COOK: Réarchitecturisation des applications industrielles à objets

Prof. Stéphane Ducasse (Porteur du projet),
Ilham Alloui, Sorana Cimpan, Hervé Verjus et Marc-Philippe Huget
Language and Software Evolution Group — LTI

LISTIC – Université de Savoie

Résumé. Ce projet de recherche s'inscrit dans le contexte de la ré-ingénierie et l'évolution d'applications industrielles à objets. La ré-ingénierie hérite des problèmes complexes liés à la maintenance (compréhension, analyses, transformations de programmes). A cela s'ajoute une complexité due à la liaison tardive, la définition incrémentale de programmes telle que les frameworks et lignes de produits (product lines). Partant de notre expérience, ce projet s'attaque aux problèmes de l'évolution de ces applications par la prise en compte explicite de la notion d'architecture, notion souvent enfouie dans les millions de lignes de code de telles applications.

Les résultats attendus sont : la *modélisation* des architectures au sein d'un environnement de ré-ingénierie, l'*identification* par l'extraction et explicitation des architectures logiciels du code des applications sous analyse, l'*analyse* de la violation d'architecture, *l'analyse de l'évolution* d'architectures et la *refactorisation* de code dirigée par l'architecture. Les résultats de COOK seront validés sur des applications industrielles.

Mots Clefs : Génie logiciel, Maintenance, Architecture, Ré-ingénierie, Rétro-conception, Evolution, Modèles, Frameworks, Patterns, Analyse

1 Introduction

Une grande partie de l'effort de développement des logiciels industriels de grande taille et longue durée de vie est passée dans leur maintenance et évolution [29, 4]. Ce document présente le projet de recherche COOK dont le but est la prise en compte des architectures logicielles (c-à-d, extraction depuis le code source, analyse, etc.) comme élément capital pour l'aide à l'évolution des applications industrielles à objets.

COOK se décompose autour de cinq points logiquement articulés et dont les résultats vont s'échelonner sur une période de trois ou quatre années. Les résultats de ce projet seront évalués sur des applications industrielles : d'une part des projets open-source et d'autre part des logiciels développés par la société Harman-Becker avec qui nous sommes en contact.

– *Modélisation de la notion d'architecture*. La première étape consiste à introduire cette notion dans Moose, l'environnement de réingénierie, développé par le porteur du projet [8]. Ceci implique de pouvoir modéliser une architecture et les éléments qui la constituent en les reliant aux entités représentant le code de l'application.

– *Approches d'identification d'architectures des applications à objets*. Identifier l'architecture à partir du code d'une application ou de son exécution est une information vitale pour comprendre et maintenir une application. Nous abordons donc l'extraction d'architecture du code

des applications sous analyse afin de pouvoir raisonner sur des modèles d'architectures de plus haut niveau d'abstraction. Il s'agit d'exprimer et de proposer des outils pour la visualisation, la compréhension des architectures et leur extraction à partir du code.

– *Identification de violation d'architecture*. Les applications industrielles ne respectent pas souvent les architectures qui les décrivent. De même, l'usure du temps crée une érosion des architectures (*architectural drift*). Identifier de telles violations est primordial pour éviter des problèmes importants accompagnant l'évolution de telles applications. Notre objectif est de fournir des outils pour aider à identifier de telles violations.

– *Analyse de l'évolution d'architectures*. La prise en compte des changements de l'architecture d'un grand logiciel est une information qualitative importante. Savoir qu'un élément architectural est stable ou change de manière fréquente est aussi une information qualitative importante. Nous proposons des analyses de l'évolution des architectures et des aides pour sa compréhension.

– *Remodularisation et définition de refactorings architecturaux*. Etre capable de transformer une architecture étape par étape ou de remodulariser une application existante est une tâche complexe mais importante. Nous comptons développer des aides à la remodularisation en utilisant des algorithmes de groupement (*clustering algorithms*). De plus, les refactorings, transformations de code avec conservation du comportement existent mais avec une granularité fine au sein de classes. Nous voulons définir des refactorings prenant en compte l'architecture de l'application (donc à un niveau plus abstrait).

