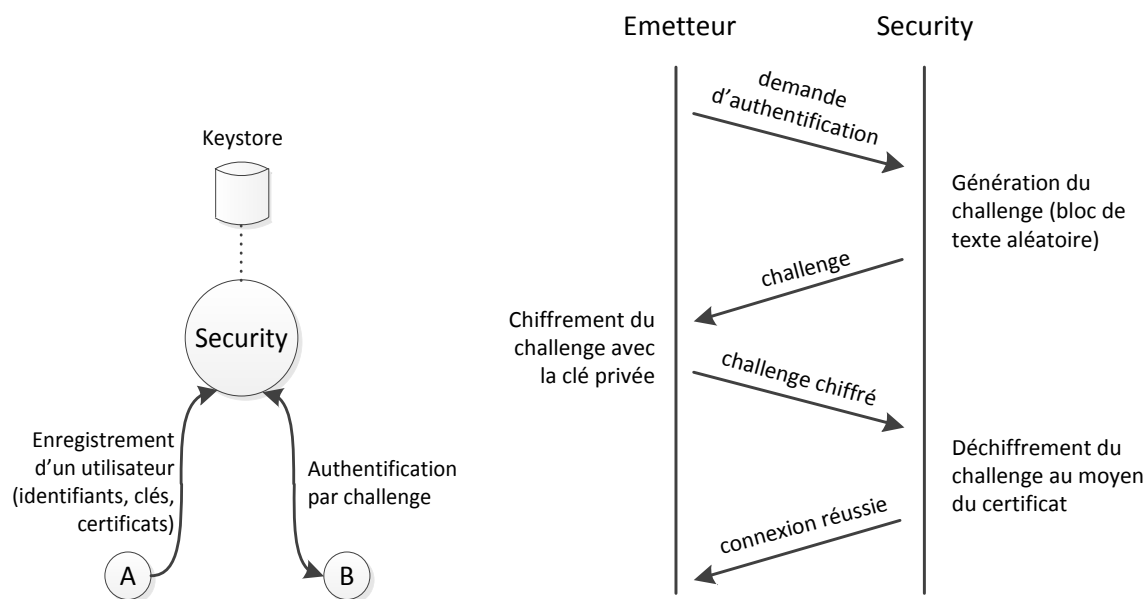


Fig. 40 – Le serveur d'authentification (à gauche) et la procédure standard d'authentification (à droite)

Le serveur de communication (proxy) permet quant à lui d'assurer la communication entre des périphériques qui ne pourraient pas se voir directement. Les requêtes et les réponses sont stockées sur ce serveur en attendant d'être récupérées par ceux à qui elles sont destinées (voir Fig. 41). De même, il permet aux terminaux de communiquer au-delà de leur réseau local, en accomplissant un rôle de médiateur, à l'image d'une messagerie : les différents acteurs déposent des messages et viennent vérifier régulièrement si de nouveaux messages leur sont destinés.

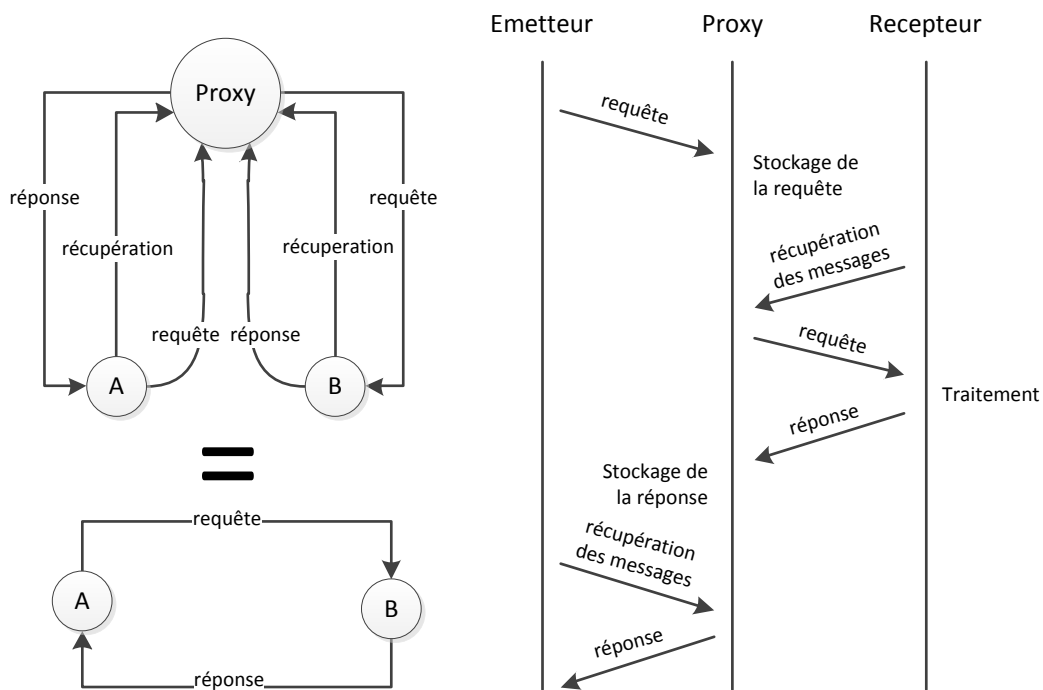


Fig. 41 – Le serveur de communication (à gauche) et une communication standard en mode requête/réponse (à droite)

Enfin, le serveur de ressources (repository) va jouer un rôle d'annuaire et un rôle d'espace partagé pour l'échange de ressources. En tant qu'annuaire, il enregistre les points d'accès (émission/réception) des terminaux et les informations associées (périphérique, utilisateur, etc.) pour permettre à ceux-ci de se découvrir et d'établir des communications. En tant qu'espace de partage, il enregistre des ressources qui pourront être récupérées par les autres terminaux.

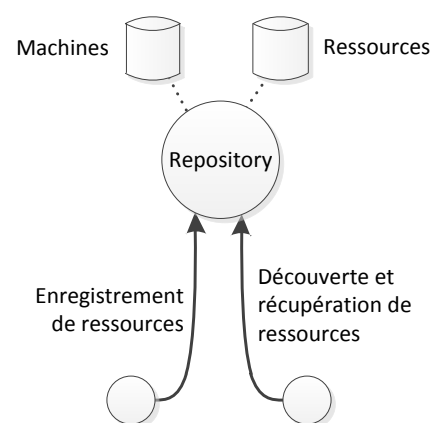


Fig. 42 – Le serveur de ressources

L'infrastructure du middleware se rapproche d'une architecture orientée service. Les services sont exposés sous forme de points d'entrées identifiés de manières uniques (ibiuirl) accessibles aux d'un mécanisme de communication par messages formalisés (nom de service, paramètres, etc.) de type requêtes/réponses. Ces différents points d'entrées sont enregistrés sur le repository qui fait office d'annuaire de services, celui-ci étant lui aussi accessible au travers de cette même mécanique de messages formalisés.

Il n'existe pas, à l'heure actuelle, de mécanisme d'orchestration ou de composition au sein du middleware, bien que ces aspects fassent toujours partie des thèmes de recherche de l'équipe ARLES.

4.3 - La simplification du processus de développement et d'évolution : apports concrets et réflexions futures

Le middleware iBICOOP et l'infrastructure qui l'accompagne met en œuvre une grande partie des concepts que j'ai exposé dans la partie concernant les architectures intergicielles (services, couches et composants distribués). L'objectif de ces techniques et de ces concepts réside dans l'atténuation des difficultés rencontrées dans un processus de développement multiplateforme.

4.3.1 - En quoi iBICOOP permet-il de simplifier le processus de développement et d'évolution ?

Nous avons vu que le middleware iBICOOP apportait deux éléments fondamentaux en réponse aux problèmes d'hétérogénéité que l'on peut rencontrer dans l'univers de la téléphonie :

- **des abstractions** pour les différents aspects d'un téléphone mobile (la communication, le stockage, etc.).
- **une infrastructure** complète pour la gestion de certificats, la médiation, la découverte et l'enregistrement de ressources indépendamment des mécanismes propres aux téléphones.

Les abstractions vont permettre de découpler les aspects génériques et spécifiques des systèmes d'exploitation pour téléphones mobiles. Dans le cadre du stockage, par exemple, on retrouvera toujours la possibilité de manipuler des fichiers et des dossiers. De même, dans le cadre de la sécurité, on pourra décrire des opérations élémentaires propres à tous les systèmes (voir figure ci-dessous).

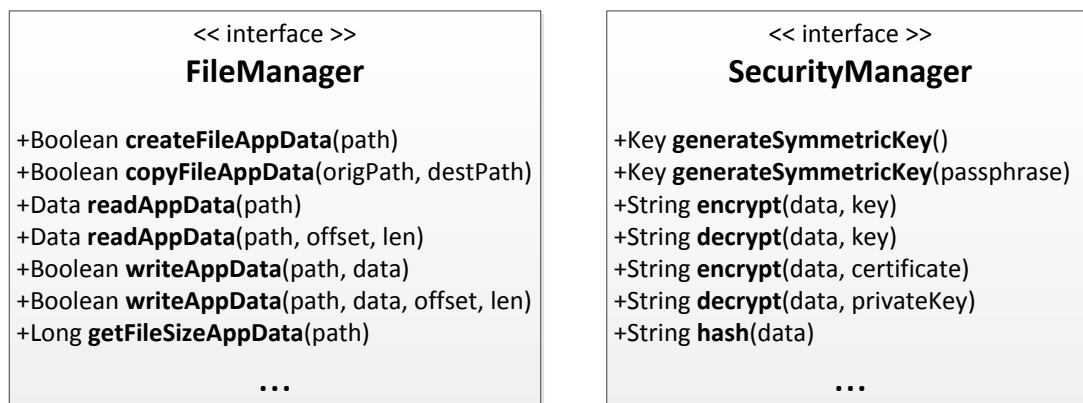


Fig. 43 – Extrait des abstractions pour la manipulation du stockage et les opérations de chiffrement

Bien entendu, tous les téléphones mobiles ne vont pas proposer les mêmes fonctionnalités concrètes et c’est pour cette raison que les abstractions offertes par iBICOOP vont unifier l’ensemble de la gestion des possibilités du téléphone en exploitant ce qui existe et en ajoutant, si nécessaire, ce qui est manquant. On peut voir le middleware comme un système de remise à niveau qui va permettre d’assurer que tout ce qui sera construit par-dessus le middleware fonctionnera de la même façon. C’est en effet lui qui prendra en charge les spécificités des terminaux et marquera la frontière entre ce qui est spécifique et ce qui est générique, tel que nous l’avons déjà mentionné.

