Interfaces homme-machine plastiques : une approche par composants dynamiques

Lionel Balme

THÈSE
présentée et soutenue publiquement par

Lionel Balme

pour obtenir le titre de
DOCTEUR de L'UNIVERSITE JOSPEH FOURIER — GRENOBLE 1
(arrêtés ministériel du 5 juillet 1984 et du 30 mars 1992)
Spécialité : Informatique

Interfaces homme-machine plastiques : Une approche par composants dynamiques

Soutenue le **20 juin 2008** devant le jury composé de

Président :	M. Philippe Lalanda
Directeur de thèse :	Mme Joëlle Coutaz
Rapporteurs :	M. Michel Riveill M. Jean Vanderdonckt
Examinateurs :	M. Philip Gray M. Jean-Claude Tarby

Thèse préparée au sein du Laboratoire d'Informatique de Grenoble (LIG)
385, rue de la Bibliothèque, BP 53, 38041 GRENOBLE CEDEX 9
Université Joseph Fourier — Grenoble 1

Enfin rédigée !

Remerciements

La rédaction de la page des remerciements, synonyme de « fin de la thèse », est un instant que j'attendais avec impatience. Le moment étant venu, il s'agit de n'oublier personne.

Mes premiers remerciements vont à Joëlle Coutaz qui m'a offert la chance de commencer une thèse sous sa responsabilité et qui a mis à ma disposition tous les moyens de l'accomplir avec succès. Succès qui m'amène à remercier les membres de mon jury : Michel Riveill et Jean Vanderdonckt qui m'ont fait l'honneur d'accepter d'être mes rapporteurs, ainsi que Philip Gray, Philippe Lalanda et Jean-Claude Tarby, qui m'ont fait l'honneur de bien vouloir être mes examinateurs.

Cette thèse n'aurait pu aboutir sans la présence à mes côtés d'une équipe remarquable. C'est pourquoi je remercie les membres de l'équipe IIHM du laboratoire LIG pour les échanges fructueux et le bon temps passé ensemble.

Puisque mener à bien un travail de ce type déborde (très) largement sur le temps normalement consacré à la vie personnelle, je remercie également ma famille, mes amis et Marie-Félicie qui ont su ne pas tenir rigueur de mon manque de disponibilité et même me soutenir dans les moments de doute et de tergiversation. Je remercie tout particulièrement mon Papa et ma Maman qui m'ont permis d'arriver jusque là, ainsi que Michelle pour la relecture très attentive de ce document ô combien obscur pour qui n'est pas de la partie.

Comme ces remerciements ne sont pas des mots en l'air, je décide d'utiliser cet espace un peu particulier pour vous faire (je l'espère !) un cadeau : vous trouverez ci-dessous ma recette (à moi) de la tartiflette, pour 4 personnes :

• env. 1,2 kg de patate (avant épluchure)
• 200g de lardons fumés
• 2 oignons (pour un poids de 250g, quoi)
• crème épaisse (bon, alors moi je prends de la vraie bonne crème à 40%, mais j'en mets moins, alors c'est pas si lourd que ça)
• 1/2 verre de vin blanc de Savoie, ou peut-être un poil plus. Bon, moins d'un verre en tout cas.
• 2 reblochons fermiers

Note : le tout tient dans un plat à lasagne, en principe.

1) On épluche les patates à l'aide du willi waller 2006, là, comme sur la vidéo. (voir http://www.tetesaclaques.tv/video.php?vid=30)

2) On cuit à l'eau les patates entières. Attention, les patates ne doivent pas être trop cuites, pour ne pas partir en purée. Mais suffisamment pour être fondantes. Il vaut mieux les faire un tout petit peu moins cuire que trop : elles finiront de cuire au four, de toutes façons.

3) Pendant ce temps, faire revenir les oignons et les lardons. (Alors personnellement, je ne les fais pas revenir ensemble, parce que ce n'est pas la même cuisson. Les lardons reviennent d'un côté, sans matière grasse et à feu vif, dans une petite poêle, de manière à les dorer sans trop les faire réduire. De l'autre côté, je fais revenir plus doucement les oignons dans un peu de beurre demi-sel, dans une grande poêle)

4) Puis, je passe en feu doux, je verse les lardons dans les oignons (qui sont dans la grande poêle), j'ajoute 2 à 3 cuillers de crème, je sale, je poivre, suivant l'humeur j'ajoute un peu de noix de muscade ou un peu de baies roses moulues, je tourne doucement pour bien mélanger le tout et fluidifier la crème. Attention, la crème doit chauffer, mais elle ne doit pas bouillir ! On est sur du feu très doux, là !

5) Une fois la crème fluide, sympathiquement colorée par les oignons et les lardons, j'ajoute les patates coupées en morceaux de taille moyenne (genre plus gros que pour les patates rissolées, mais pas énorme non plus... disons 25 à 30 mm de côté + ou – 10 %). Je tourne afin de bien mélanger les patates et la sauce.

6) Dans le plat à lasagne dont le fond contient le 1/2 verre de vin blanc ou un peu plus, je verse le contenu de la poêle. Attention à verser doucement, ou à mettre un tablier, sinon ça éclabousse la jolie chemise propre tout juste repassée. J'égalise pour obtenir une couche de mélange bien plate, et si besoin, je répartis bien les oignons et lardon à l'aide d'une cuillère en bois.

7) J'ajoute ensuite le fromage par dessus. Il y a deux écoles : ceux qui coupent le fromage en lamelles, et ceux qui le coupent en deux, dans le sens de la section. Moi, je fais des lamelles, parce que je trouve que le fromage se mélange mieux au reste comme ça. L'avantage de l'autre solution, c'est que ça fait une couche de croûte grillée à la cuisson au four et c'est pas mal non plus. Dans tous les cas, il faut penser à enlever la petite pastille de caséine verte présente dans la croûte de chaque reblochon fermier ! Sinon, c'est pas très bon... Enfin, moi j'aime pas trop quoi.

8) Je laisse reposer tout ça à température ambiante, à l'abri dans mon four. Le mieux, c'est de préparer le midi pour le soir, voir même, la veille pour le lendemain. Dans le cas d'une préparation la veille, une fois le plat froid, je couvre d'un film étirable et je mets au réfrigérateur. C'est important le film étirable, ça évite (1) d'avoir la tartiflette contaminée par les germes et bactéries présentes dans le frigo et (2) de parfumer tout le frigo au reblochon n°5 de Chamel... Bref, moi ce que j'en dis !

9) Juste avant le service, faire cuire au four à 200º pendant 20 à 30 minutes en surveillant : le fromage doit être fondu, faire bloupbloup et sa croûte doit être croustillante sans être carbonisée. Bon appétit !

Table des matières

Chapitre I

Introduction

Avant-propos

« Les technologies les plus puissantes sont celles qui disparaissent. Elles se fondent dans notre univers quotidien jusqu'à ne faire plus qu'un avec lui. Prenez l'écriture, peut-être la toute première technologie de l'information. Sa capacité à représenter le langage parlé de manière symbolique pour un stockage de longue durée libère l'information des limites de la mémoire individuelle. Aujourd'hui, l'écriture est devenue omniprésente dans nos pays industrialisés. Elle permet aux livres et aux journaux de convoyer de l'information, comme elle le permet aux panneaux routiers, aux affiches publicitaires, aux enseignes des magasins et même aux graffitis. Les emballages des bonbons sont eux aussi couverts d'inscriptions. La présence constante en arrière-plan de ces produits de la « technologie des lettres » ne requiert pas d'attention active, mais l'information à transmettre est prête à être utilisée immédiatement. Il est difficile d'imaginer la vie moderne autrement.

[…] Une telle disparition est une conséquence fondamentale, non pas de la technologie, mais de la psychologie humaine. Lorsqu'une personne apprend suffisamment bien quelque chose, elle cesse d'y faire attention. Par exemple, si vous regardez un panneau routier, vous absorbez l'information qu'il contient, sans même être conscient d'avoir effectué un acte de lecture.

[…] C'est seulement quand les choses disparaissent de cette façon que nous devenons capables de les utiliser sans y penser, que nous pouvons, au-delà d'elles, nous focaliser sur de nouveaux buts. »

Mark Weiser, "The Computer for the 21st Century", Scientific American, p 94-104, September 1991.

1. Sujet

Ces travaux de recherche doctorale s'inscrivent dans le domaine de l'Interaction Homme-Machine (IHM) et s'intéressent à la question de l'exécution des systèmes interactifs dans le cadre de l'informatique ubiquitaire.

L'informatique ubiquitaire est une idée introduite par Mark Weiser en 1991 [Weiser91], dans un article qui fait aujourd'hui référence. Weiser constate que les technologies les plus ancrées dans notre vie de tous les jours sont celles qui savent s'y fondre, jusqu'à y disparaître. Il illustre son propos avec l'exemple de l'écriture : devenue omniprésente dans nos sociétés modernes,

chacun l'utilise au quotidien sans même y prêter attention. Comparé à l'écriture, l'ordinateur tel que nous le connaissons aujourd'hui, bien que très répandu, est loin d'être aussi profondément intégré à notre vie. Selon Weiser, l'ordinateur d'aujourd'hui ne constitue qu'une étape vers l'informatique ubiquitaire où son ancrage dans notre quotidien serait tel qu'il nous rendrait continuellement des services indispensables sans que personne ne remarque plus sa présence.

Lyytinen et Yoo [Lyytinen02] résument l'évolution vers l'informatique ubiquitaire selon deux dimensions (en Figure I-1). L'une mesure le niveau d'intégration des dispositifs numériques dans notre environnement, l'autre, leur niveau de mobilité. Ainsi, l'informatique traditionnelle se caractérise par des dispositifs numériques très peu mobiles et peu intégrés dans l'environnement. C'est par exemple le cas de nos ordinateurs de bureau classiques. Avec la miniaturisation des dispositifs, le développement des réseaux sans-fil et l'apparition des assistants personnels (PDA) et des téléphones mobiles, nous sommes entrés dans l'ère de l'informatique mobile. Les systèmes interactifs qui caractérisent cette dimension nous accompagnent dans tous nos déplacements. Toutefois, ils ne sont pas intégrés à notre environnement dans la mesure où ils sont incapables de le sonder pour s'y adapter.

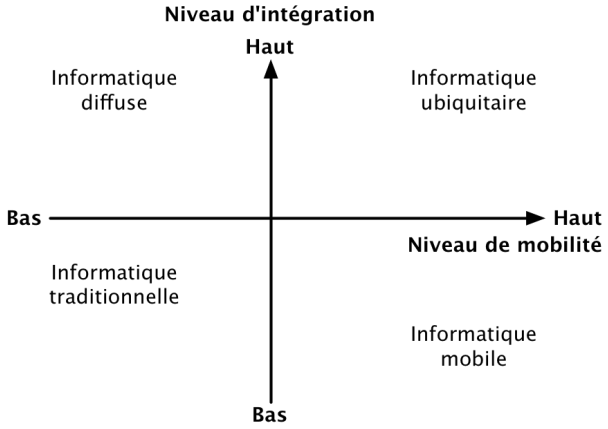


Figure I-1. Dimensions de l'informatique ubiquitaire [d'après Lyytinen02]

La capacité à sonder l'environnement et à en tenir compte caractérise le niveau d'intégration d'un système interactif. De manière réciproque, l'environnement peut être capable de détecter les dispositifs numériques et les objets qu'il contient. De cette interaction mutuelle résulte la capacité des systèmes interactifs d'agir « intelligemment » dans et sur leur environnement. C'est, selon Lyytinen et Yoo, l'idée d'informatique diffuse. L'informatique ubiquitaire dans laquelle s'inscrivent nos travaux de recherche, émerge de la convergence des univers de

l'informatique mobile et de l'informatique diffuse. Notons que de nombreux termes sont utilisés et discutés aujourd'hui pour désigner ou décliner la vision originale de Weiser, ou pour s'intéresser à un aspect particulier de cette vision : informatique ambiante, intelligence ambiante, informatique pervasive, l'ordinateur invisible[1] ou l'ordinateur évanescent[2], objets communicants, etc. Dans la suite de ce mémoire, je retiens l'esprit que couvre le terme d'informatique ubiquitaire telle qu'imaginée par Weiser.

Du point de vue de l'interaction homme-machine, l'informatique ubiquitaire pose la question du délicat équilibre entre interaction implicite et interaction explicite, entre autonomie des systèmes et maîtrise laissée à l'utilisateur, entre « simple » usager obligé de se conformer à une interface utilisateur imposée et l'utilisateur créatif qui façonne son espace interactif de manière à inventer de nouveaux services. Ainsi, la nature des interfaces utilisateur s'en trouve radicalement changée. Pour faire face à cette (r)évolution, les systèmes interactifs doivent devenir plastiques [Thevenin99].

La plasticité des interfaces homme-machine dénote la capacité que ces interfaces ont à s'adapter ou à être adaptées au contexte de l'interaction tout en préservant leur utilisabilité. Du point de vue d'un système interactif, le contexte de l'interaction se définit par un triplet constitué de la plate-forme, de l'utilisateur et de l'environnement physique et social [Thevenin99]. La notion de plate-forme désigne l'ensemble des logiciels et des ressources de calcul, de communication et d'interaction mis en jeu dans une interaction avec l'utilisateur. L'utilisateur se définit par ses capacités sensori-motrices et cognitives, ses caractéristiques socio-culturelles, ainsi que par ses objectifs et ses activités. Enfin, l'environnement physique et social décrit toutes les dimensions de l'espace réel où se déroule l'interaction entre l'utilisateur et le système interactif.

En informatique conventionnelle, le contexte de l'interaction est stable et connu dès la conception du système. Les systèmes interactifs sont alors conçus pour une plate-forme identifiée à l'avance, en vue de la réalisation de tâches précises dans un environnement physique et social connu. L'informatique ubiquitaire implique l'imprévisibilité et la variabilité du contexte d'interaction. Ainsi, les systèmes interactifs doivent évoluer pour s'adapter, à l'exécution, au comportement opportuniste d'un

[1] The invisible computer tel que le définit D. Norman dans son livre [Norman98].

[2] The disappearing computer, terme introduit par Jakub Wejchert lors du lancement du programme FET du 5ème PCRD Européen

utilisateur mobile. Mon travail de recherche s'inscrit dans le cadre de cette évolution, et je m'intéresse plus particulièrement aux mécanismes généraux nécessaires à l'adaptation des interfaces homme-machine à l'exécution.

2. Motivations

La question de l'adaptation des interfaces homme-machine est traitée dans trois domaines distincts de l'informatique : l'Intelligence Artificielle (IA), le Génie Logiciel (GL), et l'Interaction Homme-Machine (IHM). Ces domaines se côtoient sans toutefois exploiter leurs richesses mutuelles pour de nouvelles avancées.

En IA, c'est l'adaptation à l'utilisateur qui est visée pour en améliorer les performances : adaptation de l'aide et du contenu des messages d'erreur [Browne90], calcul dynamique de valeurs par défaut [Berthomé-Montoy95], génération automatique de macrocommandes ou détection de tâches répétitives comme dans Eager [Cypher91], documents hypermédia adaptatifs [Brusilovsky01], [Kobsa01], génération d'interfaces multimédia intelligentes [Maybury93], [André93]. Souvent, le modèle à agent prévaut, avec planification comme dans Roadie qui facilite la configuration d'appareils multimédia reliés en réseau domestique [Lieberman06]. Roadie est un agent IHM qui, s'appuyant sur une base de connaissances et sur l'acquisition des actions de l'utilisateur, en devine les objectifs et génère une aide contextuelle à initiative mixte.

En GL, c'est l'adaptation aux changements de ressources qui est visée. L'objectif est d'assurer un fonctionnement optimal du système et ceci de manière autonome (c'est-à-dire, sans intervention humaine). Plusieurs approches sont proposées : tissage de comportements selon les principes de la Programmation par Aspect (AOP) [Kiczales97], adaptation par transformations à différents nivaux d'abstraction selon les principes de l'Ingénierie Dirigée par les Modèles (IDM) [Favre04-1], réflexivité et reconfiguration dynamique de composants orientés services [Cervantes04]. Cependant, comme nous le verrons dans notre analyse de l'état de l'art, aucune de ces solutions ne prend en compte de manière explicite les spécificités de l'Interaction Homme-Machine. En particulier, une reconfiguration « optimale » au regard des performances techniques n'est pas nécessairement « optimale » au regard des attentes de l'utilisateur.

En IHM, au contraire du GL qui embrasse tout le système, l'attention porte sur une partie spécifique et restreinte des logiciels interactifs : l'interface utilisateur. Les approches sont multiples :

codage avec des boîtes à outils spécialisées [Grolaux05], outils de génération pour la conception d'IHM multicibles [Thevenin99], [Florins04], [Paternò02], et infrastructures logicielles d'exécution [Sousa05], [Tandler01], [Edwards01]. Bien que complémentaires, ces approches et leurs outils ont été exploités de manière exclusive fondée sur une distinction nette entre phases de conception et phase d'exécution. En conséquence, à l'exception notable de l'approche utilisée dans [Lewandowski07], les modèles de haut niveau d'abstraction produits en phase de conception et riches en sémantique centrée sur l'utilisateur comme le modèle de tâches, ne sont plus disponibles à l'exécution pour influencer de manière rationnelle l'adaptation des IHM. La rétro-conception à la volée à partir du code source, comme on la pratique sur des pages html [Bouillon02] ou des documents multimédia [Laborie06], [Layaida05] est intéressante mais est limitée à un espace technologique bien cerné : celui du Web et de ses formalismes.

La seule approche générale qui aujourd'hui fait référence en IHM est le processus de développement à la IDM : transformations successives du modèle de tâche et des concepts métier en IHM abstraite, puis en IHM concrète et en IHM finale. Bon nombre de travaux sur l'adaptation des IHM ont procédé (et procèdent encore) selon ce schéma directeur. L'idée est de spécifier l'IHM une fois, à haut niveau d'abstraction, puis, par transformation, obtenir différentes déclinaisons de cette IHM pour différents contextes d'interaction [Thevenin99], [Florins04], [Paternò02]. Cependant, cette approche est limitée en deux points :

- Les contextes d'interaction cibles doivent être connus par avance, ce qui est contraire aux principes de l'informatique ubiquitaire où imprévu et opportunisme prévalent.

- Les interfaces utilisateur obtenues par ce biais doivent se satisfaire d'interacteurs conventionnels (bouton, menu, formulaire). Ce type d'IHM, très répandu, est parfaitement pertinent dans l'univers de l'informatique traditionnelle. Mais l'informatique ubiquitaire, en propulsant les IHM bien au-delà de l'écran, du clavier et de la souris de la classique station de travail, fait la part belle aux nouvelles techniques d'interaction post-WIMP.

En synthèse, l'IA, le GL et l'IHM produisent des solutions qui visent toutes l'adaptation des logiciels pour mieux servir l'utilisateur. Mais ces solutions diffèrent par les paradigmes sous-jacents, par leur niveau de généralité, et par la place laissée à l'utilisateur et à son environnement physique et social. L'IA est plutôt confinée à une forme d'adaptation à l'utilisateur. Le GL apporte des solutions générales, mais l'utilisateur n'y est pas un concept de première classe. En IHM, la seule approche générale procède comme l'IDM du Génie Logiciel, mais les IHM ainsi

générées sont prévues pour des contextes cibles prévus à l'avance et sont de médiocre qualité comparée aux IHM que pourrait obtenir un stylicien humain ou un développeur avec une boîte à outils post-WIMP. De plus, en IA, en GL comme en IHM, on observe une forte dichotomie entre phase de conception et phase d'exécution provoquant la disparition de modèles riches en sémantique qui pourraient utilement informer le processus d'adaptation à l'exécution. Ce constat appelle les objectifs et approche suivants.

3. Objectifs et approche

L'objectif de mes travaux de thèse est de *comprendre la problématique de la plasticité des IHM à l'exécution et de proposer des bases conceptuelles générales pour la traiter.* Ce cadre général devra permettre la capitalisation de mécanismes de reconfiguration logicielle propre à la plasticité de manière à faciliter ensuite l'implémentation de systèmes interactifs plastiques. Ainsi, plutôt que de concevoir une solution ad hoc optimale pour une classe particulière de systèmes interactifs plastiques, il s'agit d'identifier des concepts, des fonctions et des mécanismes généraux, puis de proposer un cadre pour leur implémentation.

Le cadre conceptuel visé doit pouvoir intégrer le meilleur de l'apport de l'IA, du GL, et de l'IHM. Il ne s'agit pas en effet de remplacer l'existant, mais de permettre l'intégration de pratiques et d'approches qui ont chacune leurs adeptes. En particulier, mon travail doit répondre aux requis suivants :

- Estomper, voire éliminer, la distinction entre phase de développement et phase d'exécution. Satisfaire ce requis, c'est conserver l'approche IDM qui fait référence en IHM et c'est se garder la possibilité d'exploiter à l'exécution des modèles de haut niveau d'abstraction pour adapter les IHM sur des fondements sémantiques, non pas seulement sur des éléments syntaxiques cosmétiques de surface.

- Permettre la réalisation d'IHM hybrides. Satisfaire ce requis, c'est concilier la génération d'IHM conventionnelle avec le codage à la main d'IHM avancée. Autrement dit, certaines parties d'une IHM pourront être le résultat d'une génération automatique tandis que d'autres parties pourront être des composants logiciels produits directement avec une boîte à outils pour des besoins d'interaction extrêmement difficiles, voire impossible, à obtenir à partir de spécifications de haut niveau.

- Permettre la reconfiguration dynamique d'éléments logiciels pas nécessairement interopérables. Satisfaire ce requis, c'est permettre que les différentes parties d'une IHM soient issues de modèles à composants différents. Cette hétérogénéité est une donnée de l'informatique ubiquitaire. Il s'agit donc de la prendre en compte sans imposer aux développeurs un intergiciel qui n'est le leur. La Figure I-2 montre un exemple d'IHM constituée de composants hétérogènes.

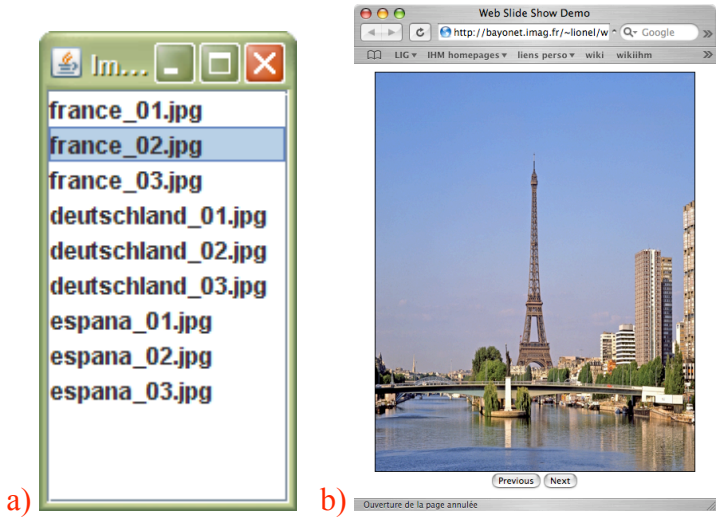


Figure I-2 . Un album de photos dont l'IHM résulte de la reconfiguration dynamique de composants hétérogènes : a) la liste des photos de l'album implémentée sous la forme d'une classe Java qui utilise la boîte à outils SWING b) la version Web sous forme d'un composant qui s'auto-décrit comme utilisateur d'un composant browser HTML compatible Ajax; c) le composant programmé à façon avec la boîte à outils postWIMP GML-Canvas [Bérard06].

Pour atteindre ces objectifs, il convient en premier lieu de bien identifier le problème des IHM plastiques et de là, étudier les offres de l'état de l'art. Je m'appuierai, naturellement, sur les pratiques de référence de l'Interaction Homme-Machine pour envisager ensuite leur généralisation avec l'apport de l'IA et du

GL. L'apport de l'IA étant étudié par ailleurs dans l'équipe [Ganneau07], j'explore ici le rapprochement avec le GL, sans exclure les possibilités d'intégrer des éléments d'inspiration IA. Il semble intéressant de confronter les requis et particularités des IHM aux solutions d'adaptation dynamique offertes par le monde du système avec les approches à composants orientés service.

4. Résultats attendus

Il est attendu un *cadre conceptuel intégrateur* qui permet l'exécution de systèmes interactifs dont les IHM plastiques peuvent résulter de l'assemblage dynamique d'éléments issus d'espaces technologiques différents, par conséquent, pas nécessairement interopérables. Certains de ces éléments pourront provenir d'une génération automatique (voire semi-automatique contrôlée par l'humain) à partir de la spécification de modèles, tandis que d'autres auront être programmés au moyen d'une boîte à outils spécialisée post-WIMP par exemple comme GMLCanvas [Bérard06], ArtToolkit [ARToolKit] ou ECT [Greenhalgh04].

Sur le plan technique, ce cadre se traduit par une *infrastructure d'exécution* de même niveau d'abstraction que les intergiciels. Toutefois, sur le plan pratique, il ne s'agit pas de développer encore un autre intergiciel pour l'informatique ubiquitaire, mais de proposer une solution de plus haut niveau d'abstraction qui peut être instanciée dans différents espaces technologiques et qui offre des moyens d'abolir les frontières technologiques posées par ces espaces.

Mais ces résultats ne prétendent pas tout résoudre. J'en précise ci-dessous la portée et les limites.

5. Portée et limites de l'étude

Je définis le périmètre de mon étude par les quatre axes de la Figure I-3. Le premier axe caractérise le type de systèmes interactifs ciblés dans cette thèse. Les trois autres caractérisent la portée des mécanismes de reconfiguration envisagés : quoi ? comment ? et par qui ? Je détaille à présent chaque axe et en précise la couverture envisagée par mes travaux.

Axe utilisateur. En ingénierie de l'interaction homme-machine, les systèmes interactifs mono et multi-utilisateurs font généralement l'objet d'études distinctes. Cette pratique se justifie car les systèmes interactifs multi-utilisateurs sous-tendent des tâches de collaboration entre utilisateurs qui introduisent une complexité

supplémentaire dans l'interaction et dans les solutions techniques logicielles. Ainsi, afin de concentrer mon étude sur la question de la plasticité des systèmes interactifs, je laisse de côté dans cette thèse les aspects propres aux systèmes collaboratifs.

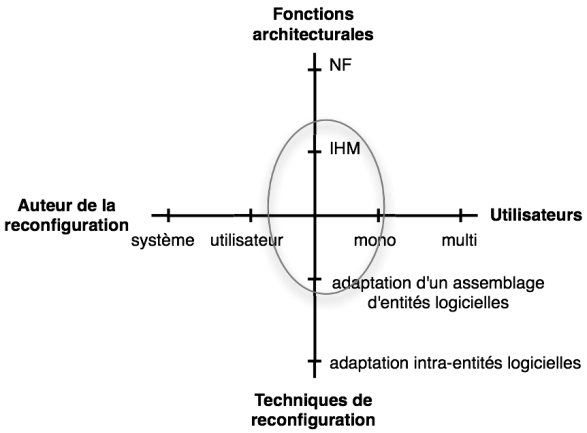


Figure I-3 Portée de l'étude.

Axe fonctions architecturales. De la même manière, dans notre domaine, nous décomposons habituellement un système interactif en cinq fonctions. D'une part, le noyau fonctionnel (NF) implémente le service rendu à l'utilisateur et d'autre part, l'adaptateur de noyau fonctionnel (ANF), le contrôleur de dialogue (CD), les présentations logique (PL) et physique (PP) constituent les fonctions propres de l'IHM [uims92]. La contribution de cette thèse s'inscrivant dans le domaine de l'interaction homme-machine, l'étude ne porte que sur l'implémentation et la reconfiguration à l'exécution des différentes fonctions de l'interface utilisateur. La reconfiguration dynamique de noyau fonctionnel n'est pas traitée.

Axe technique de reconfiguration. Comme nous le verrons dans l'analyse de l'état de l'art, l'adaptation d'une IHM peut se faire selon deux approches complémentaires : par transformation interne d'une entité logicielle IHM et/ou par transformation de l'assemblage des entités logicielles constituant l'IHM. Si la première approche est intéressante et fait l'objet de nombreux travaux, par exemple [Florins04], [Demeure07], elle n'est pas suffisante. En effet, une variation dans la plate-forme peut conduire à l'ajout ou au retrait d'entités logicielles. Pour gérer ce type d'évènements, la reconfiguration logicielle est incontournable. Mes travaux portent plus particulièrement sur cette seconde approche, mais n'excluent pas l'auto-adaptation d'entités logicielles IHM par transformation interne.

Axe auteur de la reconfiguration. Dans le cas général, l'auteur de la reconfiguration peut être l'un ou l'autre des deux acteurs de l'interaction : le système interactif ou l'utilisateur. Dans une

troisième approche possible, une collaboration s'établit entre ces deux acteurs pour mener à bien la reconfiguration souhaitée. Dans la première approche, le système interactif est capable de se reconfigurer sur sa propre initiative et sans l'aide de l'utilisateur. C'est le mode de fonctionnement envisagé en « autonomic computing » [Kephart03]. Cependant, il est bien connu en IHM que l'utilisateur doit pouvoir inspecter et contrôler l'état du système. Mieux, dans notre vision des systèmes interactifs en informatique ubiquitaire, l'utilisateur devrait être en mesure de façonner son espace interactif, par couplage dynamique de ressources d'interaction notamment [Barralon06]. Dans cette thèse, l'objectif est de proposer des mécanismes permettant une reconfiguration à l'exécution du système interactif selon les trois approches : par le système, par l'utlisateur ou par une collaboration système-utilisateur. Cependant, dans ces travaux, je ne traite pas de la problématique de la reconfiguration autonome, ni de la problématique de la reconfiguration d'un système interactif par l'utilisateur final. Par contre, les mécanismes de reconfiguration proposés devront servir de fondements techniques au développement de solutions à ces problèmes.

6. Organisation du manuscrit

La structure du rapport reflète ma démarche.

- Au chapitre suivant, je définis précisément l'espace problème de la plasticité à l'exécution, du point de vue de l'interaction avec l'utilisateur. De cet espace problème, je déduis des requis quant aux techniques d'implémentation à mettre en œuvre pour construire des systèmes interactifs plastiques.

- Au chapitre III, j'analyse l'état de l'art des approches et techniques en IHM en matière d'adaptation des interfaces utilisateurs et des systèmes interactifs pour l'informatique ubiquitaire, au regard de l'espace problème identifié au chapitre précédent.

- Au chapitre IV, j'analyse les propositions issues du monde du génie logiciel au regard des requis également identifiés au chapitre II qui portent sur la distribution, la reconfiguration dynamique et l'hétérogénéité des technologies.

- Au chapitre V, je présente mes contributions sur le plan conceptuel. Ce chapitre s'organise en deux parties. Dans la première, je propose une décomposition fonctionnelle de haut niveau d'abstraction qui décrit l'ensemble des fonctions requises pour l'exécution de systèmes interactifs plastiques. Dans la deuxième, je propose Ethylene, un modèle à

composants dynamiques conçu autour des propriétés satisfaisant les besoins en génie logiciel posés par la plasticité.

- Au chapitre VI, je présente mes contributions sur le plan technique. Elles se composent d'un langage XML destiné à la spécification de composants logiciels pour la plasticité, et d'un cadre de développement qui implémente le modèle Ethylene présenté au chapitre précédent. Ainsi, le contenu de ce chapitre se rapproche d'un mode d'emploi technique à l'usage d'un développeur. Ces deux contributions techniques sont alors mise en œuvre au sein d'un intergiciel pour la plasticité et d'un système interactif de démonstration.

- Enfin, le chapitre de conclusion résume les contributions et s'ouvre sur les prolongements possibles de ces travaux de thèse.

- On trouvera en annexe les spécifications complètes du langage EthyleneXML et les spécifications EthyleneXML des composants impliqués dans mon démonstrateur.

Chapitre II

IHM plastiques : espace problème

Avant-propos

Le problème de la plasticité des systèmes interactifs traite de l'adaptation de l'interface utilisateur au contexte de l'interaction pour en préserver l'utilisabilité. L'utilisabilité est une notion apparue dans les années 1980s au moment de l'avènement, dans le domaine de l'interaction homme machine, de la conception centrée sur l'utilisateur. Dans ce cadre, elle est une des dimensions qui caractérisent l'acceptabilité d'un système interactif. Pour [Nielsen93] (voir Figure II-1), l'utilisabilité se définit par cinq facteurs : la facilité d'apprentissage, l'efficacité, la facilité de mémorisation, sa propension à éviter que l'utilisateur fasse des erreurs (ou pro-activité) et enfin son attractivité.

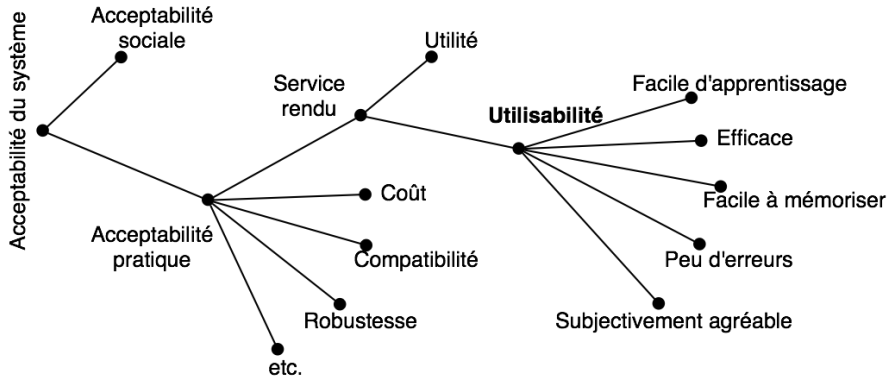


Figure II-1 L'utilisabilité : une dimension de l'acceptabilité d'un système [d'après Nielsen93].

Le standard international [ISO 9241] définit l'utilisabilité comme « La mesure selon laquelle un produit peut être utilisé par des utilisateurs donnés pour réaliser des buts donnés avec efficacité, rendement et satisfaction dans un contexte d'utilisation donné ». Dans le cadre de cette définition, l'efficacité se rapporte aux « mesures de la précision et de la complétude des buts atteints » et le rendement aux « mesures de la précision et de la complétude des buts atteints par rapport aux ressources utilisées pour les atteindre ». Ainsi, avec l'utilité, l'utilisabilité est une mesure qui permet de caractériser l'intérêt (useworthiness) d'un système interactif pour accomplir une tâche spécifique dans un contexte d'interaction précis. [Eftring99] complète ces deux grandeurs par une troisième qui mesure la capacité d'un système interactif à répondre aux besoins très prioritaires de l'utilisateur. Dans une approche centrée sur la valeur [Cockton05], l'intérêt (useworthiness) exprime la valeur qu'un utilisateur, dans un contexte d'interaction donné, attend du système ou lui attribue.

Or, l'informatique ubiquitaire engendre des variations à la fois du contexte de l'interaction et des besoins prioritaires de l'utilisateur. La plasticité vise à préserver la valeur du système interactif malgré ces variations [Dâassi06]. Ainsi, la question de la plasticité dépasse celle de la portabilité ou de la traduction de l'interface utilisateur entre des dispositifs numériques de nature différente. Puisque l'utilisateur est mobile, le système interactif nécessite une adaptation bien plus profonde, éventuellement à tous ses niveaux d'abstraction, afin de prendre en compte, outre les changements qui interviennent dans la plate-forme, l'évolution des besoins et des capacités physiques et cognitives de l'utilisateur au cours de son interaction.

Ce chapitre a pour objectif de cerner l'espace problème de la plasticité à l'exécution du point de vue de l'ingénierie de l'interaction homme-machine. Il s'organise en trois parties. La première expose et analyse un scénario-type d'interaction entre un utilisateur et un système interactif plastique. Ce scénario, qui montre par l'exemple différents cas d'adaptation au contexte de l'interaction, sert d'illustration dans les sections suivantes. Dans la deuxième partie, j'étudie et analyse plusieurs taxonomies existantes portant sur l'adaptation des IHM pour proposer dans la dernière partie un espace problème dont la finalité est triple : comprendre la portée du problème de la plasticité des IHM à l'exécution, évaluer et comparer les solutions techniques actuelles, et identifier de manière rationnelle les lacunes de l'état de l'art.

1. Scénario-type et analyse

Afin d'introduire l'espace problème de la plasticité à l'exécution, voici un scénario-type inspiré du projet européen CAMELEON[3] (auquel j'ai contribué), et du projet européen IST GLOSS[4].

1.1 Scénario-type

Bertrand est un chercheur en informatique qui doit se rendre à Grenoble pour présenter son travail lors d'un atelier organisé par le laboratoire LIG. Arrivé en gare de Grenoble, Bertrand décide de commencer par trouver son hôtel pour y déposer sa valise puis de se mettre en quête d'un restaurant pour déjeuner. Il s'approche d'une carte interactive située dans le hall de la gare (voir Figure II-2).

[3] http://giove.isti.cnr.it/cameleon.html

[4] http://iihm.imag.fr/projects/Gloss/index.html

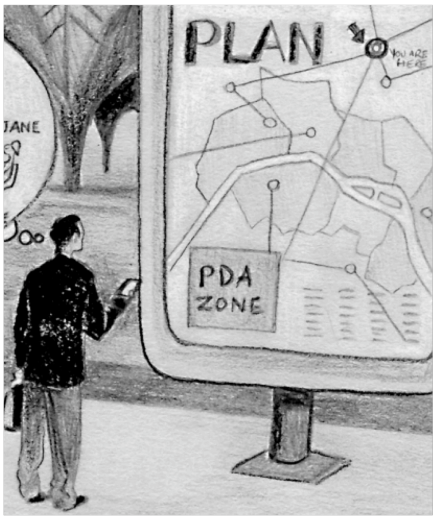


Figure II-2 La carte interactive est un grand écran plat qui affiche le plan général de la ville.

À partir du moment où Bertrand se trouve face à la carte interactive, cette dernière détecte le smartphone présent dans l'une de ses poches. Celui-ci vibre, Bertrand le saisit et voit, sur l'écran du smartphone, une invitation à utiliser les services offerts par la carte interactive. Il accepte. L'écran du smartphone affiche alors un panneau de contrôle pour la carte interactive. Sur ce panneau, Bertrand choisit « Afficher un itinéraire ». Comme l'agenda de Bertrand contient les coordonnées de son hôtel, l'interface utilisateur lui propose son hôtel comme choix de destination présélectionnée. Il valide ce choix. Sur la carte interactive, le plan de la ville est recadré afin d'afficher en détail la partie de la carte contenant à la fois la gare et l'hôtel. L'itinéraire est indiqué en surbrillance et une information textuelle indique que l'hôtel est accessible à pied depuis la gare en moins de 10 minutes.

Sur son smartphone, Bertrand choisit « Afficher les restaurants sur l'itinéraire ». En réaction, la carte interactive fait apparaître des icônes symbolisant les restaurants les plus proches du parcours. Pour chaque restaurant, une information textuelle donne davantage de détails. Satisfait par l'itinéraire proposé, Bertrand décide de le rapatrier sur son smartphone. Etant donné que le smartphone est équipé d'un récepteur GPS[5], l'itinéraire est transmis dans un format adapté. Ainsi, au moyen de son oreillette Bluetooth, Bertrand pourra être guidé oralement par le GPS sur son parcours.

[5] GPS : Global Positioning System : Actuellement le seul système de géolocalisation par satellite. Un récepteur GPS est capable de déterminer sa position, sa vitesse et sa direction dans le système de coordonnées géographiques (latitude, longitude, niveau de la mer).

Le lendemain, Bertrand se tient dans la salle de l'atelier et se met en place pour sa présentation. Il allume son ordinateur portable et lance son logiciel de présentation. Au démarrage du logiciel, le système détecte le projecteur vidéo qui équipe la salle. Le logiciel propose à Bertrand, qui accepte, d'utiliser le projecteur pour afficher les diapositives et aussi de réserver l'écran de l'ordinateur portable pour l'affichage des notes personnelles et du chronomètre. Bertrand commence sa présentation en se tenant près de son ordinateur et en utilisant le clavier pour passer d'une diapositive à une autre.

Comme il a besoin de plus de mobilité pendant son discours, Bertrand allume son PDA[6]. Ce nouveau dispositif est détecté par le système. Etant donné que le logiciel de présentation sait que la tâche prioritaire de Bertrand est de contrôler le défilement des diapositives et que le PDA offre toutes les ressources d'interaction nécessaires à cette tâche, il propose à Bertrand, par l'intermédiaire du message affiché sur le PDA, de l'utiliser comme télécommande. Bertrand accepte et l'interface utilisateur d'une télécommande apparaît sur l'écran du PDA. Cette interface utilisateur présente des boutons « suivant » et « précédent » suffisamment gros pour être facilement actionnables par un doigt en situation de mobilité. À présent, Bertrand contrôle la présentation avec son PDA.

Au cours de la présentation, le temps jusque-là couvert, se dégage. La luminosité dans la salle de l'atelier augmente et les diapositives deviennent illisibles. Automatiquement, le logiciel de présentation change le mode de rendu des diapositives pour les rendre plus contrastées et pour résoudre le désagrément. Mais cette solution ne satisfait pas Bertrand qui souhaite diffuser une vidéo sur la diapositive suivante. Comme la salle de l'atelier est équipée d'un système de volets motorisés, et que gérer la luminosité de la salle est une tâche prise en compte par le logiciel de présentation, la télécommande affichée sur l'écran du PDA intègre un interacteur pour contrôler les volets de la salle. Bertrand est alors en mesure de baisser les volets, la luminosité de la salle chute et le logiciel de présentation affiche de nouveau les diapositives normalement.

1.2 Analyse du scénario

Ce scénario est caractéristique de ce que pourrait être l'interaction entre un utilisateur et deux *systèmes interactifs*[7] à l'ère de

[6] PDA : Personal Digital Assistant : Ordinateur de poche, initialement prévu pour la gestion de données personnelles (agenda, répertoire, notes, etc.), il est actuellement bien plus polyvalent et permet de se connecter à Internet, de jouer, d'écouter de la musique et de visualiser des vidéos.

[7] Dans le cadre de cette thèse, j'appelle *système interactif* l'association d'un ensemble de services numériques et d'une interface utilisateur. Cette

l'informatique ubiquitaire. Première caractéristique, Bertrand utilise des services numériques offerts par son environnement d'une manière opportuniste. Il ne planifie pas à l'avance l'interaction avec un service numérique en particulier, mais se concentre sur les tâches qu'il doit effectuer, par exemple, trouver son hôtel et un restaurant pour déjeuner, projeter ses diapositives et les faire défiler. Pour atteindre ses objectifs, il s'aide des services numériques adéquats, disponibles là où il se trouve. Cependant, il n'a jamais besoin de les rechercher : ces services sont découverts automatiquement par les dispositifs numériques qui assistent Bertrand dans ses tâches, par exemple son smartphone lorsqu'il se déplace ou son ordinateur portable et son PDA lors de sa présentation.

Avec la carte interactive, Bertrand ne fait qu'interagir avec un système interactif donné. En revanche, au moment de sa présentation, en associant son ordinateur portable au projecteur vidéo et son PDA, ou en agissant sur les volets motorisés qui équipent les fenêtres, Bertrand fait plus qu'utiliser des services numériques découverts automatiquement : il façonne son espace interactif.

Dans mon approche, *l'espace interactif constitue le repère dans lequel se situe un utilisateur* lorsqu'il interagit avec un ensemble de services numériques. Je définis ce repère par trois dimensions : (1) le lieu physique où se déroule l'interaction, c'est-à-dire l'environnement physique et social, (2) l'ensemble des ressources de calcul, de réseau et d'interaction disponibles en ce lieu, c'est-à-dire la *plate-forme*[8] et enfin (3) le monde numérique, c'est-à-dire l'ensemble des systèmes interactifs capables d'assister l'activité humaine à cet endroit. Ainsi, pour l'utilisateur, l'espace interactif - qui sert à situer l'utilisateur - correspond au contexte de

association est classique depuis [Uims92] qui définissait : « Une application interactive consiste en un logiciel pour un domaine d'application et un logiciel d'interface utilisateur ». En termes du modèle Arch [Uims92], les services numériques correspondent au noyau fonctionnel et l'interface utilisateur comprend toutes les autres fonctions.

[8] Dans le cadre de cette thèse, j'appelle *plate-forme* un ensemble interconnecté de plates-formes élémentaires. Une *plate-forme élémentaire* est l'association au sein d'un dispositif physique indivisible, d'un ensemble de ressources de calcul et d'un ensemble de ressources d'interaction. Par exemple, un smartphone, un PDA, un ordinateur portable ou un ordinateur de bureau sont des plates-formes élémentaires. Une plate-forme peut être homogène, c'est-à-dire composée d'un ensemble de plates-formes élémentaires identiques, par exemple deux ordinateurs portables avec le même système d'exploitation. Une plate-forme peut être hétérogène, c'est-à-dire composée de plates-formes élémentaires différentes, par exemple un ordinateur portable et un PDA ou encore, un Macintosh et un PC.

l'interaction - qui, lui, permet de situer le système interactif (voir Figure II-3).

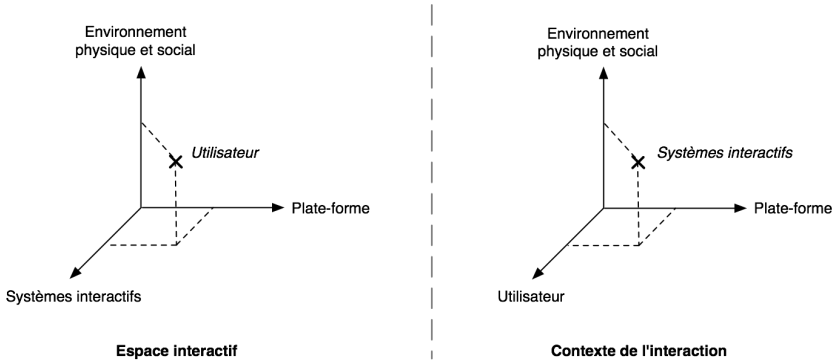


Figure II-3 Espace interactif et contexte de l'interaction.

En d'autres termes, Bertrand façonne son espace interactif en constituant dynamiquement une plate-forme et en distribuant l'interface utilisateur du système interactif sur l'ensemble de la plate-forme. Le système interactif est attentif à l'environnement physique où se déroule l'interaction et adapte son interface utilisateur (en basculant dans un mode plus contrasté) pour préserver au mieux l'utilisabilité (les diapositives doivent être lisibles par le public). Si les systèmes interactifs sont capables d'auto-adaptation, par exemple en prenant seuls l'initiative de basculer dans un mode plus contrasté, ou de proposer des services, par exemple en permettant le contrôle de la carte interactive, Bertrand garde en permanence le contrôle sur le système et peut décider d'utiliser, ou non, le service proposé ou d'accepter, ou non, l'adaptation calculée automatiquement. En somme, Bertrand peut déclencher lui-même une adaptation soit de manière implicite, par exemple en allumant son PDA qui devient détectable par le système, soit de manière explicite, par exemple en fermant les volets de la pièce.

Plus que de montrer ce que les utilisateurs ou les systèmes interactifs deviennent capables de faire, ce scénario met en scène un nouveau genre d'interfaces utilisateur. Celles-ci ne sont plus confinées au sein d'une plate-forme élémentaire, mais peuvent être distribuées sur toute ou partie d'une plate-forme. Par exemple, le panneau de contrôle de la carte interactive s'exécute sur le PDA de Bertrand alors que la carte elle-même et les informations associées sont affichées sur le grand écran public. L'interface utilisateur peut être répartie dès le démarrage d'un système interactif, mais elle peut également l'être dynamiquement au cours de l'interaction avec l'utilisateur. Par exemple, au cours de la présentation, Bertrand a décidé de migrer sur le PDA le contrôle des diapositives. Cette redistribution de l'interface utilisateur s'est faite sans arrêter le système interactif et Bertrand a

pu continuer sa tâche sans avoir à recommencer depuis la première diapositive.

La plupart du temps, la redistribution implique un remodelage de l'interface utilisateur. Le remodelage est une transformation de l'interface utilisateur conservant l'unité de distribution, dans l'objectif de l'adapter à une situation nouvelle. Par exemple, au début de la présentation, l'interface utilisateur du contrôleur de diapositives se limite à deux touches du clavier de l'ordinateur portable et aucun interacteur dédié à cette tâche n'est visible à l'écran. Lorsque Bertrand redistribue l'interface utilisateur pour être en mesure de contrôler la présentation avec son PDA, le contrôleur de diapositives est migré sur le PDA. Son interface utilisateur est alors transformée pour s'y adapter : elle se matérialise par des interacteurs sur l'écran du PDA et offre des fonctionnalités supplémentaires, comme le contrôle des volets de la pièce. Ainsi, redistribution et remodelage constituent les deux leviers pour adapter les interfaces utilisateur. Dans la section suivante, j'étudie et analyse les différentes taxonomies sur lesquelles je m'appuie pour établir les dimensions de mon espace problème.

2. Des taxonomies préexistantes pour la plasticité

Pour construire mon espace problème, je prends comme point de départ les travaux de [Thevenin01] qui fondent la problématique de la plasticité des systèmes interactifs. Si mon sujet de thèse concerne les phases d'exécution des systèmes, ceux de [Thevenin01] sont résolument centrés sur les phases de conception. Dans ses travaux, il s'agissait de proposer un cadre conceptuel et des outils pour le développement de systèmes interactifs multicibles. [Thevenin01] construit un espace problème en deux parties: la première offre un point de vue système sur les IHM multicibles, et la seconde structure l'espace des outils impliqués dans la production d'IHM multicibles.

2.1 Une taxonomie préexistante pour les interfaces utilisateur multicibles

La première de ces deux parties concerne directement mes travaux de thèse : elle permet de poser les bases de ma réflexion. [Thevenin01] construit cette première partie consacrée au point de vue système en adoptant comme référence structurelle le modèle Arch [uims92] et en s'appuyant sur les axes centrés « système » d'un éventail de taxonomies préexistantes. Ainsi, il articule sa taxonomie autour de deux axes : les constituants logiciels adaptés et les instants d'adaptation. Dans son approche, les constituants

logiciels correspondent aux différents niveaux du modèle Arch. [Thevenin01] motive ce choix par le fait que le modèle Arch est une référence dans la communauté de l'ingénierie de l'interaction homme-machine. Ainsi, il constitue une base commune d'analyse. Son association aux constituants logiciels adaptés permet de jeter un pont entre l'adaptation et l'architecture logicielle.

Selon cet axe, l'adaptation de l'IHM peut avoir lieu à tous les niveaux du modèle Arch. Une adaptation de la présentation physique (PP) conserve la nature des interacteurs physiques, mais leur rendu peut-être différent d'une plate-forme élémentaire à l'autre. L'adaptation de la présentation logique (PL) change la nature des interacteurs mais pas leurs capacités représentationnelles et fonctionnelles. Ainsi, ce type d'adaptation peut ajouter ou supprimer des tâches articulatoires, mais ne modifie pas le contrôleur de dialogue. L'adaptation au niveau contrôleur de dialogue (CD) change l'ordonnancement des tâches mais pas leur nature. En revanche, l'adaptation au niveau de l'adaptateur de noyau fonctionnel (ANF), conséquence du changement de nature des concepts et des fonctions exportées par le noyau fonctionnel, change la nature des tâches et des concepts qu'elles manipulent. Une fois les différents constituants logiciels adaptables définis, [Thevenin01] décrit les moments clés, d'un point de vue système, où l'adaptation peut avoir lieu.

Trois moments clés sont identifiés : le développement, l'exécution et l'installation. L'adaptation au développement est réalisée par les concepteurs et développeurs du système interactif. Ce moment d'adaptation impose de prévoir dès la conception l'ensemble des cas d'adaptation. L'adaptation à l'exécution permet une adaptation de l'IHM à la volée, sans qu'il soit nécessaire d'arrêter l'exécution du système interactif. Enfin, l'adaptation à l'installation consiste à adapter ou à configurer le logiciel pour une cible donnée lors de l'installation. Ces trois instants d'adaptation conduisent [Thevenin01] à distinguer trois classes d'IHM exécutables : les IHM précalculées, les IHM calculées dynamiquement et les IHM hybrides.

Les IHM multicibles précalculées sont le résultat d'adaptation réalisées avant l'exécution du logiciel, soit lors de son développement, soit à son installation. Ces IHM sont en mesure de gérer un ensemble de plates-formes cibles identifiées par avance. Les IHM multicibles calculées dynamiquement s'adaptent à la volée, au cours de l'exécution, suite à la réception d'un événement ou d'une condition d'adaptation qui devient vraie. Enfin, les IHM multicibles hybrides sont une combinaison des deux approches précédentes : ces IHM peuvent être précalculées pour une classe donnée de plates-formes élémentaires, par exemple la classe des PC, et se transforment à la volée pour s'adapter à la plate-forme élémentaire effective, par exemple un PalmPilot ou un Pocket PC.

2.2 Des taxonomies pour systèmes adaptables

D'autres taxonomies, sur lesquelles [Thevenin01] s'appuie dans sa thèse, sont plus précisément axées sur l'adaptation à l'exécution. Notamment [Totterdell90] définit six niveaux de sophistication croissante pour les mécanismes d'adaptation à l'exécution. Le niveau le plus bas constitue les *systèmes câblés* dont le comportement est fixé à la conception et ne permet aucune adaptation à l'exécution. Ensuite, les *systèmes adaptables* sont personnalisables par l'utilisateur dans la limite des degrés de liberté définis par le concepteur. Plus perfectionnés, les *systèmes adaptatifs* sont capables de reconnaître des situations prédéterminées par le concepteur et réagissent en fonction de recommandations spécifiées par le concepteur. Les *systèmes autorégulateurs* ont les mêmes propriétés que les systèmes adaptatifs mais sont en plus capables d'évaluer, a posteriori, l'effet de leur réaction. Le niveau de sophistication supérieur regroupe les *systèmes automédiateurs*, qui, en plus des capacités des systèmes autorégulateurs, peuvent évaluer a priori l'effet de leur réaction. Enfin, les *systèmes automodificateurs* peuplent le niveau le plus abouti de cette classification. Ces systèmes ont la capacité de reconnaître et d'apprendre de nouvelles situations, et de définir de nouvelles réactions.

Tableau II-A Taxonomie des systèmes adaptatifs de [Totterdell90]

Niveau des mécanismes d'adaptation	Identification des déclencheurs	Choix des réactions	Evaluation des réaction
Câblé	Concepteur	Concepteur	Concepteur
Adaptable	Concepteur	Utilisateur	Concepteur
Adaptatif	Concepteur	Système	Concepteur
Autorégulateur	Concepteur	Système	Système
Automédiateur	Concepteur	Système	Système
Automodificateur	Système	Système	Système

Le Tableau II-A synthétise la taxonomie de [Totterdell90] et explicite les rôles des différents acteurs : le concepteur, l'utilisateur et le système. De manière complémentaire, [Dieterich93] découpe le processus d'adaptation en quatre étapes successives : initiative, suggestion, décision et exécution. Chez [Dieterich93], chaque étape peut être réalisée soit par l'utilisateur soit par le système de manière autonome. Des seize combinaisons possibles, [Dieterich93] extrait six combinaisons qu'il juge les plus intéressantes (voir Figure II-4). Ces six combinaisons déterminent six classes d'adaptation qui s'inscrivent dans un gradient allant de l'adaptation totalement mise en œuvre par l'utilisateur jusqu'à l'adaptation totalement automatique.

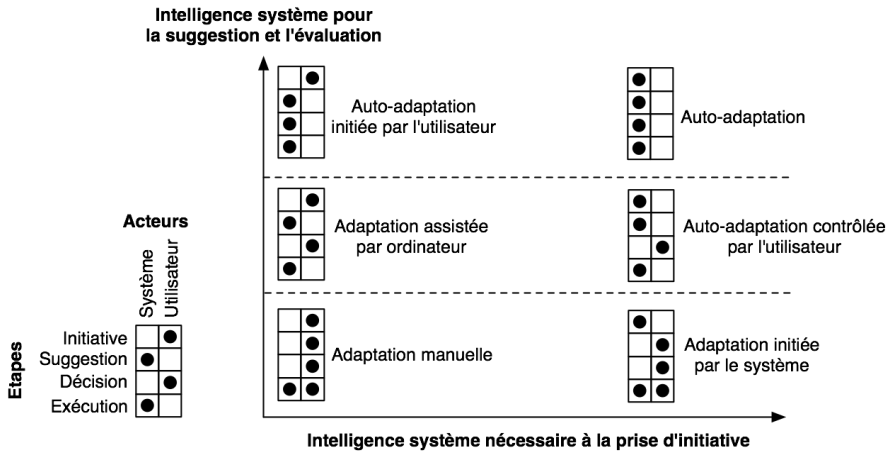


Figure II-4 Classification de [Dieterich93] : les six combinaisons les plus dignes d'intérêt.

Entre ces deux extrêmes, se situent les quatre autres classes d'adaptation. La première classe est celle de l'adaptation initiée par le système où le système est capable d'informer l'utilisateur de l'opportunité de réaliser une adaptation, l'utilisateur étant chargé de la mettre en œuvre. La deuxième classe concerne l'adaptation assistée par l'ordinateur où l'utilisateur est à l'initiative du processus et décide de l'adaptation parmi les propositions établies par le système. La troisième classe a trait à l'adaptation contrôlée par l'utilisateur où le système prend l'initiative de l'adaptation et propose à l'utilisateur un ensemble de solutions parmi lesquelles il devra faire son choix. Enfin, la quatrième classe est celle de l'auto-adaptation initiée par l'utilisateur où l'utilisateur voit son rôle réduit à celui de l'initiateur de l'adaptation, le reste du processus étant entièrement conduit par le système.

2.3 Des taxonomies pour l'informatique ubiquitaire

Parmi les travaux plus récents, on trouve dans la littérature quelques taxonomies à propos des systèmes d'information pour l'informatique ubiquitaire. Notamment [Pousman06] recense et analyse un ensemble de travaux réalisés entre les années 2001 et 2004 sur les systèmes ambiants. Cependant, ces taxonomies s'intéressent plutôt à classer les systèmes interactifs pour l'informatique ubiquitaire en termes de types d'interaction homme-machine proposées à l'utilisateur, plutôt qu'en termes d'adaptation au contexte de l'interaction. En guise de synthèse des travaux qu'il recense, [Pousman06] propose une taxonomie en quatre axes : la capacité à transmettre de l'information à l'utilisateur, le niveau de notification, la fidélité dans la représentation et l'attractivité de l'IHM. Ces quatre axes sont tous gradués en cinq niveaux : bas, plutôt bas, moyen, plutôt haut et haut.

Dans l'approche de [Pousman06] le premier axe, qui représente la capacité à transmettre l'information, caractérise le nombre de sources d'information discrètes que le système peut présenter à l'utilisateur. Par exemple, certains systèmes sont capables de n'afficher que le prix d'un élément en stock, alors que d'autres peuvent afficher des valeurs pour plus de vingt informations différentes sur le même écran. Le deuxième axe, qui représente le niveau de notification, qualifie la manière avec laquelle un système interactif alerte un utilisateur. Certains peuvent être discrets et non-intrusifs, alors que d'autres peuvent aller jusqu'à interrompre l'utilisateur dans son activité courante. Le troisième axe, qui mesure la fidélité dans la représentation, décrit de quelle manière un système interactif transmet des informations à l'utilisateur. Par exemple, certains systèmes interactifs reproduisent à l'utilisateur l'information de manière très directe telle l'image vidéo d'une personne distante, alors que d'autres utilisent des représentations plus abstraites comme le niveau de disponibilité de cette personne. Le dernier axe traite de l'importance accordée par les concepteurs du système à l'attractivité de l'interface utilisateur. Pour certains systèmes, un objectif prioritaire peut être que la perception de l'IHM par l'utilisateur soit plaisante. D'autres attacheront moins d'importance à l'attractivité de l'IHM pour se concentrer sur la capacité à communiquer l'information.

En analysant près d'une vingtaine de systèmes interactifs au regard de ces quatre axes, [Pousman06] extrait quatre patrons de conception : l'afficheur symbolique sculptural (représente peu d'information, mais de manière symbolique par des objets physiques à vocation décorative), le consolidateur de sources d'information (intègre quelques sources d'information, souvent de manière esthétique et non intrusive), le moniteur d'informations (tels les écrans annexes d'une station de travail), et l'afficheur à forte densité informationnelle.

Les travaux et taxonomies analysés dans cette section offrent des éléments utiles à la structuration de l'espace problème de la plasticité à l'exécution. J'analyse maintenant comment ils peuvent s'appliquer à mon sujet d'étude.

2.4 Analyse des taxonomies précédentes

L'axe des constituants logiciels adaptés dans la taxonomie de [Thevenin01] reste valide pour la problématique qui m'occupe. Dans son deuxième axe, qui concerne les instants de l'adaptation, [Thevenin01] se situe au niveau de l'adaptation au développement et de l'adaptation à l'installation. En revanche, mes travaux de thèse se situent au niveau de l'adaptation à l'exécution. Si ces deux axes restent pertinents dans le cadre de la plasticité à l'exécution des systèmes interactifs, les classes d'IHM exécutables doivent être revues. En effet, ces trois classes d'IHM

exécutables (précalculées, dynamiques et hybrides) ne semblent valides que si l'on considère deux hypothèses : d'une part, que le seul moyen d'adaptation de l'interface utilisateur consiste en son remodelage et, d'autre part, que le code du système interactif soit constitué d'un seul composant.

En considérant ces deux hypothèses, si le contexte de l'interaction rencontré n'a pas été prévu lors de la conception, comme le constate [Thevenin01], seule une génération à la volée d'une nouvelle interface utilisateur peut permettre d'adapter le système interactif. Or, une de mes hypothèses de travail concerne la distributivité de l'interface utilisateur sur un ensemble de plates-formes élémentaires. Cette hypothèse supplémentaire a deux conséquences. D'une part, elle fait apparaître un autre moyen d'adaptation : la redistribution de l'IHM. D'autre part, si l'on écarte la solution d'un intergiciel unique masquant la distribution au code d'un système interactif, elle remet en question le développement monolithique du code. Dans ce cas de figure, pour être distribuable, un système interactif doit être constitué de différents éléments logiciels interconnectables.

Du fait de la grande hétérogénéité potentielle de la plate-forme, comme nous le verrons au chapitre suivant, du foisonnement d'intergiciels consacrés à la question, il semble raisonnable d'écarter la solution d'un intergiciel unique pour masquer la distribution ou l'hétérogénéité de la plate-forme. En conséquence, la propriété de plasticité d'un système interactif impose une structure modulaire du code, où les différentes entités logicielles mises en jeu devront avoir la capacité d'être connectées ou déconnectées à l'exécution. Ainsi, une adaptation calculée dynamiquement n'implique pas forcément une génération à la volée d'une IHM au sens où l'entend [Thevenin01], mais peut résulter d'une nouvelle configuration d'entités logicielles, chacune mettant en œuvre un morceau d'IHM précalculée couplé éventuellement à des morceaux d'IHM générées à la volée.

De la même manière, la taxonomie proposée par [Totterdell90] ne convient plus. Dans le cas de la plasticité à l'exécution, la part belle est faite à l'utilisateur, qui voit ses prérogatives bien étendues. Ainsi, si le système peut prendre des initiatives quant à son adaptation, l'utilisateur doit également être en mesure de pouvoir initier une adaptation et de contrôler le déroulement de toute adaptation, afin de façonner l'espace interactif selon ses préférences. Dans la taxonomie de [Totterdell90], l'utilisateur semble relégué au second plan. Seuls les systèmes adaptables de cette taxonomie accordent à l'utilisateur un pouvoir de décision sur l'adaptation. En revanche, aucun ne lui permet d'être à l'initiative. Ainsi, il n'est pas possible de classer dans cette taxonomie les systèmes interactifs plastiques.

Au contraire de [Totterdell90], la taxonomie de [Dieterich93] reste totalement valide pour la plasticité à l'exécution. Notamment, les classes d'adaptation « adaptation assistée par ordinateur », « auto-adaptation contrôlée par l'utilisateur » et « auto-adaptation initiée par l'utilisateur » sont particulièrement pertinentes. Si la classe « auto-adaptation » reste intéressante pour certaines situations où la charge de l'utilisateur est importante, elle ne représente qu'un ensemble marginal de cas d'adaptation dans ma problématique.

Enfin, la taxonomie de [Pousman06] ne prend pas du tout en compte la question de l'adaptation des systèmes interactifs. Inversement, chaque patron de conception repéré par [Pousman06] peut être utilisé pour concevoir et adapter un système interactif plastique. Dans la section suivante, je présente une taxonomie qui décrit l'espace problème de la plasticité à l'exécution. Cette taxonomie est reprise par Calvary qui l'intègre dans une vision plus globale centrée sur la notion de directive d'adaptation non pas sur la préoccupation logicielle [Calvary07].

3. Classification et plasticité à l'exécution

Mon espace problème se caractérise par sept dimensions principales (voir Figure II-5, partie droite). La première d'entre elles détermine les moyens d'adaptation : remodelage et redistribution. La dimension suivante mesure la granularité de la plus petite unité logicielle qui peut-être adaptée par l'un ou l'autre des deux moyens précédents. Cette granularité varie du plus gros, le système interactif entier, au plus fin, l'interacteur.

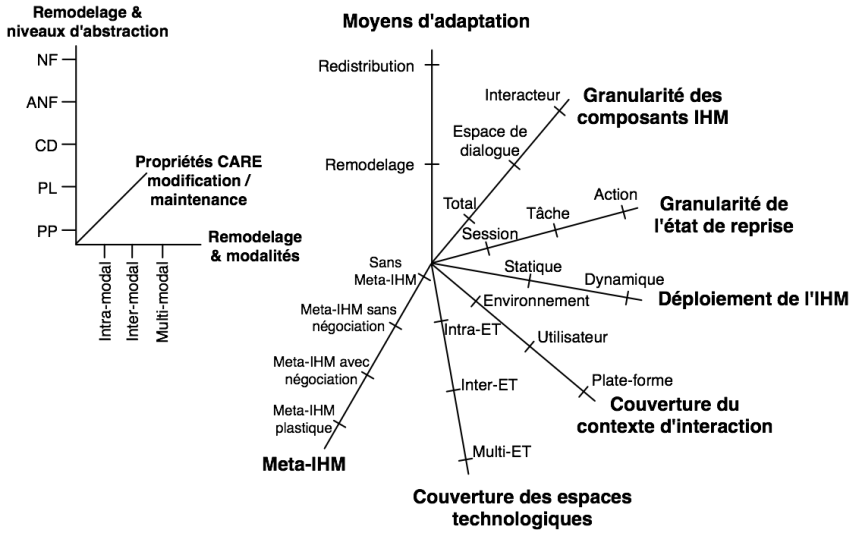
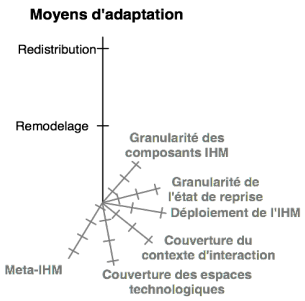


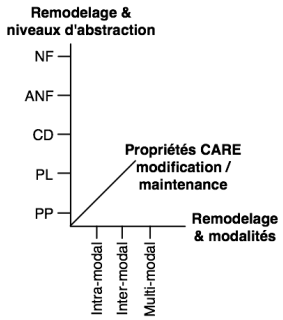
Figure II-5 Espace problème de la plasticité à l'exécution.

La troisième dimension spécifie la granularité de l'état de reprise sur adaptation, qui s'étend du niveau « session » au niveau « action ». La dimension intitulée « déploiement de l'IHM » précise le moment où l'adaptation a lieu, soit à la conception ou à l'installation (déploiement statique), soit à l'exécution (déploiement dynamique). La cinquième dimension utilise la couverture du contexte de l'interaction pour définir les causes d'adaptation que le système est capable de prendre en compte. L'avant-dernière dimension caractérise le degré d'hétérogénéité technologique auquel le système est capable de s'accommoder. Enfin, la dimension « Méta-IHM » détermine le niveau de contrôle dont dispose l'utilisateur sur le processus d'adaptation du système. Je développe maintenant chacun de ces axes.

3.1 Deux moyens d'adaptation : remodelage et redistribution



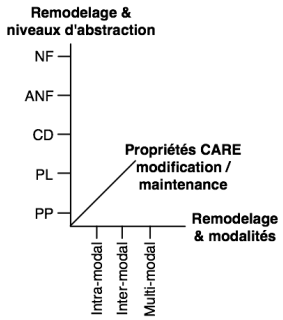
Le remodelage traduit une reconfiguration d'IHM qui est perceptible par l'utilisateur. Cette reconfiguration résulte de l'application de transformations sur cette interface. Ces transformations peuvent être appliquées à deux niveaux : soit au niveau des entités logicielles qui composent l'IHM, soit à un niveau interne à ces entités logicielles. Par remodelage au niveau « entités logicielles », j'entends l'ajout, et/ou la suppression d'entités logicielles et/ou le remaniement de leurs interconnexions. Par exemple, un remodelage de ce type peut consister à supprimer des entités logicielles IHM qui ne seraient plus adéquates dans le nouveau contexte d'interaction et d'en instancier de nouvelles, soit pour remplacer les entités supprimées, soit pour donner accès à de nouveaux services. Par remodelage interne à une entité logicielle, j'entends une transformation qui conserve la cardinalité des entités logicielles en présence ainsi que leurs interconnexions. Cependant, une évolution de l'interface utilisateur résulte d'opérations internes à tout ou partie de ces entités logicielles.



Indépendamment de ces deux niveaux d'action, la portée d'un remodelage se décrit dans un espace tridimensionnel (voir Figure II-5, partie gauche). Le premier axe de cet espace concerne les niveaux d'abstraction qui peuvent être affectés par un remodelage, les deux autres axes ont trait à la multimodalité. Le premier axe est directement inspiré de l'axe des constituants adaptés de la taxonomie de [Thevenin01]. De la même façon que dans [Thevenin01], une adaptation peut concerner tout aussi bien un simple changement d'apparence des interacteurs, un remplacement d'interacteurs par d'autres fonctionnellement équivalents, une réorganisation spatiale ou temporelle des tâches, et même l'introduction de nouvelles tâches ou le retrait d'autres qu'il devient impossible d'assurer. En conséquence, cet axe, comme celui de [Thevenin01], indique qu'un remodelage peut

avoir un impact sur tout ou partie des niveaux d'abstraction du modèle Arch [Uims92].

Le deuxième axe aborde le remodelage sous l'angle des modalités d'interaction[9]. Un remodelage peut être intramodal, intermodal ou multimodal. Il est intramodal lorsque l'interface utilisateur source, qui a besoin d'être remodelée, subit une transformation qui conserve la modalité d'interaction. Le remodelage est intermodal lorsque l'interface utilisateur source est transformée en une interface utilisateur exploitant une modalité d'interaction différente. Ainsi, un remodelage intermodal peut engendrer un gain ou une perte de modalité. Par exemple, une interface utilisateur multimodale peut être transformée en une interface utilisateur monomodale, ou inversement, une IHM monomodale peut être remodelée en une IHM multimodale. Enfin, un remodelage multimodal est une combinaison au sein d'une même transformation de remodelage intramodal et intermodal. Un exemple d'une telle transformation est décrit dans [Berti05].



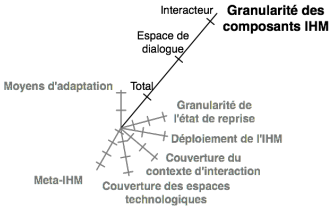
Le troisième axe caractérise l'impact d'un remodelage sur les propriétés CARE (Complémentarité, Assignation, Redondance, Equivalence) [Coutaz95] qui caractérisent la manière dont les modalités d'interaction sont exploitées. Un remodelage peut soit maintenir soit modifier ces propriétés. Par exemple, parce que la puissance de calcul disponible chute, une IHM multimodale qui utilisait une complémentarité synergique, comme dans l'exemple « put-that-there », subit un remodelage qui conserve les modalités d'interaction, mais qui n'autorise plus qu'une complémentarité alternée : l'utilisateur devant utiliser les différentes modalités de manière séquentielle.

Si l'adaptation de l'IHM est généralement assimilée au remodelage, ceci n'est vrai que si l'on considère que l'interaction ne peut avoir lieu que sur une seule plate-forme élémentaire à la fois. Or, en informatique ubiquitaire, la plate-forme est un assemblage dynamique et opportuniste de plates-formes élémentaires interconnectées dont les ressources d'interaction forment ensemble un *écosystème* dans lequel évoluent les systèmes interactifs. Dans ce type de configuration, l'interface utilisateur n'est plus *centralisée*, mais peut-être *distribuée* sur l'ensemble des ressources d'interaction de la plate-forme. Ainsi, la redistribution repose sur la réallocation des entités logicielles IHM du système interactif sur tout ou partie des plates-formes élémentaires qui composent la plate-forme. Par exemple, lorsque le PDA de Bertrand est détecté par le logiciel de présentation, le contrôleur de diapositives migre dynamiquement de l'ordinateur

[9] Une modalité d'interaction se définit par le couple (langage, dispositif) au sens de [Nigay94].

portable vers l'écran du PDA. Il est à noter comme conséquence d'une redistribution de l'interface utilisateur, que tout ou partie des éléments de l'IHM devront éventuellement être remodelés pour être adaptés aux ressources d'interaction disponibles sur la plate-forme élémentaire d'accueil. Je précise dans la section suivante la nature des entités logicielles pouvant être adaptées par le biais du remodelage et de la redistribution, ainsi que les moments où cette adaptation peut avoir lieu.

3.2 Granularité des composants d'IHM



Dans cette section, le terme composant est à prendre dans son sens général. Il ne fait donc pas référence à la notion de composant telle qu'elle existe en génie logiciel. Dans cette taxonomie pour la plasticité à l'exécution, je distingue trois niveaux de granularité sur lesquels le remodelage et la redistribution peuvent porter : l'ensemble du système interactif, l'espace de dialogue et l'interacteur.

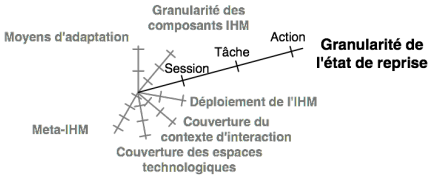
Le grain le plus gros, le niveau « total », est celui du système interactif. À ce niveau, l'adaptation s'applique au système interactif en totalité. Cela correspond au scénario-type dans le projet Aura [Sousa03] : l'utilisateur édite un texte chez lui avec Word. Il doit se rendre au bureau. Lorsqu'il arrive devant sa station de travail, le système Aura lui propose de poursuivre sa tâche d'édition de document. Comme sa station de travail est sous Unix, Aura remplace Word par Emacs. Du point de vue de l'utilisateur, l'IHM de l'éditeur de texte a subi une migration de l'ordinateur personnel de l'utilisateur à sa station de travail professionnelle et a été entièrement remodelé pour s'adapter à la nouvelle plate-forme utilisateur. Comme, à ce niveau de granularité, le système interactif est manipulé comme une entité indivisible, la redistribution n'est possible que sous la forme d'une migration totale d'une plate-forme élémentaire vers une autre.

Le niveau de granularité inférieur est celui de l'espace de dialogue. L'espace de dialogue est une notion proche de l'espace de travail de [Normand92]. Cependant, la notion d'espace de travail est très ancrée dans l'univers des interfaces graphiques. Comme la multimodalité semble devenir la règle en informatique ubiquitaire, je lui préfère la notion d'espace de dialogue, définie dans [Emode06], qui élargit cette notion d'espace de travail en incluant les micro-dialogues de clarification et de négociation induits par toute modalité d'interaction. Par exemple, pour une tâche de destruction de fichier, une boîte de dialogue de confirmation est incluse dans l'espace de dialogue correspondant à cette tâche. Il en va de même dans un milieu bruité, lorsque la « voix » du système demande à l'utilisateur de répéter ou de lever une ambiguïté. Au niveau de granularité « espace de dialogue », le

système interactif est remodelé ou redistribué selon les tâches abstraites[10] de l'utilisateur. Par exemple, lorsque le mur interactif détecte le smartphone de Bertrand, ce dernier est utilisé pour accueillir l'ensemble de l'interface utilisateur relative à la tâche abstraite « manipuler la carte ». Lors de sa présentation, la tâche abstraite « contrôler les diapositives » migre de l'ordinateur portable au PDA.

Enfin, le niveau de granularité des composants IHM le plus fin est celui de l'interacteur. À ce niveau, l'adaptation se fait à l'échelle de la tâche élémentaire. Par exemple, dans PlasticClock [Calvary05], lorsque la surface de l'écran disponible n'est plus assez grande pour afficher la montre sous la forme d'un cadran à aiguille, l'interacteur « cadran » est remplacé par un interacteur capable d'afficher l'heure de manière textuelle.

3.3 Granularité de l'état de reprise



L'axe de la granularité de l'état de reprise mesure la continuité de l'interaction homme-machine entre les moments qui précédent et qui suivent l'adaptation. Cet axe se divise en trois grains : le niveau session, le niveau tâche et le niveau action.

Le premier niveau constitue le grain le plus gros. Il englobe l'ensemble des systèmes interactifs qui doivent subir un redémarrage suite à une adaptation. L'utilisateur doit alors recommencer sa tâche depuis le début. Un exemple de cet état de reprise pourrait être donné par le scénario-type d'Aura, si l'utilisateur avait à rouvrir le fichier qu'il était en train d'éditer avant l'adaptation, à le parcourir pour retrouver l'endroit qu'il éditait, pour enfin poursuivre sa tâche d'édition. Ce niveau est à rapprocher du moment de l'adaptation « entre sessions » de [Dieterich93].

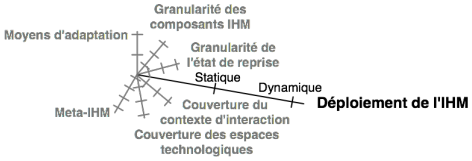
Le grain de reprise suivant, le niveau tâche, qualifie les reprises sur tâches élémentaires. À ce niveau, l'utilisateur doit reprendre à son commencement les tâches élémentaires courantes au moment de l'adaptation du système. Par exemple, s'il était en train de renseigner un champ texte qu'il n'aurait pas validé au moment de l'adaptation, l'utilisateur devra recommencer entièrement la saisie de ce champ une fois le système interactif adapté.

Enfin, le dernier grain de reprise, le niveau action, offre la plus grande continuité dans l'interaction. À ce niveau, la reprise de l'interaction s'effectue au grain de l'action physique. Par exemple, quand le contrôleur de diapositives est migré sur le PDA par le

[10] Tâche abstraite au sens de CTT (Concur Task Tree), c'est-à-dire une tâche non élémentaire.

logiciel de présentation, Bertrand continue sa présentation sans avoir besoin de faire défiler de nouveau les diapositives depuis la première de la pile.

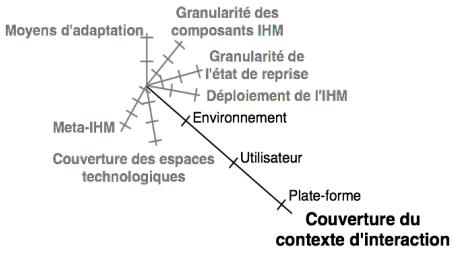
3.4 Déploiement de l'interface utilisateur et contexte de l'interaction



L'axe suivant, dédié au déploiement, caractérise le moment où peut avoir lieu l'adaptation du système interactif. Dans cette étude, je me focalise sur l'adaptation à l'exécution. En conséquence, cette dimension se partage de manière triviale en deux niveaux : les adaptations statique et dynamique.

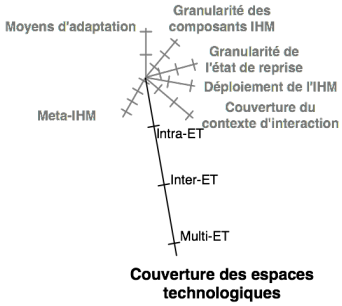
Le niveau statique concerne les adaptations réalisées alors que le système interactif n'est pas en cours d'utilisation. Cela regroupe les niveaux « d'adaptation au développement » et « d'adaptation à l'installation » de la taxonomie de [Thevenin01], ainsi que les niveaux « avant la première utilisation » et « entre sessions » de celle de [Dieterich93].

Le niveau dynamique caractérise les systèmes interactifs capables d'adaptation au moment de leur l'exécution, au cours de l'interaction avec l'utilisateur. Il serait possible d'aller plus loin dans les niveaux de déploiement en distinguant les adaptations dynamiques précalculées à la conception et les adaptations calculées dynamiquement. La première classe d'adaptation dynamique permet d'adapter un système interactif à l'exécution pour un ensemble de contextes d'interaction identifiés au moment de la conception. La deuxième classe d'adaptation dynamique est celle des systèmes capables d'adaptation à un contexte d'interaction qui n'a pas été prévu. Cependant, cela dépasse la portée de ces travaux de thèse, qui visent à proposer des mécanismes généraux d'adaptation à l'exécution. Ainsi, les mécanismes proposés peuvent être utilisés indifféremment pour construire des systèmes interactifs relevant de ces deux classes d'adaptation dynamique.