Le plan de ce projet est le suivant : nous expliquons en section 2 pourquoi la maintenance est en fait une phase de évolution des logiciels, moment majeur dans la vie d'un logiciel. En section 3 nous rappelons brièvement les travaux relatifs aux projets effectués par les membres de l'équipe. Section 4 présente de manière précise les objectifs du projet. Nous montrons que ce projet construit autour d'un existant solide et qu'il s'inscrit dans une collaboration avec des équipes de recherches européennes et dans la réorientation de l'axe de recherche génie logiciel au sein du LISTIC, le laboratoire hôte. Finalement en section 5, nous mettons en perspective ce projet dans un contexte international de recherche c-à-d réseaux de recherches et groupes de travail. Nous montrons en particulier que la France est sous-représentée dans le contexte du génie logiciel et en particulier celui de la maintenance d'applications : 0.95% d'articles publiés sur le sujet contre 31% pour les USA et 48% pour l'Europe.

2 Contexte : L'inéluctable évolution des applications

Bien que les logiciels soient devenus un des tenants de notre industrie, leur développement reste une tâche parsemée d'embûches. Même les projets ayant du succès doivent faire face à ce que Parnas appelle *software aging*, le vieillissement du logiciel [26]. Le charactère chronique des problèmes de développement a amené Pressman à préférer l'expression maladie chronique (*chronique affliction*) au lieu de crise du logiciel (*software crisis*) [27]. Plusieurs facteurs inhérents au développement du logiciel mènent à cette situation : la complexité des domaines et des tâches modélisées, le besoin *constant* d'adaptation et de changement, les problèmes liés à la gestion de projet et les relations humaines et les difficultés à comprendre les exigences du client.

Quelques faits peuvent éclairer le lecteur sur la nécessité de changer (qui est le contexte de ce projet). La maintenance logicielle est le nom donné au processus permettant de changer un logiciel après qu'il ait été livré aux clients. Sommerville [29], en faisant référence à des études conduites

dans les années 80 [19, 21], mentionne que de grandes organisations passent au moins 50% de leur développement total dans la maintenance de logiciels existants. McKee [21] mentionne que l'effort de maintenance se situe entre 65% and 75% de l'effort total. Il n'est plus à démontrer que la phase de maintenance est une des phases de développement les plus coûteuses. Cependant, le terme maintenance cache une toute autre réalité que la parfois simple correction de bugs.

Une analyse plus fine montre que la maintenance ne se limite pas à des activités de maintenance [19], [24]. Sommerville catégorise les activités de maintenance comme suit[1][29] : *Maintenance correctrice* (17%) c-à-d, fixer des bugs, *Maintenance adaptative* (18%) c-à-d, adapter le logiciel à de nouvelles plate-formes, systèmes d'exploitation, etc. et *maintenance de perfection* (perfective) (65%) c-à-d, implantation de nouvelles fonctionalités. De ces données il apparaît clairement que la maintenance, et donc *la plupart du coût de développement, est passée à faire évoluer les logiciels*.

Les lois de l'évolution du logiciel validées empiriquement par Lehman et Belady offrent une interprétation de cette situation : toute application utile pour ses utilisateurs sera forcée d'évoluer pour faire face à de nouvelles contraintes et demandes et cette évolution entraîne une complexité accrue.

De nombreux travaux sont actuellement menés pour aider à l'évolution des logiciels. D'une part on trouve des travaux situés a des niveaux très proches du code de l'application, notamment dans la réingénierie du logiciel. D'autre part dans le domaine des architectures logicielles, on trouve des travaux portant sur la prise en compte de l'évolution à un niveau plus abstrait avec les langages de description d'architectures. Cependant, les processus de développement abordés sont essentiellement déductifs, l'architecture étant réprésentée au début à un haut niveau d'abstraction, et ensuite raffinée jusqu'à la génération du code. Le projet COOK se propose de marier ces deux approches, en introduisant la notion d'architecture dans un processus de réingénierie des logiciels existants. Ceci apporte des bénéfices dans les deux domaines. D'une part les travaux existants dans la réingénierie sont enrichis avec la possibilité de raisonner sur l'évolution du logiciel à de plus hauts niveaux d'abstraction. D'autre part, les travaux existants dans les architectures logicielles, notamment ceux tournant autour des développements centrés architecture, verront leur application étendue à des formes de développement non déductives.