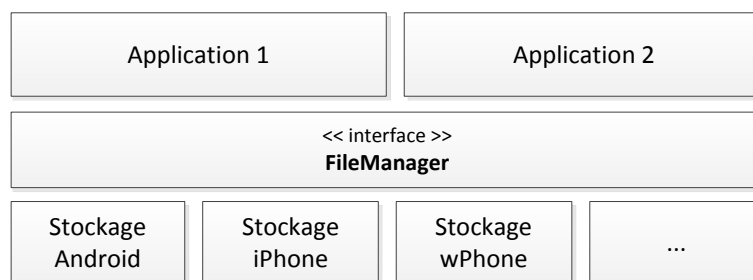


Fig. 44 – Abstraction des spécificités du stockage

Dans un second temps, l’infrastructure du middleware permet d’appliquer le principe de factorisation de façon à permettre aux terminaux d’externaliser les traitements qu’ils ne seraient pas en mesure de réaliser, pour des raisons de performance ou simplement de capacités. Par exemple, nous verrons que le mécanisme de communication du système Windows Phone 7 est extrêmement restrictif et ne permet pas de faire communiquer les terminaux en pair-à-pair. Aussi, le serveur proxy que nous avons décrit précédemment

résout ces problèmes de connectivité en fournissant un médiateur fonctionnant comme un serveur de messagerie.

L'architecture orientée service fournie par le middleware permet une factorisation bien plus fine, permettant potentiellement de distribuer les traitements incompatibles avec une plateforme sur d'autres plateformes, mobiles ou non. C'est exactement ce qui est effectué avec les différents serveurs de l'infrastructure. Le serveur proxy factorise la communication par messages, le serveur repository factorise les mécanismes de découverte et le serveur security factorise les mécanismes d'authentification.

Au final, l'ensemble de l'infrastructure est conçu de manière à généraliser et à factoriser les concepts inhérents à la forte connectivité des terminaux mobiles ainsi qu'au développement d'applications distribuées adaptées à ceux-ci.

L'impact sur le processus de développement est important. Tout d'abord, le code source écrit par-dessus IBICOOP peut être porté facilement car même si le langage change, les mécanismes restent les mêmes. Ainsi, il devient très facile d'écrire un convertisseur de code d'un langage vers un autre puisque seules les spécificités syntaxiques dudit langage devront être prises en compte. Par ailleurs, nous avons déjà pu montrer que cette opération était triviale si les frameworks du langage différaient peu (ce qui est le cas ici, puisque le framework est en grande partie remplacé par le middleware).

Disposer d'un code source proche et transformable facilement permet donc de gérer un processus organisationnel en conséquence. Etant donné que l'infrastructure reste la même et que chaque version des abstractions du middleware possède un comportement rigoureusement identique aux autres versions, les applications bâties au-dessus peuvent bénéficier d'un processus unique en ce qui concerne l'identification des différentes tâches, le découpage et le suivi des anomalies. Au final, c'est l'intégralité de l'aspect multidimensionnel qui est atténué par l'exploitation des abstractions fournies par le middleware.

4.3.2 - Exemple d'exploitation d'iBICOOP : le partage de photo multiplateforme

L'INRIA a organisé, fin juillet, une rencontre entre des équipes de recherche du centre et un groupement d'industriels intéressés par les perspectives de l'informatique future. Dans cette optique, l'équipe a présenté deux applications concrètes bâties sur iBICOOP :

- UPhoto, une application de partage de photo.
- AmbiStream, une application de diffusion multimédia en streaming.

Etant en charge de la version Windows Phone 7 du middleware, j'ai réalisé le développement de l'application UPhoto pour cette plateforme.

Très simple, celle-ci a pour vocation de permettre à une communauté hétérogène d'échanger des fichiers photographiques. Basée sur iBICOOP, on y retrouve certains mécanismes évoqués précédemment :

- les opérations de découvertes et d'échange de fichier (Fig. 45), en pair-à-pair ou au travers du serveur proxy.
- les opérations d'accès au stockage et aux données appartenant à l'utilisateur (Fig. 46), en l'occurrence la galerie d'images.

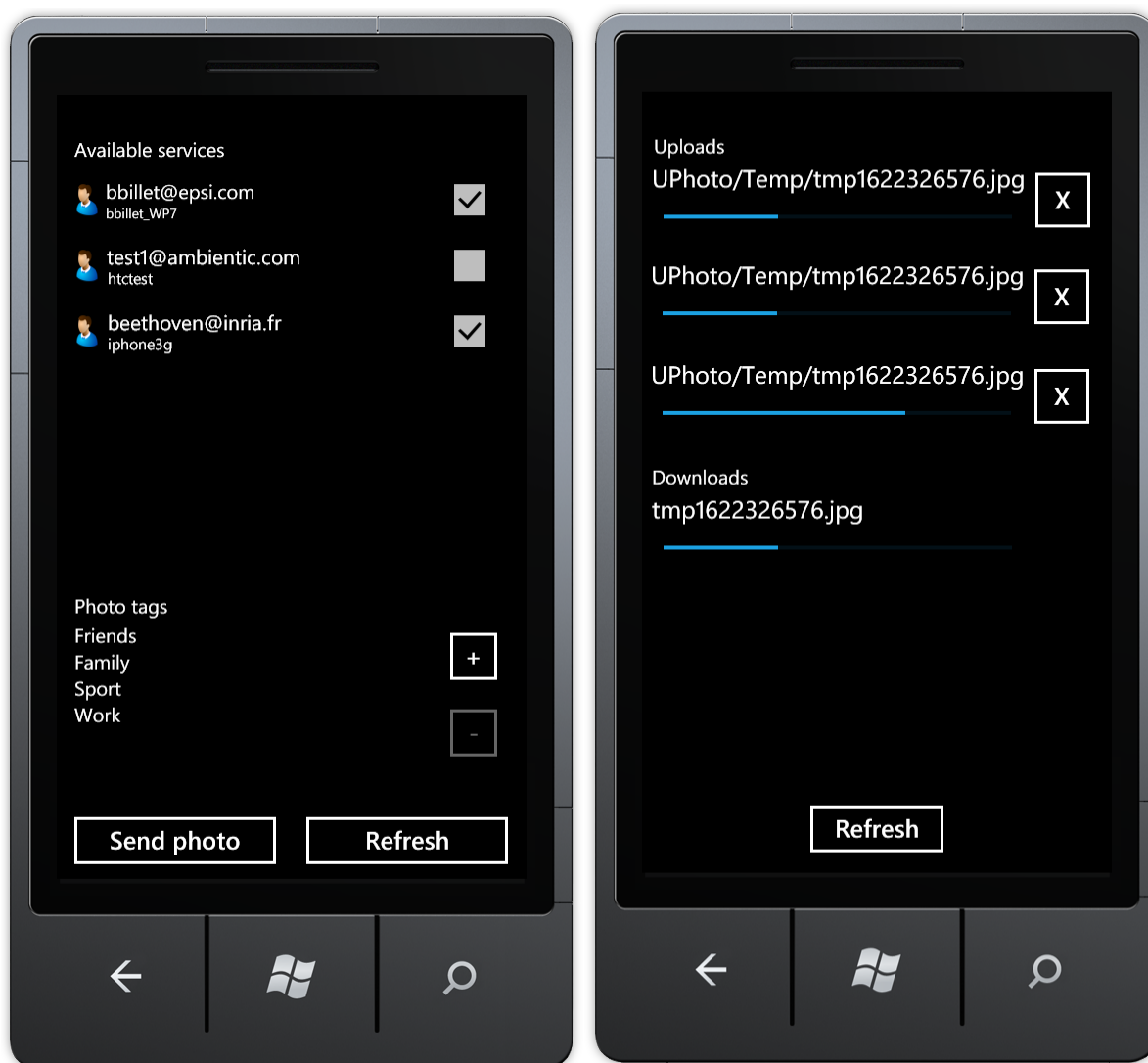


Fig. 45 – Captures de la découverte (à gauche) et de l'échange de fichiers (à droite)



Fig. 46 – Captures de l'accès à la galerie d'images du téléphone

Sans iBICOOP, cette application, bien que simple dans ses principes et ses concepts, nécessiterait beaucoup de travail pour prendre en compte un ensemble conséquent de plateformes mobiles (problèmes de communication, d'accès aux données, de protocoles, etc.). Ici, cette complexité est prise en charge par le middleware qui unifie le développement sur chaque plateforme. Par exemple, le code source pour les versions Java et les versions C# sont quasiment identiques, à quelques subtilités syntaxiques près, comme on peut le voir sur les deux extraits de code ci-dessous.

```
log.debug("Discover receivers for file");

HashMap map = new HashMap();
NamingInterface resolver = NamingManager.getInstance();
IBIURL ibiurl;
try
{
    String photoServicesIbiurl = "ibiurl://*:*/" + APPID + "/" + SERVID + "/*";
```

```
log.debug("Discover receivers for file");

Dictionary<String, ResolvInfo> map = new Dictionary<String, ResolvInfo>();
NamingInterface resolver = NamingManager.getInstance();
IBIURL ibiurl;
try
{
```

```

ibiurl = new IBIURL(photoServicesIbiurl);

Vector photoDestinations = null;
photoDestinations = resolver.solveMultipleDestinations(ibiurl, 3, true);
for (int x = 0; x < photoDestinations.size(); x++)
{
    ResolvInfo info = (ResolvInfo) photoDestinations.elementAt(x);
    IBIURL destibiurl = new IBIURL(destibiurlStr);
    String strkey = destibiurl.getTermid() + " - " + destibiurl.getUserid();

    log.debug(" - dest " + destibiurl);
    map.put(strkey, info);
}

return map;
}
catch (Exception e)
{
    return null;
}