Enfin, l'axe du contexte de l'interaction décrit ce à quoi le système interactif peut être adapté. Cet axe se conforme à la définition du contexte de l'interaction que [Thevenin01] propose dans ses travaux de thèse. Cette dimension se décline en trois niveaux : l'environnement physique et social, l'utilisateur et la plate-forme. Comme le contexte de l'interaction a déjà été décrit dans [Thevenin01] et rappelé à la section I-1 de ce mémoire, que la notion de plate-forme, telle que je l'entends dans le cadre de cette étude, est décrite en détail à la section II-1.2, je ne développe pas cet axe plus avant.

3.5 Couverture des espaces technologiques



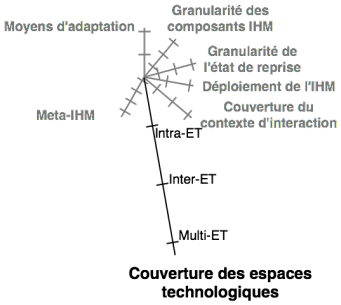
Couverture des espaces technologiques

Le sixième axe de ma taxonomie concerne le degré de couverture des espaces technologiques par le système interactif. La notion d'espace technologique, introduite par Kurtev, est définie ainsi : « contexte de travail comprenant un ensemble de concepts reliés, de connaissances, d'outils, de compétences requises et de possibilités » [Kurtev02]. Le contexte de travail est « souvent associé à une communauté donnée d'utilisateurs qui partage un savoir-faire, des supports d'enseignements, une littérature commune et même des ateliers de travail ou conférences ». Kutev cite plusieurs exemples d'espace technologique : les langages de programmation, l'ingénierie basée sur les ontologies, les langages XML, ou encore l'architecture dirigée par les modèles (MDA) telle que promue par l'OMG. L'objectif de Kurtev est d'étudier comment tirer partie de manière plus efficace des possibilités des différentes technologies en établissant des passerelles entre elles, favorisant ainsi la coopération entre les technologies, plutôt que d'organiser leur concurrence.

Dans mon étude, si j'admets cette définition, je l'applique en augmentant son pouvoir de discrimination. Par exemple, selon la définition et les exemples de [Kurtev02], l'ensemble des boîtes à outils basées sur des interacteurs de type « widget », l'ensemble des technologies à composants ou encore l'ensemble des technologies à services, peuvent être considérés comme trois espaces technologiques distincts : celui des boîtes à outils pour l'IHM, de l'approche à composants et de l'approche à services. Dans mes travaux, je cherche à être plus précis. Par exemple, je veux pouvoir distinguer, dans deux espaces technologiques différents, la boîte à outil « SWING » d'une part et la boîte à outil « Tk » d'autre part. De la même manière, je veux considérer dans deux espaces technologiques différents, la technologie à composants « EJB» de SUN d'une part, et la technologie à composants « CCM» de l'OMG d'autre part. Comme [Kurtev02], l'objectif est d'identifier des espaces technologiques afin d'établir des passerelles pour favoriser les coopérations.

En ingénierie de l'interaction homme-machine traditionnelle, un système interactif est implémenté au moyen d'un seul espace technologique. En informatique ubiquitaire, une telle homogénéité est rendue très difficile par le haut degré d'hétérogénéité des plates-formes élémentaires en présence et des solutions logicielles. Par exemple, la carte interactive de mon scénario-type est répartie sur un ordinateur puissant et un smartphone. Si l'interface utilisateur relative à la carte, qui est hébergée par l'ordinateur puissant, pouvait être développée à partir d'une boîte à outil OpenGL de dernière génération, ce n'est pas encore le cas du panneau de contrôle, qui est hébergé par un smartphone dépourvu de capacité de rendu graphique avancé. Ainsi, la construction de systèmes interactifs plastiques est difficilement envisageable à partir d'un unique espace technologique, mais requiert la

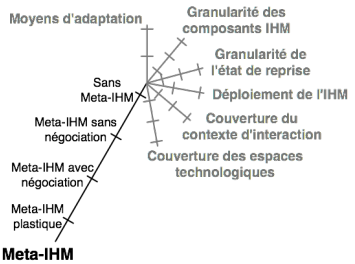
collaboration, à l'exécution, d'entités logicielles issues d'espaces technologiques distincts.



Couverture des espaces technologiques

Cette dimension de l'espace problème a donc pour vocation de caractériser la capacité d'un système à franchir les frontières posées par les espaces technologiques. Ainsi, cet axe se divise en trois niveaux : les systèmes intra-espace technologique (intra-ET), inter-espaces technologiques (inter-ET) et multi-espaces technologiques (multi-ET). Le niveau intra-espace technologique correspond aux systèmes interactifs implémentés et adaptés au sein d'un unique espace technologique. Par exemple, notre prototype de site Internet plastique « Sedan-Bouillon » [Balme05] se classe dans cette catégorie. Le niveau inter-espaces technologiques correspond à la situation où une interface utilisateur source, construite dans un espace technologique donné, est transformée vers un autre espace technologique cible. Par exemple, dans [bandelloni04-1], la migration de l'interface utilisateur provoque un changement de langage d'implémentation pour traduire un changement de modalité d'interaction : d'une IHM source implémentée en XHTML, le système passe à une IHM cible implémentée en VoiceXML. Enfin, le dernier niveau de cet axe concerne les systèmes dits multi-espaces technologiques. Ces systèmes ont leurs différents constituants implémentés dans des espaces technologiques distincts. C'est, par exemple, le cas de notre prototype « CamNote » [Demeure05], un visionneur plastique de diapositives. CamNote est capable de se répartir sur un ordinateur de type PC et le PDA. Sur le PC, l'IHM de l'afficheur de diapositives et du contrôleur de présentation sont implémentés avec la boîte à outils B207 [Demeure07]. Lorsque le contrôleur de diapositives migre sur le PDA, l'afficheur de diapositive reste sur le PC sous la forme d'un interacteur B207, alors que l'IHM du contrôleur de diapositives devient un composant exploitant la boîte à outil de Windows CE.

3.6 Méta-IHM



Meta-IHM

Une méta-IHM regroupe l'ensemble des fonctions (avec leur interface utilisateur) nécessaires et suffisantes pour contrôler et pour évaluer l'état du système interactif dans le cadre de l'informatique ubiquitaire [Coutaz06], [Roudaut06]. Cet ensemble de fonctions est « méta » dans la mesure où il transcende les services métiers qui sous-tendent l'activité humaine dans les espaces interactifs : quel que soit le domaine d'application du système interactif, les fonctions de la méta-IHM restent les mêmes et remplissent le rôle de gestionnaire de « l'espace interactif». Le desktop est un exemple de méta-IHM pour l'informatique traditionnelle.

Dans ma classification, je distingue quatre possibilités : les systèmes sans méta-IHM, les systèmes qui intègrent une méta-IHM sans négociation, les systèmes qui offrent une méta-IHM

capable de négocier et les systèmes qui incluent une méta-IHM plastique.

Une méta-IHM sans négociation permet de rendre observable l'état du processus d'adaptation, mais ne permet pas à l'utilisateur d'intervenir dans ce processus. C'est le cas de CamNote : lorsque le PDA est détecté, une redistribution et un remodelage sont appliqués automatiquement sans que l'utilisateur puisse intervenir. Cependant, pour accompagner ce processus, une animation, sous la forme d'un effet de fondu, exprime la migration du contrôleur de diapositive du PC vers le PDA. Cette animation constitue la méta-IHM qui permet à l'utilisateur d'évaluer la progression de l'adaptation. Lorsqu'une méta-IHM permet à l'utilisateur d'exercer un contrôle sur l'adaptation, elle se classe dans la catégorie des méta-IHM avec négociation. Conformément aux classes d'adaptation de [Dietericth93], ce contrôle de l'utilisateur peut prendre plusieurs formes : cela peut aller d'une simple possibilité d'annuler une adaptation en cours ou déjà réalisée, à demander à l'utilisateur de faire un choix parmi un ensemble d'adaptations possibles, ou même de permettre à l'utilisateur de prendre l'initiative d'une adaptation, voire de la conduire entièrement.

Enfin, la catégorie des méta-IHM plastiques exprime la dimension récursive de cette question. La méta-IHM étant un système interactif au même titre qu'un autre, il peut être intéressant qu'elle soit elle-même plastique afin de s'adapter aux changements du contexte de l'interaction en préservant son utilisabilité. L'existence de méta-IHM plastique pose le problème de la méta-IHM minimale « souche ». De la même manière, l'équilibre entre l'autonomie d'un système interactif dans l'adaptation et le trop grand nombre d'étapes de négociation avec l'utilisateur reste une question ouverte.

4. Synthèse

Les 7 axes de ma classification permettent d'identifier des requis sur la nature logicielle des systèmes interactifs plastiques. La couverture maximale des axes « moyen de l'adaptation » et « couverture du contexte de l'interaction » implique que le système interactif plastique puisse être déployé à façon sur un ensemble de plates-formes élémentaires. Comme nous l'avons déjà précisé, le cadre de l'informatique ubiquitaire dicte que cet ensemble de plates-formes élémentaires soit fortement hétérogène et variable au cours de l'interaction. Ainsi, un système interactif plastique est un logiciel distribué. La disparité des nœuds sur lesquels ce logiciel se distribue, associée à l'instabilité des

connexions entre ces nœuds, constitue une contrainte supplémentaire.

L'axe « déploiement de l'IHM » précise que l'adaptation par remodelage et redistribution d'un système interactif plastique peut-être réalisée dynamiquement, c'est-à-dire au cours de l'interaction avec l'utilisateur. Un tel système se classe dans la catégorie des systèmes capables de reconfiguration dynamique. Du point de vue logicielle, le degré d'autonomie dans la gestion des reconfigurations dynamiques est une question indépendante de mon espace problème.

En effet, si l'axe « méta-IHM » permet de caractériser le degré d'autonomie par rapport à l'utilisateur pour l'adaptation d'un système interactif plastique, elle n'indique pas de quelle manière technique est réalisée la reconfiguration dynamique qui en résulte. Indépendamment de l'intervention de l'utilisateur dans le processus, la reconfiguration dynamique résultante peut être mise en œuvre soit par des mécanismes propres au système interactif seul (reconfiguration interne), soit par des mécanismes logiciels tiers (reconfiguration externe), soit par la collaboration entre des mécanismes internes et externes (reconfiguration mixte).

L'approche par reconfiguration interne semble pertinente pour traiter le cas de l'adaptation d'un système à différentes situations identifiées au moment de la conception. Cependant, pour traiter l'adaptation à des situations non prévues initialement, l'approche par reconfiguration externe ou mixte devient une nécessité [Oreizy99]. Dans cette étude, qui concerne des mécanismes d'ordre général, ces trois classes de reconfiguration font partie de l'espace solution.

Enfin, l'axe « couverture des espaces technologiques » pose très directement un requis sur l'implémentation d'un système interactif plastique. Pour assurer une couverture maximale de cet axe, le logiciel qui sous-tend un système interactif plastique doit être capable, d'une part, de traiter avec l'hétérogénéité de son environnement d'exécution, et d'autre part, d'être lui-même construit à partir de constituants de nature hétérogène.

En synthèse, la couverture maximale de mon espace problème est assurée par un système interactif capable de s'adapter ou d'être adapté par remodelage et par redistribution. L'adaptation par remodelage prend en compte l'ensemble des propriétés CARE (Complémentarité, Assignation, Redondance, Equivalence) de la multimodalité synergique à la monomodalité des interfaces utilisateur, et peut intervenir à tous les niveaux d'abstraction du système interactif. L'adaptation par remodelage et par redistribution se joue au niveau de l'interacteur et permet une reprise après adaptation au niveau de l'action physique. Elle couvre l'ensemble du contexte de l'interaction, c'est-à-dire

l'environnement physique et social, l'utilisateur et la plate-forme. Le système interactif est composé de constituants issus d'espaces technologiques distincts et intègre une méta-IHM plastique. Un tel système est un logiciel distribué, dynamiquement reconfigurable par des mécanismes internes, externes ou mixtes, et capable de traiter avec l'hétérogénéité de son environnement d'exécution.

Nous exploitons maintenant les dimensions de l'espace problème pour analyser, au chapitre 3, différentes propositions de l'état de l'art en IHM, puis au chapitre 4, l'offre du Génie Logiciel qui porte sur la distribution, la reconfiguration dynamique et l'hétérogénéité des technologies.

Chapitre III

IHM plastiques : approches et techniques

Avant-propos

L'adaptation des logiciels concerne toutes les couches logicielles d'un système. Il n'est donc pas surprenant que ce problème soit traité par des communautés scientifiques distinctes et, dans chaque communauté, sous des angles d'attaque différents. La figure III-1 exprime la situation du point de vue de la communauté ingénierie de l'Interaction Homme-Machine. On y distingue deux grands angles d'attaque : l'un « centré modèle » plutôt concerné par les étapes de conception (partie droite de la figure), l'autre « centré code » plutôt concerné par la phase d'exécution (partie gauche de la figure).

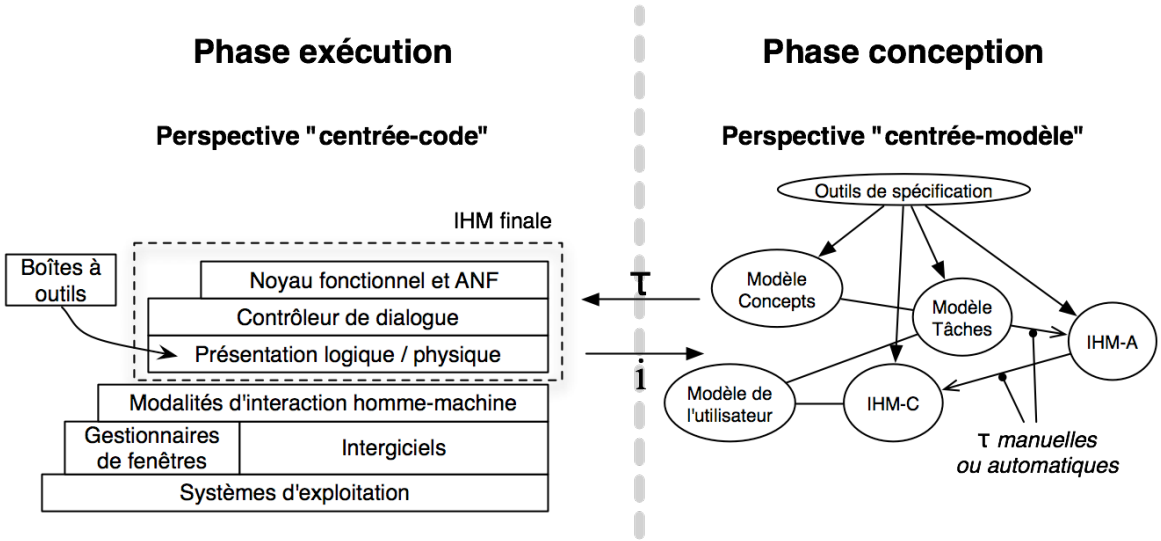


Figure III-1 Point de vue de la communauté IIHM sur l'adaptation des logiciels

Selon l'angle *centré modèle*, la mission pour la plasticité des IHM consiste essentiellement à définir les métamodèles et modèles idoines reliés par des relations de transformation et de mise en correspondance, pour in fine, produire du code exécutable ou interprétable. *A contrario*, à partir du code, procéder par abstractions successives, pour générer du code adapté au nouveau contexte d'interaction. Faisant suite au travail séminal de D. Thevenin sur la plasticité des IHM [Thevenin01], de nombreux travaux sont venus améliorer la nature des métamodèles : modèle de tâche et des concepts (MT&C), IHM abstraite (IHM-A), IHM concrète (IHM-C), IHM finale (IHM-F), et modèle du contexte d'interaction [Coutaz08]. La section 1 présente en synthèse, les dernières avancées de cette approche.

Selon l'angle *centré code*, la mission consiste essentiellement à définir des infrastructures (intergiciels) et leurs bibliothèques de

programmation pour masquer la complexité et la diversité des systèmes d'exploitation, de même, la gestion de ressources de toutes sortes y compris les ressources d'interaction. En IHM, les premiers efforts ont porté sur les gestionnaires de fenêtres, puis sur les interpréteurs de modalités (machines graphiques comme OpenGL et SVG, systèmes de reconnaissance de la parole, systèmes de suivi et de reconnaissance par vision par ordinateur, etc.). Ainsi, la plasticité des IHM, sous l'angle centré code, est-elle traitée au niveau des gestionnaires de fenêtres, des interprètes de modalités, des boîtes à outils et des infrastructures. La section 2 présente en synthèse, les dernières avancées sur ces points.

Puisque mon mémoire de thèse porte essentiellement sur les infrastructures, j'analyse en détail dans les sections 3 à 5 les solutions dont l'approche consiste à proposer ou à s'appuyer sur une infrastructure logicielle. Chaque solution est décrite puis projetée dans mon espace problème.

1. L'approche centrée modèle en synthèse

Il convient de retenir quatre avancées majeures issues de l'approche centrée modèle : l'amélioration des algorithmes de génération, l'existence d'un langage de représentation des IHM, le changement de statut des transformations et du code exécutable.

1.1 Amélioration des algorithmes de génération

SUPPLE [Gajos04] sert ici d'exemple de référence. Contrairement à Pebbles [Nichols02] qui génère des IHM pour une classe particulière de dispositifs (des PDA) ou à XIML [Puerta02] qui exige que le concepteur d'IHM spécifie explicitement le choix des widgets en fonction de la taille de l'écran, SUPPLE se veut général en assimilant la génération d'IHM à un problème d'optimisation. Le but est de choisir parmi les IHM candidates, celle qui minimise l'effort estimé d'utilisation. SUPPLE définit la meilleure façon de générer une IHM à partir de trois modèles : le modèle de fonctionnement de l'application, le modèle de l'utilisateur (en fait, ses traces d'utilisation de l'application), et le modèle du dispositif cible (c'est-à-dire les widgets disponibles et le coût utilisateur de manipulation de ces widgets).

SUPPLE n'utilise pas de modèle de tâche, mais un modèle de fonctionnement de l'application représentée sous forme d'arbre. Cet arbre exprime des relations sémantiques entre les objets typés pour lesquels il convient de produire une représentation concrète. À l'évidence, cette approche convient pour des applications simples comme le contrôle des lumières et des projecteurs vidéo d'une pièce : les types d'objets sont simples et l'arbre guide le placement des widgets dans l'IHM concrète. SUPPLE++ fait suite

à ces travaux pour la génération d'IHM pour mal-voyants [Gajos07].

Comme le remarque Nichols, la génération automatique d'IHM pour une même application donnée, mais pour des dispositifs cibles distincts peut donner lieu à des IHM incohérentes entre elles. Ceci n'est pas souhaitable pour l'utilisateur susceptible de changer de dispositif. Ou encore, l'IHM dépend étroitement de la façon dont le développeur décrit l'application : deux applications proches fonctionnellement risquent de se retrouver avec des IHM incohérentes. UNIFORM (Using Novel Interfaces For Operating Remotes that Match) vise à réduire ces incohérences, mais pour le cas simple des télécommandes [Nichols06].

1.2 Langage de représentation des IHM

UsiXML est en train d'émerger comme *lingua franca* pour la représentation des IHM à différents niveaux d'abstraction [Limbourg04]. L'existence de ce langage favorise l'interopérabilité entre les nombreux outils de manipulation des modèles comme Teresa [Mori03], MARI [Sottet07], SketchiXML [Coyette04], ReversiXML [Bouillon04] ou VisualiXML [Schlee04]. L'intérêt est de pouvoir faire coopérer des outils et donc des concepteurs aux habitudes et aux métiers différents.

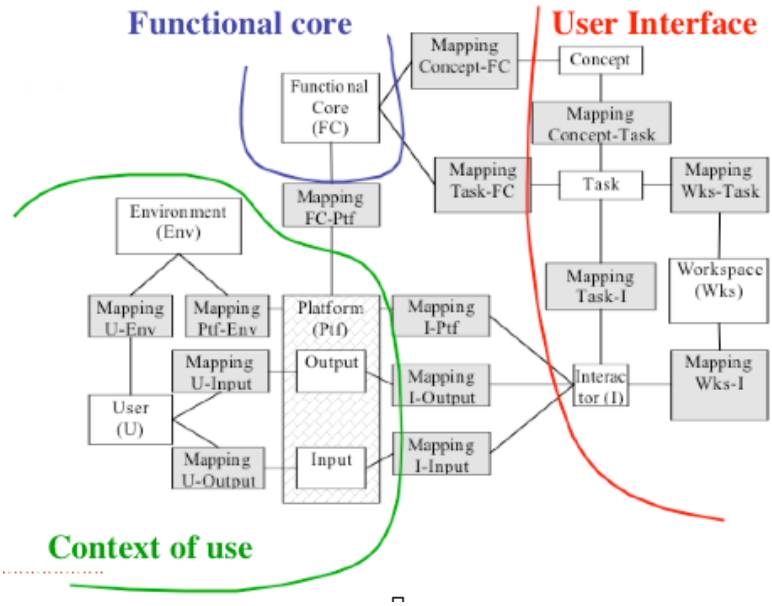


Figure III-2 Une IHM est un graphe de modèles (tiré de [Sottet07]).

1.3 Statut des transformations

D'expression diluée dans le code des générateurs, les transformations prennent maintenant le statut de modèle [Sottet07]. Devenues explicites, elles sont contrôlables et transformables en vue de produire des IHM conformes à des règles ergonomiques choisies par le concepteur [Calvary07]. Cette

solution répond à la critique de Myers qui, à propos des générateurs d'IHM, note que les générateurs ne sont ni contrôlables ni inspectables [Myers00].

1.4 Statut du code exécutable

Le code exécutable d'une IHM prend le statut de modèle : ce n'est qu'une perspective de plus sur une IHM qui vient se greffer sur le MT&C, IHM-A, IHM-C. Autrement dit, à l'exécution, une IHM est un graphe de modèles reliés par des transformations et des mises en correspondance. Si la figure III-2, tirée des travaux de thèse de Jean-Sébastien Sottet, en est un exemple, l'approche mise en œuvre dans les COTs [Lewandowski07] en est un autre. Dans cette approche, dans le but de rendre le système interactif malléable (au sens de [Mørch97]), celui-ci est un assemblage de composants logiciels « orientés tâches ». Un « composant orienté tâches » est un composant logiciel pour lequel des liens sémantiques sont établis entre le modèle de tâches qu'il sous-tend et les points d'accès correspondants au niveau du code du composant. Ces liens sémantiques facilitent le travail de l'intégrateur de composant, lorsque celui-ci ajoute un COT au sein d'un assemblage. Projetée dans mon espace problème, cette approche vise à mettre en œuvre des composants IHM d'un niveau de granularité correspondant à « l'espace de dialogue », assure une reprise de l'adaptation au niveau de la session et traite du déploiement statique de l'IHM.

2. L'approche centrée code

2.1 Gestionnaire de fenêtres (window managers)

Sous la pression des acquis et des logiciels patrimoniaux – trop d'applications graphiques dépendant des window managers imposés par les constructeurs – on relève peu de contributions sur la plasticité des IHM à ce bas niveau d'abstraction. Nous retenons deux exceptions : I-AM [Lachenal04] et Façade-Métisse [Stuerzlinger06].

2.1.1 I-AM (Interaction Abstract Machine)

I-AM facilite la mise en œuvre d'IHM graphiques distribuées et migrables au sein d'espaces interactifs multisurface, multi-instrument configurables dynamiquement. I-AM assure la découverte des ressources d'interaction, modélise leur couplage et leurs relations spatiales et forme, du point de vue du développeur comme de l'utilisateur, un espace homogène d'interaction comme si ces ressources appartenaient à une machine unique.

Concrètement, comme le montre la Figure III-3, il est possible à un utilisateur de relier plusieurs machines exécutant des systèmes

d'exploitation différents (MacOS, Windows et Linux) et d'avoir l'impression qu'il s'agit d'une même machine. Du point de vue programmation, la multiplicité et l'hétérogénéité des machines et des systèmes d'exploitation sont transparentes au programmeur, mais accessibles si le besoin s'en fait sentir.

Si les services proposés par I-AM répondent bien aux exigences de l'informatique ubiquitaire pour ce qui est de la gestion des ressources d'interaction avec notamment l'expression explicite des relations des surfaces d'affichage dans l'espace, en revanche son implémentation ne permet pas la réutilisation directe des boîtes à outils patrimoniales dont les primitives doivent être encapsulées une à une. Autrement dit, les applications doivent être réécrites avec I-AM. Cette contrainte limite *de facto* l'adoption de cette approche à moins d'exploiter, lorsqu'elles existent, les capacités d'introspection des boîtes à outils natives.

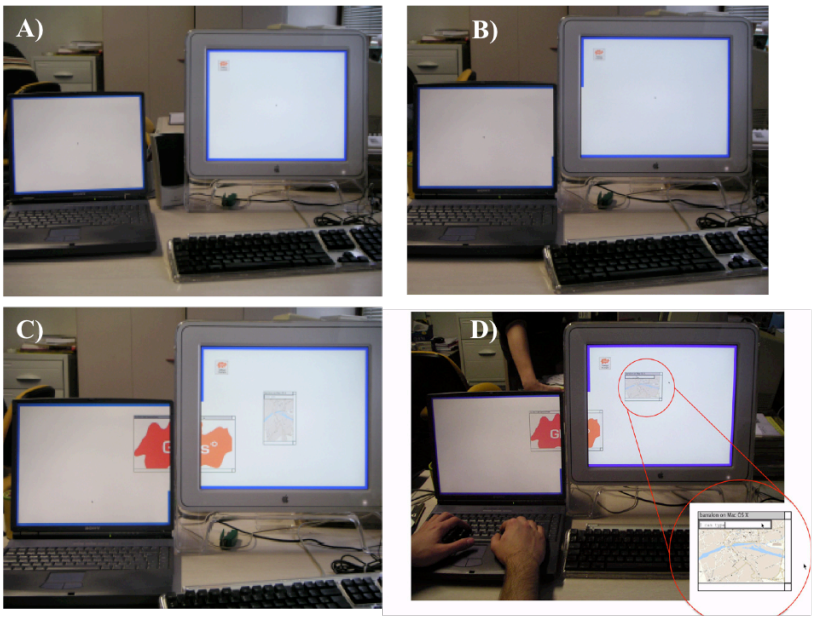


Figure III-3 I-AM picture (tiré de [Lachenal04], p. 107).

Dans l'espace problème de la plasticité des IHM, I-AM couvre la dimension redistribution à tous les niveaux de granularité avec état de reprise au niveau de l'action. Le contexte d'interaction est limité à la plate-forme, mais cette plate-forme est dynamique et hétérogène. La question du remodelage est laissée au développeur.

2.1.2 Façade

Façade prend un point de vue radicalement différent d'I-AM : les applications cibles sont patrimoniales (elles n'ont pas besoin d'être réécrite pour fonctionner avec Façade), la plate-forme cible est élémentaire, et l'utilisateur final (non pas le développeur) est au centre des préoccupations. Façade est une méta-IHM (au sens de mon espace problème) qui permet à l'utilisateur de

personnaliser les IHM graphiques des applications au niveau des interacteurs et cela de manière transparente pour ces applications. Par glisser-déposer et couper-coller, l'utilisateur découpe et recompose, voire remplace à volonté des widgets natifs. Il peut ainsi remplacer les IHM originales des applications patrimoniales par des IHM Façade personnalisées. Les fenêtres sources peuvent être fermées ou iconifiées, l'utilisateur conduisant ses activités avec les IHM Façade créées à façon. La figure III-4 montre un exemple de transformations.

Sur le plan technique, Façade fait deux hypothèse : a) les IHM des applications patrimoniales sont développées avec des boîtes à outils avec capacité d'introspection b) existence d'un minimum d'infrastructure : Métisse, qui inclut un serveur X et un compositeur qui est chargé d'une part de générer le rendu des fenêtres Métisse dans une fenêtre OpenGL plein écran du window manager natif et qui d'autre part capte les événements utilisateur et les événements en provenance des applications à destination de l'IHM. Façade s'appuie aussi sur les services d'instrospection des boîtes à outils natives pour récupérer l'arbre des widgets des applications sources et leur état.

Dans l'espace problème de la plasticité des IHM, Façade est une méta-IHM avec négociation (aucune décision d'adaptation n'est laissée au système) qui permet le remodelage au niveau de l'interacteur et ceci de manière dynamique avec un grain de reprise au niveau de l'action. Inversement, l'adaptation est intra-modale (de nature graphique) et ne touche pour l'instant que des refontes du niveau de la présentation (pas de changement possible au niveau du contrôleur de dialogue), pour un contexte d'interaction limité à une plate-forme élémentaire statique éliminant d'office la redistribution (mais ceci est un choix argumenté des auteurs [Stuerzlinger06]). La redistribution semble toutefois envisageable pour une configuration de plates-formes élémentaires équipées toutes de Métisse, donc dans un espace technologique donné.

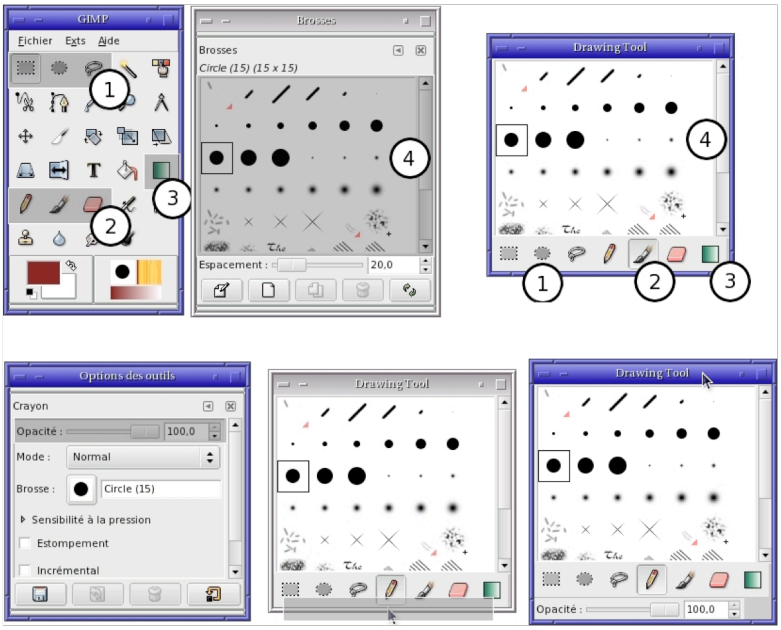


Figure III-4 Un exemple de personnalisation d'IHM graphiques avec Façade (tiré de [Stuerzlinger06]).

2.1.3 Window Managers en synthèse

I-AM et Façade, deux exemples d'adaptation des IHM sous l'angle des gestionnaires de fenêtres, s'appuient sur des modèles architecturaux bien distincts : I-AM ne traite pas les applications patrimoniales, mais procède par machine abstraite unifiant ainsi la gestion dynamique des ressources d'interaction dans l'espace. Le programmeur et l'utilisateur obtiennent 'gratuitement' la redistribution d'IHM, mais pas le remodelage. Façade, au contraire, traite les applications patrimoniales et procède par insertion d'un serveur intermédiaire qui joue le rôle de transformateur et de mappeur (au niveau des widgets et des événements) entre l'IHM patrimoniale et l'IHM remodelée. Le programmeur et l'utilisateur obtiennent 'gratuitement' le remodelage, mais pas la redistribution.

Si avec Façade on dispose d'un bon compromis entre les acquis et le remodelage, avec I-AM se pose la question de la révision du modèle de window manager introduit il y a 25 ans. L'apparition d'exemples comme le démonstrateur de Cao et Balakrishnan[11], milite pour une révision vers plus de généralité de façon à fournir au développeur des services de gestion dynamique dans l'espace et dans le temps de ressources d'interaction extrêmement hétérogènes : un objet conçu pour une mission de la vie courante

[11] Dans ce démonstrateur, l'utilisateur définit dynamiquement ses surfaces d'affichage dans l'espace au moyen d'un vidéo-projecteur dans une main et un crayon de l'autre [Cao06].

peut temporairement jouer ce rôle. Par exemple, une cuillère et un morceau de sucre peuvent, à la faveur d'une discussion autour d'une table augmentée, jouer le rôle de phicon pour représenter, une rue, un immeuble, etc.

2.2 Interprètes et générateurs de modalité

Qu'il s'agisse de rendu graphique, de reconnaissance ou de synthèse de la parole et du son, de suivi de doigts ou du regard, ou encore de reconnaissance de gestes, dans chacun de ces domaines, la recherche vise l'amélioration des performances au regard des changements des conditions environnementales physiques : niveau de bruit, variations intempestives de la lumière, diversité intra- et inter-locuteur, variation des ressources de calcul et de communication (en particulier pour les rendus de document multimédia – vidéo ou audio streaming). Les recherches au sein de chacune de ces modalités (domaines) sont extrêmement actives avec l'invention de nouveaux algorithmes y compris des algorithmes de fusion multimodale au niveau du traitement du signal.

Au-delà des progrès algorithmiques, deux tendances méritent d'être soulignées : la première est de faire des interprètes de modalité de véritables sous-systèmes autonomes capables d'assurer une qualité de service donnée. La seconde est l'utilisation des modalités d'entrée, non pas pour l'interaction explicite, mais pour l'interaction implicite. Dès lors, les résultats des interprètes de modalité ne servent pas uniquement d'entrée au contrôleur de dialogue d'un système interactif, mais également au gestionnaire du contexte d'interaction. Nous verrons au chapitre V, l'impact de ces tendances sur l'architecture logicielle que je propose.

Dans l'espace problème de la plasticité des IHM, les interprètes et les générateurs de modalité visent l'adaptation au contexte d'interaction pour améliorer leur robustesse et ce faisant, maintenir la continuité de l'interaction au niveau de l'action physique. L'adaptation se fait au mieux par remodelage (absence de redistribution), de manière intra-modale et sans méta-IHM (puisque c'est l'autonomie qui est visée). Ces interprètes et ces générateurs servent au développement des boîtes à outils d'interacteurs.

2.3 Boîtes à outils d'interacteurs

On observe un regain d'intérêt pour le développement de boîtes à outils d'interacteurs alors que la concurrence de l'industrie se fait prégnante en ce domaine. Plusieurs raisons à cela : le post-WIMP graphique multipoint [Bérard06], les IHM tangibles comme ECT [Greenhalgh04] ou [ARToolKit]. Dans cette grande diversité, j'ai retenu quatre contributions liées à la plasticité des IHM : Ubit, Multimodal Widget, ETk, et les Comets, mais d'autres

propositions comme ICON [Dragivecic01] ou HsmTk [Blanch05] sont intéressantes par leurs argumentaires architecturaux.

2.3.1 Ubit

Ubit [Lecolinet99], avec sa structure de scène sous forme de graphe orienté sans cycle, avec l'existence de nœuds conditions et avec l'exploitation de la notion usuelle de contexte graphique, rend possible le rendu multiple d'un même widget. Ainsi, Ubit offre les mécanismes de base pour le remodelage dynamique intra-modal graphique d'interacteur au niveau de l'action physique, mais c'est au programmeur de prévoir toutes les formes de rendu. Il en va de même avec Multimodal Widget.

2.3.2 Multimodal Widget

Dans Multimodal Widget [Crease00], [Crease01], le remodelage peut être inter-modal : un widget peut être simultanément rendu dans plusieurs modalités cohérentes entre elles, mais aussi avec les ressources d'interaction disponibles. Par exemple, un bouton graphique, à la visite de la souris, peut à la fois se repeindre en jaune et émettre un son si, dans l'état actuel de l'interaction, le générateur de son est disponible. L'utilisateur dispose d'une méta-IHM pour contrôler la proportion de modalité souhaitée (par exemple, 100% de graphique et 30% de retour sonore). Contrairement à ETk, Multimodal Widget ne couvre pas la redistribution d'IHM.

2.3.3 ETk

ETk ou EBL[12]/Tk [Grolaux07] fournit au programmeur les widgets usuels Tk, mais ceux-ci peuvent être découpés, recomposés, remplacés (comme dans Façade), voire redistribués s'ils sont dotés de la capacité[13] « migration ». La redistribution et le remodelage ont lieu sur des plates-formes composées dynamiques hétérogènes et ceci de manière transparente pour le programmeur d'application : comme dans I-AM, c'est comme si les widgets résidaient sur une même plate-forme élémentaire, mais l'application en est avertie par des événements de migration. ETk couvre une bonne partie de mon espace problème, mais au sein d'un unique espace technologique : le système Mozart [Mozart], [Van Roy04], son langage Oz et Tk, dont ETk exploite les propriétés pour la redistribution et le remodelage.

2.3.4 Les Comets

Les Comets [Demeure07] sont, à notre connaissance, la seule solution capable de couvrir les objectifs-clé de la plasticité à partir

[12] EBL : Enhanced Binding Layer

[13] Au sens de la notion de capability utilisée en système

d'une boîte à outils. Cette boîte à outil s'appuie sur le modèle d'architecture du même nom qui distribue les perspectives Tâches, IHM-A, IHM-C et IHM-F d'un système interactif en un arbre de comets[14]. C'est l'idée de PAC, mais ici poussée à l'extrême. Cette arborescence est isomorphe au modèle de tâche du système grâce à des comets qui jouent le rôle d'opérateurs de tâche, tel l'entrelacement. Pour les autres comets de l'arborescence, chacune couvre une sous-tâche du système, avec plusieurs IHM-A possibles correspondant chacune à cette sous-tâche, et pour chaque IHM-A, plusieurs IHM-C et IHM-F pouvant être implémentées dans des espaces technologiques distincts.

Contrairement à toutes les boîtes à outils, le remodelage d'une IHM exprimée en Comets n'est pas limité aux niveaux de Présentation Logique et Physique, mais couvre tous les niveaux d'abstraction. Par exemple, le remplacement d'une comet d'entrelacement par une comet de séquencement traduit le changement d'un entrelacement de tâches par une obligation sémantique de les exécuter en séquence. Avec le langage CSS++ et COMET/RE, le programmeur peut contrôler le rendu du graphe des comets, remplacer une modalité par une autre ou présenter une comet dans plusieurs modalités simultanément.

2.3.5 Les boîtes à outils en synthèse

En synthèse, les boîtes à outils couvrent de manière partielle nos requis. La plupart supposent un espace technologique unique. En outre, dans le domaine des IHM multimodales, les propriétés CARE sont à peine couvertes. On relèvera néanmoins l'équivalence fonctionnelle et la redondance avec les Multimodal Widgets et les Comets. La complémentarité de modalités reste un problème complexe qui peut d'ailleurs nécessiter des connaissances du niveau du contrôleur de dialogue, donc au-dessus du niveau des présentations physique et logique supposées couvert par les interacteurs des boîtes à outils.

Nous avons vu jusqu'ici que la répartition et le remodelage des IHM n'étaient que partiellement couverts par les windows managers, par les interprètes de modalités et par les boîtes à outils d'interacteurs. Analysons maintenant l'apport des infrastructures.

2.4 Infrastructures

L'avantage des infrastructures ou intergiciels est de reporter dans les couches basses, mais au-dessus des systèmes d'exploitation, des problèmes récurrents dont les solutions sont communes à un grand nombre d'applications ou tout au moins à une classe d'applications.

[14] Il s'agit en vérité d'un graphe. Pour simplifier l'exposé, nous considérons pour l'instant, qu'il s'agit d'un arbre.

Nous analysons ici, les infrastructures dont les services tendent à résoudre le problème de la plasticité des IHM. Nous les répartissons en deux groupes selon que le problème traité est la redistribution ou le remodelage. BEACH et Aura relèvent de la première catégorie tandis que ICrafter, Supple et Huddle s'intéressent plutôt au remodelage. Seul, Eloquence, effectue du remodelage inter-modal, mais en sortie seulement. Speakeasy est un cas intéressant centré sur l'agrégation de services sous le pur contrôle de l'utilisateur.

3. Infrastructures et redistribution d'IHM

Dans cette catégorie d'infrastructure, j'ai retenu *BEACH* et *Aura* qui visent des objectifs complémentaires : la première, les activités de collaboration dans une salle interactive, la seconde, la migration de l'utilisateur entre lieux sans perdre son travail.

3.1 BEACH

À notre connaissance, BEACH (Basic Environment for Active Collaboration with Hypermedia) [Tandler01] est l'une des toutes premières infrastructures motivées par la mise en œuvre de salles de réunion interactives (smart rooms), en l'occurrence i-LAND [Streitz99]. Le projet i-LAND inclut quatre « roomware »[15] : DynaWall, InteracTable, CommChair et ConnecTable. DynaWall est un mur interactif de près de $5m^2$ qui permet à plusieurs utilisateurs de travailler simultanément, de manière soit indépendante, soit collaborative. InteracTable est une table interactive mobile qui offre à un maximum de six utilisateurs disposés tout autour, le moyen de créer, d'afficher, de discuter et d'annoter des éléments d'information. Les CommChair sont des fauteuils mobiles qui intègrent un ordinateur de type « tablette » avec lequel il est possible d'interagir à l'aide d'un stylet. Enfin, les ConnecTable [Tandler01-2] sont des tables interactives dotées d'un écran également accompagné d'un stylet. Deux ConnecTables disposées côte à côte ont la particularité de mettre en commun leurs ressources interactives : par exemple, leur deux écrans n'en forme plus qu'un. Ainsi, deux ConnecTables mises l'une en face de l'autre deviennent adaptées à un travail collaboratif entre deux personnes se faisant face.

3.1.1 Analyse

La Figure III-5 montre comment BEACH facilite la mise en œuvre de ces roomware.

[15] Dans ce contexte, le terme « roomware » désigne des éléments de mobilier augmentés de capacité de calcul.

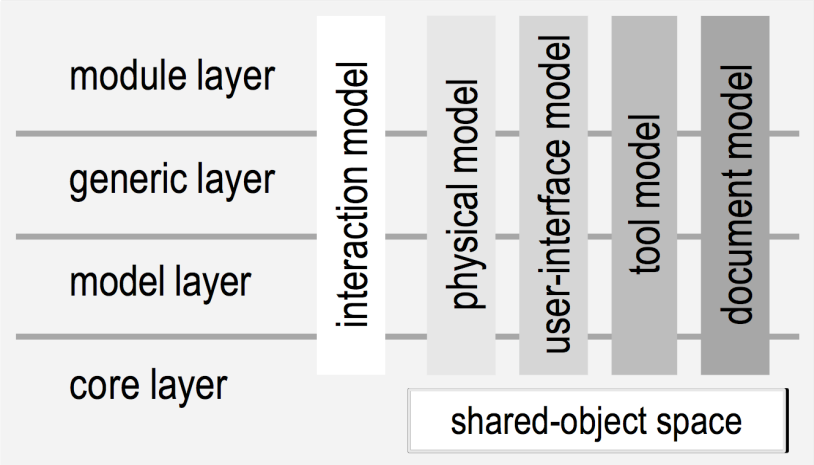


Figure III-5 Vue générale de l'architecture logicielle de BEACH

BEACH est structuré en 4 niveaux d'abstraction (core layer, model layer, generic layer, module layer) qui facilitent la mise en œuvre des roomware dont la structuration est censée respecter les 5 préoccupations suivantes : interaction model, physical model, UI model, tool model, et document model. Pour simplifier et en reprenant la décomposition Arch qui fait référence en IHM, le *tool model* et le *document model* correspondent à l'adaptateur de noyau fonctionnel et au noyau fonctionnel de Arch, mais avec une orientation collecticiel. Le *user interface model* aborde le problème du remodelage : il a la charge de définir une IHM en fonction des dispositifs cibles. La redistribution d'IHM est gérée par le *physical model* qui inclut un modèle de la plate-forme physique avec ses ressources d'interaction. Il a la charge de la présentation physique au sens de Arch.

Comme le montre la Figure III-6, cette décomposition fonctionnelle est déployée selon le modèle client-serveur. Chaque PC doit exécuter un client BEACH. Un serveur central, situé sur une machine dédiée, permet la synchronisation et la persistance des données issues des clients répartis sur chaque roomware.

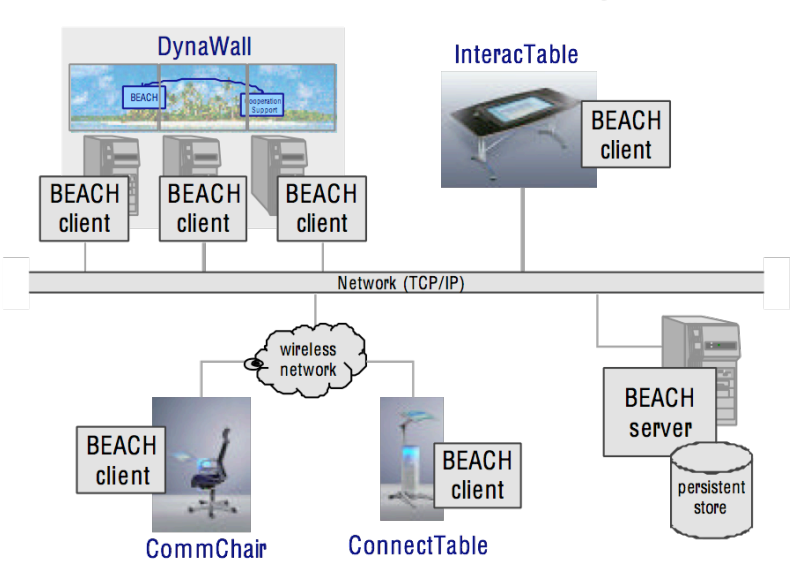


Figure III-6 Les clients BEACH, situés sur chaque plate-forme élémentaire, sont synchronisés par un serveur unique.

La mise en œuvre de ces cinq aspects s'appuie sur les services de base des couches de BEACH. La couche core layer complète les insuffisances des systèmes d'exploitation et des window managers natifs. Elle inclut notamment la gestion des capteurs qui permet de déterminer que deux tablettes sont connectées selon des orientations données. Le core layer inclut la gestion d'objets partagés qui sert notamment au maintien de la cohérence des vues graphiques réparties ou répliquées sur différents écrans, mais aussi, comme le pratique I-AM, les transformations (affines) de ces vues graphiques par l'introduction de wrappers, et la gestion des événements provenant des dispositifs d'entrée gérés par des machines distinctes. La couche « modèle » fournit les classes abstraites nécessaires aux roomware. Par exemple, c'est ici, que sont offerts les mécanismes de création et de mise à jour de vues et de controller (au sens de MVC). La couche « générique » implémente des services d'utilité publique orientée collecticiel et la couche « module » (module layer) permet l'implémentation d'éléments sur mesure pour des tâches applicatives spécifiques.

3.1.2 BEACH en Synthèse

Dans mon espace problème (voir Figure III-7), BEACH couvre la redistribution des IHM au niveau du pixel avec reprise au niveau de l'action physique : comme dans I-AM, le rendu d'un interacteur peut être réparti sur deux écrans contigus. Le remodelage est à la charge du développeur. Comme I-AM, BEACH s'occupe des transformations de la scène graphique pour tenir compte de l'orientation et du couplage des écrans.

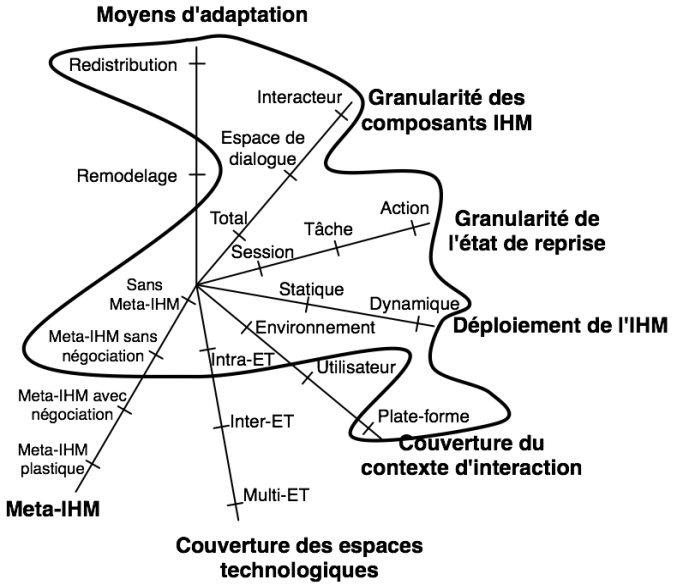


Figure III-7 Projection de BEACH dans l'espace problème.

La plate-forme est dynamique mais homogène : toutes les plates-formes élémentaires sont supposées être des stations de travail de type PC. Les ressources d'interaction peuvent être hétérogènes, mais pas au sein d'un roomware. Par exemple, le Dynawall est constitué de plusieurs SmartBoard de taille et de résolution identiques. Il en va de même pour les connectTables composées de deux tablettes identiques. Tous les roomware sont censés avoir été développés dans la technologie BEACH. Autrement dit, des clients développés avec une autre technologie ne peuvent ni s'intégrer au sein d'un système interactif développé avec BEACH, ni bénéficier des mécanismes et outils offerts. Nous sommes donc dans une situation intra-espace technologique et monomodale graphique. Dans le cas des ConnecTables, le couplage dynamique des deux tablettes par rapprochement est un exemple de méta-IHM sans négociation.

Alors que BEACH est une infrastructure pour les activités humaines de collaboration au sein d'une même salle, Aura vise, non pas la collaboration, mais la migration des activités d'une personne d'un lieu à l'autre : une activité d'édition de document peut être commencée en un lieu, suspendue, puis reprise ailleurs.

3.2 Aura

Le projet Aura[16], lancé au début des années 2000 à l'université de Carnegie Mellon, s'appuie sur l'hypothèse suivante : l'attention humaine est devenue la plus précieuse des ressources qu'un système informatique se doit de protéger. Partant de là, l'objectif est de fournir à chaque utilisateur un halo computationel qui le

[16] http://www.cs.cmu/~aura/

suit dans ses activités et ceci sans rupture, d'où le nom du projet : Aura personnelle d'informations.

3.2.1 Analyse

La notion de tâche est le concept central de la solution Aura assortie de l'expression de qualité de service. Ici, la notion de tâche désigne une activité humaine, par exemple l'édition de document avec l'écoute de morceau musical. En IHM, une telle tâche serait la racine d'un arbre de profondeur 1 dont les feuilles seraient des applications comme MS Word. Aura procède donc à très gros-grain. L'expression de la qualité de service accessible à l'utilisateur permet de guider le système dans ses opérations de reconfiguration et de reprise. Par exemple, si le contexte le permet, jouer un morceau musical tout en éditant un document, sinon s'en tenir à l'édition de document.

Une tâche Aura est l'unité d'exécution entre contextes d'interaction. Ceci signifie que si l'utilisateur commence l'édition d'un document en un lieu avec MS Word et qu'il s'arrête à la page 24, cette tâche sera reprise automatiquement en un autre lieu ne disposant pas de Word mais d'un autre éditeur avec le document ouvert à la même page, et ceci sans que l'utilisateur ait à faire autre chose que se déplacer : son aura informationnelle le suit de manière transparente de contexte en contexte. La Figure III-8 illustre ce principe. La couche « Task Management » détermine ce dont l'utilisateur a besoin en ce lieu et à cet instant, tandis que la couche « Managed Environment » détermine comment configurer l'environnement pour satisfaire ce besoin. Les modèles de tâches, qui expriment ces besoins à haut niveau d'abstraction, sont partagés entre environnements par le biais d'un système de gestion de fichiers distribués.

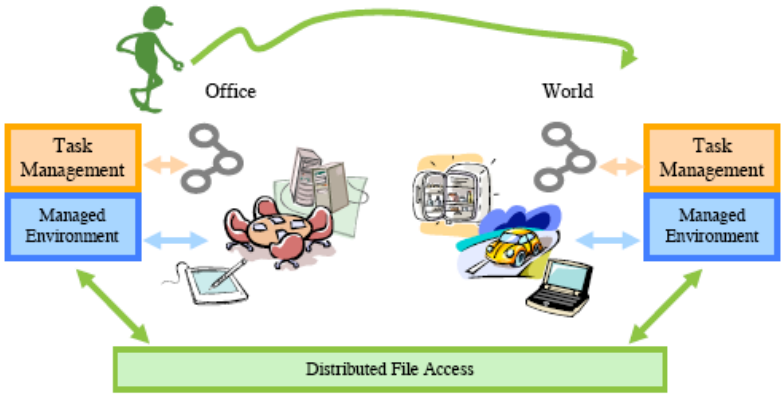


Figure III-8 L'aura informationnelle suit l'utilisateur dans ses déplacements [Extrait de Sousa05, p. 10].

La Figure III-9 montre un exemple de modèle de tâche Aura. Cette tâche (l'identificateur 34 la désigne de manière unique dans l'espace de noms de l'utilisateur) comprend deux services : « play

video » (id=1) et « edit text » (id =2). Ces services utilisent un « material » (matériau) c'est-à-dire une ressource d'information. Par exemple, le service « play video » joue le matériau « 11 » qui se trouve être à l'arrêt et au début (cursor = 0). Ce matériau (un film) est visualisé dans une fenêtre d'une certaine taille placée en un certain endroit de l'écran. La réalisation de la tâche Aura 34 peut se faire selon deux configurations : la configuration de nom « all » est préférée à la configuration de nom « only video » : son poids (weight = « 1.0 ») est supérieur à celui de « only video » (weight = « 0.7 »). La configuration « all » spécifie l'exécution des deux services « play video » et « edit text » (désignés par leur id) ouverts sur les deux matériaux fichier vidéo et fichier texte (id respectifs 11 et 21).

On le voit, le niveau d'abstraction d'un modèle de tâche est indépendant des applications effectives. De plus, l'état de reprise (checkpoint) correspond à un état perçu par l'utilisateur : les applications ne sont pas nommées directement, mais accessibles à travers le concept de service et les fichiers sont modélisés sous forme de matériau dont l'état est exprimé en caractéristiques externes (le film vidéo est à l'arrêt et en début). Pour cela, Aura fait une hypothèse forte sur la capacité d'ouverture des applications patrimoniales. Il convient en effet que les applications patrimoniales comme MS Word offrent les bonnes «API» d'inspection en sorte d'élaborer et de maintenir l'état perçu en question.

```
<auraTask id="34">
  <preferences>
    <service template="default" id="1"/>
    <service template="default" id="2"/>
  </preferences>
  <service type="play Video" id="1">
    <settings mute="true"/>
  </service>
  <material id="11">
    <state>
      <video state="stopped" cursor="0"/>
      <position xpos="645" ypos="441"/>
      <dimension height="684" width="838"/>
    </state>
  </material>
  <service type="edit Text" id="2">
    <settings>
      <format overtype="0"/>
      <language checkLanguage="1"/>
    </settings>
  </service>
  <material id="21">
    <state>
      <cursor position="31510"/>
      <scroll horizontal="0" vertical="7"/>
      <zoom value="140"/>
      <spellchecking enabled="1" language="1033"/>
      <window height="500" xpos="20" width="600" mode="min" ypos="100"/>
    </state>
  </material>
  <configuration name="all" weight="1.0">
    <service id="2">
      <uses materialId="21"/>
    </service>
    <service id="1">
      <uses materialId="11"/>
    </service>
  </configuration>
  <configuration name="only video" weight="0.7">
    <service id="1">
      <uses materialId="11"/>
    </service>
  </configuration>
</auraTask>
```

Figure III-9 Un modèle de tâche Aura [Extrait de Sousa05, p. 29].

```
<task state="pending" id="34">
  <description>
    <name>review semifinals game</name>
    <notes>commentary on the European Soccer Championship games
           for the company newsletter</notes>
    <collaborators>Barney;</collaborators>
  </description>
  <history due="2/07/04" created="30/06/04 9:51 PM">
    <accessed at="home" stop="30/06/04 10:32 PM" start="30/06/04 9:53 PM"/>
    <accessed at="office" stop="1/07/04 10:03 AM" start="1/07/04 9:27 AM"/>
  </history>
  <links>
    <link label="team A previous game" tId="27"/>
    <link label="team B previous game" tId="23"/>
  </links>
</task>
```

Figure III-10 Exemple de méta-desripteur pour la tâche Aura 34 [Extrait de Sousa05, p. 38].

Le modèle de la Figure III-9 sert à l'infrastructure Aura pour la migration de tâche entre environnements. Comme le montre la Figure III-10, ce modèle est complété par un méta-descripteur orienté utilisateur dont certains champs, comme le nom (« review semifinals game ») et date de terminaison prévue (« due = 2/07/04 »), sont spécifiés par l'utilisateur. D'autres, comme

l'historique (lieu d'accès « at home », « at office »), sont maintenus par Aura.

La Figure III-8 montre les principes de l'architecture sur deux nœuds (ou environnements). La Figure III-11détaille l'architecture d'Aura en un environnement selon le modèle composant-connecteur.

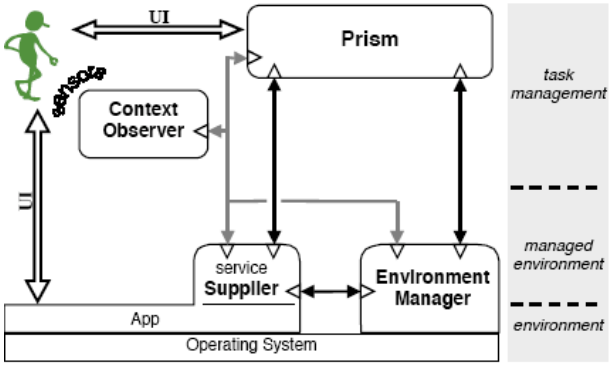


Figure III-11 Les composants d'Aura pour un environnement. [d'après Sousa05, p.42].

L'utilisateur interagit avec les applications natives App via leur IHM native. Il interagit aussi avec Aura via une méta-IHM pour définir ses tâches et exprimer ses préférences. La Figure III-12 en donne un exemple. Cette méta-IHM est reliée au composant Prism qui est le représentant de l'utilisateur (proxy).

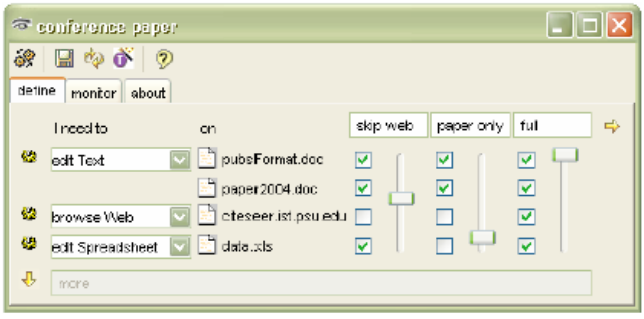


Figure III-12 Méta-IHM de Aura pour définir une tâche incluant une édition de texte (« edit text » avec les « materials » pubsFormat.doc et paper2004.doc), l'utilisation du web sur citeseer et d'un tableur. La partie droite de l'écran permet de spécifier des préférences.

Prism construit et maintient un modèle de tâche et son méta-descripteur conformément aux spécifications de l'utilisateur (comme illustré par les Figure III-9 et Figure III-10). Il suspend et relance les tâches à la demande explicite de l'utilisateur ou lorsque le « context observer » signale le départ/l'arrivée de l'utilisateur dans l'environnement. Les préférences de l'utilisateur sont traduites en QoS à l'adresse des suppliers (fournisseurs) de

service, wrappers chargés d'interfacer les applications natives « App ». Le « management environment » quant à lui, trace la disponibilité des fournisseurs et met en correspondance les demandes de services et les préférences issues de Prism avec les fournisseurs effectifs. Cette mise en correspondance pose le problème d'alignement entre les espaces de nom utilisés respectivement par Prism et les fournisseurs.

3.2.2 AURA en synthèse

Dans mon espace problème (voir Figure III-13), Aura traite de la redistribution au niveau de granularité « total » : l'unité la plus fine de migration est la tâche qui elle-même correspond à une, voire à plusieurs, applications. Une application (ou configuration d'applications) peut être remplacée par une autre. S'il y a redistribution fine de l'IHM, remodelage ou multimodalité, c'est uniquement le fait des applications natives. L'état de reprise dépend de la finesse des inspections (API) permises par les applications. Il est au moins du niveau tâche. Aura a une bonne couverture du contexte d'interaction en termes de plate-forme (mais pour les services logiciels seulement), utilisateur (en termes d'expression des préférences), et environnement physique (avec le context observer, mais ce composant est peu décrit dans les travaux de Sousa). Aura est clairement inter-espace technologique, mais à gros-grain. Comme le montre la Figure III-12, Aura inclut une méta-IHM de type négociation.

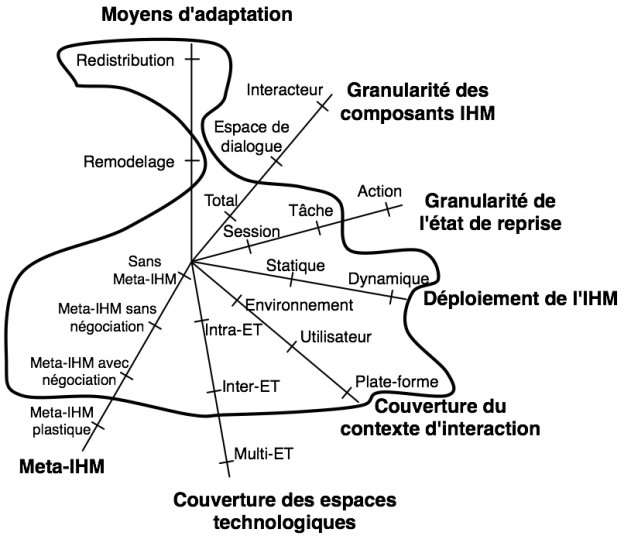


Figure III-13 Aura dans l'espace problème de la plasticité.

En somme, Aura est un middleware dont la force est de permettre à l'utilisateur de retrouver, à moindre effort et selon ses préférences, son aura d'informations alors qu'il change d'environnement. L'unité de migration n'est pas l'application ou partie d'application, mais le desktop ou espace de travail au sens de Rooms [Card87]. Avec ICrafter, l'espace de travail est celui

d'une salle interactive, et cette fois-ci le problème central traité est celui du remodelage.

4. Infrastructures et remodelage d'IHM

Dans cette catégorie d'infrastructure, j'ai retenu *Eloquence* pour sa capacité à générer des IHM multimodales en sortie et *ICrafter* pour sa vision informatique ambiante.

4.1 Eloquence

Eloquence [Rousseau06] désigne un ensemble d'outils pour concevoir, générer et exécuter des IHM multimodales de sortie capables de remodelage à l'exécution. La Figure III-14 illustre les capacités d'Eloquence.

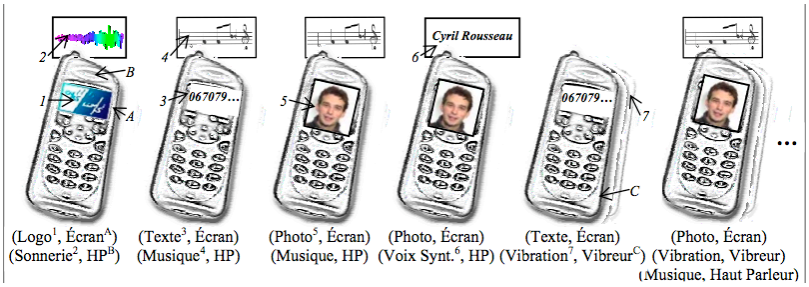


Figure III-14 Génération de présentations multimodales exprimant l'arrivée d'un appel téléphonique en fonction du contexte d'interaction (d'après [Rousseau06], p.141).

4.1.1 Analyse

L'architecture de l'infrastructure d'exécution reflète un processus de génération organisé en 4 étapes (voir Figure III-15) :

(1) Quelles informations présenter ? Ces informations sont fournies au Gestionnaire des Présentations Multimodales (GPM) par le Contrôleur de Dialogue (au sens de Arch). Le GPM les découpe en unités informationnelles élémentaires. Par exemple, l'« Appel de Cyril Rousseau » en provenance du Contrôleur de Dialogue est découpé en deux unités informationnelles élémentaires : « Appel » et «appelant».

(2) Quelles modalités choisir pour chacune des unités élémentaires d'information ? La réponse revient au moteur d'allocation qui, à partir des informations contextuelles maintenues par le serveur de contexte et d'une base de règles comportementales[17], détermine le couple « modalité-média » qui

[17] Exemples de règle (spécifiées par avance par le concepteur d'IHM) : a) si l'unité informationnelle est un appel et si la batterie est faible, alors exclure la

convient (voire plusieurs couples si la redondance est recommandée). Par exemple, le couple « musique-haut parleur » pour dénoter l'événement d'appel et le couple « photo-écran » pour l'unité élémentaire « appelant ». Les modalités sont ensuite recomposées pour reconstituer l'information initiale provenant du Contrôleur de Dialogue. Le Gestionnaire de Présentation Multimodale et l'allocateur de modalité couvrent le niveau Présentation Logique de Arch.

(3) Comment instancier les modalités choisies ? Nous entrons-là dans le niveau Présentation Physique de Arch. Le Gestionnaire de Présentation Multimodale fait appel au moteur d'instanciation qui, aidé des informations contextuelles, décide de l'incarnation des modalités : par exemple la modalité photo est instanciée par la photo de l'appelant complétée d'attributs de niveau lexical (couleur ou Noir et Blanc par exemple) et la musique par La Marseille jouée selon un certain rythme. Ces informations sont fournies au moteur de rendu pour générer l'IHM finale sur les ressources d'interaction de sortie cibles (appelées médium dans la terminologie de Eloquence).

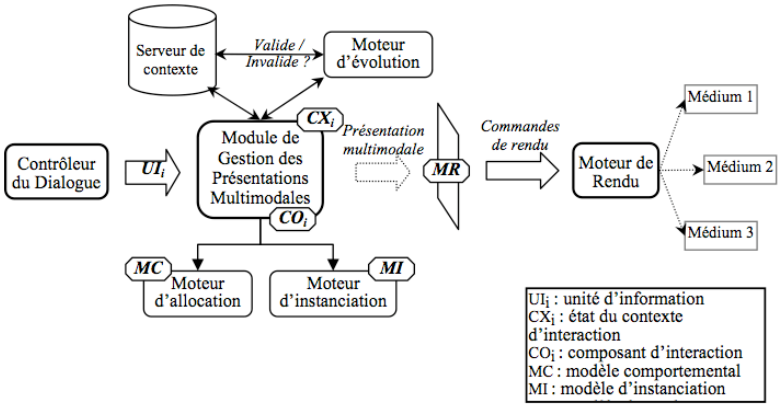


Figure III-15 L'architecture de l'infrastructure d'exécution de Eloquence. (d'après [Rousseau06], p. 156).

(4) Quand l'adaptation doit-elle avoir lieu ? Le Gestionnaire de Présentation Multimodale mémorise la liste des modalités actives. Lorsque le contexte d'interaction change, cette liste est inspectée. Le processus de regénération est repris si l'une d'entre elles n'est plus valide.

modalité photographie et exclure le médium vibreur. b) si l'unité d'information élémentaire est une identité d'appelant, alors utiliser si possible une modalité analogique (NB : une photo de l'appelant est une modalité analogique au sens de Bernsen [Bernsen93]).

4.1.2 Eloquence en synthèse

Dans mon espace problème, Eloquence traite du remodelage multimodal (avec éventuellement la Redondance et l'Equivalence des propriétés CARE). Le remodelage multimodal se pratique au niveau des interacteurs avec reprise au niveau de l'action (mais ce dernier aspect n'est pas clairement exposé dans les travaux publiés). Le contexte d'interaction semble couvrir les trois dimensions – environnment, utilisateur et plate-forme bien que l'infrastructure d'acquisition du contexte ne soit pas détaillée. L'adaptation se fait sans Méta-IHM et la couverture des espaces technologiques est au mieux inter-ET.

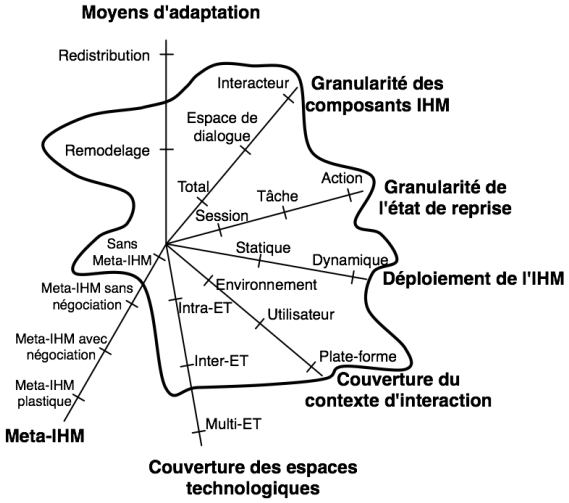


Figure III-16 Eloquence dans l'espace problème.

4.2 ICrafter

Contrairement à Eloquence, ICrafter reste centré sur les IHM graphiques conventionnelles de type WIMP. Mais comme i-LAND, ICrafter [Ponnekanti01] cible les salles interactives sans toutefois s'intéresser à la dimension collaborative ni, contrairement à Aura à la reprise de tâche utilisateur suite à un changement de salle. ICrafter vise trois objectifs principaux : l'adaptabilité des IHM au contexte d'interaction (et notamment à la diversité des salles interactives – configuration physique et hétérogénéité des dispositifs d'interaction), la capacité de déploiement de nouveaux services et la capacité d'agrégation de services.

4.2.1 Analyse

Dans la terminologie de ICrafter, un *service* désigne une application (par exemple, un navigateur Web, MS PowerPoint), mais aussi des dispositifs utilitaires physiques comme une lampe, un scanner, un vidéoprojecteur. L'utilisateur interagit avec les

services au moyen d'*appareils*[18] (un laptop, un PDA). C'est sur les appareils que sont exécutées les IHM des services. L'*agrégation de services*, telle que l'entend ICrafter, rappelle la notion de tâche Aura sans toutefois véhiculer la QoS centrée utilisateur : alors qu'Aura choisit dynamiquement les services les mieux adaptés aux préférences de l'utilisateur, dans ICrafter ce choix revient à l'utilisateur par le biais de la méta-IHM de la Figure III-17. La sélection du bouton « Show interface » retourne à l'utilisateur une IHM agrégat, union des IHM des services choisis.

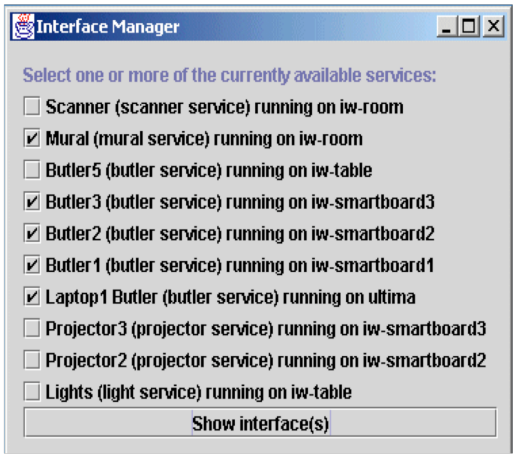


Figure III-17 Méta-IHM permettant à l'utilisateur de choisir les services dont il a besoin parmi les services actuellement disponibles dans la salle interactive.

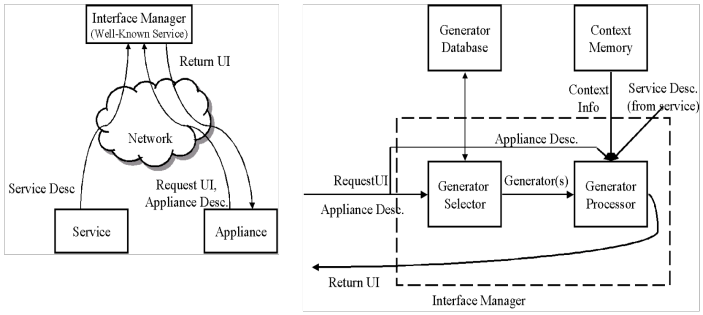


Figure III-18 Le Gestionnaire d'Interface dans ICrafter. La partie (a) représente les relations entre ce gestionnaire et les autres principaux acteurs logiciels. La partie (b) détaille la structure interne du Gestionnaire d'Interface.

La génération d'IHM est organisée selon l'architecture logicielle de la Figure III-18-a. Le Gestionnaire d'Interface (Interface Manager) en est l'élément central. À la réception de la requête

[18] En anglais, *appliances*.

initiée par l'utilisateur depuis l'appareil où s'exécute la méta-IHM de la Figure III-17, le Gestionnaire d'Interface sélectionne le générateur d'IHM idoine et renvoie à l'appareil demandeur l'IHM produite. La Figure III-18-b détaille la structure interne du Gestionnaire d'Interface.

Comme le montre la Figure III-18-b, les générateurs sont répertoriés dans une base de données centrale. À la réception d'une requête de génération d'IHM, le « Sélecteur de Générateur » y choisit le générateur adéquat qu'il transmet à un « Processeur de Générateur » chargé de mettre en œuvre la génération proprement dite. Le processus de génération s'appuie pour cela sur les connaissances suivantes :

- Description de l'appareil à l'origine de la requête. Cette description inclut notamment l'ensemble des langages et interprètes exécutables sur cet appareil (par exemple, html et un navigateur Web). La description est transmise lors de la requête initiale faite au Gestionnaire d'Interface ;

- Définition des services logiciels disponibles (en gros, leur API). Les services disposent d'un médium de diffusion[19] pour annoncer périodiquement leur présence ;

- Description de la configuration de la salle interactive : localisation physique et dimensions des différents dispositifs utilitaires équipant la pièce (par exemple, localisation des sources lumineuses et des tableaux), informations d'ordre sémantique sur ces utilitaires (« Ecran Z est l'écran central ») ou encore expression de relations entre ces utilitaires (« Projecteur X projette sur Ecran Y »). Cette description est répertoriée dans une mémoire commune de type t-uple space appelée « Context Memory ».

Toutes ces connaissances sont représentées dans des langages « maison » non-standard, mais fondés sur XML (par exemple, SDL pour la description des services). La Figure III-19 illustre le processus de génération d'une IHM pour le service « Projecteur » demandé par l'utilisateur depuis un appareil équipé d'un navigateur Web. La description du service « projecteur » indique l'existence de l'opération « input » dont l'exécution permet de changer l'allocation du projecteur vidéo à l'un des ordinateurs connectés (partie a). La mémoire de contexte (partie b) décrit ProjectorService1, un exemplaire de service « projecteur ». Le sélecteur de générateur choisit le générateur d'IHM adapté aux

[19] « Broadcast medium » dans le texte original. C'est un médium sur lequel toutes les entités logicielles sont capables d'écouter. Ainsi, un message émis par un service peut-être reçu par tous les générateurs.

services de type « projecteur » et lui passe le paramètre $serviceName (partie c). Au moyen de ce paramètre, le générateur récupère l'information de configuration dans la mémoire de contexte et produit le code html de la partie d. Ce code est transmis à l'appareil à l'origine de la requête. Son interprétation par le navigateur de l'appareil engendre l'IHM finale de la partie e.

La partie e de la Figure III-19 montre un exemple de générateur d'IHM. Tous les générateurs disponibles dans ICrafter sont exprimés sous forme de template qui incluent des scripts Python ou Tcl pour accéder à la mémoire de contexte et aux descriptions de service. Quand le générateur est exécuté, les scripts sont remplacés par leurs résultats de sortie. Les templates peuvent être génériques ou spécifiques : un template générique est capable de produire une IHM pour plusieurs classes de services mais pour une seule classe d'appareils cibles. Un template spécifique produit une IHM pour une seule classe de services, mais pour des classes d'appareils cibles éventuellement distinctes.

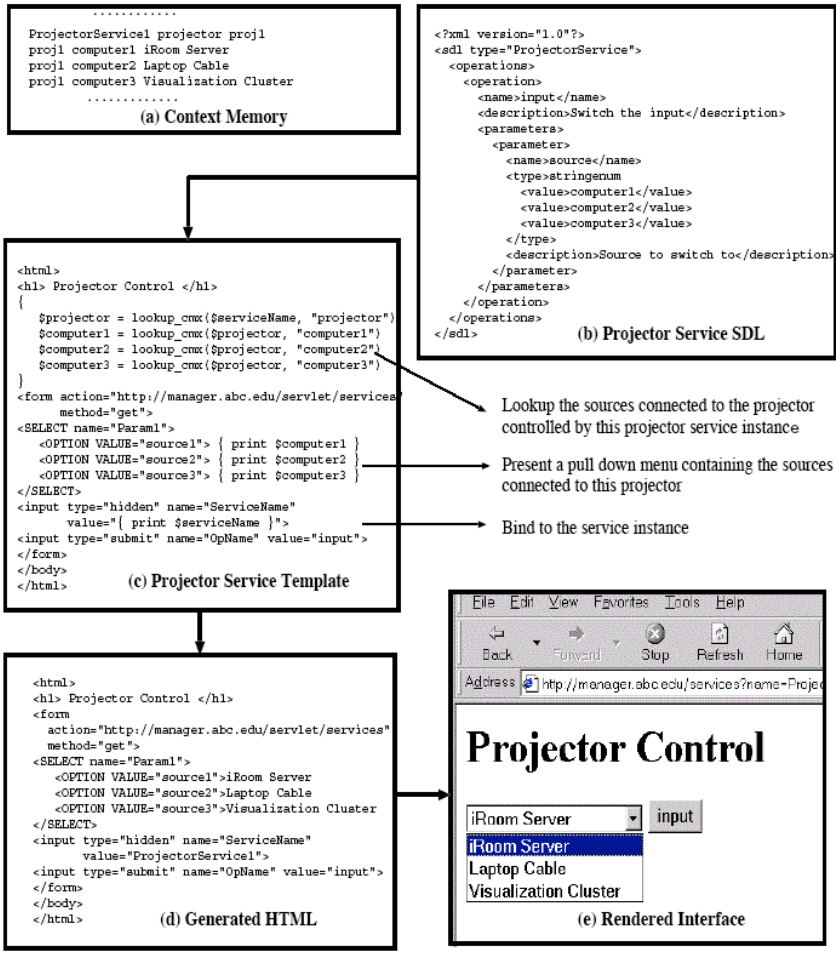


Figure III-19 Illustration du processus de génération d'IHM dans ICrafter pour un service « projecteur » demandé depuis un appareil équipé d'un navigateur Web [D'après Ponnekanti01, p.11].

L'exemple de la Figure III-19 correspond au cas où l'utilisateur a choisi un seul service. Si comme le montre la Figure III-17, l'utilisateur retient plusieurs services, et si, parmi les services sélectionnés certains sont du type producteur et consommateur (de données), le Gestionnaire d'Interface présente également à l'utilisateur l' « IHM des flux de données »[20] de la Figure III-20 qui lui permet de lier les services producteurs aux services consommateurs.

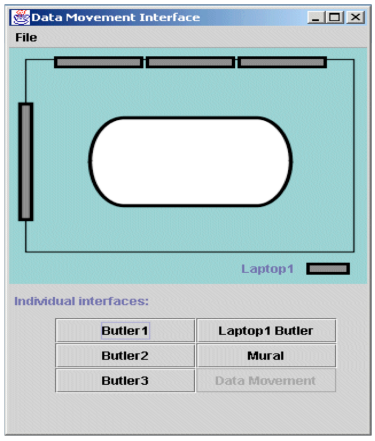


Figure III-20 Méta-IHM pour spécifier les flux de données entre services de type producteur-consommateur.

L'IHM des flux de données montre une vue de dessus de la salle interactive et localise sur cette vue la position des différents services qui intéressent l'utilisateur. Par glisser-déposer, l'utilisateur configure les connexions entre les sorties des services producteurs et les entrées des services consommateurs.

4.2.2 ICrafter en synthèse

La Figure III-21 situe ICrafter dans mon espace problème. ICrafter ne traite pas la redistribution d'IHM puisque le processus de génération d'IHM cible un appareil à la fois – celui d'où émane la requête de génération. Le remodelage d'IHM est entièrement externe aux services. L'avantage est de faciliter le déploiement de nouveaux services par réutilisation de générateurs génériques, mais aussi la génération d'IHM agrégat dans laquelle figurent des informations contextuelles (comme la représentation de relations spatiales entre des services physiques). L'existence de générateurs spécialisés permet d'envisager du remodelage à tous les niveaux d'abstraction, y compris du remodelage intermodal. Inversement, les services qui incluent leur propre IHM (tel MS Word) échappent aux mécanismes de remodelage de ICrafter.

[20] Data movement UI.

ICrafter ne traite pas le problème de la reprise : la granularité de l'état de reprise est celui de la session. Toutefois, le déploiement d'IHM est commandé dynamiquement sous le contrôle de l'utilisateur grâce à une méta-IHM de négociation générée de la même façon que les autres IHM. Il s'agit donc de méta-IHM plastique (mais remodelage seulement) avec adaptation à l'environnement physique (grâce à la mémoire de contexte) et à l'appareil cible. Deux appareils pouvant offrir des espaces technologiques différents (rappelons que la description d'un appareil inclut la liste des langages d'expression des IHM avec leur interprète), les IHM générées via ICrafter peuvent être exprimées dans des espaces technologiques différents. Sur ce point, ICrafter est de niveau inter-ET.