3 Compétences de l'équipe

Le projet COOK est la convergence de deux activités d'une part la ré-ingénierie et d'autre part la définition de langages architecturaux. COOK se fonde sur la complémentarité des participants du projet.

Contributions en ré-ingénierie. COOK s'inscrit dans la lignée de travaux qui ont débuté dans le cadre du projet Esprit FAMOOS[2] et du projet RECAST[3]sur l'évolution des applications à objets . Les contributions sont d'un côté la publication d'articles et de thèses, et d'un autre côté, l'implémentation de prototypes de qualité qui ont été validés sur des applications industrielles et utilisés par plusieurs universités (Berne, Lugano, Bruxelles). Nous montrons les éléments relatifs à COOK.

– FAMIX : *un méta-modèle indépendant des langages* ([5]). Comme nous devions analyser différents langages Smalltalk, C++, Java, nous avons défini un méta-modèle permettant de représenter les

1. les pourcentages sont relatifs à l'effort total de maintenance

2. FAMOOS (A Framework based Approach for Mastering Objet-Oriented Systems-IST-1996-1999). FAMOOS était précurseur et un des premiers projets de recherches à traiter de la ré-ingénierie dans le contexte spécifique des applications à objets

3. Swiss National Fund 2002-2005

aspects centraux des langages à objets et d'étendre ce méta-modèle.

– MOOSE : *Implantation d'une plate-forme de ré-ingénierie* ([30], [12] [8]). Nous avons construit une plate-forme permettant d'extraire, charger, stocker et d'analyser plusieurs modèles simultanément, de calculer des métriques, de définir des analyseurs spécialisés. MOOSE est utilisé par plusieurs chercheurs des Universités de Bruxelles, Berne et Lugano.

– CODECRAWLER : *compréhension de grands systèmes* ([3], [9], [18], [11], [10]). Nous avons développé une approche pour permettre de comprendre de grands systèmes. L'outil développé, CODECRAWLER, se fonde sur FAMIX et MOOSE. CODECRAWLER a été utilisé lors de l'analyse d'applications industrielles.

– VAN : *Compréhension de l'évolution de logiciels* ([14], [15] [28]). Nous avons étendu Moose pour la prise en compte de l'évolution des logiciels.

Contributions en Architectures Logicielle. COOK se situe dans la continuité des travaux menés au sein des projets ARCHWARE[4] et PIE[5] pour lesquels il est possible d'emprunter certaines approches (centrée architecture, Langages de Description d'Architectures évolutives, stratégies et outils d'évolution) liées à la prise en compte de l'évolution.

Les contributions sont d'un côté par la publication d'articles et de thèses, et d'un autre côté, par la définition de langages et de prototypes logiciels. Nous ne nous citons que les éléments relatifs à COOK.

– ARCHWARE-ADL : *un langage de description d'architectures évolutives* ([25, 2]). Ce langage basé sur le π-calcul est un langage structuré en plusieurs couches. Cette structuration est articluée autour d'un noyau π-ADL (qui propose un calcul pour les composants logiciels) et un mecanisme d'extension. Il dispose de plusieurs syntaxes (textuelle concrète, XML, etc.). Il permet d'exprimer des propriétés structurelles et comportementales et est doté de mécanismes permettant la définition de proprietés (structurelles et comportementales) et de styles architecturaux.

– plusieurs prototypes dans le domaine des architectures logicielles dont (1) un animateur qui permet de représenter graphiquement la structure et le comportement d'une architecture exprimée avec le langage π-ADL et (2) un outil d'analyse de propriétés.

4 COOK

De très nombreux langages architecturaux existent et sont actuellement développés dans un contexte de recherche académique [20, 1, 22]. Ces langages bien que très riches dans la granularité ou les comportements qu'ils permettent de représenter, sont cependant peu utilisés dans l'industrie.

L'objectif de COOK n'est pas de définir un nouveau langage de description architecturale mais d'aider l'évolution applications industrielles existantes par la prise en compte de leur architecture comme élément central, c-à-d, l'extraction depuis le code source, l'analyse et la transformation de

4. ARCHWARE (ARCHitecting Evolvable softWARE, IST 5 32360, 2002-2005). ARCHWARE propose une approche centrée architecture du développement logiciel ainsi qu'un ADL basé sur le π-calcul permettant d'exprimer l'évolution dès la phase de conception.