```

```

String photoServicesIbiurl = "ibiurl:/*:*/" + APPID + "/" + SERVID + "/*";
ibiurl = new IBIURL(photoServicesIbiurl);
List<ResolvInfo> photoDestinations = null;
photoDestinations = resolver.solveMultipleDestinations(ibiurl, 3, true);

foreach(ResolvInfo info in photoDestinations)
{
    IBIURL destIbiurl = new IBIURL(info.getIbiurlStr());
    String key = destIbiurl.getTermid() + " - " + destIbiurl.getUserid();

    log.debug(" - dest " + destIbiurl);
    map[key] = info;
}

return map;
}
catch (Exception)
{
    return null;
}

```

Fig. 47 – Une méthode réalisant la découverte des destinataires, en Java (à gauche) et en C# (à droite)

Qu'il s'agisse simplement du code source ou, plus généralement, de l'infrastructure des applications, le processus de développement est unifié par l'utilisation du middleware iBICOOP. La phase de conception devient commune à toutes les plateformes car le comportement du système est identique sur chacune d'entre elles. Les phases de développement et de test sont simplifiées par les mécanismes du middleware, eux aussi identiques sur chaque plateforme. Seules des considérations purement syntaxiques doivent être prises en compte, et celles-ci peuvent souvent être gérées de manière automatique ou semi-automatique (la conversion du code Java vers le code C#, ou l'inverse, est triviale pour le code source exposé en Fig. 47).

4.3.3 - iBICOOP, lui aussi victime des difficultés inhérentes au processus de développement des environnements multiplateformes

Le middleware iBICOOP prenant en charge la complexité liée au multiplateforme, il est lui-même sujet aux difficultés propres aux domaines multiplateformes. En effet, puisqu'il fournit une frontière entre le spécifique et le générique il appartient lui-même à la partie spécifique et doit donc être décliné, en tout ou partie, sur plusieurs plateformes mobiles différentes. Dans cette optique, l'équipe en charge du projet a mis en place différentes choses pour simplifier le développement et l'évolution du middleware.

Tout d'abord, parmi la liste des plateformes ciblées par le middleware, on en distingue quatre à base de Java :

- Les téléphones JME
- Les téléphones BlackBerry

- Les téléphones Android
- Les terminaux non mobiles équipés d'un environnement d'exécution java (ordinateur fixe, ordinateur portable, serveur, etc.)

Ces différentes plateformes partagent des frameworks différents mais présentent un tronc commun : la technologie Java. C'est pour cela qu'une partie des mécanismes a été factorisée sous le nom d'iBICOOP Core. Celle-ci définit les interfaces pour les différents composants ainsi que les traitements abstraits permettant de manipuler ceux-ci. La réflexion est, par ailleurs, utilisée pour garantir l'utilisation transparente de composants concrets qui seront ajoutés plus tard par les extensions de ce paquetage iBICOOP Core (voir Fig. 48).

On retrouve ici la notion de dénominateur commun, c'est-à-dire la plus grande entité générique pour chaque plateforme (celle-ci étant principalement restreinte par les faibles capacités des téléphones JME). C'est dans ce dénominateur commun que l'on retrouvera la couche d'abstraction évoquée précédemment et c'est elle qui sera manipulée par les applications bâties au-dessus.

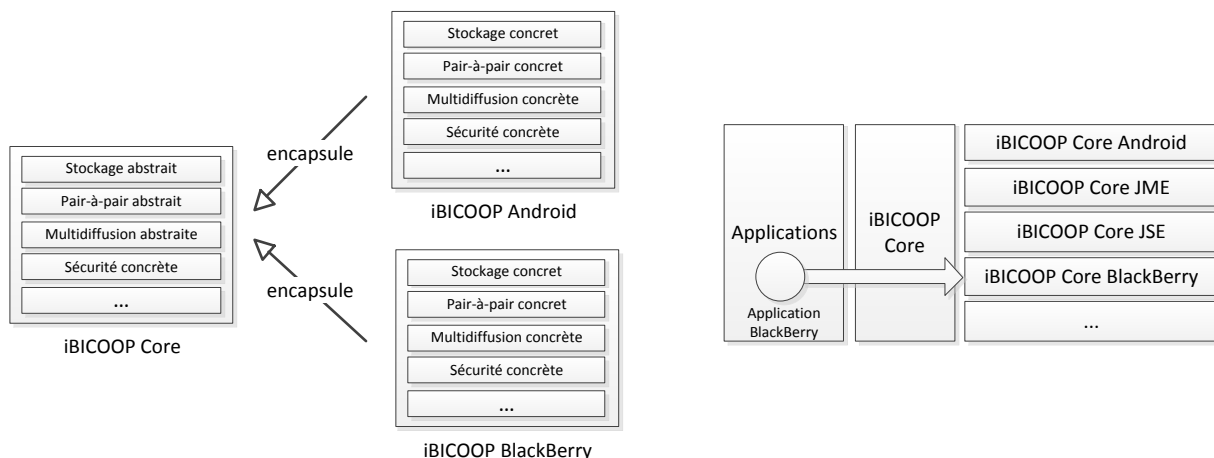


Fig. 48 – L'entité commune iBICOOP Core et ses extensions

Cette structure n'est toutefois viable qu'au sein d'un même langage et ne touche pas les plateformes à base de C# (Windows Phone) et à base d'Objective C (iPhone). De fait, parallèlement à cette architecture, l'organisation du développement a été faite de manière à ce que chaque membre de l'équipe prenne en charge une plateforme spécifique. Ainsi, les

versions iPhone et Windows Phone sont maintenues par des personnes spécialisées dans ces domaines respectifs.

Au long de ma mission, j'ai été en charge des versions Windows Phone du middleware et des applications conçues pour l'utiliser. Comme le middleware existait déjà sur les autres plateformes lorsque je suis arrivé, j'ai été amené à suivre plusieurs pistes pour simplifier et accélérer le développement du middleware, au vu des spécificités des terminaux Windows Phone.

Dans un premier temps, j'ai abordé une approche par transformation de code. Mais les simples substitutions, qui me permettaient effectivement de transformer à la fois la syntaxe du langage et une partie des éléments triviaux du framework, se sont révélées inefficaces pour gérer des aspects très particuliers tels que la sécurité. En effet, le Windows Phone fournit très peu d'interfaces de programmation pour les opérations de cryptographie (chiffrement, signature, hashage, génération de nombres pseudos-aléatoires sûrs, etc.) et il a été nécessaire de se pencher sur des bibliothèques tierces. Celles-ci, souvent déclinées en C# natif ou obsolète, ont dû être transformées pour pouvoir fonctionner avec le framework réduit du Windows Phone, impliquant la encore des adaptations non évidentes pour un simple transformateur substitutif.

Par la suite, l'approche dirigée par les modèles m'est apparue comme une solution nettement plus efficace, bien que plus complexe. Pour ce faire, j'ai utilisé un projet développé par la société Setinnov pour laquelle j'ai effectué de précédentes missions de recherche et développement. Ce projet, baptisé Nestene, visait à produire un métalangage simple, modulable (orienté objet et orienté aspect) et suffisamment abstrait pour que chacune des fonctionnalités de son framework soit transformable dans n'importe quel langage à partir du moment où cette implémentation est connue.

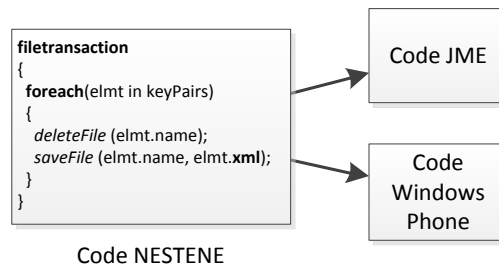


Fig. 49 – Le principe de la conversion Nestene

Le framework, décomposé en implémentations atomiques et en implémentations composées (combinaisons d'implémentations atomiques), nécessite donc d'être alimenté une première fois pour que celui-ci prenne connaissance desdites implémentations atomiques, c'est-à-dire de l'implémentation concrète des fonctionnalités basiques pour les différents langages ciblés. Ainsi, une fois toutes ces implémentations basiques fournies par les développeurs, les fonctionnalités de plus haut niveau coulent forcément de source, s'appuyant sur les implémentations atomiques, comme exposé sur la figure ci-après (Fig. 50). Ainsi, le processus de production logicielle s'effectue en deux temps :

- **capitalisation** des traitements basiques pour les langages voulus. Ce travail doit être effectué par les développeurs et peut, bien entendu, être réutilisé.
- **génération** du code source final par le générateur Nestene, en utilisant les traitements basiques précédemment saisis.