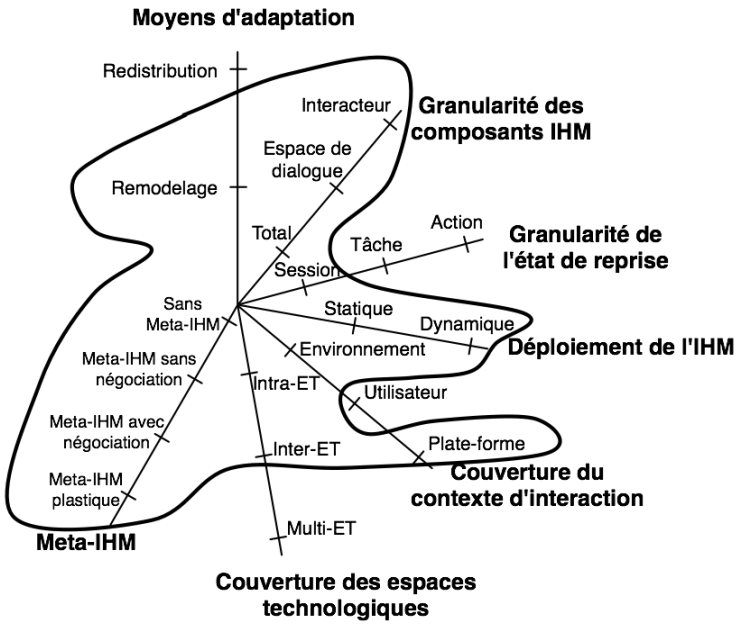


Figure III-21 Projection d'ICrafter dans l'espace problème.

5. Infrastructure et agrégation de services

Nous avons vu rapidement quelques éléments sur l'agrégation de services avec la présentation des systèmes Aura et ICrafter. Dans Aura, l'agrégation de services est véhiculée par la notion de tâche, mais Aura ne traite pas le problème de la génération d'IHM pour ces services agrégés : chaque service dispose de sa propre IHM indépendante de celle des autres. ICrafter aborde cette question pour des dispositifs connectables selon la relation producteur-consommateur. Par exemple, la sortie d'une caméra est couplée à l'entrée d'une imprimante pour fournir un nouveau service : l'impression automatique de photos. La génération de l'IHM de services agrégés se fait soit par l'appel à un « agrégateur » générique (qui ne considère que les fonctions communes aux deux

services, ignorant les spécificités des services unitaires), soit un agrégateur spécifique qu'un programmeur se doit de produire.

L'approche choisie dans *Huddle* [Nichols06-1] vise davantage d'automatisation que ICrafter. Toutefois, Huddle fait des hypothèses sur l'existence d'une description des connexions physiques entre les dispositifs présents dans l'environnement. *Speakeasy* [Edwards01] est susceptible de répondre à cette limitation.

5.1 Speakeasy

L'originalité principale de Speakeasy tient au concept d'*informatique recombinante*[21] [Edwards01]. Cette notion née du constat par [Edwards01] qu'en informatique ambiante, il « n'est pas raisonnable d'attendre que chaque dispositif ait une connaissance préalable de tous les autres types de dispositifs, services logiciels ou applications qu'il sera susceptible de rencontrer »[22]. Speakeasy explore une solution, l'informatique recombinante, destinée à permettre à des entités d'interopérer sans avoir de connaissance préalable les unes sur les autres, de manière à favoriser *l'effet réseau* [Economides96].

5.1.1 Analyse

Informatique recombinante et effet réseau

D'après [Edwards01], la meilleure illustration de l'effet réseau est le téléphone : « Mon téléphone à cadran datant des années 1975 fonctionne toujours parfaitement bien et n'a subi aucune mise à jour. Malgré ce manque de maintenance et d'administration de ma part, je suis toujours en mesure de passer un coup de téléphone vers n'importe quel endroit du monde. De nouvelles fonctions, comme la mise en attente, sont aujourd'hui disponibles sans que j'aie eu autre chose à faire que d'en faire la demande. Et je peux, de manière transparente, utiliser le même dispositif pour me connecter à d'autres qui n'existaient même pas lorsque mon téléphone a été construit, comme, par exemple, des téléphones mobiles qui existent sur une variété de réseaux mondiaux différents et qui utilisent une variété de protocoles différents »[23].

---

[21] Recombinant computing.

[22] In the future world of ubiquitous computing, it is unreasonable to expect that every device will have prior knowledge of every other type of device, software services, or application that it may encounter.

[23] My rotary dial phone, circa 1975, still functions perfectly well and has never had an upgrade. Despite this lack of administration and maintenance on my part, I am able to place calls anywhere in the world. New functions, like call waiting, become available to me without me having to do anything other than request them. And I can transparently use this same device to connect to

Ainsi, l'effet réseau est manifeste quand un élément du système voit sa valeur augmenter par le fait d'actions en d'autres endroits du système. Comme [Edwards01] pense que cet effet réseau est incontournable pour rendre possible la vision de l'informatique ambiante, l'objectif de l'informatique recombinante est de permettre ce même type de phénomène dans un domaine bien plus riche que celui de la téléphonie, composés de dispositifs numériques et logiciels arbitraires.

L'informatique recombinante s'inspire d'approches connues et éprouvées : les approches à composants pour l'encapsulation et la réutilisabilité, les approches à services pour la découverte automatique ou encore des paradigmes de la programmation orientée-objet. Si ces approches sont nécessaires, les auteurs de Speakeasy montrent qu'elles ne sont pas suffisantes. En particulier, les approches à composants classiques ne sont pas adaptées à l'interconnexion de composants logiciels arbitraires car elles nécessitent que les composants aient une connaissance préalable non seulement des interfaces de leurs partenaires, mais également de leur sémantique. Les mécanismes de découverte automatique ne sont pas totalement adaptés à la question. En effet, les approches classiques sont capables de localiser des services distants, mais si le pilote qui permet d'accéder à ces services n'est pas disponible localement, le lien ne peut être établi. De plus, [Newman02] ajoute une contrainte importante : les entités produites sont exposées à l'utilisateur final afin qu'elles soient utilisées et configurées par lui de manière souple et opportuniste.

Pour répondre à ce concept d'informatique recombinante, Speakeasy s'organise autour de quatre concepts : les composants, les connexions, le contexte, et le contrôle.

Le concept de composant dans Speakeasy

Dans Speakeasy, un composant est une entité qui peut être connectée à d'autres composants et être utilisée par d'autres composants ou par des logiciels compatibles avec Speakeasy. Par exemple, des microphones, des imprimantes ou des caméras, des services logiciels comme un serveur de fichier, une base de donnée ou un client de messagerie instantanée, voire le carnet d'adresses d'un PDA sont, au sens de Speakeasy, des composants. *Par projection dans ma taxonomie, le niveau de granularité qui correspond à ce type de composant est le niveau « total », le plus bas de cette dimension*.

devices that were not even in existence when my phone was built, such as cellular telephones that exist on a variety of worldwide networks and use widely varying protocols.

Les composants Speakeasy sont décrits en termes d'interfaces programmatiques recombinantes. Une interface recombinante est une interface programmatique qui spécifie la manière syntaxique dont un composant interagit avec un autre, en laissant à l'écart toute préoccupation d'ordre sémantique. Ainsi, une interface recombinante est indépendante du domaine ou du métier du composant. Elle permet alors de constituer le fondement d'un langage d'interaction entre composants indépendants du domaine d'application des composants.

Selon [Edwards01], cette approche est intéressante car un ensemble réduit d'interfaces recombinantes qui autorise des composants arbitraires à interagir, est une condition nécessaire à la réalisation de l'effet réseau. Cependant, pour qu'une connexion entre composants ait un sens, des connaissances sur la sémantique des composants sont indispensables. La grande originalité de l'informatique recombinante repose sur la place accordée à l'utilisateur final qui est chargé d'apporter la connaissance sémantique sur les composants à assembler. *Ainsi, dans cette approche, c'est l'humain qui est en mesure de décider de l'opportunité et du moment où tels ou tels composants doivent interagir*.

Cependant, cette approche ne s'affranchit pas d'une limitation évidente : pour certaines interactions, le composant émetteur et le composant récepteur doivent partager un type de donnée particulier. Par exemple, une imprimante uniquement capable d'interpréter du PostScript ne pourra pas imprimer directement des images issues d'une webcam qui produit des clichés au format JPEG. Le type de données pris en charge étant exprimé au niveau de l'interface programmatique, celle-ci n'est donc pas entièrement indépendante des aspects sémantiques. Les auteurs de l'étude argumentent que si ce type d'accord sur des types de données limite l'universalité des interfaces recombinantes, celles-ci apportent néanmoins un bénéfice significatif, d'autant plus qu'il est possible de dépasser cette limitation par l'insertion, dans les assemblages, de composants intermédiaires chargés de réaliser les conversions de types nécessaires.

Le concept de connexion

Le second concept de Speakeasy est celui de connexion. Deux (ou plus) composants Speakeasy se connectent l'un à l'autre via leurs interfaces recombinantes. Ce lien leur permet d'échanger des données selon un modèle à flux : l'un des deux incarne l'émetteur des données, l'autre (ou les autres) joue le rôle de récepteur. Comme la diversité des données échangeables entre des classes de composants arbitraires est infinie, il n'est pas possible ni de déterminer a priori un protocole de transmission d'information unique valable pour toutes les situations ni d'implémenter une

entité logicielle capable de mettre en œuvre tous les protocoles existants.

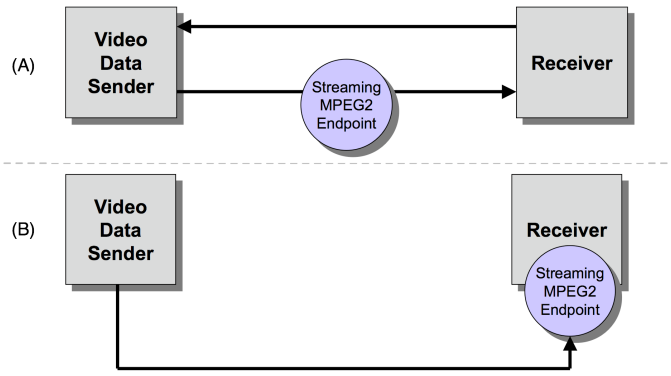


Figure III-22 La réalisation d'une connexion dans Speakeasy.

L'approche adoptée dans Speakeasy consiste à permettre à l'émetteur d'envoyer aux récepteurs le gestionnaire de transmission approprié[24], destiné à prendre en charge le transfert des données et à les délivrer au récepteur dans le format idoine (voir Figure III-22). Ainsi, il n'est pas nécessaire qu'un récepteur partage un protocole de communication particulier avec l'émetteur. En revanche, cette solution n'est envisageable que s'il est possible d'envoyer sur le récepteur le code du gestionnaire de transmission. Dans Speakeasy, ceci est rendu possible par l'utilisation de Java. Tous les composants Speakeasy doivent soit, être écrits en Java, soit, être manipulables via un mandataire[25] écrit en Java. *En ce sens, Speakeasy se classe au niveau « intra-espace technologique » de ma taxonomie.*

Contexte et contrôle

Dans Speakeasy, le concept de contexte désigne un ensemble d'attributs qui caractérisent une instance de composant Speakeasy. Par exemple, il peut s'agir de son nom, de sa localisation, de son propriétaire, de sa version, etc. Ces informations de contexte sont principalement destinées à l'utilisateur final, de manière à l'aider à faire un choix entre plusieurs composants possibles. Contrairement à la notion de contexte exprimée dans ma taxonomie, il ne s'agit pas d'adapter un composant au contexte de l'interaction. Au contraire, en fonction du contexte d'interaction courant, l'utilisateur constituera lui-même un assemblage de composants adéquats. *Au sens de ma taxonomie, Speakeasy proprement dit ne couvre donc aucun des trois niveaux de l'axe « couverture du contexte de l'interaction ».*

[24] « sender transfers a *source-provided endpoint* to its receiver. »

[25] proxy

Le concept de contrôle correspond à l'interaction entre l'humain et le système à deux niveaux distincts : celui de la configuration des assemblages et celui de l'utilisation des composants. Pour que les IHM dédiées au contrôle soient disponibles sur une grande variété de dispositifs, la solution retenue par Speakeasy est celle du navigateur. Le contrôle des assemblages s'effectue grâce au navigateur Speakeasy (voir Figure III-23) selon deux modes d'interaction : le mode « connexion directe » (Figure III-23, partie A) et le mode « patron de tâche »[26] (Figure III-23, partie B).

Le mode « connexion directe » permet à l'utilisateur de trouver et de sélectionner les composants disponibles selon plusieurs critères (par localisation, par propriétaire, par tâche, etc.). Grâce à cette interface, l'utilisateur peut créer ou détruire à sa guise des connexions entre n'importe lesquels de ces composants. Il peut également accéder aux IHM de ceux-ci, soit dans le but de les paramétrer, soit dans le but de simplement les utiliser.

Le mode « patron de tâche » consiste à guider l'utilisateur au moyen d'un patron de configuration. Dans l'exemple de la Figure III-23 (B), le patron correspond à une tâche de présentation. Pour guider l'utilisateur, le patron de configuration prédéfinit un assemblage-type dont il ne reste qu'à préciser les instances de composants à utiliser pour chaque connexion préconisée par le patron.

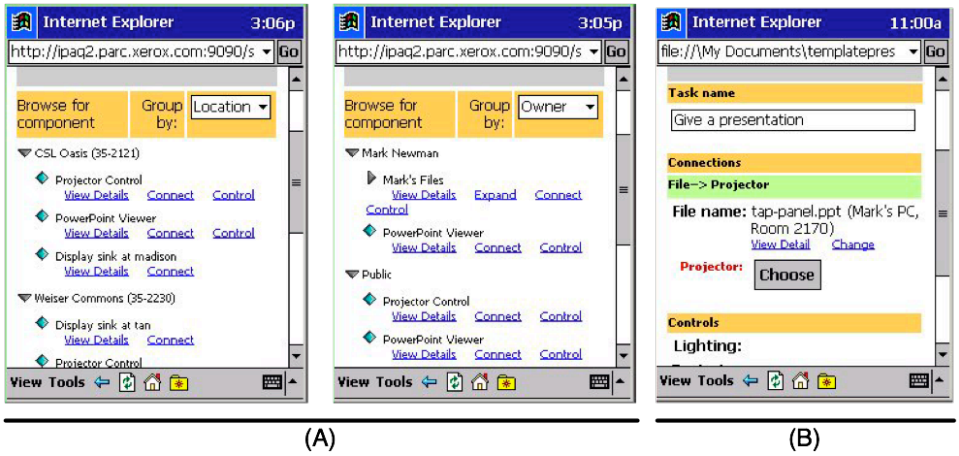


Figure III-23 Le navigateur Speakeasy : méta-IHM de configuration.

Dans les deux modes de configuration, l'humain garde le contrôle sur le système. Au sens de ma taxonomie, ces interfaces utilisateur correspondent au concept de méta-IHM. *La place importante réservée aux choix de l'utilisateur classe ce système au niveau méta-IHM avec négociation.* Enfin, la notion de contrôle, dans Speakeasy s'étend à l'interaction entre l'humain et chacun des

[26] Task-oriented template

composants mis en œuvre dans un assemblage. Ces composants sont capables d'exposer à l'utilisateur leur propre IHM.

De la même manière que pour les IHM de configuration, l'IHM propre aux composants est rendue perceptible à l'utilisateur dans un navigateur. Comme le montre la Figure III-24 dans sa partie (A), le navigateur Speakeasy demande au composant son interface utilisateur. Celle-ci est alors envoyée au navigateur pour qu'elle y soit affichée, permettant l'interaction avec l'utilisateur. Dans l'exemple de la Figure III-24, il s'agit d'une IHM de paramétrage d'un diffuseur de vidéo. La partie (B) de cette figure illustre la possibilité offerte par Speakeasy de mémoriser les paramètres de configuration d'une instance de composant donnée pour une réutilisation future. Ces paramètres, propres à chaque composant, sont stockés par le composant dans le format de son choix. Seule une référence permettant de retrouver ces réglages est envoyée au navigateur Speakeasy qui la sauvegardera.

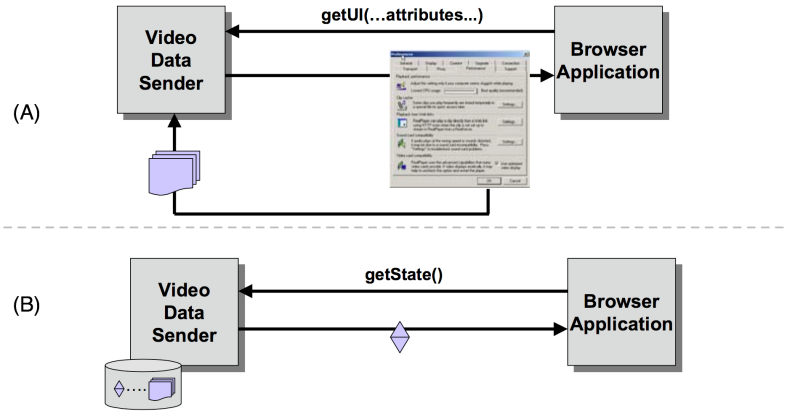


Figure III-24 Les composants Speakeasy et la gestion de leur IHM.

Cette étude rapide du mode de rendu des IHM des composants dans Speakeasy permet de mettre en lumière l'absence de projection sur l'axe « moyen d'adaptation » de ma classification. En effet, Speakeasy ne s'intéresse pas à l'adaptation de l'IHM des composants en fonction du dispositif d'interaction sur lequel elle est rendue. Ainsi, aucun remodelage n'est appliqué aux IHM lors de leur rendu par un navigateur Speakeasy. De la même manière, le style de protocole utilisé pour obtenir l'IHM d'un composant, décrit par la Figure III-24, ne permet pas de répartir cette IHM sur plusieurs dispositifs. Aucune redistribution de l'interface utilisateur, pour un composant donné, n'est donc possible.

5.1.2 Speakeasy en synthèse

Comme le résume la Figure III-25, Speakeasy ne couvre qu'une partie réduite de mon espace problème. L'axe des moyens d'adaptation n'est pas traité, la plupart des autres axes le sont aux niveaux les plus bas. Seuls, les axes déploiement de l'IHM et

méta-IHM sont couverts de manière avancée. Cependant, Speakeasy reste un travail de recherche intéressant au regard de ma problématique. En effet, comme dans Speakeasy, la recherche de l'effet réseau semble être une condition de réussite pour la question de la plasticité à l'exécution. Si l'approche de Speakeasy apporte une réponse à cette question pour un espace technologique donné, la question de la plasticité à l'exécution requiert de favoriser l'effet réseau dans un contexte multi-espaces technologiques.

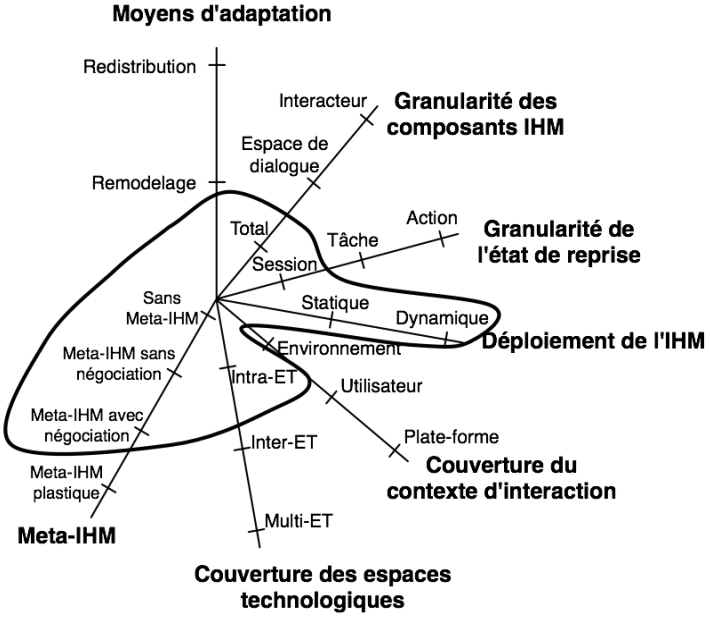


Figure III-25 Projection de Speakeasy dans l'espace problème.

5.2 Huddle

Comme ICrafter, Huddle [Nichols06-1] s'appuie sur un modèle de flux de données, mais s'intéresse au cas particulier de la configuration des appareils multimédia personnels (magnétoscope, TV, Appareil photo, imprimante). On le verra, Huddle n'a pas la généricité sous-jacente de Speakeasy, mais le complète par sa capacité à générer des IHM agrégées.

5.2.1 Analyse

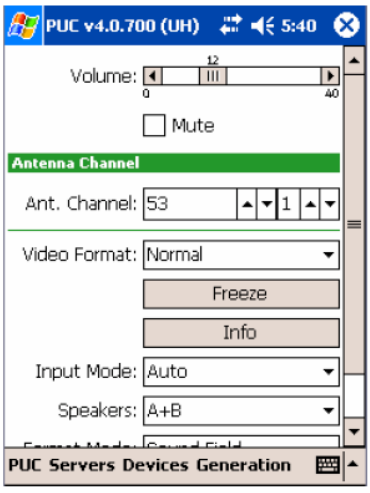
Grâce à une méta-IHM, l'utilisateur spécifie le cheminement souhaité des informations entre les ports d'appareils multimédia : par exemple, connecter le flux image en provenance de l'antenne TV à l'écran de télévision et relier le flux audio de l'antenne aux haut-parleurs de la chaîne HI-FI. Pour générer une IHM agrégée correspondant à une telle configuration, Huddle utilise trois sources d'information :

1. La description des connexions physiques entre les appareils : une spécification XML décrite manuellement que Speakeasy serait susceptible de fournir.

2. La description de chaque appareil dans le langage PUC [Nichols02] complétée par une base de connaissances qui indique les informations fonctionnellement similaires d'un appareil à l'autre. Ces connaissances sont utiles pour refléter des alignements sémantiques dans l'IHM concrète et pratiquer des fusions.

3. La spécification, par l'utilisateur final au moyen d'une méta-IHM, des flux d'information entre les ports d'entrée et de sortie des appareils.

Huddle produit quatre sortes d'IHM agrégées : l'IHM agrégée qui permet de contrôler la configuration en cours de fonctionnement (boutons pour augmenter le volume audio, boutons d'arrêt et de reprise du flux audio-vidéo) (voir Figure III-26-a); l'IHM agrégée pour la spécification des préférences spécifiques à chacun des appareils de la configuration (qualité de l'image pour la vidéo, équilibre des aigus et graves pour le son, etc.) (voir Figure III-26-b) ; l'IHM agrégée pour l'expression de préférences générales (limite parentale) ; l'IHM fusionnée pour donner accès aux fonctions applicables simultanément à tous les dispositifs de la configuration (par exemple, changement d'heure et de langue). Toutes ces IHM sont générées automatiquement, mais les modèles et heuristiques sont clairement influencés par le domaine applicatif.

a) 

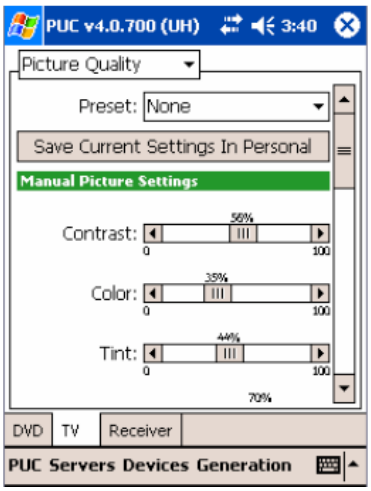
b) 

Figure III-26 a) IHM agrégée correspondant à la configuration de flux suivante : Flux vidéo sur ecran TV, et Flux audio sur des haut-parleurs A+B de l'appareil radio. Les widgets de contrôle du volume audio viennent en premier, suivi du choix de la chaîne. b) IHM agrégée pour contrôle individuel des appareils de la configuration (des onglets permettent de sélectionner l'appareil voulu).

5.2.2 Huddle en synthèse

Comme le montre la Figure III-27, Huddle ne fait que du remodelage, avec une granularité de niveau total puisque toute

l'IHM est (re)générée, avec une reprise au niveau de la session et le déploiement dynamique que permet PUC. Le contexte d'interaction est une adaptation à la plate-forme qui comprend ici le PDA de rendu de l'IHM et les appareils multimédia. L'espace technologique couvert est intra-ET (celui de PUC) mais inclut une méta-IHM de négociation permettant de construire les flux d'information. Au bilan, Huddle aborde un problème important, mais la solution offerte est limitée par son manque de généricité.

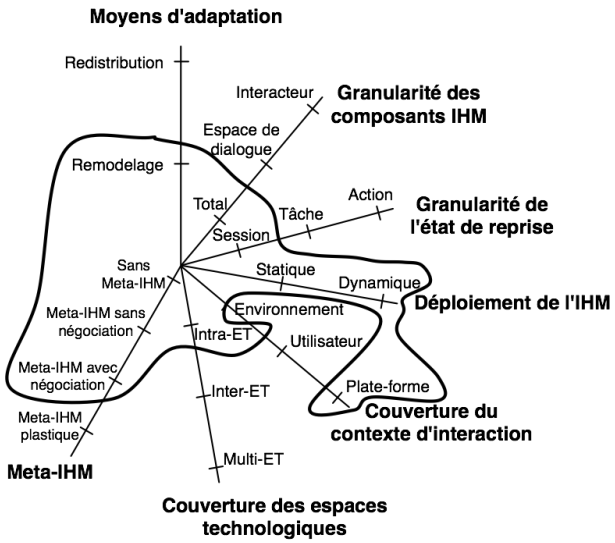


Figure III-27 Projection de Huddle dans l'espace problème.

6. Synthèse

L'analyse de l'état de l'art sur l'adaptation des IHM révèle une grande diversité d'approches, chacune d'elles véhiculant des progrès sensibles au regard de la plasticité des IHM sans qu'aucune ne couvre tous les aspects de mon espace problème.

L'approche dirigée par les modèles convient aux domaines métier pour lesquels il existe une réelle capitalisation des savoir-faire en matière d'IHM. Cette capitalisation s'exprime sous forme de patrons mis en pratique par des assemblages d'interacteurs WIMP de référence. Des boîtes à outils post-WIMP commencent à apparaître, mais leur intégration dans les approches génératives n'a pas été envisagée.

En l'état, l'approche centrée code est incontournable. Nous avons relevé l'absence de gestionnaire de ressources d'interaction adapté aux conditions de l'informatique ambiante. En conséquence, les développeurs de boîtes à outils comme Ubit ou ETk tentent de pallier ce manque, mais au sein d'un espace technologique unique. En tout état de cause, il est des préoccupations comme la reconfiguration, l'agrégation ou la migration dynamiques d'IHM,

qui ne relèvent pas des missions d'une boîte à outils. Ces problèmes récurrents sont généralement traités par des intergiciels ou infrastructures d'exécution.

Nous avons relevé trois classes d'infrastructures selon que la classe de problèmes traités concerne la redistribution (BEACH, Aura), le remodelage (ICrafter, Eloquence) ou l'agrégation d'IHM - un remodelage particulier dont l'importance va croissant (Speakeasy, Huddle). Aucune de ces solutions ne couvre à la fois le remodelage et la redistribution. La raison tient essentiellement aux domaines d'applications qui définissent *de facto* le contour de la solution : BEACH, ICrafter, Huddle traitent de milieux confinés relativement stables et contrôlables ; Aura introduit un concept nouveau (celui d'espace informationnel qui suit l'utilisateur en tout lieu), mais s'appuie sur des applications patrimoniales traditionnelles. Speakeasy, avec sa notion d'informatique recombinante paraît plus ambitieux sans toutefois remplir le requis d'inter espace technologique. Il s'inspire toutefois de la notion de composant et de liaison dynamique introduites en Génie Logiciel. Au chapitre suivant, nous étudions cet apport.

Chapitre IV

Distribution, reconfiguration et hétérogénéité des technologies : apports et limites du Génie Logiciel

Avant-propos

Sur le plan technique, un système interactif plastique est un logiciel distribué, dynamiquement reconfigurable, et capable de traiter avec l'hétérogénéité de son environnement d'exécution. Cette nature logicielle s'inscrit à la croisée de plusieurs domaines de recherche en Génie Logiciel : les intergiciels pour la distribution, les approches à composants et à services pour la reconfiguration dynamique, et l'Ingénierie Dirigée par les Modèles (IDM) et sa notion de transformation pour le traitement de l'hétérogénéité des technologies.

Ce chapitre commence par une revue de l'état de l'art sur les intergiciels et les approches à composants et à services. Je ne reviendrai pas sur l'IDM dont j'ai résumé les avancées et les limites au chapitre précédent sous l'angle des IHM plastiques. Ce chapitre s'achève sur une discussion des limites des solutions du GL au regard de mes requis.

1. Intergiciels et distribution : le problème de leur interopérabilité

Depuis le début des années 1990, les intergiciels constituent la solution la plus communément admise aux problèmes du logiciel réparti. L'objectif d'un intergiciel est d'offrir aux développeurs d'applications réparties un niveau d'abstraction qui masque la distribution des différents constituants d'une application et l'hétérogénéité des matériels, systèmes d'exploitation, et langage de programmation. [Mascolo02] montre que les intergiciels traditionnels, qui ont été conçus et utilisés avec succès dans le cadre de systèmes distribués sédentaires et construits sur des réseaux informatiques fixes, ne sont pas utilisables en l'état dans le domaine de l'informatique mobile, et donc, par extension dans celui de l'informatique ambiante.

Si certains ont été adaptés au cas de l'informatique mobile, notamment CORBA [Haahr99], beaucoup d'autres, comme ReMMoC [Grace03], ont été développés spécialement pour ce domaine. [Roman01] note que le foisonnement des dispositifs et des systèmes pour l'informatique ambiante introduit le problème de l'hétérogénéité des intergiciels. [Mascolo02] montre qu'il sera très difficile d'isoler un intergiciel suffisamment général capable de prendre en compte l'ensemble des possibles de l'informatique ambiante. Il paraît donc inévitable de traiter la question de

l'interopérabilité entre intergiciels. Les propositions les plus abouties dans ce sens concernent les intergiciels réflexifs et les approches par traduction de protocole. J'en étudie à présent un exemple représentatif pour chacune de ces deux classes: ReMMoc [Grace03] et AMIGO [Amigo-D2.1].

1.1 Un intergiciel réflexif : ReMMoC

La capacité réflexive d'un intergiciel se caractérise par la possibilité donnée à une application d'inspecter et de modifier des constituants de l'intergiciel sur lequel elle est construite pour changer le comportement de cet intergiciel [Roman01]. ReMMoC[27] [Grace03] est un intergiciel réflexif dynamiquement reconfigurable conçu pour le développement d'applications dans le contexte très hétérogène de l'informatique mobile.

ReMMoC comprend deux constituants principaux : un environnement de liaison[28] et un environnement de découverte de service[29] (voir Figure IV-1). Le premier fournit les mécanismes pour assurer l'interopérabilité entre différents types d'intergiciels. Le deuxième permet la découverte de services dont l'existence est diffusée par une gamme de protocoles de découverte automatique.

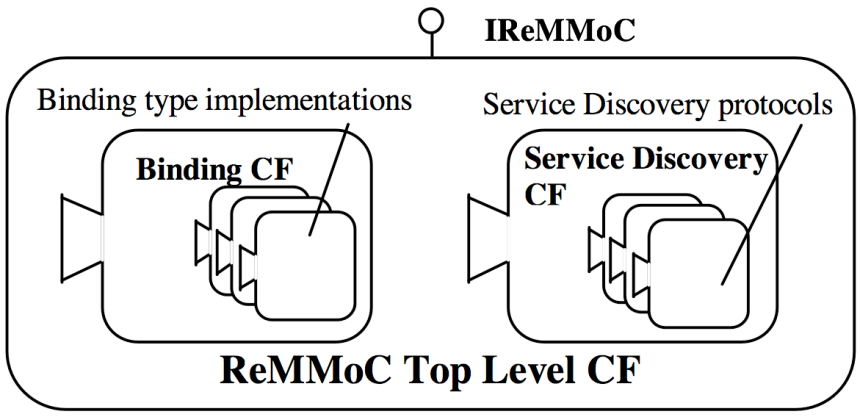


Figure IV-1Vue générale de la plate-forme ReMMoC d'après [Grace03].

L'environnement de liaison se configure par ajouts ou retraits d'implémentation de type de liaison[30]. Chacune de ces implémentations est dédiée à un intergiciel donné. Elle permet d'établir une liaison de communication entre un client développé au-dessus de ReMMoC et un service implémenté au-dessus de

[27] ReMMoC :Reflective Middleware for Mobile Computing

[28] Binding Component Framework (Binding CF)

[29] Service Discovery Component Framework (Service Discovery CF)

[30] Binding type implementations

l'intergiciel ciblé. De la même manière, l'environnement de découverte de services se configure par ajouts ou retraits de protocoles de découverte et offre à un client développé au-dessus de ReMMoC le moyen de découvrir des services annoncés via ces protocoles. D'autre part, ReMMoC fournit une interface programmatique à l'usage de la couche applicative afin que les applications implémentées au-dessus de ReMMoC puissent utiliser directement l'implémentation de type de liaison ou le protocole de découverte de service qui lui convient.

L'une des faiblesses de ReMMoC concerne l'absence de réciprocité dans les mécanismes : s'il est possible, avec ReMMoC, d'implémenter un client capable d'interagir avec des services implémentés avec d'autres intergiciels, il est impossible, avec ReMMoC, de programmer un service qui serait disponible pour des clients implémentés avec d'autres intergiciels. C'est notamment cette limitation que se proposent de dépasser les approches à traduction de protocoles. Les travaux de recherche effectués au sein du projet AMIGO [Amigo04] sont de ceux-là.

1.2 Une approche par traduction de protocole : AMIGO

AMIGO [Amigo04] est un projet européen[31] qui s'inscrit dans le cadre des réseaux domestiques, c'est-à-dire des réseaux informatiques pour la maison. Actuellement, ces réseaux informatiques relient les consoles de jeux ou les différents ordinateurs présents dans la maison entre eux et à Internet. Cependant, selon AMIGO, ces réseaux sont sous-utilisés. De nos jours, les habitations contiennent pléthore de dispositifs devenus numériques (télévision, chaîne HIFI, téléphone, électroménager, etc.) dont le raccordement au réseau domestique démultiplierait les possibilités pour l'utilisateur. Or, les procédures d'installation complexes, le manque d'interopérabilité entre les dispositifs issus de différents constructeurs et l'absence de service intéressant pour l'utilisateur freinent ce genre d'avancée. L'objectif d'AMIGO est de proposer une solution logicielle pour résoudre ces principales difficultés, notamment en proposant une infrastructure logicielle fondée sur la technique de traduction basée sur des évènements[32] et explorée par [Ryan04].

L'étude de [Ryan04] s'intéresse à la question de l'interopérabilité entre les différentes versions d'un même protocole. À partir d'une spécification de protocole, un *générateur* crée une *unité* composée

[31] Amigo : Ambient Intelligence for the networked home environment, European IP project, IST 004182.

[32] Event Based Translation

d'un *analyseur* et d'un *compositeur*[33]. Le processus de traduction est le suivant (voir Figure IV-2) : Un composant A transmet, au moyen de la version X d'un protocole, des données à l'analyseur de l'unité générée pour le protocole X. Cet analyseur est capable de transformer sous forme d'événements ces données entrantes. Ces événements sont transmis via un *mandataire* au compositeur de l'unité générée pour le protocole Y qui va les transformer dans le format de la version Y du protocole. Ce compositeur délivre alors ces données transformées au composant B. Ainsi, bien que le composant A et le composant B ne partagent pas le même protocole de communication, ils peuvent interagir sans que l'un ou l'autre ait eu à subir de modification. De leur point de vue, les mécanismes d'adaptation de protocole sont transparents. Aujourd'hui, cette technique ne semble avoir été appliquée qu'au couple de protocole de communication HTTP 1.0 et HTTP 1.1, c'est-à-dire à deux protocoles très similaires. Dans sa solution technique, AMIGO reprend les concepts introduits par cette étude pionnière en la matière, mais les adapte selon deux axes : améliorer les mécanismes pour traduire des protocoles dont les différences sont plus importantes d'une part, et adapter cette approche pour les protocoles de découverte de services et les protocoles d'interaction d'autre part [Amigo-D2.1].

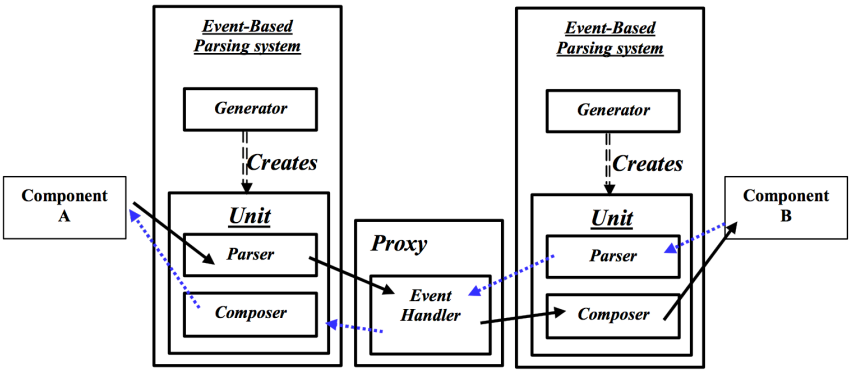


Figure IV-2 L'interaction entre deux composants via le mécanisme de traduction d'évènements de [Ryan04]. Le composant A utilise le protocole UPnP, tandis que le composant B utilise SLP (Schéma issu de [Amigo-D2.1])

Contrairement à ReMMoC, l'approche mise en œuvre dans [Amigo-D2.1] autorise l'existence d'entités qui sont à la fois cliente et service. Mieux, leur solution permet de gérer de manière transparente l'hétérogénéité des espaces technologiques sous-jacents aussi bien du point de vue des services et des clients, mais également pour leurs développeurs. De plus, comme c'est le cas dans notre problématique, leur solution s'inscrit dans le cadre de

[33] Generator, unit, parser, composer

plates-formes constituées de manière dynamique et opportuniste avec des plates-formes élémentaires des plus puissantes au plus légères.

Toutefois, cette approche par traducteur de protocole s'avère gourmande en puissance de calculs, notamment au moment de la génération des traducteurs. Dans un contexte de réseaux domestiques ouverts [Amigo04] ceci ne pose pas de problème car la solution AMIGO ne nécessite pas un déploiement sur l'ensemble de la plate-forme : il suffit au minimum qu'une seule des plates-formes élémentaires du réseau héberge le générateur de traducteur.

Ainsi, comme nous venons de le voir, le monde des intergiciels propose des solutions pour traiter ces questions de distribution du logiciel et de certains aspects de l'hétérogénéité des systèmes. Le traitement des questions relatives à la reconfiguration dynamique des applications passe par l'intégration d'approches complémentaires. Les approches à composants logiciels en font partie.

2. Approches à composants et reconfiguration dynamique

Dans la littérature, le terme « composant » apparaît largement, mais fait référence à des concepts souvent très différents. [Cervantes04] note que cela est probablement dû au fait que « la description des aspects technologiques est souvent privilégiée par rapport à celle des concepts ». Ainsi, d'une manière générale, l'approche à composant vise à construire des applications par l'assemblage de briques logicielles réutilisables. Dans [Cervantes04] un modèle à composant est défini par les « caractéristiques des composants, de leurs assemblages et est accompagné d'un support d'exécution ». Cependant, depuis le milieu des années 1990, des travaux de fond sur cette question ont permis d'identifier les concepts-clé et récurrents de cette approche. Dans cette section, je présente les motivations et les concepts-clé des approches à composants, puis j'étudie plus particulièrement les différentes propositions relatives à la reconfiguration dynamique des systèmes à base de composants.

2.1 Motivation et concepts-clé de l'approche à composants

Les travaux relatifs à l'approche à composants sont innombrables et les détailler tous ici n'est pas envisageable. Je me focalise donc

sur les études qui me semblent les plus représentatives de ces approches.

Historiquement, la notion de composant est issue d'une longue lignée de travaux de recherche sur les architectures logicielles qui commence avec les MIL (Module Interconnection Languages) [Deremer75] en 1975 et qui continue avec les ADL (Architecture Description Languages). D'une manière quasi-générale, tous les travaux relatifs à l'architecture logicielle manipulent deux principaux types d'objets : les entités chargées d'une fonction de calcul – les composants, et celles qui « matérialisent » une relation entre plusieurs entités du type précédent – les connecteurs [Shaw95]. Toutefois, les notions de composant et de connecteur restent floues : il semble qu'il n'existe pas de définition précise et consensuelle.

Néanmoins, pour le concept de composant, la définition de [Szyperski02][34] semble être la plus citée dans la littérature. [Szyperski02] définit un composant comme « une unité de composition uniquement associée à des interfaces spécifiées de manière contractuelle et à des dépendances de contexte explicites. Un composant logiciel peut être déployé de manière indépendante et sujet à composition par des tiers »[35]. Selon [Szyperski02], cette définition met en avant trois caractéristiques : c'est une unité indépendante de déploiement, une unité sujette à composition avec des tiers et une unité qui n'a pas d'état observable de l'extérieur.

2.1.1 Les caractéristiques générales d'un composant

La première caractéristique (unité de déploiement) implique qu'un composant ne peut jamais être déployé partiellement. La seconde caractéristique (unité de composition) justifie l'obligation de spécifications claires et précises des interfaces programmatiques fournies et requises d'une part, et des propriétés requises sur l'environnement d'accueil d'autre part, pour permettre un assemblage des instances de composants par une tierce partie. La dernière caractéristique (pas d'état observable) implique qu'un composant ne peut pas être distinguable d'une copie de lui-même. En d'autre termes, un composant ne peut pas exposer d'attributs à des tiers, exceptés ceux ayant trait à des préoccupations extra-fonctionnelles. Par exemple, si un composant peut exposer un attribut représentant un numéro de série, il ne peut pas exposer un attribut représentant une variable dont la valeur sera modifiée lors

[34] L'ouvrage de 2002 est une seconde édition. La première date de 1997.

[35] « A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties ».

de l'exécution du composant. Ainsi, [Szyperski02] fait la différence entre le composant, code inerte, qui n'a pas d'état propre et son instance, code chargé en mémoire et en cours d'exécution, qui peut avoir un état. La question de la persistance de l'état d'une instance de composant est considérée indépendante du composant. Dans la mesure où elle doit avoir lieu, cette persistance doit être assurée par un tiers, à l'extérieur du composant.

Le concept de composant défendu par [Garlan97] est compatible avec celui de [Szyperski02]. Il représente l'unité primaire de calcul ou de stockage de données dans un système. En guise d'exemple de composants [Garlan97] cite « les clients, les serveurs, les filtres ou les bases de données. » Proche des travaux de [Garlan97], [Shaw95] ne diffère que sur certains points de la terminologie. Selon ces deux travaux de référence, « dans un schéma d'architecture logicielle à base de traits et de boîtes, les composants correspondent aux boîtes ». Quant aux traits, ils correspondent à la notion de connecteur.

2.1.2 Les interactions entre composants

Si [Szyperski02] détaille le concept de composant, il n'aborde la question de leur interconnexion que sous l'angle technologique des solutions existantes, comme l'ORB de CORBA ou RMI pour Java. Les travaux de [Garlan97] et [Shaw95] abordent cette question sous un angle plus conceptuel, mettant en avant la notion de connecteur. Dans ACME, un langage destiné à l'échange d'information entre différents ADL, [Garlan97] définit, en plus de celui de composant, six concepts principaux : le connecteur, le système, le port, le rôle, la représentation et les correspondances[36]. Mise à part la notion de correspondance, assez spécifique à ACME et que je ne détaillerai pas, les autres concepts se retrouvent dans la plupart des approches, parfois sous d'autres appellations.

Selon [Garlan97], un connecteur représente une interaction entre plusieurs composants. D'un point de vue informatique, les connecteurs sous-tendent la communication et les activités de coordination entre les composants. À l'instar de [Shaw95], [Garlan97] illustre cette notion de connecteur avec les mécanismes de « pipes, d'appel de procédure, ou de diffusion d'événements ou même des mécanismes d'interaction plus complexes comme un protocole client-serveur ou un lien SQL entre une base de données et une application ».

[36] Component, connector, system, port, role, representation and rep-map

Chez [Garlan97], un connecteur relie un ensemble de composants par l'intermédiaire de leurs ports (voir Figure IV-3). Les ports constituent les différents points d'interaction d'un composant. Ils représentent une simple interface programmatique, un ensemble d'appels de procédure qui doivent être réalisés dans un certain ordre dûment spécifié, voire une interface de diffusion d'événement. Dans chaque port, se connecte une extrémité de connecteur suivant le rôle qu'elle incarne. Le rôle, qui dépend du type de connecteur, définit de quelle manière les composants participent à l'interaction. Par exemple, un connecteur binaire du type appel de procédure à distance définit les rôles appelant et appelé. De la même manière, un connecteur binaire de type bus à message définit les rôles émetteur et récepteur.

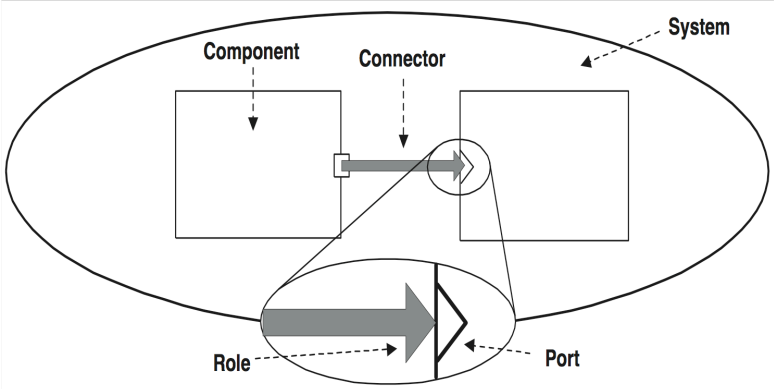


Figure IV-3 Les principaux concepts de l'approche à composants selon [Garlan97].

Dans UniCon[37] [Shaw95], un ADL centré sur la notion de connecteur, les concepts liés à l'interaction entre composants sont plus approfondis. Dans [Shaw95], le concept de port est appelé interface. Les interfaces d'UniCon se caractérisent par un ensemble d'acteurs[38]. L'acteur constitue « l'unité sémantique visible par laquelle un composant interagit avec l'extérieur »[39]. Chacun de ces acteurs est typé suivant la nature de l'interaction souhaitée. Par exemple, un acteur peut consister en un flux d'entrée ou de sortie, un ensemble d'appels de procédure ou de définitions de procédure ou des accès disques en lecture ou écriture. Du type des acteurs d'une interface, dépend le type de connecteurs qui y sont raccordables. En effet, une interface est raccordable à un connecteur seulement si celui-ci incarne les rôles

[37] « UniCon: Language for Universal Connector Support ».

[38] Players

[39] « The players, which form the bulk of the interface, are the visible semantic units through which the component can interact, request and provide services, or be influenced by external state or events ».

compatibles avec le type des acteurs qui constituent l'interface. Par exemple, un connecteur de type « pipe » incarne les rôles « source » et « puits ». Ces deux rôles ne sont compatibles qu'avec des acteurs de type « flux d'entrée ou de sortie ». En conséquence, seules les interfaces composées d'acteurs de type « flux » sont raccordables à un connecteur de type « pipe ».

2.1.3 D'outils pour le concepteur aux outils pour l'exécution

Dans la terminologie de [Garlan97], composants et connecteurs sont assemblés et forment un système (voir Figure IV-3). Il semble qu'il faille dissocier la notion de système de celle de composants composites présente dans UniCon [Shaw95], Wright [Allen97] ou Fractal [Brunneton02]. En effet, la notion de composant composite permet de décrire un composant comme étant lui-même, tel un système, constitué d'un assemblage de composants. En revanche, et à l'inverse d'un système, un composant composite a vocation à être partie d'un assemblage, au même titre qu'un composant standard.

Au départ, ces travaux autour du thème des architectures logicielles s'appliquent à fournir aux concepteurs de logiciels complexes des formalismes et des outils pour raisonner au niveau de l'organisation d'un système et de s'abstraire des détails d'implémentation. D'abord outils d'aide à la spécification, puis outils d'aide à la simulation, l'architecture logicielle investit les phases d'exécution en devenant dynamique, c'est-à-dire reconfigurable lors de l'exécution du système. Dans la littérature, cette question est traitée suivant différentes approches que je classe en deux catégories. Les approches de la première catégorie abordent cette question sous l'angle de l'architecture logicielle dans son ensemble, par exemple [Allen98], [Wile01], [Oquendo04], [Joolia05] et [Batista05]. La deuxième catégorie se focalise sur les aspects technologiques à mettre en œuvre au niveau de l'implémentation des composants pour permettre une telle dynamique, par exemple [Brunneton02] ou [Bures06].

2.1.4 Reconfiguration dynamique et architecture logicielle

Parmi les travaux qui traitent de la reconfiguration dynamique au niveau de l'architecture logicielle, [Allen98] est l'un des pionniers. Dans son approche, un système est modélisé sous la forme d'un ensemble de sous systèmes statiques, c'est-à-dire non reconfigurables, reliés par des transitions explicites. Ces transitions sont déclenchées par des événements de reconfiguration du système. Ainsi, la reconfiguration dynamique du système s'effectue par le passage d'un sous-système statique à un autre.

Pour parvenir à ce résultat, [Allen98] introduit la notion d'événements de contrôle dans les interfaces programmatiques des

composants. Par le biais de leurs interfaces, ils deviennent alors capables d'exprimer, pour chaque interaction dans lesquelles ils sont engagés, les conditions d'une reconfiguration. Ces évènements sont traités par une entité extérieure au système, le configurateur, qui embarque la description de l'ensemble des sous-systèmes statiques possibles. Par exemple, dans la Figure IV-4, la configuration en partie gauche constitue le fonctionnement normal. Lorsque le serveur primaire subit une défaillance, il produit un événement « panne » à destination du configurateur. Le configurateur est programmé pour répondre à cet événement en modifiant la configuration du système pour obtenir celle correspondant à la partie droite de la figure. Dans cette approche, l'objectif est de préserver une séparation nette entre la spécification des sous-systèmes statiques d'une part, et le comportement dynamique d'autre part.

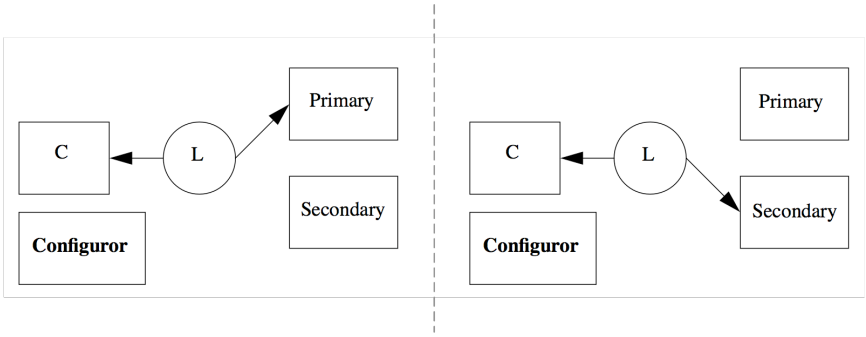


Figure IV-4 Architecture dynamique d'un système tolérant aux fautes dans [Allen98].

Dans ses travaux, [Wile01] s'appuie sur une approche différente pour traiter la question de la spécification des architectures dynamiques. Il se propose d'étendre ACME [Garlan97], jusque-là cantonné aux architectures statiques, pour lui permettre d'exprimer des architectures modulables lors de l'exécution d'un système. Dans cette approche, chaque constituant du système (composants, connecteurs, sous-systèmes) est décoré avec des propriétés complémentaires comme l'optionalité, la multiplicité, ou l'ouverture[40] (voir Figure IV-5). Ces propriétés introduisent dans l'architecture des degrés de liberté qu'il devient possible de faire varier à l'exécution.

[40] [Wile01] distingue les entités ouvertes et fermées. Une entité fermée est une entité dont la spécification est considérée complète. Une entité ouverte n'est spécifiée qu'à minima. À l'exécution, elle peut donc avoir des caractéristiques plus larges que celles spécifiées. Par exemple, un composant « ouvert » peut exposer un port qui n'aurait pas été décrit dans la spécification. En revanche, tout port spécifié d'un composant doit être exposé et respecter la spécification pour que ce composant soit conforme.

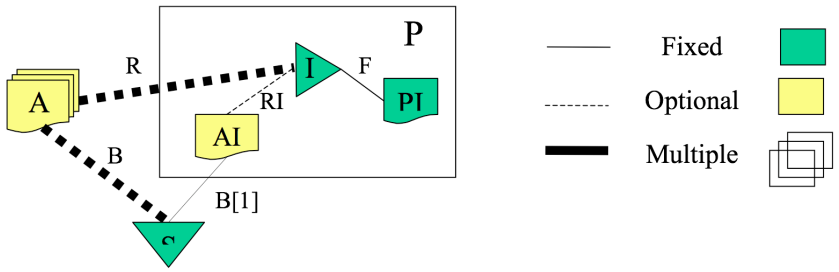


Figure IV-5 Architecture dynamique d'un système dans [Wile01]

Par exemple, l'architecture modélisée dans la Figure IV-5 précise que le composant A peut être instancié en plusieurs exemplaires ou pas du tout. En revanche, les composants S, I et PI doivent avoir une instance et une seule. Le connecteur F est requis entre I et PI, mais la connexion entre AI et I n'est pas obligatoire. Ainsi, ce type d'approche permet de capturer à un même niveau de spécification, l'architecture logicielle et l'ensemble de ses variations possibles à l'exécution. À l'instar des travaux précédents, l'ensemble des degrés de liberté de l'architecture est fixé à la conception. Cependant, pour permettre un maximum de flexibilité, [Wile01] prévoit qu'un système puisse être décrit de manière ouverte, c'est-à-dire spécifié à minima, laissant la possibilité d'instancier des composants et/ou d'établir des connexions supplémentaires à l'exécution.

Dans ses travaux, [Oquendo04] propose π-ADL, un ADL formel basé sur le π-calcul, qui partage avec ses prédécesseurs les concepts de composant, de port et de connecteur. Par rapport aux travaux précédents, [Oquendo04] se démarque en introduisant une nouvelle classe d'architecture logicielle reconfigurable : l'architecture mobile. Si les architectures dynamiques classiques se caractérisent par la possibilité, à l'exécution, de créer, détruire, reconfigurer ou déplacer des composants, les architectures mobiles se caractérisent par la possibilité donnée aux composants de se déplacer logiquement au sein de l'architecture. Ainsi, dans leur exemple (voir Figure IV-6), un sous-composant est capable de migrer d'un composant composite à un autre.

Pour être capable de modéliser de telles architectures, π-ADL fournit une notation qui combine une logique de prédicat et une logique temporelle pour en exprimer à la fois les spécifications structurelle et comportementale, de manière à faciliter la tâche de l'architecte. Cependant, comme dans toutes les approches précédentes, si l'architecture est dynamiquement reconfigurable, l'ensemble des reconfigurations possibles a été identifié lors de la conception d'un système. À l'exécution, il n'est pas possible de reconfigurer un système autrement que sous une forme prévue à la conception. L'approche décrite dans [Joolia05] et [Batista05] vise

à dépasser cette limitation. Dans cette approche, une reconfiguration peut être programmée, c'est-à-dire prévue lors de la conception ou ad hoc, c'est-à-dire décidée et calculée à l'exécution. Le résultat de l'étude de [Joolia05] et [Batista05] est une infrastructure nommée Plastik (voir Figure IV-7) dont l'objectif est de couvrir le cycle de vie d'un système depuis la phase de spécification jusqu'à son exécution.

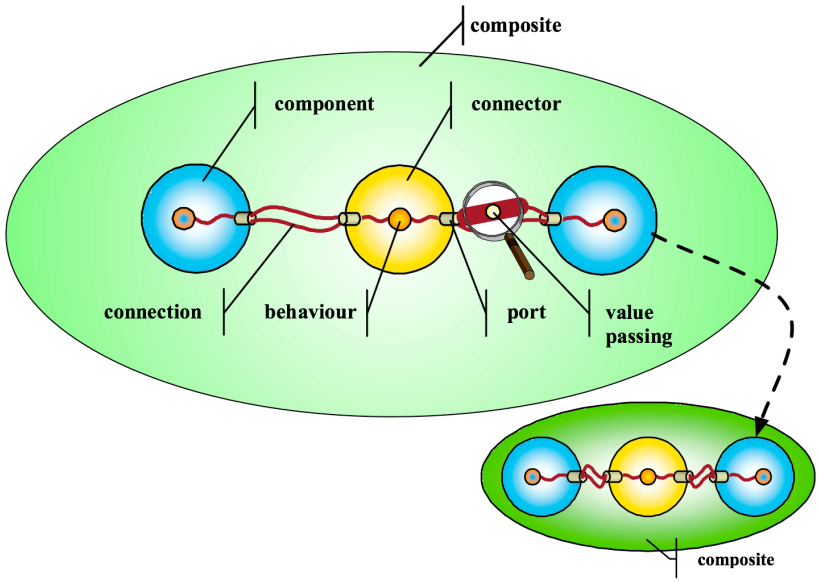


Figure IV-6 Les architectures mobiles [Oquendo04] se caractérisent par la faculté des composants composites à échanger des sous-composants.

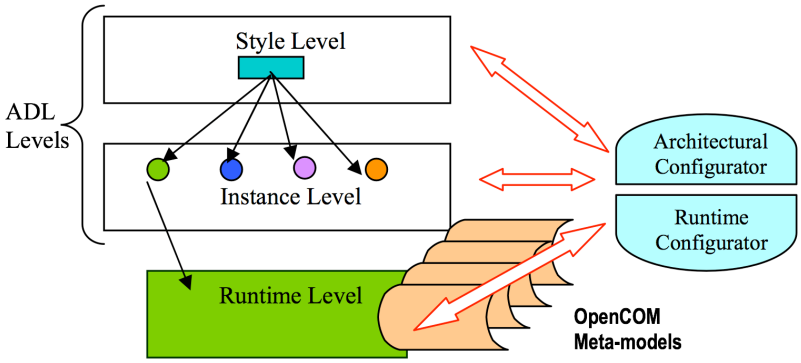


Figure IV-7 Vue générale de Plastik dans [Batista05].

L'infrastructure Plastik se décompose en trois niveaux : style, instance et exécution. Au plus haut niveau d'abstraction, le niveau style, l'architecture d'une classe de système est décrite sous la forme de patron générique. L'illustration donnée par [Batista05] est un patron générique pour des systèmes qui s'appuient sur le principe d'un empilement de protocoles. Le niveau d'abstraction intermédiaire, le niveau instance, spécialise un patron générique par l'ajout de nouvelles contraintes pour l'adapter à un contexte

spécifique. Dans l'exemple précédent, l'architecture générique pour des systèmes basés sur un empilement de protocole peut être spécialisé en architecture pour l'implémentation d'une pile TCP/IP.

À ces deux niveaux d'abstraction (style et instance), les architectures sont décrites en termes de composants, de connecteurs, et d'invariants au moyen d'une version enrichie de l'ADL ACME/Armani[41]. L'enrichissement de cet ADL par [Batista05] consiste en quatre constructions additionnelles destinées à exprimer les reconfigurations envisagées dès la conception. Notamment, ces nouveaux éléments du langage permettent d'exprimer l'instant où une reconfiguration donnée peut avoir lieu (`on (<condition>) do <actions>`), et d'indiquer de quelle manière doit être transformé l'assemblage de composants (`detach <role> from <port>`, `remove <element>`, et `dependencies <statements>`).

Enfin, le niveau exécution héberge des assemblages de composants OpenCOM[42] conformes aux architectures définies au niveau instance. Le passage de la spécification de l'architecture d'un système, de niveau instance, vers le système lui-même, de niveau exécution , est réalisé en deux étapes. La première consiste à obtenir un script Lua[43] chargé de l'instanciation des composants du système par compilation de la spécification ACME/Armani enrichie. Cette phase de compilation génère également des machines d'états finis qui implémentent les invariants et les reconfigurations programmées. La deuxième étape consiste à interpréter les scripts Lua chargés du chargement en mémoire des composants et de leur assemblage. Les machines d'états finis, quant à elles, sont exécutées par le « configurateur » de l'infrastructure Plastik dont le rôle est d'assurer la reconfiguration du système, qu'elle soit programmée lors de la conception ou décidée au cours de l'exécution.

Ce configurateur se décompose en deux parties. La première partie est consacrée à la reconfiguration au niveau de l'architecture. La deuxième, où sont hébergées les machines d'états finis, est en charge de la reconfiguration au niveau des composants OpenCOM. Ainsi, les reconfigurations ad hoc peuvent avoir lieu aussi bien au niveau instance par modification

[41] Armani est un sous langage d'extension pour ACME, basé sur la logique du premier ordre, destinée permettre l'expression de contraintes architecturales sur les assemblages de composants.

[42] OpenCOM est un modèle et une technologie à composants réflexifs maison.

[43] Lua est un langage interprété extensible.

de l'architecture, qu'au niveau exécution par manipulation directe des composants OpenCOM. Dans le premier cas, il s'agit de fournir au configurateur une spécification, écrite en ACME/Armani enrichi, des changements à apporter à l'architecture. Cette spécification sera compilée sous la forme d'un script Lua chargé d'appliquer la transformation au système en cours d'exécution. Dans le deuxième cas, il s'agit de fournir au configurateur des opérations directement applicables sur un assemblage OpenCOM. Qu'elles soient provoquées au niveau instance ou au niveau exécution, les reconfigurations ad hoc ne sont réalisables que si leur résultat est un assemblage de composants conforme aux invariants spécifiés initialement par le concepteur.

Comme l'illustre l'infrastructure proposée par [Joolia05] et [Batista05], l'application de reconfigurations à l'exécution décrite au niveau ADL repose sur la capacité de la technologie à composants sous-jacente à mettre en œuvre les adaptations envisagées. Les travaux précédents centrent leurs études au niveau ADL sans aborder, pour la plupart, la question de la technologie à composant. D'autres travaux, à l'inverse, délaissent le niveau ADL et traitent la question de la reconfiguration dynamique du point de vue de la technologie. Notamment, l'étude de [Bureš06] et [Hnětynka06], autour du modèle à composant SOFA 2.0 sont de ceux-ci.

2.2 Reconfiguration dynamique et technologies à composants

Le modèle à composant SOFA est un modèle académique à composants hiérarchiques. Ainsi, dans SOFA, les composants peuvent être composites, c'est-à-dire construits par assemblage de sous-composants. Ces sous-composants peuvent être eux-mêmes composites ou primaires, un composant primaire ne contenant aucun sous-composant. SOFA offre deux niveaux de description des composants : le cadre et l'architecture[44]. Le cadre est une vue « boîte noire » où le composant est seulement décrit par les interfaces programmatiques qu'il requiert et qu'il fournit. L'architecture est une vue « boîte grise » qui spécifie les sous-composants et leurs interconnexions. Indépendamment, un composant SOFA est organisé en deux parties : une partie contrôle, relative à la gestion du composant et une partie fonctionnelle, relative au métier du composant. Cependant, [Bureš06] et [Hnětynka06] identifient plusieurs limitations au modèle SOFA, en particulier concernant la reconfiguration dynamique, limitée à la mise à jour dynamique.

[44] Frame and architecture.

2.2.1 SOFA 2.0

La mise à jour dynamique consiste à remplacer, à l'exécution, un composant par un autre ayant des interfaces compatibles. Il s'agit d'une véritable reconfiguration dynamique car la structure du composant remplaçant peut être radicalement différente, en termes de sous-composants, de celle du composant remplacé. Cependant, une reconfiguration dynamique, dans le cas général, est une modification arbitraire de l'architecture d'une application [Bureš06]. Pour dépasser, entre autres, cette limitation, [Bureš06] et [Hnětynka06] proposent une évolution du modèle, baptisée SOFA 2.0. Cette évolution se caractérise principalement par trois points : l'intégration de trois patrons de reconfiguration, la structuration en microcomposants de la partie contrôle des composants, et l'introduction de connecteurs dans le modèle SOFA [Bureš06]. Le premier et le dernier de ces trois points sont les plus en rapport avec la problématique explorée par mes travaux.

La première avancée de SOFA 2.0 est donc l'intégration de trois patrons de reconfiguration dont seulement les deux principaux sont décrits dans [Hnětynka06] : Le patron « fabrique imbriquée » et le patron « interface utilitaire »[45]. Le patron « fabrique imbriquée » couvre les opérations d'ajout de nouveaux composants et de création de nouvelles connexions dans l'architecture. Dans le contexte d'une architecture plate[46], c'est-à-dire composée uniquement de composants primaires (voir Figure IV-8-a), l'ajout d'un nouveau composant ne soulève pas de problème particulier, tous les composants étant au même niveau hiérarchique. En revanche, dans les modèles à composants hiérarchiques comme SOFA 2.0, une question-clé concerne l'emplacement dans la hiérarchie où doit être ajouté le composant nouvellement instancié.

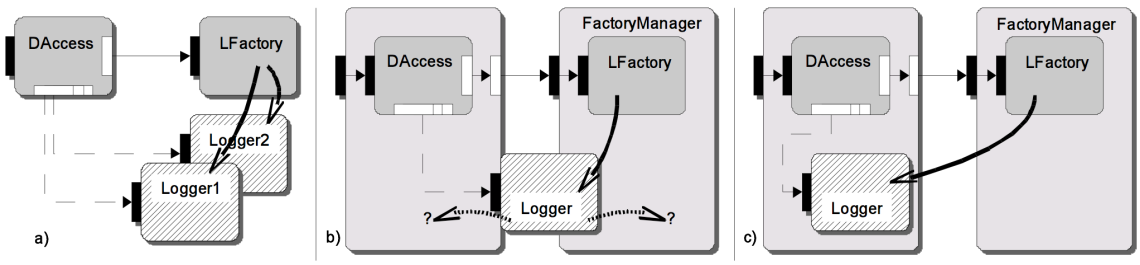


Figure IV-8 Choix du point d'insertion d'un nouveau composants dans une architecture SOFA 2.0 [Hnětynka06]

[45] Nested factory pattern and utility interface pattern.

[46] Architecture plate : tous les composants qui la composent sont primaires.

Comme le montre la Figure IV-8-b, la nouvelle instance de composant « Logger » peut être placée suivant trois possibilités : soit au plus haut niveau de la hiérarchie, soit au sein du composant contenant « DAccess », demandeur de l'instanciation de « Logger », soit au sein du composant « FactoryManager », contenant le composant fabrique « LFactory » créateur de l'instance de « Logger ». La règle choisie par le patron « fabrique hiérarchique » de SOFA 2.0, est de créer le nouveau composant au même niveau architectural que le composant ayant initié la création. Ainsi, comme l'illustre la Figure IV-8-c, le composant « Logger » est instancié au sein du composant conteneur de « DAccess ».

L'autre patron-clé de SOFA 2.0 est « interface utilitaire ». Il répond à la situation où une fonction fournie par un composant, par exemple, la fonction « impression », est nécessaire à plusieurs autres composants, situés à différents niveaux de la hiérarchie. En principe, un sous-composant ne peut pas accéder directement à un composant extérieur au composant composite auquel il appartient. Il a obligation de passer par les ports du composant composite. Le patron « interface utilitaire » apporte de la flexibilité à ce modèle, en introduisant un nouveau type d'interfaces, dites « utilitaires », dont la référence peut être passée librement à travers les composants. Les connexions à partir de cette référence peuvent être établies de manière indépendante, comme l'illustre la Figure IV-9 de la hiérarchie des composants du système.

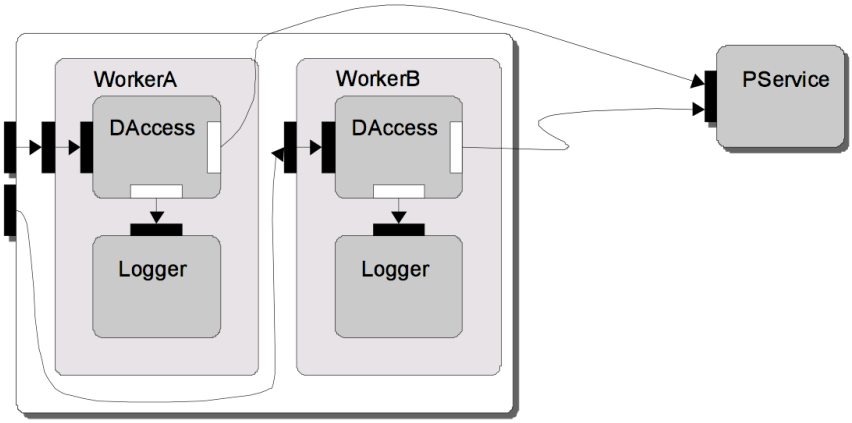


Figure IV-9 Illustration du patron « interface utilitaire ». Les composants « DAccess » peuvent accéder directement au composant externe « PService » [Hnětynka06].

Si les patrons de reconfiguration constituent une contribution originale de ces travaux, le traitement du concept de connecteur, dans SOFA 2.0, qui distingue une vue conception et une vue exécution, complète cette originalité. La vue conception spécifie les connecteurs en termes, d'une part, de style et de propriétés de communication et, d'autre part, des interfaces de composants

impliquées dans la communication. La vue exécution correspond au code d'implémentation des connecteurs obtenu par génération automatique à partir des spécifications de la vue de conception. Dans le cycle de vie d'une application, cette phase de génération des connecteurs intervient au moment du déploiement, c'est-à-dire juste avant le démarrage de l'application. L'avantage mis en avant par [Bureš06] d'une génération aussi tardive des connecteurs, concerne la possibilité d'optimisation maximale de leur implémentation, du fait de la connaissance complète, par le système, des caractéristiques de l'environnement d'exécution à cet instant du cycle de vie. Cette prise en compte dans le modèle SOFA 2.0 du concept de connecteur est une des caractéristiques qui le distinguent du modèle Fractal.

2.2.2 Fractal

Les travaux de [Bruneton02] s'intéressent à l'implémentation de systèmes distribués, hautement flexibles et dynamiquement configurables. Pour [Bruneton02] l'approche à composants semble être une bonne candidate pour traiter cette problématique, à la condition que la technologie choisie satisfasse sept requis-clé. Ces requis concernent les dimensions de composition et de dynamisme et s'appuient sur l'hypothèse que le concept de composant trouve une incarnation en tant que telle à l'exécution d'une part et qu'il constitue l'unité de construction d'un système d'autre part. Les requis de [Bruneton02] s'organisent autour des thèmes de l'encapsulation, de l'identité, de l'activité, de la composition, du partage, du cycle de vie, du contrôle et de la mobilité.

Les requis autour du thème de l'encapsulation se ramènent aux concepts classiques de l'approche à composants : les composants n'interagissent avec leur environnement qu'au moyen d'interfaces clairement définies et ne révèlent à l'environnement que le strict nécessaire de leur composition interne.

Le requis d'identité d'un composant, souvent implicite, et rendu explicite par [Bruneton02] : « un composant doit avoir une identité clairement définie, qui le distingue de manière non-ambiguë d'un autre composant, afin qu'il puisse être spécifiquement accédé et manipulé »[47].

Dans le même esprit, sur le thème de l'activité, [Bruneton02] recommande qu'un composant puisse expliciter les activités auxquelles il participe au sein de l'architecture, afin de rendre

[47] Identity means that components should have well-defined identity that unambiguously distinguishes one component from another so as to allow a component to be accessed and manipulated specifically.

possible la manipulation directe de ces activités par un système, même si chacune implique plusieurs composants. Dans le cadre d'un composant pour l'IHM, le concept d'activité décrit par [Bruneton02] semble correspondre à celui de tâche utilisateur.

Sur le thème de la composition, [Bruneton02] précise qu'un modèle général doit permettre l'assemblage dynamique de composants afin de former des composants de plus haut niveau, sans privilégier une manière particulière de composer des composants. Au contraire, pour maximiser la réutilisation et pour répondre aux besoins de configuration dynamique, plusieurs sémantiques de composition doivent être disponibles.

Connexe au thème de la composition, celui du partage s'intéresse au partage des ressources système (par exemple, une unité de calcul, des dispositifs d'entrée/sortie, ou même des ressources de plus haut niveau comme un serveur ou une connexion). Ici, cette question est traitée par la possibilité donnée à plusieurs configurations (c'est-à-dire d'assemblage de composants) de partager des composants entre elles, et de permettre plusieurs formes de partage possible.

Sur le thème du cycle de vie des composants, afin d'être applicable à la problématique des gros systèmes distribués, [Bruneton02] requiert qu'un modèle général doit permettre la gestion de différents modèles de cycle de vie. Ce thème se rapproche de celui du contrôle. Pour [Bruneton02] il doit être possible de créer plusieurs types de contrôleurs de composants. Le contrôleur de composants est une fonction qui donne des moyens d'inspection et de contrôle de l'exécution d'une collection de composants.

Enfin, le thème de la mobilité aborde la modification arbitraire d'un assemblage de composants. [Bruneton02] insiste sur la possibilité d'instanciation dynamique de nouveaux composants ou de nouveaux assemblages de composants et même de migration de composants d'un assemblage vers un autre. Cette possibilité apparaît comme étant une condition nécessaire pour permettre de modéliser des processus de reconfiguration complexe.

Comme, les technologies à composants existantes ne satisfont pas à la fois l'ensemble des sept requis posés par [Bruneton02], celui-ci propose le modèle à composants Fractal. Ce modèle s'articule autour de trois concepts-clé : les composants, les interfaces et les noms. Un composant Fractal est constitué d'une membrane et d'un contenu. Le contenu se compose d'un ensemble fini de composants Fractal, encapsulé par une membrane qui regroupe les interfaces fonctionnelles, d'introspection et de configuration du composant, ainsi que différents éventuels contrôleurs. Une originalité notable du modèle Fractal est de permettre à plusieurs composants de partager des sous-composants. Un sous-composant

partagé obéit alors aux contrôleurs des différents composants qui l'englobent. Pour éviter les conflits, l'ensemble pertinent des composants qui partagent des sous-composants doivent être réunis au sein d'un même composant Fractal dont les contrôleurs décideront de la politique de contrôle.

Conformément aux requis, un composant Fractal interagit avec son environnement à travers des points d'interaction appelés interfaces. Si un composant expose, par définition, ses interfaces au monde extérieur, il est du ressort du contrôleur du composant de décider de l'exposition au monde extérieur de chaque interface des sous-composants. Ainsi, un composant Fractal peut exposer un nombre variable d'interfaces au cours de son exécution. Les interfaces se composent d'opérations qui, dans le modèle Fractal, constituent l'unité d'interaction. Le modèle définit deux types d'opérations : les opérations unidirectionnelles et les opérations bidirectionnelles. Une opération unidirectionnelle se ramène à une invocation de méthode qui ne retourne aucun résultat, au contraire de l'opération bidirectionnelle qui retourne un résultat. Ces deux types d'opération peuvent véhiculer des arguments. Les arguments d'opération peuvent être des références (d'interface, par exemple), de simples valeurs ou même des composants, sous une forme passive. La manière dont doit être utilisées les opérations caractérise le type d'une interface. Une interface est dite serveur si elle est destinée à recevoir les appels d'opération, et, suivant le type de l'opération, à retourner un résultat. Une opération est dite cliente si elle est destinée à émettre des appels d'opération, et, suivant le type de l'opération, à recevoir un résultat.

Pour interagir les unes avec les autres, les interfaces doivent faire partie d'une liaison. Le modèle Fractal distingue les liaisons primitives des liaisons composites. Une liaison primitive est une connexion directe entre une interface cliente et une interface serveur compatibles. Une interface serveur est compatible avec une interface cliente si, au minimum, elle est capable de répondre à tous les appels d'opération que l'interface cliente peut émettre. De manière symétrique, une interface cliente est compatible avec une interface serveur si, au minimum, elle est capable d'accepter tous les résultats retournés par les opérations appelées de l'interface serveur. Une liaison composite met en œuvre des interactions entre composants plus complexes, à partir d'une combinaison de liaisons primitives et de composants. Une liaison composite est donc un composant Fractal à part entière.

Selon [Bruneton02], l'objectif de la notion de liaison est de généraliser le concept de connecteur, qui existe dans l'approche à composants, afin qu'il corresponde à la notion de liaison, qui existe dans les intergiciels pour systèmes distribués flexibles, par exemple l'ORB Jonathan [Dumant99]. Enfin, dernier concept

autour duquel s'articule Fractal, les noms sont des symboles destinés à servir de référence. Si le modèle Fractal n'impose aucune contrainte quant à la structure de ces références, il impose qu'elles soient organisées au sein d'espaces de noms afin d'assurer leur unicité.

Ainsi, Fractal est un modèle à composant conçu pour le développement de systèmes distribués dynamiquement configurables. S'appuyant sur ces travaux, [David06] propose SAFRAN[48], une extension du modèle Fractal pour traiter la question des systèmes adaptatifs, c'est-à-dire des systèmes capables de s'adapter de manière autonome à la variation de leur contexte d'exécution.

2.2.3 SAFRAN

[David06] montre que l'adaptation d'un système est « une préoccupation transverse au logiciel métier ». Il prône la modularisation de cette préoccupation « afin de garantir une meilleure réutilisation et maintenabilité du logiciel métier ». Pour atteindre cet objectif, la démarche de [David06] est d'appliquer à l'approche à composants le principe de la programmation par aspect. La programmation par aspect [Kiczales97] est une technique de programmation qui simplifie la mise en œuvre du principe de séparation des préoccupations. Ainsi, chaque préoccupation donne lieu à un aspect. L'aspect se décompose en un couple (coupe, action) chargé de modifier la sémantique du code fonctionnel. La coupe est un ensemble de points de jonction situés dans le code fonctionnel. L'action constitue le code à exécuter à l'endroit de la coupe. Une fois les aspects définis, ils sont tissés avec le code fonctionnel, soit de manière statique, soit de manière dynamique, suivant la technologie employée. Dans les approches par aspects classiques, les points de jonction qui composent les coupes sont des points d'entrée de méthode ou des événements relatifs à l'exécution du programme.

Comme le processus d'adaptation est dynamique et réactif[49], [David06] privilégie l'approche événementielle des points de jonction. Il identifie alors l'ensemble des événements pertinents pour constituer des coupes destinées au tissage des aspects d'adaptation. Son étude montre que si les évènements relatifs à l'exécution d'un système, dits événements endogènes, sont nécessaires, ils ne suffisent pas pour traiter la question de l'adaptation dans le cadre des systèmes sensibles au contexte. Pour cette raison, [David06] propose l'extension des points de

[48] SAFRAN : Self-Adaptive FRActal compoNents

[49] Ce processus est dit réactif car c'est un changement significatif dans l'environnement qui déclenche une décision d'adaptation du système.

jonction aux événements exogènes, c'est-à-dire relatifs au contexte de l'exécution. Ainsi, le processus d'adaptation peut être déclenché au moyen d'une dizaine d'événements endogènes, relatifs à l'échange de messages entre composants[50] et à la modification dans l'architecture du système[51], et de quatre types d'événements exogènes, traduisant les évolutions du contexte de l'exécution[52]. Dans SAFRAN, les actions de reconfiguration sont définies au moyen du langage dédié FScript, développé dans le cadre de ce projet. C'est un langage procédural qui donne accès aux opérations sous-tendues par les composants Fractal. FScript se caractérise par deux points : d'une part, il dispose d'une notation spécifique, FPath, fortement inspirée de XPath, pour naviguer dans les architectures Fratcal, et d'autre part, il garantit la consistance d'une application après reconfiguration, notamment en termes d'intégrité transactionnelle (atomicité, consistance de l'état final, isolation) et de terminaison en temps borné des reconfigurations.

Pour traiter de la question de l'adaptation dynamique au contexte, SAFRAN partage avec WCOMP [Cheung-Foo-Wo06] l'idée d'associer l'approche à composant et la technique de programmation par aspects. Cependant, cette association est réalisée de manière différente. Dans SAFRAN, les aspects gèrent directement l'adaptation, de son déclenchement, la coupe modélisant une condition d'adaptation, à sa réalisation, l'action consistant à réorganiser l'assemblage des composants au moyen d'un programme FScript. En revanche, dans WCOMP, les aspects n'entretiennent qu'un lien indirect avec le processus d'adaptation. En effet, ceux-ci modélisent des interconnexions possibles entre des composants. Suivant l'évolution de conditions contextuelles, certains de ces aspects seront appliqués, d'autres verront leur application suspendue et les derniers seront laissés de côté, induisant un réassemblage dynamique des composants.

2.2.4 WCOMP

WCOMP se présente à la fois comme une plate-forme de développement rapide pour la construction d'applications par assemblage de composants logiciels et comme un environnement d'exécution pour ces applications. Un composant WCOMP est une instance de classe qui interagit avec ses pairs par un ensemble

[50] Message-received, message-returned, message-failed

[51] component-created, component-start, component-stop, parameter-changed, subcomponent-add, subcomponent-remove, binding-created, binding-destroyed

[52] changed(expression), realized(condition), appears(chemin), desappears(chemin)

de ports d'entrée et de sorties. Un port de sortie est une source d'événements. Un port d'entrée est une méthode. La liaison entre un port de sortie et un ou plusieurs ports d'entrée s'effectue par un mécanisme d'abonnement : une méthode est appelée suite à l'émission d'un événement auquel elle est abonnée. Ainsi, contrairement à Fractal, où l'interaction logicielle passe par des interfaces connectées les unes aux autres, dans WCOMP les composants interagissent par diffusion de messages. Les règles de diffusion des messages, c'est-à-dire l'ensemble des abonnements entre méthodes et événements, sont fixées par des *aspects d'assemblage*.