5. PIE (Process Instance Evolution, ESPRIT IV LTR, 1998-2001)

l'architecture d'applications existantes. COOK prend en compte la notion d'architecture dans le cadre de l'évolution et l'extraction d'architecture des logiciels existants .

COOK doit permettre d'extraire, analyser, faire évoluer l'architecture réelle d'applications industrielles existantes, nous comptons adopter une approche itérative de construction de la solution par raffinements successifs [13, 31, 7, 16, 6]. L'échéancier du projet se structure autour de 5 étapes correspondant aux cinq axes de recherche principaux ; ces étapes, bien qu'ayant une suite chronologique, pourront être revisitées au besoin. Ceci nous permettra par exemple de raffiner les concepts liés à la modélisation d'architecture en fonction du retour d'expérience provenant des études concrètes.

4.1 Modélisation de la notion d'architecture

Ce premier axe consiste à introduire la notion d'architecture dans Moose, l'environnement de ré-ingénierie développé par le porteur du projet [8]. Ceci implique de pouvoir modéliser une architecture et les éléments qui la constituent en référençant les éléments du code (classes, packages....).

Ainsi, des concepts de plus haut niveau que ceux existant actuellement dans l'environnement seront introduits en tenant compte des travaux actuels dans le domaine de la description des architectures. L'adéquation du paradigme composant-connecteur, largement adopté dans les langages de description architecturales, à la modélisation des architectures d'applications à objets sera étudiée. La proposition d'autres paradigmes que celui de composant-connecteur est également envisagée.

4.2 Approches d'identification d'architectures des applications à objets

Identifier l'architecture à partir du code d'une application ou de son exécution est une information vitale pour comprendre et maintenir une application ayant plusieurs millions de lignes de code. Nous abordons donc l'extraction d'architecture du code des applications sous analyse afin de pouvoir raisonner sur des modèles d'architectures de plus haut niveau d'abstraction. Il s'agit et de proposer des outils pour (1) des outils pour l'aide à l'extraction de l'architecture depuis le code source, (2) la visualisation, la compréhension du code et des éléments d'architectures ainsi connectés. Nous abordons l'utilisation d'algorithmes de groupement (*cluster analysis*)[17] pour aider à la découverte semi-automatique d'architectures. Cet axe est lié à celui portant sur la remodularisation présenté plus loin.

4.3 Identification de la violation d'architecture

Les applications industrielles ne respectent pas souvent les architectures qui les décrivent, ce phénomène est communément appelé *architectural drift*. Identifier de telles violations est primordial pour éviter des problèmes importants accompagnant l'évolution de telles applications. En particulier, l'architecture ne pourra plus être utilisée comme support lors de la ré-ingénierie (drift conscient) ou alors elle sera utilisée alors qu'elle ne correspond plus au code (drift inconscient). Dans cet axe, nous comptons adapter le modèle de réflexion (*Reflection Model*)[23] pour extraire un mapping entre les éléments du code et l'architecture telle que pensée par le ré-ingénieur. Le langage SOUL [32] est un candidat pour exprimer des mappings complexes car il est déclaratif et permet la manipulation fine d'entités représentant le code des applications et peut s'interfacer avec l'environnement Moose.

4.4 Analyse de l'évolution d'architectures

Alors que les autres axes ne considèrent qu'une version d'une application, cet axe prend en compte son historique. Ainsi l'idée est de comprendre les changements architecturaux en analysant de mul-

tiples versions d'une application. D'une part nous voulons analyser les différences architecturales, notamment en vue de qualifier plus finement une violation d'architecture. Par exemple, une violation d'architecture dans une partie du code ne changeant jamais dans la vie d'une application est moins gênante qu'une violation relative à une partie du code sujette à des changements fréquents.

D'autre part, nous voulons prendre en compte des facteurs extérieurs au code. En particulier nous voulons analyser : la corrélation entre les bugs rapportés et les changements architecturaux, la prise en compte de nouvelles fonctionnalités et leurs impacts sur l'architecture.