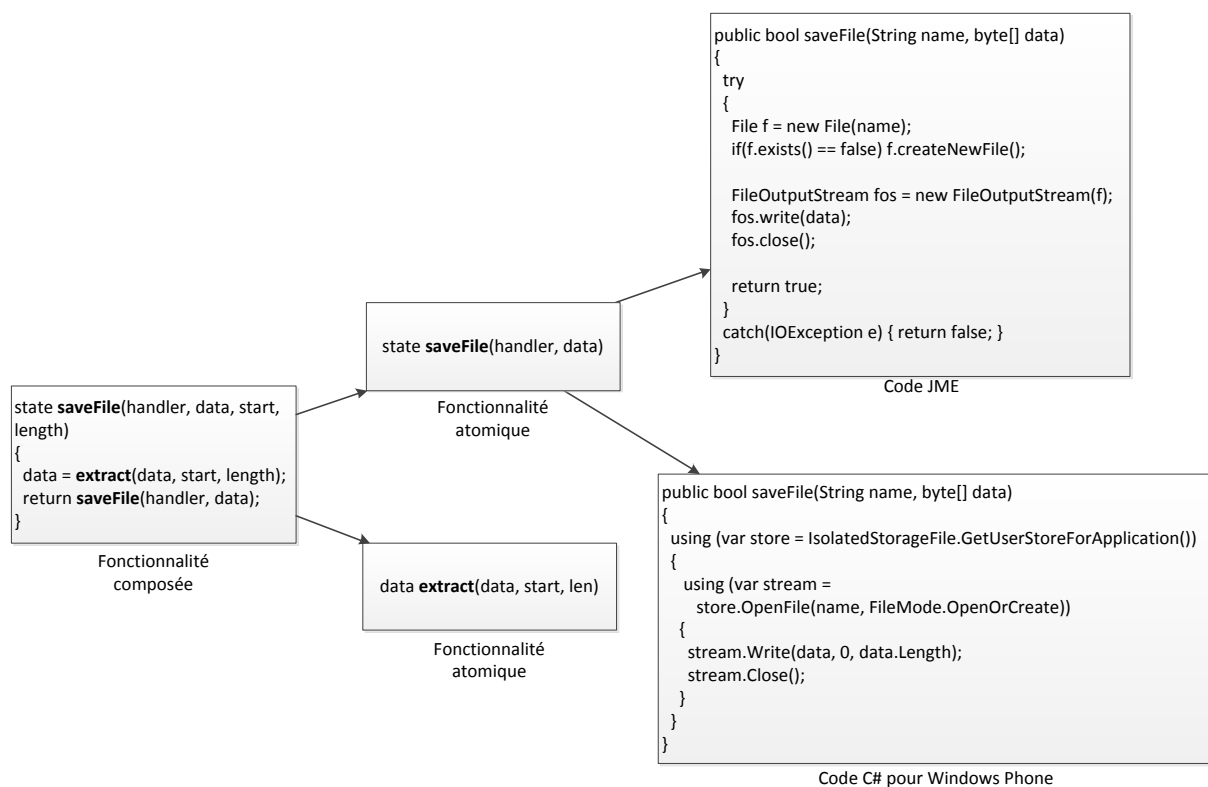


Fig. 50 – Exemple de fonctionnalités composées et fonctionnalités atomiques

Les apports concrets d'une telle approche sont multiples. Tout d'abord, le mécanisme permet d'effectuer une capitalisation intelligente des implémentations simples. Nestene agit comme un catalogue d'implémentation et les combine de lui-même pour former d'autres implémentations plus complexes selon un schéma défini par les composants du métaframework pré-établi.

Ensuite, une fois le catalogue constitué, l'utilisation du métalangage se révèle suffisamment simple pour pouvoir décrire les fonctionnalités basiques d'une application ainsi que les interfaces pouvant manipuler ces fonctionnalités. Nestene n'étant pas, à l'heure actuelle, un projet très complet (simple prototype), il n'est pas en mesure de fournir des abstractions pour des éléments aussi avancés que la gestion de composants distribués ou l'appel de services web, cependant l'utilisation des abstractions permet de générer assez efficacement l'ensemble des éléments concrets de base. Je pense, par exemple, à la manipulation du stockage, au chiffrement ou à une gestion minimaliste du réseau.

Enfin, bien entendu, le code écrit au travers du métalangage est un code abstrait, un modèle décrit sous forme textuelle. De fait, il est extensible et peut-être transformé dans d'autres langages au travers du processus strict et automatisé imposé par Nestene. De plus, il est

facile de répercuter une modification du modèle sur l'ensemble des codes sources concrets déjà généré, ceci étant le principal avantage de l'ingénierie pilotée par les modèles.

4.3.4 - Les limitations et les perspectives envisageables

Fournir une couche abstraite de manipulation des fonctionnalités des téléphones mobiles implique que ceux-ci disposent d'éléments communs pour permettre cette abstraction. Aussi, certains éléments sont difficiles à factoriser pour plusieurs raisons. Leur complexité ou leur spécificité peut-être telle que l'on ne retrouvera pas ou peu d'équivalents possibles d'une plateforme à une autre, ce qui entraînerait une certaine pauvreté ou une forte disparité dans l'abstraction résultante. C'est le cas, par exemple, des mécanismes de connexion entre fils d'exécution au sein d'un programme.

Il est possible que l'élément à factoriser couvre un champ extrêmement large, ce qui demanderait beaucoup de travail pour obtenir une abstraction couvrant la totalité du champ. De fait, il est préférable de laisser aux applications bâties au-dessus du middleware la responsabilité de la gestion de cet aspect. L'interface graphique, souvent constituée de multiples composants, de langages descriptifs (description des interfaces en XML, par exemple) et de mécanismes spécifiques (événementiel, échange de message, etc.) est difficile à factoriser et nécessite de construire des abstractions pour chaque élément, en veillant à ne pas se montrer trop restrictif pour éviter que cela conduise à des interfaces graphiques pauvres.

En ce qui concerne iBICOOP les problématiques liées aux interfaces graphiques ne sont pas prises en charge pour ces raisons-ci. En effet, les interfaces graphiques ne sont pas forcément traitées de la même façon sur les terminaux :

- Sur Android et Windows Phone elles peuvent être décrites en XML.
- Sur iPhone, Android, Windows Phone, BlackBerry, JME et JSE, elles peuvent être construites par programmation, manuellement ou au travers d'un éditeur graphique.

Même si l'approche par programmation peut faire office de dénominateur commun, la richesse des composants et leur nombre élevé complexifient le processus consistant à extraire de toutes ces plateformes un moyen de construire des interfaces graphiques abstraites, par programmation ou au travers de descriptions. Il s'agit cependant d'une des

perspectives futures en ce qui concerne le middleware, un système existant déjà pour rationaliser le développement multiplateforme mobile sans toutefois être aussi abouti qu'iBICOOP en ce qui concerne le développement des applications distribuées. Ce système s'appelle Phone Gap et propose de fabriquer des interfaces graphiques en utilisant les standards de fabrication d'interface graphique pour les sites internet, généralement bien connus et maîtrisés par les développeurs et les infographistes web. Cela se traduit par la description des interfaces graphiques en HTML/CSS et le scripting des traitements en JavaScript, la conversion s'effectuant ensuite au travers du compilateur Phone Gap tel que le montre la figure ci-dessous.

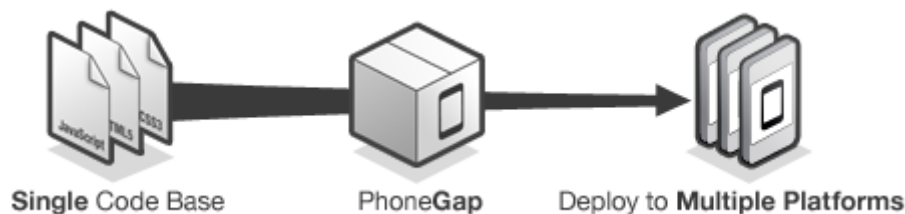


Fig. 51 – Le processus Phone Gap

L'API en Javascript permet de gérer l'ensemble des fonctionnalités graphiques du téléphone telles que l'échange de message entre composants, la capture d'évènements, la caméra, les notifications ou encore les différentes galeries de médias (photographies, musiques, vidéos, etc.). Au final, l'utilisation de technologies ouvertes et connues du plus grand nombre permet d'assurer une excellente portabilité à partir du moment où le compilateur est décliné pour la plateforme ciblée.

5 - Conclusion

Les systèmes informatiques sont devenus des éléments anodins de nos paysages respectifs, qu'il s'agisse de notre vie de tous les jours avec les systèmes embarqués (téléphones, voitures, électroménager, etc.) et les ordinateurs personnels, ou de notre vie en entreprise avec les grandes infrastructures de gestion ou l'utilisation des applications mobiles d'entreprise.

Toutefois, si l'on s'y penche de plus près, la nature de ces systèmes varie, qu'il s'agisse de leur matériel ou des logiciels qui y sont exécutés et l'on prend conscience du très grand nombre de plateformes disparates qui nous entourent et qui, peu à peu, doivent s'efforcer de communiquer entre elles pour toutes sortes de raison allant de la mobilité croissante des individus jusqu'à la nécessité de disposer de systèmes toujours plus intelligents (et de fait, plus communicants).

Tous ces éléments matériels et logiciels qui interagissent entre eux, ou qui devront un jour pouvoir le faire, doivent composer avec des protocoles de communication, des environnements d'exécution, des frameworks et des contraintes diverses et variées (temps réel, sécurité, vélocité, résilience, etc.) : tous ces éléments que l'on peut considérer comme une plateforme ou comme appartenant à une plateforme.

De nombreux problèmes découlent de cette hétérogénéité et ceux-ci ne peuvent pas être ignorés par les entreprises. L'aspect multiplateforme de notre environnement est une réalité dont il faut avoir conscience à la fois pour le grand public, par respect des consommateurs, mais aussi pour l'informatique d'entreprise qui, plus encore, ne peut se permettre d'exclure des clients potentiels au profit des concurrents.

C'est pour cela que les applications qui nous entourent doivent tendre, et tendent de plus en plus, vers une diffusion au travers d'une multitude de systèmes. On connaît aujourd'hui des technologies de développement multiplateforme (Java) ou potentiellement multiplateforme (.NET), des infrastructures multiplateformes (les bus d'entreprise, par exemple), des frameworks et des bibliothèques multiplateformes (interfaces graphiques

multiplateformes, moteurs 3D multiplateformes, etc.) et leur nombre ne fait que croître tout en permettant aux applications qui les exploitent de s'affranchir des notions complexes qui régissent les systèmes sur lesquels et avec lesquels elles doivent fonctionner.