Comme dans toutes les approches par aspects, un *aspect d'assemblage* est défini par un couple (coupe, action). Ici, la coupe peut être soit l'émission d'un événement, soit l'appel d'une méthode. L'action est un *schéma d'interaction*, c'est-à-dire une description des méthodes devant être appelées et/ou des messages devant être émis et la façon de les émettre. Les aspects d'assemblage sont spécifiés à l'aide du langage ISL4WCOMP. Ce langage étend l'Interaction Specification Language (ISL) de [Berger01] pour WCOMP. Notamment, il reprend d'ISL la propriété d'être *composable*, c'est-à-dire la faculté de pouvoir composer plusieurs schémas d'interaction en un seul, en combinant les comportements que chacun décrit. L'interprétation d'un schéma d'interaction ISL4WCOMP est réalisée par une entité logicielle appelée *designer* et produit en sortie une combinaison d'opérations élémentaires (ajouter un composant, retirer un composant, connecter, déconnecter) qui visent à transformer l'assemblage de composants cible. Cette combinaison d'opérations élémentaires est appliquée sur l'assemblage de composants par l'intermédiaire du *container,* l'entité logicielle en charge de la gestion de l'assemblage de composants. Ainsi, dans WCOMP, les mécanismes d'adaptation reposent, d'une part, sur le langage ISL4WCOMP, et d'autre part, sur l'architecture *designer-container-assemblage de composant*.

Les mécanismes d'adaptation de WCOMP permettent la modification dynamique d'un assemblage de composants en fonction de données contextuelles. Ces mécanismes disposent en entrée d'un ensemble de *schémas contextuels* définis par le concepteur. Un schéma contextuel résulte de l'association d'un aspect d'assemblage avec des conditions contextuelles. Si les conditions contextuelles sont remplies, l'aspect d'assemblage doit être appliqué. Dans le cas contraire, deux cas se présentent : si l'aspect d'assemblage était appliqué, il doit être défait, sinon l'aspect d'assemblage est laissé de côté. L'ensemble des aspects d'assemblage à appliquer et à défaire est transmis au *designer*. Grâce aux possibilités offertes par le langage ISL4WCOMP, le *designer* les fusionne et interprète le schéma d'interaction qui en

résulte. Par l'intermédiaire du *container*, le *designer* est alors en mesure de modifier l'assemblage de composants.

2.2.5 Approches à composants en synthèse

En résumé, si les approches à composants s'appuient toutes sur la notion de composant logiciel, aucune ne partage tout à fait la même définition de cette notion. Ainsi, un composant issu d'un modèle particulier est difficilement comparable à un autre composant issu d'un autre modèle. La notion de composant permet la construction d'application par assemblage de composants, mais ceux-ci doivent alors tous être issus du même modèle à composant. Cependant, cette construction par assemblage permet une forme de reconfiguration dynamique indispensable aux systèmes interactifs plastiques. Si certaines approches nécessitent de prévoir tous les cas possibles de reconfiguration lors de la conception, d'autres autorisent des marges de manœuvres plus importantes.

Bien que dynamique et possiblement envisagée que partiellement lors des phases de conception, la reconfiguration d'une application reste limitée parce que la notion de disponibilité dynamique des composants n'est pas une hypothèse des approches à composants [Cervantes04]. En effet, l'ensemble des composants qui peuvent entrer en jeu lors de reconfigurations doit être connu au moment dès la conception. Au contraire, la notion de disponibilité dynamique des composants précise qu'au cours de l'exécution, de nouveaux composants sont susceptibles de devenir disponibles tandis que d'autres sont susceptibles de devenir indisponibles. Cette hypothèse implique qu'un système doit être capable de découvrir dynamiquement de nouveaux composants et de prendre en compte la disparition imprévisible de composants en cours d'utilisation. L'idée de disponibilité dynamique est au cœur des approches à services.

3. Approche à services et disponibilité dynamique

La définition la plus répandue d'un service est donnée par [Bieber01] : « Un service est un comportement défini contractuellement qui peut être implémenté et fourni par un composant quelconque pour être utilisé par un autre composant ; l'interaction entre les deux composants s'appuie uniquement sur le

contrat »[53]. Cette approche, qui s'inscrit explicitement dans le cadre des systèmes distribués, est caractérisée par la découverte des fournisseurs de services par leurs clients au moment de l'exécution. Ainsi, les liaisons entre clients et fournisseurs de services sont établies tardivement, c'est-à-dire au moment où cette liaison devient nécessaire. Le client ne connaît ni l'adresse ni le nom d'un fournisseur de service, mais uniquement la fonctionnalité que doit offrir le service qu'il recherche. Les fonctionnalités offertes par les fournisseurs de services disponibles sont déclarées par ces fournisseurs de services dans un annuaire au moyen de descripteurs de service (voir Figure IV-10).

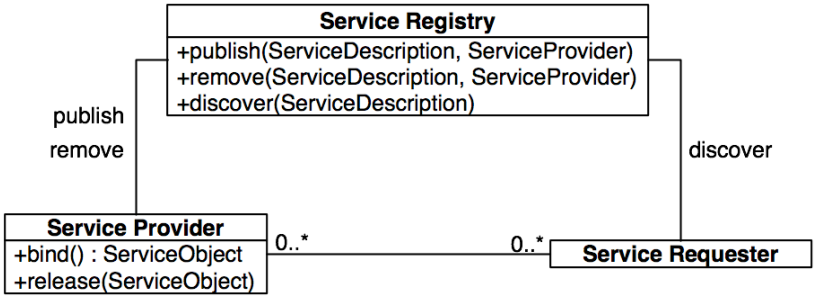


Figure IV-10 Approche à services : les acteurs en présence. Tiré de [Cervantes04], p.65.

Pour découvrir un fournisseur de service, un client adresse une requête à l'annuaire qui décrit la fonctionnalité attendue du fournisseur de service sous la forme, également, d'un descripteur de service. En retour, l'annuaire indique au client une référence vers le fournisseur de service capable de le satisfaire. Le client établit alors la connexion avec le fournisseur de services. Pour qu'un client puisse utiliser les fonctionnalités offertes par un fournisseur de service, peu importent les technologies utilisées pour implémenter l'un et l'autre. Seul compte le « contrat » défini dans le descripteur de service et le protocole de communication. Ainsi, chaque instanciation de l'approche à services implémente au minimum un annuaire et définit un protocole de communication permettant aux clients d'interagir avec l'annuaire, d'interpréter la référence vers un fournisseur de service retourné par l'annuaire et d'établir la liaison avec le fournisseur de service découvert.

Un exemple d'application caractéristique de ce principe de l'approche à services est donné par les *Services Web*. Un *service Web* « repose sur un système logiciel conçut pour sous-tendre une

[53] « A service is a contractually defined behavior that can be implemented and provided by any component for use by any component, based solely on the contract. »

interaction entre deux machines à travers un réseau » [W3C04]. Dans cet espace technologique, les services sont décrits à l'aide du langage WSDL[54], qui, comme tous langages XML, est interprétable à la fois par l'homme et par la machine. Un descripteur de service WSDL est centré sur l'interaction logicielle et la caractérise à différents niveaux de détails. Ainsi, une description WSDL embarque la définition des messages échangés et des structures de données que ceux-ci véhiculent, indique la localisation du service sur le réseau sous la forme d'une URI, donne des indications sur les patrons d'interaction logicielle en vigueur, par exemple en qualifiant les échanges de message de synchrones ou d'asynchrones, et enfin, précise les protocoles de transport et de sérialisation des données que le service utilise pour interagir avec ses clients. Le plus souvent HTTP joue le rôle du protocole de transport. Il est généralement associé au protocole SOAP, qui fixe le format des messages échangés entre les parties. Enfin, dans l'espace technologique des services Web, les annuaires de service s'appuient sur le protocole standard UDDI[55], une initiative industrielle portée par le consortium OASIS. Par ailleurs, dans [W3C04], une différence est faite entre le *service*, notion abstraite caractérisant une fonction, de *l'agent*, entité logicielle ou matérielle concrète, qui l'implémente. Ainsi, dans le cadre des services Web, les protocoles et formats WSDL, SOAP, UDDI et HTTP, indépendants des plates-formes, permettent la spécification des *services*, laissant au concepteur le libre choix des technologies pour implémenter des *agents* chargés de les rendre.

Les *services Web* proposent donc un moyen de spécifier un service indépendamment du ou des agents logiciels chargés de le rendre. À l'inverse, OSGi est un cadre technologique fondé sur Java, proposé par l'Alliance OSGi[56], qui offre un modèle d'implémentation ainsi qu'un environnement de déploiement et d'exécution de services. Dans la terminologie OSGi, le terme *service* correspond au terme *agent* de la terminologie des services Web. Ainsi, OSGi est une technologie avec laquelle il serait envisageable d'implémenter des agents de services Web. Ces agents de service sont conditionnés au sein de *bundles*. Un bundle se présente sous la forme d'une archive Java (JAR) qui contient des métadonnées, un activateur et l'implémentation de la logique métier. Les métadonnées associées au bundle décrivent les

[54] WSDL : Web Service Description Language. Ce langage est un standard du W3C.

[55] UDDI : Universal Description, Discovery, and Integration protocol. (Voir http://uddi.xml.org).

[56] Voir http://www.osgi.org

dépendances de code et les fonctions offertes par le ou par les agents de service embarqués. Quant à l'activateur, c'est un programme exécutable en charge de la gestion du cycle de vie de ces agents.

Le bundle est donc un vecteur destiné au déploiement d'un ou de plusieurs agents de service au sein d'une plate-forme d'exécution appelée *passerelle* dans la terminologie OSGi. Si un bundle peut embarquer un agent correspondant à un service Web, il est avant tout conçu pour le déploiement de « services OSGi ». Alors qu'un service Web est accessible à travers le réseau, un service OSGi standard n'est accessible que pour des clients situés sur la même passerelle que lui. Ainsi, chaque passerelle OSGi dispose de son propre annuaire de service. Les annuaires de service OSGi mémorisent les services disponibles sous la forme d'un nom d'interface et d'un ensemble de propriétés de type (clé, valeur). En outre, OSGi définit une API à destination l'activateur de bundle pour ajouter ou retirer des services de l'annuaire, ainsi qu'une API à destination des clients pour que ceux-ci puissent découvrir et établir des liaisons avec des services disponibles sur la passerelle.

Ainsi, OSGi, qui s'inscrit dans l'approche à services, offre les mécanismes nécessaires à la disponibilité dynamique des agents de service. Cependant, ces mécanismes seuls n'étaient pas suffisants : il était difficile pour le concepteur d'un client de pouvoir tirer parti de l'arrivée dynamique d'un nouveau service ou de s'adapter à la disparition soudaine d'un service utilisé. En effet, il incombait au concepteur du client de prendre en compte ce type d'événement extérieur et d'implémenter, à chaque fois, les mécanismes propres à l'adaptation dynamique. Ainsi, pour pallier cette faiblesse, la dernière version en date de la spécification OSGi (OSGi R4) intègre les avancées issues des travaux de [Cervantes04] qui proposent Service Binder, un « modèle à composants orienté services ».

Dans le modèle de [Cervantes04], un composant est une entité logicielle disponible dynamiquement qui offre et requiert des services. L'instance d'un composant est encadrée par un « gestionnaire d'instance ». Ce gestionnaire d'instance est en charge d'enregistrer les services fournis par le composant qu'il encapsule auprès de l'annuaire. En s'appuyant sur des descriptions de services enrichies, il est également en charge de sélectionner pour le compte du composant encadré les composants nécessaires pour satisfaire les services requis, et enfin, il est en charge d'établir les connexions avec les composants sélectionnés. En cas d'apparition ou de disparition de service, c'est toujours à ce gestionnaire d'instance qu'il incombe de gérer la découverte de composants de remplacement et l'établissement de nouvelles connexions pour que le composant encapsulé puisse continuer à fonctionner. Ainsi, la prise en charge des mécanismes

d'adaptation dynamique par le gestionnaire d'instance, simplifie le travail du concepteur de composant qui peut centrer son effort sur la logique métier de celui-ci. Cependant, d'une part, il n'est pas possible d'étendre la couverture fonctionnelle du gestionnaire d'instance, et d'autre part, cette approche impose au programmeur l'utilisation d'un style de programmation et d'une API particulière [Escoffier07]. Avec iPOJO, [Escoffier07] se propose d'apporter une solution à ces deux limitations.

À l'instar de Service Binder, pour lequel il constitue l'étape suivante, iPOJO (injected Plain Old Java Object) est un modèle à composant à service développé au-dessus d'OSGi. Son objectif est de simplifier au maximum la tâche du développeur de composants en lui permettant de se focaliser sur la conception de la logique métier et non sur les aspects propres à l'approche à service et aux propriétés non-fonctionnelles. Ainsi, le développeur implémente la logique métier sous une forme très proche d'un POJO (Plain Old Java Object), et se contente de configurer le conteneur de composant iPOJO destiné à l'encapsuler. Comme dans l'approche de [Cervantes04], un composant iPOJO expose les fonctions qu'il offre sous la forme de services fournis et exprime ses dépendances sous la forme de services requis. Les liaisons entre les composants sont établies à l'exécution et sont réalisées par le conteneur du composant iPOJO. En s'appuyant sur une description détaillée des composants, le conteneur est capable de prendre en charge les aspects non-fonctionnels de l'exécution du composant, notamment les opérations de publication des services auprès de l'annuaire, de découverte et de sélection des services.

Si la fonction du conteneur de composant d'iPOJO correspond à celle du gestionnaire de services dans Service Binder, sa structure logicielle est différente. Ainsi, un conteneur de composant iPOJO rassemble un certain nombre de *handlers*, chacun prenant en charge la gestion d'un aspect non-fonctionnel de l'exécution d'un composant. Six handlers sont fournis par défaut. Le handler *Dependency* résout les dépendances de service, *Provided Service* gère l'offre de service, *Lifecycle Callback* permet d'appeler des méthodes de l'instance de composant lorsque son état d'exécution change, *Configuration* permet la configuration dynamique, *Architecture* permet d'exposer l'architecture interne du composant, et enfin, *Controller Lifecycle* permet au code interne d'influer sur le cycle de vie de l'instance de composant. Contrairement aux gestionnaires de services de [Cervantes04], les conteneurs de composants d'iPOJO sont extensibles : il est possible de leur ajouter de nouveaux handlers afin de prendre en compte de nouvelles préoccupations non-fonctionnelles.

Contrairement à l'approche à composants, les concepts de l'approche à services semblent bien mieux définis et mieux

partagés par les différentes technologies à services. Dans leur fondement, ces technologies n'envisagent pas leur hétérogénéité : c'est-à-dire qu'un client implémenté dans une technologie à services particulière ne peut pas, tel quel, interagir avec un service implémenté dans une autre technologie. Par exemple, un service Web ne peut pas interagir directement avec un composant iPOJO. Cependant la proximité des concepts mis en œuvre dans chaque technologie permet d'envisager ce type d'interaction. C'est, par exemple, le cœur des travaux de recherche du projet européen Amigo [Amigo-D2.1].

4. Apports et limites du GL en synthèse

Pris séparément, les requis techniques pour l'implémentation et pour l'exécution de systèmes interactifs plastiques trouvent quelques solutions. Les intergiciels répondent aux problèmes de la distribution et de certains niveaux d'hétérogénéité, les approches à composants proposent des pistes quant à la reconfiguration dynamique et les approches à services semblent adaptées pour traiter la question de la disponibilité dynamique des différents éléments, matériels ou logiciels, d'un système interactif plastique. Malheureusement, aucune approche ne couvre à la fois tous ces requis. Si certaines combinaisons sont possibles, notamment aux niveaux des intergiciels [Grace03] et des services [Amigo-D2.1][Cervantes04][Escoffier07], ces tentatives ne répondent pas totalement à la problématique posée par les systèmes interactifs plastiques.

Si la sélection d'un élément d'interface utilisateur à la manière de l'approche à services est envisageable, elle ne peut pas se faire sur la base des descripteurs de service en vigueur dans les technologies actuelles. En effet, ces descripteurs ne permettent la sélection que sur la fonctionnalité offerte par le service à son client, voire sur la qualité de service attendue, en termes, par exemple, de latence, de précision, de fiabilité ou de sécurité. Un élément d'interface utilisateur, lui, sert deux types de clients à la fois : un client logiciel : le noyau fonctionnel et un client humain : l'utilisateur. Sa sélection doit être faite sur la fonctionnalité qu'il offre au noyau fonctionnel d'une part, et à l'utilisateur d'autre part. En plus de la prise en compte des critères traditionnels de qualité de service, il est impératif d'ajouter celui de l'utilisabilité, propre au domaine l'interaction homme–machine, et au cœur de la question de la plasticité des systèmes interactifs. En outre, si la localisation d'un service sur le réseau importe peu dans l'approche traditionnelle, ce n'est pas le cas pour une interface utilisateur : dans l'exemple d'une interface graphique il est primordial de pouvoir décider sur quel écran elle sera affichée. Par conséquent,

il doit être possible de décider de la localisation d'un service d'interaction homme–machine.

Au-delà de l'insuffisance descriptive des descripteurs de services traditionnels, la question de l'hétérogénéité des technologies reste à traiter. L'intergiciel ReMMoC [Grace03], permet d'implémenter des clients capables d'interagir avec un ensemble de services implémentés au-dessus d'un ensemble de technologies hétérogènes. Ce niveau d'hétérogénéité est géré de manière transparente par l'intergiciel. Cependant, la réciproque n'est pas possible : un service ne peut pas être implémenté au-dessus de ReMMoC de manière à être accessible par n'importe quel client. Or, la fonction d'une interface utilisateur est de permettre d'une part de rendre observable à l'utilisateur des concepts du noyau fonctionnel et d'autre part de permettre à l'utilisateur d'agir sur le système, c'est-à-dire de manipuler le noyau fonctionnel. Dans cette mesure, une interface homme–machine est à la fois service et client.

L'approche à services dans [Amigo-D2.1] autorise l'existence d'entités à la fois cliente et service. Mieux, leur solution permet de gérer de manière transparente l'hétérogénéité des technologies à services sous-jacentes aussi bien pour les services et pour les clients, mais également pour leurs développeurs. Leur solution s'inscrit également dans le cadre de plates-formes constituées de manière dynamique et opportuniste avec des plates-formes élémentaires des plus puissantes au plus légères. Pourtant, leur solution par traducteur de protocole s'avère gourmande en puissance de calculs, notamment au moment de la génération des traducteurs. Dans leur contexte de réseaux domestiques ouverts [Amigo04] ceci ne pose pas de problème car leur solution n'a pas besoin d'être déployée sur l'ensemble de la plate-forme : il suffit au minimum qu'une seule des plates-formes élémentaires héberge le générateur de traducteur. Or, la plate-forme cible dans [Amigo04] comporte toujours un dispositif assez puissant pour cette tâche.

Dans notre problématique, deux plates-formes élémentaires aux ressources de calculs limités peuvent former ensemble une plate-forme. Dans ce cas, aucune des deux ne pourrait prendre en charge la génération des traducteurs nécessaires. Notre approche doit être capable de résoudre la question de l'hétérogénéité des technologies également dans ce type de configuration. Enfin, l'utilisateur doit garder le contrôle sur le système. À l'instar de la manière avec laquelle il compose sa plate-forme, l'utilisateur doit pouvoir décider du remodelage et de la distribution de l'interface homme–machine du système interactif qu'il utilise [Coutaz06]. Or, dans l'approche à services, l'initiative de la sélection d'un service et la liaison au service est de la responsabilité du client. À

l'inverse, l'initiative de la sélection des composants et de la liaison entre composants relève de la responsabilité d'un tiers. Ainsi, il apparaît que des concepts-clés des approches à composants et à services doivent être inévitablement associées au sein d'une même approche pour pouvoir construire des systèmes interactifs plastiques. L'approche de [Cervantes04] a ouvert la voie en proposant le premier « modèle à composants orienté services ».

Cependant, dans l'approche de [Cervantes04], une reconfiguration ne peut avoir lieu que sur la disparition ou l'apparition d'un composant dans le référentiel et non sur un événement contextuel. La reconfiguration est alors gérée par auto–adaptation grâce aux facultés des gestionnaires d'instance. Cette approche rend donc difficile la possibilité de donner à l'utilisateur l'initiative et le contrôle sur la reconfiguration du système. Comme nous le montrons dans [Balme04], un système interactif plastique doit être capable à la fois, selon les termes définis par [Oreizy99] de reconfiguration interne et externe, c'est-à-dire respectivement par ses propres moyens ou grâce au concours d'un tiers. Si un rapprochement entre les paradigmes à composants et à services est nécessaire, l'approche de [Cervantes04] ne peut pas convenir, en l'état, à notre problématique. La notion de disponibilité dynamique des composants est indispensable. Cependant, le principe de l'approche à services, qui consiste à laisser seul au client l'initiative de la connexion à un service, doit être complétée par la possibilité laissée à un tiers d'accomplir cette fonction. En revanche, les lacunes de l'approche de [Cervantes04] semblent être, au regard des requis de la plasticité, comblées par le caractère extensible des conteneurs de composant iPOJO [Escoffier07]. Il conviendrait alors de développer des handlers spécialisés pour la plasticité qui permettraient, d'une part, la délégation à un tiers des prises de décision quant à la sélection et à l'établissement des liaisons, et d'autre part, d'élargir à l'ensemble des évènements relatifs au contexte de l'interaction la possibilité de déclancher une reconfiguration.

Chapitre V

Contribution aux outils conceptuels pour la réalisation et l'exécution d'IHM plastiques

Avant-propos

L'espace problème de la plasticité des systèmes interactifs établi au chapitre II démontre la complexité du sujet. Il inclut l'opération de *remodelage* qui consiste à restructurer tout ou partie d'une interface utilisateur en fonction des contraintes imposées par le contexte de l'interaction. Il concerne également l'opération de *redistribution* de tout ou partie de l'interface utilisateur sur tout ou partie de l'ensemble des ressources d'interaction disponibles dans l'espace interactif courant. Ces deux opérations peuvent avoir un impact sur tous les niveaux d'abstraction d'un système interactif, allant du plus superficiel réarrangement de l'interface utilisateur à une réorganisation en profondeur du noyau fonctionnel ou du modèle de tâche. Suivant la transformation envisagée et la nature de l'interface utilisateur, le remodelage peut agir sur tous les aspects décrits par les propriétés CARE, de la multimodalité en complémentarité-synergique aux interfaces graphiques monomodales, en passant par les interfaces utilisateur post-WIMP.

De plus, les opérations de remodelage et de redistribution doivent être capables d'agir à tous les niveaux de granularité d'une IHM, de l'interacteur élémentaire à l'ensemble du système interactif, tout en garantissant une reprise de l'état de l'interaction au niveau « action-utilisateur ». Comme l'informatique ambiante implique une grande hétérogénéité des dispositifs, un système interactif plastique est condamné à devoir traiter simultanément avec plusieurs espaces technologiques. Par exemple, son interface utilisateur peut être construite à partir d'un mélange d'éléments de nature différente, certains programmés en Tk, d'autres en Swing, les suivants à l'aide d'une boîte à outils OpenGL particulière et les derniers issus d'un processus de génération automatique. Enfin, l'adaptation du système interactif plastique doit être dynamique et effectuée sous le contrôle de l'utilisateur par l'intermédiaire d'une méta-IHM appropriée.

Ce chapitre présente mes contributions conceptuelles permettant de couvrir ces aspects de la plasticité pour la phase d'exécution.

1. Introduction

Comme je l'évoquais au chapitre 1, j'adopte le génie logiciel comme angle d'attaque de la question de la plasticité des systèmes interactifs, mon approche étant guidée par quatre observations sur les pratiques et état de l'art actuels.

Premièrement, la communauté de l'ingénierie logicielle pour l'interaction homme-machine a développé un processus de conception qui sert maintenant de référence à beaucoup d'outils et de méthodes. À partir d'un modèle de tâche initial, une interface utilisateur abstraite (AUI) est obtenue, laquelle permet l'élaboration d'une interface concrète (CUI) qui donnera lieu à une interface finale (FUI) produite pour un contexte d'interaction particulier. Si ce processus de conception fonctionne, il n'est pas complètement adapté à l'informatique ambiante qui implique une organisation des tâches de l'utilisateur tout à fait opportuniste et imprévisible.

En deuxième lieu, les outils et les mécanismes logiciels traditionnels maintiennent une séparation nette entre les phases de conception et les phases d'exécution d'un logiciel. Cette séparation rend difficiles les adaptations dynamiques qui s'appuient sur des descriptions sémantiquement riches, établies lors de la conception. Par exemple, lors du processus de conception classique en IHM, les liens entre l'interface finale (FUI) et le modèle de tâches initial sont perdus. Dans ce cas, il est très difficile de calculer un remodelage de l'interface utilisateur qui dépasse le niveau superficiel de son apparence.

En troisième lieu, la génération automatique des IHM à partir de modèles de haut niveau d'abstraction est une approche intéressante et indispensable dans le cas où aucune interface utilisateur traditionnelle n'est disponible. Cependant, puisque ces techniques ne produisent que des interfaces utilisateur simples, voire simplistes, ces techniques ne peuvent être envisagées seules. En effet, la sophistication des interfaces multimodales ou post-WIMP implique des modèles génératifs dont l'écriture pourrait être bien plus complexe que la programmation directe de l'interface finale (FUI) avec la boîte à outils appropriée. De plus, les outils de génération traditionnels ne ciblent qu'une seule boîte à outils. Ainsi, ils ne sont pas en mesure de franchir, comme mon espace problème l'impose, différents espaces technologiques.

En dernier lieu, l'adaptation dynamique des logiciels a été traitée de bien des manières ces dernières années, notamment en utilisant les techniques issues du domaine de l'intelligence artificielle, de l'ingénierie dirigée par les modèles (MDE), des intergiciels, des approches à composants et des approches à services. Les différentes solutions ont été développées sans jamais prendre en compte les requis spécifiques posés par l'interaction homme-machine. Par exemple, la reconfiguration dynamique d'un intergiciel peut être considérée acceptable si elle préserve la cohérence sémantique du système. Elle n'est pas directement observable par l'utilisateur, contrairement à une adaptation par remodelage ou redistribution de l'interface utilisateur. Ainsi, une adaptation dynamique d'une IHM ajoute des contraintes

supplémentaires, comme, par exemple, rendre explicite pour l'utilisateur la transition entre l'interface utilisateur de départ et celle d'arrivée, pour que, comme le préconise Norman, l'utilisateur soit en mesure d'évaluer le nouvel état.

Mes contributions visent à satisfaire les requis logiciels posés par l'espace problème du chapitre II en m'appuyant sur les quatre observations précédentes. Ces contributions s'organisent en deux phases. La première phase consiste à identifier l'ensemble des fonctions indispensables à la plasticité, à haut niveau d'abstraction, ainsi que les relations qu'elles partagent. Cette décomposition fonctionnelle conduit à ajouter des requis logiciels à ceux établis précédemment. Le chapitre IV montre que les solutions présentes dans l'état de l'art en génie logiciel ne sont pas satisfaisantes, sous leurs formes actuelles, au regard de l'ensemble des requis logiciels. Ainsi, la deuxième phase de mes travaux consiste à combiner certaines approches existantes sous la forme d'un modèle à composants dynamiques, afin de constituer une mécanique logicielle générale adaptée à la plasticité. Par mécanique logicielle générale, j'entends un ensemble d'outils conceptuels indépendants d'un espace technologique particulier mais aussi d'une classe particulière de systèmes interactifs. La suite de ce chapitre détaille les deux phases dans lesquelles s'inscrivent mes contributions.

2. Décomposition fonctionnelle

La première phase de mes contributions consiste donc à dériver de l'espace problème de la plasticité l'ensemble des fonctions nécessaires à l'exécution de systèmes interactifs plastiques. Cette décomposition en fonctions reste à haut niveau d'abstraction, c'est-à-dire indépendante des langages et des technologies d'implémentation. L'espace problème établi au chapitre II caractérise, à gros-grain, un système interactif plastique par sa capacité à s'adapter ou à être adapté dynamiquement au contexte de l'interaction. De cette caractéristique principale résultent deux fonctions centrales pour la plasticité : les mécanismes de capture du contexte de l'interaction, et les mécanismes liés à l'adaptation dynamique. Ma décomposition fonctionnelle à gros-grains s'appuie sur les travaux de [Rey05] qui proposent une modélisation formelle du contexte de l'interaction associée à une infrastructure de capture du contexte, et sur les travaux de [Calvary01] qui proposent un processus en trois étapes pour l'adaptation dans le cadre de la plasticité.

2.1 Un modèle pour le contexte de l'interaction

Le modèle de [Rey05] décrit le contexte de l'interaction sous la forme d'un réseau de contextes *Rc* défini sur trois ensembles et un

prédicat. L'ensemble *Entités* est celui des entités considérées par le concepteur du système interactif, l'ensemble *Relations* celui des relations possibles entre les entités et l'ensemble *Rôles* celui des rôles que les entités peuvent endosser. Le prédicat *joueRôle(e,r)* se vérifie si et seulement si l'entité *e* joue le rôle *r*, tel que $e \in Entités$ et $r \in Rôles$. Chaque nœud de *Rc* est un *contexte*, noté C_i, défini par le couple $(Rôles_i, Relations_i)$, tel que $Rôles_i \subseteq Rôles$ et $Relations_i \subseteq Relations$. Un changement de contexte a lieu lorsque l'ensemble $Rôles_i$ des rôles remplis change (au moins un rôle apparaît ou disparaît) ou lorsque l'ensemble $Relations_i$ des relations entretenues est altéré (au moins une relation apparaît ou disparaît).

Chaque *contexte* C_i de *Rc* se décompose en un réseau de *situations* tel que chaque situation, notée S_j, partage les même ensembles de rôles $Rôles_i$ et de relations $Relations_i$. Une situation est définie sur trois ensembles (*Ent*, *AssoRôlesEntités* et *AssoRelationsEntités*) et un prédicat *estPrésent(e)*. *Ent* constitue l'ensemble des entités présentes dans la situation considérée tel que $Ent \subseteq Entités$. Le prédicat *estPrésent(e)* est vrai si et seulement si l'entité *e* est présente dans la situation *S*. L'ensemble *AssoRôlesEntités* est celui des associations entre les rôles appartenant à $Rôles_i$ et les entités appartenant à *Ent*. L'ensemble *AssoRelationsEntités* est celui des associations entre les relations appartenant à $Relations_i$ et les entités appartenant à *Ent*. Il y a changement de situation si l'une des conditions suivantes est remplie : l'ensemble *Ent* change (une entité apparaît ou disparaît), l'ensemble *AssoRôlesEntités* change (une association entre un rôle et une entité change) ou l'ensemble *AssoRelationsEntités* change (une association entre les relations et les entités change).

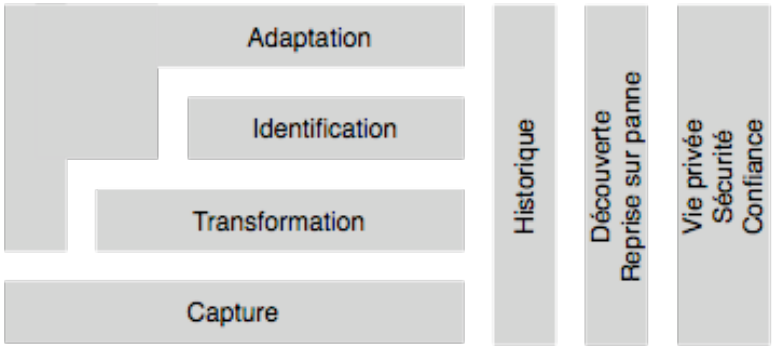


Figure V-1 La pyramide du contexte et sa relation avec le système interactif [Rey05].

[Rey05] exploite sa modélisation du contexte de l'interaction au sein d'une infrastructure logicielle de capture du contexte. Cette infrastructure se structure en couches suivant un modèle qu'il nomme la *pyramide du contexte*. Cette pyramide comporte quatre

niveaux d'abstraction : capture, transformation, identification et adaptation (voir Figure V-1).

Le niveau le plus bas est celui de la *capture*. Il abstrait la diversité des capteurs physiques sous la forme d'observables numériques, c'est-à-dire des grandeurs physiques rendues accessibles au monde numérique. La couche *capture* est exploitée par la couche *transformation*. Cette seconde couche construit des observables symboliques, à haut niveau sémantique, en transformant les observables numériques de la couche inférieure en éléments de type *entité* et *relations,* comme définis dans le modèle du contexte. Les observables symboliques sont organisés, au niveau de la couche *identification*, en réseau de situations et de contexte, afin de permettre la résolution de requête de haut niveau d'abstraction. Enfin, le plus haut étage de la pyramide est constitué de la couche *adaptation* qui, de manière analogue à l'adaptateur de noyau fonctionnel de [Uims92], permet la liaison entre l'infrastructure de capture du contexte et la couche applicative qui l'utilise.

2.2 Un processus d'adaptation en trois étapes

Par ailleurs, les travaux de [Calvary01] introduisent un processus d'adaptation de l'IHM, adapté à la plasticité, en trois étapes : identification de la situation courante, puis calcul de la réaction, et finalement exécution de cette réaction. L'objectif de la première étape est de détecter des changements de contexte (et de situation) et d'identifier ces changements à partir de données obtenues par sondage du contexte de l'interaction. Le produit de cette étape alimente la fonction de calcul de la réaction. Cette fonction de calcul de la réaction peut être abordée suivant différentes approches. Par exemple, [Ganneau07] s'appuie sur le paradigme « événement — condition — action » (ECA). Les événements peuvent être un changement de contexte (ou de situation) ou un événement système, la condition portant sur des observables numériques ou symboliques de la situation courante, et l'action concernant une adaptation (par remodelage ou distribution) ou le déclenchement d'une autre règle d'adaptation. L'ensemble des actions déclenchées lors de l'exécution de cette fonction constitue un plan d'adaptation. Ce plan d'adaptation est transmis à la fonction d'exécution de la réaction. Cette réaction se compose d'un prologue (suspension des tâches en cours, sauvegarde de l'état d'exécution, etc.), puis de la mise en œuvre de la réaction, et finalement d'un épilogue (reprise de l'état d'exécution, restauration des tâches en cours).

À gros-grain, ma proposition de décomposition fonctionnelle (voir Figure V-2) s'articule autour de fonctions et niveau d'abstraction mis en place par [Calvary01] et [Rey05]. Ainsi, elle se constitue d'une infrastructure de capture du contexte de l'interaction, qui couvre les niveaux *capture* et *transformation* de la pyramide du

contexte de [Rey05]. Cette infrastructure alimente un « gestionnaire de l'adaptation » qui reprend les trois fonctions du processus de [Calvary01]. La première fonction, « l'identificateur de situation », couvre les niveaux *identification* et *adaptation* de [Rey05], de manière à alimenter un « moteur d'évolution ». Cette deuxième fonction est en charge de composer un plan d'adaptation qui sera transmis au « producteur de l'adaptation » dont la fonction est de le mettre en œuvre. Dans le cadre de cette thèse, je focalise mon étude sur le « gestionnaire de l'adaptation », le thème de l'infrastructure de capture du contexte ayant déjà été traité dans [Rey05]. Tout en restant au niveau d'abstraction que je me suis fixé, j'affine à présent les fonctions du « gestionnaire de l'adaptation » qui peuvent l'être.

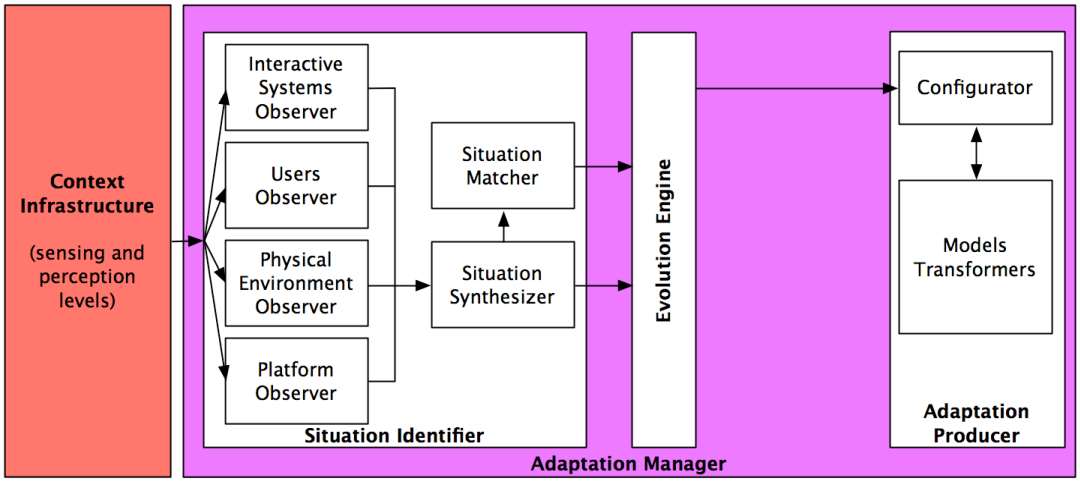


Figure V-2 Les fonctions de capture du contexte de l'interaction et de l'adaptation. Les boîtes englobantes représentent de grandes fonctions, les boîtes englobées des sous-fonctions. Les flèches dénotent des échanges d'information et leurs directions.

2.3 Une fonction « identification » spécialisée pour la plasticité

À grains plus fins, « l'identificateur de situation » se décompose en deux classes de fonctions : les fonctions « observateurs » d'une part, et les fonctions « synthétiseur de situation » et « comparateur de situation » d'autre part. Les observateurs sont chargés de filtrer et de recombiner de manière adéquate les observables symboliques issus de l'infrastructure de capture du contexte afin d'alimenter le « synthétiseur de situation ». Ces observateurs se répartissent en quatre classes. Comme l'adaptation des systèmes interactifs plastiques est fonction du contexte de l'interaction, les trois premières classes d'observateurs, incarnées par les observateurs de la plate-forme, de l'environnement physique et social et de l'utilisateur, couvrent l'ensemble des observables symboliques caractéristiques des trois dimensions du contexte de l'interaction.

À ces trois classes d'observateurs s'ajoute une quatrième relative à l'observation du système interactif. En effet, si la plate-forme est dynamique et le système interactif distribué sur tout ou partie de ses composantes, le retrait d'une plate-forme élémentaire peut entraîner la disparition d'une partie du code en cours d'exécution du système interactif. A contrario, lorsqu'une plate-forme élémentaire s'ajoute à la plate-forme de l'utilisateur, elle peut donner accès à de nouvelles entités logicielles mieux adaptées à la nouvelle situation. Dans les deux cas, le gestionnaire d'adaptation doit être en mesure de connaître l'ensemble des entités logicielles disponibles ainsi que la constitution et la répartition du système interactif afin de l'adapter convenablement en cas de variation de la plate-forme.

Les observables produits par les quatre classes d'observateurs sont organisés sous forme d'une « situation » par le « synthétiseur de situation ». Cette situation modélise l'état courant du contexte dans lequel se déroule l'interaction entre l'utilisateur et le système interactif, les tâches utilisateurs sous-tendues par celui-ci et son état courant de distribution sur la plate-forme. Ainsi, il est du ressort du « synthétiseur de situation » de détecter le changement d'une situation vers une autre et d'un contexte vers un autre. La modélisation de cette nouvelle situation courante est ensuite transmise au « moteur d'évolution » d'une part et à la fonction « comparateur de situation » d'autre part. La fonction « comparateur de situation » vise à établir une correspondance avec une situation d'interaction envisagée lors de la conception ou rencontrée précédemment lors de l'exécution du système interactif. Le « comparateur de situation » indique alors au « moteur d'évolution » si la situation courante correspond à une situation connue ou pas. Ainsi, selon que la situation courante est une situation d'interaction prévue à la conception ou rencontrée par le passé, le « moteur d'évolution » est en mesure d'appliquer, soit les règles d'adaptation définies par le concepteur, soit celles qui auront été inférées lors des expériences précédentes. Si le « comparateur de situation » est incapable d'établir une correspondance avec une situation d'interaction connue, le « moteur d'évolution » doit donc mettre en œuvre d'autres types de mécanismes pour proposer un plan d'adaptation du système interactif.

2.4 Mise en œuvre de l'adaptation

Le plan d'adaptation calculé par le « moteur d'évolution » est transmis au « producteur de l'adaptation », chargé de le mettre en œuvre. Ce plan contient un ensemble d'opérations à réaliser sur le système interactif en vue de l'adapter. La nature de ces opérations dépend de la nature logicielle du système interactif. Dans le cadre de la plasticité, je partitionne la nature logicielle des systèmes interactifs en deux classes (voir Figure V-3). Ma première classe, que j'appelle « traditionnelle » comprend les systèmes interactifs

développés suivant un processus de conception classique où les modèles de conception sont progressivement réifiés pour obtenir un programme exécutable. Ma deuxième classe, que j'appelle « IDM[57] », regroupe les systèmes interactifs développés suivant une approche dirigée par les modèles, où les modèles de haut niveau d'abstraction ne sont pas « consommés » par le processus de réification, mais, au contraire, sont sémantiquement liés les uns aux autres puis embarqués au sein du programme exécutable final en vue d'être utilisés pendant son exécution.

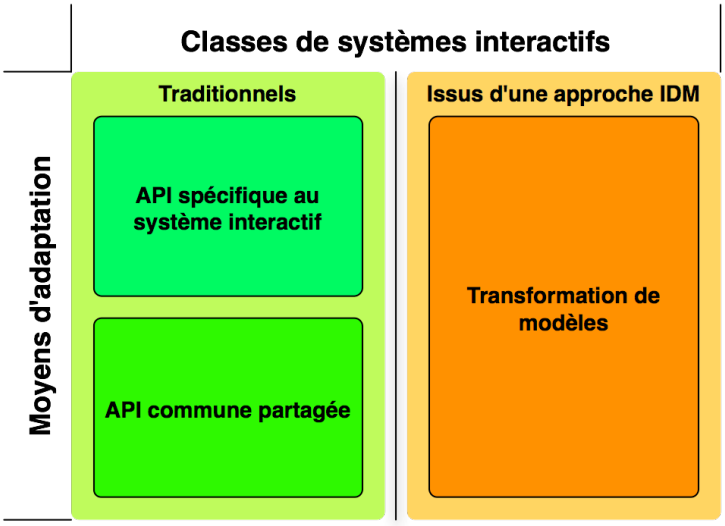


Figure V-3 Moyens d'adaptation des systèmes interactifs en fonction de leur classe logicielle.

Les systèmes interactifs issus de la classe « traditionnelle » sont adaptables par le biais d'une API[58], c'est-à-dire un ensemble d'opérations paramétrables définies à la conception. L'API constitue à la fois la seule connaissance que le « producteur de l'adaptation » possède sur les systèmes interactifs de cette classe et le seul moyen qu'il ait pour les reconfigurer. Une API peut être soit conçue spécifiquement pour un système interactif donné, soit conçue en fonction de mécanismes d'adaptation généraux. Dans le premier cas, la conception des mécanismes d'adaptation se fera en fonction de l'API offerte par le système interactif. Dans la deuxième approche, cette relation de dépendance est renversée : les systèmes interactifs doivent être conçus en fonction de l'API offerte par les mécanismes d'adaptation.

[57] IDM : Ingénierie dirigée par les modèles.

[58] API : Application Programming Interface : interface de programmation logicielle

Par ailleurs, les systèmes interactifs issus de la classe « IDM », parce qu'ils embarquent des modèles de haut niveau d'abstraction, sont adaptables par transformation de modèles. Par exemple, à partir d'un modèle de tâches et d'un modèle de concepts, il est possible de générer l'IHM finale la plus adaptée à la situation courante [Mori03]. Si certains modèles de haut niveau d'abstraction manquent, des approches par réingénierie [Bouillon04] permettent de les recalculer à partir de modèles plus concrets, voire à partir de la FUI.

Comparées aux techniques de conception d'IHM traditionnelles utilisées par les concepteurs humains, les approches par réingénierie et génération automatique ne permettent pas d'obtenir des résultats de très bonne qualité. Cependant, elles ont le mérite d'apporter des solutions lorsqu'aucune IHM développée traditionnellement n'est disponible pour une situation donnée. Ainsi, en plasticité, les approches par transformation de modèle sont indispensables et complémentaires des approches traditionnelles. Afin de prendre en considération les moyens d'adaptation de ces deux classes de systèmes interactifs, la fonction « producteur de l'adaptation » se décompose en deux sous-fonctions : l'une, le « transformateur de modèles », consacrée à la transformation de modèles pour l'IHM et l'autre, le « configurateur », destinée à l'adaptation par le biais d'API.

2.5 Localisation des fonctions d'adaptation

À haut niveau d'abstraction, les trois grandes fonctions du « gestionnaire de l'adaptation » se résument donc à « l'identificateur de situation », au « moteur d'évolution » et au « producteur de l'adaptation ». S'il est possible de détailler les sous-fonctions de « l'identificateur de situation » et du « producteur de l'adaptation », il n'est pas possible de les préciser plus finement sans faire d'hypothèses supplémentaires sur leur implémentation. Notamment, la question de leur localisation influe directement sur les principes à mettre en œuvre pour les réaliser. Cette question admet un spectre de réponses : à une extrémité, toutes les fonctions du gestionnaire de l'adaptation sont intégrées au système interactif lui-même, à l'autre extrémité, elles sont toutes externalisées dans un intergiciel. Ces deux extrêmes correspondent aux notions de reconfigurations internes et externes proposées par [Oreizy99]. Entre ces deux possibilités, les différentes ventilations de ces fonctions d'adaptation entre système interactif et intergiciel forment autant de réponses acceptables à la question de leur localisation.

Chacune des solutions du spectre de réponses a un intérêt qui lui est propre. À la première extrémité, toutes les fonctions nécessaires à la plasticité sont internes au système interactif. Le développeur gagne une liberté totale dans le choix des techniques logicielles pour implémenter le système interactif et ses

mécanismes d'adaptation. Comme les mécanismes d'adaptation ne concernent qu'un système interactif en particulier, ceux-ci peuvent être efficacement optimisés. En revanche, d'une part, les mécanismes développés seront difficilement transposables à un autre système interactif et, d'autre part, les adaptations qui consistent, comme dans le projet Aura [Souza03], à remplacer un système interactif entier par un autre seront difficilement réalisables. À l'autre extrémité du spectre, l'ensemble du « gestionnaire de l'adaptation » est conçu avec l'objectif d'être partagé par tous les systèmes interactifs plastiques. D'autre part, des mécanismes d'adaptation externalisés constituent une solution incontournable pour la réutilisation de systèmes patrimoniaux. De dimension plus générale, les mécanismes d'adaptation seront par définition moins optimisés que leurs homologues de l'approche précédente et leur utilisation inévitablement plus coûteuse en termes de ressources systèmes. En outre, cette approche nécessite la définition d'une API d'adaptation à laquelle l'ensemble des systèmes interactifs devra se conformer.

Entre ces deux extrêmes, plusieurs approches mixtes sont imaginables. Par exemple, certaines fonctions comme les « observateurs » et le « synthétiseur de situation » pourraient être offertes par l'intergiciel, et toutes les autres embarquées au sein du système interactif. Une autre solution consisterait en des systèmes interactifs qui embarqueraient d'une part des mécanismes d'adaptation appropriés pour un ensemble prédéfini de situations d'interaction et, d'autre part, qui se reposeraient sur l'intergiciel lorsque la situation rencontrée n'entre pas dans le cadre de ses possibilités d'adaptation. Bien d'autres solutions sont envisageables. Il est difficile de déterminer a priori si une approche est meilleure qu'une autre. En revanche, il est intéressant de laisser ouvert l'ensemble des possibilités et de permettre, comme dans le modèle Arch [Uims92], un certain effet « Slinky[59] » dans l'allocation de ces fonctions. C'est dans cette optique que j'affine ma décomposition fonctionnelle de la plasticité.

2.6 Un modèle en couche

Un premier affinage de ma décomposition fonctionnelle se présente sous la forme d'un empilement de couches logicielles et matérielles (voir Figure V-4). Cette représentation en couches et la sémantique qui s'y associe sont classiques en informatique : la surface d'une couche inférieure symbolise l'API qu'elle offre aux couches qui lui sont supérieures. Ici, du matériel aux systèmes interactifs, le système se décompose en quatre couches : de la plus

[59] En référence au Slinky Toy. Ce célèbre jouet inventé en 1943 se présente sous la forme d'un ressort capable, entre autres, de descendre des escaliers. Voir http://www.poof-slinky.com/history.asp

basse à la plus haute, se trouvent la couche « matériel », la couche « systèmes d'exploitation », la couche « intergiciel » et enfin, la couche applicative dédiée aux systèmes interactifs. La couche « matériel » représente l'ensemble des plates-formes élémentaires qui composent la plate-forme de l'utilisateur. Ces plates-formes élémentaires se caractérisent par la puissance et le type de leur microprocesseur, la bande passante offerte par leur technologie réseaux, les capteurs et actuateurs qu'elles contrôlent et les ressources d'interaction dont elles disposent.

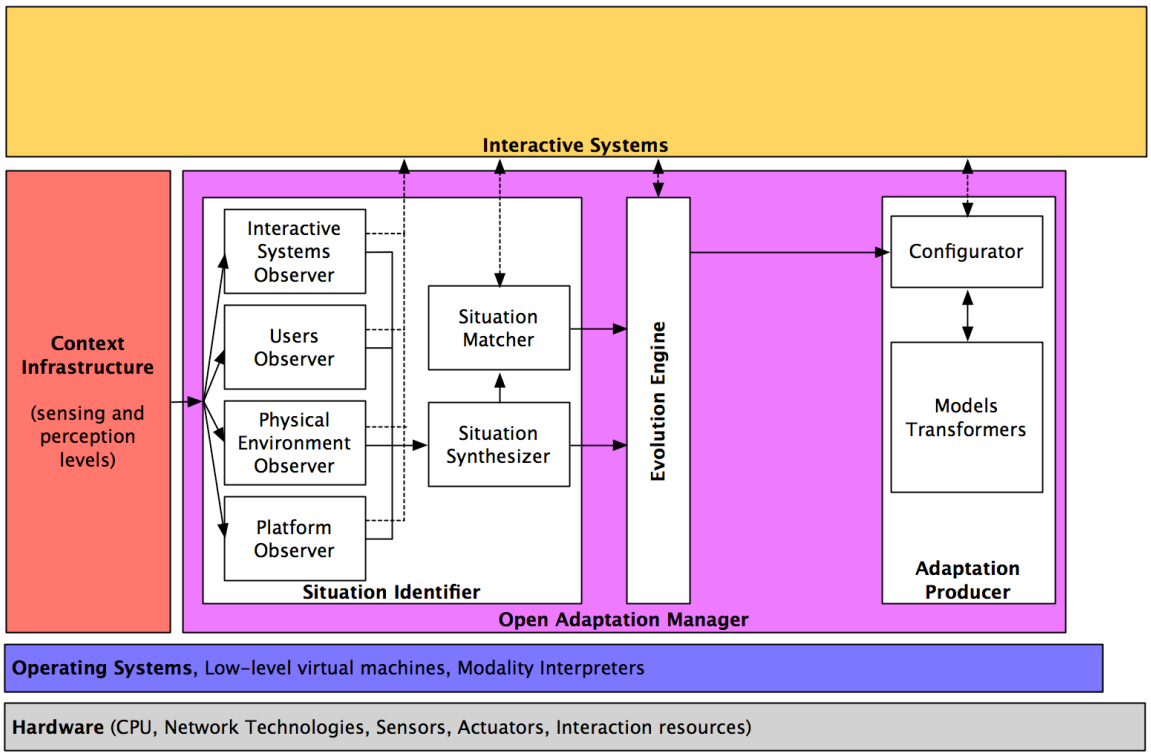


Figure V-4 Décomposition fonctionnelle : un premier affinage. Dans ce modèle en couches, les côtés supérieurs des boîtes offrent une API à destination des couches supérieures. Les flèches continues dénotent des échanges d'information et leur direction. Les flèches en pointillé indiquent les fonctions rendues accessibles par API aux couches supérieures.

Sur cette couche « matériel » repose la couche « système d'exploitation ». Ici, la notion de système d'exploitation est à prendre au sens large : je regroupe également sous ce terme les machines virtuelles, par exemple Java ou .Net, et les interpréteurs de modalité. Par interpréteur de modalités, je désigne les boîtes à outils spécialisées dans le rendu d'interface utilisateur. Par exemple, Swing[60], OpenGL[61] et IAM [Lachenal04] sont des

[60] http://java.sun.com/javase/6/docs/technotes/guides/swing/index.html

[61] http://www.opengl.org

interpréteurs de modalité dédiés aux interfaces utilisateur graphiques. DirectSound[62] et ALSA[63] sont des interpréteurs de modalités spécialisés pour les interfaces utilisateur sonores.

Au-dessus des couches « matériel » et « système d'exploitation » se trouve la couche intergiciel pour la plasticité. Elle héberge l'infrastructure de capture du contexte de l'interaction et la version intergiciel du gestionnaire de l'adaptation. Cette couche offre un ensemble de fonctions à l'usage de la couche applicative. Les flèches en pointillés détaillent les interactions entre les fonctions du gestionnaire de l'adaptation de niveau intergiciel et la couche des « systèmes interactifs ». Leurs sens exprime la direction des échanges de données. Ainsi, les « observateurs » peuvent alimenter la couche « systèmes interactifs » en observables symboliques, mais l'inverse n'est pas possible. Pour respecter le modèle de [Rey05], un système interactif doit passer par l'intermédiaire des niveaux capture et transformation de l'infrastructure de capture du contexte pour alimenter les « observateurs ». En revanche, les échanges avec toutes les autres fonctions du « gestionnaire de l'interaction » peuvent être bidirectionnelles. Voici quelques exemples d'échanges qu'il est possible d'envisager entre chacune de ces fonctions et la couche applicative.

- Le « synthétiseur de situation » et le « comparateur de situation » pourraient notifier directement un changement de situation à un système interactif qui embarquerait son propre moteur d'évolution. Dans l'autre sens, un système interactif pourrait indiquer la description d'une situation à laquelle il est capable de s'adapter au « comparateur de situation », afin que ce dernier soit en mesure de la reconnaître, si celle-ci se présentait.

- Suite à un changement de situation, le « moteur d'évolution » pourrait souhaiter indiquer un plan d'adaptation à un système interactif qui embarque ses propres mécanismes de reconfiguration. Dans l'autre sens, un système interactif pourrait souhaiter ajouter un ou plusieurs plans d'adaptation préprogrammés lors de la conception au « moteur d'évolution » de l'intergiciel, afin que celui-ci soit en mesure de proposer des solutions d'adaptation plus fines.

- Le « configurateur », chargé de mettre en œuvre les plans d'adaptation issus du « moteur d'évolution », a nécessairement besoin d'interagir avec la couche applicative. Par exemple,

[62] http://www.microsoft.com

[63] http://www.alsa-project.org

dans un premier temps, son action pourrait consister à demander à un système interactif de retourner son état d'interaction et de s'arrêter, puis, dans un deuxième temps de démarrer un autre système interactif. Dans l'autre sens, un système interactif qui possède son propre moteur d'évolution, pourrait demander au « configurateur » de mettre en œuvre une reconfiguration ou au « transformateur de modèles » de réaliser une transformation.

Si les interactions logicielles entre la couche applicative et les fonctions de « l'identificateur de situation » et du « moteur d'évolution » s'expriment au moyen d'API tout à fait classiques, les choses sont différentes pour la relation entre la couche applicative et le « producteur de l'adaptation ». En effet, dans le cas de la collaboration entre « l'identificateur de situation », le « moteur d'évolution » et la couche applicative, les interactions logicielles interviennent entre un ensemble de processus en cours d'exécution. En revanche, si certaines opérations de reconfiguration à la charge du « producteur de l'adaptation » consistent en des interactions logicielles entre processus en cours d'exécution, il faut également en considérer d'autres qui consistent à démarrer et à arrêter l'exécution de certaines entités logicielles qui composent les systèmes interactifs. Ainsi, en plus d'API au sens classique du terme, les couches « intergiciel » et « applicative » doivent partager un accord sur la nature des entités logicielles qui composent les systèmes interactifs et sur les moyens de contrôler leur cycle de vie. Il convient donc d'affiner un peu plus ma décomposition fonctionnelle, afin de fixer cet accord à haut niveau d'abstraction.

2.7 Décomposition fonctionnelle finale

Comme la partie de mon état de l'art consacré au génie logiciel montre que les approches de type « à composants » sont particulièrement adaptées aux problématiques de reconfiguration dynamique, c'est vers ce type de solution que j'oriente ma proposition. Ainsi, dans l'affinage final de ma décomposition fonctionnelle (voir Figure V-5), la couche applicative englobe des systèmes interactifs décomposés en entités logicielles de type « composants » et « connecteurs », au sens de [Shaw95] : un composant est vu comme une « unité de calcul » et les connecteurs comme un « médium » destiné à sous-tendre la communication entre les composants. Dans cette décomposition fonctionnelle, il ne s'agit pas de faire référence à un modèle à composants en particulier, mais d'établir une sorte de cahier des charges de l'approche à composants la plus adaptée à la question de la plasticité des systèmes interactifs. En outre, il ne s'agit pas non plus d'imposer aux développeurs un nouveau modèle à composants et une nouvelle technologie pour implémenter des systèmes interactifs plastiques. Au contraire, l'objectif est de trouver un modèle qui constitue une API pour reconfigurer des

systèmes interactifs construits suivant une approche à composants. Comme toute API, son rôle sera de masquer, pour les mécanismes de la plasticité, l'hétérogénéité des différents modèles et technologies concrètement utilisés pour développer des systèmes interactifs plastiques.

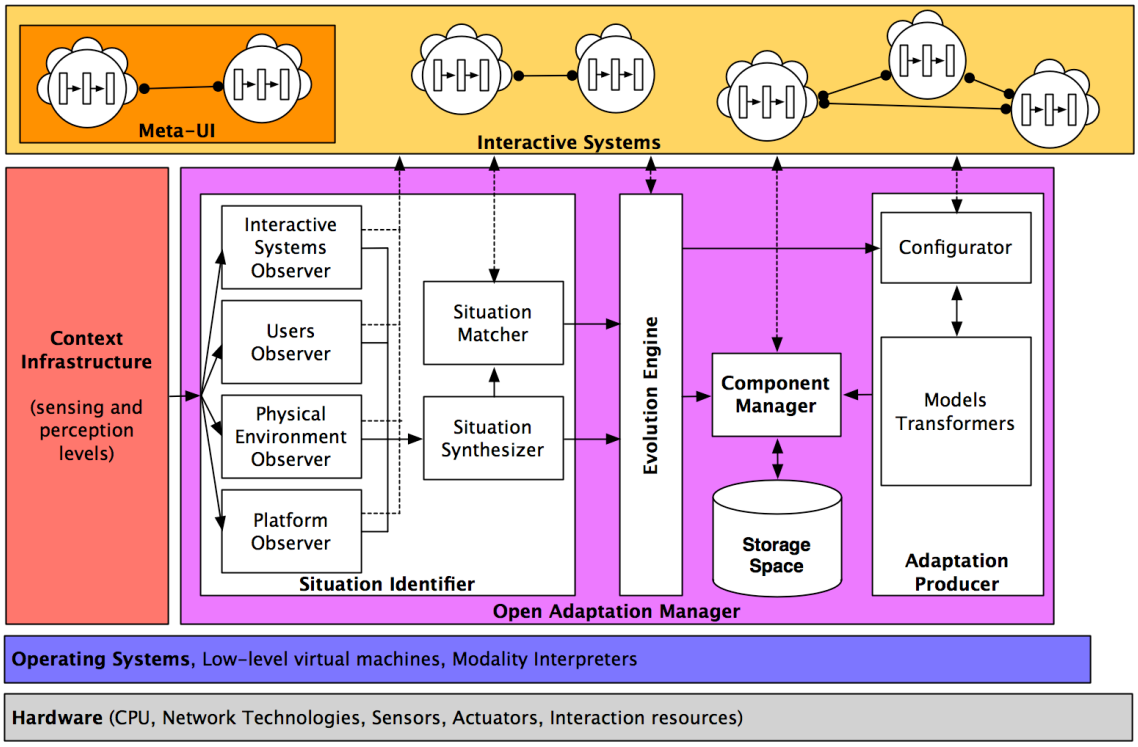


Figure V-5 Décomposition fonctionnelle : affinage final. La sémantique de cette figure est la même que dans la figure précédente. Les symboles en forme de fleur représentent des composants logiciels. Les segments terminés par de petits disques noirs symbolisent des liaisons entre composants.

2.7.1 Composants et connecteurs

Sur la Figure V-5, dans la couche applicative, les systèmes interactifs sont représentés par des assemblages de composants et de connecteurs. Les composants sont incarnés par des symboles en forme de fleurs et les connecteurs par des segments terminés par de petits disques noirs. Les pétales caractérisent la couverture fonctionnelle d'un composant, en termes des fonctions du modèle Arch [Uims92] : les fleurs à cinq pétales figurent un composant qui couvre l'ensemble des fonctions du modèle Arch, les fleurs composées de trois pétales, dans l'exemple de la Figure V-5, couvrent les fonctions allant du contrôleur de dialogue à la présentation physique. L'aspect « protubérant » des pétales dénote une fonction descriptive : dans mon approche, un composant est une entité logicielle « méta-décrite » en termes de rôle architectural assumé et pour chaque rôle l'étendue de sa couverture. Le « corps » du composant contient trois rectangles

reliés par des flèches. Ils incarnent les trois fonctions principales du « gestionnaire de l'adaptation » dans leur version embarquée au sein du composant : conformément à « l'effet Slinky » recherché, aucune, toutes, ou partie, des fonctions du « gestionnaire de l'adaptation » peuvent faire l'objet d'une implémentation au sein de chaque composant. Comme les systèmes interactifs qui s'inscrivent dans le cadre de la couche applicative reposent sur une plate-forme matérielle distribuée et dynamique, la disponibilité dynamique des composants est une caractéristique essentielle de mon approche. Par disponibilité dynamique, j'entends, d'une part, que les composants présents peuvent disparaître à tout moment, et d'autre part, que de nouveaux composants peuvent apparaître, suivant la variation de la plate-forme dont dispose l'utilisateur. En conséquence, les systèmes interactifs sont donc, par nécessité, des assemblages dynamiques de composants.

Au sein d'un assemblage, les composants d'un système interactif interagissent par le biais de connecteurs. Je décompose l'interaction logicielle en deux niveaux : sémantique et syntaxique. Le niveau sémantique est fourni par les composants. Il se constitue de l'ensemble des opérations et des messages que les composants peuvent effectuer ou échanger. Le niveau syntaxique est assuré par les connecteurs. Il couvre les moyens et les protocoles de communication employés pour transmettre les informations entre les composants. En termes des niveaux du modèle OSI [ISO7498], qui décrit les fonctionnalités nécessaires à la communication, les connecteurs, dans mon approche, couvrent les niveaux 4 à 7, c'est-à-dire de la couche transport à la couche application. Ces connecteurs permettent d'établir entre plusieurs composants (au moins deux), des liaisons dites « tardives », c'est-à-dire réalisées lors de l'exécution, et « temporaires », c'est-à-dire qu'elles peuvent être faites et défaites au cours de l'exécution. Ainsi, en plus des adaptations éventuellement réalisées de manière interne à chaque composant, l'adaptation des systèmes interactifs est réalisée par transformation des assemblages de composants.

2.7.2 Opérations d'adaptation propres aux approches à composants - connecteurs

Classiquement, la transformation des assemblages de composants passe par quatre opérations élémentaires : ajout et retrait d'un composant, établissement et suppression d'une liaison. Comme toute entité logicielle, un composant existe sous deux formes : une forme « inerte » lorsqu'il est stocké dans une mémoire de masse, et une forme « instanciée » lorsqu'il est chargé en mémoire vive et en cours d'exécution. L'ajout d'un composant dans un assemblage peut être réalisé à partir d'un composant inerte ou instancié. Si le composant à ajouter est inerte, il doit d'abord être instancié, puis relié, par l'établissement d'une liaison, à au moins un composant de l'assemblage cible. L'établissement d'une liaison consiste en

l'instanciation d'un connecteur et du branchement de celui-ci sur l'ensemble des composants devant partager des interactions logicielles. Le retrait d'un composant est un processus en deux phases : les liaisons qu'il partage avec d'autres composants doivent être supprimées, puis, le composant détruit.

En plus de ces quatre opérations sur les assemblages, suivant le type d'implémentation choisie, le gestionnaire de l'adaptation devra éventuellement être capable de réaliser des opérations supplémentaires sur les composants eux-mêmes. Par exemple, le gestionnaire de l'adaptation pourrait avoir besoin, avant la destruction d'un composant, de sauvegarder l'état courant de son interaction avec l'utilisateur, puis de l'injecter dans le ou les composants chargés de prendre sa suite, afin d'assurer une reprise sur action physique. Cependant, si cette opération et d'autres du même type sont envisageables, elles ne sont pas fonction de l'utilisation d'une approche « composants—connecteur » pour caractériser la nature logicielle des systèmes interactifs. Au contraire, elles dépendent directement de la façon d'implémenter le gestionnaire de l'adaptation. Ainsi, dans ma décomposition fonctionnelle, je m'en tiens à définir les moyens d'adaptation des assemblages de composants par les seules opérations propres à l'approche « composants—connecteur » : l'instanciation et la destruction de composants, l'instanciation, le branchement et la destruction de connecteurs.

2.7.3 Composants exécutables et composants transformables

Mon hypothèse sur la nature logicielle des systèmes interactifs qui peuplent la couche applicative me conduit à distinguer deux types de composants : les composants exécutables et les composants transformables. Cette distinction permet de traiter de manière unifiée dans le « gestionnaire de l'adaptation », l'adaptation des IHM conçues selon les approches traditionnelles et l'adaptation des IHM conçues selon les approches IDM (cf. Figure V-3).

Les plates-formes élémentaires hébergent des composants, sous une forme inerte, afin qu'ils soient exploités, sous une forme instanciée, au sein de systèmes interactifs. Dans l'acception commune, un composant inerte est un code exécutable, c'est-à-dire écrit dans un langage directement interprétable par une machine physique ou virtuelle. Cependant, dans l'espace solution de la plasticité figurent en bonne place les approches par génération automatique d'interfaces utilisateurs à partir de modèles de haut niveau d'abstraction.

Or, il semble judicieux de pouvoir associer les approches par génération automatique aux approches traditionnelles afin d'obtenir des systèmes interactifs mixtes dont l'IHM est constituée de parties conçues par un concepteur humain et d'autres obtenues par génération. L'intérêt est de pouvoir obtenir

par ce biais des solutions lorsque, pour une tâche utilisateur donnée, aucun composant traditionnel ne convient pour la situation courante.

Pour atteindre cet objectif, je considère les modèles pour l'IHM de haut niveau d'abstraction comme des composants. Ainsi, je distingue les composants inertes exécutables des composants inertes transformables. Les composants inertes exécutables sont directement instanciables. En revanche, les composants inertes transformables doivent subir une ou plusieurs étapes de transformations pour devenir exécutables, afin d'être instanciés.

2.7.4 Espace de stockage et gestionnaire des composants

Les composants inertes exécutables et transformables sont regroupés dans un « espace de stockage » représenté par un cylindre dans la Figure V-5. À l'instar de la plate-forme qui se compose d'un ensemble interconnecté de plates-formes élémentaires, l'espace de stockage est réparti entre les différentes plates-formes élémentaires dont dispose l'utilisateur, voire hébergé par des serveurs distants. L'espace de stockage stocke l'ensemble des composants inertes exécutables et transformables disponibles sur la plate-forme.

Le « gestionnaire de composants » a pour objet de faire l'interface entre cet « espace de stockage » et ses clients, c'est-à-dire le « moteur d'évolution », le « producteur de l'adaptation » et la couche applicative. Sa fonction correspond à celle de l'annuaire de services utilisée dans les approches à services : il permet la découverte dynamique de composants. Pour construire un plan d'adaptation, le « moteur d'évolution » interagit avec le « gestionnaire de composants » pour sélectionner les composants les plus adaptés à la situation courante. Pour cela, il formule des requêtes basées sur les propriétés fonctionnelles et extra-fonctionnelles du type de composants qu'il recherche. Par exemple, le « moteur d'évolution » recherche un composant capable de sous-tendre telle *tâche utilisateur* (propriété fonctionnelle), capable de s'insérer dans un système interactif conçu selon telle *architecture logicielle* (propriété extra-fonctionnelle) et utilisant telle *modalité d'interaction* (propriété extra-fonctionnelle).

En retour, le « gestionnaire de composants » doit fournir au « moteur d'évolution » les références et les descriptions de l'ensemble des composants qui correspondent à cette requête. Le « moteur d'évolution » compose son plan d'adaptation avec les références des composants qu'il a retenus et la description des liaisons qu'il convient d'établir. Pour mener à bien l'adaptation, le « producteur de l'adaptation » accède aux composants et à leur description grâce aux références présentes dans le plan d'adaptation. Suivant l'état des composants référencés (inertes ou

instanciés) et suivant leur nature (inerte exécutable ou inerte transformable), le « producteur de l'adaptation » mettra en œuvre les mécanismes idoines de transformation, d'instanciation ou de suppression de composants, puis de suppression et d'établissement de liaisons. Au même titre et de la même manière que le « moteur d'évolution » et le « producteur de l'adaptation », dans le cadre de l'effet Slinky, la couche applicative a la possibilité d'interagir avec le « gestionnaire de composants ». Ainsi, il devient possible d'adopter le mécanisme propre aux approches à services, où le client prend l'initiative de se connecter à un service.

2.7.5 Méta-IHM

Mon hypothèse est que toutes les étapes du processus d'adaptation des systèmes interactifs doivent potentiellement rester sous le contrôle de l'utilisateur. Par conséquent, les fonctions qui composent le « gestionnaire de l'adaptation » sont susceptibles de posséder leur propre IHM. Ces IHM de contrôle sont regroupées sous le concept de « méta-IHM ».

Dans mon approche, la méta-IHM est un système interactif plastique comme les autres. Cependant, son rôle n'est pas de sous-tendre une tâche métier de l'utilisateur, mais de sous-tendre l'ensemble des tâches de gestion des systèmes interactifs. Ainsi, dans ma décomposition fonctionnelle, la méta-IHM s'intègre naturellement au niveau de la couche applicative. À l'instar des autres systèmes interactifs, elle partage une structuration en composants et connecteurs. Enfin, de la même manière que les fonctions du « gestionnaire de l'adaptation » peuvent être distribuées entre les systèmes interactifs et une couche intergicielle, les éléments constituant la méta-IHM peuvent être répartis entre les systèmes interactifs standard et un système interactif dédié, à la manière d'un gestionnaire de fenêtre ou de l'inspecteur de Comets de [Demeure07].

2.7.6 En résumé

En résumé, ma décomposition fonctionnelle comprend quatre couches qui couvrent l'ensemble des niveaux d'abstraction, du matériel aux systèmes interactifs. À gros-grain, les fonctions nécessaires à la plasticité se partagent entre une infrastructure dédiée à la capture du contexte de l'interaction et un gestionnaire de l'adaptation. L'infrastructure de capture du contexte de l'interaction couvre les niveaux « capture » et « transformation » décrits par [Rey05]. Elle se situe au sein de la couche intergicielle de ma décomposition fonctionnelle. Le gestionnaire de l'adaptation se décompose en trois fonctions principales : l'identificateur de situation qui implémentent les niveaux « identification » et « adaptation » de [Rey05], le moteur d'évolution dont le rôle est de calculer un plan d'adaptation suite à un changement de situation et le producteur de l'adaptation chargé de l'application des plans d'adaptation. Chacune de ces fonctions

peut exister au niveau intergiciel comme au niveau applicatif. Dans le premier cas, elles sont génériques et chargées d'adapter les systèmes interactifs de l'extérieur et offrent une API à la couche applicative. Dans le deuxième cas, elles sont spécifiques, internes aux systèmes interactifs et peuvent être optimisées de manière bien plus fine.

Les systèmes interactifs qui s'inscrivent dans la couche applicative de ma décomposition fonctionnelle sont vus comme des assemblages de composants et de connecteurs. Dans mon approche, un composant consiste soit en un code directement exécutable, soit en un modèle pour l'IHM de haut niveau d'abstraction, transformable à la volée, afin d'être exploité au sein d'un système interactif. Dans les deux cas, les composants sont méta-décrits d'un point de vue fonctionnel et extra-fonctionnel. Ils sont issus d'un espace de stockage réparti sur l'ensemble de la plate-forme dont le contenu est rendu disponible par le biais d'un gestionnaire de composants. Pour le moteur d'évolution, ce gestionnaire s'apparente à un annuaire qui permet de connaître et de sélectionner les composants disponibles à un instant donné. Pour le producteur de l'adaptation, le gestionnaire de composants permet d'obtenir la référence vers un composant donné afin de lui appliquer le ou les opérations prévues par le plan d'adaptation.

Ainsi, dans mon approche, l'adaptation des systèmes interactifs est réalisée soit par adaptation interne des composants par des mécanismes embarqués dans les systèmes interactifs, soit par transformation des assemblages de composants par ajout ou retrait de composant et création ou suppression de liaisons entre composants. Le producteur de l'adaptation agit sur les assemblages de composants de la couche applicative à travers une « API à composants » dont le rôle est de masquer l'hétérogénéité des technologies à composants[64] et de communications utilisées pour construire les systèmes interactifs.

2.7.7 Conclusion

Cette décomposition fonctionnelle constitue un cadre intégrateur pour l'ensemble des travaux relatifs à l'exécution de systèmes interactifs plastiques. Par exemple, les travaux de Christophe Lachenal sur IAM [Lachenal04] se situe au niveau de la couche « système d'exploitation ». Ceux de Gaëtan Rey sur la capture du contexte [Rey05] s'insèrent au niveau intergiciel et se répartissent entre la fonction « infrastructure de capture du contexte » et

[64] Technologie à composants : j'appelle technologie à composants l'association d'un modèle à composant particulier et d'un framework qui l'implémente. Par exemple, Fractal (le modèle) et Julia (implémentation Java de Fractal) forme une technologie à composants. Fractal et Think (implémentation C++ de Fractal) en forme une autre.

« identificateur de la situation ». Plus récents, les travaux d'Alexandre Demeure [Demeure07] présentent, d'une part les COMETS, une architecture logicielle qui pourrait s'appliquer à la structuration des assemblages de composants dans la couche applicative et d'autre part, son graphe des descriptions (GDD) constitue un candidat sérieux pour l'implémentation de la fonction « gestionnaire de composants ». Les travaux en cours de Jean-Sébastien Sottet sur la génération automatique d'IHM par transformations de modèles [Sottet07] fournissent des mécanismes pour la fonction « transformateur de modèle ». Enfin, les travaux de Vincent Ganneau sur un moteur d'adaptation pour la plasticité [Ganneau07] propose une approche pour réaliser la fonction « moteur d'évolution » de ma décomposition fonctionnelle.

Par ailleurs, la question de l'interaction entre le producteur de l'adaptation du niveau intergiciel et de la couche applicative reste ouverte. Le cahier des charges envisagé dans ma décomposition fonctionnelle ne correspond à aucune solution de l'état de l'art en Génie Logiciel. En effet, si des approches à composants existantes permettent la reconfiguration dynamique d'assemblages de composants, aucune n'envisage la disponibilité dynamique de ces composants. Cette notion est traitée par les approches à services. Cependant, dans ces approches, aucun tiers ne peut intervenir dans l'établissement de la liaison entre un client et un service. En outre, la gestion du cycle de vie des clients et des services n'est pas envisagée : clients et service sont obligatoirement des processus en cours d'exécution. L'approche de [Cervantes04] tente une forme d'association entre approches à composants et à services, afin d'apporter la notion de disponibilité dynamique aux composants. Si cette voie semble la plus prometteuse pour la plasticité, l'association de [Cervantes04] n'est pas la bonne : dans son approche, une reconfiguration peut avoir lieu sur apparition ou disparition de composants, mais pas sur un événement contextuel comme un changement de situation d'interaction. En outre, si tous les modèles à composants ou à services actuels sont conçus dans l'objectif de donner lieu à une ou plusieurs technologies particulières, aucun n'est envisagé comme moyen pour masquer l'hétérogénéité de ces technologies vis-à-vis d'un tiers, ni comme moyen d'offrir l'interopérabilité entre ces technologies. Ainsi, dans le cadre de la plasticité, il est nécessaire d'explorer une approche nouvelle pour traiter la reconfiguration dynamique des systèmes interactifs plastiques. La section suivante présente ma seconde contribution conceptuelle : Ethylene, un modèle à composants dynamiques destiné à sous-tendre d'une part, l'interaction entre le « producteur de l'adaptation » et la couche applicative, et d'autre part, l'interopérabilité entre technologie à composants hétérogènes.

3. Ethylene : un modèle à composants dynamiques pour la plasticité des IHM

La deuxième phase de mes contributions consiste à proposer un modèle à composants qui soit conçu autour des propriétés satisfaisant les besoins posés par la plasticité. Comme le requiert l'un de ces besoins, ce modèle est un modèle à composants dynamiques, c'est-à-dire de composants disponibles dynamiquement. Son objectif est triple (voir Figure V-6) :

- À la manière d'une API, il doit offrir un moyen uniforme de manipuler les assemblages et les composants de la couche applicative quelles que soient les différentes technologies utilisées pour les développer ;

- Il doit offrir à la couche applicative le moyen de composer des assemblages de composants hétérogènes, c'est-à-dire issus de modèles à composants différents ;

- Il doit offrir un cadre homogène pour décrire des composants implémentés suivant différents modèles à composants.

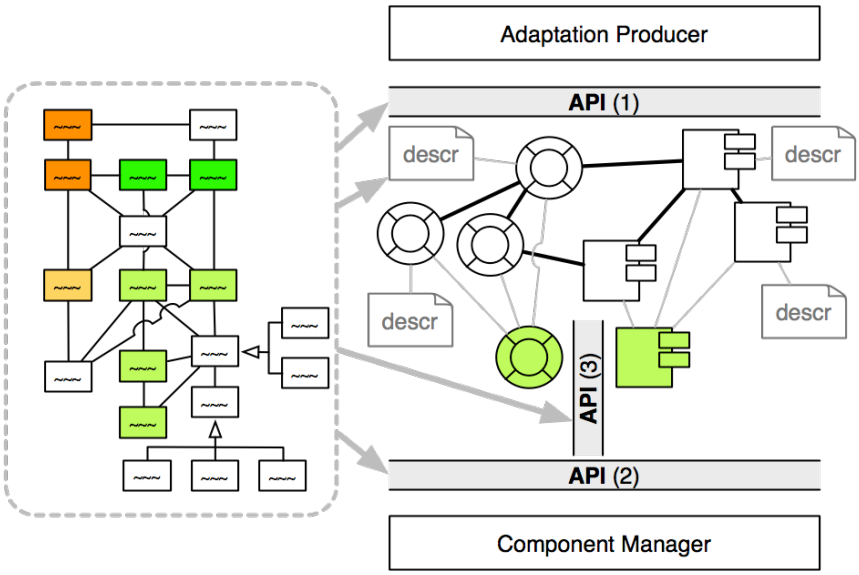


Figure V-6 Ethylene et ses objectifs : le modèle Ethylene (symbolisé en partie gauche) offre un cadre pour décrire des composants hétérogènes et uniformise, via les API *handling* (1) et *availability* (2), les interactions entre, d'une part, des composants hétérogènes et, d'autre part, le producteur de l'adaptation et le gestionnaire de composant. Par ailleurs, via l'API *interoperability* (3), Ethylene permet également l'interopérabilité entre des composants issus de différentes technologies.

Une bonne API est généralement écrite pour résoudre une seule préoccupation dont le périmètre est clairement délimité. Dès lors,

les objectifs fixés nécessitent de produire trois API distinctes : une première, que j'appelle *handling*, est dédiée à l'interaction entre le « producteur de l'adaptation » et la couche applicative, la deuxième, que j'appelle *availability,* est consacrée aux mécanismes liés à la disponibilité dynamique et la dernière, que j'appelle *interoperability*, est destinée à l'interopérabilité entre composants hétérogènes.

Afin d'établir les signatures de fonction et de structure de données nécessaires à ces trois API, il convient de définir l'objet sur lequel elles portent toutes les trois : un modèle de composants logiciels disponibles dynamiquement et conforme aux requis posés par la plasticité. Pour décrire ce modèle, que j'appelle Ethylene[65], j'adopte successivement deux points de vue : le premier aborde cette description sous l'angle du cycle de vie des composants, alors que le second l'aborde sous l'angle des concepts et relations qui définissent la notion de composant et de connecteur. À partir de ce modèle, je propose l'ensemble des opérations et les diagrammes d'activité associés qui forment ces trois API.

3.1 Cycle de vie d'un composant Ethylene

Le cycle de vie d'un composant fait référence à la succession d'étapes qu'il est amené à traverser au cours de sa vie. Celui d'Ethylene s'étend des étapes de spécification à la fin de l'exécution d'un composant. Ce cycle de vie est construit autour de deux propriétés : la disponibilité dynamique et la nature logicielle des composants Ethylene. En effet, un composant Ethylene est soit constitué, classiquement, de code exécutable, soit constitué d'un ou de plusieurs modèles de plus haut niveau d'abstraction. Contrairement à un composant classique, un composant constitué de modèles doit subir une ou plusieurs étapes de transformation successives jusqu'à devenir exécutable par une machine, une machine virtuelle ou un interpréteur.