4.5 Remodularisation et définition de refactorings architecturaux

Etre capable de transformer une architecture étape par étape ou de remodulariser une application existante est une tâche complexe mais souvent capitale pour la survie d'un logiciel. Ceci a par exemple été le cas chez Nokia où une grande application de gestion de centrales téléphoniques a dû être remodularisée; en effet, les clients ne voulaient pas acheter l'application dans son ensemble mais seulement certaines parties. Identifier des sous-produits et repenser une grande application nécessite alors des outils de prospection et de simulation pour comprendre l'impact de certaines actions avant qu'elles ne soient effectivement mises en œuvre par les développeurs (le what-if), et des outils pour la remodularisation et la transformation des applications. Dans cet axe, nous travaillons sur la définition :

- d'un système d'aide à la décision prenant en compte la sémantique des relations entre les éléments de code ainsi que les éléments architecturaux,
- de techniques semi-automatiques de remodularisation fondées sur des approches de groupement,
- de refactorings dirigés par l'architecture. En effet, les refactorings, transformations de code avec conservation du comportement existent mais avec une granularité fine au sein de classes. Nous voulons définir des refactorings prenant en compte l'architecture de l'application (donc à un niveau plus abstrait).

5 Enjeux internationaux et locaux

International. Ce projet se situe dans un contexte international important. La recherche sur l' évolution des logiciels est en train de se structurer en Europe. L'équipe fait partie des réseaux ERCIM Working Group on Software Evolution et ESF Release dont S. Ducasse est un des fondateurs. S. Ducasse a participé au précédent réseau Scientific Research Network Foundations of Software Evolution financé par le Fund for Scientific Research - Flanders (Belgique).

Ce projet s'inscrit dans une volonté de collaboration à long terme avec les équipes suivantes qui font partie des réseaux et avec lesquelles des contacts de longue date ont été établis : SCG de l'Université de Berne et Lugano en Suisse, les équipes PROG de la Vrije Universiteit Brussels (VUB), DeComp de l'Université Libre de Bruxelles (ULB), LORE de l'Université d'Anvers en Belgique, ainsi que l'équipe LOOSE de l'Université de Timisoara en Roumanie. Nous avons commencé à signer des contrats Erasmus avec VUB, ULB, Berne, et nous allons le faire également avec Timisoara et Lugano. Nous avons aussi soumis une demande de collaboration Egide (Germaine de Stael avec Berne et Tournesol avec ULB) autour de la prise en compte des architectures dans le cycle d'évolution des applications. L'enjeu international est d'autant plus important que la France occupe une place mineure dans le domaine de recherche du projet. En effet, nous avons calculé sur un échantillon de 8 conférences internationales (International Conference on Sowftare Maintenance 2001, 2002, 2003,

2004, International Conference on Software Engineering 2001, 2002, Conference on Software Maintenance and Reengineering 2004, Working Conference on Reverse Engineering 2002) pour un total de 522 articles, la place de la présence française : elle s'avère être de 0.95% contre 31% pour les Etats-Unis, 48% pour l'Europe (Suisse et Europe de l'Est inclues), Canada 10.9 % et 8% pour le reste du monde. Ce projet constituera le premier pas dans la création d'une équipe de renommée nationale et internationale centrée autour de l'évolution du logiciel.

Relations Industrielles. Les résultats de ce projet seront évalués sur des applications industrielles : d'une part des projets open-source et d'autre part des logiciels développés par la société Harman-Becker avec qui nous sommes en contact.

6 Conclusion

Nous avons montré que la maintenance et l'évolution des logiciels est un point crucial lors du développement durable du grands logiciels. COOK s'incrit dans cette problèmatique en introduisant la notion d'architecture comme un point central d'abstraction à partir du code existant pour la compréhension, l'analyse et la transformation de tels systèmes. Nous avons montré que ce projet résulte de la complémentarité de ces membres et correspond à une nouvelle impulsion dont l'intention finale est la création d'un pôle d'expertise nationale et internationale en maintenance, qui manque dans le paysage de recherche français.

Références

[1] R. Allen and D. Garlan. The Wright architectural specification language. CMU-CS-96-TB, School of Computer Science, Carnegie Mellon University, Pittsburgh, Sept. 1996.