Pourtant, faire du multiplateforme sans utiliser un quelconque dispositif pour le simplifier peut avoir un coût important. Un coût humain, d'une part, car les compétences nécessaires pour l'utilisation et la manipulation de chaque plateforme doivent exister au sein des équipes et être disponibles pour toute la durée du processus de développement. Un coût en temps d'autre part, car la durée du développement est accrue par le nombre de plateformes à cibler (on pourrait considérer, pour avoir une estimation grossière, que la durée d'un projet multiplateforme est équivalente à la durée du projet pour la plateforme la plus complexe et la plus gourmande en temps, multiplié par le nombre de plateformes), ce même si les projets sont parallélisés car il est nécessaire d'assurer une cohérence dans les différents processus de développement.

Ces coûts sont, de plus, à mettre en relation avec les risques du multiplateforme. Une approche chaotique dans un projet multiplateforme démultiplie les risques par rapport à une approche chaotique dans un projet monoplatforme (on peut considérer ici aussi que les risques sont équivalents aux risques de la plateforme la plus risquée, multipliés par le nombre de plateformes). En effet, les risques sont impactés par l'aspect multidimensionnel du multiplateforme qui peut aboutir rapidement à des écarts significatifs entre les versions, voire des fonctionnements fondamentalement différents entre deux versions censées être identiques.

Au vu des coûts potentiels et des risques, il est légitime de se poser la question de l'intérêt concret du développement multiplateforme. La réponse est simple : l'intérêt est relativement faible pour les clients finaux mais fort d'un point de vue commercial (en plus d'un intérêt technique certain). A l'heure actuelle, le multiplateforme doit être vu comme un besoin incompressible du développement logiciel ; faire du multiplateforme a un coût certain et n'apporte rien de plus aux utilisateurs d'une plateforme en particulier mais ne pas se préoccuper de cette dimension peut avoir un impact fort en termes de pénétration de marché, de consolidation d'un marché existant ou simplement d'exhaustivité quant à la couverture de ce marché. Par exemple, Microsoft s'efforce de couvrir le marché des

ordinateurs Apple en distribuant une version de sa suite bureautique Microsoft Office pour le système d'exploitation Mac OS. Cependant, le résultat est très discutable car les deux versions n'interprètent pas un même document de la même façon, générant une présentation différente de son contenu. C'est en soi un bon exemple du décalage fonctionnel qui peut survenir dans un projet multiplateforme chaotique.

La téléphonie mobile, et plus particulièrement la téléphonie mobile intelligente (smartphone), est un exemple probant d'univers multiplateforme pour lequel il n'est pas possible de se concentrer sur une seule plateforme en particulier. Cet univers remplit, en effet, les conditions essentielles pour que la non prise en compte de l'aspect multiplateforme soit préjudiciable à une entreprise. Concrètement, il s'agit d'un milieu basé sur des technologies de pointe assez mal structurées et peu standardisées où le nombre de gros acteurs est important (Google, Apple, Microsoft, Nokia, etc.) et qui connaît une très forte croissance depuis quelques années. La forte concurrence et l'ébullition technologique qui en résultent obligent les différents intervenants à adopter des visions à court voire à très court terme étant donné l'incertitude relative à la survie de tel ou tel acteur du domaine. Au final le nombre d'acteurs fragmente le paysage global et, même si certains se démarquent plus ou moins, les parts de marché de chacun d'eux dépassent rarement l'ordre des 35 %.

De fait, pour une entreprise qui développe des applications mobiles, il n'est pas envisageable de se restreindre à une part si faible alors qu'il s'agit d'un marché aussi lucratif. De même, il n'est pas question de se focaliser sur tel ou tel acteur car il est difficile de prédire si celui-ci sera encore dans la course demain. On citera, pour l'exemple, l'arrêt en 2011 du développement du système d'exploitation WebOS (HP, 2011) alors maintenu par la société HP. Malheureusement, un système qui disparaît aura nécessité un investissement pour pouvoir être pris en compte dans le processus de développement d'applications. C'est pour cette raison que l'aspect multiplateforme ne doit pas être traité de manière naïve, c'est-à-dire en affectant des processus de développement totalement distincts pour la réalisation d'une application multiplateforme. Au contraire, c'est un processus qui doit être maîtrisé, au travers de méthodes de travail spécifiques et de technologies adaptées à la prise

en compte des aspects complexes du multiplateforme (multidimensionnalité et hétérogénéité, pour ne citer que les plus critiques).

C'est de ce constat que provient ce fort besoin en technologies et en architectures capables d'absorber la complexité du multiplateforme. En effet, les coûts en ressources humaines et temporelles peuvent être diminués si l'on parvient à restreindre l'ensemble des plateformes à un ensemble plus petit ou à un singleton, dans le meilleur des cas. Pouvoir manipuler des projets multiplateformes comme des projets monoplatformes est l'enjeu primordial de ma réflexion à ce sujet car, s'il est possible de restreindre le champ du multiplateforme pour le ramener en terrain connu et maîtrisé, alors les aspects complexes précédemment évoqués s'atténueront naturellement, réduisant les risques et les coûts qui y sont associés.

Bien entendu, restreindre un univers multiplateforme à un univers monoplatforme est une impossibilité physique si les plateformes en question n'ont pas été initialement prévues pour cela à l'origine, c'est-à-dire si celles-ci n'ont pas été sujettes à un important effort de standardisation et d'unification. Aussi, le mieux que l'on puisse espérer à l'heure actuelle c'est une simplification de la manipulation des univers multiplateformes ou d'une partie de ceux-ci. C'est encore plus vrai pour les domaines où le multiplateforme est une nécessité (la téléphonie mobile, encore une fois).

Aussi, dans ce mémoire je me suis attaché à mener une réflexion sur les techniques permettant de restreindre l'univers multiplateforme et de simplifier le processus de développement des applications conçues pour ces univers, ce pour répondre à la question « en quoi les architectures logicielles et intergicielles peuvent-elles simplifier le processus de développement et d'évolution des applications multiplateformes ? ». Pour ce faire, je me suis basé sur les travaux que j'ai effectués à l'INRIA sur le middleware iBICOOP, une infrastructure logicielle destinée à rationaliser le développement d'applications distribuées multiplateformes pour les téléphones mobiles.

Avant toute chose il faut bien avoir conscience d'un fait essentiel : l'exhaustivité absolue n'existe pas. Quels que soient les apports des techniques du développement, de l'organisation et de la conception multiplateforme, il faut garder à l'esprit qu'il n'est pas possible de développer des applications sur l'ensemble des plateformes connues. Il est nécessaire, à un moment donné, de restreindre cet ensemble pour la simple raison qu'au plus une abstraction couvre d'éléments spécifiques au plus celle-ci risque de s'appauvrir vis-à-vis de ce qu'elle décrit. On peut trouver une excellente analogie dans le langage naturel ; lorsque l'on emploie le terme « animal » on décrit, au travers de ce concept, l'ensemble des animaux existants, ce qui fait disparaître les notions propres à toutes les entités de la taxonomie du genre animal (les oiseaux, les mammifères, etc.) ainsi que les notions propres à chaque individu physique appartenant à ces entités.

Concrètement, il serait difficile de décrire des représentations abstraites exploitables pour un ensemble de plateformes composé de systèmes embarqués et de supercalculateurs, car ceux-ci ont des spécificités tellement distinctes et opposées que le seul moyen de les abstraire serait, en caricaturant un peu :

- soit de les ignorer, ce qui aboutirait à produire une abstraction vide.
- soit de les intégrer entièrement dans l'abstraction et par conséquent de ne rien abstraire du tout.

Néanmoins, cette restriction des plateformes ciblées n'est pas forcément figée, et pourra accueillir de nouvelles plateformes dans le futur. Celles-ci se plieront alors aux abstractions déjà obtenues, pouvant même conduire à une extension ou un raffinement de celles-ci.

Une fois cette restriction effectuée, il existe au final trois grandes solutions intellectuelles pour simplifier le processus de développement et d'évolution des applications sur un ensemble donné de plateformes.

La première consiste à cloisonner, c'est-à-dire à considérer les éléments comme des entités distinctes pour chaque plateforme prise en compte. Ainsi ceux-ci se déroulent indépendamment, avec leurs propres ressources et leurs propres objectifs, mais toujours vers une direction unique.

La seconde consiste à transformer ce que l'on connaît (que ce soit une connaissance d'ordre générale telle qu'un modèle ou une connaissance déjà ancrée dans une plateforme

spécifique) vers l'ensemble des plateformes prises en compte. Ceci sous-entend des approches telles que la conversion (au travers d'un processus automatisé ou non) ou l'adaptation de ces connaissances.

Enfin, la dernière porte sur l'action de factoriser, c'est-à-dire séparer ce que l'on sait être générique de ce que l'on sait être spécifique. Cela permet de poser d'une part des abstractions exploitables sur l'ensemble des plateformes prises en compte (et sur celles qui pourraient être prises en compte ultérieurement) et d'autre part des entités concrètes et spécifiques à chacune de ces plateformes.

Ces trois réponses intellectuelles aux difficultés de gérer un processus de développement multiplateforme se retrouvent plus ou moins dans les trois niveaux de réflexion que j'ai identifiés lorsqu'il m'a fallu chercher à répondre de manière concrète à ma problématique. L'organisation, la technique et la conception sont en effet les trois niveaux qui représentent les différents points de vue des acteurs d'un projet logiciel : chefs de projets, développeurs et architectes. Chacune de ces familles d'acteurs entrevoit le multiplateforme selon son cœur de métier et apporte des solutions propres à son niveau d'action pour répondre au mieux au problème initial. Il n'existe pas, et il n'existera jamais de vision unifiée du multiplateforme, car c'est un problème vaste qui touche des cœurs de métier aux préoccupations bien différentes. Ceux-ci ne peuvent que combiner leurs approches, pas les imposer comme étant la meilleure réponse possible.