Ainsi, le cycle de vie Ethylene (voir Figure V-7), fortement inspiré par celui des paquetages de composants d'OSGi[66] [Cervantes04] avec lequel il partage la préoccupation de disponibilité dynamique, se découpe en douze états répartis en trois phases. L'unique état de la première phase regroupe l'ensemble des activités de conception d'un composant. Il est représenté sur la figure par un encadrement continu épais.

---

[65] L'éthylène, C_2H_4, est un hydrocarbure insaturé de la famille des alcènes dont il est la molécule la plus simple. L'éthylène est à la base d'un grand nombre de molécules dans l'industrie chimique. Avec ses dérivés immédiats, il est à la source d'un grand nombre de polymères et de matières plastiques. En anglais, éthylène s'écrit ethylene. Par pragmatisme, je choisis l'orthographe anglaise pour nommer mon modèle à composants dynamiques.

[66] OSGi Alliance : voir http://www.osgi.org/About/Technology

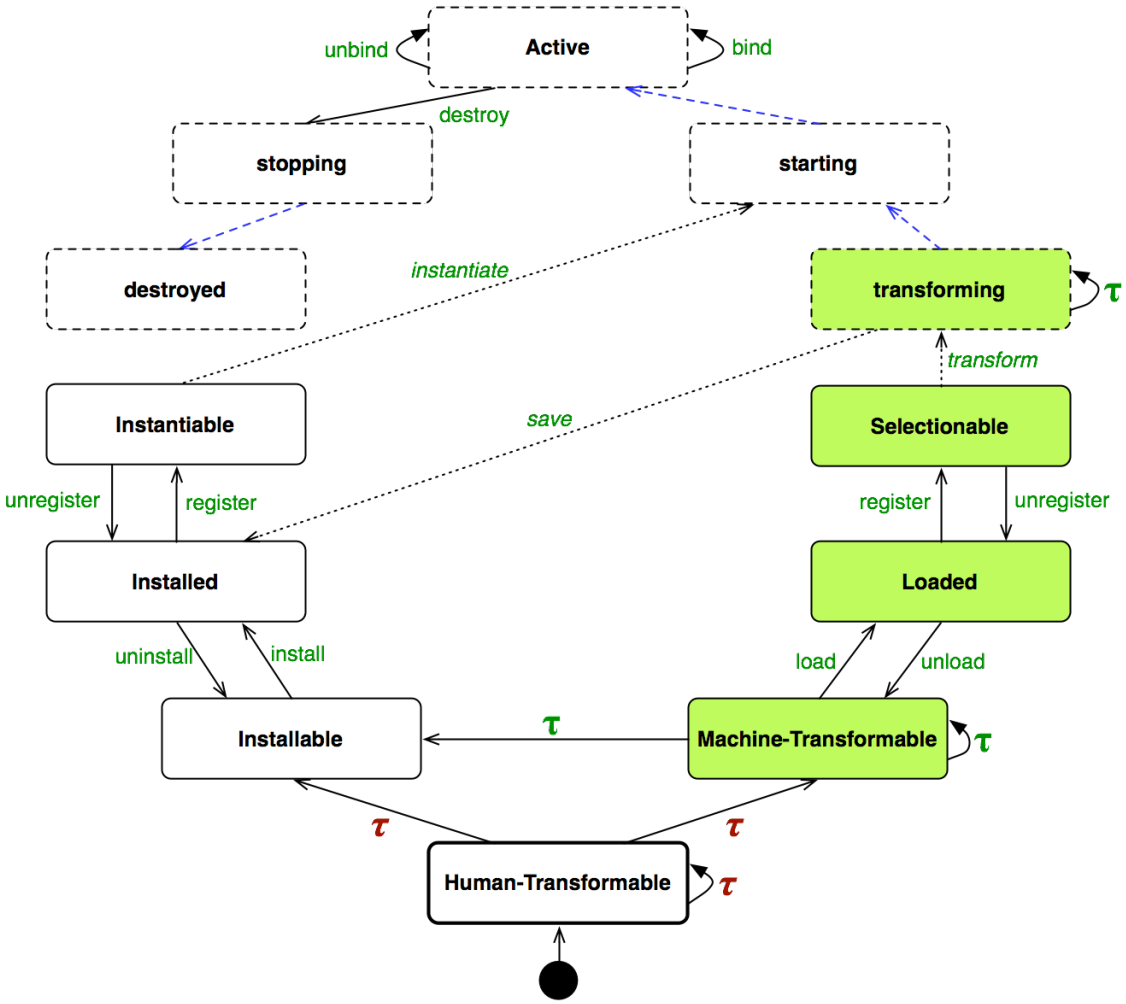


Figure V-7 Cycle de vie des composants Ethylene. Les nœuds du graphe sont des états. Le style de l'encadrement de chaque état dénote son appartenance à l'une des trois phases du cycle de vie (*conception*, *inerte*, *instance*). Les états colorés sont spécifiques aux composants « à modèles ». Les flèches dont le trait est continu sont des transitions. Les transitions figurées par des flèches dont le trait est discontinu sont automatiques. Les flèches en pointillé dont les étiquettes sont écrites en italique, dénotent la création d'une nouvelle entité. Le disque noir surmonté d'une flèche pointe l'état initial du cycle de vie.

Cet état constitue logiquement l'état initial du cycle de vie Ethylene. Il donne accès aux six états de la deuxième phase qui décrivent les différents états d'un composant lorsqu'il se présente sous sa forme « inerte ». Sur la figure, les états de la deuxième phase sont identifiés par un encadrement continu fin. Les cinq états de la troisième phase concernent le déroulement de la vie d'un composant lorsqu'il se présente sous sa forme « instanciée », c'est-à-dire lorsqu'il est chargé en mémoire vive afin d'y être exécuté. Ils sont désignés sur la figure par un encadrement en pointillé fin. Par ailleurs, le schéma illustre par un remplissage coloré les états spécifiques aux composants constitués de modèle

de haut niveau d'abstraction. Dans mon approche, le contrôle et la gestion du cycle de vie d'un composant est assuré par une entité logicielle que j'appelle *fabrique*[67]. Son rôle est de sous-tendre les différentes opérations qui sont à l'origine des transitions entre les états. Je détaille à présent le rôle et la couverture de chacun des douze états de ce cycle de vie et leurs transitions.

3.1.1 État initial : `Human-Transformable`

L'état initial du cycle de vie Ethylene s'enracine dans les phases de conception d'un composant. À ce stade, le composant n'existe que sous la forme de spécifications et de modèles de différents niveaux d'abstraction obtenus par l'application d'un processus de conception mis en œuvre par un concepteur humain. Tous ces modèles et ces spécifications, à cette étape du cycle de vie, possèdent au moins une caractéristique commune : le concepteur humain est indispensable à leur exploitation. À l'issue d'un certain nombre de transformations conduites par le concepteur humain (elles sont symbolisées sur la figure par les transitions étiquetées d'un τ italique), le composant est uniquement constitué d'entités dont l'exploitation est réalisée de manière autonome par la machine. Il entre alors dans la deuxième phase, dite phase « inerte ».

3.1.2 États `Installable` et `Machine-Transformable`

Suivant son degré d'aboutissement, un composant débute la phase « inerte » soit à l'état `installable`, soit à l'état `machine-transformable`. Dans ces deux états, les composants ont une incarnation numérique : ils existent sous la forme d'un ou de plusieurs fichiers stockés dans une mémoire de masse. Cependant, ils ne sont pas encore disponibles pour une utilisation au sein d'un système interactif. Un composant débute la phase inerte à l'état `installable` s'il est constitué de code directement exécutable. Par exemple, les composants binaires (un programme exécutable, une DLL ou du bytecode java) ou les composants écrits dans un langage interprétable (comme Tcl ou HTML) font partie de cette catégorie. Les autres composants, constitués de modèle de plus haut niveau d'abstraction, ne sont pas exécutables sans une ou plusieurs étapes de transformation. Les composants appartenant à ce deuxième type débutent la phase inerte à l'état `machine-transformable`. À ce stade, il est possible, mais pas obligatoire, d'effectuer des transformations sur un composant à cet état (elles sont symbolisées sur la figure par les transitions étiquetées τ). Si le produit de ces transformations aboutit à un code exécutable, le

[67] Fabrique : un des rôles primordiaux d'une fabrique, d'où elle tire son nom, est la création d'instances de composant à partir de composants inertes. En ce sens, elle a une fonction similaire à la fabrique du patron de [Gamma95].

composant passe à l'état `installable`. Sinon, il reste à l'état `machine-transformable`.

3.1.3 États `installed` et `loaded`

Un composant à l'état `installable` peut subir l'opération *`install`* qui a pour effet de le faire passer à l'état `installed`. Symétriquement, un composant à l'état `machine-transformable` peut subir l'opération *`load`* qui a pour effet de le faire passer à l'état `loaded`. Ces deux opérations sont des opérations de déploiement réalisées par un administrateur humain. Elles ont pour objet de paramétrer une *fabrique* pour que celle-ci prenne en compte de nouveaux composants. Aux états `installed` et `loaded` les composants ont intégré la plate-forme de l'utilisateur et deviennent disponibles pour une utilisation au sein d'un système interactif. Il est à remarquer que les états `installed` et `loaded` du cycle de vie Ethylene sont similaires à l'état `installed` du cycle de vie des paquetages de composants d'OSGi [Cervantes04]. Les opérations inverses, *`uninstall`* et *`unload`*, également à la charge d'un administrateur humain, retirent un composant de la plate-forme de l'utilisateur. Il retrouve alors son état précédent.

3.1.4 États `instantiable` et `selectionnable`

D'un état `installed`, un composant passe à l'état `instantiable` suite à une opération *`register`*. De la même manière, d'un état `loaded`, un composant passe à l'état `selectionable` suite à la même opération. Cette opération *`register`* est à la charge de la fabrique dans laquelle le composant a été installé. Elle consiste à déclarer à un annuaire la présence et les caractéristiques de ce composant. Ainsi, qu'il soit à l'état `instantiable` ou `selectionable`, ce composant est maintenant sujet à être découvert par des tiers. L'opération *`unregister`* est l'opération inverse de la précédente. Son effet est de désinscrire de l'annuaire le composant auquel on l'applique. Disparu de l'annuaire, celui-ci n'a plus aucune chance d'être utilisé au sein d'un système interactif, et repasse à son état précédent (`installed` ou `loaded`).

3.1.5 L'état `transforming`

Sur un composant à l'état `selectionable`, un tiers peut effectuer l'opération *`transform`*. Cette opération a pour effet d'extraire du composant une copie des modèles de haut niveau d'abstraction qu'il embarque afin que ceux-ci soient traités par les outils de transformation adéquats. Cette copie forme en mémoire vive une instance de ce composant. L'opération *`transform`* ne constitue pas véritablement une transition, mais une création d'instance de composant. Ainsi, le composant reste à l'état `selectionable` et son instance nouvellement créée débute son existence à l'état `transforming` de la phase « instance » du cycle de vie. À l'état `transforming`, les transformations se succèdent jusqu'à aboutir à un code exécutable. L'instance de composant passe alors

automatiquement à l'état `starting`. Dans le même temps, le code exécutable issu du processus de transformation peut être sauvegardé (opération *`save`*) au sein d'une « fabrique ». Le produit de cette sauvegarde devient alors un composant à l'état `installed`, qui pourra être réutilisé plus tard de la même manière qu'un composant classique, en économisant le temps de calculs des transformations.

3.1.6 L'état `starting`

Sur un composant à l'état `instantiable`, un tiers peut effectuer l'opération *`instantiate`*. De la même manière que l'opération *`transform`*, l'opération *`instantiate`* réalise une instanciation en mémoire vive du composant inerte. Ainsi, cette opération n'est pas non plus une transition, mais une création d'instance de composant. Le composant reste donc à l'état `instantiable`. Comme un composant, dans cet état, est directement exécutable, son instance débute la phase « instance » du cycle de vie dans l'état `starting`. Dans cet état, l'instance de composant effectue toutes ses procédures d'initialisation. Lorsqu'elle est prête à assurer sa fonction et à prendre place au sein d'un assemblage de composants, l'instance de composant passe automatiquement à l'état `active`.

3.1.7 L'état `active`

L'état `active` représente l'état de fonctionnement normal d'une instance de composant. Sa boucle d'exécution est lancée et l'instance est en mesure d'assurer sa fonction. Généralement, une instance de composant assure une fonction au sein d'un assemblage d'instances de composant. L'état `active` admet donc l'opération *`bind`* qui établit une liaison entre cette instance et une ou plusieurs autres, ainsi que l'opération inverse *`unbind`* qui la détruit. Ces deux opérations sont réalisées par un tiers par l'intermédiaire des fabriques des instances de composants engagés dans cette liaison. Les opérations *`bind`* et *`unbind`* ne provoquent pas de changement d'état de l'instance de composant qui reste à l'état `active`.

3.1.8 États `stopping` et `destroyed`

Lorsqu'une instance de composant n'est plus nécessaire dans un assemblage, elle doit être détruite. La destruction d'une instance de composant passe par l'opération *`destroy`* qu'il est possible de lui appliquer lorsqu'elle se trouve à l'état `active`. Cette opération fait passer l'instance de composant de l'état `active` à l'état `stopping`. Dans cet état, elle se prépare à l'arrêt. Suivant l'approche ou la technologie à composants employée, elle peut avoir à rendre persistant son état interne ou envoyer des signaux particuliers à des tiers. Si des liaisons avec d'autres instances de composant sont encore présentes, la fabrique de l'instance de composant à l'état `stopping` les coupe. Une fois que l'instance est

prête à être détruite, elle passe automatiquement à l'état `destroyed`. La fabrique qui en est responsable peut alors la détruire et libérer l'espace-mémoire qu'elle occupait.

3.1.9 Résumé et conclusion

Le cycle de vie des composants du modèle Ethylene se partage donc en trois phases qui correspondent aux trois formes dans lesquelles un composant s'incarne au cours de son existence. La première phase est la phase de conception où un composant n'existe que sous la forme de spécification et de modèles manipulés par son concepteur humain. Une fois les spécifications et les modèles suffisamment enrichis pour être traités de manière autonome par une machine, le composant entre alors dans une phase « inerte » où il existe sous la forme de fichiers numériques enregistrés dans une mémoire de masse. Lors de cette phase, le composant est déployé par un administrateur humain au sein d'une ou plusieurs fabriques. Les fabriques sont des entités logicielles chargées de contrôler et de gérer le cycle de vie des composants et de leurs instances. Leur première tâche est de déclarer les composants qu'elles contrôlent auprès d'annuaires. Ces composants déclarés sont alors susceptibles d'être découverts par le reste du système et d'être utilisés au sein de systèmes interactifs. Lorsqu'un composant est sélectionné pour faire partie d'un système interactif, une opération d'instanciation ou de transformation engendre la création d'une instance de ce composant en mémoire vive. L'instance de composant est la forme *active*, c'est-à-dire *en état d'exécution* d'un composant. Son existence est régie par la phase « instance » du cycle de vie d'Ethylene.

Le cycle de vie d'Ethylene a la particularité de faire la différence et de mettre en relation les deux formes principales que prend un composant logiciel au cours de sa vie : la forme inerte qui correspond au composant stocké sur une mémoire de masse, prête à être utilisée, et la forme instanciée qui correspond au composant en mémoire vive, en cours d'exécution. Cette différenciation met en évidence certaines propriétés. Notamment, bien qu'une instance de composant soit issue d'un composant, une fois créée, l'instance devient une entité logicielle indépendante de celui-ci. Ainsi, une opération *`unregister`*, *`uninstall`* ou *`unload`* sur un composant n'a aucune incidence sur ses éventuelles instances en cours d'exécution. Par contre, aucune nouvelle instance de ce composant ne pourra être créée avant que celui-ci ne retrouve un état `instantiable` ou `selectionable`. Par ailleurs, le cycle de vie introduit les différentes opérations qui régissent l'existence des composants et de leurs instances. Parmi ces opérations, certaines sont réalisées par un administrateur humain et les autres sont réalisées par une *fabrique*. Ces dernières constituent les opérations des API relatives au « producteur de l'adaptation » et au « gestionnaire de composants » de ma décomposition

fonctionnelle. Pour être plus précis quant à la description de ces opérations, il est nécessaire d'établir les concepts et les relations qui définissent les notions de composant et de connecteur. Ainsi, la section suivante détaille le modèle à composants dynamiques Ethylene.

3.2 Le modèle à composants dynamiques Ethylene

Le modèle Ethylene est très librement inspiré du métamodèle à composants du projet RNTL ACCORD proposé par [Legond-Aubry05] et de la modélisation proposée par UML 2 [Omg07]. Du modèle de [Legond-Aubry05], je conserve la décomposition des composants, j'adapte la notion de connecteur d'assemblage d'UML 2 à ma problématique et je garde l'idée d'assemblage des composants dirigé par des contrats que prône [Legond-Aubry05]. Par ailleurs, j'ajoute les notions d'instance de composant et de fabrique.

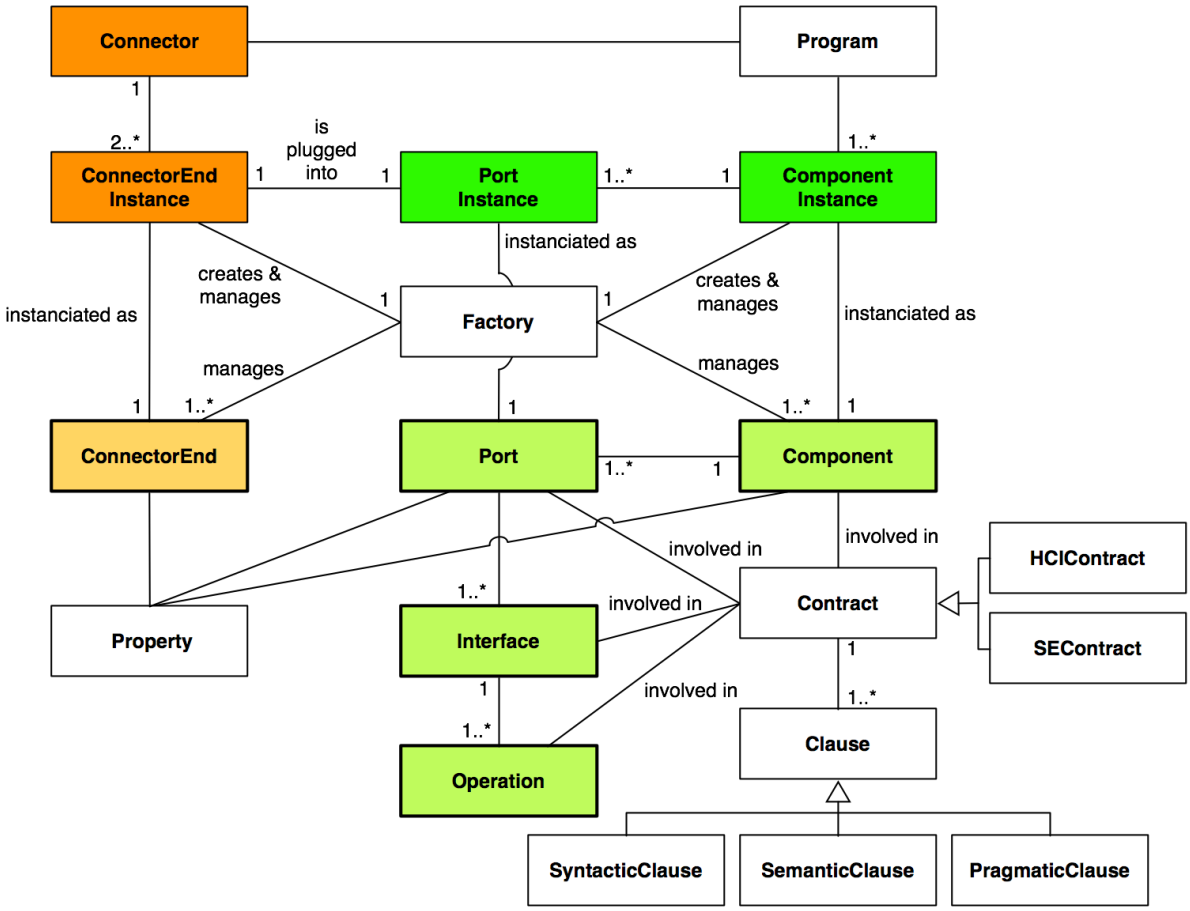


Figure V-8 Diagramme de classes du modèle Ethylene. Les boîtes colorées à encadrement épais modélisent les composants et connecteurs sous leur forme inerte. Les boîtes colorées à encadrement fin modélisent les instances de composants et de connecteurs. Les boîtes blanches modélisent les concepts périphériques. Les relations sans étiquette symbolisent la relation *est composé de*. Lorsque la cardinalité d'une relation n'est pas indiquée, cette cardinalité vaut *.

Les concepts du modèle Ethylene et leurs relations sont illustrés par le diagramme de classe de la Figure V-8. Ce diagramme comprend trois parties. Les boîtes colorées à encadrement épais modélisent les composants et connecteurs sous leur forme inerte. Les boîtes colorées à encadrement fin modélisent les composants et connecteurs sous leur forme instanciée. Les boîtes blanches modélisent des concepts périphériques aux composants et connecteurs, mais apportent, d'une part, les propriétés de reconfiguration dynamique des assemblages et de disponibilité dynamique des composants, et d'autre part, la capacité de sous-tendre l'interopérabilité entre les différentes technologies d'implémentation des systèmes interactifs plastiques.

3.2.1 Vue générale d'Ethylene

Sous sa forme inerte, un *composant* encapsule l'implémentation d'une unité fonctionnelle. Cette encapsulation se décompose en *ports* qui se décomposent en *interfaces*. À leur tour, les interfaces se décomposent en *opérations*. Par ailleurs, une *extrémité de connecteur* encapsule un moyen et un protocole de communication. Les *composants*, *ports* et *extrémités de connecteurs* sont associés à des *propriétés* qui caractérisent leurs capacités fonctionnelles et/ou extra-fonctionnelles. *Composants* et *extrémités de connecteur* sont associés à une ou plusieurs *fabriques*. Ces *fabriques* sont chargées d'un triple rôle : assurer l'enregistrement des *composants* auprès d'annuaires, instancier les *composants* et *extrémités de connecteur* et contrôler leur cycle de vie. Les *contrats* encadrent la façon dont les instances des composants sont assemblées. Ils impliquent les *propriétés* des *composants* et des *ports*, ainsi que les spécifications des *interfaces* et *opérations*. Je détaille à présent chacun de ces concepts et les relations qu'ils partagent. Les substantifs en italique font référence aux entités du modèle Ethylene.

3.2.2 Opérations

Operation
name : string parameters : list of (identifier, type) result : type

La spécification d'une *opération* consiste en une signature de fonction. Elle se définit par un nom, une liste de paramètres d'appel et, si cette fonction est destinée à retourner un résultat, le type du résultat. En ce sens, une *opération* est similaire aux *méthodes* des langages de programmation à objets. Cependant, les composants ne peuvent partager que des liaisons dites faibles, ce qui pose des contraintes sur les paramètres d'appel et de retour. Ainsi, ceux-ci ne peuvent pas transmettre de données par référence. Seuls sont possibles les passages de données par valeur. Suivant le contexte dans lequel est spécifiée l'opération, celle-ci indique soit une fonction que le composant est capable de réaliser

et qui est mise à disposition d'un composant tiers, soit une fonction dont le composant a besoin et dont il souhaite déléguer la réalisation à un composant tiers.

3.2.3 Interfaces

Interface
name : string operations : list of **Operation**

Une *interface* est un ensemble d'opérations identifié par un nom. Dans l'interaction entre deux composants, *l'interface*, qui définit des messages et leurs paramètres, fixe la sémantique des échanges. Ainsi, ce concept *d'interface* est tout à fait similaire au concept d'interface du langage Java. Elle regroupe des opérations organisées autour d'une préoccupation fonctionnelle unique et précise. Suivant le contexte de sa spécification, soit toutes les opérations de *l'interface* représentent un service, c'est-à-dire des opérations mises à disposition par un composant à destination d'un tiers, soit toutes les opérations de *l'interface* expriment un besoin dont le composant est le client. Cependant, dans le domaine de l'interaction homme-machine, les interfaces utilisateur sont bien souvent à la fois services et clientes vis-à-vis du noyau fonctionnel. Elles sont clientes lorsqu'elles transmettent au noyau fonctionnel les actions de l'utilisateur et elles sont services lorsqu'elles permettent au noyau fonctionnel de rendre perceptible une information à l'utilisateur. Ainsi, si une *interface* est définie pour une préoccupation fonctionnelle unique, une préoccupation fonctionnelle est généralement définie par la réunion de plusieurs *interfaces*.

3.2.4 Ports

Port
name : string mode : SYNC or ASYNC interfaces : list of (**Interface**, IN or OUT) properties : list of **Property**

Les interfaces relatives à une même préoccupation fonctionnelle sont réunies au sein d'un *port*. Ce concept incarne un point d'interaction entre un composant et son environnement. Au sein d'un *port*, chaque interface est associée à un attribut *direction* qui admet la valeur IN ou OUT. Une interface déclarée IN spécifie les messages en provenance de l'extérieur qui peuvent être reçus. À l'inverse, une interface déclarée OUT spécifie les messages qui peuvent être émis vers l'extérieur. Le rôle du *port* est d'introduire un niveau de généricité entre composants et connecteurs afin de garantir un certain niveau d'indépendance entre les deux. Ainsi, si deux composants qui interagissent doivent partager une sémantique pour se comprendre, le niveau de généricité introduit par le *port* assure que le connecteur qui achemine les messages

n'a pas besoin de connaître cette sémantique. Connecteurs et composants peuvent donc être développés indépendamment.

Cependant, l'indépendance entre le code interne à un composant et la façon dont est conduite la communication avec l'extérieur n'est pas totale : par exemple, les algorithmes utilisés par un composant ne seront pas les mêmes suivant que l'interaction avec l'extérieur est synchrone ou asynchrone. De la même manière, le composant peut requérir du connecteur une certaine qualité de service. Par exemple, le monde médical requiert d'un système informatique que toutes les transmissions d'information au sujet d'un patient soient chiffrées. Ces contraintes se spécifient au niveau du *port*. Ainsi, un *port* possède un attribut *mode* qui accepte les valeurs SYNC ou ASYNC suivant le mode de communication requis. Généralement, si la spécification de l'une des *opérations* d'une des *interfaces* du port retourne un résultat, alors l'interaction est synchrone (*mode* vaut SYNC). Si aucune des opérations ne retourne de résultat, la communication est asynchrone (*mode* vaut ASYNC). Les autres contraintes, par exemple celles qui sont liées à la qualité de service, sont modélisées sous la forme d'une liste de *propriétés* associée au *port*.

3.2.5 Composants et instances de composant

Component
name : string ports : list of (**Port**, USED or PROVIDED) properties : list of **Property** content : string

Comme dans la plupart des modèles à composants, le composant représente l'unité de calcul et de déploiement. Dans mon approche, il est associé à une *fabrique* par un administrateur du système. Un composant est identifié par l'attribut *name* dont la valeur doit être unique au sein de l'ensemble des noms des composants associés à une même fabrique. Par ailleurs, un attribut *content* est destiné à accueillir les modèles qui forment les composants à l'état `human-transformable`. Pour les composants à l'état `installable`, cet attribut peut rester vide.

Si un composant peut être totalement autonome, sa vocation est généralement d'offrir des services à des tiers et/ou de déléguer la réalisation de certaines fonctions à d'autres composants. La spécification de *ports* permet à un *composant* de définir des points et sémantiques d'interaction avec ses pairs. Chaque *port* est annoté comme étant PROVIDED ou USED suivant qu'il représente, respectivement, une offre ou une demande de service. Si un *port* est noté PROVIDED, ses *interfaces* IN correspondent à la description des messages qu'il est capable de recevoir et ses *interfaces* OUT correspondent à la description des messages qu'il

est capable d'envoyer. Un port noté USED se comporte à l'envers : les *interfaces* IN correspondent à la description de messages sortants et les *interfaces* OUT correspondent à la description des messages entrants. Ainsi, un port noté PROVIDED se relie à un port identique noté USED.

Si les *ports* décrivent la capacité d'un composant à interagir avec ses pairs, ils n'indiquent rien ou très peu de la tâche utilisateur que celui-ci entend sous-tendre, ni de la qualité de service qu'il est en mesure d'assurer, ni des ressources d'interaction ou de calculs dont il a besoin pour cela. L'ensemble de ce type d'informations est décrit par une liste de *propriétés* (voir au § 3.2.8) associée au *composant*. Ainsi, les *ports* et les *propriétés* sont les éléments qui permettent de déterminer la capacité d'un composant à intégrer un assemblage et son adéquation à la situation courante de l'utilisateur. Pour être exploité au sein d'un assemblage, un *composant*, selon qu'il est dans l'état `instantiable` ou `selectionable`, doit être instancié ou transformé, afin de créer une *instance de composant* à l'état `active`.

L'*instance de composant* est la version chargée en mémoire vive et en état d'exécution d'un *composant*. Plusieurs transformations ou instanciations successives d'un même *composant* donnent lieu à autant d'instances indépendantes les unes des autres. Une fois créée, chacune de ces instances est également indépendante du *composant* dont elle est issue. Si le composant évolue, ses instances ne sont donc pas mises à jour. Pour chaque *port* d'un *composant*, une *instance de composant* expose une *instance de port* qui permet d'accueillir une liaison avec d'autres *instances de composants*. Pour garantir une souplesse maximale, cette liaison ne peut s'inscrire que dans le cadre d'un couplage faible : aucune liaison ne doit avoir un caractère indispensable à l'exécution sans faute d'une instance de composant. Autrement dit, si la réalisation d'une ou de plusieurs fonctions d'une instance de composant dépendent d'entités tierces et que les liaisons avec ces entités tierces ne sont pas établies, alors les fonctions qui en dépendent ne sont pas tenues d'être assurées, en revanche, l'absence de liaisons ne doit pas provoquer la panne de l'instance de composant.

3.2.6 Extrémités de connecteur et connecteurs

ConnectorEnd
name : string mode : SYNC or ASYNC properties : list of **Property**

Dans mon approche, les systèmes interactifs sont construits par assemblage d'*instances de composant* reliées par des *connecteurs* au moyen d'*instances de port*. Ainsi, le *connecteur* est un concept qui n'existe que par l'interconnexion d'un minimum de deux *instances d'extrémités de connecteur*. Une *instance d'extrémité de*

connecteur est créée à partir d'une *extrémité de connecteur*. L'*extrémité de connecteur* est une abstraction qui offre aux *ports* un accès transparent à diverses technologies de communication. Ainsi, une *extrémité de connecteur* implémente ce que j'appelle un moyen de communication, c'est-à-dire, en termes de niveaux du modèle OSI [Zimmermann80], la mise en œuvre de mécanismes du niveau de la couche transport et d'un protocole de communication du niveau de la couche application. De la même manière que les *composants*, l'existence des *extrémités de connecteur* est régie par un cycle de vie (voir Figure V-9).

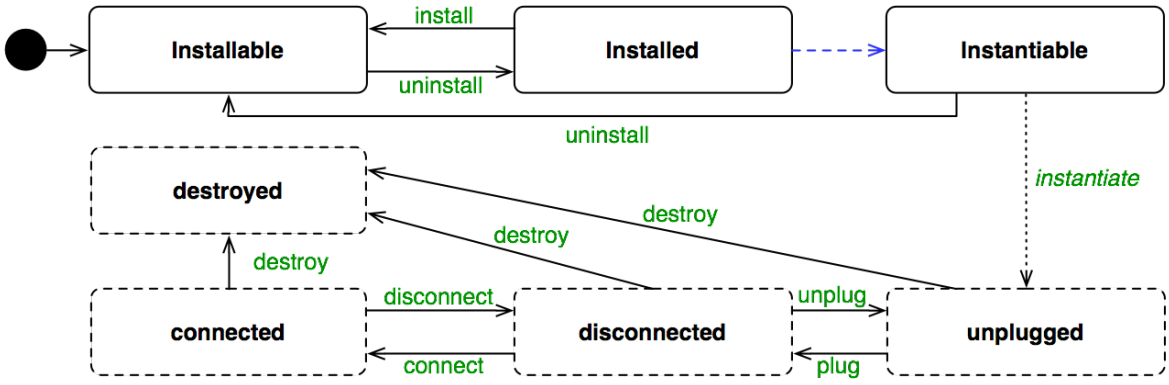


Figure V-9 Cycle de vie des extrémités de connecteurs Ethylene. Le disque noir pointe l'état initial. Les états dont l'encadrement est un trait continu s'inscrivent dans la phase « inerte ». Les états dont l'encadrement est un trait discontinu s'inscrivent dans la phase « instance ». Les flèches dont le trait est discontinu dénote une transition automatique. La flèche dont le trait est en pointillé dénote la création d'une nouvelle entité.

Une fois codée par le développeur, l'*extrémité de connecteur* existe sous la forme de code exécutable stocké sur une mémoire de masse : elle se trouve alors à l'état initial de son cycle de vie incarné par l'état `installable`. Comme les *composants*, les *extrémités de connecteurs* sont associés à des *fabriques* qui sont chargées de les mettre en œuvre. Cette association, représentée par l'opération *`install`*, est réalisée par l'administrateur du système. Elle a pour effet de faire passer l'extrémité de connecteur à l'état `installed`, puis par une transition automatique à l'état `instantiable`. L'opération inverse, *`uninstall`*, a pour but de retirer à une fabrique la possibilité d'utiliser cette extrémité de connecteur. Cette opération, accessible depuis les états `installed` et `instantiable` est également à la charge de l'administrateur du système. À partir d'une *extrémité de connecteur* à l'état `instantiable`, une *fabrique* peut l'instancier (opération *`instantiate`*) pour donner lieu à une *instance d'extrémité de connecteur*.

Cette instance débute son existence à l'état `unplugged` : elle n'est encore reliée à rien. L'application par la *fabrique* de l'opération

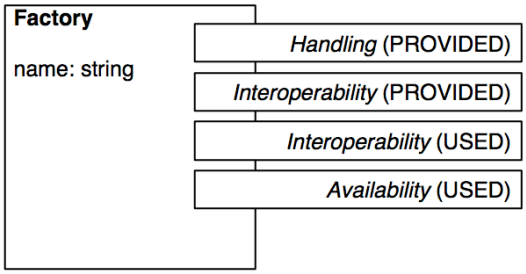
`plug` attache l'*instance d'extrémité de connecteur* à une *instance de port* : l'*instance d'extrémité de connecteur* passe alors à l'état `disconnected`. À cet état, deux opérations lui sont applicables `unplug` et `connect` : l'opération `unplug` la détache de l'*instance de port*, alors que l'opération `connect` établit la connexion avec une ou plusieurs autres *instances d'extrémité de connecteur*. À la suite de l'opération `connect`, l'*instance d'extrémité de connecteur* passe à l'état `connected` : les *instances d'extrémité de connecteur* interconnectées forment alors un *connecteur*. L'opération `disconnect` permet de revenir à l'état précédent. Dans le même temps, la connexion est rompue et le *connecteur* détruit. À chacun des trois états `unplugged`, `disconnected` et `connected`, la fabrique peut appliquer l'opération `destroy` qui provoque la destruction de l'*instance d'extrémité de connecteur* et le cas échéant, la rupture de la connexion et le détachement de l'*instance de port*. Si toutes ces opérations sont à la charge de la *fabrique*, l'opération `disconnect` et `unplugged` sont éventuellement accessibles à l'*instance de composant* auquel l'*instance d'extrémité de connecteur* est attachée.

Si ce cycle de vie est commun à toutes les *extrémités de connecteur*, quel que soit le moyen de communication qu'elles mettent en œuvre, ces aspects technologiques doivent être décrits car ils constituent un critère de choix lors de la sélection du type d'*extrémité de connecteur* à instancier. Ainsi, l'attribut *name* d'une *extrémité de connecteur* caractérise le type de technique et de protocole de communication employés par celle-ci. Dès lors, une instance de *connecteur* résulte de la connexion d'un ensemble d'*instances d'extrémité de connecteur* qui partagent le même nom, c'est-à-dire la même technique et le même protocole de communication. L'attribut *mode* caractérise la capacité du moyen de communication de permettre des appels synchrones ou asynchrones selon que sa valeur soit SYNC ou ASYNC. Ainsi, une *instance d'extrémité de connecteur* destinée à mettre en œuvre une communication asynchrone ne peut être reliée qu'à une *instance de port* qui ne comporte que des appels asynchrones. En revanche, une *instance d'extrémité de connecteur* destinée à mettre en œuvre une communication synchrone peut être reliée à une *instance de port* qui comporte des appels synchrones ou asynchrones.

Le découpage en *composant*, *port* et *extrémité de connecteur* a l'avantage de dissocier totalement ce qui a trait aux composants et ce qui a trait à la liaison entre ces composants, laissant la possibilité de sélectionner une *extrémité de connecteur* tardivement, c'est-à-dire au moment où une liaison doit être établie, en fonction de la situation courante de l'espace interactif. Afin que cette sélection puisse être réalisée de la façon la plus pertinente, il convient de décrire, en plus des propriétés fonctionnelles de l'*extrémité de connecteur*, ses propriétés extra-

fonctionnelles. Cette description est réalisée par une liste de *propriétés*. Par exemple, ces *propriétés* peuvent indiquer que la communication mise en œuvre dans le *connecteur* est chiffrée, fiable ou que la bande passante est garantie. Sélectionner la bonne extrémité de connecteur parmi celles qui sont disponibles revient à faire correspondre les propriétés de l'*extrémité de connecteur* avec les requis exprimés par les *propriétés* des *instances de port* qu'il faut relier. La sélection des *extrémités de connecteur* est une tâche du ressort de la *fabrique*.

3.2.7 Fabriques



La fonction d'une *fabrique* est de gérer le cycle de vie des *composants* et des *extrémités de connecteur* qui lui ont été associés par un administrateur du système. Ainsi, c'est par son intermédiaire que toutes les opérations concernant les cycles de vie des *composants* et des *extrémités de connecteur* sont appliquées à ces derniers. La *fabrique* est une sorte de composant : c'est une entité logicielle qui interagit avec son environnement par l'intermédiaire de services offerts ou utilisés, décrits par des interfaces programmatiques regroupées par thème au sein de ports accessibles individuellement via un moyen de communication. Cependant, quelques différences la démarquent de la notion de composant. Contrairement à un composant Ethylene, une fabrique n'est constituée que de code exécutable. Son cycle de vie n'est pas géré par une autre fabrique mais par des processus de la couche « système d'exploitation » de ma décomposition fonctionnelle. Enfin, son cycle de vie se limite à deux états : un état « inerte », lorsqu'elle n'existe que sous la forme d'éléments stockés sur une mémoire de masse et un état « actif » lorsqu'elle est en état d'exécution.

La fonction de gestion du cycle de vie des composants et des extrémités de connecteur fait de la *fabrique* la pierre angulaire de l'interopérabilité entre les différentes technologies à composants, le « producteur de l'adaptation » et le « gestionnaire de composant ». Ainsi, c'est l'incarnation technologique de cette entité qui implémente les API *handling*, *availability* et *interoperability*, que je détaille dans la section suivante. Le concept de fabrique se définit donc par un attribut *name* qui l'identifie de manière unique sur la plate-forme et de quatre ports. Le premier port définit une offre de service constituée de l'API

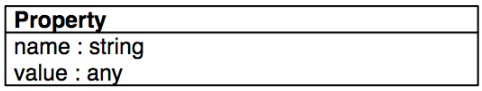
handling. Par son intermédiaire, des entités tierces ont la possibilité de manipuler les *composants* et les *instances de composant* gérés par la *fabrique*. Le deuxième port héberge l'API *availability*, relative à la disponibilité dynamique. Ce port constitue une demande de service à destination d'annuaires. Il permet à la fabrique d'inscrire et de désinscrire des composants (opération register/unregister) dans des annuaires utilisés pour la découverte dynamique de composants. Enfin les deux derniers ports hébergent l'API *interoperability*, l'un la présentant comme une offre de service, et l'autre comme une demande de service.

L'objet de l'API *interoperability* est de permettre aux *fabriques* de négocier entre elles le type d'*extrémité de connecteur* à instancier et à relier, lors de l'établissement d'une liaison entre des instances de composants. Une fabrique est alors tour à tour le demandeur de service, lorsqu'elle prend l'initiative d'une phase de négociation, ou le fournisseur de service, lorsqu'une autre fabrique est à l'initiative de cette phase. L'interopérabilité entre deux technologies à composants est donc soumise à deux conditions : dans chacune de ces technologies, de telles fabriques doivent être implémentées, et à chacune de ces fabriques doivent être associées un ensemble d'extrémités de connecteur. L'interopérabilité est possible si l'intersection des ensembles d'*extrémités de connecteur* associés à chaque fabrique est non nul. Ainsi, suivant cette approche, augmenter l'interopérabilité entre deux technologies à composants se résume à étoffer l'ensemble d'extrémités de connecteur associé à chaque fabrique.

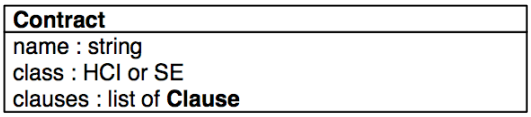
3.2.8 Propriétés et contrats

La description des concepts du modèle à composant Ethylene s'achève avec la description de deux concepts périphériques : les *propriétés* et les *contrats*. Si ces deux concepts sont périphériques, ils n'en sont pas moins indispensables. En effet, pour qu'un système[68] puisse constituer des assemblages de composants, il doit être capable de faire par lui-même, parmi les composants disponibles, une sélection de ceux qui sont capables, d'une part, de remplir la fonction attendue avec la qualité de service requise, et d'autre part, de s'insérer convenablement dans l'assemblage d'accueil. Dans ce cadre, les *propriétés* permettent de décrire les propriétés fonctionnelles et extra-fonctionnelles des *composants*, des *ports* et des *extrémités de connecteur*. Les *contrats* permettent au système d'exprimer des attentes quant aux entités recherchées et d'évaluer leur capacité à s'intégrer au sein d'un assemblage existant.

[68] Par opposition à un concepteur ou un administrateur



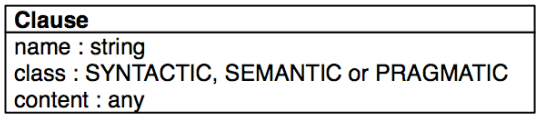
À la manière du langage ACME [Garlan97], Ethylene ne définit pas un ensemble de propriétés, qui serait immanquablement incomplet, mais propose un mécanisme pour les énumérer. Ainsi, dans Ethylene, une propriété se définit par un attribut *name* qui l'identifie de manière unique, et par un attribut *value* qui contient sa valeur. Comme dans ACME, Ethylene ne fixe pas de format pour représenter la valeur d'une propriété et celle-ci, au niveau du modèle, demeure non-interprétable. Une propriété et sa valeur deviennent utiles et interprétables dès lors qu'un outil extérieur est en mesure de leur donner un sens, par exemple en leur faisant correspondre un type et une sémantique, afin de les exploiter. Dans mon approche, l'objectif des propriétés est double : elles doivent fournir, d'une part des critères de sélection permettant d'élire les composants les plus adaptés au contexte de l'interaction courant, et d'autre part un moyen de déterminer leurs capacités à s'intégrer au système interactif en cours d'exécution. Ainsi, les *propriétés* forment les caractéristiques sur lesquelles s'appuient les *contrats* pour évaluer la capacité d'un composant à rejoindre un assemblage.



En Génie Logiciel, la notion de contrat apparaît au début des années 90 avec les travaux de Bertrand Meyer autour du langage Eiffel [Meyer91] [Meyer92]. Cette notion vise à expliciter les conditions encadrant l'interaction entre deux entités logicielles, à l'aide de pré-conditions, de post-conditions et d'invariants, afin de limiter les erreurs de programmation liées à la réutilisation de code existant. Adaptés aux approches à composants, les contrats offrent un moyen d'anticiper de manière fiable le comportement d'un composant au sein d'un futur assemblage [Beugnard99]. Dans ce cadre, [Beugnard99] distingue quatre niveaux de contrat. Le niveau *syntaxique* spécifie les interfaces programmatiques. Le niveau *comportemental* précise, dans un contexte transactionnel, les conditions d'utilisation de chaque opération et leurs effets. Le niveau *synchronisation* caractérise le comportement du service rendu par un composant dans un contexte de concurrence. Enfin, le niveau *qualité de service* permet de quantifier certaines propriétés liées au comportement attendu d'un composant, par exemple le temps de réponse maximum accordé ou le degré de précision attendu du résultat. De manière similaire, les travaux de [Legond-Aubry05] partitionne les contrats en trois classes : syntaxique, sémantique et pragmatique. Le niveau syntaxique de

[Beugnard99] se projette dans le niveau syntaxique de [Legond-Aubry05], le niveau comportemental dans le niveau sémantique, et les niveaux synchronisation et qualité de service dans le niveau pragmatique.

Cependant, les approches « par contrat » existantes abordent la question de l'assemblage de composants sous l'angle exclusif de l'interaction logicielle. Or, lorsqu'un assemblage de composants est destiné à produire une IHM, résoudre la question de l'assemblage sur le plan de l'interaction logicielle est nécessaire mais pas suffisant : si l'assemblage doit fonctionner sur un plan logiciel, l'interface utilisateur qui en résulte doit être utile et utilisable. En outre, dans la problématique de la plasticité des systèmes interactifs, en plus de déterminer la capacité d'un composant à s'intégrer à un assemblage, il s'agit d'élire, parmi l'ensemble des composants disponibles, le meilleur candidat pour assurer la fonction recherchée. Ainsi, dans mon approche, je distingue deux classes de contrats : les contrats propres à l'interaction homme-machine (*HCIContract*) et les contrats relatifs aux aspects de Génie Logiciel (*SEContract*). Les premiers s'accordent à définir les requis à respecter en termes d'utilité et d'utilisabilité pour l'interface utilisateur à produire. Les deuxièmes définissent les conditions d'intégration sous l'angle de l'interaction logicielle. Les contrats des deux classes se définissent par une liste de *clauses*.



Une *clause* représente un élément de contrat. À la manière de [Legond-Aubry05], les clauses se répartissent en trois classes : syntaxique, sémantique et pragmatique. Pour les contrats de type Génie Logiciel (*SEContract*), ces trois classes reprennent la répartition de [Legond-Aubry05]. Pour les contrats de type IHM (*HCIContract*), ces trois classes s'interprètent différemment. Elles se distinguent par leur objet et leur niveau de négociabilité. Ainsi, la classe sémantique caractérise, indépendamment du contexte de l'interaction, la fonction, en termes de tâches utilisateur, que les éléments d'IHM issus des composants recherchés devront fournir. Les clauses de cette classe sont partiellement négociables : si aucun composant disponible ne sous-tend la tâche utilisateur au niveau de spécialisation attendue (par exemple « consulter une température »), un composant capable de sous-tendre cette tâche utilisateur à un niveau plus général pourrait constituer une solution de repli (par exemple « consulter une valeur numérique »). Par ailleurs, les classes syntaxique et pragmatique modélisent les contraintes apportées par le contexte de l'interaction. Les clauses de ces deux classes ont pour objet

l'évaluation des éléments d'IHM issus des composants candidats au regard de critères d'utilisabilité. Les clauses de la classe syntaxique sont non-négociables : une IHM qui ne les respecterait pas ne serait pas utilisable par l'utilisateur. En revanche, les clauses de la classe pragmatique, également relatives à l'utilisabilité, sont totalement négociables : elles n'ont pas de caractère indispensable, mais les respecter maximise le confort et l'efficacité de l'utilisateur dans la réalisation de sa tâche.

Concernant les contrats et leurs clauses, Ethylene se borne à établir une classification. À l'instar des *propriétés*, aucun format pour les exprimer n'est fixé au niveau du modèle. D'une part, face à la diversité des contraintes exprimables dans le cadre de ces contrats, un format ou langage unique ne suffirait probablement pas à les décrire toutes, et d'autre part, leur expression dans un modèle du niveau d'Ethylene ne semble pas utile. Ainsi, le format de spécification des contrats et des clauses est laissé à la discrétion des outils chargés de les écrire et de les évaluer.

3.2.9 Résumé et Conclusion

En résumé, le modèle à composants dynamique Ethylene se caractérise par une propriété, la disponibilité dynamique des composants, et trois objectifs :

(1) permettre la constitution d'assemblage de composants hétérogènes,

(2) fournir au « producteur de l'adaptation » ainsi qu'au « gestionnaire de composants » une API pour masquer l'hétérogénéité des technologies utilisées pour l'implémentation des composants, et

(3) offrir un moyen de description des composants dans un format indépendant des technologies et adapté au requis de l'ingénierie de l'interaction homme-machine.

Ces trois objectifs sont atteints, d'une part, en séparant clairement le concept de composant de celui de connecteur, et d'autre part, en confiant la gestion du cycle de vie des composants et des connecteurs à des fabriques. Séparer clairement au niveau du modèle le concept de composant de celui de connecteur permet, au niveau technologique, une certaine indépendance entre les technologies à composant et celles des connecteurs, rendant ces dernières substituables les unes aux autres. Par ailleurs, la décomposition d'un connecteur en extrémités de connecteur, chacune des extrémités pouvant s'implémenter dans des technologies différentes, confère au connecteur un statut de pont technologique. Ainsi, deux technologies à composants peuvent interopérer si elles partagent au moins une technologie de connecteur.

Le concept de fabrique, qui concentre la gestion du cycle de vie des composants et connecteurs, permet de spécifier la fonction et le fonctionnement de l'entité logicielle qui, au niveau technologique, est en charge :

- de l'application des mécanismes liés à la disponibilité dynamique des composants,
- de la négociation tardive du connecteur à mettre en œuvre pour relier deux ports de composants, et
- d'offrir aux mécanismes de la plasticité la possibilité d'intervenir dans les assemblages de composants.

Enfin, Ethylene remplit son objectif de description à haut niveau d'abstraction des composants et connecteurs, en explicitant, d'une part la décomposition structurelle des composants et des connecteurs, et d'autre part en associant aux extrémités de connecteur, aux ports et aux composants, des propriétés chargées de décrire aussi bien leurs aspects fonctionnels qu'extra-fonctionnels, sans imposer de langage ou de format. Grâce aux informations de description véhiculées par ces propriétés, l'assemblage des composants peut être dirigé par des contrats qui spécifient les contraintes posées par des requis relatifs, d'une part à l'interaction homme-machine, et d'autre part à l'interaction logicielle. Chaque contrat se décompose en clauses qui, suivant leur objet et leur niveau de négociabilité, se répartissent parmi les classes syntaxique, sémantique et pragmatique. À l'instar des propriétés, Ethylene ne définit pas de langage ou de formalisme pour exprimer les contrats, et se borne à proposer une classification qui explicite la couverture d'expression des différentes classes de contrats et de leurs clauses.

Les concepts et relations du modèle étant posés, il reste à définir dans le détail les API offertes aux mécanismes de la plasticité par le biais des fabriques, ainsi que l'API que les fabriques entre elles doivent partager pour interopérer. Cette description détaillée est l'objet de la section suivante.

3.3 Spécification à haut niveau d'abstraction des API *Handling*, *Availability* et *Interoperability*.

À partir des concepts du modèle Ethylene, ainsi que du cycle de vie des composants et de celui des connecteurs, je spécifie trois API pertinentes pour la plasticité. Bien qu'elles s'adressent toutes les trois à des acteurs différents, elles partagent néanmoins quelques points. Toutes les trois s'inscrivent dans une interaction logicielle à laquelle participe un nombre indéterminé d'acteurs : *Handling* et *Availability* régissent l'interaction, respectivement, du « producteur de l'adaptation » et du « gestionnaire de composant » avec l'ensemble des fabriques en cours d'exécution sur la plate-

forme, et *Interoperability* règle l'interaction logicielle entre l'ensemble de ces fabriques entre elles. Dans ce type d'interactions logicielles, un message envoyé par l'un des acteurs est reçu par l'ensemble des autres. Ainsi, pour construire ces trois API, je choisis un mode de communication de type asynchrone : aucune opération n'est bloquante et aucune ne retourne de résultat directement. Afin de ne pas les surcharger, et puisque cette question dépasse le cadre de cette thèse, ces trois API n'intègrent pas, ou très peu, de mécanisme de tolérance aux fautes. Enfin, elles sont spécifiées à haut niveau d'abstraction : la sémantique des paramètres d'appel est explicite, mais ceux-ci ne sont pas typés.

Pour lever toute ambiguïté, les spécifications sont écrites dans un pseudo-code identifiable par une police à chasse fixe dont la syntaxe est la suivante :

```
PORT nom_port

     IN : nom_interface

          NomOperation(nomParam1, nomParam2, …);

     OUT : nom_interface

          NomOperation();

Factory : USING | PROVIDING
```

La première ligne, qui débute par le mot-clé PORT, définit le nom de l'API. Sur la deuxième ligne, le mot-clé IN, précédé d'une tabulation, introduit le nom de l'interface qui définit les messages entrants. Les lignes suivantes, chacune précédée de deux tabulations, listent les différentes opérations de l'interface. Leurs paramètres d'appel sont présentés entre parenthèses et séparés par des virgules. La définition d'une opération s'achève par un point-virgule. À la suite de la définition de l'interface IN, vient, sur le même principe, la description de l'interface OUT qui définit les messages sortants. La direction des messages (entrante ou sortante) est toujours donnée par rapport à l'offre de service. Par exemple, un service qui offre une fonction la spécifie dans une interface IN. En revanche, si ce service est capable d'émettre une notification vers son client, il la spécifie au sein de l'interface OUT. À la suite de la spécification du port, une ligne indique de quelle manière la fabrique utilise le port. Le mot-clé USING indique que la fabrique utilise un service fourni par un tiers : l'interface IN représente alors des fonctions implémentées par un tiers et rendues accessibles à la fabrique. Inversement, l'interface OUT décrit les messages que la fabrique doit être capable de recevoir et de traiter. Le mot-clé PROVIDING indique le contraire. Le port est alors un service fourni par la fabrique à destination d'un tiers. L'interface IN représente les opérations

implémentées par la fabrique et l'interface OUT les messages émis par la fabrique vers ses correspondants.

3.3.1 Availability

L'API Availability caractérise l'interaction logicielle entre les fabriques et entre un ou plusieurs services d'annuaire de composants. Elle comprend de deux interfaces. L'une permet à une fabrique d'inscrire ou désinscrire ses composants auprès d'un annuaire, l'autre permet à un annuaire d'annoncer son arrivée auprès des fabriques existantes. Le service décrit par cette API est offert par les annuaires et est utilisé par les fabriques.

```
PORT Availability

     IN : DirectoryService

          Register(   factoryId,
                      componentName,
                      componentDescription,
                      watchDog);

          Unregister(factoryId, componentName);

     OUT : DirectoryNotification
          DirectoryHasArrived();

Factory : USING
```

L'opération Register()

L'opération Register() permet l'inscription d'un composant et accepte quatre paramètres : *factoryId, componentName*, *componentDescription* et *watchDog*. Le premier représente l'identifiant de la fabrique. Il doit être construit de telle manière à être unique parmi l'ensemble des fabriques en cours d'exécution sur la plate-forme. Le deuxième paramètre correspond au nom du composant. Il doit être unique au sein d'une fabrique. Ainsi, le couple (*factoryId, componentName*) identifie de manière unique un composant donné sur la plate-forme. Le troisième paramètre contient la description Ethylene complète du composant. Enfin, le dernier paramètre représente un délai à l'issue duquel la fabrique s'engage à réenregistrer le composant. À son démarrage, une fabrique est tenue d'envoyer un message Register pour chacun des composants dont elle gère le cycle de vie. À l'issue de l'expiration d'un délai *watchDog*, sans le réenregistrement, par la fabrique, du composant concerné, l'annuaire est tenu de le retirer de son index : la plate-forme élémentaire qui héberge la fabrique a probablement été retiré de la plate-forme.

L'opération Unregister()

L'opération Unregister() permet de désinscrire un composant et accepte deux paramètres : *factoryId* et *componentName*. Ils constituent les deux informations nécessaires et suffisantes pour

identifier sans risque d'ambiguïté le composant à retirer des annuaires. Lorsqu'elle est en cours d'arrêt, une fabrique est tenue de désinscrire l'ensemble de ses composants des annuaires.

L'opération DirectoryHasArrived()

L'opération DirectoryHasArrived() permet aux annuaires d'annoncer leur arrivée aux fabriques existantes. En retour, celles-ci sont tenues de réitérer l'émission des messages Register(). Ainsi, peu importe l'ordre d'arrivée des fabriques et des annuaires. Ces derniers sont toujours assurés de connaître, dans les meilleurs délais, l'ensemble des composants disponibles sur la plate-forme.

3.3.2 Handling

L'API Handling décrit le service offert par une fabrique aux mécanismes du « producteur de l'adaptation ». Elle se compose d'une interface qui regroupe quatre des opérations élémentaires exposées par le cycle de vie des composants du modèle Ethylene, et d'une interface destinée aux notifications. Si cette API offre une opération Instantiate(), elle n'offre ni l'opération symétrique Transform(), ni d'opération τ() ou d'opération Save() qui sont, pourtant, du même niveau qu'Instantiate() dans le cycle de vie des composants. En effet, si la fabrique gère l'intégralité du cycle de vie des composants directement exécutables, ce sont des outils extérieurs qui sont en charge de la transformation des composants « à modèles ». Par ailleurs, l'étude de la génération de composant exécutable à partir de modèle dépasse le cadre de cette thèse, pour rejoindre les travaux de Jean-Sébastien Sottet.

```
PORT Handling
     IN : HandlingService

          Instantiate(      factoryId,
                            componentName,
                            instantiationId);

          Bind(peers, bindingId);

          Unbind(connectorId);

          Destroy(instanceId);

     OUT : HandlingNotifications

          ComponentInstantiationNotification(
                                  instantiationId,
                                  instanceId);

          BindindNotification(    bindingId,
                                  connectorId,
                                  errorCode);

Factory : PROVIDING
```

L'opération Instantiate()

L'opération Instantiate() permet à un tiers de demander à un fabrique l'instanciation d'un composant particulier.

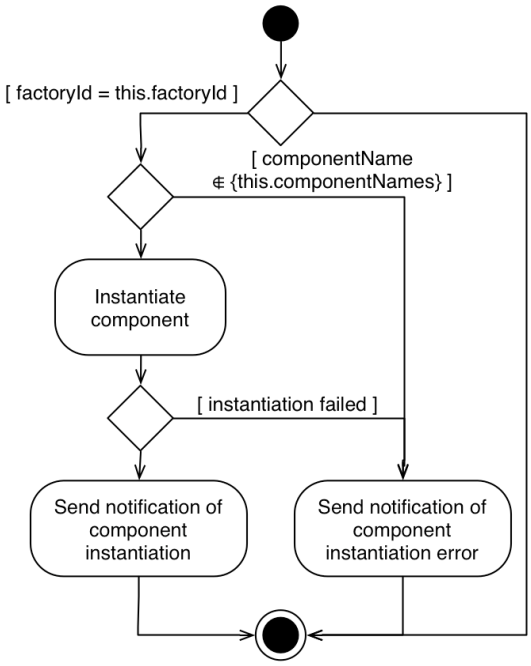


Figure V-10 Diagramme d'activité de l'opération Instantiate().

Cette opération accepte trois paramètres : *factoryId, componentName*, et *instantiationId*. Les deux premiers paramètres permettent d'identifier sans ambiguïté le composant à instancier sur la plate-forme. L'opération étant asynchrone, aucun résultat ne sera retourné directement à l'appelant, mais l'issue de l'opération est indiquée ultérieurement dans un message de notification. Dans ce cadre, le troisième paramètre spécifie une référence qui permet à l'appelant d'identifier, par la suite, le message de notification relatif à sa demande.

L'opération ComponentInstantiationNotification()

L'opération ComponentInstantiationNotification() constitue le message de notification qui indique le bon ou mauvais déroulement d'une opération Instantiate(). Elle accepte deux paramètres : *instantiationId* et *instanceId*. Le premier permet au récepteur de la notification de déterminer à quelle demande d'instanciation se rapporte la notification. Le deuxième paramètre contient l'identifiant de l'instance de composant créée, s'il y a lieu, ou est vide, si l'opération d'instanciation a échoué. Un identifiant de l'instance de composant doit être construit de manière à être unique sur l'ensemble de la plate-forme.

L'opération Destroy()

L'opération Destroy() permet de demander la destruction d'une instance de composant. Elle accepte, comme unique paramètre, *instanceId* contenant l'identifiant de l'instance de composant à détruire.

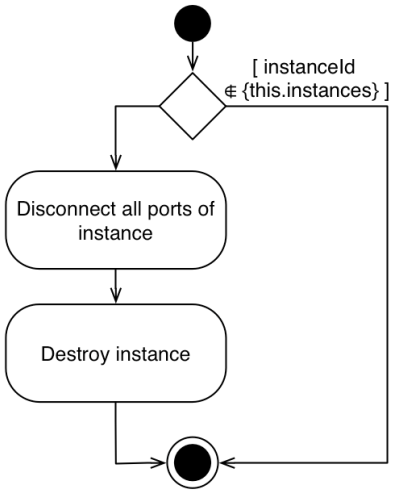


Figure V-11 Diagramme d'activité de l'opération Destroy().

Pour être détruit, une instance de composant ne doit pas nécessairement être préalablement déconnectée de ses pairs. Si l'instance possède des liaisons active avec des pairs, l'opération Destroy() prend en charge la déconnexion. Cette opération, qui ne peut échouer, ne renvoie aucun message de notification.

L'opération Bind()

L'opération Bind() permet d'établir une liaison entre deux instances de composant ou plus. Elle accepte deux paramètres : *peers* et *bindingId*. Le premier précise la liste des pairs, c'est-à-dire des identifiants d'instance de port, à relier. L'identifiant d'instance de port doit embarquer l'identifiant de l'instance de composant à laquelle l'instance de port appartient, le nom du port et sa direction. À l'instar de l'opération Instantiate(), le deuxième paramètre spécifie une référence qui permet, par la suite, à l'appelant d'identifier le message de notification relatif à sa demande de liaison. Le déroulement de cette opération consiste à élire le connecteur destiné à sous-tendre l'interaction logicielle entre les pairs. Deux situations sont possibles : soit tous les pairs à relier sont gérés par la même fabrique, soit ces pairs sont répartis entre différentes fabriques. Dans la première situation, l'opération bind() consiste alors, pour la fabrique concernée, de choisir, parmi les extrémités de connecteur qu'elle est capable de mettre en œuvre, celle qui satisfait les requis posés par les ports à relier. Sur la Figure V-12, ce processus est décrit par le processus qui forme la colonne de droite.

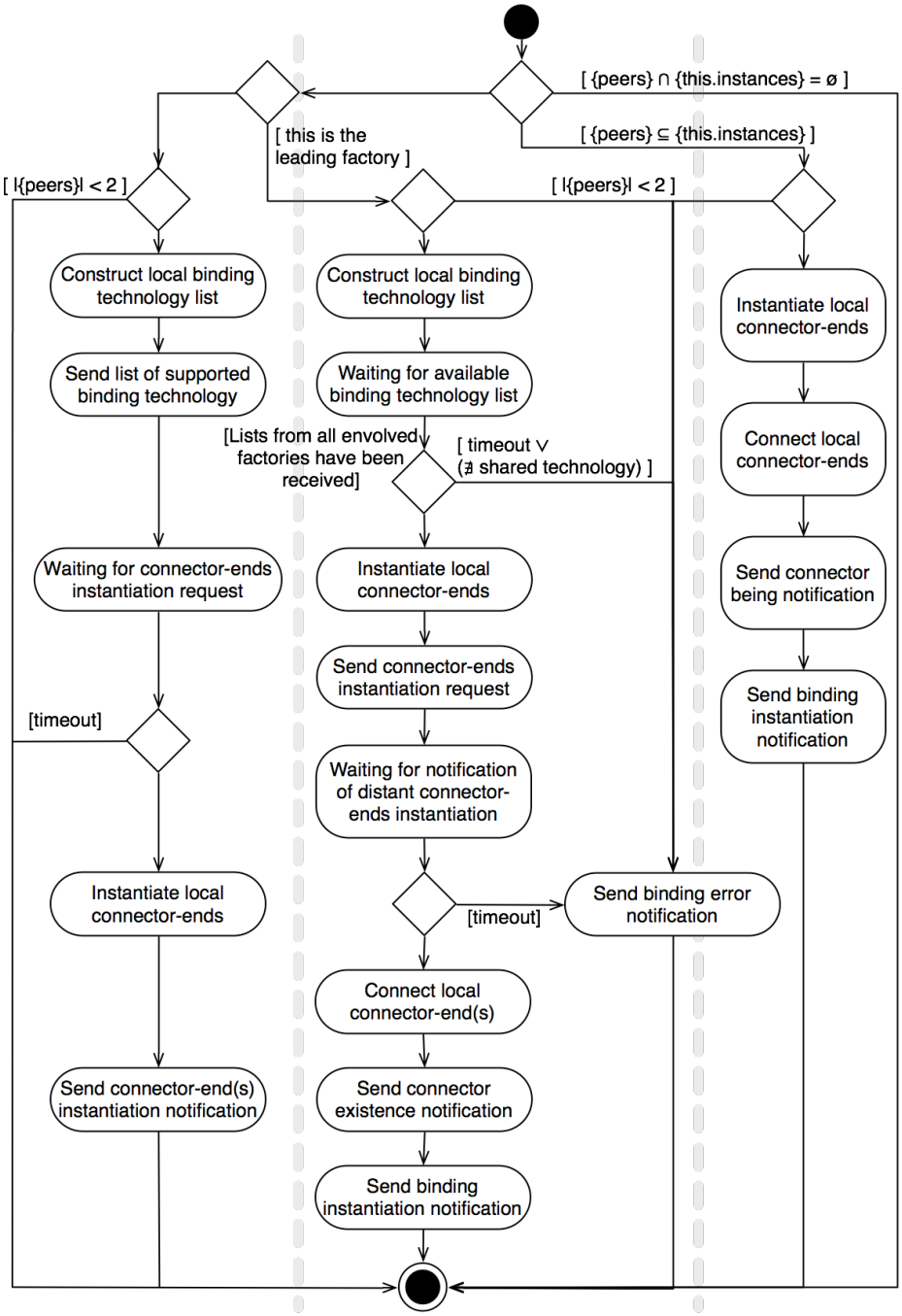


Figure V-12 Diagramme d'activité de l'opération Bind().

Dans la deuxième situation, les différentes fabriques doivent négocier, parmi les différentes extrémités de connecteur que chacune est capable de mettre en œuvre, celles qui partagent une technologie de communication commune à toutes les fabriques et qui satisfont les requis posés par les ports à relier. Ainsi, dans cette deuxième situation, le déroulement de l'opération Bind() définit les opérations de l'API Interoperability et la façon de les ordonner. La première étape de ce déroulement procède à l'élection, parmi les fabriques impliquées dans l'établissement de la liaison, de la fabrique dite « leader » qui sera chargée de diriger

la négociation. Dans la Figure V-12, la fabrique « leader » suit le processus de négociation décrit dans la colonne centrale, alors que les autres fabriques impliquées, dites « subordonnées », suivent celui de la colonne de gauche. Une fois le « leader » élu, celui-ci se met en attente des listes des technologies de communication disponibles dans chaque fabrique « subordonnée ». Une fois l'ensemble des listes reçues, parmi les technologies de communication partagées par toutes les fabriques impliquées dans la liaison à établir, le « leader » choisit la technologie qui satisfait au mieux les requis posés par les ports. Si aucune des technologies de communication partagées ne convient, une erreur est notifiée à l'appelant. Sinon, la fabrique « leader » envoie aux « subordonnées » l'ordre d'instancier et de brancher sur les instances de ports concernées les extrémités de connecteur choisies, puis se met en attente. Une fois reçue, de la part de chaque fabrique « subordonnée », la confirmation de l'instanciation des extrémités de connecteur, la fabrique « leader » établit la connexion entre les différentes extrémités de connecteur. Dès lors, le connecteur existe. La fabrique « leader » lui attribue un identifiant construit de façon à être unique sur la plate-forme et le communique, d'une part aux fabriques « subordonnées » afin qu'elles puissent gérer une éventuelle future opération Unbind(), et d'autre part à l'appelant, sous la forme d'un message de notification signifiant l'établissement de la liaison.

L'opération BindingNotification()

L'opération BindingNotification() constitue le message de notification qui indique le bon ou mauvais déroulement d'une opération Bind(). Elle accepte deux paramètres : *bindingId, connectorId* et *errorCode*. Le premier permet au récepteur de la notification de déterminer à quelle demande de liaison se rapporte la notification. Le deuxième paramètre contient l'identifiant du connecteur créé, s'il y a lieu, ou est vide, si l'opération de liaison a échoué. Dans ce cas, le dernier paramètre indique l'erreur survenue au cours de l'établissement de la liaison.

L'opération Unbind()

L'opération Unbind() permet de détruire un connecteur. Elle admet un paramètre, *connectorId*, qui contient l'identifiant du connecteur à détruire. Chacune des fabriques concernées par la suppression de ce connecteur est tenue de détruire les extrémités du connecteur dont elles ont la charge. Contrairement à l'opération Bind(), l'opération Unbind() peut-être menée à bien sans concertation préalable des différentes fabriques impliquées dans le connecteur à détruire. Cette opération ne pouvant pas échouer, aucune notification n'est envoyée en retour à l'appelant.

3.3.3 Interoperability

L'API Interoperability est établie à partir de la spécification de l'opération Bind(), précédemment décrite. Dans la mesure où

l'opération Bind() aurait certainement pu être conçue autrement, les opérations de l'API Interoperability pourraient être tout à fait différentes de ce qu'envisage ma proposition. Cependant, son objectif reste le même : décrire l'ensemble des opérations qui permettent aux fabriques de négocier le type d'extrémité de connecteur à utiliser dans le cadre de l'établissement d'une liaison entre deux ou plusieurs composants. Cette API présente la particularité d'être à la fois offerte et utilisée par chaque fabrique, chacune étant amenée à prendre, tour à tour, le rôle de « leader » de la négociation et celui de « subordonnée ».

```
PORT Interoperability

     IN : LeaderFactory

          SetBindingTechnologiesList(
                                bindingId,
                                nbManagedPeers,
                                technologiesList);

          NotifyConnectorEndInstantiation(
                                bindingId,
                                nbManagedPeers,
                                address,
                                errorCode);

     OUT : SubordinateFactory

          InstantiateConnectorEnd(bindingId,
                                peers,
                                technology,
                                address);

          NotifyConnectorExistence(
                                bindingId,
                                peers,
                                connectorId);

Factory : PROVIDING & USING
```

L'API Interoperability se compose de deux interfaces. La première, entrante, expose les deux opérations offertes par la fabrique « leader » pour que les « subordonnées » puissent la renseigner. La deuxième interface, sortante, expose les deux opérations utilisées par la fabrique « leader » pour diriger les « subordonnées ». Les opérations de ces interfaces étant asynchrones, il est possible de mener de front l'établissement de plusieurs liaisons. Les signatures de ces opérations sont donc conçues pour que leurs paramètres représentent l'ensemble des données nécessaires et suffisantes pour la réalisation de leur fonction. Ainsi, leur implémentation s'en trouve simplifiée : il n'y a pas à prévoir de gestion de l'historique des liaisons en cours d'établissement ailleurs que dans la fabrique « leader ».

L'opération SetBindingTechnologiesList()

L'opération SetBindingTechnologiesList() est utilisée par les fabriques « subordonnées » pour transmettre à la fabrique « leader » la liste des technologies de communication disponibles, qui soient adéquates au regard de la liaison en cours d'établissement. Elle accepte trois paramètres : *bindingId*, *nbManagedPeers*, et *technologiesList*. Le premier paramètre contient l'identifiant de la demande de liaison, transmis par l'opération Bind() de l'API Handling. Ainsi, la fabrique « leader » est en mesure de savoir à quelle demande de liaison les listes de technologies de communication fournies correspondent. Le deuxième paramètre indique le nombre de pairs que la fabrique « subordonnée » appelante gère. Cette information permet à la fabrique « leader » de déterminer si l'ensemble des fabriques impliquées dans l'établissement d'une liaison ont proposé leur liste de technologies de communication. Enfin, le dernier paramètre contient la liste des technologies de communication à transmettre.

L'opération InstantiateConnectorEnd()

L'opération InstantiateConnectorEnd() est utilisée par la fabrique « leader » pour demander à l'ensemble des fabriques « subordonnées » d'instancier les extrémités de connecteur qui correspondent à la technologie de communication choisie pour la liaison. Elle accepte quatre paramètres : *bindingId*, *peers*, *technology*, et *address*. Le premier contient l'identifiant de la demande de liaison. Le deuxième indique les pairs auxquels il faut brancher les extrémités de connecteur une fois celles-ci instanciées. Le troisième précise la technologie qui a été retenue pour la liaison. Le dernier représente l'adresse ou la référence qui sera nécessaire ultérieurement pour établir la connexion. Le format de ce paramètre est fortement dépendant de la technologie choisie.

L'opération NotifyConnectorEndInstantiation()

L'opération NotifyConnectorEndInstantiation() permet aux fabriques « subordonnées » de transmettre à la fabrique « leader » la confirmation de l'instanciation des extrémités de connecteur déclanchée par une opération InstantiateConnectorEnd(). Elle accepte quatre paramètres : *bindingId*, *nbManagedPeers*, *address*, et *errorCode*. Le premier contient l'identifiant de la demande de liaison. Le deuxième paramètre indique le nombre de pairs que la fabrique « subordonnée » appelante gère. Le troisième indique, si c'est nécessaire, par exemple pour une technologie où des connexions sont à établir dans les deux sens, l'adresse ou la référence qui sera nécessaire ultérieurement, pour établir la connexion. Comme pour l'opération InstantiateConnectorEnd(), le format de ce paramètre est fortement dépendant de la technologie choisie. Enfin, le dernier paramètre permet d'indiquer, si tel est le

cas, l'erreur rencontrée lors de l'instanciation de l'extrémité de connecteur.

L'opération NotifyConnectorExistence()

Enfin, l'opération NotifyConnectorExistence() permet à la fabrique « leader » d'informer les fabriques « subordonnées » de la réussite du processus d'établissement de la liaison. Elle accepte trois paramètres : *bindingId*, *peers* et *connectorId*. Le premier contient l'identifiant de la demande de liaison. Le deuxième paramètre rappelle l'ensemble des pairs impliqués dans la liaison. Le dernier paramètre indique l'identifiant de connecteur attribué à cette liaison. Les fabriques « subordonnées » ont alors toutes les informations nécessaires pour traiter une opération Unbind().

3.3.4 Conclusion

La description à haut niveau d'abstraction, c'est-à-dire qui soit indépendante des langages et des types de données, des API Availability, Handling et Interoperability achève ce chapitre de mes contributions sur le plan conceptuel. Le chapitre suivant, regroupant mes contributions sur le plan de l'implémentation, expose une façon d'appliquer, d'une part ma décomposition fonctionnelle, et d'autre part le modèle à composant dynamique Ethylene.

Chapitre VI

Contribution aux outils logiciels pour la réalisation d'IHM plastiques

Avant-propos

Au chapitre précédent, j'ai présenté mes contributions sur le plan des concepts, qui se situent à un niveau d'abstraction suffisamment élevé pour être indépendants de toutes technologies d'implémentation. Pour démontrer la validité de cette approche, il était nécessaire d'implémenter un prototype mettant en œuvre les mécanismes que je propose. La réalisation de ce prototype a nécessité plusieurs étapes :

(1) Afin d'être exploitable pour la conception de systèmes interactifs et pour leur exécution, le prototype devait correspondre au modèle Ethylene, un format à la fois utilisable par l'humain et exploitable par la machine.

(2) Afin d'être en mesure de développer le prototype, il fallait équiper une technologie à composants avec le mécanisme de fabrique et encapsuler sous la forme d'extrémité de connecteur quelques protocoles et technologies de communication.

(3) Un minimum des fonctions principales décrites par ma décomposition fonctionnelle devaient être disponible pour sous-tendre l'adaptation du prototype.

(4) Enfin, en s'appuyant sur le produit des trois points précédents, le dernier a consisté au développement du système interactif prototype proprement dit.

Les travaux issus des points (1) et (2) sont pleinement réutilisables dans un contexte différent de celui de mon démonstrateur. Du premier point est issu un langage XML qui sert à la description des composants et qui pourrait servir à générer du code de manière automatique. Du deuxième point est issu un cadre de développement, disponible en C++ et en Java, constitué d'une technologie à composants minimaliste, développé pour l'occasion, et un ensemble de classes permettant d'implémenter facilement des fabriques et des extrémités de connecteur. La troisième étape a consisté à participer au développement de MARI, une infrastructure logicielle pour la plasticité conçue dans le cadre des travaux de thèse de Jean-Sébastien Sottet, en y intégrant mes travaux autour d'Ethylene. Cette intégration a consisté à ajouter à cette infrastructure une fonction configurateur compatible avec Ethylene. Le produit de cette étape est donc fortement lié à MARI et ne peut être réutilisé sans modification dans un autre contexte. Enfin, le choix du cas d'étude de la dernière étape s'est porté sur le thème d'un outil de consultation de collections de photos. Le système interactif qui en résulte est constitué d'un arbre des tâches

hébergé par l'écosystème de l'infrastructure MARI, et de plusieurs composants, qui se classent en trois catégories : les premiers sont patrimoniaux, les deuxièmes sont exécutables et développés spécifiquement à cette occasion et les derniers sont générés à la volée par MARI à partir de l'arbre des tâches de l'application.

Ce chapitre s'organise autour de ces quatre points. La première section présente EthyleneXML, le langage XML issu du modèle Ethylene dont les spécifications complètes sont données en annexe. La deuxième section détaille le cadre de développement dédié à Ethylene. La troisième introduit l'infrastructure MARI développée dans le cadre de la thèse de Jean-Sébastien Sottet et expose le principe mis en œuvre pour y intégrer l'approche Ethylene. La dernière section décrit l'architecture et le fonctionnement de PhotoBrowser, le prototype développé dans le cadre de cette thèse.

1. EthyleneXML : Ethylene sous la forme d'un langage interprétable par la machine

La première étape nécessaire à l'implémentation de mon prototype consiste à donner au modèle Ethylene une incarnation technique à la fois interprétable par la machine et manipulable par un concepteur humain. Cette incarnation est conçue en fonction de quatre requis. Les deux premiers sont directement posés par le modèle Ethylene. Les deux suivants se positionnent sur un plan plus technique.

(1) Afin de respecter le niveau d'abstraction d'Ethylene, son incarnation technique doit rester indépendante de toutes technologies à composants ou à services, outils de développement ou langages de programmation.

(2) Comme le format des valeurs des *propriétés* du modèle Ethylene n'est pas défini par Ethylene, son incarnation technique doit permettre de représenter des valeurs exprimées dans des formats tiers.

(3) L'incarnation technique doit permettre, autant que possible, la réutilisation de formats existants dont l'usage est répandu dans l'état de l'art.

(4) Enfin, pour assurer l'unicité des identifiants des entités produites, l'incarnation technique du modèle Ethylene doit comporter un mécanisme de gestion d'espace de noms.

1.1 Le choix de XML

Au regard de ces quatre critères, le langage XML apparaît comme un support particulièrement adapté. Son extensibilité permet la construction de langages exprimant différentes sémantiques, tout en partageant la même syntaxe. Grâce à la mécanique d'espace de noms aboutie qu'il intègre, XML offre le moyen, d'une part, de construire des documents par assemblage d'éléments qui peuvent chacun se conformer à des langages différents, et d'autre part, de garantir l'unicité des noms des entités produites par un langage. En outre, la disponibilité d'une variété d'analyseurs dans la quasi-totalité des langages de programmation garantit son indépendance technologique vis-à-vis de langages de programmation, d'outils de développement ou de technologies à composants ou à services. Enfin, plusieurs langages s'appuyant sur XML, tels que WSDL ou XMLSchema, largement utilisés dans le domaine des approches à services, présentent des qualités qui justifient leur utilisation dans mon approche. Ainsi, XML semble être le meilleur candidat pour donner corps sur un plan technique au modèle Ethylene.

L'incarnation technique du modèle Ethylene, que j'appelle EthyleneXML, est un langage XML dont la structure est spécifiée par le schéma XML disponible en annexe. L'objectif principal de ce langage est d'être un outil de description, d'un point de vue externe, de composants Ethylene. À ce titre, EthyleneXML trouve son utilité à la fois dans les phases de conception et d'exécution. À la conception, il est un moyen donné au concepteur humain, d'une part pour spécifier des composants, et d'autre part à terme, pour générer automatiquement le code propre à la technologie à composants cible choisie. Cependant, aucun outil de génération automatique de code à partir d'EthyleneXML n'a été développé à ce jour. À l'exécution, EthyleneXML est le format d'échange de description de composants entre les différentes parties logicielles. Par exemple, il constitue le langage à l'aide duquel une fabrique enregistre les composants dont elle est responsable auprès des annuaires et sert de support d'expression des termes selon lesquels sont négociés les contrats d'assemblage.