[2] S. Cimpan, F. Leymonerie, and F. Oquendo. Handling dynamic behaviour in software architectures. In *Proceedings of the European Workshop on Software Architecture*. Springer Verlag, 2005.

[3] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proceedings WCRE '99 (6th Working Conference on Reverse Engineering)*. IEEE, Oct. 1999.

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[5] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[6] A. v. Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. In *Proceedings Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, pages 122–134. IEEE Computer Society Press, 2004.

[7] L. Ding and N. Medvidovic. Focus: A light-weight, incremental approach to software architecture recovery and evolution. In *Proceedings Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, pages 191–201. IEEE Computer Society Press, 2001.

[8] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a Collaborative and Extensible Reengineering Environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55 – 71. Franco Angeli, 2005.

[9] S. Ducasse and M. Lanza. Towards a methodology for the understanding of object-oriented systems. *Technique et science informatiques*, 20(4):539–566, 2001.

[10] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, 31(1):75–90, 2005.

[11] S. Ducasse, M. Lanza, and L. Ponisio. Butterflies: A visual approach to characterize packages. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*. IEEE Computer Society, 2005. To appear.

[12] S. Ducasse and S. Tichelaar. Dimensions of reengineering environment infrastructures. *International Journal on Software Maintenance: Research and Practice*, 15(5):345–373, Oct. 2003.

[13] H. Gall, R. Klosch, and R. M. T. O.-O. Re-Architecturing, editors. *Proceedings of ESEC '95*, volume 989 of *LNCS*, 1995.

[14] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes. In *Proceedings of ICSM '04 (International Conference on Software Maintenance)*, pages 40–49. IEEE Computer Society Press, 2004.

[15] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of European Conference on Software Maintenance (CSMR 2005)*, 2005.

[16] H. Gomaa and M. Hussein. Software reconfiguration patterns for dynamic evolution of software architectures. In *Proceedings Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, pages 79–88. IEEE Computer Society Press, 2004.

[17] A. Jain, M. Murty, and P. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[18] M. Lanza and S. Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, Sept. 2003.

[19] B. Lientz and B. Swanson. *Software Maintenance Management*. Addison Wesley, 1980.

[20] J. Magee, N. Dulay, and J. Kramer. Structuring parallel and distributed programs. In *Proceedings of the International Workshop on Configurable Distributed Systems*, London, Mar. 1992.

[21] J. R. McKee. Maintenance as a function of design. In *Proceedings of AFIPS National Computer Conference*, pages 187–193, 1984.

[22] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of ESEC FSE'97*, pages 60–76. ACM Press, 1997.

[23] G. C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington, 1996.

[24] J. T. Nosek and P. Palvia. Software maintenance management: changes in the last decade. *Software Maintenance: Research and Practice*, 2(3):157–174, 1990.

[25] F. Oquendo. π-adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–14, 2004.

[26] D. L. Parnas. Software Aging. In *Proceedings of ICSE '94 (International Conference on Software Engineering)*, pages 279–287. IEEE Computer Society / ACM Press, 1994.

[27] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1994.

[28] D. Raţiu, S. Ducasse, T. Gîrba, and R. Marinescu. Using history information to improve design flaws detection. In *Proceedings of CSMR 2004 (European Conference on Software Maintenance and Reengineering)*, pages 223–232, 2004.

[29] I. Sommerville. *Software Engineering*. Addison Wesley, fifth edition, 1996.

[30] S. Tichelaar, J. C. Cruz, and S. Demeyer. Design guidelines for coordination components. In J. Carroll, E. Damiani, H. Haddad, and D. Oppenheim, editors, *Proceedings ACM SAC 2000*, pages 270–277. ACM, Mar. 2000.

[31] J. Weidl and H. Gall. Binding object models to source code: An approach to object-oriented rearchitecting. In *Proceedings of the 22nd Computer Software and Application Conference (COMPSAC 1998)*. IEEE Computer Society Press, 1998.

[32] R. Wuyts. Declarative reasoning about the structure object-oriented systems. In *Proceedings of the TOOLS USA '98 Conference*, pages 112–124. IEEE Computer Society Press, 1998.