De fait, il n'existe pas, à proprement parler, de solution miracle pour parvenir à mener à bien un projet multiplateforme. Chacune des techniques, dans les trois niveaux de réflexion n'apporte qu'une réponse incomplète à la problématique initiale pour la simple raison que l'aspect multiplateforme ne peut pas être restreint à un niveau de réflexion bien particulier. Chaque niveau propose des solutions capables d'atténuer certains aspects spécifiques du multiplateforme.

Tout d'abord, l'approche organisationnelle apporte des solutions lourdes et coûteuses en termes de ressources humaines et de méthodes de travail, mais permet de cloisonner les

projets et d'ignorer l'aspect multidimensionnel du multiplateforme tout en favorisant la parallélisation de ces différents processus de travail.

Ensuite, l'approche technique apporte des solutions complexes, intrusives et contraignantes, mais permet d'accroître la généricité (bibliothèques portables, préprocessing, compilation croisée) au plus bas niveau des applications, sans complexifier les architectures et sans nécessiter l'élaboration d'abstractions complexes. De plus, son aspect fortement local permet d'agir avec précision sur les spécificités de chaque plateforme.

Enfin, l'approche conceptuelle, thème central de ce mémoire, unifie les deux autres approches, s'intercalant entre la technique du plus bas niveau (en fournissant des abstractions et des modèles, en prenant en compte les aspects techniques, etc.) et les aspects organisationnels de la gestion de projet (en permettant un meilleur découpage, en simplifiant l'évolutivité de l'ensemble, en assurant la réutilisabilité des développements antérieurs, etc.). Les architectures logicielles fournissent le ciment qui va permettre de lier les considérations techniques aux considérations organisationnelles, en découplant les différents constituants des systèmes, en isolant le générique du spécifique et en fournissant les outils nécessaires à la manipulation des plateformes. De même, les architectures intergielles vont fournir les factorisations nécessaires pour que chaque plateforme puisse exploiter des infrastructures identiques et, de fait, délestent certains traitements à ces entités autonomes. Pour résumer, les approches conceptuelles, en plus de servir d'intermédiaire entre les approches technique et organisationnelle, jouent un rôle multiple, allant de la simple définition de la structure d'une application (architecture logicielle, composants, aspects, etc.) jusqu'aux enchaînements complexes entre les applications (virtualisation, systèmes distribués, etc.). Elles permettent ainsi de simplifier le processus de développement en factorisant des morceaux du système et en transformant du générique en spécifique lorsque c'est nécessaire (génération de code à partir de modèles, synthèse d'adaptateurs à la volée, etc.). Cela a pour effet, d'une part, de restreindre l'ensemble des plateformes pour toutes les applications bâties au-dessus des parties génériques et, d'autre part, d'adapter les applications à leur environnement lorsqu'il n'est pas possible, ou très difficile, d'extraire des abstractions valables.

Au final, ce n'est pas une approche unique qu'il suffit de privilégier, mais les trois approches simultanément car le multiplateforme est un problème complet, présent à tous les niveaux de réflexion possible du monde informatique. Chacun de ces niveaux apporte des éléments de réponses pouvant être combinés pour parvenir aux résultats finaux de la manière la plus efficace possible. De même, au sein de chaque approche, les différentes techniques sont complémentaires, permettant de résoudre telle ou telle partie du problème. Il est, par exemple, tout à fait possible d'envisager une architecture orientée service construite au moyen d'un processus d'ingénierie dirigée par les modèles et déployée sur une infrastructure virtualisée.

C'est cette façon de faire qui fut suivie pour le développement du middleware iBICOOP : une combinaison de cloisonnements organisationnels, de transformations techniques et de factorisations conceptuelles au niveau logiciel et au niveau intergiciel.

Pour conclure, je dirais que le multiplateforme est encore une notion incertaine. L'histoire a montré que les environnements multiplateformes finissent toujours, tôt ou tard, par se rationaliser et par se standardiser suffisamment pour pouvoir communiquer et évoluer simplement. Malgré tout, les difficultés subsistent, d'une manière ou d'une autre, et encore plus sur les marchés porteurs d'innovations technologiques.

Il est possible que, demain, devant la forte croissance des petits périphériques et l'intégration intelligente de ceux-ci au sein de notre environnement, le développement multiplateforme deviennent une priorité, celui-ci s'établissant cette fois sur de nouvelles technologies directement conçues pour prendre en charge ces aspects.

6 - Bibliographie

Citations intext

Fr. Gov., 2004. *Loi n°2004-575*. Texte de loi.

Gartner Group, 2008. *Forecast: PC Installed Base, Worldwide, 2004-2012*. Rapport technique.

Gartner, 2011. *Smartphone Sales Grew 74 Percent*. [Online] Available at: <http://www.gartner.com/it/page.jsp?id=1764714> [Accessed 2 juillet 2011].

Hammoudi, S., 2010. *Contribution à l'étude et à l'application de l'ingénierie dirigée par les modèles*. Habilitation à diriger les recherches.

HP, 2011. *HP Makes Other Announcements*. [Online] Available at: <http://www.hp.com/hpinfo/newsroom/press/2011/110818b.html> [Accessed 2 Septembre 2011].

IDC, 2011. *Worldwide Smartphone Market to Grow by Nearly 50% in 2011*. [Online] Available at: <http://www.idc.com/getdoc.jsp?containerId=prUS22762811> [Accessed 19 Juin 2011].

Issarny, V., Bennaceur, A. & Bromberg, Y.-D., 2011. *Middleware-layer Connector Synthesis: Beyond State of the Art in Middleware Interoperability*.

Nielsen, 2011. *In U.S. Smartphone Market, Android is Top Operating System, Apple is Top Manufacturer*. [Online] Available at: http://blog.nielsen.com/nielsenwire/online_mobile/in-u-s-smartphone-market-android-is-top-operating-system-apple-is-top-manufacturer/ [Accessed 11 juin 2011].

OASIS, 2006. *Reference Model for Service Oriented Architecture 1.0*. Rapport technique.

OMG, 2010. *Meta Object Facility (MOF) Core Specification 2.4*. Rapport technique.

OMG, 2011. *Unified Modeling Language Infrastructure 2.4*. Rapport technique.

OMG, 2011. *Unified Modeling Language Superstructure 2.4*. Rapport technique.

OMG, 2011. *Unified Modeling Language Superstructure 2.4*. Rapport technique.

Tanenbaum, A.S., 2008. *Systèmes d'exploitation*. 3rd ed. Pearson Education.

Références

Remarque : il n'existe pas, à proprement parler, d'ouvrage consacré au développement multiplateforme. En effet, le développement multiplateforme sous-entend plusieurs réponses possibles, qui peuvent être combinées ou non, et qui se basent sur des concepts

issus d'autres domaines informatiques. C'est pour cela que la liste d'ouvrages que je présente ici est principalement constituée de documents scientifiques et techniques (articles, thèses, rapports et spécifications).

Référence bibliographique : M.D. Mc Ilroy, 1968. Mass Produced Software Components. Dans Software Engineering. Garmisch, Germany.

Description : Compte-rendu de conférence traitant de l'industrialisation du développement logiciel tel qu'il était envisagé dans les années 70.

Référence bibliographique : Erwan Breton, 2002. *Contribution à la représentation de processus par des techniques de méta-modélisation*. Nantes: Université de Nantes.

Description : Thèse concernant les méthodes MDE (Model Driven Engineering) appliquées à la modélisation d'un processus. Décrit l'ensemble des technologies existantes pour représenter un processus et propose plusieurs réflexions autour de ceux-ci (évolution, couplage avec d'autres métamodèles, dynamisme).

Référence bibliographique : Bertrand Meyer, 1992. Applying « Design by Contract ».

Description : Introduction à la programmation par contrat, par son inventeur Bertrand Meyer. Les concepts clés et les mécanismes associés à cette méthode semi-formelle sont abordés (contrat, assertion, invariant, postcondition, précondition) ainsi que son application dans des contextes variés (programmation objet, système concurrentiel, etc.).

Référence bibliographique : Fabrice Legond-Aubry, 2005. *Un modèle d'assemblage de composants par Contrat et Programmation Orientée Aspect*. Conservatoire national des arts et métiers.

Description : Thèse traitant des apports de la programmation orientée aspect dans la résolution des problèmes techniques inhérents aux technologies appliquant le paradigme de programmation orientée composants (complexité de déploiement, difficulté d'assemblage et d'extension, etc.)

Référence bibliographique : Gregor Kiczales et al., 1997. Aspect-Oriented Programming. Dans European Conference on Object-Oriented Programming (ECOOP). Finland, p. 25.

Description : Document sur la programmation orientée aspect et ses concepts (tisseur d'aspect, transversalité, greffon, transversalité, etc.), tels qu'ils ont été formulés au Xerox Palo Alto Research Center.

Référence bibliographique : Lionel Seinturier, 2005. *Réflexivité, aspects et composants pour l'ingénierie des intergiciels et des applications réparties*. Paris: Université Pierre et Marie Curie, Paris VI.

Description : Thèse d'habilitation (HDR) concernant la programmation distribuée. Le document aborde des ensembles de concepts théoriques concernant l'introspection, les services distribués, les systèmes à base de composants et la programmation par aspects. Au-delà de ça, le document traite des technologies associées à ces concepts (CORBA, JAC, Fractal, etc.)

Référence bibliographique : Raphaël Marvie & Marie-Claude Pellegrini, 2002. Modèles de composants, un état de l'art.