1.2 Les éléments du langage EthyleneXML

Les balises d'EthyleneXML s'inscrivent dans l'espace de noms `http://iihm.imag.fr/namespaces/ethylenexml/1.0/`. Le préfixe recommandé pour désambiguïser les balises EthyleneXML est `c2h4`. Une spécification EthyleneXML est un document XML chargé de décrire un ensemble d'entités et de les inscrire au sein d'un espace de noms. Ainsi, l'élément racine d'un document EthyleneXML, qui se nomme `description`, accepte un attribut `targetNamespace` contenant l'URI de l'espace de noms défini par le document. Cet élément racine se déclare comme suit :

```
<c2h4:description
    xmlns:c2h4=http://iihm.imag.fr/namespaces/ethylenexml/1.0
   targetNamespace="the_uri_of_the_namespace_defined_by_this_document">
```

L'objectif d'un document EthyleneXML est la description de composants Ethylene. Un composant Ethylene est défini, d'une part par des propriétés caractérisées par un nom et un type, et d'autre part par des ports qui sont eux-mêmes caractérisés par des interfaces qui regroupent chacune un ensemble d'opérations qui, généralement, véhiculent des données typées. En outre, une propriété donnée ou un port donné sont applicables à plusieurs composants. De la même manière, une interface donnée est applicable à plusieurs ports, et un type donné à plusieurs propriétés ou données véhiculées par des opérations. Ainsi, un document EthyleneXML a pour vocation de décrire l'ensemble des six classes d'entités suivantes : composants, ports, interfaces, opération, types et propriétés. En outre, le langage permet de décrire chacune de ces entités indépendamment les unes des autres afin de faciliter le partage d'entités composantes (comme les interfaces, les propriétés ou les ports) entre plusieurs entités composées (comme les ports ou les composants). La structure du contenu de l'élément racine `description` est définie par l'extrait de schéma XML suivant :

```
<xs:element name="description">
  <xs:complexType mixed="false">
    <xs:sequence>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="import" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="types" minOccurs="0"/>
      <xs:element ref="interfaces" minOccurs="0"/>
      <xs:element ref="properties" minOccurs="0"/>
      <xs:element ref="ports" minOccurs="0"/>
      <xs:element ref="component" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="targetNamespace" type="xs:anyURI" use="required"/>
  </xs:complexType>
</xs:element>
```

Le contenu de l'élément `description` se constitue donc d'une séquence d'éléments, tous optionnels (*minOccurs*="0"). L'élément `annotation` est un moyen donné à l'auteur du document d'introduire des éléments de documentation. Il est disponible à tous les niveaux d'une spécification EthyleneXML. L'élément `import` offre la possibilité d'importer des entités décrites au sein d'espace de noms tiers. Je reviendrai sur ce mécanisme d'importation à la fin de cette section. L'élément `types` héberge les descriptions d'un ensemble de structures de données exprimées à l'aide du langage XMLSchema. Il est possible de décrire ces structures de données, soit en incluant directement sous la balise `types` des balises XMLSchema, soit en référençant, au moyen d'un élément `import` un document XMLSchema donné. De la même manière, l'élément `interfaces` héberge les descriptions

d'un ensemble d'interfaces programmatiques. Comme le langage WSDL est particulièrement adapté pour ce type de spécification, le contenu de l'élément `interfaces` se compose de descriptions écrites à l'aide du langage WSDL 2.0[69]. Comme précédemment, soit une description WSDL est insérée directement dans le document EthyleneXML, soit elle forme un document WSDL à part entière qui sera référencé sous la balise `interfaces` au moyen d'un élément `import`. Enfin, l'élément `properties` contient une liste de descriptions de propriétés, l'élément `ports` présente une liste de descriptions de ports et enfin un ensemble d'éléments `component` permet la description des composants peuplant l'espace de noms défini par le document EthyleneXML.

1.2.1 Définition d'une propriété

L'élément `properties` regroupe la liste des définitions des propriétés attenantes à l'espace de noms cible du document EthyleneXML. Une propriété se définit par un identifiant dont l'unicité au sein de l'espace de noms doit être garanti, et un type de données dont la structure doit être explicitée par un schéma XML. En EthyleneXML, une propriété est représentée par un élément `property` défini par l'extrait de schéma XML suivant :

```
<xs:element name="property">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:NCName" use="required"/>
    <xs:attribute name="type" type="xs:QName" use="required"/>
  </xs:complexType>
</xs:element>
```

Un élément `property` comprend donc deux attributs dont la présence est obligatoire. L'attribut `name` contient l'identifiant de la propriété. L'unicité de sa valeur doit être garantie au sein de l'espace de noms que définit le document EthyleneXML. L'attribut `type`, dont la valeur est un nom qualifié[70], désigne la définition d'un élément `complexType` ou `simpleType` du

[69] Lors de mes travaux de thèse, WSDL 2.0 avait le statut de « W3C Candidate Recommandation » et était défini par l'espace de noms http://www.w3.org/2006/01/wsdl. Aujourd'hui, WSDL 2.0 est une « W3C Recommandation » définie par l'espace de noms http://www.w3.org/ns/wsdl. Les modifications mineures entre ces deux versions n'ont aucun d'impact sur l'utilisation de WSDL 2.0 dans EthyleneXML. Dans la suite de ce rapport, la mention WSDL fait référence exclusivement à WSDL 2.0.

[70] En XML, un nom qualifié est un nom associé à un préfixe d'espace de noms destiné à résoudre une éventuelle ambiguïté. Par exemple, `xs:QName` désigne l'élément `QName` appartenant à l'espace de noms dont le préfixe est `xs`. Les préfixes se déclarent au moyen de l'attribut `xmlns` de l'élément racine d'un document XML.

langage XMLSchema qui définit une structure de données. Cette définition peut se trouver soit directement sous la balise `types` du document EthyleneXML courant, soit dans un document tiers importé par une balise `import`. Le contenu d'un élément `property` se limite à de la documentation, par le biais d'un ou plusieurs éléments `annotation`.

1.2.2 Définition d'une interface

Les interfaces qui composent les ports des composants Ethylene sont spécifiées par une description WSDL. Habituellement, une description WSDL permet de décrire une multitude d'éléments. Pour une utilisation dans le cadre d'EthyleneXML, une description WSDL ne doit être composée que d'éléments `interface`. L'ensemble des extraits de schéma XML qui décrivent l'élément `interface` de WSDL étant trop volumineux pour être présenté ici, j'emploie une description abrégée dite « pseudo-schéma » définie par le W3C[71]. Un élément `interface` est donc défini de la sorte par WSDL :

```
<wsdl:description>
  <wsdl:interface     name="xs:NCName"
                      extends="list of xs:QName"?
                      styleDefault="list of xs:anyURI"? >
    <wsdl:documentation/>*
    [ <wsdl:fault/> | <wsdl:operation/> ]*
  </wsdl:interface>
</wsdl:description>
```

La spécification d'un élément WSDL `interface` s'inscrit donc dans un document dont la racine est l'élément WSDL `description`. L'élément `interface` se caractérise par trois attributs et un ensemble de sous-éléments. L'attribut `name` est le seul dont la présence est obligatoire. Il spécifie l'identifiant de l'interface. L'unicité de cet identifiant doit être garantie au sein de l'espace de noms décrit par la description WSDL. L'attribut `extends` précise la liste des identifiants des interfaces desquelles celle-ci dérive. Ces interfaces peuvent être définies dans un espace de noms tiers. L'attribut `styleDefault` définit le style par défaut utilisé par les opérations de l'interface. Dans le cadre d'EthyleneXML, cet attribut ne peut prendre que la valeur « http://www.w3.org/ns/wsdl/style/rpc » correspondant au style RPC (Remote Procedure Call). Il n'est donc pas nécessaire de le préciser. Le contenu de l'élément `interface` est constitué d'éléments de documentation (`documentation`), de description d'opérations (`operation`) et de description d'événements de propagation de fautes (`fault`). Dans le cade d'EthyleneXML, les éléments `fault` ne sont pas pris en compte à ce jour. Les

[71] Voir http://www.w3.org/TR/wsdl20/#bnfpseudoschemas.

éléments `operation` décrivent les opérations prises en charge par l'interface de la manière suivante :

```
<wsdl:operation        name="xs:NCName"
                       pattern="xs:anyURI"?
                       style="list of xs:anyURI"? >
   <wsdl:documentation/>*
   [ <wsdl:input/> | <wsdl:output/> | <wsdl:infault/> | <wsdl:outfault/> ]*
</wsdl:operation>
```

Dans WSDL, une opération se définit par trois attributs et un ensemble de sous-éléments. L'attribut `name` est le seul dont la présence est requise par WSDL. Sa valeur spécifie le nom de l'opération. Il doit être unique au sein d'une interface. L'attribut `pattern` précise le patron d'interaction logicielle que l'opération respecte. Il accepte trois valeurs : « In-Only », « Robust-In-Only » et « In-Out »[72]. Dans le cadre d'EthyleneXML, la présence de cet attribut est requise et ne peut prendre pour valeur que « In-Only » ou « In-Out ». La première valeur précise que l'opération ne retourne aucun résultat. À l'inverse, la deuxième précise le contraire. Le dernier attribut, `style`, précise de la même manière que l'attribut `styleDefault` de l'élément `interface`, le style d'interaction logicielle mis en œuvre par l'opération. Dans le cadre d'EthyleneXML, cet attribut est ignoré. Le contenu d'un élément `operation` est constitué d'éléments de documentation (`documentation`) et d'éléments qui décrivent les messages induits par l'opération. WSDL autorise quatre types de messages qui sont représentés par quatre types d'éléments : `input`, `output`, `infault` et `outfault`. Dans le cadre d'EthyleneXML, seuls les messages `input` et `output` sont utilisés. Leur rôle est de préciser quels types de données s'échangent dans le cadre de l'opération. Ils se définissent comme suit :

```
<wsdl:input            messageLabel="xs:NCName"?
                       element="union of xs:QName, xs:token"? >
   <wsdl:documentation />*
</wsdl:input>

<wsdl:output           messageLabel="xs:NCName"?
                       element="union of xs:QName, xs:token"? >
   <wsdl:documentation />*
</wsdl:output>
```

Sur le plan de leur spécification, les éléments `input` et `output` ne diffèrent que par leur nom. Par analogie avec le concept de fonction, au sens d'un langage de programmation, l'élément `input` modélise l'ensemble des paramètres d'appels de cette

[72] En réalité, ces valeurs sont des URI : http://www.w3.org/ns/wsdl/in-only correspond à In-Only, http://www.w3.org/ns/wsdl/robust-in-only à Robust-In-Only et http://www.w3.org/ns/wsdl/in-out à In-Out.

fonction et l'élément `output` modélise les données qu'elle retourne. En WSDL, les éléments `input` et `output` se définissent par deux attributs optionnels. L'attribut `messageLabel` permet de nommer le message. L'attribut `element` indique une référence vers la définition d'un élément `element` définissant, en XMLSchema, un type de structure de données associée à une entité nommée. Si la spécification WSDL prévoit qu'une opération peut comporter un nombre quelconque d'éléments `input` et `output`, les patrons d'interaction logicielle imposés par EthyleneXML contraignent l'opération à ne comporter, dans le cas d'une opération « In-Only », qu'un unique élément `input`, ou bien, dans le cas d'une opération « In-Out », qu'un seul élément `input` et un seul élément `output`. En conséquence, dans le cadre d'EthyleneXML, l'attribut `messageLabel` perdant de son utilité est ignoré.

1.2.3 Définition d'un port

Les interfaces spécifiées à l'aide de WSDL sont utilisées pour peupler des ports. La notion de port telle qu'elle est envisagée par le modèle Ethylene ne trouve pas d'incarnation dans le langage WSDL. Ainsi, EthyleneXML propose sa propre définition :

```
<xs:element name="port">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="interface" maxOccurs="unbounded"/>
      <xs:element ref="propertyRef" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:NCName " use="required"/>
    <xs:attribute name="mode" type="PortCommunicationMode" use="required"/>
  </xs:complexType>
</xs:element>
```

Un élément `port` se définit par deux attributs dont la présence est requise et deux types de sous-élément en plus du sous-élément `annotation` relatif à la documentation,. L'attribut `name` spécifie l'identifiant du port. Sa valeur doit être unique au sein de l'espace de noms défini par le document EthyleneXML qui le contient. L'attribut `mode` traduit le caractère synchrone ou asynchrone de l'interaction logicielle que ce port vise à sous-tendre. Ainsi, il peut prendre pour valeur les chaînes de caractères « SYNC » ou « ASYNC ». Le contenu de l'élément `port` spécifie, d'une part, les interfaces qui le composent, et d'autre part, les propriétés extra-fonctionnelles qui le caractérisent. Plutôt que les interfaces et les propriétés soient décrites directement dans l'élément `port`, celles-ci sont associées par référence. Ces références se construisent comme suit :

```
<xs:element name="interface">
<xs:complexType>
  <xs:attribute name="ref" type="xs:QName" use="required"/>
  <xs:attribute name="direction"
                type="InterfaceDirectionType"
                use="required"/>
</xs:complexType>
</xs:element>
```

Dans la description d'un port, une référence vers une interface est réalisée par l'élément `interface` du langage EthyleneXML. Cet élément, qui modélise une référence, ne doit pas être confondu avec l'élément `interface` du langage WSDL qui, lui, modélise l'interface elle-même. Ainsi, l'élément `interface` du langage EthyleneXML se caractérise par deux attributs dont la présence est requise. L'attribut `ref` spécifie l'identifiant de l'interface référencée. Sa valeur est un nom qualifié qui désigne la définition d'une interface WSDL. L'attribut `direction` indique le sens de l'interface référencée au sein du port. Conformément au modèle Ethylene, cet attribut accepte les valeurs « IN » et « OUT ». Pour avoir une raison d'être, un élément `port` doit contenir au moins une référence vers une interface de direction « IN ». En revanche, il peut contenir un nombre quelconque de références vers des propriétés.

```
<xs:element name="propertyRef">
  <xs:complexType  mixed="true">
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:any processContents="lax"/>
    </xs:sequence>
    <xs:attribute name="ref" type="xs:QName" use="required"/>
  </xs:complexType>
</xs:element>
```

La référence vers une propriété est modélisée par l'élément `propertyRef` du langage EtyhleneXML. Cet élément se compose d'un attribut dont la présence est requise et un contenu optionnel. L'attribut `ref` spécifie l'identifiant de la propriété référencée. Sa valeur est un nom qualifié qui désigne la définition d'une propriété EthyleneXML. Le contenu de l'élément `propertyRef` offre le moyen d'assigner une valeur à cette propriété. Ainsi, ce contenu accepte n'importe quel contenu textuel. Cependant, celui-ci doit être valide au regard de la définition du type de la propriété référencée. L'élément `propertyRef` sert de façon identique dans la spécification d'un composant.

1.2.4 Définition d'un composant

La spécification d'un composant est réalisée au moyen de l'élément `component`. Cette spécification consiste en un attribut dont la présence est requise et, en plus du sous-élément `annotation` relatif à la documentation, trois sous-éléments optionnels :

```
<xs:element name="component">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="propertyRef" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="portRef" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="content" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:NCName" use="required"/>
  </xs:complexType>
</xs:element>
```

L'attribut `name` spécifie l'identifiant du composant. L'unicité de cet identifiant doit être garantie au sein de l'espace de noms défini par le document EthyleneXML. Le contenu de l'élément `component` précise les caractéristiques du composant. Ces caractéristiques se composent des propriétés et des ports exposés par le composant, ainsi que d'un éventuel contenu si le composant modélisé est un composant transformable. Comme précédemment pour les ports, les propriétés d'un composant sont spécifiées par référence au moyen de l'élément `propertyRef`. Les ports du composant sont spécifiés en suivant le même principe :

```
<xs:element name="portRef">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="ref" type="xs:QName" use="required"/>
    <xs:attribute name="direction" type="PortDirectionType" use="required"/>
  </xs:complexType>
</xs:element>
```

Une référence sur un port est modélisée par l'élément `portRef`. Il se caractérise par deux attributs dont la présence est requise. L'attribut `ref` spécifie l'identifiant du port référencé. Sa valeur est un nom qualifié qui désigne la définition d'un port. L'attribut `direction` indique si le port décrit une offre de service ou un service que le composant souhaite utiliser. Conformément au modèle Ethylene, cet attribut peut prendre soit la valeur « PROVIDING », soit la valeur « USING ».

```
<xs:element name="content">
  <xs:complexType mixed="true">
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:any processContents="lax"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Enfin, l'élément `content` est un moyen d'embarquer directement dans la description d'un composant Ethylene, des modèles qui le caractérisent. Ainsi, il est possible d'embarquer au sein de cet élément, par exemple, un arbre des tâches ou une interface concrète. EthyleneXML n'impose aucun format pour structurer le

contenu de cet élément. Il appartient à l'auteur du document de préciser le format dans lequel est représenté ce contenu, par le biais d'une ou plusieurs propriétés du composant.

1.2.5 Balise `import` et espaces de noms

La spécification d'un composant comporte des références vers des ports ou des propriétés. Cependant, ces ports et propriétés peuvent être définis dans un espace de noms différent de celui du composant. Afin d'assurer le sens et la validité de la spécification, il convient d'indiquer, d'une part, dans quel espace de noms ces ports et propriétés sont définis, et d'autre part, la description de cet espace de noms. EthyleneXML attribue ce rôle à l'élément `import`, qui se définit comme suit :

```
<xs:element name="import">
  <xs:complexType>
    <xs:attribute name="namespace" type="xs:anyURI" use="required"/>
    <xs:attribute name="location" type="xs:anyURI" use="required"/>
  </xs:complexType>
</xs:element>
```

L'élément `import` se compose de deux attributs dont la présence est requise. L'attribut `namespace` spécifie l'URI de l'espace de noms à importer. L'attribut `location` indique, sous la forme d'une URL, la localisation du document qui définit cet espace de noms.

1.2.6 Recommandations et conventions de désignation

Afin de tirer le meilleur parti possible du langage EthyleneXML, je propose quelques recommandations d'utilisation qui s'appuient sur l'expérience acquise par l'implémentation de mon démonstrateur. Ces recommandations portent, d'une part sur la portée d'un espace de noms et sur l'opportunité de répartir sur plusieurs fichiers une description EthyleneXML, et d'autre part sur des conventions de désignation des fichiers et de construction des URI chargés d'identifier les espaces de noms produits.

En premier lieu, l'espace de noms est un moyen qui permet à une organisation de garantir l'unicité des identifiants des composants qu'elle spécifie. En outre, c'est également un moyen pour cette organisation de publier des spécifications de propriétés et de ports à destination d'autres organisations, afin que celles-ci soient en mesure de développer des composants compatibles. Dans ce cadre, il est intéressant, d'une part de construire un espace de noms autour d'un thème précis et clairement délimité, et d'autre part d'isoler la spécification de chaque composant. Ainsi, il est judicieux de séparer la spécification d'un espace de noms en différents documents : d'un côté un document qui regroupe les spécifications des propriétés et ports, et de l'autre un ensemble de documents qui spécifient chacun un composant. Par ailleurs, les outils disponibles à ce jour pour éditer des descriptions

XMLSchema ou WSDL ne savent le faire que si l'ensemble du document est un schéma XML ou une description WSDL. Cette limitation actuelle de ces outils conduit à définir dans des documents distincts les types et les interfaces relatifs à une description EthyleneXML. J'encourage donc l'auteur d'une spécification EthyleneXML à la répartir en plusieurs documents en adoptant la politique suivante :

- Une description XMLSchema qui regroupe les spécifications de l'ensemble des types et des éléments.

- Une description WSDL qui regroupe les spécifications de l'ensemble des interfaces.

- Un document EthyleneXML « racine », qui, d'une part, importe les descriptions XMLSchema et WSDL précédentes, et d'autre part, regroupe la spécification de l'ensemble des propriétés et des ports.

- Un document EthyleneXML par composant à spécifier.

Pour nommer ces différents documents, je propose la convention de désignation suivante :

- Le document EthyleneXML « racine » est nommé du préfixe choisi pour l'espace de noms. L'extension de nom de fichier à utiliser est : `.c2h4.xml`.

- Le document XMLSchema est nommé du préfixe choisi précédemment, auquel on accole un « d » minuscule. L'extension de nom de fichier à utiliser est : `.xsd`. Ce document se place dans un sous-dossier nommé `data/`.

- Le document WSDL est nommé du préfixe choisi précédemment, auquel on accole un « s » minuscule. L'extension de nom de fichier à utiliser est : `.wsdl`. Ce document se place dans un sous-dossier nommé `services/`.

- Les document EthyleneXML qui contiennent les descriptions des composants sont nommés des identifiants des composants. L'extension de nom de fichier à utiliser est : `.c2h4.xml`. Ces documents se placent dans un sous-dossier nommé `components/`.

Enfin, il reste à déterminer quelques règles pour construire l'URI d'un espace de noms EthyleneXML et choisir un préfixe pertinent. L'objectif premier de l'URI et de son préfixe est de garantir l'unicité des identifiants définis dans l'espace de noms. En pratique, il apparaît que c'est également un moyen de distinguer

les différentes versions d'une même spécification. En outre, l'espace de noms est aussi un moyen de publication. Ainsi, je propose de construire les URI identifiant une spécification EthyleneXML comme une URL HTTP suivant les structures suivantes :

- Pour l'espace de noms de la spécification dans lequel s'inscrivent les propriétés, ports et composants, une URI de la forme :

```
http://nom_organisation/namespaces/nom_specification/id_version
```

- Pour le sous-espace « structures de données », une URI de la forme :

```
http://nom_organisation/namespaces/nom_specification/id_version/data
```

- Pour le sous-espace « interfaces programmatiques», une URI de la forme :

```
http://nom_organisation/namespaces/nom_specification/id_version/services
```

Si le mécanisme de gestion d'espace de noms d'XML ne requiert pas qu'une URI identifiant un espace de noms corresponde à une URL existante, il semble que le W3C conçoive les URI de ses espaces de noms de la sorte. Cette approche lui permet de publier à l'adresse correspondante la spécification de l'espace de noms. Dans le cadre d'EthyleneXML, je propose d'appliquer les mêmes conventions. Ainsi, dans une URI identifiant une spécification EthyleneXML, il est recommandé que `nom_organisation` corresponde à l'adresse du site Internet de l'organisation auteur de la spécification.

1.3 Comment écrire une spécification d'espace de noms à l'aide d'EthyleneXML ?

Afin d'illustrer l'ensemble de cette section, je présente un exemple de spécification EthyleneXML. Cette spécification est directement tirée de mon démonstrateur. Il s'agit de l'espace de noms « mcomponents » qui, nous le verrons à la section 3 de ce chapitre, participe à l'intégration de l'approche Ethylene à MARI. Je présente ici quatre documents de cette spécification : le document XMLSchema qui définit les structures de données, le document WSDL qui définit les services, le document « racine » qui spécifie les propriétés et ports, et enfin un document EthyleneXML spécifiant un composant. En réalité, la spécification de « mcomponents » contient un document supplémentaire qui porte sur la description d'un deuxième composant.

Comment choisir l'URI et le préfixe de l'espace de noms ?

Cette spécification a été développée dans le cadre de mon équipe de recherche. Elle définit la première version d'une description

relative à l'interaction logicielle entre des composants logiciels et l'infrastructure MARI. Je décide de la nommer « mcomponents ». En suivant les recommandations décrites précédemment, je construis son URI :

```
http://iihm.imag.fr/namespaces/mcomponents/1.0
```

De la même manière que l'URI, son préfixe doit permettre l'identification de l'espace de noms et de sa version. Je décide d'utiliser comme préfixe : `mcomp10`.

Comment spécifier les structures de données attachées à « mcomponents » ?

En premier lieu, il convient de fixer l'URI de cette spécification. En suivant les recommandations précédentes, l'URI obtenue est :

```
http://iihm.imag.fr/namespaces/mcomponents/1.0/data
```

et son préfixe est : `mcomp10d`. Le nom de fichier recommandé pour ce document est : `data/mcomp10d.xsd`, et son contenu est le suivant (j'insère au fur et à mesure des commentaires) :

```
<xs:schema
     xmlns:xs=          "http://www.w3.org/2001/XMLSchema"
     xmlns:mcomp10d=    "http://iihm.imag.fr/namespaces/mcomponents/1.0/data"
     targetNamespace=   "http://iihm.imag.fr/namespaces/mcomponents/1.0/data">
```

Ces premières lignes constituent l'en-tête classique d'un schéma XML. L'attribut `xmlns:xs` précise que les balises préfixées par `xs` dans ce document se rapportent à l'espace de noms XMLSchema. L'attribut `xmlns:mcomp10d` indique que les balises préfixées par `mcomp10d` dans ce document se rapportent à l'espace de noms de la spécification des structures de données de « mcomponents ». Enfin, l'attribut `targetNamespace` spécifie que ce schéma définit l'espace de noms http://iihm.imag.fr/namespaces/mcomponents/1.0/data/. La suite du document spécifie une à une les structures de données (`complexType`) qui pourront être utilisées pour définir des propriétés ou des éléments de message :

```
  <xs:complexType name="ConceptSet">
    <xs:sequence>
      <xs:element name="concepts" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="ConceptValueSet">
    <xs:sequence>
      <xs:element name="tasks" type="mcomp10d:TaskVector"/>
    </xs:sequence>
  </xs:complexType>
```

```
  <xs:complexType name="Task">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="values" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="TaskVector">
    <xs:sequence>
      <xs:element name="count" type="xs:integer"/>
      <xs:element name="task" type="mcomp10d:Task"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="UserTask">
    <xs:sequence>
      <xs:element name="annotation" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute name="name" type="xs:string" use="required"/>
              </xs:extension>
           </xs:simpleContent>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>

  <xs:complexType name="UserTaskList">
    <xs:sequence>
      <xs:element name="task" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
```

Certains types de donnée définis ci-dessus, notamment `ConceptSet` et `ConceptValueSet`, ont pour vocation de servir dans des interfaces. WSDL requiert qu'ils soient encapsulés dans un élément, ce qui est réalisé par les lignes suivantes :

```
  <xs:element name="action" type="xs:boolean"/>
  <xs:element name="conceptSet" type="mcomp10d:ConceptSet"/>
  <xs:element name="conceptValueSet" type="mcomp10d:ConceptValueSet"/>
  <xs:element name="taskList" type="xs:string"/>
</xs:schema>
```

Comment spécifier les interfaces programmatiques relatives à l'espace de noms « mcomponents »

En suivant les recommandations exposées précédemment, l'URI de cette spécification est la suivante :

```
http://iihm.imag.fr/namespaces/mcomponents/1.0/services
```

De la même manière, son préfixe est : `mcomp10s`. Le nom de fichier de ce document est donc : `services/mcomp10s.wsdl`.

Le contenu du document est le suivant :

```
<wsdl:description
   xmlns:wsdl=        "http://www.w3.org/2006/01/wsdl"
   xmlns:mcomp10d=    "http://iihm.imag.fr/namespaces/mcomponents/1.0/data"
   xmlns:grl10d=      "http://iihm.imag.fr/namespaces/general/1.0/data"
 targetNamespace="http://iihm.imag.fr/namespaces/mcomponents/1.0/services">
```

L'en-tête d'une description WSDL reprend le principe de fonctionnement des en-têtes XMLSchema. On remarque qu'un préfixe d'espace de noms est déclaré. Il va permettre d'utiliser au sein de ce document des éléments définis par ailleurs. Cette déclaration ne fait qu'identifier l'espace de noms de ces éléments. Celui-ci ne sera importé que par l'intermédiaire du document racine de la spécification EthyleneXML. La suite du document WSDL spécifie les interfaces programmatiques :

```
  <wsdl:interface name="FunctionalCoreAdapter">
    <wsdl:operation  name="setConceptValues"
                     pattern="http://www.w3.org/2006/01/wsdl/in-only">
      <wsdl:input element="mcomp10d:conceptValueSet"/>
    </wsdl:operation>
```

La première opération de l'interface `FunctionalCoreAdapter` est du type « In-Only ». Elle accepte comme paramètre un élément `conceptValueSet` défini dans l'espace de noms identifié par le préfixe `mcomp10d`.

```
    <wsdl:operation  name="getConceptValues"
                     pattern="http://www.w3.org/2006/01/wsdl/in-out">
      <wsdl:input element="mcomp10d:conceptSet"/>
      <wsdl:output element="mcomp10d:conceptValueSet"/>
    </wsdl:operation>
```

La deuxième opération de l'interface `FunctionalCoreAdapter` est du type « In-Out». Elle accepte comme paramètre un élément `conceptSet` et retourne en résultat un élément `conceptValueSet`. Ces deux éléments sont définis dans l'espace de noms identifié par le préfixe `mcomp10d`.

```
    <wsdl:operation  name="doAction"
                     pattern="http://www.w3.org/2006/01/wsdl/in-only">
      <wsdl:input element="mcomp10d:action"/>
    </wsdl:operation>
  </wsdl:interface>

  <wsdl:interface name="UiControl">
    <wsdl:operation  name="setActiveTasks"
                     pattern="http://www.w3.org/2006/01/wsdl/in-only">
      <wsdl:input element="mcomp10d:taskList"/>
    </wsdl:operation>

    <wsdl:operation  name="setEnableTasks"
                     pattern="http://www.w3.org/2006/01/wsdl/in-only">
      <wsdl:input element="mcomp10d:taskList"/>
    </wsdl:operation>
```

```
    <wsdl:operation  name="setDisableTasks"
                     pattern="http://www.w3.org/2006/01/wsdl/in-only">
      <wsdl:input element="mcomp10d:taskList"/>
    </wsdl:operation>

    <wsdl:operation  name="updateConceptValues"
                     pattern="http://www.w3.org/2006/01/wsdl/in-out">
      <wsdl:input element="mcomp10d:conceptValueSet"/>
      <wsdl:output element="grl10d:booleanAck"/>
    </wsdl:operation>

    <wsdl:operation  name="getConceptValues"
                     pattern="http://www.w3.org/2006/01/wsdl/in-out">
      <wsdl:input element="mcomp10d:conceptSet"/>
      <wsdl:output element="mcomp10d:conceptValueSet"/>
    </wsdl:operation>
  </wsdl:interface>

  <wsdl:interface name="UiNotification">
    <wsdl:operation  name="conceptsChange"
                     pattern="http://www.w3.org/2006/01/wsdl/in-only">
      <wsdl:input element="mcomp10d:conceptValueSet"/>
    </wsdl:operation>

    <wsdl:operation  name="userAction"
                     pattern="http://www.w3.org/2006/01/wsdl/in-only">
      <wsdl:input element="mcomp10d:taskList"/>
    </wsdl:operation>
  </wsdl:interface>

  <wsdl:interface name="FunctionalCoreAdapterNotification">
    <wsdl:operation  name="conceptUpdate"
                     pattern="http://www.w3.org/2006/01/wsdl/in-only">
      <wsdl:input element="mcomp10d:conceptValueSet"/>
    </wsdl:operation>
  </wsdl:interface>
</wsdl:description>
```

Comment spécifier le document racine de l'espace de noms « mcomponents »

Une fois les structures de données et les interfaces définies, il convient d'écrire le document racine de la spécification. Ce document importe les différents espaces de noms dont dépend la spécification de « mcomponents » et regroupe les définitions de propriétés et de ports. Le nom de fichier de ce document est : `mcomp10.c2h4.xml`.

```
<c2h4:description
    xmlns:c2h4=        "http://iihm.imag.fr/namespaces/ethylenexml/1.0"
    xmlns:mcomp10s="http://iihm.imag.fr/namespaces/mcomponents/1.0/services"
    xmlns:mcomp10d=    "http://iihm.imag.fr/namespaces/mcomponents/1.0/data"
    targetNamespace=   "http://iihm.imag.fr/namespaces/mcomponents/1.0">
```

L'en-tête d'une description EthyleneXML reprend le principe de fonctionnement des en-têtes XMLSchema et WSDL. La suite du document décrit dans l'ordre : les importations, les types, les interfaces, les propriétés et enfin, les ports.

```
<c2h4:import
     namespace="http://iihm.imag.fr/namespaces/general/1.0"
     location="http://iihm.imag.fr/namespaces/general/1.0/grl10.c2h4.xml"/>
```

La balise `import` ci-dessus permet de faire le lien entre l'espace de noms utilisé précédemment dans la définition du document WSDL et sa spécification. Les sections `types` et `interfaces` qui suivent indiquent où trouver les spécifications des types et des interfaces utilisés dans ce document.

```
  <c2h4:types>
    <c2h4:import
namespace="http://iihm.imag.fr/namespaces/mcomponents/1.0/data"
location="http://iihm.imag.fr/namespaces/mcomponents/1.0/data/mcomp10d.xsd"/>
  </c2h4:types>

  <c2h4:interfaces>
    <c2h4:import
namespace="http://iihm.imag.fr/namespaces/mcomponents/1.0/services"
location="http://iihm.imag.fr/namespaces/mcomponents/1.0/services/mcomp10s.ws
dl"/>
  </c2h4:interfaces>

  <c2h4:properties>
    <c2h4:property name="supportedUserTasks" type="mcomp10d:UserTaskList"/>
    <c2h4:property name="taskModelUri" type="xs:string"/>
    <c2h4:property name="adapter" type="xs:string"/>
    <c2h4:property name="conceptSet" type="mcomp10d:ConceptSet"/>
  </c2h4:properties>

  <c2h4:ports>
    <c2h4:port name="DialogController" mode="SYNC">
      <c2h4:interface ref="mcomp10s:UiControl" direction="IN"/>
      <c2h4:interface ref="mcomp10s:UiNotification" direction="OUT"/>
    </c2h4:port>

    <c2h4:port name="FunctionalCoreAdapter" mode="SYNC">
      <c2h4:interface ref="mcomp10s:FunctionalCoreAdapter" direction="IN"/>
      <c2h4:interface ref="mcomp10s:FunctionalCoreAdapterNotification"
                      direction="OUT"/>
    </c2h4:port>
  </c2h4:ports>
</c2h4:description>
```

Ce document définit quatre propriétés et deux ports. Les deux ports mettent en œuvre une interaction logicielle synchrone, et contiennent chacun une interface en entrée qui modélise une offre de service et une interface en sortie qui modélise un système de notification. Ces six entités peuvent ainsi être utilisées par des composants qui s'inscrivent dans ce même espace de noms ou dans un espace de noms tiers.

Comment spécifier un composant de l'espace de noms « mcomponents »

Enfin, j'achève cet exemple en commentant la spécification d'un des composants de « mcomponents ». Un composant est spécifié

dans un document EhyleneXML qui lui est consacré à part entière. Ainsi, l'en-tête du document et les éléments `import` fonctionnent de la même manière que dans le document « racine » précédent. Le nom de fichier du document est formé à partir de l'identifiant du composant. Dans ce cas précis, le nom de fichier du document est donc `HciComponentProxy.c2h4.xml`.

```
<c2h4:description
   xmlns:c2h4=          "http://iihm.imag.fr/namespaces/ethylenexml/1.0"
   xmlns:grl10=         "http://iihm.imag.fr/namespaces/general/1.0"
   xmlns:mcomp10=       "http://iihm.imag.fr/namespaces/mcomponents/1.0"
   targetNamespace=     "http://iihm.imag.fr/namespaces/mcomponents/1.0">

   <c2h4:import
namespace="http://iihm.imag.fr/namespaces/mcomponent/1.0 "
location="http://iihm.imag.fr/namespaces/mcomponent/1.0/mcomp10.c2h4.xml"/>

   <c2h4:component name="HciComponentProxy">
      <c2h4:propertyRef ref="grl10:lifeCycleState">
         installable
      </c2h4:propertyRef>
      <c2h4:portRef ref="mcomp10:DialogController" direction="USING"/>
   </c2h4:component>
</c2h4:description>
```

La description de ce composant est succincte. Le composant est décrit par une propriété et un port. La propriété est définie dans l'espace de noms identifié par le préfixe `grl10`, dont l'importation est faite dans le document « racine » commenté précédemment. La description précise pour cette propriété, la valeur textuelle « installable ». Le port est déclaré « utilisé ». Ainsi, pour que ce composant puisse remplir son office, il doit être relié à un autre composant déclarant ce port « fourni ». Ce document regroupe les informations de description nécessaires et suffisantes pour enregistrer ce composant auprès d'un annuaire.

La présentation de cet exemple achève la section de description du langage EthyleneXML. Si ce langage est un outil de spécification de composants Ethylene, il ne permet pas de les implémenter. La section suivante présente le cadre de développement que j'ai réalisé au cours de mes travaux de thèse.

2. Un cadre de développement Ethylene

Le modèle Ethylene et son incarnation technique EthyleneXML sont conçus de manière à être indépendants de toutes technologies d'implémentation. Dans ce contexte, l'objectif d'un cadre de développement est d'apporter à une technologie d'implémentation donnée la capacité de fonctionner selon les principes posés par le modèle Ethylene. Cette capacité repose essentiellement sur deux éléments : d'une part l'existence d'entités logicielles en charge de

jouer le rôle de fabrique tel que le décrit Ethylene, et d'autre part des mécanismes logiciels en charge d'assurer la fonction d'extrémité de connecteur. Il était possible d'implémenter un cadre de développement Ethylene pour des technologies existantes comme Fractal ou WComp. Cependant, pour réaliser mon démonstrateur, j'ai trouvé plus simple et plus rapide d'implémenter une technologie à composants minimaliste, directement issue du découpage « composant-port-connecteur » introduit par le modèle Ethylene, dont j'avais la totale maîtrise. Cette section est donc organisée en deux parties. La première présente SlimComponent, la technologie à composants minimaliste développée dans le cadre de cette thèse qui incarne les concepts de composant, port et extrémité de connecteur. La deuxième expose EthyleneFramework, un cadre de développement Ethylene pour SlimComponent destiné à aider le programmeur dans le développement de fabriques.

2.1 Une technologie à composant minimaliste

La technologie SlimComponent n'a pas été développée avec l'ambition de devenir une technologie robuste et utilisable dans un contexte différent de celui de mon démonstrateur. Néanmoins, ce démonstrateur constitue une première mise en pratique des principes posés au chapitre précédent. Cette technologie, à ce jour disponible en C++ et en Java, repose sur quatre classes et cinq interfaces représentées sur la Figure VI-1.

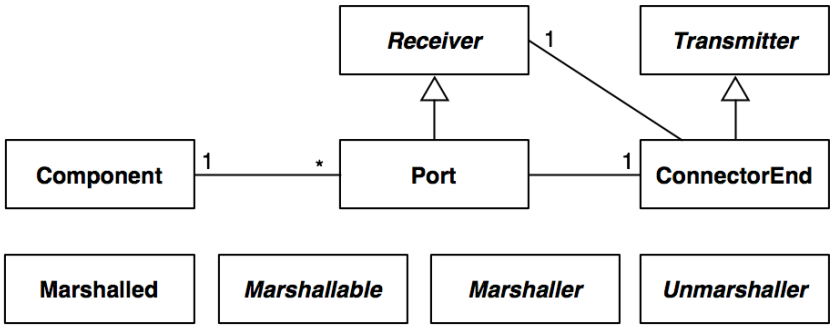


Figure VI-1 Vue d'ensemble des classes et interfaces de SlimComponent. Les noms des interfaces sont écrits en italique.

Ces neuf entités sont réparties en trois catégories : celles dont la fonction est d'encapsuler une logique interne au sein d'un composant (`Component` et `Port`), celles dont la fonction est d'encapsuler des logiques de communication (`ConnectorEnd`, `Transmitter` et `Receiver`) et enfin, les entités relatives à l'empaquetage et au dépaquetage des données échangées entre les composants (`Marshalled`, `Marshaller`, `Unmarshaller` et `Marshallable`). La logique interne d'une instance de `Component` est capable d'interagir avec une logique interne d'une autre instance de `Component` par l'intermédiaire d'instances de `Port`

reliées entre elles par le biais d'instances de `ConnectorEnd`, elles-mêmes interconnectées. Dans le cadre de ce chapitre, je m'appuie sur la version Java pour expliquer le principe de fonctionnement de cette technologie et le principe d'implémentation de composants SlimComponent. Cependant, ces principes sont directement transposables dans n'importe quel langage de programmation à objets.

2.1.1 Principe de fonctionnement

Le développeur de composants procède par dérivation : à une spécification EthyleneXML de composant correspond une sous-classe dérivée de `Component`. À une spécification EthyleneXML de port correspondent deux sous-classes dérivées de `Port`. L'une d'elles implémente le port dans une version « fourni », tandis que l'autre l'implémente dans une version « utilisé ». Le développement des extrémités de connecteur s'effectue sur le même principe : à une logique de communication à mettre en œuvre correspond une sous-classe dérivée de `ConnectorEnd`. Chacun de ces trois types de sous-classes endosse des fonctions précises. Les sous-classes de `Component` assurent l'instanciation et l'initialisation de la logique interne, ainsi que sa mise en relation avec les différentes instances de port du composant. Les sous-classes de `Port` assurent l'empaquetage et dépaquetage des données en transit et, suivant leur direction, les transmettent à l'extrémité de connecteur ou les délivrent à la logique interne. Enfin, les sous-classes de `ConnectorEnd` ont pour fonctions de fournir aux instances de port le moyen adéquat d'empaqueter et dépaqueter les données en transit, de mettre en œuvre des logiques de communication et de masquer leur hétérogénéité aux instances de ports. Le principe de fonctionnement de SlimComponent étant posé, je détaille à présent la façon de passer de la spécification d'un composant en EthyleneXML à son implémentation sous la forme d'un composant SlimComponent.

2.1.2 Comment implémenter un composant ?

L'implémentation d'un composant SlimComponent s'incarne sous la forme d'une sous-classe de `Component` chargée de la gestion de la logique interne et de sa mise en relation avec les différents ports du composant. L'interaction logicielle entre la logique interne et les ports du composant s'effectue par le biais des interfaces programmatiques qui définissent chaque port. Ainsi le développement de cette sous-classe se déroule en trois temps : le premier consiste à transcrire en Java les spécifications WSDL des interfaces programmatiques en jeu, le deuxième vise à implémenter la logique interne en fonction des interfaces programmatiques précédentes et le dernier temps est consacré à l'intégration du code correspondant à la logique interne au sein de la sous-classe de `Component`.

Transcription des interfaces programmatiques

La transcription d'une spécification d'une interface programmatique WSDL en java est réalisée en appliquant de façon systématique les règles suivantes :

- L'identifiant de l'interface WSDL devient le nom de l'interface Java.

Puis, pour chaque opération de l'interface considérée :

- Le nom de l'opération WSDL devient le nom de la méthode correspondante.
- Le nom de l'élément du message input de l'opération devient le nom du paramètre d'appel de la méthode.
- Le type de l'élément du message `input` de l'opération devient le type du paramètre d'appel de la méthode. Si ce type est complexe, il se traduit par le nom de la classe Java correspondante. Cette classe doit être une sous-classe de `Marshallable`.
- Si un message `output` est spécifié pour l'opération, le type de son élément devient le type du paramètre d'appel de la méthode. De la même manière que précédemment, si ce type est complexe, il se traduit par le nom de la classe Java correspondante. Cette classe doit être une sous-classe de `Marshallable`.

L'exemple ci-dessous illustre l'application de ces règles à l'interface `UiControl` présentée au point 1.2.7 précédent, dont je reproduit l'extrait de spécification ici :

```
<wsdl:interface name="UiControl">
     <wsdl:operation  name="setActiveTasks"
                      pattern="http://www.w3.org/2006/01/wsdl/in-only">
       <wsdl:input element="mcomp10d:taskList"/>
     </wsdl:operation>

     <wsdl:operation  name="setEnableTasks"
                      pattern="http://www.w3.org/2006/01/wsdl/in-only">
       <wsdl:input element="mcomp10d:taskList"/>
     </wsdl:operation>

     <wsdl:operation  name="setDisableTasks"
                      pattern="http://www.w3.org/2006/01/wsdl/in-only">
       <wsdl:input element="mcomp10d:taskList"/>
     </wsdl:operation>

     <wsdl:operation  name="updateConceptValues"
                      pattern="http://www.w3.org/2006/01/wsdl/in-out">
       <wsdl:input element="mcomp10d:conceptValueSet"/>
       <wsdl:output element="grl10d:booleanAck"/>
     </wsdl:operation>
```

```
    <wsdl:operation  name="getConceptValues"
                     pattern="http://www.w3.org/2006/01/wsdl/in-out">
      <wsdl:input element="mcomp10d:conceptSet"/>
      <wsdl:output element="mcomp10d:conceptValueSet"/>
    </wsdl:operation>
  </wsdl:interface>
```

L'application systématique des règles de retranscription produit l'interface Java suivante :

```
public interface IUiControl
{
  public void setActiveTasks(String tasks);
  public void setEnableTasks(String tasks);
  public void setDisableTasks(String tasks);
  public boolean updateConceptValues(ConceptValueSet list);
  public ConceptValueSet getConceptValues(ConceptSet list);
}
```

Dans l'interface Java présentée ci-dessus, les classes `ConceptValueSet` et `ConceptSet` sont des sous-classes de `Marshallable` et sont obtenues à partir de leur spécification XMLSchema.

Implémentation de la logique interne

Les interfaces Java obtenues à l'étape précédente modélisent les interactions logicielles entre le code dit interne, qui implémente la logique interne d'un composant, et le code hébergé au sein d'autres composants. Du point de vue du code interne, ces interfaces sont soit entrantes, soit sortantes. Elles sont entrantes lorsque leurs méthodes mettent une fonction à la disposition d'un autre composant. Elles sont sortantes lorsque l'implémentation de leurs méthodes est réalisée au sein d'un autre composant. Dans le code interne, une classe Java modélise une interface entrante par une interface déclarée implémentée. Une classe sortante est modélisée par un attribut de classe associé à un setter. L'extrait de code suivant illustre ce principe :

```
public class InternLogic implements Interface_A {
  private Interface_B interface_B;

  public void setInterface_B(Interface_B interface_B){
    this.interface_B = interface_B;
  }
…
}
```

Dans cet exemple, `Interface_A` est une interface entrante. Ces méthodes sont implémentées au sein de la classe `InternLogic`. L'interface `Interface_B` est sortante. Elle est modélisée par l'attribut `interface_B` qui est assignable via la méthode `setInterface_B()`.

Encapsulation de la logique interne

Une fois la logique interne implémentée, celle-ci doit être encapsulée afin de prendre la forme d'un composant. Cette encapsulation passe par le développement d'une classe dérivée de `Component`. Pour le développeur de composant, cette étape consiste à introduire le code interne sous la forme d'attribut de classe et de surcharger le constructeur de la classe `Component`.

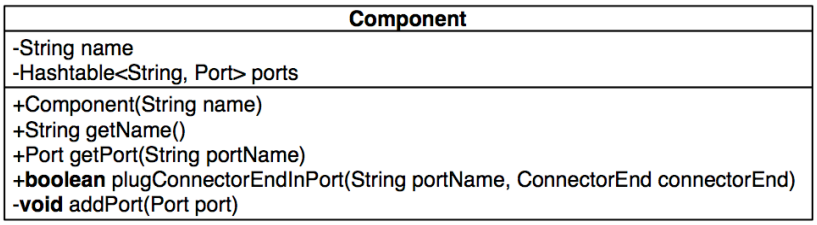


Figure VI-2 Attributs et méthode de la classe Component.

La surcharge du constructeur couvre trois points que je décris ci-dessous et qui sont illustrées par un extrait de code ensuite :

(1) *Nommer le composant* : le constructeur de la classe `Component` accepte un paramètre `name` destiné à fixer l'identifiant du composant. Le constructeur de la classe dérivée ne doit accepter aucun paramètre et il doit affecter au paramètre `name` du constructeur de la super-classe, l'identifiant du composant sous la forme d'une URI. Cette URI concatène celle de l'espace de noms de la spécification EthyleneXML du composant avec l'identifiant du composant qu'elle spécifie.

(2) *Instancier et ajouter les ports* : la spécification EthyleneXML d'un composant le décrit en termes de ports. Le rôle du constructeur de composant surchargé est d'instancier les sous-classes de `Port` correspondantes et de les ajouter au composant par l'intermédiaire de la méthode protégée `addPort()` (voir Figure VI-2).

(3) *Instancier et initialiser le code interne* : le code interne se présente sous la forme d'attributs de classe qu'il faut instancier et initialiser. Afin que ces instances puissent recevoir, ou envoyer, des messages vers, ou en provenance, de l'extérieur du composant, elles doivent être reliées aux instances de ports du composant. Cette liaison est établie par le développeur en utilisant les setters définis lors de l'implémentation de la logique interne et ceux définis dans les classes dérivées de `Port`.

L'extrait de code suivant illustre ces trois points en encapsulant la classe `InternLogic`, vue à l'étape précédente, dans un composant `SampleComponent` qui fournit un port `SamplePort` constitué de

l'interface « IN » `Interface_A` et de l'interface « OUT » `Interface_B`.

```
public class SampleComponent extends Component {
  private InternLogic internLogic;

  public SampleComponent(){
    // Nommer le composant
    super("http://specification_uri/SampleComponent");

    // Instancier et ajouter le port
    Port port = new SamplePort();
    this.addPort(port);

    // Instancier et initialiser la logique interne
    internLogic = new InternLogic ();
    internLogic.setInterface_B(port);
    port.setInterface_A(internLogic);
    …
  }
…
}
```

Comme la transcription des spécifications d'interfaces WSDL vers du code Java, l'encapsulation de la logique interne peut être réalisée, en partie, en suivant des règles de façon systématique. Seule la dernière étape, fortement dépendante de la structure du code interne, ne peut être dirigée de façon systématique à partir de la spécification EthyleneXML d'un composant.

2.1.3 Comment implémenter un port ?

De manière similaire à l'implémentation d'un composant, l'implémentation d'un port est réalisée à partir de sa spécification EthyleneXML par dérivation de la classe `Port`.

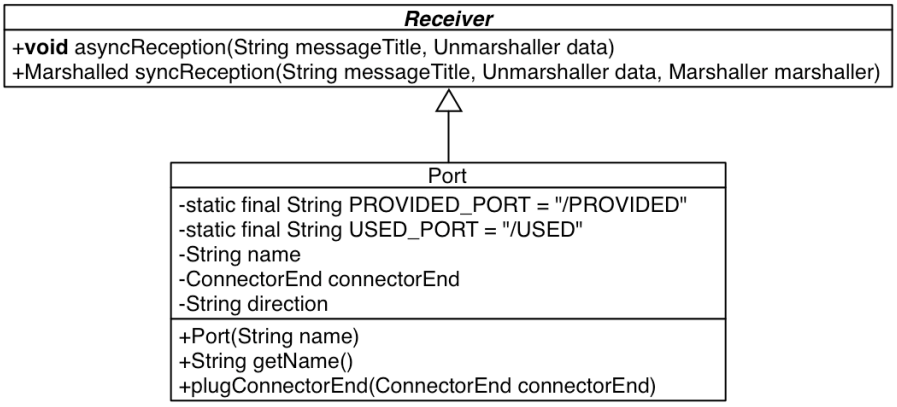


Figure VI-3 Attributs et méthodes de la classe Port.

La classe `Port` implémente l'interface `Receiver`. De cette interface, elle hérite des deux méthodes `asyncReception()` et `syncReception()`, destinées à l'interaction logicielle du port avec l'extrémité de connecteur. Par ailleurs, elle déclare deux

constantes, `PROVIDED_PORT` et `USED_PORT`, destinées à représenter la direction du port, au sens posé par le modèle Ethylene. Un port se caractérise par trois attributs : l'attribut `name` contient l'identifiant du port tel qu'il a été déclaré dans la spécification EthyleneXML, préfixé par l'URI de son espace de noms et suffixé, suivant la direction du port, par la valeur de `PROVIDED_PORT` ou de `USED_PORT`. L'attribut `connectorEnd` contient une référence vers l'extrémité de connecteur associée au port ou vaut `null` si aucune extrémité de connecteur n'est associée. Enfin l'attribut `direction` contient la valeur `PROVIDED_PORT` ou `USED_PORT` suivant la direction du port. Lors de l'implémentation d'un port, le développement de la classe dérivée de `Port` couvre trois points : l'intégration des interfaces spécifiées par la description EthyleneXML du port, l'implémentation des méthodes `asyncReception()` et `syncReception()`, et l'implémentation des méthodes de chaque interface sortante.

Intégration des interfaces

Comme pour le code interne, une interface est soit entrante, soit sortante. Lorsqu'un port est implémenté dans sa version « fourni », les interfaces dont la direction est spécifiée « IN » par la spécification EthyleneXML sont entrantes. Symétriquement, les interfaces dont la direction est spécifiée « OUT » par la spécification EthyleneXML sont sortantes. À l'inverse, lorsque le port est implémenté dans sa version « utilisé », les interfaces « IN » sont sortantes et les interfaces « OUT » sont entrantes. L'intégration d'une interface au sein de l'implémentation d'un port suit le principe inverse de l'intégration de cette interface au sein du code interne. Ainsi, dans une sous-classe de `Port`, une interface entrante est modélisée par un attribut de classe associé à un setter, alors qu'une interface sortante est déclarée comme interface implémentée. En outre, le développeur du port doit surcharger le constructeur de la classe afin d'initialiser les attributs `name` et `direction` issus de la classe mère, avec les valeurs adéquates. L'extrait de code suivant illustre ces principes appliqués au port `SamplePort` utilisé dans l'exemple précédent (voir au point 2.1.2).

```
public class SamplePort extends Port implements Interface_B {
   private Interface_A interface_A;

   public SamplePort(){
      super("http://specification_uri/SamplePort" + Port.PROVIDED_PORT);
      direction = Port.PROVIDED_PORT;
      interface_A = null;
   }
   public setInterface_A(Interface_A interface_A){
      this.interface_A = interface_A;
   }
…
}
```

Implémentation de `asyncReception()` et `syncReception()`

Les méthodes `asyncReception()` et `syncReception()` sont utilisées par l'extrémité de connecteur pour délivrer au code interne des messages en provenance d'un autre composant. Selon la terminologie d'Ethylene, ces messages représentent des opérations. La méthode `asyncReception()` est utilisée pour traiter les messages asynchrones, la méthode `syncReception()` traitant les messages synchrones. Les messages asynchrones correspondent aux opérations qui ne renvoient aucun résultat. Inversement, les messages synchrones correspondent aux opérations qui retournent un résultat. Les méthodes `asyncReception()` et `syncReception()` ont pour fonction d'identifier le message reçu, de dépaqueter les données qu'il véhicule et d'appeler la méthode correspondante du code interne. En outre, la méthode `syncReception()` doit attendre la réponse de la logique interne, l'empaqueter et la transmettre à l'extrémité de connecteur. Les algorithmes d'empaquetage et de dépaquetage des données dépendent de la technologie de communication employée par l'extrémité de connecteur. Ainsi l'extrémité de connecteur est en charge de fournir au port les algorithmes adéquats. Ces algorithmes sont fournis sous la forme d'instances de sous-classes de `Marshaller` et de `Unmarshaller`. Je reviendrai sur le fonctionnement de ces deux classes au point 2.1.5.

La méthode `asyncReception()` accepte deux paramètres : `messageTitle` et `data`. Le premier contient l'identifiant du message. Il doit correspondre à l'identifiant de l'opération qu'il représente. Le deuxième paramètre contient les données à transmettre à cette opération sous une forme empaquetée. La méthode `syncReception()` accepte les deux paramètres décrits précédemment plus un troisième nommé `marshaller`. Ce paramètre est une référence qui pointe sur une instance de classe destinée à prendre en charge l'empaquetage des données à retourner. Quels que soient les messages à transmettre ou les données véhiculées, l'algorithme des méthodes `asyncReception()` est le suivant :

```
SI le code interne et le port sont reliés l'un à l'autre ALORS
  SI messageTitle == "nom_operation" ALORS
    donnees_depaquetees <- DEPAQUETER data;
    code_interne.nom_operation(donnees_depaquetees);
```

Dans le cas des méthodes `syncReception()`, le code interne retourne un résultat qui doit être transmis à l'extrémité de connecteur. L'algorithme précédent est donc complété de la manière suivante :

```
SI le code interne et le port sont reliés l'un à l'autre ALORS
  SI messageTitle == "nom_operation" ALORS
    donnees_depaquetees <- DEPAQUETER data;
    reponse <- code_interne.nom_operation(donnees_depaquetees);
    reponse_empaquetee <- EMPAQUETER reponse
    RETOURNER reponse_empaquetee
```

Implémentation des interfaces sortantes

Si les méthodes `asyncReception()` et `syncReception()` permettent à un port de recevoir les messages en provenance d'autres composants, un port a également pour fonction d'émettre des messages à destination de ces autres composants. Ces messages qui représentent également des opérations selon la terminologie d'Ethylene sont émis par le biais des méthodes des interfaces sortantes. Ces méthodes acceptent un paramètre conforme à la spécification WSDL contenant les données à transmettre. Au niveau du port, l'implémentation de ces méthodes consiste à tester si une extrémité de connecteur est associée au port, à empaqueter les données à transmettre, à solliciter l'extrémité de connecteur de façon idoine pour émettre le message et, suivant la nature synchrone ou asynchrone de l'opération, à attendre une réponse du destinataire. Comme pour les méthodes `asyncReception()` et `syncReception()`, celles des interfaces sortantes sont construites suivant le même algorithme quels que soient le message et les données à transmettre. L'algorithme pour les méthodes qui représentent une opération asynchrone est le suivant :

```
SI une extrémité de connecteur et le port sont reliés l'un à l'autre ALORS
    empaqueteur <- connector_end.getMarshaller();
    donnees_empaquetees <- EMPAQUETER donnees_en_parametres;
    connector_end.asyncTransmission(   "nom_operation",
                                       donnees_empaquetees);
```

L'algorithme pour les méthodes qui représentent une opération synchrone est le suivant :

```
SI une extrémité de connecteur et le port sont reliés l'un à l'autre ALORS
    empaqueteur <- connector_end.getMarshaller();
    donnees_empaquetees <- EMPAQUETER donnees_en_parametres;
    reponse_empaquetee <- connector_end.syncTransmission(
                                                    "nom_operation",
                                                donnees_empaquetees);
    reponse <- DEPAQUETER reponse_empaquetee;
    RETOURNER reponse;
```

Comme le montre les algorithmes présentés précédemment, les méthodes des interfaces sortantes s'appuient sur les méthodes `getMarshaller()`, `asyncTransmission()` et `syncTransmission()` fournies par les instances des classes d'extrémité de connecteur.

2.1.4 Comment implémenter une extrémité de connecteur ?

L'implémentation d'une extrémité de connecteur a pour objectif d'encapsuler une technologie de communication afin qu'elle soit utilisable par des instances de ports. Le développeur d'extrémité de connecteur procède par dérivation de la classe abstraite `ConnectorEnd`, et par implémentation des interfaces `Marshaller` et `Unmarshaller`. La dérivation de la classe `ConnectorEnd` consiste à mettre en œuvre le protocole de communication choisi. L'implémentation des interfaces `Marshaller` et `Unmarshaller` est destinée à produire l'empaqueteur et le dépaqueteur de données adaptés à ce protocole de communication. Par exemple, le développeur décide d'utiliser HTTP comme protocole de communication. La classe dérivée de `ConnectorEnd` est alors chargée de la mise en œuvre des sockets TCP adéquates pour sous-tendre la communication et des algorithmes relatifs à l'échange de messages HTTP. La classe issue de `Marshaller` implémente un algorithme qui permet la transformation d'une structure de données en mémoire sous la forme de champs HTTP. De manière symétrique, la classe issue de `Unmarshaller` implémente un algorithme inverse qui permet la transformation de champs HTTP sous la forme d'une structure de données en mémoire conforme à l'originale. Dans cette partie, je détaille tour à tour ces deux interfaces et cette classe abstraite, tout en expliquant comment obtenir des classes concrètes à partir d'elles.

Dériver la classe `Marshaller`

L'interface `Marshaller` définit une façon standard d'interagir avec un empaqueteur quel que soit son type. Cette interface comprend six méthodes (voir Figure VI-4) : les quatre premières sont destinées à l'empaquetage de données de types simple (booléen, entiers, nombre à virgule flottante et chaîne de caractère), les deux suivantes permettent la description de structures complexes. Ces cinq premières méthodes sont à l'usage de l'entité logicielle en charge de l'empaquetage. Je reviendrai sur cette entité logicielle au paragraphe 2.1.5 . La dernière méthode permet d'obtenir les données empaquetées. Elle est utilisée dans les sous-classes de `Port`. À l'instar de l'ensemble des mécanismes propres à SlimComponent, cette interface est minimaliste. Elle permet d'empaqueter des structures de données dont la complexité est limitée mais suffisante dans le cadre mon démonstrateur. Pour un usage dans un cadre plus large, cette interface pourrait être enrichie sensiblement.

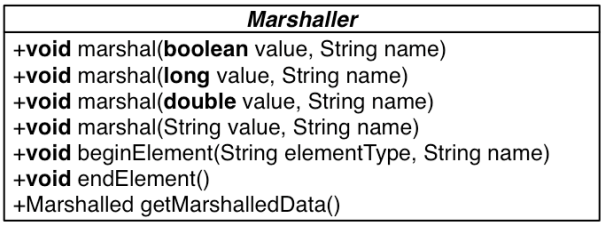


Figure VI-4 Méthodes de l'interface Marshaller.

Les quatre premières méthodes (`marshal(boolean,String)`, `marshal(long,String)`, `marshal(double,String)` et `marshal(String,String)`) fonctionnent sur le même principe. Elles acceptent deux paramètres : le premier est une valeur de type simple et le deuxième est une chaîne de caractères permettant de lui associer un nom au sein des données empaquetées. Si la donnée à empaqueter est de type simple, l'entité chargée de l'empaquetage appelle directement la méthode `marshal()` correspondante. Si la donnée à empaqueter est une structure complexe, l'entité chargée de l'empaquetage doit commencer par appeler `beginElement()`. Le premier paramètre de cette méthode précise un nom de type et le deuxième paramètre indique un nom à associer à la donnée. Puis, pour chaque attribut de type simple de la structure complexe à empaqueter, l'entité logicielle chargée de l'empaquetage appelle la méthode `marshal()` correspondante. Pour chaque attribut de type complexe, elle appelle de nouveau `beginElement()`. La fin de chaque structure complexe est marquée par un appel à la méthode `endElement()`.

L'empaqueteur est chargé d'allouer l'espace mémoire destiné à recevoir les données sous leur forme empaquetée. Une fois les données empaquetées, la méthode `getMarshalledData()` permet à un port d'obtenir une référence vers cet espace mémoire. Les données empaquetées au moyen d'un empaqueteur peuvent être dépaquetées au moyen du dépaqueteur correspondant.

Dériver la classe `Unmarshaller`

L'interface `Unmarshaller` définit une façon standard d'interagir avec un dépaqueteur quel que soit son type. Cette interface comprend cinq méthodes (voir Figure VI-5), toutes destinées à l'usage de l'entité logicielle en charge du dépaquetage. Je reviendrai sur cette entité logicielle au paragraphe 2.1.5 .

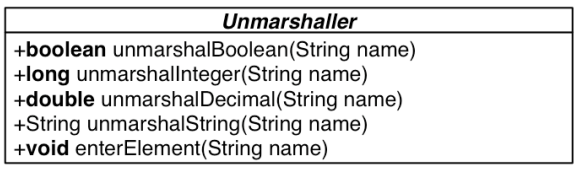


Figure VI-5 Méthodes de l'interface Unmarshaller.

Les quatre premières méthodes (`unmarshalBoolean(String)`, `unmarshalInteger(String)`, `unmarshalDecimal(String)` et `unmarshalString(String)`) permettent de dépaqueter des valeurs de type simple et fonctionnent sur le même principe. Elles acceptent un paramètre qui représente le nom donné à l'empaquetage de la valeur à dépaqueter. Elles retournent la valeur correspondante sous sa forme originale. La cinquième méthode, `enterElement(String)`, permet de dépaqueter des valeurs de type complexe. Comme pour les quatre premières méthodes, le son paramètre précise le nom de la valeur à dépaqueter. Cette méthode ne retourne aucun résultat, mais change l'état interne du dépaqueteur pour que les prochains appels de méthode sur celui-ci portent sur la valeur de type complexe spécifiée. Puis, pour chaque attribut de type simple de la structure complexe à dépaqueter, l'entité logicielle chargée du dépaquetage appelle la méthode `unmarshal()` correspondante et pour chaque attribut de type complexe, elle appelle de nouveau `enterElement()`.

De manière similaire à l'empaqueteur, le dépaqueteur gère un espace mémoire destiné à contenir les données sous leur forme empaquetées. Les classes concrètes issues des interfaces `Marshaller` et `Unmarshaller` étant écrites, le développeur d'extrémité de connecteur peut alors dériver la classe `ConnectorEnd` afin de mettre en œuvre le protocole de communication choisi.

Dériver la classe `ConnectorEnd`

La classe `ConnectorEnd` définit neuf méthodes, dont deux sont héritées de l'interface `Transmitter` (voir Figure VI-6). Ces neuf méthodes se répartissent en deux catégories : la première regroupe les méthodes à l'usage de la fabrique, chargée du choix et de la mise en place des extrémités de connecteurs ainsi que de l'établissement des liaisons ; la deuxième catégorie regroupe les méthodes à l'usage des ports pour leur permettre d'échanger des messages à travers un connecteur. Par ailleurs, la classe `ConnectorEnd` définit trois attributs : `connected`, `technology` et `receiver`. L'attribut `connected` représente l'état connecté ou non de l'extrémité de connecteur, l'attribut `technology` contient une chaîne de caractères qui identifie la technologie de communication mise en œuvre par l'extrémité de connecteur, et enfin, l'attribut `receiver` héberge une référence vers le port associé à l'extrémité de connecteur.

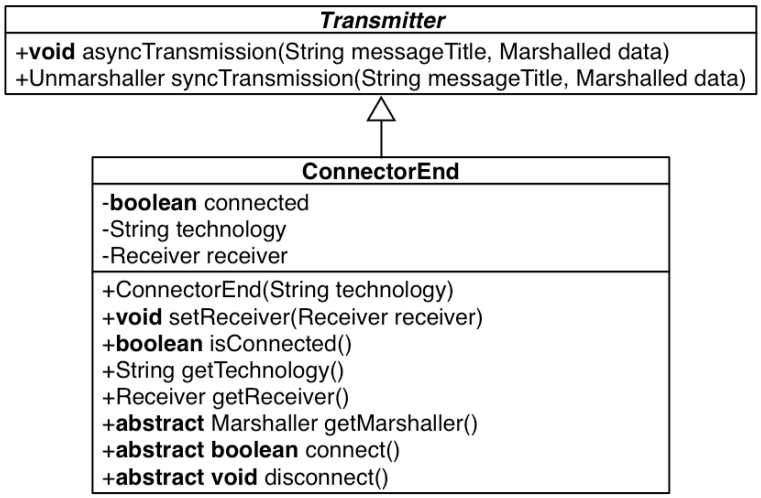


Figure VI-6 Attributs et méthodes de la classe ConnectorEnd.

La dérivation de la classe `ConnectorEnd` est un processus en trois étapes. Dans la première, le développeur ajoute et implémente les méthodes nécessaires au protocole de communication. Afin de garantir la propriété d'interchangeabilité des extrémités de connecteur vis-à-vis des ports, ces méthodes doivent seulement être destinées à l'usage interne du code de l'extrémité de connecteur, ou à l'usage d'une fabrique. La deuxième étape consiste à surcharger le constructeur. Cette surcharge a deux objectifs. Le premier est d'assigner l'identifiant de la technologie de communication mise en œuvre à l'attribut `technology`. Ainsi, le constructeur de la classe dérivée n'a pas besoin d'accepter de paramètre. Le deuxième objectif est d'initialiser les entités et les algorithmes relatifs à la mise en œuvre du protocole de communication.

La dernière étape consiste à implémenter les méthodes abstraites de la classe `ConnectorEnd`. Les deux premières méthodes à implémenter sont à l'usage des fabriques. L'objectif de la méthode `connect()` est d'établir la liaison avec la ou les extrémités de connecteur distantes qui auront été préalablement identifiées. À l'issue de l'exécution réussie de cette méthode, l'attribut `connected` doit valoir `true` sinon `connect()` doit retourner `false`. Inversement, l'objectif de la méthode `disconnect()` est de rompre cette liaison, interdisant tout échange de message entre l'extrémité de connecteur concernée et les autres. À l'issue de l'exécution de cette méthode, l'attribut `connected` doit valoir `false`.

Les trois dernières méthodes abstraites définies par la classe `ConnectorEnd` sont à l'usage des ports. La méthode `getMarshaller()` a pour objectif de retourner une instance de la sous-classe adéquate de `Marshaller`, afin que le port soit en mesure d'empaqueter les données des messages qu'il souhaite

transmettre. La méthode `asyncTransmission()` permet à un port de transmettre un message asynchrone via l'extrémité de connecteur. Cette méthode accepte deux paramètres : `messageTitle` et `data`. Le paramètre `messageTitle` contient le nom de l'opération représentée par le message. Le paramètre `data` contient, sous une forme empaquetée, les données correspondant au paramètre de l'opération. Le résultat attendu de l'exécution de cette méthode est la transmission du message aux extrémités de connecteur connectées. De manière similaire, la méthode `syncTransmission()` permet à un port de transmettre un message synchrone via l'extrémité de connecteur. Les paramètres qu'elle accepte sont les mêmes que ceux de la méthode précédente. Lors de son exécution, une fois le message envoyé, elle doit se mettre en attente du résultat de l'opération que celui-ci représente. Lorsque le résultat de l'opération arrive, la méthode `syncTransmission()` instancie la sous-classe adéquate de `Unmarshaller` et lui fournit les données empaquetées du résultat. Enfin, cette instance d'une sous-classe de `Unmarshaller` est retournée au port qui sera en mesure de dépaqueter.

Ainsi, via les méthodes `getMarshaller()` et `syncTransmission()` une extrémité de connecteur est en mesure de fournir à un port l'empaqueteur et le dépaqueteur adéquats. Cependant, si l'empaqueteur et le dépaqueteur encapsulent les algorithmes d'empaquetage et de dépaquetage, ils n'ont aucune connaissance sur les structures des données à empaqueter ou à dépaqueter. Seuls, ils ne sont donc pas en mesure de les empaqueter ou de les dépaqueter. Dans le point suivant, je traite des gestionnaires chargés de diriger les opérations d'empaquetage et de dépaquetage.

2.1.5 Comment implémenter des structures de données empaquetables et dépaquetables ?

Dans mon implémentation, chaque structure de données destinée à être paramètre d'opération est son propre gestionnaire d'empaquetage et de dépaquetage. Ainsi, toutes doivent implémenter l'interface `Marshallable` (voir Figure VI-7). En outre, ces structures de données ne peuvent être composées que d'attributs de type simple ou d'attributs de classes qui implémentent également l'interface `Marshallable`. Cette interface définit deux méthodes : `marshal(Marshaller, String)` et `unmarshal(Unmarshaller, String)`.

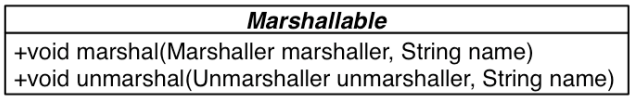


Figure VI-7 Méthodes de l'interface Marshallable.

La première méthode est chargée de diriger l'empaquetage de la structure de données. Elle accepte deux paramètres : `marshaller` et `name`. Le paramètre `marshaller` doit être une instance d'un empaqueteur fournie par une extrémité de connecteur. Le paramètre `name` est une chaîne de caractères qui permet d'associer un identifiant aux données dans leur version empaquetée. Cette méthode ne retourne aucun résultat. Cependant, à l'issue de son exécution, `marshaller` contient une version empaquetée des données. L'extrait de code suivant montre un exemple d'implémentation de la méthode `marshal()` pour la structure de données `SampleData` constituée de deux attributs de type simple et d'un attribut de type complexe :

```
public class SampleData implements Marshallable {
   private long attribut1;
   private AComplexType attribut2;
   private String attribut3;
   …
   public void marshal(Marshaller marshaller, String name) {
      marshaller.beginElement("SampleData", name);
      marshaller.marshal(attribut1, "attribut1");
      attribut2.marshal(marshaller, "attribut2");
      marshaller.marshal(attribut3, "attribut3");
   }
   …
}
```

La deuxième méthode est chargée de diriger le dépaquetage de la structure de données. Elle accepte deux paramètres : `unmarshaller` et `name`. Le paramètre `unmarshaller` est une instance de dépaqueteur contenant les données à dépaqueter, fournie par une extrémité de connecteur. Le paramètre `name` est une chaîne de caractères qui correspond au nom qui identifie les données à dépaqueter. Cette méthode a un fonctionnement similaire à la précédente. Elle ne retourne aucun résultat. Cependant, à l'issue de son exécution, l'instance de la structure de données contient les valeurs dépaquetées. L'extrait de code suivant montre un exemple d'implémentation de la méthode `unmarshal()` pour la structure de données `SampleData` de l'exemple précédent :

```
public class SampleData implements Marshallable {
   …
   public void unmarshal(Unmarshaller unmarshaller, String name) {
      unmarshaller.enterElement(name);
      this.attribut1 = unmarshaller.unmarshalInteger("attribut1");
      attribut2.unmarshal(unmarshaller, "attribut2");
      this.attribut3 = unmarshaller.unmarshalString("attribut3");
   }
   …
}
```

Ces deux méthodes sont utilisées par les ports, afin d'empaqueter les données à transmettre via une extrémité de connecteur, ou de

dépaqueter des données reçues par l'intermédiaire d'une extrémité de connecteur. L'extrait de code suivant montre l'implémentation d'une méthode d'une interface sortante d'un port chargée d'émettre un message asynchrone qui représente l'opération asynchrone `sampleOperation(SampleData data)` :

```
public class SamplePortProvided extends Port implements Interface_B {
…
   public void sampleOperation(SampleData data) {
      if(connectorEnd != null) {
         Marshaller m = connectorEnd.getMarshaller();
         data.marshal(m, "data");
         connectorEnd.asyncTransmission(  "sampleOperation"
                                          m.getMarshalledData());
      }
   }
…
}
```

Symétriquement à l'exemple précédent, l'extrait de code suivant montre l'implémentation de la partie de la méthode `asyncReception()` du port qui reçoit le message issu de l'exemple précédent :

```
public class SamplePortUsed extends Port {
…
   public void asyncReception(String messageTitle, Unmarshaller data) {
      if(internLogic != null) {
         …
         if(messageTitle.compareTo("sampleOperation") == 0) {
            SampleData d = new SampleData();
            d.unmarshal(data, "data");
            internLogic.sampleOperation(d);
         }
         …
      }
   }
…
}
```

2.1.6 En résumé

Dans cette partie, j'ai présenté SlimComponent, une technologie à composants minimaliste dont les capacités sont limitées mais suffisantes au regard de mon démonstrateur. Son originalité vient de sa conception : SlimComponent est une instanciation très directe du modèle Ethylene. Ainsi, un composant SlimComponent se décompose d'une part en code interne encapsulé par une classe chargée de sa gestion, et d'autre part en ports. Par ailleurs, SlimComponent permet le développement d'extrémités de connecteur chargées de l'implémentation de protocoles de communication. Afin que plusieurs composants puissent interagir, chacun de leurs ports impliqués dans cette interaction logicielle doit être associé à une extrémité de connecteur, puis ces extrémités de connecteur doivent être interconnectées. De cette interconnexion résulte un connecteur. Lorsqu'un composant

souhaite appliquer une opération sur un ou plusieurs autres composants, cette action se traduit par un échange de message sur le connecteur. Les données véhiculées par ces messages doivent être empaquetées préalablement par le port chargé de les envoyer, puis dépaquetées par le port qui les reçoit. Les algorithmes de paquetage et de dépaquetage, qui dépendent des protocoles de communication, sont fournis aux ports par les extrémités de connecteur.

Ainsi, comme le modèle Ethylene le préconise, SlimComponent met en œuvre un découplage fort entre les composants d'une part, et les protocoles de communication entre composants d'autre part. Cependant, cette technologie à composants minimaliste ne fournit aucun mécanisme de gestion dynamique des composants, ni de gestion de l'interopérabilité avec d'autres technologies à composants. Ces mécanismes, incarnés par le concept de fabrique dans le modèle Ethylene, doivent donc être appliqués à SlimComponent pour que cette technologie à composants puisse convenir au développement de systèmes interactifs plastiques. L'application à SlimComponent des principes définis par le modèle Ethylene a été capitalisée au sein d'un cadre de développement que je présente dans la section suivante.

2.2 Un cadre de développement Ethylene pour SlimComponent

Le cadre de développement Ethylene, que j'appelle EthyleneFramework, offre au programmeur un ensemble de classes et d'interfaces sur lesquelles s'appuyer, d'une part pour développer des composants SlimComponent capables de fonctionner suivant le cycle de vie proposé par le modèle Ethylene, et d'autre part pour développer des fabriques qui assurent la disponibilité dynamique de ces composants et leur interopérabilité avec des composants logiciels issus d'autres technologies. Cette section s'articule autour de trois parties. La première introduit brièvement les conventions de désignation des entités et les notations graphiques utilisées dans le cadre d'EthyleneFramework. La deuxième partie présente les éléments du cadre de développement destinés à l'implémentation de composants contrôlables, c'est-à-dire capables de gérer les opérations `start`, `stop` et `destroy` conformément au cycle de vie proposé par le modèle Ethylene. Enfin, la dernière partie présente les éléments du cadre de développement relatifs à l'implémentation du concept de fabrique du modèle Ethylene.

2.2.1 Convention de désignation et notation graphique

Le cadre de développement EthyleneFramwork regroupe un ensemble de classes abstraites, d'interfaces programmatiques, de ports et de composants. Pour faciliter l'identification de ces différentes entités par le programmeur, je propose la convention de désignation suivante :

- Les noms des interfaces programmatiques destinées à définir des ports débutent par un « I » majuscule suivi du nom de l'interface. Par exemple, l'interface `LeaderFactory` devient `ILeaderFactory`.

- Au niveau d'un document de spécification, les noms de ports débutent par un « P » majuscule suivi du nom du port. Au niveau du code, s'ajoute au nom du port le mot « Used » ou « Provided » suivant la version implémentée du port. Par exemple, le port `Handling` devient `PHandling` dans un document de spécification `PHandlingUsed` et `PHandlingProvided` lors de son implémentation.

- Les noms des composants débutent par un « C » majuscule suivi du nom du composant. Par exemple, le composant `PhotoShuffler` devient `CPhotoShuffler`.

Afin de faciliter la représentation graphique des composants, des ports et des interfaces dans un document, je propose d'utiliser la notation UML de la manière suivante : Un composant est représenté par une classe qui hérite de la classe `Component`. Un port est représenté par une classe qui hérite de la classe `Port`. Une interface est représentée par une classe dont le nom est inscrit en italique. Les interfaces qui définissent un port sont indiquées par des relations étiquetées « IN » ou « OUT » suivant la direction de l'interface considérée (voir Figure VI-8).

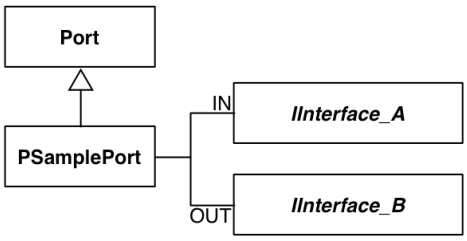


Figure VI-8 Exemple d'un port représenté selon la notation graphique définie par EthyleneFramework : le port PSamplePort est constitué d'une interface entrante IInterface_A et d'une interface sortante IInterface_B.

De la même manière, les ports qui définissent un composant sont indiqués par des relations étiquetées « USING» ou « PROVIDING» suivant la direction du port considéré (voir Figure VI-9).

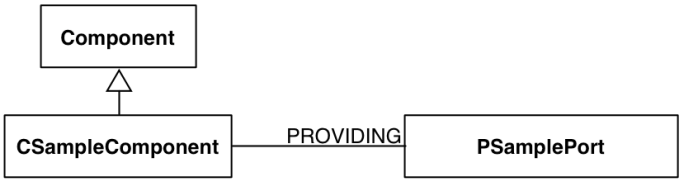


Figure VI-9 Exemple de composant représenté selon la notation graphique définie par EthyleneFramework : le composant CSampleComponent fournit un port PSamplePort.

Enfin, les fabriques et les composants qu'elles hébergent se représentent suivant le même principe : une fabrique est représentée par une classe qui hérite de la classe `Factory`. Les composants hébergés par une fabrique lui sont associés par des relations étiquetées « MANAGE » (voir Figure VI-10).

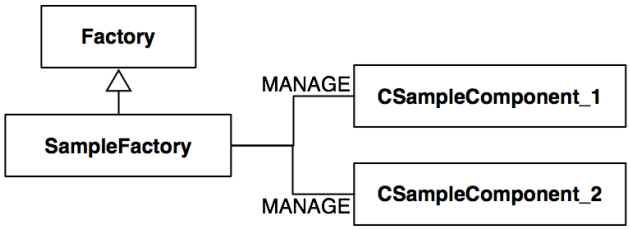


Figure VI-10 Exemple d'une fabrique représentée selon la notation graphique définie par EthyleneFramework. La fabrique SampleFactory est capable de mettre en œuvre les composants CSampleComponent_1 et CSampleComponent_2.

En s'appuyant sur ces conventions de désignation et cette notation graphique, la suite de cette section présente les différents éléments de EthyleneFramework en commençant par les éléments relatifs aux composants contrôlables.

2.2.2 Comment implémenter un composant contrôlable ?

Selon le cycle de vie des composants proposé par le modèle Ethylene, l'exécution d'un composant est contrôlée par la fabrique responsable de son instanciation au moyen des opérations `start`, `stop` et `destroy`. Dans SlimComponent, la classe abstraite `Component` n'offre pas cette possibilité. Le cadre de développement EthyleneFramework propose donc la classe `CEthyleneComponent`, sous-classe de `Component`, qui offre un cadre au programmeur pour le traitement des opérations `start`, `stop` et `destroy`.

La classe `CEthyleneComponent`

La classe `CEthyleneComponent` (voir Figure VI-11) a pour vocation de servir de super-classe à tous les composants SlimComponent destinés à être utilisés dans une fabrique et régis

suivant le cycle de vie du modèle Ethylene. Un composant issu de `CEthyleneComponent` fournit par défaut un port `PComponentManagement`. Ce port se compose d'une interface entrante `IComponentManagement` qui déclare les trois opérations `start()`, `stop()` et `destroy()`. Ces trois opérations fonctionnent sur le même principe : elles acceptent un paramètre `instanceId` contenant l'identifiant d'une instance de composant ; si cet identifiant correspond à celui du composant, l'opération `start()`, `stop()` ou `destroy()` est transmise à la logique interne afin que cette dernière puisse prendre les dispositions adéquates. Ainsi, la classe `CEthyleneComponent` déclare un attribut `instanceId`, de type chaîne de caractères destiné à stocker l'identifiant de l'instance de composant et un attribut `internalLogicControl` de type `InternalLogicControl` qui référence le point de contrôle sur la logique interne. `InternalLogicControl` (voir Figure VI-11) est une interface constituée des trois opérations `start()`, `stop()` et `destroy()` dépourvues de paramètre d'appel. L'attribut `instanceId` est mis à jour par la fabrique responsable de l'instanciation du composant via la méthode `setInstanceId()`. La valeur de l'attribut `internalLogicControl` doit être fixée par le développeur du composant au moyen de la méthode `setInternalLogicControl()`.

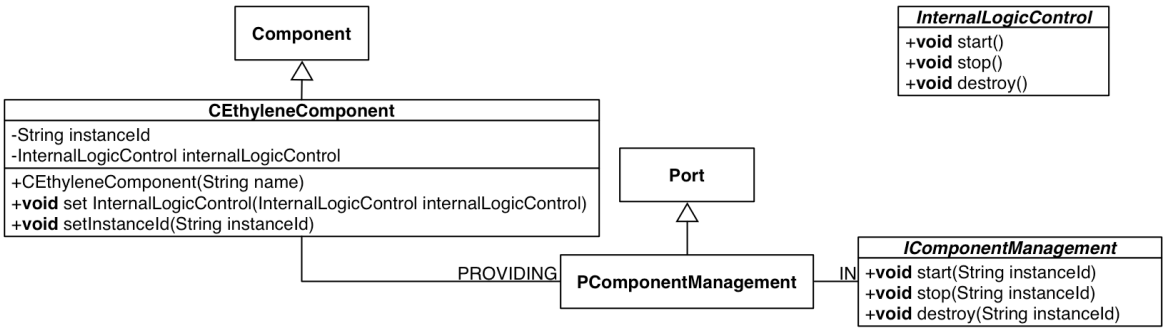


Figure VI-11 CEthyleneComponent et InternalLogicControl.

Implémentation d'un composant contrôlable

L'implémentation d'un composant contrôlable suit un processus de dérivation similaire à celui de l'implémentation d'un composant SlimComponent classique. Cependant, la logique interne doit offrir un point d'accès qui implémente l'interface `InternalLogicControl`. Le développeur procède ensuite par dérivation de la classe `CEthyleneComponent`. Comme pour l'implémentation d'un composant SlimComponent classique, le processus de dérivation consiste à surcharger le constructeur de façon à nommer le composant, déclarer les ports qu'il fournit et qu'il utilise, puis instancier et initialiser le code de la logique interne. Cependant, la surcharge du constructeur doit inclure, en plus, un appel à la méthode `setInternalLogicControl()` afin

d'initialiser l'attribut `internalLogicControl` de manière adéquate.

Les composants contrôlables sont destinés à être exploités par l'intermédiaire d'une fabrique. Le cadre de développement EthyleneFramwork capitalise l'ensemble des mécanismes propres au concept de fabrique décrit par le modèle Ethylene au sein d'une fabrique abstraite que je détaille au point suivant.

2.2.3 Comment implémenter une fabrique ?

Selon le modèle Ethylene, une fabrique a pour fonction la gestion du cycle de vie des composants et des extrémités de connecteur qui lui sont associées. Une fabrique assure ainsi la disponibilité dynamique de ses composants et, sur la requête d'un tiers, leur instanciation, le démarrage des instances qui en résultent, ainsi que leurs arrêt et destruction. Une fabrique est également chargée de l'établissement des liaisons entre composants. Dans cette phase, elle sous-tend la négociation du type d'extrémités de connecteur à utiliser avant d'assurer leur mise en œuvre. Le modèle Ethylene détermine :

(1) les diagrammes d'activité qui spécifient les processus d'établissement des liaisons, d'instanciation des composants et de destruction de ces derniers,

(2) des interfaces programmatiques, réparties en trois types de ports, qui spécifient le cadre dans lequel les fabriques interagissent avec des tiers.

Les fonctions des diagrammes d'activité du modèle Ethylene se classent en deux catégories :

(1) les fonctions indépendantes du type de composants et du type d'extrémités de connecteur associés à la fabrique,

(2) les fonctions dépendantes du type de composants et du type d'extrémités de connecteur associés à la fabrique.

Dans EthyleneFramework, l'implémentation des fonctions indépendantes est capitalisée au sein d'une fabrique abstraite dans laquelle les fonctions dépendantes apparaissent sous la forme de méthodes abstraites. Cette fabrique abstraite permet la production de fabriques concrètes par dérivation. Ainsi, pour le programmeur, implémenter une fabrique concrète se résume à l'implémentation des méthodes abstraites en fonction des composants et des extrémités de connecteur qu'il souhaite associer à la fabrique. En outre, selon modèle Ethylene, une fabrique interagit avec des tiers par l'intermédiaire de ports, eux-mêmes définis par des interfaces programmatiques. En ce sens, la fabrique se rapproche de la notion de composant. Ainsi, dans EthyleneFramework, le code

d'une fabrique concrète est encapsulé au sein d'un composant SlimComponent.

La classe `AbstractFactory`

Dans EthyleneFramework, la fabrique abstraite est incarnée par la classe `AbstractFactory`. Cette classe implémente donc la partie de la logique interne d'une fabrique qui est indépendante de tout composant et de toute extrémité de connecteur. D'autre part la classe `AbstractFactory` explicite la logique interne dépendante des composants et extrémités de connecteur par la déclaration de quatre méthodes abstraites (voir Figure VI-12).

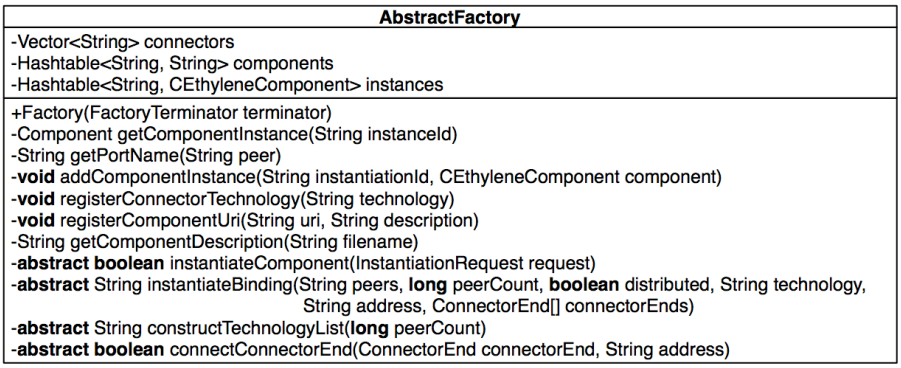


Figure VI-12 Vue partielle des méthodes et attributs de la fabrique abstraite.

La classe `AbstractFactory` offre au développeur un ensemble de méthodes protégées destinées à faciliter l'implémentation des fabriques concrètes. À l'instar de la Figure VI-12, je focalise ma description de la classe `AbstractFactory` sur les principes et les éléments utiles au programmeur de fabriques concrètes, sans m'attarder sur les éléments liés plus généralement à la mécanique du concept de fabrique, déjà largement décrit à la section 3.3 du chapitre précédent.

La logique interne d'une fabrique s'articule autour de trois ensembles d'entités : un ensemble des composants qu'elle est en mesure d'instancier, un ensemble des extrémités de connecteur qu'elle est capable de mettre en œuvre et enfin, d'un ensemble des instances de composants dont elle est responsable. La classe `AbstractFactory` offre la possibilité de gérer ces trois ensembles de manière générique quelles que soient les classes concrètes sous-jacentes de composants et d'extrémités de connecteur. Ainsi, l'ensemble des extrémités de connecteur s'incarne dans le vecteur `connectors` où celles-ci sont représentées par l'identifiant de la technologie de communication qu'elle met en œuvre. Pour peupler ce vecteur, le développeur de fabrique concrète a à sa disposition la méthode protégée `registerConnectorTechnology()` qui accepte comme paramètre un identifiant de technologie de

communication et qui doit être appelée pour chaque extrémité de connecteur à ajouter.

L'ensemble des composants instanciables s'incarne dans la table de hachage `components`. Dans cette table, les clés sont les URI des composants et les valeurs contiennent la description EthyleneXML complète de chaque composant. La méthode protégée `getComponentDescription()` permet au développeur de fabrique concrète de charger à partir de différents fichiers les descriptions EthyleneXML de chaque composant. La table de hachage `components` doit être peuplée au moyen de la méthode `registerComponentUri()`. Enfin, l'ensemble des instances de composants s'incarne dans la table de hachage `instances`. Dans cette table, les clés sont des chaînes de caractères générées de manière unique par la fabrique. Chacune de ces clés identifie une référence sur une instance de composant. Lors de l'instanciation d'un composant, le développeur de fabrique concrète doit utiliser la méthode protégée `addComponentInstance()` pour ajouter la nouvelle instance à la table de hachage `instances`.

L'ensemble de ces méthodes protégées a pour objectif de faciliter la tâche du développeur de fabriques concrètes lors du processus de dérivation de la classe `AbstractFactory`.

Comment dériver `AbstractFactory` en une fabrique concrète ?

La logique interne d'une fabrique concrète est obtenue par dérivation de la classe `AbstractFactory`. Ce processus de dérivation consiste à surcharger le constructeur de la classe `AbstractFactory` et à implémenter les quatre méthodes abstraites qu'elle définit.

La surcharge du constructeur est succincte. Le développeur procède en trois étapes :

(1) le constructeur de la classe `AbstractFactory` accepte un paramètre `terminator`. Ce paramètre d'appel doit être conservé dans la signature du constructeur de la classe dérivée et doit être transmis au constructeur de la classe mère. Il s'agit d'une référence sur l'objet qui sera chargé de l'arrêt de la fabrique.

(2) Puis, le constructeur déclare, par des appels successifs à la méthode protégée `registerConnectorTechnology()`, les différentes technologies mises en œuvre par les extrémités de connecteur associées à la fabrique.

(3) Enfin, pour chaque composant associé à la fabrique, le constructeur déclare l'URI du composant et sa description EthyleneXML par un appel à `registerComponentUri()`.

Une fois le constructeur surchargé, il reste au développeur à implémenter les méthodes abstraites `instantiateComponent()`, `constructTechnologyList()`, `instantiateBinding()` et `connectConnectorEnd()`.

```
/**
 * @param request contient les informations nécessaires à l'instanciation.
 * @return vrai si l'instanciation réussie, faux en cas d'échec.
 */
boolean instantiateComponent(InstantiationRequest request);
```

La première méthode est chargée de l'instanciation des composants. Elle accepte un paramètre `request` qui contient, notamment, l'URI du composant à instancier. Cette URI permet de déterminer à partir de quelle classe dérivée de `CEthyleneComponent` l'instance de composant doit être produite. Une fois l'instance de composant produite, elle doit être associée à la fabrique par un appel à `addComponentInstance()` qui lui attribuera un identifiant d'instance et achèvera le processus d'installation de cette nouvelle instance de composant. Si l'instanciation réussit, cette méthode doit retourner « vrai ». En cas d'échec, elle est tenue de retourner « faux ». Les trois méthodes abstraites restantes ont trait à l'établissement de liaisons entre composants. Je les présente dans l'ordre chronologique de leur appel.

```
/**
 * @param peerCount contient le nombre de participants à la future liaison.
 * @return une liste d'identifiant de technologie de communication.
 */
String constructTechnologyList(long peerCount);
```

La méthode `constructTechnologyList()` n'intervient que dans le cas où l'ensemble des instances de composant à relier est distribué sur plusieurs fabriques. Elle permet de débuter la phase d'élection du type d'extrémité de connecteur à employer pour sous-tendre une liaison donnée. Cette méthode a pour objet de constituer dans chaque fabrique une liste des technologies de communication en mesure d'assurer la liaison en cours d'établissement, à partir de l'ensemble des technologies de communication dont elle dispose. Cette liste doit prendre la forme d'une chaîne de caractères dans laquelle les identifiants de technologie de communication sont séparés par des espaces. Dans l'implémentation actuelle d'EthyleneFramework, cette méthode ne se base que sur le nombre de participants à la liaison pour déterminer lesquelles des technologies de communication disponibles dans la fabrique sont adéquates. Dans une implémentation plus réaliste, il faudrait déterminer ce choix en fonction des propriétés extra-fonctionnelles (par exemple les requis en terme de fiabilité de la liaison ou de bande passante) des ports à relier.

```
/**
 * @param peers contient une liste de paire composant-port à relier ensemble.
 * @param peerCount contient le nombre de participants à la future liaison.
 * @param distributed vrai si l'ensemble des composants à relier est
 *              distribué sur plusieurs fabriques.
 * @param technology contient l'identifiant de la technology à utiliser.
 * @param address contient l'adresse de connexion avec laquelle doivent être
 *              configurer les instances d'extrémité de connecteur
 *              produites.
 * @param connectorEnds permet de retourner à l'appelant les références vers
 *          les instances d'extrémités de conencteur non configurées.
 * @return suivant les valeurs des paramètres d'appel soit :
 *          - une adresse de connection
 *          - un identifiant de technologie de communication
 *          - null
 *          En cas d'échec, un code d'erreur peut également être retourné par
 *          ce biais.
 */
String instanciateBinding(  String peers, long peerCount, boolean distributed
                            String technology, String address,
                            ConnectorEnd[] connectorEnds);
```

La méthode `instantiateBinding()` a pour objet l'instanciation d'extrémités de connecteur et leur configuration en vue de leur future interconnexion. Les nombreux paramètres d'appel qu'elle accepte permettent de déterminer quelle classe d'extrémité de connecteur doit être instanciée, combien d'instances sont à produire et comment configurer les instances obtenues. Pour cela, les deux premiers paramètres, `peers` et `peerCount` précisent les composants à relier, par quels ports ils doivent l'être et combien de participants compte la future liaison. Le paramètre `distributed` indique si l'ensemble des composants à relier est géré par la même fabrique ou s'il est distribué sur plusieurs fabriques. Dans le premier cas, les trois derniers paramètres d'appel sont omis. Il est alors du ressort de l'algorithme de la méthode `instantiateBinding()` de choisir la classe d'extrémités de connecteur à employer, d'instancier le nombre d'extrémités de connecteur nécessaires, de les associer aux ports de composants impliqués dans la liaison et de les configurer de manière à ce qu'elles soient prêtes à être connectées.

Dans le deuxième cas, le paramètre `technology` indique l'identifiant de la technologie de communication à mettre en œuvre. Celui-ci permet de déterminer la classe d'extrémité de connecteur à instancier. Le paramètre `address` représente l'adresse de connexion avec laquelle la ou les extrémités de connecteur à instancier devront être configurées puis connectées. Si cette adresse n'est pas encore déterminée, ce paramètre est omis : il est alors du ressort de l'algorithme de la méthode `instantiateBinding()` de choisir l'adresse de connexion qui devra être retournée afin d'être communiquée aux autres fabriques impliquées dans l'établissement de la liaison. Si l'adresse de connexion ne peut pas être déterminée par la méthode `instantiateBinding()`, le dernier paramètre, `connectorEnds`,

permet de retourner à l'appelant les références sur les instances d'extrémité de connecteur qui n'ont pas pu être configurées totalement. Celles-ci le seront dans une phase ultérieure du processus, au moyen de la méthode `connectConnectorEnd()`.

```
/**
 * @param connectorEnd est une référence sur l'instance d'extrémité de
 *              connecteur à connecter.
 * @param address contient l'adresse de connexion avec laquelle l'instance
 *              d'extrémité de connecteur doit finir d'être configurée, si
 *              nécessaire.
 * @return une liste d'identifiant de technologie de communication.
 */
boolean connectConnectorEnd(ConnectorEnd connectorEnd, String address);
```

La méthode `connectConnectorEnd()` est la dernière que le développeur de fabrique concrète doit implémenter. Son rôle est d'achever, si cela est nécessaire, la configuration de l'instance d'extrémité de connecteur qui lui est passée en paramètre puis d'initialiser la connexion. La configuration de l'instance de d'extrémité de connecteur doit être réalisée avec l'adresse de connexion spécifiée par le second paramètre. L'initialisation de la connexion est ensuite réalisée par l'intermédiaire de la méthode `connect()` de l'instance d'extrémité de connecteur.

Ce processus de dérivation de la classe `AbstractFactory` permet de produire la logique interne d'une fabrique concrète. Pour être utilisable, cette logique interne doit être encapsulée sous la forme d'un composant SlimComponent. Le dernier point de cette section décrit cette ultime étape.

Comment encapsuler une fabrique concrète

La classe `CFactory` (voir Figure VI-13) a pour vocation de servir d'encapsulation pour toutes les logiques internes de fabriques concrètes dérivées de la classe `AbstractFactory`.

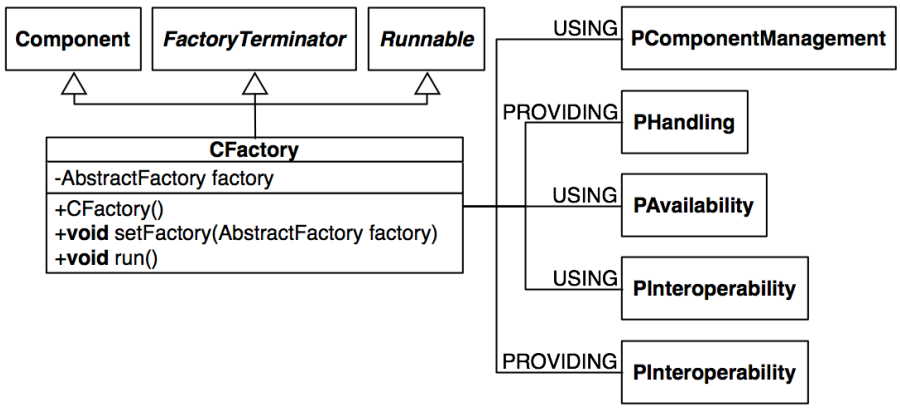


Figure VI-13 Spécification de l'encapsulation d'une fabrique concrète.

La classe `CFactory` est une spécialisation de la classe `Component` qui déclare par défaut les cinq ports nécessaires au bon fonctionnement d'une fabrique. Les quatre premiers ports, `PHandling`, `PAvailability`, `PInteroperability` dans sa version « fourni » et `PInteroperability` dans sa version « utilisé », permettent l'interaction logicielle de la fabrique avec ses partenaires, conformément à la spécification du modèle Ethylene (voir la section 3.3 du chapitre précédent). Le dernier port, `PComponentManagement`, permet l'interaction logicielle entre la logique interne de la fabrique et les instances de composant dont elle a la responsabilité, en sorte que le cycle de vie de ces instances soit contrôlé par la fabrique.

En outre, la classe `CFactory` implémente les interfaces `FactoryTerminator` et `Runnable`. La première interface permet à la logique interne, le moment voulu, d'être en mesure d'indiquer à l'instance de `CFactory` qui l'encapsule son intention de s'arrêter. L'instance de `CFactory` est alors chargée de gérer l'arrêt et la destruction de la fabrique. L'interface `Runnable` permet de placer l'exécution de la fabrique dans un processus léger. Le programme principal peut donc attendre la terminaison de l'exécution de la fabrique en se mettant en attente de la terminaison du processus léger. Enfin, la classe `CFactory` fournit une méthode publique, `setFactory()`, qui accepte une référence vers une sous-classe de `AbstractFactory` en paramètre, que le développeur doit utiliser pour associer la logique interne de la fabrique avec son encapsulation.

En résumé, l'encapsulation de la logique interne d'une fabrique au sein d'une instance de classe `CFactory` se réalise en trois étapes :

(1) instanciation d'une instance de `CFactory`,

(2) instanciation de la logique interne et

(3) association de la logique interne et de l'instance de `CFactory` au moyen de la méthode `setFactory()`.

L'extrait de code suivant illustre ces trois étapes sous la forme d'un programme principal qui démarre une fabrique, puis se met en attente de la fin de son exécution :