Description : Document traitant de l'état actuel des approches par composants. Décrit un ensemble de définitions théoriques puis présente quelques modèles académiques et industriels de programmation par composant.

Référence bibliographique : Sylvain Sauvage, 2001. Des patterns orientés Systèmes Multi-agents.

Description : Article traitant de deux patterns spécifiques applicables aux systèmes multiagents : Marques et Influences

Référence bibliographique : Organization for the Advancement of Structured Information Standards (OASIS), 2006. Reference Model for Service Oriented Architecture.

Description : Référence sur l'architecture orientée service. Décrit les concepts théoriques clés (service, contrat, politique, etc.) de l'architecture ainsi que ses avantages et ses inconvénients.

Référence bibliographique : Franco Cicirelli, Angelo Furfaro & Libero Nigro, 2010. A service-based architecture for dynamically reconfigurable workflows. *Journal of Systems and Software*, 83, p.1148-1164.

Description : Article traitant des réseaux de Pétri orientés service pour la modélisation de processus métiers, au travers d'une technologie appelée PN-Engine.

Référence bibliographique : Denivaldo Cicero Pavão Lopes, 2011. *Étude et applications de l'approche MDA pour des plates-formes de Services Web*. Nantes: Université de Nantes.

Description : Thèse décrivant une application de l'approche MDA (Model Driven Architecture) pour la fabrication de services web.

Référence bibliographique : Nicolas Pessemier, 2007. *Unification des approches par aspects et à composants*. Lille: Université Lille 1.

Description : Thèse traitant du rapprochement des méthodes à base de composants et des méthodes à base d'aspect.

Référence bibliographique :

Object Management Group (OMG), OMG Formal Specifications. Available at:

<http://www.omg.org/spec/> [Consulté avril 2, 2011].

Description : Page web regroupant l'ensemble des spécifications de l'OMG et, plus spécifiquement, celles liées à l'architecture dirigée par les modèles (MOF, QVT, OCL, etc.).

Référence bibliographique : CMMI Product Team, 2010. *CMMI® for Development, Improving processes for developing better products and services*, Software Engineering Institute.

Description : Référence du Capability Maturity Model Integration (un ensemble structuré de bonnes pratiques, destiné à appréhender, évaluer et améliorer les activités des entreprises d'ingénierie).

Référence bibliographique : Rémi Leblond, 1999. *Vers une architecture n-tiers*.

Description : Rapport technique sur le multitiers et les middlewares en général.

Référence bibliographique : Lucas Bonnet, 2008. État de l'art des solutions libres de virtualisation pour une petite entreprise. Available at:

<http://systeme.developpez.com/tutoriels/virtualisation/livre-blanc-lucas-bonnet/> [Consulté juin 22, 2011].

Description : Livre blanc sur les aspects théoriques de la virtualisation ainsi que sur l'ensemble des solutions libres existantes.

7 - Glossaire

Univers desktop, monde desktop : le monde desktop représente les machines comme les ordinateurs fixes, les ordinateurs portables ou les netbooks. Le terme « desktop » fait ici référence au « bureau » des ordinateurs.

Bas niveau : Le terme « bas niveau » fait référence aux mécanismes proches du matériel (bas niveau d'abstraction). Par opposition, le terme « haut niveau » fait référence aux mécanismes abstraits ne tenant pas compte des détails inhérents au fonctionnement des machines (haut niveau d'abstraction).

Framework : Base logicielle offrant un cadre technique et structurel imposant une philosophie au développeur, tout en lui apportant une vision simplifiée des mécanismes couramment exploités dans un projet informatique.

ISO9000 : Désigne un ensemble de normes relatives à la gestion de la qualité publiées par l'Organisation internationale de normalisation (ISO).

CMMi (Capability Maturity Model + Integration) : ensemble de bonnes pratiques permettant d'appréhender, d'évaluer et d'améliorer les activités des entreprises d'ingénierie.

Architecture logicielle : Architecture interne d'une application.

Architecture intergicielle : Architecture des liens entre applications concourant à constituer un système plus vaste.

Mainframe : Un ordinateur central, ou mainframe, est un ordinateur de grande puissance de traitement.

Progiciel de gestion intégrée (ERP) : Logiciel permettant de gérer l'ensemble des processus opérationnels d'une entreprise, en intégrant l'ensemble des fonctions de cette dernière

comme la gestion des ressources humaines, la gestion comptable, financière, mais aussi la vente, la distribution, l'approvisionnement ou le commerce électronique.

Bus d'entreprise (Enterprise Service Bus) : Technique informatique intergicielle permettant la communication d'applications qui n'ont pas été initialement conçues pour cela.

Virtualisation de poste : Activité consistant à virtualiser les postes clients et à les diffuser à travers le réseau vers des terminaux vidés de leur substance (OS Streaming).

Virtualisation d'application : Activité consistant à virtualiser des applications et à les diffuser à travers le réseau.

Virtualisation de serveur : Activité consistant à virtualiser plusieurs serveurs sur un serveur physique de grande capacité (généralement un mainframe) pour assurer un cloisonnement entre lesdits serveurs ainsi que des fonctionnalités de sauvegarde et de restauration accrue.

Client léger : Désigne une application où la totalité de la logique métier est traitée du côté du serveur.

Cloud, cloud computing : Concept qui consiste à déporter sur des serveurs distants des traitements informatiques traditionnellement localisés sur des serveurs locaux ou sur le poste client de l'utilisateur.

Intégration continue : Pratique de génie logiciel consistant à réaliser les processus de test automatiquement et plus ou moins en permanence, pour s'assurer que le code source ne régresse pas.

Paradigme de programmation : Philosophie de développement qui traite de la manière dont les problèmes doivent être formulés et résolus.

Application monolithique : Application non distribuée, c'est-à-dire constituée d'un exécutable et de ses ressources.

Personal cloud : Ensemble des périphériques (embarqués ou non) qui composent l'univers informatique d'un individu (téléphone, électroménager, home cinéma, ordinateurs, etc.).

Réseau pair-à-pair : Modèle de réseau informatique proche du modèle client-serveur mais où chaque client est aussi un serveur.

Réseau ad-hoc : Réseaux sans fil capables de s'organiser sans infrastructure définie préalablement.

8 - Tables

8.1 - Table des matières

1 - INTRODUCTION	1
2 - LE MULTIPLATEFORME, POURQUOI FAIRE ?	12
2.1 - DE L'HETEROGENEITE DES ENVIRONNEMENTS ACTUELS	12
2.2 - LA NECESSITE DE S'OUVRIRE, D'ETRE COMPATIBLE, EXHAUSTIF ET INTEROPERABLE	15
2.2.1 - <i>Qu'est-ce qu'une plateforme, la compatibilité, l'interopérabilité ?</i>	15
La plateforme, une base de travail	15
La compatibilité	16
Le standard et la norme.....	17
L'interopérabilité	17
2.2.2 - <i>La nécessité d'une stratégie d'ouverture</i>	18
2.2.3 - <i>Pourquoi cibler plusieurs plateformes à l'heure actuelle ?</i>	19
2.3 - LA COMPLEXITE ORGANISATIONNELLE ET TECHNIQUE INDUITE PAR LE MULTIPLATEFORME PAR RAPPORT AUX PROJETS CLASSIQUES	21
2.3.1 - <i>La nécessité de raisonner de manière multidimensionnelle</i>	22
2.3.2 - <i>Des difficultés pour gérer le développement et l'évolution</i>	24
Les spécifications	24
La conception	24
La production.....	25
La maintenance et l'évolution	26
2.3.3 - <i>La démultiplication de la charge et des coûts</i>	27
2.4 - AU FINAL, QUELS ENJEUX POUR LE MULTIPLATEFORME ?.....	28
3 - COMMENT SIMPLIFIER L'APPROCHE MULTIPLATEFORME ?	30
3.1 - PAS DE SOLUTION EVIDENTE, MAIS UN EVENTAIL DE POSSIBILITES.....	30
3.2 - AU SEIN MEME DU PROCESSUS DE DEVELOPPEMENT, DES METHODES EXISTENT	30
3.2.1 - <i>Exploiter un code source de référence, une solution fastidieuse</i>	30
3.2.2 - <i>Un mécanisme historique : le preprocessing</i>	33
3.2.3 - <i>Les composants portables, une méthode très appréciée</i>	35
3.2.4 - <i>La conversion automatique de code, un processus complexe et insatisfaisant</i>	36
3.2.5 - <i>Mais ces approches s'intéressent à des aspects locaux</i>	38
3.3 - L'ORGANISATION DU PROJET MULTIPLATEFORME APPORTE ELLE AUSSI DES REPONSES.....	38
3.3.1 - <i>En parallélisant les projets</i>	38
3.3.2 - <i>Ou en utilisant un projet pilote</i>	40
3.4 - LES ARCHITECTURES LOGICIELLES ET INTERGICIELLES, AU SERVICE DE LA TECHNIQUE ET DE L'ORGANISATION, POUR SIMPLIFIER A LA FOIS LE PROCESSUS DE DEVELOPPEMENT, MAIS AUSSI LE PROCESSUS D'EVOLUTION	40

3.4.1 - La granularité et la réflexion au service du découpage intelligent	41
L'importance de la granularité	42
La réflexion, pour factoriser dynamiquement	43
3.4.2 - Abstraire les spécificités techniques au travers des aspects	45
3.4.3 - Aller plus loin dans l'approche objet au travers des composants distribués	48
3.4.4 - Adopter une approche en couches réparties	51
L'architecture 2-tiers	52
L'architecture 3-tiers	53
Pourquoi accroître le nombre de tiers ?	54
3.4.5 - Raisonner par services, pour orchestrer et factoriser	54
3.4.6 - Virtualiser, pour que les systèmes s'adaptent à tout	58
3.4.7 - Placer les modèles au cœur de la fabrication logicielle	61
Concepts de base : le modèle, le métamodèle et le métamétamodèle	64
Le MOF	67
L'approche MDA (Model Driven Architecture)	68
Les métalangages	70
Pourquoi modéliser ?	71
4 - UN CAS PRATIQUE : LA TELEPHONIE MOBILE	73
4.1 - LA TELEPHONIE MOBILE, UN UNIVERS MULTIPLATEFORME PAR EXCELLENCE	73
4.1.1 - Un certain nombre de plateformes mobiles	73
4.1.2 - Un marché inégal et des disparités importantes entre les constructeurs	74
4.1.3 - Une effervescence aux enjeux importants	77
4.2 - UN MIDDLEWARE POUR RATIONALISER LA COMMUNICATION ENTRE LES PLATEFORMES ET LE DEVELOPPEMENT D'APPLICATION DE HAUT NIVEAU	79
4.2.1 - L'INRIA, l'équipe ARLES et la startup Ambientic	79
4.2.2 - iBICOOP, un middleware pour unifier les échanges entre terminaux mobiles	80
Classification des middlewares	80
Présentation du middleware iBICOOP et de son infrastructure	82
4.3 - LA SIMPLIFICATION DU PROCESSUS DE DEVELOPPEMENT ET D'EVOLUTION : APPORTS CONCRETS ET REFLEXIONS FUTURES	87
4.3.1 - En quoi iBICOOP permet-il de simplifier le processus de développement et d'évolution ?	87
4.3.2 - Exemple d'exploitation d'iBICOOP : le partage de photo multiplateforme	90
4.3.3 - iBICOOP, lui aussi victime des difficultés inhérentes au processus de développement des environnements multiplateformes	93
4.3.4 - Les limitations et les perspectives envisageables	98
5 - CONCLUSION	100
6 - BIBLIOGRAPHIE	108
Citations intext	108
Références	108

7 - GLOSSAIRE	113
8 - TABLES	116
8.1 - TABLE DES MATIERES	116
8.2 - TABLE DES ILLUSTRATIONS	118
Figures	118
Tableaux	119

8.2 - Table des illustrations

Figures

FIG. 1 – EVOLUTION DES ORDINATEURS VERS DES RESEAUX DE PLUS EN PLUS PERVASIFS, C'EST-A-DIRE DES RESEAUX D'ORDINATEURS CAPABLES DE SE DECOUVRIR ET DE COMMUNIQUER ENTRE EUX SANS INTERVENTION HUMAINE.	13
FIG. 2 – LA COMPATIBILITE (LE SYSTEME 1 EST COMPATIBLE AVEC LE SYSTEME 2 ET LE SYSTEME 2 EST COMPATIBLE AVEC LE SYSTEME 3)	17
FIG. 3 – LE STANDARD (LE SYSTEME 1 EST UN STANDARD, ET LES AUTRES SONT COMPATIBLES AVEC LUI).....	17
FIG. 4 – L'INTEROPERABILITE	18
FIG. 5 – LE CYCLE EN V.....	21
FIG. 6 – UNE PROGRESSION LINEAIRE (A GAUCHE) ET UN ENSEMBLE DE PROGRESSIONS LINEAIRES PARALLELISEES (A DROITE)	22
FIG. 7 – EXEMPLE DE DOUBLE DECOUPAGE POUR UN PROJET MULTIPLATEFORME A DEUX PLATEFORMES.....	23
FIG. 8 – ILLUSTRATION DU DECALAGE ENTRE DEUX VERSIONS	28
FIG. 9 – PROCESSUS SIMPLIFIE DE TRANSFORMATION D'UN CODE SOURCE DE REFERENCE	31
FIG. 10 – REPERCUTION DES ERREURS.....	32
FIG. 11 – LES ETAPES DE PRODUCTION D'UN CODE EXECUTABLE PAR LE COMPILATEUR GNU COMPILER COLLECTION (GCC), A PARTIR D'UN CODE SOURCE ECRIT EN C	34
FIG. 12 – LE MECANISME DE PREPROCESSING EN C (EXEMPLE)	34
FIG. 13 – PREPROCESSEUR GENERIQUE CAPABLE DE MANIPULER DES INSTRUCTIONS INTEGREES DANS LE CODE SOURCE DE PLUSIEURS LANGAGES.....	35
FIG. 14 – UNE PREMIERE FORME D'ABSTRACTION DES SPECIFICITES LIEES AUX PLATEFORMES	36
FIG. 15 – TRANSFORMATION AUTOMATIQUE DU CODE SOURCE	36
FIG. 16 – PARRALELISATION DES PROJETS	39
FIG. 17 – UTILISATION D'UN PROJET PILOTE	40
FIG. 18 – DEUX APPLICATIONS AVEC DEUX PARTIES SPECIFIQUES (S1 ET S2) IDENTIQUES.....	43
FIG. 19 – INSTANCIATION D'UN ELEMENT PAR L'UNITE GENERIQUE	44
FIG. 20 – ILLUSTRATION DU FAIBLE COUPLAGE ET DE LA FORTE COHESION	45
FIG. 21 – EXEMPLE D'EXIGENCE NON FONCTIONNELLE TRANSVERSALE	46
FIG. 22 – PRINCIPE GENERAL DE LA PROGRAMMATION ORIENTEE ASPECT	48
FIG. 23 – EXEMPLE DE COMPOSANT DISTRIBUE AVEC COMMUNICATION INTERMACHINES (RESEAU) : COUPLAGE TRES LACHE.....	50

FIG. 24 – EXEMPLE DE COMPOSANT DISTRIBUE AVEC MEDiateur (BUS LOGICIEL, COMMUNICATION INTER-PROCESSUS, ETC.) :	
COUPLAGE PLUS IMPORTANT DU AU SYSTEME D’EXPLOITATION COMMUN	50
FIG. 25 – CLASSIFICATION DES TYPES D’ARCHITECTURES CLIENT-SERVEUR PAR LE GARTNER GROUP	52
FIG. 26 – ARCHITECTURE 3-TIERS GENERIQUE (EN HAUT) ET EXEMPLE COURANT D’IMPLEMENTATION	53
FIG. 27 – ARCHITECTURE D’UN VIRTUALISEUR TOTAL (A GAUCHE), D’UN PARAVIRTUALISEUR (AU CENTRE) ET D’UN HYPERVISEUR (A DROITE)	60
FIG. 28 – ILLUSTRATION DE L’IMPORTANCE QUE PEUT AVOIR LE POINT DE VUE DE L’OBSERVATEUR SUR LA DEFINITION DES PRINCIPES DE SPECIFICITE/GENERICITE	62
FIG 29 – METAMODELE DU DIAGRAMME DE CLASSE UML (OMG, 2011).....	65
FIGURE 30 – LE SYSTEME, LE MODELE, LE LANGAGE ET LE METAMODELE	66
FIG. 31 – EXEMPLE D’APPLICATION DE LA HIERARCHIE A QUATRE NIVEAUX (OMG, 2011)	67
FIG. 32 – EXTRAIT DU MOF (OMG, 2010)	68
FIG. 33 – PROCESSUS DE DEVELOPPEMENT SELON L’APPROCHE MDA (HAMMOUDI, 2010)	70
FIG. 34 – EXTRAIT DU METAMODELE AS REPRESENTANT LA CONSTRUCTION ET LA DESTRUCTION D’OBJET (A GAUCHE) ET EXEMPLE DE CONTRAINTE OCL (A DROITE).....	71
FIG. 35 – PART DE MARCHÉ DES SMARTPHONES AUX ETATS-UNIS (NIELSEN, 2011)	75
FIG. 36 – MIDDLEWARE RPC (A GAUCHE) ET MIDDLEWARE ORIENTE EVENEMENTS (A DROITE)	81
FIG. 37 – MIDDLEWARE A ESPACE PARTAGE (A GAUCHE) ET MIDDLEWARE ORIENTE MESSAGE (A DROITE).....	81
FIG. 38 – ARCHITECTURE DU MIDDLEWARE IBICOOP	83
FIG. 39 – LES COUCHES DE COMMUNICATION D’IBICOOP	84
FIG. 40 – LE SERVEUR D’AUTHENTIFICATION (A GAUCHE) ET LA PROCEDURE STANDARD D’AUTHENTIFICATION (A DROITE).....	85
FIG. 41 – LE SERVEUR DE COMMUNICATION (A GAUCHE) ET UNE COMMUNICATION STANDARD EN MODE REQUETE/REPONSE (A DROITE)	86
FIG. 42 – LE SERVEUR DE RESSOURCES	86
FIG. 43 – EXTRAIT DES ABSTRACTIONS POUR LA MANIPULATION DU STOCKAGE ET LES OPERATIONS DE CHIFFREMENT	88
FIG. 44 – ABSTRACTION DES SPECIFICITES DU STOCKAGE.....	88
FIG. 45 – CAPTURES DE LA DECOUVERTE (A GAUCHE) ET DE L’ECHANGE DE FICHIERS (A DROITE)	91
FIG. 46 – CAPTURES DE L’ACCES A LA GALERIE D’IMAGES DU TELEPHONE.....	92
FIG. 47 – UNE METHODE REALISANT LA DECOUVERT DES DESTINATAIRES, EN JAVA (A GAUCHE) ET EN C# (A DROITE).....	93
FIG. 48 – L’ENTITE COMMUNE IBICOOP CORE ET SES EXTENSIONS	94
FIG. 49 – LE PRINCIPE DE LA CONVERSION NESTENE.....	96
FIG. 50 – EXEMPLE DE FONCTIONNALITES COMPOSEES ET FONCTIONNALITES ATOMIQUES	97
FIG. 51 – LE PROCESSUS PHONE GAP	99

Tableaux

TABLEAU 1 – REPARTITION DES DIFFERENTS SYSTEMES D’EXPLOITATION POUR SMARTPHONES DANS LE MONDE (IDC, 2011).....	74
--	----