```
public static void main(String[] args) throws Exception {

     CFactory cFactory = new CFactory();
     cFactory.setFactory(new SampleFactory(cFactory));
     cFactory.getThread().join();
}
```

2.2.4 En résumé

Le cadre de développement EthyleneFramework s'appuie sur la technologie à composants SlimComponent. Il est constitué d'un ensemble de classes et d'interfaces destinées, d'une part à l'implémentation de composants dont le cycle de vie est conforme au modèle Ethylene et, d'autre part à l'implémentation de fabriques pour les rendre disponibles et assemblables dynamiquement. Dans les deux cas, les mécanismes généraux sont capitalisés au sein de classes abstraites que le développeur doit exploiter par dérivation. Le développeur limite donc son effort à l'implémentation des mécanismes fortement liés aux composants et extrémités de connecteur qu'il souhaite mettre en œuvre. Dans la version actuelle de ces mécanismes, une fabrique est capable de mettre en œuvre des composants directement exécutables, tout en laissant de côté la gestion de la partie du cycle de vie relative aux composants transformables. J'apporte des précisions sur ce point en fin de chapitre.

L'implémentation actuelle d'EthyleneFramework est dédiée à la technologie SlimComponent, qu'il augmente des éléments nécessaires pour la rendre conforme au modèle Ethylene. Cependant il constitue également un exemple d'application du modèle Ethylene sur lequel pourrait s'appuyer un programmeur pour implémenter un autre cadre de développement adapté à une autre technologie à composants, comme Fractal ou Wcomp, afin de rendre celles-ci conformes au modèle Ethylene et utilisables dans le contexte de la plasticité des systèmes interactifs. SlimComponent associé à EthyleneFramework constitue une mécanique de base pour implémenter des systèmes interactifs plastiques. Pour aller plus loin, il est nécessaire de disposer de certaines des fonctions introduites par la décomposition fonctionnelle présentée à la section 2 du chapitre précédent. L'infrastructure MARI, développée dans l'équipe IIHM, implémente la plupart de ces fonctions. La section suivante présente une vue d'ensemble de MARI puis focalise sur la manière dont l'approche à composants Ethylene a pu y être intégrée.

3. Infrastructure MARI et Ethylene

MARI (Model At Runtime Infrastructure) [Sottet07] est une infrastructure logicielle pour la plasticité issue des travaux de thèse de Jean-Sébastien Sottet. Cette infrastructure vise à atténuer le cloisonnement entre la phase de conception d'un système interactif et sa phase d'exécution. Pour cela, MARI s'appuie sur des techniques issues de l'ingénierie dirigée par les modèles (IDM) : les systèmes interactifs et le contexte de l'interaction existent sous la forme de graphes de modèles conformes à des

métamodèles et liés entre eux par des relations, elles-mêmes explicitées sous la forme de modèles. Ensemble, ces modèles forment un « écosystème » [Sottet07-2]. Dans cette approche, les interfaces utilisateurs sont entièrement générées à la volée par un processus automatique et leur adaptation au contexte de l'interaction est obtenue par des opérations de transformation sur ces modèles.

3.1 Architecture logicielle de MARI

L'implémentation du métier de MARI repose sur cinq services OSGi (voir Figure VI-14). Un gestionnaire du modèle de l'écosystème (Ecosystem Model Manager) regroupe et maintient l'ensemble des instances de modèles qui définissent le système interactif en cours d'exécution, son état et le contexte de l'interaction. Dans ma décomposition fonctionnelle présentée au chapitre V, ce gestionnaire correspond à la fonction « synthétiseur de situation ». Toutes ces instances de modèle hébergées par le gestionnaire de l'écosystème sont conformes à des métamodèles dont les descriptions sont embarquées au sein du gestionnaire de métamodèles (Metamodels Manager).

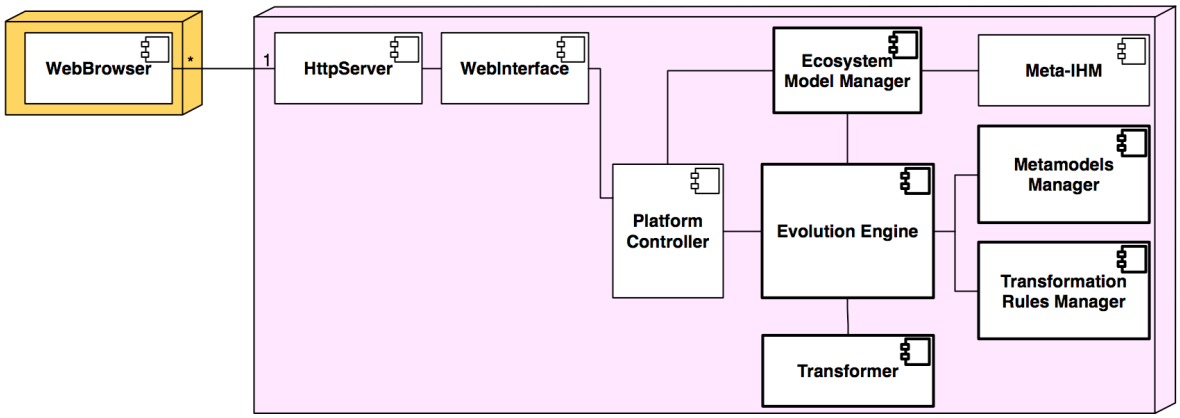


Figure VI-14 Architecture initiale de l'infrastructure MARI. Les services dont l'encadrement est épais forment le cœur fonctionnel de MARI. Les boîtes en « trois dimensions » représentent des plates-formes matérielles.

Parallèlement, le gestionnaire de règles de transformation (Transformation Rules Manager) regroupe l'ensemble des règles de transformations envisageables sur les instances de modèles du système interactif en cours d'exécution, en vue de son adaptation. Le moteur d'évolution, en fonction des changements qu'il constate dans le contexte de l'interaction et des métamodèles auxquels se conforment les modèles du système interactif, sélectionne les règles de transformation à appliquer pour adapter convenablement le système interactif au nouveau contexte de l'interaction. Ainsi, l'ensemble constitué du moteur d'évolution de MARI, du gestionnaire de métamodèles et du gestionnaire de règles de transformation se projette dans la fonction « moteur

d'évolution » et « situation matcher » de ma décomposition fonctionnelle du chapitre V (figure V-6). Enfin, le moteur d'évolution de MARI repose sur le transformateur de modèles (Transformer) pour exécuter les règles de transformation sélectionnées et produire l'adaptation. Ce transformateur de modèles assure donc la fonction de producteur de l'adaptation de ma décomposition fonctionnelle du chapitre V.

À ces cinq services métiers s'ajoutent deux services extra-fonctionnels : une méta-IHM et un contrôleur de plates-formes. Le service méta-IHM permet à un utilisateur humain d'observer les instances de modèle hébergées par l'écosystème et d'appliquer des transformations aux modèles du système interactif en vue d'adapter ce dernier à la volée selon ses préférences. Dans l'implémentation actuelle de MARI, cette méta-IHM s'adresse plutôt à un « concepteur de système » qu'à un utilisateur final « grand public ». Elle se présente sous la forme d'une fenêtre à l'intérieur de laquelle les différentes instances de modèles sont représentées par un réseau de graphes (voir Figure VI-15). La méta-IHM offre la possibilité à l'utilisateur de transformer ces instances de modèles par ajout ou suppression de nœud et de relations entre des nœuds. L'adaptation qui en résulte provoque la mise à jour de l'IHM perçue par l'utilisateur final.

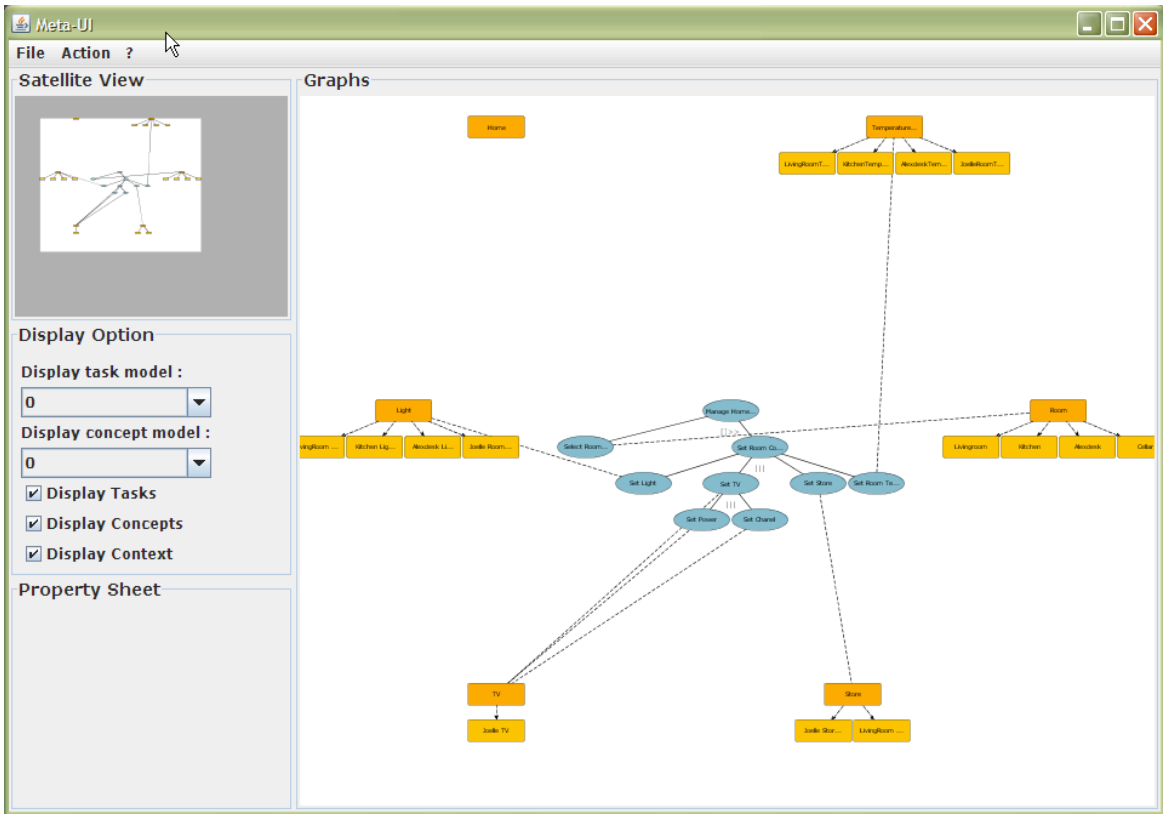


Figure VI-15 Capture d'écran de la méta-IHM offerte par l'infrastructure MARI.

Le contrôleur de plates-formes joue un rôle d'interface entre les services métiers de MARI et les différentes technologies de rendu de l'IHM à l'utilisateur. Dans l'implémentation initiale de MARI, ces technologies de rendu se limitent à des interfaces utilisateur HTML et XUL affichées dans un navigateur Web. Les navigateurs des utilisateurs interagissent avec un serveur HTTP classique, auquel est couplé un service « WebInterface » qui joue le rôle de mandataire entre le serveur HTTP et le contrôleur de plates-formes. Dans un sens « montant », il communique au contrôleur de plates-formes, d'une part l'arrivée ou le départ des navigateurs Web et d'autre part les actions de l'utilisateur. Dans un sens « descendant » où il adapte les données produites par le contrôleur de plates-formes de façon à les rendre disponibles, via le serveur HTTP, aux navigateurs utilisés par l'utilisateur.

Dans son implémentation initiale, MARI permet de mettre en œuvre des systèmes interactifs du type application Web dont l'interface utilisateur est générée automatiquement en fonction du contexte de l'interaction, à partir de modèles à haut niveau d'abstraction. Cependant, si cette solution a l'avantage d'être en mesure de proposer une IHM pour toutes les plates-formes pour lesquelles il existe un navigateur Web et pour tous les contextes de l'interaction pour lesquels il existe des règles de transformation, elle a l'inconvénient d'être seulement capable de proposer des IHM de type formulaire générées automatiquement, en privant l'utilisateur d'IHM précalculées de haute qualité, minutieusement élaborées par des concepteurs humains au talent incomparable. En outre, l'implémentation initiale de MARI n'aborde pas la question de l'interaction logicielle avec le noyau fonctionnel des systèmes interactifs. Un moyen de remédier à ces inconvénients est de permettre à l'infrastructure MARI de manipuler et d'utiliser des adaptateurs de noyau fonctionnel et des éléments d'IHM précalculés encapsulés sous la forme de composants Ethylene.

3.2 Intégration de l'approche Ethylene dans MARI

L'intégration d'Ethylene dans MARI a été effectuée de la manière suivante : le modèle de la plate-forme et la méta-IHM ont été modifiés pour prendre en compte les fabriques et les composants inertes qu'elles déclarent. Puis, sur l'exemple du service « WebInterface », un service « EthyleneInterface » a été développé afin d'assumer le rôle de mandataire entre les fabriques et les composants d'une part, et le contrôleur de plates-formes d'autre part.

3.2.1 Modification du modèle de plate-forme

Dans sa version initiale, le modèle de plate-forme qui ne constitue pas le cœur du travail de Jean-Sebastien Sottet est basique. Une instance de navigateur Web joue le rôle d'une plate-forme élémentaire dont les deux propriétés caractéristiques principales

sont la taille de la fenêtre et la capacité à afficher une interface utilisateur décrite en XUL ou en HTML. La plate-forme d'un utilisateur est constituée de l'ensemble des plates-formes élémentaires dont il a le contrôle (voir Figure VI-16).

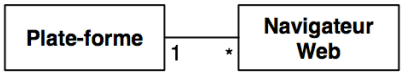


Figure VI-16 Modèle de plate-forme initial.

Dans la méta-IHM, la plate-forme de l'utilisateur est représentée sous la forme d'un arbre dont les feuilles représentent les plates-formes élémentaires. L'utilisateur a la possibilité de créer ou de détruire des relations entre des tâches du modèle de tâches du système interactif et des plates-formes élémentaires, ce qui provoque, au niveau de l'IHM du système interactif, la distribution et le remodelage correspondants. Afin d'intégrer l'approche Ethylene, le modèle de plate-forme a été modifié pour prendre en compte les fabriques et les composants (voir Figure VI-17) :

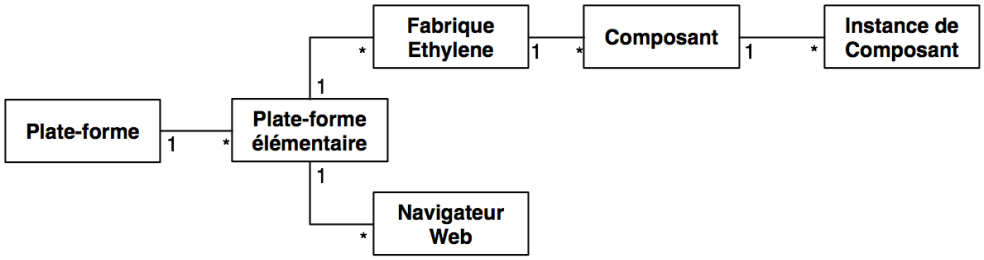


Figure VI-17 Modèle de plate-forme modifié pour intégrer l'approche Ethylene.

Dans le nouveau modèle de plate-forme, la plate-forme d'un utilisateur reste l'ensemble des plates-formes élémentaires dont il a le contrôle. Cependant, le concept de plate-forme élémentaire fait maintenant référence à un dispositif matériel tel qu'il est défini au chapitre II. Une plate-forme élémentaire sous-tend l'exécution, d'une part d'un ensemble d'instances de navigateur Web et d'autre part d'un ensemble de fabriques Ethylene. Chaque fabrique Ethylene est capable de mettre en œuvre un ensemble non vide de composants Ethylene. Pour chacun de ces composants correspond un ensemble d'instances de composants.

Au sein de la méta-IHM, la plate-forme de l'utilisateur est maintenant représentée sous la forme d'un arbre dont les nœuds représentent les plates-formes élémentaires et où un nœud « plate-forme élémentaire » accepte comme fils des feuilles de type « navigateur Web » et des fils de type « fabrique ». Un noeud « fabrique » regroupe des fils du type « composants » qui, eux-

mêmes, regroupe des feuilles de type « instance de composant ». L'utilisateur a la possibilité d'ajouter ou de retirer des feuilles de type « instance de composant », ce qui provoque l'instanciation ou la destruction de l'instance de composant correspondante. Il a également la possibilité de créer ou détruire des relations entre les tâches du modèle de tâches du système interactif et les feuilles de type « navigateur Web » et « instance de composant ». Comme dans la version précédente, les relations entre le modèle de tâche et les navigateurs Web correspondent à un état de distribution de l'IHM du système interactif sur les différentes instances de navigateurs Web. Une relation entre une tâche et une instance de composant indique à MARI que cette tâche est sous-tendue par cette instance de composant : en conséquence, les actions de l'utilisateur sur l'IHM produite par cette instance de composant modifient, dans l'écosystème, les concepts du domaine associés à cette tâche. Inversement, les changements de valeurs de ces concepts, dans l'écosystème, doivent provoquer la mise à jour de l'IHM produite par cette instance de composant. Pour cela, l'infrastructure MARI interagit avec les instances de composant par l'intermédiaire du contrôleur de plate-forme. Pour atteindre les instances de composant, celui-ci passe par un mandataire incarné par le service « EthyleneInterface ».

3.2.2 Implémentation d'« EthyleneInterface »

Par son rôle de mandataire, « EthyleneInterface » assume trois fonctions: la mise à jour du modèle de plate-forme, la manipulation des instances de composant par MARI et le maintien de la cohérence entre les instances des modèles de tâches et de concepts présents dans l'écosystème et leurs pendants embarqués dans chaque instance de composant.

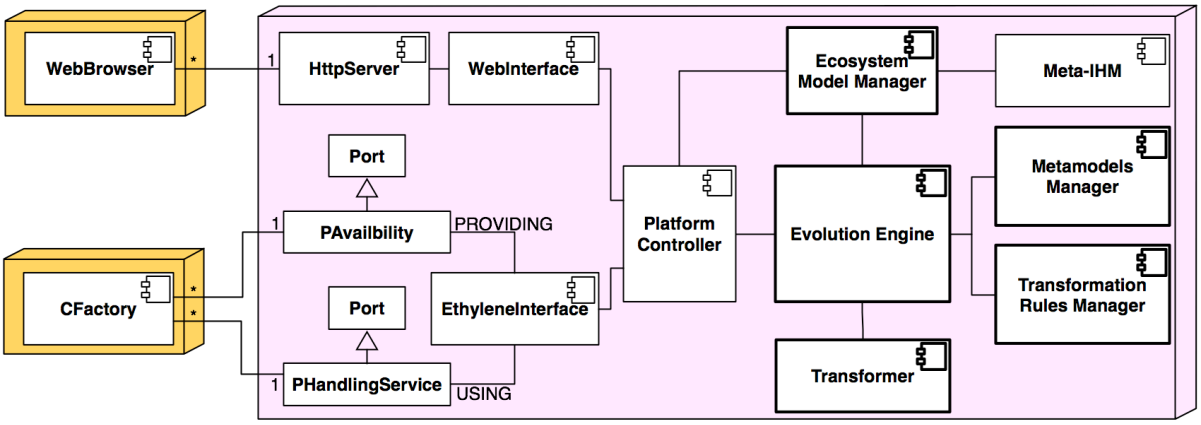


Figure VI-18 Architecture modifiée de l'infrastructure MARI qui intègre une interface d'accès aux technologies à composants conformes au modèle Ethylene.

Concernant la première fonction, « EthyleneInterface » est chargé de signaler à l'infrastructure MARI l'apparition ou la disparition

de fabriques et de composants sur la plate-forme. De ce point de vue, « EthyleneInterface » se comporte comme un annuaire de composants Ethylene. À ce titre, comme le montre la Figure VI-18, il fournit un port `PAvailability` afin d'être informé par les fabriques de leur arrivée et de leur départ, ainsi que des composants dont elles ont la responsabilité. Les informations en provenance des fabriques sont adaptées pour être transmises de façon adéquate au contrôleur de plates-formes.

Pour la deuxième fonction, « EthyleneInterface » est chargé de faire appliquer, au niveau des fabriques, les décisions d'instanciation et de destruction de composants prises par l'infrastructure MARI. De ce point de vue, « EthyleneInterface » se comporte comme un client de l'API Handling offerte par les fabriques. À ce titre, il utilise un port `PHandlingService` pour interagir avec les fabriques présentes sur la plate-forme de l'utilisateur.

Pour la troisième fonction, « EthyleneInterface » est chargé de sous-tendre l'interaction logicielle entre les instances de composants et le gestionnaire du modèle de l'écosystème de l'infrastructure MARI, afin de préserver la cohérence entre les instances des modèles de tâches et de concepts de l'écosystème et leur incarnation dans les instances de composants Ethylene qui participent au système interactif. Par son principe de fonctionnement, l'infrastructure MARI se comporte comme le contrôleur de dialogue des systèmes interactifs : d'une part, l'écosystème centralise les informations relatives à l'enchaînement des tâches ainsi que les valeurs prises par les concepts du domaine et, d'autre part, les IHM sont mises à jours en fonction de l'évolution des informations qu'il centralise.

En conséquence, les composants logiciels qui embarquent des adaptateurs de noyau fonctionnel (ANF) ou des IHM précalculées doivent s'accommoder de ce principe de fonctionnement pour participer à des systèmes interactifs accompagnés de l'infrastructure MARI. Ainsi, un composant ANF doit être capable de signaler au gestionnaire de l'écosystème les nouvelles valeurs des concepts du domaine qu'il modifie. Inversement, un composant ANF doit être capable de mettre à jour son modèle de données en fonction des modifications que lui répercute le gestionnaire de l'écosystème. De la même façon, un composant IHM doit signaler au gestionnaire de l'écosystème les actions de l'utilisateur et les concepts du domaine qu'il modifie. Inversement, un composant IHM doit être en mesure de mettre à jour l'interface utilisateur qu'il sous-tend en fonction des informations transmises par le gestionnaire de l'écosystème.

Pour mettre en œuvre l'interaction logicielle entre le gestionnaire de l'écosystème et les composant ANF et IHM, ma proposition

s'appuie sur le principe de fonctionnement suivant : « EthyleneInterface » interagit avec les composants ANF et IHM par le biais de composants mandataires. Les interactions logicielles entre ces composants mandataires et les composants ANF et IHM s'appuient sur deux ports standard, `PDialogController` et `PFunctionalCoreAdapter`, dont les spécifications complètes sont fournies en annexe. Ces deux ports sont indépendants de l'interface utilisateur produite par un composant IHM ou du noyau fonctionnel adapté par un composant ANF. Ainsi un composant mandataire IHM a la capacité d'interagir avec n'importe quel composant IHM et un composant mandataire ANF a la capacité d'interagir avec n'importe quel composant ANF. Ce principe de fonctionnement donne lieu, pour « EthyleneInterface », à l'architecture présentée en Figure VI-19.

L'architecture du service « EthyleneInterface » distingue trois fonctions : un gestionnaire de composants, un configurateur et une fabrique de composants mandataires. Le gestionnaire de composants joue le rôle d'annuaire. C'est donc lui qui offre le port `PAvailability` d'« EthyleneInterface » afin de recevoir les informations relatives aux apparitions et disparition de fabriques et de composants. Il adapte alors ces informations de manière adéquate et les transmet, via le contrôleur de plate-forme, au gestionnaire de l'écosystème. En ce sens, ce gestionnaire de composant incarne la fonction « gestionnaire de composant » de ma décomposition fonctionnelle du chapitre V. En outre, ce gestionnaire de composants maintient une référence sur chaque instance de composant mandataire. Ainsi il est également responsable de la transmission au gestionnaire de l'écosystème des événements en provenance des instances de composants IHM et ANF et, inversement, de la transmission des modifications dans l'écosystème aux instances de composants. Si le gestionnaire de composants assure le dialogue entre les instances de composants et le gestionnaire de l'écosystème, l'instanciation des composants et l'établissement des liaisons est du ressort du configurateur.

Le configurateur est en charge d'instancier les composants sélectionnés par l'infrastructure MARI et d'établir les liaisons adéquates entre les composants mandataires et les composants IHM et ANF. C'est donc ce configurateur qui utilise le port `PHandlingService` d'« EthyleneInterface » afin d'interagir avec les fabriques présentes sur la plate-forme en fonction des indications fournies par l'infrastructure MARI. En ce sens, il couvre la fonction « configurateur » de ma décomposition fonctionnelle présentée au chapitre V. Ce configurateur est implémenté pour constituer trois types d'assemblage.

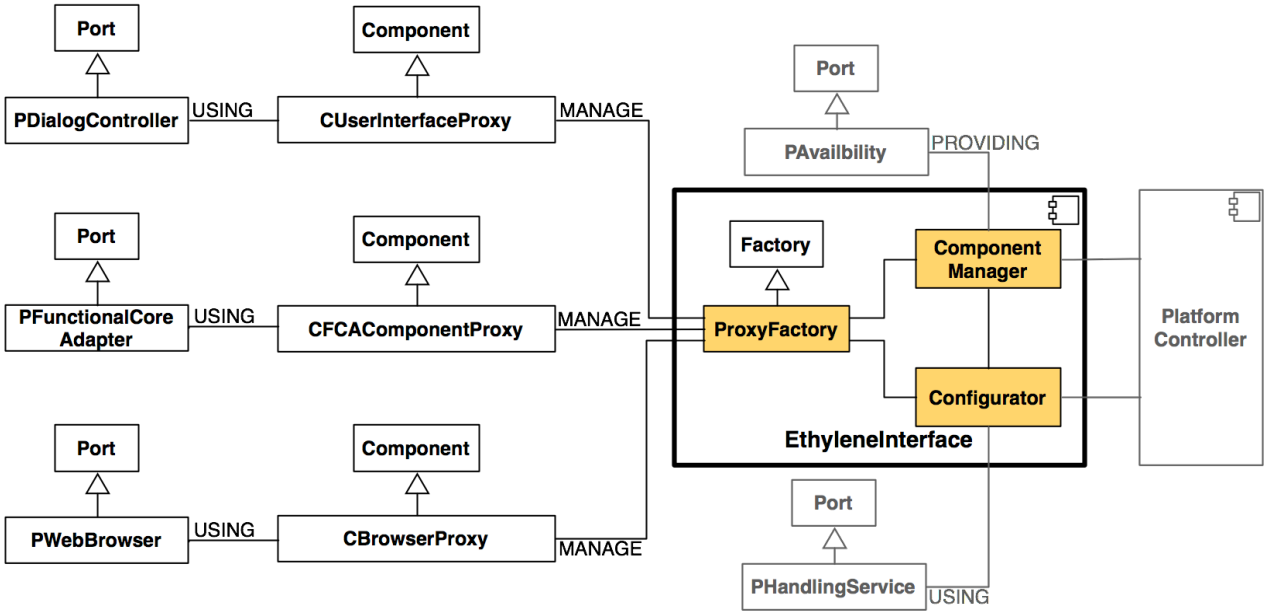


Figure VI-19 Détail du service « EthyleneInterface ». Ce service se décompose en trois fonctions : le gestionnaire de composant chargé d'assumer le rôle d'annuaire de composant, le configurateur chargé de la manipulation des instances de composants et une fabrique de composants mandataires destinés à l'interaction logicielle avec les composants IHM et les composants ANF.

Le premier type d'assemblage est le plus simple : le composant IHM à instancier fournit un port `PDialogController` ; son instance peut donc être reliée directement à l'instance de composant mandataire IHM. De la même manière, le composant ANF à instancier fournit un port `PFunctionalCoreAdapter`; son instance peut donc être reliée directement à l'instance de composant mandataire ANF. Le deuxième type d'assemblage fait intervenir un tiers entre l'instance de composant IHM ou ANF et le composant mandataire IHM ou ANF. Si le composant IHM, respectivement ANF, ne fournit pas de port `PDialogController`, respectivement `PFunctionalCoreAdapter`, alors le configurateur cherche à instancier un composant adaptateur, pour être relié d'une part au composant IHM, respectivement ANF, et d'autre part au composant mandataire correspondant (voir Figure VI-20). En outre, ce composant adaptateur doit avoir pour objet de convertir les entrées et les sorties sur le port `PDialogController`, respectivement `PFunctionalCoreAdapter`, en entrées et sorties adéquates pour le composant IHM, respectivement ANF.

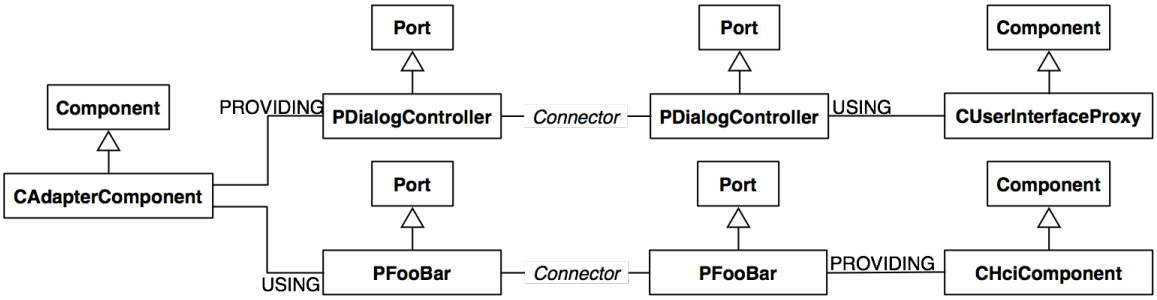


Figure VI-20 Assemblage d'un composant IHM (CHciComponent) avec l'infrastructure MARI représentée par un composant mandataire IHM (CUserInterfaceProxy) par l'intermédiaire d'un composant adaptateur (CAdapterComponent).

Enfin, le dernier type d'assemblage que le configurateur est en mesure d'établir concerne les composants IHM qui s'appuient sur un tiers pour rendre perceptible leur interface. Par exemple, un composant de type « interface Web » a besoin d'un navigateur Web pour afficher son IHM. Dans ce cas, le configurateur est en mesure, d'une part d'instancier le composant « interface Web » et de le relier à un composant mandataire IHM, et d'autre part d'instancier un composant « navigateur Web » qu'il relie à un composant mandataire « navigateur ». Puis il transmet par l'intermédiaire du composant mandataire « navigateur » l'URL pointant sur « interface Web » (voir Figure VI-21).

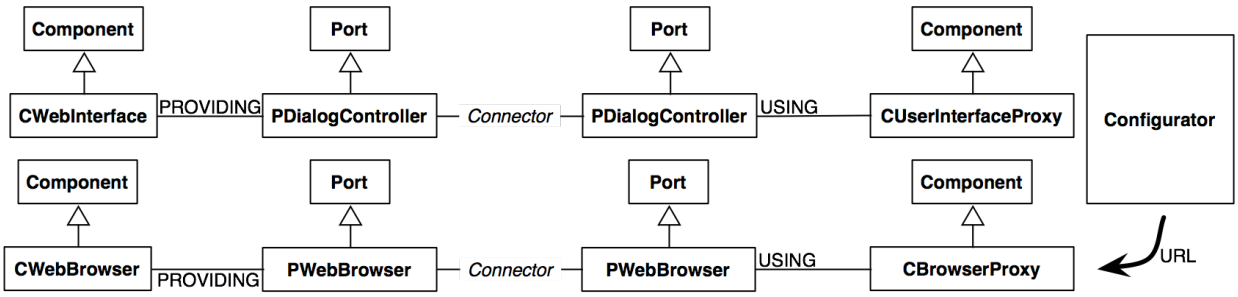


Figure VI-21 Assemblage d'un composant Web (CWebInterface), d'un composant navigateur web (CWebBrowser) et passage de l'URL de l'IHM de CWebInterface au navigateur.

Pour instancier les composants IHM, ANF et « navigateur », le configurateur utilise les fabriques disponibles sur la plate-forme de l'utilisateur. Pour instancier les composants mandataires, le configurateur utilise la troisième fonction du service « EthyleneInterface » qui s'incarne sous la forme d'une « fabrique de composants mandataires ». Ainsi cette fabrique est en mesure d'instancier les trois composants mandataires `CUserInterfaceProxy`, `CFunctionnalCoreAdapter` et `CBrowserProxy`.

3.2.3 En résumé

MARI est une infrastructure logicielle pour la plasticité des systèmes interactifs développée dans l'équipe IIHM dans le cadre des travaux de thèse de Jean-Sébastien Sottet. Afin de réduire le cloisonnement entre la phase de conception et la phase d'exécution des systèmes interactifs, l'infrastructure s'appuie sur les principes de l'ingénierie dirigée par les modèles : les systèmes interactifs et le contexte de l'interaction sont modélisés par un graphe de modèles qui peut être transformé à la volée, au cours de l'exécution, provoquant l'adaptation de l'IHM du système interactif. La solution initiale de MARI permet de mettre en œuvre des IHM de type formulaire générées automatiquement et affichées par un navigateur Web. Si cette solution permet, pour un système interactif donné, de proposer à l'utilisateur une IHM sur toutes les plates-formes équipées d'un navigateur Web et pour tous les contextes d'interaction pour lesquels il existe des règles de transformation, cette solution ne permet pas d'exploiter des éléments d'IHM précalculée produits par des concepteurs humains et qui s'appuient sur des techniques d'interaction avancées.

Un moyen de palier cette limitation est d'offrir à l'infrastructure MARI la possibilité de manipuler et d'utiliser des éléments d'IHM précalculée encapsulés sous la forme de composants Ethylene. Pour intégrer l'approche Ethylene, le modèle de plate-forme utilisé par MARI a été modifié pour prendre en compte les fabriques et leurs composants. Sur l'exemple du mandataire qui permet à MARI de transmettre aux navigateurs Web les éléments d'IHM générés, un mandataire, baptisé « EthyleneInterface », a été implémenté sous la forme d'un service OSGi afin de donner à MARI la capacité d'interagir avec des composants Ethylene. Ce mandataire s'articule autour de trois fonctions. La fonction « gestionnaire de composants » offre un port `PAvailability` qui permet à ce gestionnaire d'être informé de l'arrivée et du départ des fabriques et de leurs composants sur la plate-forme. Il présente ces informations de manière adéquate à l'infrastructure MARI qui les utilise pour mettre à jour l'instance du modèle de plate-forme. Ce gestionnaire gère également un ensemble d'instances de composants mandataires. Ces instances de composants mandataires sont chargées de l'interaction logicielle avec les composants Ethylene qui encapsulent des éléments de l'IHM ou de l'adaptateur du noyau fonctionnel (ANF). Les assemblages entre les instances de composants mandataires et les instances de composants IHM et ANF sont mis en place par la fonction « configurateur » d'« EthyleneInterface ». Pour cela, cette fonction utilise un port `PHandlingService` qui lui permet d'interagir avec les fabriques présentes sur la plate-forme. L'une de ces fabriques incarne la fonction « fabrique de composants mandataires » d'« EthyleneInterface » dont le rôle est d'instancier les composants mandataires nécessaires à l'interaction logicielle

entre l'infrastructure MARI et les instances de composants IHM et ANF.

L'infrastructure MARI ainsi modifiée a permis la réalisation du prototype du système interactif plastique « PhotoBrowser » que je présente dans la section suivante.

4. PhotoBrowser : un système interactif plastique de consultation de photographies

Le démonstrateur PhotoBrowser est le premier système interactif plastique implémenté par un ensemble de composants Ethylene et dont l'adaptation est sous-tendu par l'infrastructure MARI. Le système interactif PhotoBrowser est défini à haut niveau d'abstraction par un modèle de tâches et de concepts hébergé au sein de l'écosystème géré par MARI. Par ailleurs, trois éléments d'IHM précalculée qui implémentent tout ou partie de ce modèle de tâches ont été encapsulés sous la forme de composants Ethylene : le premier est une application écrite en TCL qui exploite les boîte à outils GML-Canevas [Bérard06], le deuxième est une application Web écrite en PHP et Ajax qui nécessite un navigateur Web pour être affichée et le troisième est une simple classe Java qui exploite la boîte à outils Java SWING. En outre, un composant Ethylene encapsule le noyau fonctionnel de PhotoBrowser et un autre encapsule Safari, le navigateur Web du Mac. Cette section présente le modèle de tâche de PhotoBrowser et les composants qui le mettent en œuvre.

4.1 Le modèle de tâche PhotoBrowser

La première étape du développement de PhotoBrowser a été de spécifier son modèle de tâches et son modèle des concepts. Le modèle de tâches et de concepts de PhotoBrowser, volontairement simplifié à l'extrême, comporte huit tâches et trois concepts (voir Figure VI-22). La tâche racine permet à l'utilisateur de « Parcourir les images ». Elle se décompose en deux sous-tâches entrelacées : l'une permet de « consulter toutes les images » de la collection à la fois, tandis que l'autre permet de « consulter une image en détail » de manière itérative. Cette deuxième sous-tâche se décompose en deux sous-tâches séquentielles. En premier lieu, l'utilisateur doit « sélectionner une image » à consulter, puis s'offre à lui la possibilité de « consulter les détails de l'image » sélectionnée. Cette tâche se décompose à son tour en trois sous-tâches entrelacées, toutes optionnelles : l'utilisateur a la possibilité de « consulter les informations à propos de l'image », « consulter l'image » en elle-même et « consulter le nom de l'image ».

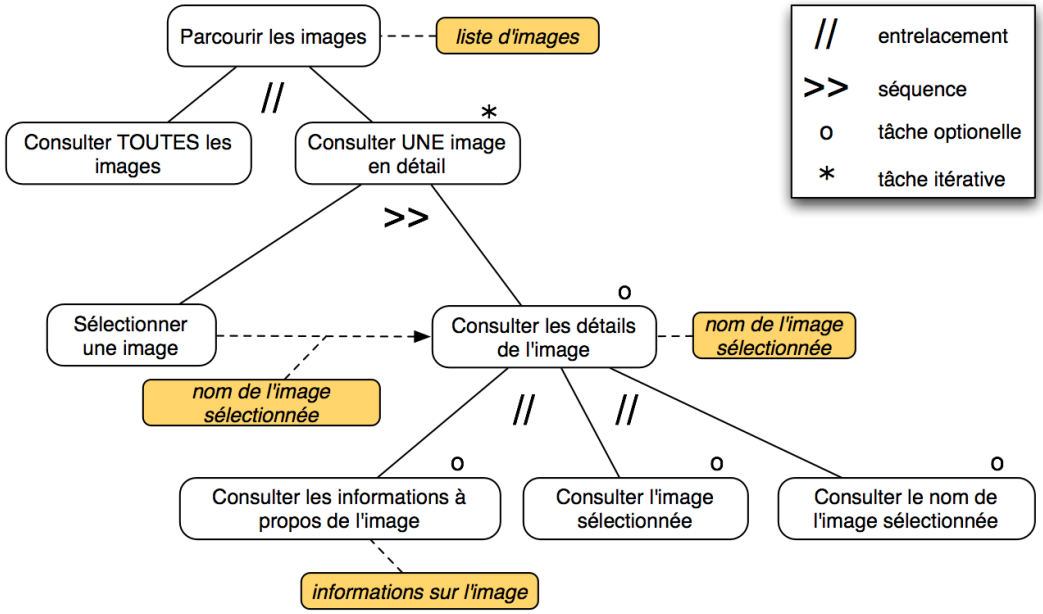


Figure VI-22 Le modèle de tâche de PhotoBrowser.

Le modèle de concepts du domaine associé au modèle de tâches de PhotoBrowser comporte donc trois concepts : une « liste d'images » constituée des noms des fichiers contenant les images de la collection, un « nom d'image sélectionnée » sous la forme d'un nom de fichier et, pour chaque image de la collection, des « informations sur l'image » sous la forme d'un petit texte descriptif.

Les modèles de tâche et de concepts de PhotoBrowser ont été décrits de manière conforme aux métamodèles de tâches et de concepts utilisés par MARI et ajoutés à l'écosystème. L'étape suivante du développement de PhotoBrowser a consisté à encapsuler sous la forme de composants Ethylene des IHM précalculées correspondant à tout ou partie du modèle de tâches de PhotoBrowser, ainsi le noyau fonctionnel de ce système interactif. Les points qui suivent décrivent chacun des composants développés dans le cadre de PhotoBrowser. Leurs spécifications complètes EthyleneXML sont données en annexe.

4.2 Le composant PhotoShuffler

La première IHM précalculée est constituée du système interactif PhotoShuffler (voir Figure VI-23). Ce système interactif a été développé dans l'équipe IIHM par François Bérard dans le cadre du projet ANR DigiTable. Il est programmé en TCL couplé à la boîte à outils GML-Canevas et couvre l'ensemble du modèle de tâches de PhotoBrowser. Ce système interactif peut être contrôlé à distance par l'intermédiaire de deux ports TCP. Le premier est un port de contrôle. L'envoi d'un message « exit» sur ce port provoque l'arrêt de PhotoShuffler. Sur le deuxième port TCP, qui est un port de service, il est possible d'envoyer des messages

textuels qui permettent de sélectionner une image, d'afficher un panneau contenant des informations sur l'image ou de le faire disparaître. Par ailleurs, PhotoShuffler envoie sur ce port TCP le nom de l'image que l'utilisateur sélectionne. Sur le principe, ces deux ports TCP qui permettent de contrôler le dialogue avec l'utilisateur à distance rendent le système PhotoShuffler compatible avec le principe de fonctionnement de MARI. Afin que cela soit possible sur le plan pratique, PhotoShuffler doit être encapsulé sous la forme d'un composant Ethylene.

Figure VI-23 L'IHM du composant PhotoShuffler projetée sur une table DiamondTouch.

Cependant aucune technologie à composants conforme à Ethylene n'existe à ce jour en TCL. En outre, la sémantique des échanges imposés par PhotoShuffler sur ses deux ports TCP ne correspond pas à la logique d'interaction logicielle implémentée par le port `PDialogController` des composants mandataires de MARI. J'ai donc procédé en quatre étapes :

(1) implémenter une extrémité de connecteur qui implémente le protocole imposé par PhotoShuffler, dont l'objectif est d'être directement connectable sur le port de service de PhotoShuffler,

(2) implémenter avec SlimComponent et EthyleneFramework un composant « classique », dont le cycle de vie peut être pris en charge par une fabrique et dont l'objectif est de représenter PhotoShuffler dans les annuaires de composants et de répercuter sur PhotoShuffler les décisions d'instanciation et de destruction,

(3) implémenter la fabrique chargée de la gestion du cycle de vie du composant issu de l'étape précédente,

(4) implémenter un composant adaptateur dont l'objectif est de réaliser l'adaptation de la logique d'interaction logicielle incarnée par le port `PDialogController` dans la logique d'interaction logicielle utilisée par PhotoShuffler.

La première étape aboutit à l'implémentation d'une classe dérivée de `ConnectorEnd` qui implémente le protocole imposé par PhotoShuffler. Ce protocole, qui orchestre la communication entre un client et un serveur sur la base de simples messages textuels, ne pose aucun problème d'implémentation particulier. Je ne détaille donc pas cette implémentation ici. Cette extrémité de connecteur, nommée UnicastTextSocketConnector, sera notamment mise en œuvre par la fabrique du composant produit à l'étape 4, chargée de faire l'intermédiaire entre PhotoShuffler et l'infrastructure MARI.

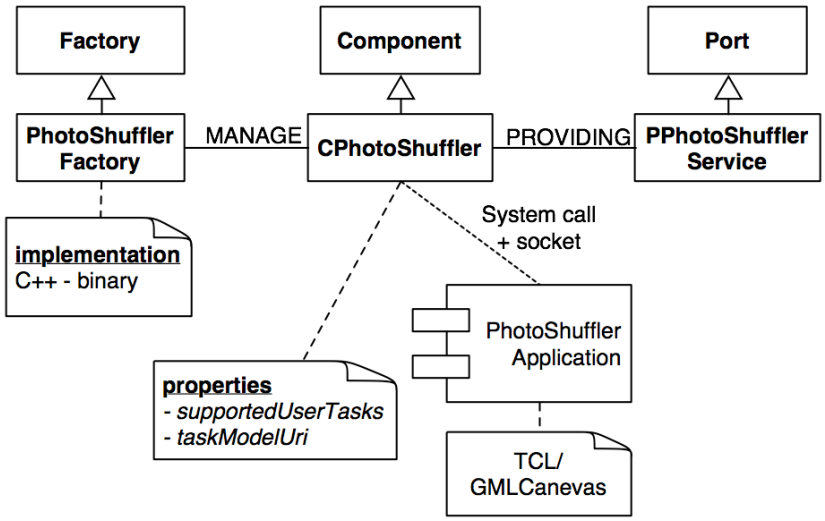


Figure VI-24 Spécification générale du composant PhotoShuffler et de sa fabrique.

La deuxième étape aboutit à l'implémentation du composant `CPhotoShuffler` chargé de représenter PhotoShuffler. Du point de vue de sa spécification, ce composant est tout à fait classique. Il déclare fournir un port `PPhotoShufflerService` (voir Figure VI-24) défini par des interfaces qui représentent sous la forme d'opérations les messages acceptés et envoyés par PhotoShuffler sur son port de service. Par ailleurs, cette spécification associe deux propriétés à `CPhotoShuffler`. La première précise les tâches utilisateurs que PhotoShuffler couvre et la deuxième contient l'identifiant du modèle de tâches indiqué dans la première propriété.

Si la spécification de `CPhotoShuffler` est classique, en revanche, son implémentation est moins traditionnelle. Son constructeur extrait d'un fichier de configuration les numéros des ports de contrôle et de service de PhotoShuffler, puis réalise un appel

système chargé de démarrer ce dernier. La logique interne de `CPhotoShuffler` se limite à l'implémentation de la méthode `stop()` et `destroy()`. Ces méthodes établissent une connexion TCP sur le port de contrôle de PhotoShuffler et lui envoie un message « exit» afin de provoquer son arrêt.

La troisième étape aboutit à l'implémentation de la fabrique `PhotoShufflerFactory`, dédiée à la mise en œuvre du composant `CPhotoShuffler`. Le constructeur de cette fabrique déclare sa capacité à mettre en œuvre l'extrémité de connecteur UnicastTextSocketConnector. Cependant, la méthode `instanciateBinding()` qui est traditionnellement chargée d'instancier et de connecter les extrémités de connecteur est implémentée de manière spécifique pour PhotoShuffler : le principe retenu est de relier l'extrémité de connecteur installée du côté du composant « client » de PhotoShuffler directement au port de service de PhotoShuffler. Ainsi, la méthode `instanciateBinding()` de cette fabrique n'instancie aucune extrémité de connecteur : elle renvoie simplement une notification d'instanciation d'extrémité de connecteur réalisée avec succès, en indiquant dans le champ `address` de cette notification les coordonnées du port de service de PhotoShuffler. Ceci provoque, de manière totalement transparente pour la fabrique du composant « client » de PhotoShuffler, la connexion d'une extrémité de connecteur UnicastTextSocketConnector directement sur le port de service de PhotoShuffler.

À l'issue des ces trois premières étapes, le système PhotoShuffler peut être manipulé à la manière d'un composant Ethylene par l'intermédiaire de la fabrique `PhotoShufflerFactory` et du composant `CPhotoShuffler`. Cependant, celui-ci n'offre qu'un port `PPhotoShufflerService` qui ne peut pas être relié directement au port `PDialogController` d'une instance de composant mandataire IHM de l'infrastructure MARI. Pour cela, il reste à accomplir la dernière étape du processus qui vise à implémenter un composant adaptateur. Cette étape est traitée au point suivant.

4.3 Les composants PhotoShufflerAdapter, PhotoBrowserFC et ImageList

Lorsqu'un composant qui encapsule une IHM ou un ANF n'offre pas de port `PDialogController`, il ne peut pas être relié directement à un composant mandataire de l'infrastructure MARI. Cependant, le configurateur de cette infrastructure est capable de rechercher un composant adaptateur et, s'il le trouve, de l'insérer entre le composant mandataire et le composant IHM ou ANF à relier à MARI. Ainsi, pour relier PhotoShuffler à MARI, un composant adaptateur a été développé. Ce composant (voir Figure VI-25) déclare fournir un port `PDialogController` et utiliser un port `PPhotoShufflerService`.

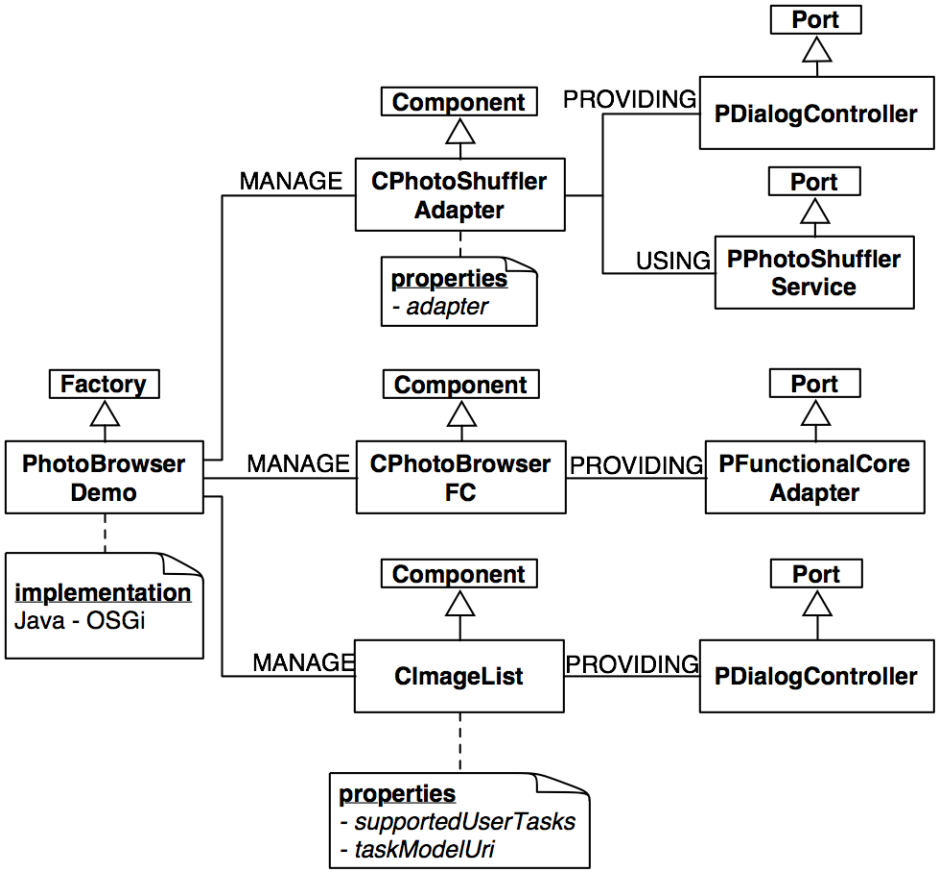


Figure VI-25 Spécification générale des composants ImageList, PhotoShufflerAdapter et PhotoBrowserFC et de leur fabrique.

En outre, une propriété « adapter » permet d'indiquer que ce composant est un composant adaptateur et de préciser l'URI du composant dont il est l'adaptateur. Sa logique interne est chargée de convertir les messages en provenance de l'infrastructure MARI de façon adaptée pour PhotoShuffler et vice-versa. Ce composant adapteur a été développé en Java et embarqué au sein d'une fabrique implémentée sous la forme d'un bundle OSGi.

Cette fabrique embarque deux autres composants relatifs au démonstrateur PhotoBrowser : le composant « noyau fonctionnel » et un composant IHM appelé « ImageList » dont je décris à présent les caractéristiques. Le composant `CPhotoBrowserFC`, qui incarne le noyau fonctionnel de PhotoBrowser, est minimaliste. Il se contente de lire un fichier de configuration contenant la liste des images que PhotoBrowser est capable de parcourir et, pour chacune d'elles, une courte description textuelle. Pour communiquer avec l'infrastructure MARI, il offre un port `PFunctionalCoreAdapter` qui peut être directement connecté à un composant mandataire ANF.

Le composant `CImageList` implémente une IHM simpliste en Java SWING (voir Figure VI-26) destinée à couvrir les tâches

« sélectionner une image » et « consulter le nom de l'image sélectionnée » du modèle de tâche PhotoBrowser.

Figure VI-26 Capture d'écran de l'IHM du composant ImageList.

Ces deux tâches se réalisent par l'intermédiaire d'une liste de noms d'image dont l'utilisateur peut sélectionner un élément à la fois, soit au clavier soit à la souris. Le nom de l'image sélectionnée est mis en évidence par un effet de surbrillance. La spécification de ce composant (voir Figure VI-25) indique qu'il offre un port `PDialogController` qui le rend directement connectable à l'infrastructure MARI. D'autre part, il est défini par deux propriétés, « supportedUserTasks » et « taskModelUri » qui indiquent quelles tâches `CImageList` correspond et à quel modèle de tâches celles-ci se rapportent.

Comme il utilise la boîte à outils Java SWING, le composant `CImageList` est en mesure de rendre perceptible son IHM de manière autonome. Cependant, certains composants peuvent avoir besoin d'un tiers pour rendre perceptible leur IHM. C'est par exemple le cas des « composants Web » dont le point suivant présente un exemple.

4.4 Le composant WebSlideShow

L'application WebSlideShow est une application Web, écrite en PHP et Ajax, conçue pour parcourir une liste d'images. Son interface utilisateur (voir Figure VI-27) permet de visualiser une image courante et de passer à l'image suivante ou précédente au moyen des boutons « Previous» et « Next». Elle permet donc de réaliser les tâches « consulter l'image sélectionnée » et « sélectionner une image » du modèle de tâche PhotoBrowser. Du point de vue de son fonctionnement, WebSlideShow lit la liste d'images à afficher et le nom de l'image sélectionnée à partir d'un fichier « data ». Lorsque l'utilisateur passe à l'image suivante ou précédente, ce fichier est mis à jour.

Figure VI-27 Capture d'écran de l'IHM du composant WebSlideShow.

Pour que WebSlideShow soit exploitable par l'infrastructure MARI, il doit être encapsulé sous la forme d'un composant Ethylene. Comme dans le cas de PhotoShuffler, aucune technologie à composants conforme à Ethylene n'est disponible en PHP ou en Ajax à ce jour. De manière similaire à PhotoShuffler, j'ai procédé, en implémentant, à l'aide d'EthyleneFramework, un composant `CWebSlideShow` chargé de représenter WebSlideShow dans l'annuaire de composants de MARI et d'assurer son interaction logicielle avec l'infrastructure.

La spécification du composant `CWebSlideShow` (voir Figure VI-28) prévoit que celui-ci offre un port `PDialogController`. Il peut donc être assemblé avec un composant mandataire IHM de MARI sans intermédiaire. En outre, la spécification de `CWebSlideShow` comporte quatre propriétés. Les deux premières concernent les tâches utilisateur que WebSlideShow permet et le modèle de tâches auquel elles se rapportent. La troisième, `needRenderer`, est destinée à indiquer au « configurateur » que l'IHM produite par ce composant nécessite un tiers pour être affichée. La valeur de cette propriété précise que ce tiers doit être capable d'interpréter une description HTML et des scripts Ajax.

Enfin, la dernière propriété, `url`, contient l'URL à laquelle se connecter pour obtenir l'IHM.

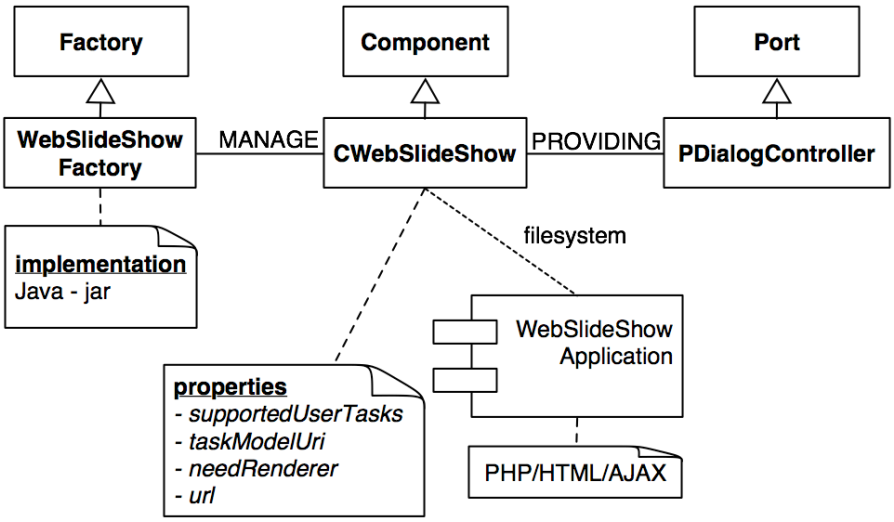


Figure VI-28 Spécification générale du composant WebSlideShow et de sa fabrique.

À l'inverse de PhotoShuffler qui se présente comme une application qu'il faut démarrer pour l'utiliser, WebSlideShow est disponible à tout moment par l'intermédiaire d'un serveur Web. L'implémentation de la logique interne de `CWebSlideShow` se limite donc à mettre œuvre l'interaction logicielle entre WebSlideShow et l'infrastructure MARI. Le principe que j'ai retenu consiste à utiliser le fichier « data » à la façon d'une mémoire partagée. Dans le sens MARI vers WebSlideShow, `CWebSlideShow` modifie le contenu du fichier pour mettre à jour la valeur des concepts. Dans le sens inverse, lorsque `CWebSlideShow` détecte une modification dans le fichier « data », il transmet à MARI la mise à jour effectuée par WebSlideShow.

Enfin, pour que l'IHM de WebSlideShow soit visible par l'utilisateur, il reste à mettre en œuvre un composant « navigateur Web » en lui indiquant à quelle URL il doit se connecter. Le point suivant explique comment Safari, le navigateur de Apple, a été encapsulé sous la forme d'un composant Ethylene.

4.5 Le composant Safari

Comme la plupart des applications du Macintosh, Safari est contrôlable par AppleScript. Cette caractéristique permet d'envisager son encapsulation sous la forme d'un composant Ethylene de la même façon que PhotoShuffler ou WebSlideShow. Ainsi, à l'aide d'EthyleneFramework, j'ai développé, un composant `CSafari` (voir Figure VI-29) qui déclare offrir un port `PWebBrowser`, ce qui lui permet d'être exploité par le « configurateur » de l'infrastructure MARI.

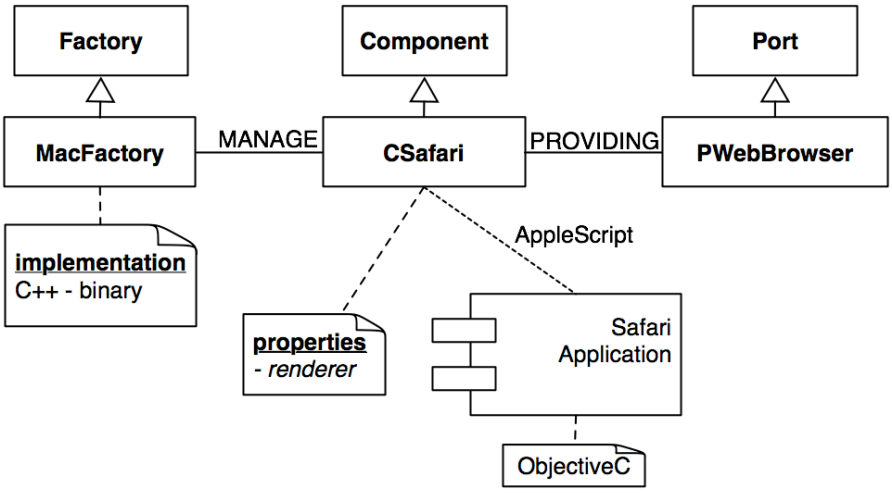


Figure VI-29 Spécification générale du composant Safari et de sa fabrique.

En outre, la spécification de `CSafari` comporte une propriété `renderer` destinée à indiquer au « configurateur » que ce composant permet de rendre perceptible à l'utilisateur l'IHM d'un tiers. La valeur de cette propriété précise le format de la description d'IHM (HTML/Ajax dans le cas de Safari) qu'il est capable d'interpréter. La logique interne de `CSafari` se résume à l'implémentation de trois méthodes : en utilisant AppleScript, la méthode `start()` consiste à démarrer Safari, la méthode `stop()` consiste à provoquer son arrêt et la méthode `goTo()`, introduite par le port `PWebBrowser`, consiste à indiquer à Safari l'URL à laquelle se connecter.

4.6 En résumé

PhotoBrowser est une première mise en œuvre de l'approche Ethylene. À travers ce démonstrateur, il s'agissait de montrer :

(1) comment utiliser l'approche Ethylene et les outils qui lui sont associés (EthyleneXML et EthyleneFramework),

(2) comment des composants issus d'espaces technologiques différents peuvent être rendus dynamiquement disponibles et assemblés grâce à l'approche Ethylene,

(3) comment une approche de l'adapatation dynamique d'IHM basée sur de la transformation de modèles peut, grâce à Ethylene, tirer parti éléments d'IHM précalculée disponibles sur la plate-forme de l'utilisateur (ou ailleurs, dès l'instant où l'on dispose d'un service de recherche de composants ans une base de données répartie).

Ainsi le démonstrateur PhotoBrowser se compose de composants aussi différents qu'ImageList, une simple classe Java qui utilise SWING, PhotoShuffler, un système interactif post-WIMP

développé en TCL et WebSlideShow, une application Web développée en PHP et Ajax qui requiert l'utilisation d'un navigateur Web pour afficher son IHM.

`CImageList` a montré comment encapsuler une entité logicielle dont le code est disponible et dont le langage de programmation dispose du cadre de développement EthyleneFramework. `CWebSlideShow` et `CPhotoShuffler` ont montré deux façons différentes d'encapsuler une entité logicielle dont le code est disponible, mais dont le langage de programmation ne dispose pas d'EthyleneFramework. Enfin, `CSafari` a montré qu'il était également possible, dans une certaine mesure, d'encapsuler sous la forme de composant Ethylene une entité logicielle dont le code source n'est pas disponible.

À ce jour, aucune règle de transformation n'a été ajoutée dans l'infrastructure MARI pour adapter le système interactif PhotoBrowser en fonction du contexte de l'interaction. Ainsi l'adaptation dynamique de PhotoBrowser ne peut être réalisée que par l'utilisateur lui-même au moyen de la méta-IHM de l'infrastructure MARI.

5. Conclusion

Ce chapitre a présenté mes contributions sur le plan technique, qui se répartissent en quatre parties : la première traite plus spécifiquement de l'étape de spécification des composants. Il en est issu EthyleneXML, un langage XML fondé sur le modèle Ethylene dont le but est de permettre l'écriture de spécifications de composants conformes à Ethylene et interprétables par une machine. La deuxième partie s'ancre dans la phase d'implémentation de fabriques et de composants disponibles dynamiquement. Il en est issu un cadre de développement pour Ethylene, disponible en C++ et Java, qui s'appuie sur une technologie à composants minimaliste, SlimComponent, développée à cette occasion. Ce cadre de développement propose, d'une part une façon d'utiliser la notation UML pour représenter les fabriques, les composants, les ports et leurs interfaces et, d'autre part un ensemble de classes et de classes abstraites pour simplifier la tâche du développeur de fabriques et composants disponibles dynamiquement.

La troisième partie de mes contributions techniques aborde la question de l'exécution de systèmes interactifs constitués de composants Ethylene. À ce niveau, mes contributions s'appuient sur l'infrastructure logicielle pour la plasticité MARI, développée dans le cadre des travaux de thèse de Jean-Sébastien Sottet. Cette infrastructure permet l'adaptation dynamique par transformation

de modèles de systèmes interactifs dont les IHM sont automatiquement générées à la volée à partir de modèles de haut niveau d'abstraction. Ma contribution à MARI a été de l'aménager de façon à lui donner la possibilité d'exploiter, en plus de la génération automatique d'IHM, des éléments d'IHM précalculée encapsulés sous la forme de composants Ethylene.

Enfin, la dernière partie de mes contributions techniques a consisté à développer un démonstrateur en utilisant l'ensemble des contributions précédentes. Ce démonstrateur a conduit au développement d'un ensemble de composants logiciels qui s'inscrivent dans des espaces technologiques différents, mais qui, sous l'orchestration de MARI, forment ensemble un système interactif qui peut être adapté dynamiquement en fonction du changement du contexte de l'interaction.

Cependant, l'implémentation actuelle du cadre de développement Ethylene ne permet pas de prendre en charge les composants transformables envisagés au chapitre 5. Une évolution de ce cadre de développement en ce sens est prévue à très court terme. La solution envisagée pour cela se rapproche de ce qui a déjà été fait pour le composant WebSlideShow : ce composant produit une IHM sous la forme de code HTML associé à du Javascript. Pour qu'elle soit perceptible par l'utilisateur, le code de cette IHM doit être interprété par un composant tiers, incarné par Safari dans mon démonstrateur. Le principe serait le même que pour les composants transformables : EthyleneXML permet d'embarquer les modèles de haut niveau d'abstraction qui définissent un composant transformable et de préciser le ou les métamodèles auxquels ils se conforment. Instancier un tel composant passe par une étape de transformation. De la même manière que l'infrastructure MARI recrute un composant Safari pour interpréter l'IHM de WebSlideShow, un ou plusieurs composants transformateurs pourraient être recrutés dynamiquement, en fonction des métamodèles auxquels se conforment les modèles à transformer, pour obtenir l'IHM finale issue de composants transformables.

Chapitre VII

Conclusion

Avant-Propos

Ce chapitre de conclusion s'articule en deux parties. La première conclut ce rapport en rappelant le sujet de cette thèse, les objectifs fixés, et met en rapport les résultats qui en étaient attendus avec les résultats obtenus. La deuxième partie présente quelques perspectives pour ces travaux de recherche doctorale.

1. Conclusion

Cette thèse traite de la question de l'exécution des systèmes interactifs dans le cadre de l'informatique ubiquitaire. Dans ce cadre, la variabilité et l'imprévisibilité du contexte de l'interaction impliquent que les systèmes interactifs doivent s'adapter ou être adaptés à l'exécution afin de préserver leur utilisabilité. J'ai choisi d'aborder cette question sous l'angle du génie logiciel. Le sujet de cette thèse concerne donc l'étude de mécanismes logiciels généraux nécessaires à l'adaptation des interfaces homme-machine à l'exécution. Les objectifs de ces travaux de recherche sont :

(1) de comprendre, d'un point de vue génie logiciel, la problématique de la plasticité des IHM à l'exécution,

puis, pour traiter cette problématique, de proposer des bases générales qui permettent de :

(2) garder la possibilité d'exploiter à l'exécution des modèles de haut niveau d'abstraction pour adapter les IHM sur des fondements sémantiques,

(3) concilier la génération automatique d'IHM conventionnelle avec le codage à la main d'IHM avancée,

(4) construire des systèmes interactifs dynamiquement reconfigurables à partir d'éléments logiciels hétérogènes disponibles dynamiquement.

Pour couvrir ces objectifs, il était attendu des résultats sous la forme d'un cadre conceptuel intégrateur ainsi que, sur le plan technique, la production d'une solution qui incarne les mécanismes relatifs à la plasticité et dont l'objet est de faciliter le développement d'intergiciels destinés à sous-tendre la plasticité des systèmes interactifs.

L'étude de la problématique de la plasticité des IHM à l'exécution, du point de vue du génie logiciel, a débouché sur ma

première contribution sous la forme d'**une taxonomie servant d'espace problème de la plasticité**. Les sept axes de ma taxonomie caractérisent les moyens de l'adaptation, la granularité des éléments d'IHM pouvant être adaptés, la granularité de l'état de reprise de la tâche de l'utilisateur après adaptation, le mode de déploiement de l'IHM, la couverture du contexte de l'interaction et des espaces technologiques et, enfin, la place accordée au contrôle de l'adaptation par l'utilisateur. Cet espace problème, qui répond à l'objectif (1), a permis de classer les solutions de l'état de l'art, d'en identifier les lacunes et finalement de déduire un **cadre conceptuel intégrateur sous la forme d'une décomposition fonctionnelle qui rassemble et organise l'ensemble des fonctions nécessaires à la plasticité** de façon à répondre aux objectifs (2) et (3). Cette décomposition fonctionnelle constitue ma deuxième contribution.

Au sein de ma décomposition fonctionnelle, les fonctions pour la plasticité se partagent, à gros-grain, entre une infrastructure de capture du contexte de l'interaction et un gestionnaire de l'adaptation. À son tour, le gestionnaire de l'adaptation comprend trois fonctions : un identificateur de situation, un moteur d'évolution et un producteur d'adaptation. Cette décomposition fonctionnelle permet l'application d'un effet Slinky : chacune de ses fonctions peut être soit implémentée de façon générique dans un intergiciel, soit de façon spécifique au niveau de la couche applicative. La couche applicative est vue comme un assemblage de composants et de connecteurs. Dans ce cadre, les composants consistent soit, classiquement, en un code directement exécutable, soit en des modèles pour l'IHM de haut niveau d'abstraction transformables à la volée. Dans les deux cas, ces composants sont décrits d'un point de vue fonctionnel et extra-fonctionnel, stockés de façon répartie sur l'ensemble de la plate-forme de l'utilisateur et disponibles dynamiquement. L'adaptation des systèmes interactifs est donc réalisée soit par adaptation interne des composants, soit par transformation dynamique des assemblages de composants.

Je montre que la question de l'interaction logicielle entre le gestionnaire de l'adaptation et la couche applicative, destinée à transformer dynamiquement des assemblages de composants, ainsi que la question de constituer dynamiquement des assemblages de composants hétérogènes, dans le cadre de l'Interaction Homme-Machine, sont des questions ouvertes. Dans le cadre de cette thèse, je focalise donc la suite de mon étude sur les moyens de mettre en œuvre cette interaction logicielle. En réponse à ces deux questions, je propose **Ethylene,** ma troisième contribution, **un modèle à composants dynamiques** conçu autour des propriétés satisfaisant les besoins posés par la plasticité de façon à atteindre l'objectif (4). Le modèle Ethylene assure trois fonctions : (1) manipulation uniforme des assemblages de

composants de la couche applicative (quelles que soient leurs technologies de mise en œuvre), (2) composition des assemblages de composants hétérogènes, (3) description des composants implémentés suivant des modèles à composants différents. Ethylene se décompose en trois parties. En premier lieu, Ethylene décrit un cycle de vie des composants dynamiques, que ceux-ci soient constitués de code exécutable ou de modèles transformables. En deuxième lieu, Ethylene décrit structurellement les différents concepts du modèle et leurs relations. Notamment, Ethylene sépare clairement le concept de composant de celui de connecteur, propose le concept de fabrique destiné à l'instanciation des composants et à l'établissement de leurs liaisons, propose un cadre pour la description fonctionnelle et extra-fonctionnelle des composants et des connecteurs, puis, introduit deux classes de contrats et trois classes de clauses destinées à encadrer l'assemblage dynamique des composants. Enfin, le dernier apport d'Ethylene est une spécification à haut niveau d'abstraction de trois API motivées par la plasticité et destinées à régir l'interaction: (1) entre les fabriques, (2) entre les fabriques et le producteur de l'adaptation et, (3) entre les fabriques et le gestionnaire de composants.

Ethylene constitue, sur le plan conceptuel, le deuxième résultat attendu de cette thèse. Ma dernière contribution a consisté à lui faire correspondre une incarnation technique. Cette incarnation a pris la forme d'un langage XML baptisé **EthyleneXML destiné à la spécification de composants** et d'un **cadre de développement baptisé EthyleneFramework, disponible en Java et C++, destiné à faciliter le développement d'intergiciels compatibles avec l'approche Ethylene.** Enfin, EthyleneXML et EthyleneFramework sont mis en œuvre dans un **démonstrateur** pour lequel, d'une part une infrastructure logicielle pour la plasticité existante a été modifiée pour intégrer l'approche Ethylene, et d'autre part un ensemble de composants IHM implémentés dans des espaces technologiques différents.

Ce travail de recherche doctorale qui a permis de produire les résultats attendus répartis sous la forme de quatre contributions, ouvre quelques perspectives.

2. Perspectives

J'identifie quatre principales perspectives à ces travaux de recherche que je classe en trois catégories : les perspectives à court, moyen et long terme.

À court terme, je souhaite ajouter à EthyleneFramework des mécanismes destinés au traitement des composants

transformables. En effet, actuellement, une fabrique n'est capable d'instancier que des composants exécutables. Cependant, il ne s'agit pas pour autant de lui ajouter des capacités de transformation : la transformation d'un composant transformable dépend d'une part des métamodèles des modèles qui constituent le composant et d'autre part du contexte de l'interaction courant. Ainsi il semble préférable de réaliser la transformation par le biais d'un transformateur externe à la fabrique qui serait recruté à la volée en fonction des métamodèles des modèles à transformer et du contexte de l'interaction courant. Si la fonction de sélection du bon transformateur fait partie de la couverture fonctionnelle d'un moteur d'évolution, la fonction de mise en œuvre de la transformation par le transformateur sélectionné fait partie de la couverture fonctionnelle d'un configurateur. Enfin, une fois atteint un état exécutable, la gestion du cycle de vie de l'instance de composant résultante doit être placée sous la responsabilité d'une fabrique, à l'instar des composants directement exécutables.

À moyen terme, il serait intéressant d'évaluer dans quelle mesure l'approche Ethylene pourrait être appliquée à d'autres technologies à composants. Actuellement, EthyleneFramework applique Ethylene à SlimComponent, une technologie à composants minimalistes développée à l'occasion de cette thèse. L'objectif serait alors d'implémenter d'autres cadres de développement, l'un appliquant Ethylene à Fractal, l'autre l'appliquant à iPOJO, ou un troisième l'appliquant à WCOMP. Dans le même esprit, il serait intéressant d'utiliser EthyleneFramework pour intégrer l'approche Ethylene à un autre intergiciel pour l'informatique ubiquitaire, de façon à évaluer dans quelle mesure Ethylene peut être réutilisé dans un contexte différent de celui des travaux de l'équipe IIHM.

Enfin, à long terme, se pose la question d'assembler des composants dont l'hétérogénéité dépasse celle des technologies à composants. En effet, l'approche Ethylene permet de construire des systèmes interactifs à partir de composants implémentés dans des technologies à composants différentes. Cependant, l'assemblage fonctionne si les composants hétérogènes qui le constituent sont organisés conformément à une architecture logicielle dont il partage les tenants et aboutissants. Par exemple, dans le cas du démonstrateur PhotoBrowser, l'infrastructure MARI joue le rôle du contrôleur de dialogue et impose que les composants IHM en présence soient en mesure d'appliquer ses décisions relatives à la gestion du dialogue. Un composant qui ne fonctionnerait pas sur ce principe, quelle que soit sa technologie d'implémentation, ne pourrait pas être utilisé par MARI. Ainsi, au niveau de l'objectif d'assemblage de composants hétérogènes, Ethylene ne repousse que la limite « syntaxique » posée par les technologies à composant d'implémentation. Il reste des efforts de

recherche à fournir pour repousser la limite « sémantique » posée par les différentes architectures logicielles utilisées en IHM.

Bibliographie

1. [Allen97] Allen, R., "A Formal Approach to Software Architecture", In: Ph.D. Thesis, Carnegie Mellon University, CMU Technical Report CMU-CS-97-144, May 1997.
2. [Allen98] Allen, R., Douence, R., Garlan, D., "Specifying and analyzing dynamic software architectures", In: Proceedings of Fundamental Approaches to Software Engineering conference (FASE'98), Lisbon, Portugal, Springer LNCS 1382, ISBN 978-3-540-64303-6, p 21, March 1998.
3. [Amigo04] Amigo : Ambient Intelligence for the networked home environment, European IP project, IST 004182, http://www.hitech-projects.com/euprojects/amigo/, September 2004.
4. [Amigo-D2.1] Amigo WP2, "Specification of the Amigo Abstract Middleware Architecture", In: Deliverable D2.1, AMIGO Project (IST 2004-004182), April 11th, 2005.
5. [André93] André, E., Rist, T., "The design of illustrated documents as a planning task", In: Intelligent Multimedia Interfaces, Maybury M. T. (éd.), Chapitre 4, p. 94-116, AAAI Press / The MIT Press, Menlo Park, CA, 1993.
6. [ARToolKit] ARToolKit website http://www.hitl.washington.edu/artoolkit/.
7. [Balme05] Balme, L., Demeure, A., Calvary, G., Coutaz, J., "Sedan-Bouillon: a plastic web site", In: Plastic Services for Mobile Devices (PSMD), Workshop held in conjunction with the 10th IFIP International conference on human-computer interaction, INTERACT 2005, Rome, Italy, September 2005.
8. [Bandelloni04] Bandelloni, R., Paternò, F., "Flexible interface migration", In: Proceedings of ACM International Conferene on Intelligent User Interfaces, IUI 2004, p. 148-155, Funchal, Island of Madeira, Portugal, January 2004.

9. [Bandelloni04-1] Bandelloni, R., Berti, S., Paternò F., “Mixed-Initiative, Trans-Modal Interface Migration”, In: Proceedings Mobile HCI 2004, LNCS 3160, p. 216-227, Glasgow, September 2004.

10. [Barralon06] Barralon, N., "Couplage de Ressources d'Interaction en Informatique Ambiante", In: In: PhD thesis, Université Joseph Fourier, Grenoble, France, December 2006.

11. [Batista05] Batista, T., Joolia, A., Coulson, R., "Managing dynamic reconfiguration in component-based systems", In: Proceedings of 2nd European Workshop on Software Architecture, EWSA 2005, p 1-17, LNCS 3527, ISBN: 978-3-540-26275-6, Pisa , Italy, June 2005.

12. [Bérard06] Bérard, F., "The GML canvas: Aiming at Ease of Use, Compactness and Flexibility in a Graphical Toolkit", Technical Report, 2006.

13. [Berger01] Berger, L., "Mise en Oeuvre des Interactions en Environnements Distribués, Compilés et Fortement Typés: Le Modèle MICADO", In: PhD thesis, Université de Nice - Sophia Antipolis, October 2001.

14. [Bernsen93] Bernsen, N. O., "Taxonomy of HCI Systems: State of the Art", ESPRIT BR GRACE, deliverable 2.1, 1993.

15. [Berthomé-Montoy95] Berthomé-Montoy, A., "Une approche descriptive de l'auto-adaptativité des interfaces homme-machine", In: PhD Thesis, Université de Lyon 1, 1995.

16. [Berti05] Berti, S., Paternò, F., "Migratory MultiModal interfaces in MultiDevice environments", In: Proceedings of the 7th international Conference on Multimodal interfaces, ICMI 2005, p. 92-99, Torento, Italy, October 2005.

17. [Beugnard99] Beugnard, A., Jézéquel, J-M., Plouzeau, N., Watkins, D., "Making Components Contract Aware", In: (IEEE) Computer, July 1999, p. 38-45.

18. [Bieber01] Bieber, G., Carpenter, J., "Introduction to Service-Oriented Programming (Rev2.1)", Online document http://www.openwings.org, April 2001.

19. [Blanch05] Blanch, R., "Architecture logicielle et outils pour les interfaces hommes-machines graphiques avancées", In: PhD thesis, Université Paris XI, Orsay, France, Septembre 2005.

20. [Bouillon02] Bouillon, L., Vanderdonckt, J., Souchon, N., "Recovering Alternatives Presentation Models of a Web Page with Vaquita", In: Proceedings of 4th International Conference on Computer-Aided Design of User Interfaces, CADUI 2002, Kluwer Academics Pub., p. 311-322, Valenciennes, France, May 2002.

21. [Bouillon04] Bouillon, L., Vanderdonckt, J., Chow, K.C., "Flexible Re-engineering of Web Sites", In: Proceedings of the 8th ACM International Conferene on Intelligent User Interfaces, IUI 2004, p. 132–139, Funchal, Island of Madeira, Portugal, January 2004.

22. [Browne90] Browne, D., Totterdell, P., Norman, M., "Adaptive User Interfaces", Computers And People series, Academic Press, ISBN 0-12-137755-5, 1990.

23. [Brunneton02] Brunneton, E., Coupaye, T., Stefani, J.B., "Recursive and Dynamic Software Composition with Sharing", In: Proceedings of the Seventh International Workshop on Component-Oriented Programming, WCOP 2002, Malaga, Spain, June 2002.

24. [Brusilovsky01] Brusilovsky, P., "Adaptive hypermedia", In: User Modeling and User Adapted Interaction, Ten Year Anniversary Issue (Alfred Kobsa, ed.) 11 (1/2), p. 87-110, March 2001.

25. [Bureš06] Bureš, T., Hnětynka, P., and Plášil, F., "SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model", In: Proceedings of the Fourth international Conference on Software Engineering Research, Management and Applications, SERA 2006, p.40-48, Washington, DC, USA, August 2006.

26. [Calvary01] Calvary, G., Coutaz, J., Thevenin, D., "Supporting Context Changes for Plastic User Interfaces: a Process and a Mechanism", In Joint Proceedings of HCI 2001 and IHM 2001, BCS Conference series, Springer Publication, ISBN 1-85233-515-7, pp. 349-363, Lille, France, September 2001.

27. [Calvary05] Calvary, G., Daassi, O., Coutaz, J., Demeure, A., "Des widgets aux comets pour la Plasticité des Systèmes Interactifs", In: Revue d'Interaction Homme-Machine (RIHM), Volume 6, Numero 1, p. 33-53, Europia, Paris, 2005.

28. [Calvary07] Calvary, G., "Plasticité des Interfaces Homme-Machine", In: Habilitation à Diriger des Recherches, Université Joseph Fourier, Grenoble, France, 2007.

29. [Cao06] Cao, X., Balakrishnan, R., "Interacting with dynamically defined information spaces using a handheld projector and a pen", In: Proceedings of the 19th Annual ACM Symposium on User interface Software and Technology, UIST 2006, p. 225-234, Montreux, Switzerland, October 2006.

30. [Card87] Card, S., Henderson, A., "A multiple virtual workspace interface to support user task switching", In: Proceedings of the 5th ACM Conference on human factors in computing systems, CHI+GI 1987, p. 53-59, Toronto, Ontario, Canada, 1987.

31. [Cervantes04] Cervantes, H., "Vers un modèle à composants orienté services pour supporter la disponibilité dynamique", PhD Thesis, Université Joseph Fourier, Grenoble, Mars 2004.

32. [Cheung-Foo-Wo06] Cheung-Foo-Wo, D., Blay-Fornarino, M., Tigli, J.-Y., Lavirotte, S., Riveill, M., "Adaptation dynamique d'assemblages de dispositifs dirigée par des modèles", In: Proceedings of the 2èmes Journées sur l'Ingénierie Dirigée par les Modèles (IDM), June 2006.

33. [Cockton05] Cockton, G., "A development framework for value-centred design", In: Proceedings of ACM International Conference On Human Factors In Computing Systems, CHI 2005, Late breaking results, p. 1292-1295, Portland, OR, USA, April 2005.

34. [Coutaz95] Coutaz, J., Nigay, L., Salbert, D., Blandford, A., May, J., "Four easy pieces for assessing the usability of multimodal interaction: the CARE properties", In: Proceedings of the IFIP Interantional Conference on Human-Computer Interaction, INTERACT 1995, p. 115-120, Lillehammer, Norway, June 1995.

35. [Coutaz06] Coutaz, J., "Meta-User Interfaces for Ambient Spaces", Invited Talk, In: Proceedings of the fifth Tasks Models and Diagrams for UI design workshop, TAMODIA 2006, LNCS 4385, p. 11-18, Hasselt, Belgium, October 23-24, 2006.

36. [Coutaz08] Coutaz, J., Calvary, G., "HCI and Software Engineering: Designing for User Interface Plasticity", In: The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies, and Emerging Applications, Second Edition, ISBN 9780805858709, Taylor & Francis CRC Press, Human Factor and Ergonomics series, A. Sears, J. Jacko Edsp. 1107-1125, 2008.

37. [Coyette04] Coyette, A., Faulkner, S., Kolp, M., Limbourg, Q., Vanderdonckt, J., "SketchiXML: towards a multi-agent design tool for sketching user interfaces based on USIXML", In: Proceedings of the 3rd Annual Conference on Task Models and Diagrams, TAMODIA 2004, Prague, Czech Republic, November 2004.

38. [Crease00] Crease, M., Brewster, S.A., Gray, P., "Caring, Sharing Widgets: a toolkit of sensitive widgets", In: Proceedings of the 14th Annual Conference of the British HCI Group, BCS-HCI 2000, Springer, pp 257-270, Sunderland, UK, September 2000.

39. [Crease01] Crease, M., "A Toolkit of Resource Sensitive, Multimodal Widgets", PhD Thesis, Department of Computing Science, University of Glasgow, Glasgow, Scotland, UK, 2000.

40. [Cypher91] Cypher, A., "Eager: Programming Repetitive Tasks by Example", In: Proceedings of the ACM Conference on human factors in computing systems, CHI 1991, p. 33-39, New Orleans, LA, USA, April 1991.

41. [Dâassi07] Dâassi, O., "Les comets : une nouvelle génération d'Interacteurs pour la Plasticité des Interfaces Homme-Machine", In: PhD thesis, Université Joseph Fourier, Grenoble, France, January 2007.

42. [David06] David, D., Ledoux, T., "Une approche par aspects pour le développement de composants Fractal adaptatifs", In: Revue des Sciences et Technologies de l'Information - L'Objet, Volume 12, numéro 2-3, p. 113-132, 2006.

43. [Demeure05] Demeure, A., Balme, L., Calvary, G., Coutaz, J., "CamNote: a plastic slide viewer", In: Plastic Services for Mobile Devices (PSMD), Workshop held in conjunction with the 10th IFIP

International conference on human-computer interaction, INTERACT 2005, Rome, Italy, September 2005.

44. [Demeure07] Demeure, A., "Modèles et outils pour la conception et l'exécution d'interfaces homme-machine plastiques", In: PhD Thesis, Université Joseph Fourier, Grenoble, France, October 2007.

45. [Deremer75] DeRemer, F., Kron, H., "Programming-in-the large versus programming-in-the-small", In: Proceedings of the international Conference on Reliable Software, p 114-121, Los Angeles, California, April 21-23, 1975.

46. [Dieterich93] Dieterich, H., Malinowski, U., Kühme, T., Schneider-Hufschmidt, M., "State of the art in adaptive user interfaces", In: Adaptive User Interfaces, Principle and Practice, p. 13-48, Human Factors in Information Technology series, Volume 10, Elsevier Science Publishers, 1993.

47. [Dragicevic01] Dragicevic, P., Fekete, J.-D., "Input Device Selection and Interaction Configuration with ICON", In: Joint Proceedings of HCI 2001 and IHM 2001, BCS Conference series, Springer Publication, ISBN 1-85233-515-7, Lille, France, September 2001.

48. [Dumant99] Dumant. B., Horn, F., Dang Tran, F., Stefani, J. B., "Jonathan: an open distributed processing environment in Java", In: Distributed Systems Engineering Journal, Volume 6, Number 1, p. 3-12, March 1999.

49. [Economides96] Economides, N., "The Economics of Networks", In: International Journal of Industrial Organization, Volume 14, Numero 2, March 1996.

50. [Edwards01] Edwards, W. K., Newman, M. W., Sedivy, J. Z., "The case of recombinant computing", In: Technical report CSL-01-1, Xerox Palo Alto Research Center, Palo Alto, CA, USA, April 2001.

51. [Eftring99] Eftring, H., "The Useworthiness of Robots for People with Physical Disabilities", In: Doktorsavhandling, Certec, Institutionen för Designvetenskaper, ISBN 91-628-3711-7, Lunds Tekniska Högskola, September 1999.

52. [Emode06] The EMODE Consortium, "Models Definition", In: Deliverable D2.2, EMODE project (ITEA if04046), July 2006.

53. [Escoffier07] Escoffier, C., Hall, R. S., "Dynamically Adaptable Applications with iPOJO Service Components", In: Proceedings of the 6th International Symposium on Software Composition, SC 2007, Braga, Portugal, March 2007.

54. [Favre04-1] Favre, J.-M., "Foundation of models (driven) (reverse) engineering: models - episode I, stories of the Fidus papyrus and of the Solarus", In: Proceedings of Dagstuhl Seminar 04101 on Language Engineering for Model-Driven Software Development, Dagstuhl, Germany, March 2004.

55. [Florins04] Florins, M., Vanderdonckt, J., "Graceful Degradation of User Interfaces as a Design Method for Multiplatform Systems", In:

Proceedings of the 9th ACM International Conference on Intelligent User Interfaces, IUI 2004, p. 140-147, Funchal, Madeira Island, Portugal, January 2004.

56. [Gajos04] Gajos, K., Weld, D., "SUPPLE : Automatically Generating User Interfaces", In: Proceedings of ACM International Conferene on Intelligent User Interfaces, IUI 2004, p. 93-100, Funchal, Island of Madeira, Portugal, January 2004.

57. [Gajos07] Gajos, K., Wobbrock, J., Weld, D., "Automatically Generating User Interfaces Adapted To Users' Motor And Vision Capabilities", In: Proceedings of the 20th Annual ACM Symposium on User interface Software and Technology, UIST 2007, Newport, RI, USA, 2007.

58. [Gamma95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., "Design patterns: Elements of reusable object-oriented software", Professional Computing Series, Addison-Wesley, January 1995.

59. [Ganneau07] Ganneau, V., Calvary, G., Demumieux, R., "Métamodèle de Règles d'Adaptation pour la Plasticité des Interfaces Homme-Machine", In proceedings of the 19th Conférence francophone sur l'Interaction Homme-Machine, IHM'2007, p. 8, Paris, France, November 2007.

60. [Garlan97] Garlan, D., Monroe, R. T., Wile, D., "An Architecture Description Interchange Language", In: Proceedings of CASCON 1997, November 1997.

61. [Grace03] Grace, P., Blair, G. S., Samuel, S., " Middleware awareness in mobile computing", In: Proceedings of the 1st International Conference on Distributed Computing Systems Workshop, ICDCS Workshop 2003, p. 382-387, May 2003.

62. [Greenhalgh04] Greenhalgh, C., Izadi, S., Mathrick, J., Humble, J., Taylor, I., "ECT: A Toolkit to Support Rapid Construction of Ubicomp Environments", In: Proceedings of International Workshop on System Support for Ubiquitous Computing, UbiSys 2004, at the 6th International Conference on Ubiquitous Computing, UBICOMP 2004, Nottingham, England, UK, September 2004.

63. [Grolaux05] Grolaux, D., Vanderdonckt, J., Van Roy, P., "Attach Me, Detach Me, Assemble Me Like You Work", In: Proceedings of 10th IFIP International conference on human-computer interaction, INTERACT 2005, p. 198-212, Rome, Italy, September 2005.

64. [Grolaux07] Grolaux, D., "Transparent Migration and Adaptation in Graphical User Interface Toolkit", In: PhD Thesis, Faculté des sciences appliquées, Université Catholique de Louvain, Septembre 2007.

65. [Haahr99] Haahr, M., Cunningham, R., Cahill, V., "Supporting CORBA applications in a mobile environment", In: Proceedings of the 5th Annual ACM/IEEE international Conference on Mobile Computing and Networking, MobiCom 1999, p. 36-47, Seattle, WA, USA, August 1999.

66. [Hnětynka06] Hnětynka, P., Plášil, F., "Dynamic Reconfiguration and Access to Services in Hierarchical Component Models", In: Proceedings of The 9th International SIGSOFT Symposium on Component-Based Software Engineering, CBSE 2006, LNCS 4063, p. 352-359, Västerås near Stockholm, Sweden, June 2006.

67. [ISO 9241] UsabilityNet.org, "International standards for HCI and usability", In: Online document at http://www.usabilitynet.org/tools/r_international.htm.

68. [Joolia05] Joolia, A., Batista, T., Coulson, G., and Gomes, A. T., "Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform", In: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005), p 131-140, November 6-10, 2005.

69. [Kiczales97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., Irwin, J., "Aspect-Oriented Programming", In: Proceedings of 11th European Conference on Object-Oriented Programming, ECOOP 1997, LNCS 1241, p. 220-242, Jyvskyl, Finland, June 1997.

70. [Kephart03] Kephart, J. O., Chess, D. M., "The Vision of Autonomic Computing", In: IEEE Computer, Volume 36 , Issue 1, p. 41-50, January 2003.

71. [Kobsa01] Kobsa, A., "Generic User Modeling Systems", In: User Modeling and User Adapted Interaction, Ten Year Anniversary Issue (Alfred Kobsa, ed.) 11 (1/2), p. 49-63, March 2001.

72. [Kurtev02] Kurtev, I., Bézivin, J., Aksit, M., "Technological spaces: an initial appraisal", In: CoopIS, DOA'2002 Federated Conferences, Industrial track, Irvine, CA, USA, October 2002.

73. [Laborie06] Laborie, S., Euzenat, J., Layaïda, N., "Adaptation spatiale efficace de documents SMIL", In: Proceedings of 15e congrès francophone AFRIF-AFIA Reconnaissance des Formes et Intelligence Artificielle, RFIA 2006, Tours, France, January 2006.

74. [Lachenal04] Lachenal, C., "Modèle et infrastructure logicielle pour l'interaction multi-instrument multisurface"m In. PhD thesis, Université Joseph Fourier, Grenoble, France, 2004.

75. [Layaida05] Layaïda, N., Lemlouma, T., Quint, V., "NAC, une architecture pour l'adaptation multimédia sur le web", In: Technique et Science Informatiques (TSI), vol. 24 (7), pages 789-813, 2005.

76. [Lecolinet99] Lecolinet, E., "A Brick Construction Game Model for Creating Graphical User Interfaces: The Ubit Toolkit", In: Proceedings of the 7th IFIP TC13 International Conference on Human-Computer Interaction, INTERACT 1999, Edinburgh, Scotland, UK, September 1999.

77. [Legond-Aubry05] Legond-Aubry, F., "Un modèle d'assemblage de composants par Contrat et Programmation Orientée Aspect", In:

PhD Thesis, Conservatoire National des Arts et Métiers, July 2005.

78. [Lewandowski07] Lewandowski, A., Bourguin, G., Tarby, J.-C., "De l'Orienté Objet à l'Orienté Tâches – Des modèles embarqués pour l'intégration et le traçage d'un nouveau type de composants", In: Revue d'Interaction Homme-Machine, Vol 8, n° 1, p. 1-34, December 2007.

79. [Lieberman06] Lieberman, H., Espinosa, J., "A Goal-Oriented Interface to Consumer Electronics using Planning and Commonsense Reasoning", In: Proceedings of the 2006 International Conference on Intelligent User Interfaces, IUI 2006, p.226-233, Sydney, Australia, January 2006.

80. [Limbourg04] Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., López-Jaquero, V., "USIXML: A Language Supporting Multi-path Development of User Interfaces", In: Proceedings of the 9th IFIP Working Conference on Engineering for Human-Computer Interaction, EHCI-DSVIS 2004, LNCS 3425, p. 200-220, Hamburg, Germany, July 2004.

81. [Luyten02] Luyten, K., Vandervelpen, C., Coninx, K., "Migratable user interface descriptions in component-based development", In: Proceedings of 9th international workshop on Design, Specification and Verification of Interactive Systems, DSV-IS 2002, LNCS 2545, p. 44-58, Rostock, Germany, June 2002.

82. [Lyytinen02] Lyytinen, K., Yoo, Y., "Issues and challenges in ubiquitous computing: Introduction", In: Communication of the ACM, Volume 45, Issue 12, p. 62-65, December 2002.

83. [Mørch97] Mørch, A., "Three levels of end-user tailoring: customization, integration, and extension", In: Method and Tools for Tailoring Object-Oriented Applications: An Evolving Artifacts Approach, PhD thesis, Dept of Informatics, University of Oslo, p. 41-51, 1997.

84. [Mori03] Mori, G., Paternò, F., Santoro, C., "Tool support for designing nomadic applications", In: Proceedings of ACM International Conferene on Intelligent User Interfaces, IUI 2003, p. 141-148, Miami, FL, USA, January 2003.

85. [Maybury93] Maybury, M. T., (ed.) "Intelligent Multimedia Interfaces", AAAI Press / The MIT Press, Menlo Park, CA, 1993.

86. [Mascolo02] Mascolo, C., Capra, L., Emmerich, W., "Mobile Computing Middleware", In: Advanced Lectures on Networking : NETWORKING 2002 Tutorials, LNCS 2497, p. 20-58, 2002.

87. [Myers00] Myers, B., Hudson, S.E., Pausch, R., "Past, Present, and Future of User Interface Software Tools", Transactions on Computer-Human Interaction (TOCHI), ACM Publ., Vol 7(1), p. 3-28, 2000.

88. [Newman02] Newman, M. W., Sedivy, J. Z., Neuwirth, C. M., Edwards, W. K., Hong, J. I., Izadi, S., Marcelo, K., Smith, T. F., "Designing for

Serendipity: Supporting End-User Configuration of Ubiquitous Computing Environments", In: Proceedings of Designing Interactive Systems, DIS 2002, p. 147-156, London, UK, June 2002.

89. [Meyer91] Meyer, B., "Eiffel : The Language", Prentice Hall, October 1991.

90. [Meyer92] Meyer, B., "Applying 'Design by contract', In: (IEEE) Computer, October 1992, p. 40-52.

91. [Mozart] The Mozart Programming System, accessible at http://www.mozart-oz.org/.

92. [Nichols02] Nichols, J., Myers, B. A., Higgins, M., Hughes, J., Harris, T. K., Rosenfeld, R., Pignol, M., "Generating remote control interfaces for complex appliances", In: Proceedings of the 15th Annual ACM Symposium on User interface Software and Technology, UIST 2002, p. 161-170, Paris, France, October 2002.

93. [Nichols06] Nichols, J., Myers, B. A., Rothrock, B., "UNIFORM : Automatically Generating Consistent Remote Control User Interfaces", In: Proceedings of ACM International Conference On Human Factors In Computing Systems, CHI 2006, p. 611-620, 2006.

94. [Nichols06-1] Nichols, J., Rothrock, B., Chau, D. H., Myers, B. A., "Huddle: Automatically Generating Interfaces for Systems of Multiple Connected Appliances," In: Proceedings of the 19th Annual ACM Symposium on User interface Software and Technology, UIST 2006, p. 279-288, Montreux, Switzerland, October 2006.

95. [Nielsen93] Nielsen, J., "Usability Engineering", Academic Press, Boston, ISBN 0-12-518406-9, 1993.

96. [Nigay94] Nigay, L., "Conception et modélisation logicielles des systèmes interactifs : application aux interfaces multimodales", In: PhD thesis, Université Joseph Fourier, Grenoble, January 1994.

97. [Norman98] Norman, D., "The Invisible Computer", MIT Press, 1998.

98. [Normand92] Normand, V., "Le modèle SIROCO : de la spécification conceptuelle des interfaces utilisateur à leur réalisation", In: PhD thesis, Université Joseph Fourier, Grenoble, France, April 1992.

99. [Omg07] Object Management Group, "Unified Modeling Language: Superstructure, version 2.1.2", In: Normative specification formal/2007-11-02, http://www.omg.org/, November 2007.

100. [Oquendo04] Oquendo, F. "π-ADL: an Architecture Description Language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures", ACM SIGSOFT Softw. Eng. Notes, Volume 29, Number 4, p 1-14, May 2004.

101. [Oreizy99] Oreizy, P., et al., "An Architecture-Based Approach to Self-Adaptive Software", In: IEEE Intelligent Systems, Volume 14, no. 3, p 54-62, May-June 1999.

102. [Paternò02] Paternò, F., Santoro, C., "One Model, Many Interfaces", In: Proceedings of 4th International Conference on Computer-Aided Design of User Interfaces, CADUI 2002, Kluwer Academics Pub., Valenciennes, France, May 2002.

103. [Ponnekanti01] Ponnekanti, S., Lee, B., Fox, A., Hanrahan, P., Winograd, T., "ICrafter : A Service Framework for Ubiquitous Computing Environments", In: Proceedings of ACM UBICOMP 2001, p 56-75, Atlanta, Georgia, USA, September 2001.

104. [Pousman06] Pousman, Z., Stasko, J., "A taxonomy of ambient information systems: Four patterns og design", In: Proceedings of the 8th International Working Conference on Advanced Visual Interfaces, ACM AVI 2006, p. 67-74, Venezia, Italy, May 2006.

105. [Puerta02] Puerta, A., Eisenstein, J., "XIML: a common representation for interaction data", In: Proceedings of the 7th international Conference on intelligent User interfaces, IUI 2002, p. 214-215, San Francisco, California, USA, January 2002.

106. [Rey05] Rey, G., "Contexte en Interaction Homme-Machine : le contexteur", In: PhD thesis, Université Joseph Fourier, Grenoble, France, 2005.

107. [Roman01] Roman, M., Kon, F., Campbell, R. H., "Reflective middleware : from your desk to your hand", In: IEEE Distributed Systems Online, Volume 2, Number 5, 2001.

108. [Roudaut06] Roudaut, A., Coutaz, J., "Méta-IHM ou comment contrôler l'espace interactif ambiant", In: Proceedings of the 3rd Journées Francophones Mobilité et Ubiquité, Ubimob 2006, ACM Publication, Paris, France, September 2006.

109. [Rousseau06] Rousseau, C., Bellik, Y., Vernier, F., "A Conceptual Model for Multimodal and Contextual Presentation of Information", In: French Human-Computer Interaction Journal, RIHM, Volume 7, 2006.

110. [Ryan04] Ryan, N. D., Wolf, A. L., "Using Event-Based Parsing to Support Dynamic Protocol Evolution", In: Proceedings of the 26th International Conference on Software Engineering, ICSE 2004, p. 408-417, Edinburgh, Scotland, UK, May 2004.

111. [Schlee04] Schlee, M., Vanderdonckt, J., "Generative Programming of Graphical User Interfaces", In: Proceedings of the 7th International Working Conference on Advanced Visual Interfaces, AVI 2004, p. 403–406, Gallipoli, Italy, May 2004.

112. [Shaw95] Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., Zelesnik, G., "Abstractions for Software Architecture and Tools to Support Them", IEEE transactions on software engineering, Volume 21, Issue 4, p 314-335, April 1995.

113. [Sousa03] Sousa, J.P., Garlan, D., "The Aura Software Architecture: an Infrastructure for Ubiquitous Computing", In: Carnegie Mellon Technical Report, CMU-CS-03-183, August 2003.

114. [Sousa05] Sousa, J.-P., "Scaling Task Management in Space and Time: Reducing User Overhead in Ubiquitous-Computing Environments", In: Ph.D. Thesis, Carnegie Mellon University, CMU Technical Report CMU-CS-05-123, March 2005.

115. [Sottet07] Sottet, J.-S., Ganneau, V., Calvary, G., Coutaz, J., Favre, J.-M., Demumieux, R., "Model-Driven Adaptation for Plastic User Interfaces", In: Proceedings of the 11th IFIP TC13 International Conference on Human-Computer Interaction, INTERACT 2007, Rio De Janeiro, Brasil, September 2007.

116. [Stuerzlinger06] Stuerzlinger, W., Chapuis, O., Phillips, D., Roussel, N., "User interface façades: towards fully adaptable user interfaces", In: Proceedings of the 19th Annual ACM Symposium on User interface Software and Technology, UIST 2006, p. 309-318, Montreux, Switzerland, October 2006.

117. [Szyperski02] Szyperski, C., Gruntz, D., Murer, S., "Component Software: Beyond Object-Oriented Programming, Second edition", ACM Press, ISBN 0-201-74572-0, 2002.

118. [Tandler01] Tandler, P., "Software Infrastructure for Ubiquitous Computing Environments: Supporting Synchronous Collaboration with Heterogeneous Devices", In: Proceedings of UBICOMP 2001, LNCS 2201, p. 96-115, Atlanta, GA, USA, October 2001.

119. [Tandler01-2] Tandler, P., Prante, T., Müller-Tomfelde, C., Streitz, N., Steinmetz, R., "ConnecTables: Dynamic Coupling of Displays for the Flexible Creation of Shared Workspaces", In: Proceedings of the 14th Annual ACM Symposium on User interface Software and Technology, UIST 2001, p. 11-20, Orlando, Florida, November 2001.

120. [Thevenin99] Thevenin, D., Coutaz, J., "Plasticity of user-interfaces: framework and research agenda", In: Proceedings of 7th IFIP conference on human-computer interaction, INTERACT 1999, p. 110-117, Edinburgh, Scotland, September 1999.

121. [Thevenin01] Thevenin, D., "Adaptation en Interaction Homme-Machine : Cas de la plasticité", In: PhD thesis, Université Joseph Fourier, Grenoble, France, 2001.

122. [Totterdell90] Totterdell, P., Rautenbach, P., "Adaptation as a problem design", In : Adaptive User Interfaces, p. 59-84, Computer and People Series, Academic Press, 1990.

123. [UIMS92] The UIMS tool developers workshop, "A metamodel for the runtime architecture of an interactive system", ACM SIGCHI Bulletin, Volume 24 , Issue 1, p. 32-37, January 1992.

124. [Van Roy04] Van Roy, P., Haridis, S., "Concepts, Techniques, and Models of Computer Programming", MIT Press, ISBN 0-262-22069-5, March 2004.

125. [W3c04] W3C, "Web Services Architecture", In: W3C Working group Note, http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/ , February 2004.

126. [Weiser91] Weiser, M., "The Computer for the 21st Century", Scientific American, p 94-104, September 1991.

127. [Wile01] Wile, D. S., "Using Dynamic Acme", In: Proceedings of a Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia, December 2001.

128. [Zimmermann80] Hubert Zimmermann, "OSI Reference Model — The ISO Model of Architecture for Open Systems Interconnection", In: IEEE Transactions on Communications, vol. 28, no. 4, April 1980, pp. 425 – 432

Annexes

1. Spécifications complète d'EthyleneXML

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
   xmlns:xs=                          "http://www.w3.org/2001/XMLSchema"
   xmlns=                             "http://iihm.imag.fr/namespaces/ethylenexml/1.0"
   targetNamespace=    "http://iihm.imag.fr/namespaces/ethylenexml/1.0">

   <xs:import  namespace="http://www.w3.org/XML/1998/namespace"
                     schemaLocation="http://www.w3.org/2001/xml.xsd"/>

   <xs:annotation>
      <xs:documentation xml:lang="fr">

      Ce schéma XML constitue la spécification initiale du langage EthyleneXML. Ce langage,
      conforme au métamodèle à composants dynamiques Ethylene, a pour objectif la
      description de composants logiciels destinés à la construction       d'interfaces
      utilisateur distribuées et dynamiquement adaptables. Une bonne connaissance de ce
      métamodèle est recommandée pour une meilleure compréhension de cette spécification.

      Un composant logiciel décrit dans un document EthyleneXML s'inscrit dans un espace de
      noms. Un espace de noms peut contenir la spécification de plusieurs composants
      logiciels. L'objectif d'un espace de noms est de regrouper en son sein la
      spécification d'un ensemble d'éléments constituant un certain domaine. Par exemple,
      l'espace de nom http://iihm.imag.fr/namespaces/demo/ contient la description de
      l'ensemble des composants logiciels impliqués dans le démonstrateur relatif au
      fonctionnement du métamodèle à composants dynamiques M2R-Component, ainsi que
      l'ensemble des types de données, interfaces et ports spécifiques à ces composants.

      Un espace de noms EthyleneXML se décompose en un espace de noms racine et deux
      sous-espaces. L'espace de noms racines spécifies les propriétés, les ports et les
      composants logiciels. Le premier sous espace de noms, /data,  contient la
      spécification des structures de données manipulées par les composants logiciels de
      l'espace de noms racine. Il est définit par un schéma XML se conformant à l'espace de
      noms http://www.w3.org/2001/XMLSchema. Le deuxième sous-espace, /service, contient la
      spécification des interfaces programmatiques mises en oeuvre dans les ports de
      l'espace de noms racine. Il est définit par une spécification au format WSDL se
      conformant à l'espace de noms http://www.w3.org/2006/01/wsdl.

      # Recommandations

      - Le préfixe EthyleneXML : c2h4

      - Décomposition d'une spécification EthyleneXML :
            Il est recommandé de séparer d'une part la spécification des propriétés et les
            ports, et d'autre part la spécification des composants logiciels. Ainsi, dans
            un document EthyleneXML principal seront importé les spécifications XMLSchema
            et WSDL, décrit les propriétés de composants et les ports. Puis, dans un sous-
            dossier /components seront placé un document EthyleneXML par composants.

      - Convention de nommage des documents EthyleneXML
         + Document EthyleneXML principal : your_document_xml_prefix.c2h4.xml
         + Documents EthyleneXML composants : components/your_component_name.c2h4.xml
         + Document XMLSchema du sous-espace data : data/your_document_xml_prefix.xsd
         + Document WSDL su sous-espace service : service/your_document_xml_prefix.wsdl

      - Construction des URIs :
         + espace de noms racine
               http://your_organisation_name/namespaces/your_namespace/version_number/

         + sous-espace de noms des structures de données
               http://your_organisation_name/namespaces/your_namespace/version_number/data/

         + sous-espace de noms des interfaces programmatiques
               http://your_organisation_name/namespaces/your_namespace/version_number/service/


      # Objectifs

      Une spécification EthyleneXML permet d'atteindre trois buts. En premier lieu, lors des
      phases de conception, un document EthyleneXML peut permettre de valider par un moyen
      logiciel la correction d'une description de composant, notamment si les valeurs spécifiées
```

```
      pour les propriétés sont bien conforment à la spécification de ces propriétés. En deuxième
      lieu, suivant le cadre de travail à composants disponible, une spécification EthyleneXML
      permet la génération automatique du code relatif à l'encapsulation sous forme de composant
      du code métier. Enfin, à l'exécution, une spécification EthyleneXML permet l'évaluation
      des composants au regard de contrats d'assemblage.

      </xs:documentation>
   </xs:annotation>


<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <xs:element name="description">

      <xs:annotation>
         <xs:documentation xml:lang="fr">

         Un document EthyleneXML a pour racine un élément c2h4:description. Ce élément contient,
         dans l'ordre, les liens vers les spécifications des autres espaces de noms utilisés dans
         la description, puis la description des structures de données, des interfaces
         programmatiques, des propriétés de composants et les ports. Il est possible d'ajouter
         ensuite la description des composants. Néanmoins, il est recommandé de placer ces
         spécifications dans des fichiers à part (voir ci-dessus).

         </xs:documentation>
      </xs:annotation>

      <xs:complexType mixed="false">
         <xs:sequence>
            <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element       ref="import" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="types" minOccurs="0"/>
            <xs:element ref="interfaces" minOccurs="0"/>
            <xs:element ref="properties" minOccurs="0"/>
            <xs:element ref="ports" minOccurs="0"/>
            <xs:element ref="component" minOccurs="0" maxOccurs="unbounded"/>
         </xs:sequence>
         <xs:attribute name="targetNamespace" type="xs:anyURI" use="required"/>
      </xs:complexType>

   </xs:element>


<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <xs:element name="import">

      <xs:annotation>
         <xs:documentation xml:lang="fr">

         L'élément "import" permet d'importer au sein d'un document EthyleneXML un autre document
         EthyleneXML définissant un autre espace de nom. L'utilisation de cet élément n'a
         d'intétêt que dans le cadre de la validation automatique des documents EthyleneXML.

         </xs:documentation>
      </xs:annotation>

      <xs:complexType>
         <xs:attribute name="namespace" type="xs:anyURI" use="required"/>
         <xs:attribute name="location" type="xs:anyURI" use="required"/>
      </xs:complexType>

   </xs:element>


<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <xs:element name="types">

      <xs:annotation>
         <xs:documentation xml:lang="fr">

         L'élément "types" contient généralement un lien, via un sous-élément "import" vers la
         spécification XMLSchema des structures de données nécessaire à la description des
         paramètres d'opérations dans les interfaces WSDL et des propriétés des composants.

         Il est également possible, mais non recommandé, d'inclure la description XMLSchema des
         structures de données directement dans l'élément "types".
```

```
         </xs:documentation>
      </xs:annotation>

      <xs:complexType>
         <xs:sequence>
            <xs:element ref="import" minOccurs="0" maxOccurs="unbounded"/>
            <xs:sequence>
               <xs:any namespace="http://www.w3.org/2001/XMLSchema" minOccurs="0"
                                     maxOccurs="unbounded" processContents="strict"/>
            </xs:sequence>
         </xs:sequence>
      </xs:complexType>

   </xs:element>


<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <xs:element name="interfaces">

      <xs:annotation>
         <xs:documentation xml:lang="fr">

         L'élément "interfaces" contient généralement un lien, via un sous-élément "import" vers
         la spécification WSDL des interfaces programmatiques nécessaires à la description des
         ports des composants logiciels.

         Il est également possible, mais non recommandé, d'inclure la description WSDL des
         interfaces directement dans l'élément "interfaces".

         </xs:documentation>
      </xs:annotation>

      <xs:complexType>
         <xs:sequence>
            <xs:element ref="import" minOccurs="0" maxOccurs="unbounded"/>
            <xs:sequence>
               <xs:any namespace="http://www.w3.org/2006/01/wsdl" minOccurs="0"
                                     maxOccurs="unbounded" processContents="strict"/>
            </xs:sequence>
         </xs:sequence>
      </xs:complexType>

   </xs:element>


<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <xs:element name="properties">

      <xs:annotation>
         <xs:documentation xml:lang="fr">

         L'élément "properties" contient une liste de définitions de propriétés. Ces propriétés
         pourront être exposées par des composants.

         </xs:documentation>
      </xs:annotation>

      <xs:complexType>
         <xs:sequence>
            <xs:element ref="property" minOccurs="1" maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>

   </xs:element>


<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <xs:element name="ports">

      <xs:annotation>
         <xs:documentation xml:lang="fr">

         L'élément "ports" contient une liste de définitions de ports. Ces ports pourront être
         fournis ou utilisés par des composants.

         </xs:documentation>
```

```
      </xs:annotation>

      <xs:complexType>
         <xs:sequence>
            <xs:element ref="port" minOccurs="1" maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>

   </xs:element>


<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <xs:element name="property">

      <xs:annotation>
         <xs:documentation xml:lang="fr">

         L'élément "property" définit une caractéristique du composant ou du port qui
         l'intègre. Il sert alors d'éléments sur lequel pourront porter des contrats
         d'assemblage.

         </xs:documentation>
      </xs:annotation>

      <xs:complexType>
         <xs:sequence>
            <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
         </xs:sequence>
         <xs:attribute name="name" type="xs:NCName" use="required"/>
         <xs:attribute name="type" type="xs:QName" use="required"/>
      </xs:complexType>

   </xs:element>

<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <xs:element name="propertyRef">

      <xs:annotation>
         <xs:documentation xml:lang="fr">

         L'élément "propertyRef" permet de faire une référence vers la spécification d'une
         propriété. Cet élément est utilisé dans le cadre de la spécification d'un composant
         ou d'un port.

         </xs:documentation>
      </xs:annotation>

      <xs:complexType  mixed="true">
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:any processContents="lax"/>
         </xs:sequence>
         <xs:attribute name="ref" type="xs:QName" use="required"/>
      </xs:complexType>

   </xs:element>


<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
  <xs:simpleType name="InterfaceDirectionType">

   <xs:annotation>
         <xs:documentation xml:lang="fr">

         Le type "InterfaceDirectionType" définit la direction d'une interface attribuée à un
         port.

         </xs:documentation>
      </xs:annotation>

      <xs:restriction base="xs:string">
         <xs:enumeration value="IN"/>
         <xs:enumeration value="OUT"/>
      </xs:restriction>

   </xs:simpleType>
```

```
   <!---->

   <xs:element name="interface">

      <xs:annotation>
         <xs:documentation xml:lang="fr">

         L'élément "interface" permet, dans la définition d'un port, de faire référence à une
         description d'interface WSDL. Cette référence permet de préciser la direction que cette
         interface adopte au sein du port. Une interface de direction IN décrit un ensemble
         d'opérations en entrée, c'est à dire accessibles aux composants connectés à ce port. Une
         interface de direction OUT décrit un ensemble d'opérations en sortie, c'est à dire
         utilisées par le composants propriétaire du port pour interagir avec les composants
         connectés à ce port.

         Ainsi, dans le contexte d'une description EthyleneXML, les interfaces sont toujours
         décrites dans les termes d'une offre de service. Les opérations des interfaces WSDL
         doivent donc seulement utiliser les patrons d'opération "In-Out" et "In-Only".

         </xs:documentation>
      </xs:annotation>

      <xs:complexType>
         <xs:attribute name="ref" type="xs:QName" use="required"/>
         <xs:attribute name="direction" type="InterfaceDirectionType" use="required"/>
      </xs:complexType>

   </xs:element>


<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <xs:simpleType name="PortCommunicationMode">

    <xs:annotation>
         <xs:documentation xml:lang="fr">

         Le type "PortCommunicationMode" définit le mode de communication requis par un port.

         </xs:documentation>
      </xs:annotation>

      <xs:restriction base="xs:string">
         <xs:enumeration value="SYNC"/>
         <xs:enumeration value="ASYNC"/>
      </xs:restriction>

   </xs:simpleType>

  <!---->

   <xs:element name="port">

      <xs:annotation>
         <xs:documentation xml:lang="fr">

         L'élément "port" décrit un port de composant en termes d'interfaces. Il doit toujours
         être décrit comme une offre de service. L'attribut "mode" permet de précisé le type de
         communication nécessaire au port. Si au moins une opération d'une des interfaces le
         constituant est une opération synchrone, alors l'attribut "mode" du port doit valoir
         "SYNC". Si toute les opérations de toutes les interfaces constituant le port sont
         asynchrones, alors l'attribut "mode" du port doit valoir "ASYNC".

         </xs:documentation>
      </xs:annotation>

      <xs:complexType>
         <xs:sequence>
            <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="interface" maxOccurs="unbounded"/>
            <xs:element ref="propertyRef" minOccurs="0" maxOccurs="unbounded"/>
         </xs:sequence>
         <xs:attribute name="name" type="xs:NCName" use="required"/>
         <xs:attribute name="mode" type="PortCommunicationMode" use="required"/>
      </xs:complexType>

   </xs:element>
```

```
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <xs:simpleType name="PortDirectionType">

      <xs:annotation>
         <xs:documentation xml:lang="fr">

         Le type "PortDirectionType" définit la direction d'un port attribué à un composant.

         </xs:documentation>
      </xs:annotation>

      <xs:restriction base="xs:string">
         <xs:enumeration value="PROVIDING"/>
         <xs:enumeration value="USING"/>
      </xs:restriction>

   </xs:simpleType>

   <!---->

   <xs:element name="portRef">

      <xs:annotation>
         <xs:documentation xml:lang="fr">

         L'élément "portRef" permet de faire une référence vers la spécification d'un port
         Cet élément est utilisé dans le cadre de la spécification d'un composant. L'attribut
         "direction" doit contenir la valeur "PROVIDING" si ce port est fourni par le composant ou
         "USING" si ce port est utilisé par le composant. Un port "fourni" correspond à une offre
         de service. Ainsi, le(s) composant(s) distant(s) disposeront des interfaces IN pour
         contacter le composant fournisseur. Le composant fournisseur disposera des interfaces OUT
         pour contacter les composants distants. À l'inverse, un port "utilisé" correspond à
         l'utilisation d'un service par le composant déclarant. Ainsi, les interfaces IN sont à sa
         disposition pour contacter les composant(s) fournisseur(s) distant(s) et les interfaces
         OUT permettent au(x) composant(s) distant(s) de contacter le composant utilisateur.

         </xs:documentation>
      </xs:annotation>

      <xs:complexType>
         <xs:sequence>
            <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
         </xs:sequence>
         <xs:attribute name="ref" type="xs:QName" use="required"/>
         <xs:attribute name="direction" type="PortDirectionType" use="required"/>
      </xs:complexType>
   </xs:element>


<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <xs:element name="content">
      <xs:complexType mixed="true">
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:any processContents="lax"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="component">

      <xs:annotation>
         <xs:documentation xml:lang="fr">

         L'élément "component" spécifie un composant logiciel. Cette spécification consiste à lui
         attribuer un nom au moyen de l'attribut "name", puis de le décrire en termes de
         propriétés exposées et de ports fournis ou utilisé.

         </xs:documentation>
      </xs:annotation>

      <xs:complexType>
         <xs:sequence>
            <xs:element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="propertyRef" minOccurs="0" maxOccurs="unbounded"/>
```

```
            <xs:element ref="portRef" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="content" minOccurs="0"/>
         </xs:sequence>
         <xs:attribute name="name" type="xs:NCName" use="required"/>
      </xs:complexType>

   </xs:element>

<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <xs:element name="appinfo">
      <xs:complexType mixed="true">
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:any processContents="lax"/>
         </xs:sequence>
         <xs:attribute name="source" type="xs:anyURI"/>
         <xs:anyAttribute namespace="##other" processContents="lax"/>
      </xs:complexType>
   </xs:element>


   <xs:element name="documentation">
      <xs:complexType mixed="true">
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:any processContents="lax"/>
         </xs:sequence>
         <xs:attribute name="source" type="xs:anyURI"/>
         <xs:attribute ref="xml:lang"/>
         <xs:anyAttribute namespace="##other" processContents="lax"/>
      </xs:complexType>
   </xs:element>


   <xs:element name="annotation">
      <xs:complexType>
         <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="appinfo"/>
            <xs:element ref="documentation"/>
         </xs:choice>
         <xs:attribute name="id" type="xs:ID"/>
      </xs:complexType>
   </xs:element>

</xs:schema>
```

2. Spécifications des ports des composants mandataires utilisé par MARI

2.1 data/mcomp10d.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
      xmlns:xs=                                    "http://www.w3.org/2001/XMLSchema"
      xmlns:xsi=                                "http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation=     "http://www.w3.org/2001/XMLSchema
                                                          http://www.w3.org/2001/XMLSchema.xsd"
      xmlns:mcomp10d=                "http://iihm.imag.fr/namespaces/mcomponents/1.0/data"
      targetNamespace=        "http://iihm.imag.fr/namespaces/mcomponents/1.0/data">

   <xs:complexType name="ConceptSet">
      <xs:sequence>
         <xs:element name="concepts" type="xs:string"/>
      </xs:sequence>
   </xs:complexType>

   <xs:complexType name="ConceptValueSet">
      <xs:sequence>
         <xs:element name="tasks" type="mcomp10d:TaskVector"/>
      </xs:sequence>
   </xs:complexType>

   <xs:complexType name="Task">
      <xs:sequence>
         <xs:element name="name" type="xs:string"/>
         <xs:element name="values" type="xs:string"/>
      </xs:sequence>
   </xs:complexType>

   <xs:complexType name="TaskVector">
      <xs:sequence>
         <xs:element name="count" type="xs:integer"/>
         <xs:element name="task" type="mcomp10d:Task"/>
      </xs:sequence>
   </xs:complexType>

   <xs:complexType name="UserTask">
      <xs:sequence>
         <xs:element name="annotation" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                       <xs:attribute name="name" type="xs:string" use="required"/>
                  </xs:extension>
               </xs:simpleContent>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
   </xs:complexType>

   <xs:complexType name="UserTaskList">
      <xs:sequence>
         <xs:element name="task" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>

   <xs:element name="action" type="xs:boolean"/>

   <xs:element name="conceptSet" type="mcomp10d:ConceptSet"/>

   <xs:element name="conceptValueSet" type="mcomp10d:ConceptValueSet"/>

   <xs:element name="taskList" type="xs:string"/>
</xs:schema>
```

2.2 service/mcomp10s.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:description
    xmlns:wsdl=                                "http://www.w3.org/2006/01/wsdl"
    xmlns:xsi=                                 "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=       "http://www.w3.org/2006/01/wsdl

http://www.w3.org/2006/01/wsdl/wsdl20.xsd"
    xmlns:mcomp10d=                   "http://iihm.imag.fr/namespaces/mcomponents/1.0/data"
    xmlns:grl10d=                     "http://iihm.imag.fr/namespaces/general/1.0/data"
    targetNamespace=                  "http://iihm.imag.fr/namespaces/mcomponents/1.0/service">

   <wsdl:interface name="FunctionalCoreAdapter">

      <wsdl:operation name="setConceptValues"
                                             pattern="http://www.w3.org/2006/01/wsdl/in-only">
         <wsdl:input messageLabel="In" element="mcomp10d:conceptValueSet"/>
      </wsdl:operation>

      <wsdl:operation name="getConceptValues"
                                             pattern="http://www.w3.org/2006/01/wsdl/in-out">
         <wsdl:input messageLabel="In" element="mcomp10d:conceptSet"/>
         <wsdl:output messageLabel="Out" element="mcomp10d:conceptValueSet"/>
      </wsdl:operation>

      <wsdl:operation name="doAction"
                                             pattern="http://www.w3.org/2006/01/wsdl/in-only">
         <wsdl:input messageLabel="In" element="mcomp10d:action"/>
      </wsdl:operation>

   </wsdl:interface>



   <wsdl:interface name="UiControl">

      <wsdl:operation name="setActiveTasks"
                                             pattern="http://www.w3.org/2006/01/wsdl/in-only">
         <wsdl:input messageLabel="In" element="mcomp10d:taskList"/>
      </wsdl:operation>

      <wsdl:operation name="setEnableTasks"
                                             pattern="http://www.w3.org/2006/01/wsdl/in-only">
         <wsdl:input messageLabel="In" element="mcomp10d:taskList"/>
      </wsdl:operation>

      <wsdl:operation name="setDisableTasks"
                                             pattern="http://www.w3.org/2006/01/wsdl/in-only">
         <wsdl:input messageLabel="In" element="mcomp10d:taskList"/>
      </wsdl:operation>

      <wsdl:operation name="updateConceptValues"
                                             pattern="http://www.w3.org/2006/01/wsdl/in-out">
         <wsdl:input messageLabel="In" element="mcomp10d:conceptValueSet"/>
         <wsdl:output messageLabel="Out" element="grl10d:booleanAck"/>
      </wsdl:operation>

      <wsdl:operation name="getConceptValues"
                                             pattern="http://www.w3.org/2006/01/wsdl/in-out">
         <wsdl:input messageLabel="In" element="mcomp10d:conceptSet"/>
         <wsdl:output messageLabel="Out" element="mcomp10d:conceptValueSet"/>
      </wsdl:operation>

   </wsdl:interface>



   <wsdl:interface name="UiNotification">

      <wsdl:operation name="conceptsChange"
                                             pattern="http://www.w3.org/2006/01/wsdl/in-only">
         <wsdl:input messageLabel="In" element="mcomp10d:conceptValueSet"/>
      </wsdl:operation>
```

```
      <wsdl:operation name="userAction"
                                          pattern="http://www.w3.org/2006/01/wsdl/in-only">
         <wsdl:input messageLabel="In" element="mcomp10d:taskList"/>
      </wsdl:operation>
   </wsdl:interface>



   <wsdl:interface name="FunctionalCoreAdapterNotification">
      <wsdl:operation name="conceptUpdate"
                                          pattern="http://www.w3.org/2006/01/wsdl/in-only">
         <wsdl:input messageLabel="In" element="mcomp10d:conceptValueSet"/>
      </wsdl:operation>

   </wsdl:interface>

</wsdl:description>
```

2.3 mcomp10.c2h4.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<c2h4:description
    xmlns:c2h4=                              "http://iihm.imag.fr/namespaces/ethylenexml/1.0"
    xmlns:xs=                                "http://www.w3.org/2001/XMLSchema"
    xmlns:xsi=                               "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=       "http://iihm.imag.fr/namespaces/ethylenexml/1.0

http://iihm.imag.fr/namespaces/ethylenexml/1.0/ethylenexml.xsd"
    xmlns:mcomp10s=                  "http://iihm.imag.fr/namespaces/mcomponents/1.0/service"
    xmlns:mcomp10d=                  "http://iihm.imag.fr/namespaces/mcomponents/1.0/data"
    targetNamespace=                 "http://iihm.imag.fr/namespaces/mcomponents/1.0">

   <c2h4:types>
      <c2h4:import
                namespace="http://iihm.imag.fr/namespaces/mcomponents/1.0/data"
                location="http://iihm.imag.fr/namespaces/mcomponents/1.0/data/mcomp10d.xsd"/>
   </c2h4:types>

   <c2h4:interfaces>
      <c2h4:import
            namespace="http://iihm.imag.fr/namespaces/mcomponents/1.0/service"
            location="http://iihm.imag.fr/namespaces/mcomponents/1.0/service/mcomp10s.wsdl"/>
   </c2h4:interfaces>

   <c2h4:properties>
      <c2h4:property name="supportedUserTasks" type="mcomp10d:UserTaskList"/>
      <c2h4:property name="taskModelUri" type="xs:string"/>
      <c2h4:property name="adapter" type="xs:string"/>
      <c2h4:property name="conceptSet" type="mcomp10d:ConceptSet"/>
   </c2h4:properties>

   <c2h4:ports>

      <c2h4:port name="DialogController" mode="SYNC">
         <c2h4:interface ref="mcomp10s:UiControl" direction="IN"/>
         <c2h4:interface ref="mcomp10s:UiNotification" direction="OUT"/>
      </c2h4:port>

      <c2h4:port name="FunctionalCoreAdapter" mode="SYNC">
         <c2h4:interface ref="mcomp10s:FunctionalCoreAdapter" direction="IN"/>
         <c2h4:interface ref="mcomp10s:FunctionalCoreAdapterNotification" direction="OUT"/>
      </c2h4:port>

   </c2h4:ports>

</c2h4:description>
```

3. Spécifications des composants du démonstrateur PhotoBrowser

3.1 PhotoShuffler.c2h4.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<c2h4:description
    xmlns:c2h4=               "http://iihm.imag.fr/namespaces/ethylenexml/1.0"
    xmlns:xsi=                "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=       "http://iihm.imag.fr/namespaces/ethylenexml/1.0
                        http://iihm.imag.fr/namespaces/ethylenexml/1.0/ethylenexml.xsd"
    xmlns:demo10=             "http://iihm.imag.fr/namespaces/demo/1.0"
    xmlns:grl10=              "http://iihm.imag.fr/namespaces/general/1.0"
    xmlns:mcomp10=                    "http://iihm.imag.fr/namespaces/mcomponents/1.0"
    targetNamespace=          "http://iihm.imag.fr/namespaces/demo/1.0">

   <c2h4:component name="PhotoShuffler">

      <!-- Définition des propriétés exposées par le composant -->
      <c2h4:propertyRef ref="grl10:nbInstancesMax">1</c2h4:propertyRef>

      <c2h4:propertyRef ref="mcomp10:supportedUserTasks">
         <task name="BrowseImages">
            <annotation name="fullysupported">no</annotation>
         </task>
         <task name="ConsultAllImages">
            <annotation name="deactivable">no</annotation>
         </task>
         <task name="ConsultOneImage">
            <annotation name="fullysupported">no</annotation>
         </task>
         <task name="SelectImage">
            <annotation name="subtype">choice 1/n</annotation>
            <annotation name="interactionTechnic">pointing</annotation>
         </task>
         <task name="ConsultImageDetails">
            <annotation name="fullysupported">no</annotation>
         </task>
         <task name="ConsultImageInfo"/>
         <task name="ConsultSelectedImage"/>
      </c2h4:propertyRef>

      <c2h4:propertyRef ref="mcomp10:taskModelUri">PhotoBrowserTaskModel</c2h4:propertyRef>

      <!-- Définition des ports fournis et utilisés par le composant -->
      <c2h4:portRef ref="demo10:PhotoShufflerService" direction="PROVIDING"/>

   </c2h4:component>

</c2h4:description>
```

3.2 PhotoShufflerAdapter.c2h4.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<c2h4:description
    xmlns:c2h4="http://iihm.imag.fr/namespaces/ethylenexml/1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://iihm.imag.fr/namespaces/ethylenexml/1.0
http://iihm.imag.fr/namespaces/ethylenexml/1.0/ethylenexml.xsd"
    targetNamespace="http://iihm.imag.fr/namespaces/demo/1.0"
    xmlns:mcomp10="http://iihm.imag.fr/namespaces/mcomponents/1.0/service"
    xmlns:mcomp10d="http://iihm.imag.fr/namespaces/mcomponents/1.0/data"
    xmlns:pss10="http://iihm.imag.fr/namespaces/photoshuffler/1.0/service"
    xmlns:demo10="http://iihm.imag.fr/namespaces/demo/1.0">

   <c2h4:component name="PhotoShufflerAdapter">
      <c2h4:propertyRef ref="mcomp10d:adapter">
         http://iihm.imag.fr/namespaces/photoshuffler/1.0/PhotoShuffler
      </c2h4:propertyRef>
      <c2h4:portRef ref="mcomp10:DialogController" direction="PROVIDING"/>
      <c2h4:portRef ref="pss10:PhotoShufflerService" direction="USING"/>
   </c2h4:component>
</c2h4:description>
```

3.3 ImageNameList.c2h4.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<c2h4:description
    xmlns:c2h4="http://iihm.imag.fr/namespaces/ethylenexml/1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://iihm.imag.fr/namespaces/ethylenexml/1.0
http://iihm.imag.fr/namespaces/ethylenexml/1.0/ethylenexml.xsd"
    targetNamespace="http://iihm.imag.fr/namespaces/demo/1.0"
    xmlns:mcomp10="http://iihm.imag.fr/namespaces/mcomponents/1.0/service"
    xmlns:mcomp10d="http://iihm.imag.fr/namespaces/mcomponents/1.0/data"
    xmlns:pss10="http://iihm.imag.fr/namespaces/photoshuffler/1.0/service"
    xmlns:demo10="http://iihm.imag.fr/namespaces/demo/1.0">

   <c2h4:component name="ImageNameList">
      <c2h4:propertyRef ref="mcomp10d:supportedUserTasks">
         <task name="SelectImage">
            <annotation name="subtype">choice 1/n</annotation>
            <annotation name="interactionTechnic">
               pointing, bidirectional-sequential-access
            </annotation>
            <annotation name="deactivable">no</annotation>
         </task>
         <task name="ConsultImageDetails">
            <annotation name="deactivable">no</annotation>
            <annotation name="fullysupported">no</annotation>
         </task>
         <task name="ConsultSelectedImageName">
            <annotation name="deactivable">no</annotation>
         </task>
      </c2h4:propertyRef>
      <c2h4:propertyRef ref="mcomp10d:taskModelUri">PhotoBrowserTaskModel</c2h4:propertyRef>
      <c2h4:portRef ref="mcomp10:DialogController" direction="PROVIDING"/>
   </c2h4:component>
</c2h4:description>
```

3.4 WebSlideShow.c2h4.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<c2h4:description
    xmlns:c2h4=               "http://iihm.imag.fr/namespaces/ethylenexml/1.0"
    xmlns:xsi=                "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=       "http://iihm.imag.fr/namespaces/ethylenexml/1.0
                    http://iihm.imag.fr/namespaces/ethylenexml/1.0/ethylenexml.xsd"
    xmlns:mcomp10=                          "http://iihm.imag.fr/namespaces/mcomponents/1.0"
    xmlns:grl10=                     "http://iihm.imag.fr/namespaces/general/1.0"
    targetNamespace=                 "http://iihm.imag.fr/namespaces/demo/1.0">

   <c2h4:component name="WebSlideShow">

      <c2h4:propertyRef ref="mcomp10:supportedUserTasks">
         <task name="SelectImage">
            <annotation name="subtype">choice 1/n</annotation>
            <annotation name="interactionTechnic">
               bidirectional-sequential-access
            </annotation>
            <annotation name="deactivable">no</annotation>
         </task>
         <task name="ConsultImageDetails">
            <annotation name="deactivable">no</annotation>
            <annotation name="fullysupported">no</annotation>
         </task>
         <task name="ConsultSelectedImage">
            <annotation name="deactivable">no</annotation>
         </task>
      </c2h4:propertyRef>

      <c2h4:propertyRef ref="mcomp10:taskModelUri">
        PhotoBrowserTaskModel
        </c2h4:propertyRef>

      <c2h4:propertyRef ref="grl10:needRenderer">html-ajax</c2h4:propertyRef>
```

```
      <c2h4:propertyRef ref="grl10:url">
        http://bayonet.imag.fr/~lionel/www/webslideshow/webslideshow.php
        </c2h4:propertyRef>

      <c2h4:portRef ref="mcomp10:DialogController" direction="PROVIDING"/>

   </c2h4:component>

</c2h4:description>
```

3.5 Safari.c2h4.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<c2h4:description
    xmlns:c2h4="http://iihm.imag.fr/namespaces/ethylenexml/1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://iihm.imag.fr/namespaces/ethylenexml/1.0
http://iihm.imag.fr/namespaces/ethylenexml/1.0/ethylenexml.xsd"
    xmlns:demo10="http://iihm.imag.fr/namespaces/demo/1.0"
    xmlns:grl10="http://iihm.imag.fr/namespaces/general/1.0"
    targetNamespace="http://iihm.imag.fr/namespaces/demo/1.0">

   <c2h4:component name="Safari">

        <c2h4:propertyRef ref="grl10:renderer">html-ajax</c2h4:propertyRef>

        <c2h4:portRef ref="demo10:WebBrowser" direction="PROVIDING"/>

   </c2h4:component>
</c2h4:description>
```

3.6 PhotoBrowserFC.c2h4.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<c2h4:description
    xmlns:c2h4="http://iihm.imag.fr/namespaces/ethylenexml/1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://iihm.imag.fr/namespaces/ethylenexml/1.0
http://iihm.imag.fr/namespaces/ethylenexml/1.0/ethylenexml.xsd"
    targetNamespace="http://iihm.imag.fr/namespaces/demo/1.0"
    xmlns:mcomp10="http://iihm.imag.fr/namespaces/mcomponents/1.0/service"
    xmlns:mcomp10d="http://iihm.imag.fr/namespaces/mcomponents/1.0/data"
    xmlns:demo10="http://iihm.imag.fr/namespaces/demo/1.0">

   <c2h4:component name="PhotoBrowserFC">
      <c2h4:propertyRef ref="mcomp10d:conceptSet">
         NF:listOfImages NF:selectedImageName NF:selectedImageInfo
      </c2h4:propertyRef>
      <c2h4:portRef ref="mcomp10:FunctionalCoreAdapter" direction="PROVIDING"/>
   </c2h4:component>
</c2h4:description>
```

Résumé

Cette thèse s'inscrit dans le domaine de l'interaction homme-machine (IHM) et s'intéresse à l'adaptation dynamique des systèmes interactifs dans le cadre de l'informatique ubiquitaire. Dans ce cadre, les interfaces utilisateurs doivent devenir plastiques, c'est-à-dire être capables de s'adapter ou d'être adaptées au contexte de l'interaction tout en préservant leur utilisabilité. L'objectif de cette thèse est de comprendre la problématique de la plasticité des IHM à l'exécution et de la traiter sous l'angle du génie logiciel. L'étude de cette problématique montre qu'un système interactif plastique est un logiciel réparti, reconfigurable dynamiquement et constitué d'entités logicielles hétérogènes. Or, aucune proposition de l'état de l'art des systèmes interactifs plastiques ne couvre totalement l'espace problème de la plasticité à l'exécution. De même, aucune solution du génie logiciel pour la construction de logiciels répartis reconfigurables dynamiquement ne prend en compte les spécificités des IHM plastiques. Cette thèse propose une décomposition logicielle de référence qui identifie l'ensemble des fonctions nécessaires à la plasticité des systèmes interactifs, ainsi qu'Ethylene, un modèle à composants dynamiques issu d'une combinaison des approches à composants et à services. Ethylene est conçu pour répondre aux spécificités de l'IHM, et permet de constituer dynamiquement des assemblages reconfigurables de composants IHM de nature hétérogène. Enfin, un langage XML et un cadre de développement incarnent le modèle Ethylene sur un plan technique.

Mots-Clefs :
Interaction Homme-Machine (IHM), Interface Utilisateur plastique, adaptation dynamique d'interface homme-machine, adaptation à l'exécution, composant orienté service, assemblage hétérogène de composants logiciels.

Abstract

This dissertation addresses the problem of run-time adaptation of user interfaces (UI) for ubiquitous computing (Ubicomp). In Ubicomp, user-interfaces have to be plastic: they must be able to adapt, or to be adapted, to the context of use while preserving their usability. The goal of this thesis is to understand the problem of UI plasticity at runtime and to explore UI plasticity from the software engineering perspective. This study shows that plastic interactive systems are dynamically reconfigurable distributed software built from heterogeneous software entities. We observe that the state of the art for adaptable-at-runtime interactive systems fails to provide solutions that cover the whole problem space of UI plasticity. Similarly, main-stream software engineering for distributed software adaptation does not cover the particular requirements of UI plasticity. The contribution of this thesis is three-fold: (1) A functional decomposition that serves as a reference model for the development of plastic UI's. (2) Ethylene, a dynamic component oriented model that combines the component approach with that of the service model to address HCI specificities as well as the dynamic assembly of heterogeneous components. (3) Ethylene-XML and a development framework to support the technical development of plastic UIs.

Keywords:
Human-Computer Interaction (HCI), plastic user interface, dynamic adaptation of user interfaces, run-time adaptation, service-oriented component, heterogeneous assembly of software components.