MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

وزارة التعليـــم العـالـي والبـحـث العلمـي

BADJI MOKHTAR-ANNABA UNIVERSITY
UNIVERSITÉ BADJI MOKHTAR-ANNABA

جامعة باجي مختار– عنابـــــــــة

FACULTE DES SCIENCES DE L'INGENIORAT
DEPARTEMENT D'INFORMATIQUE

Année : 2014/2015

THÈSE

Présentée en vue de l'obtention du diplôme de

Doctorat 3ième Cycle en Informatique

PROGRAMMATION VISUELLE POUR LA SEPARATION AVANCEE DES PREOCCUPATIONS

Spécialité : **Ingénierie des Logiciels Complexes**

Par

M. Sassi BENTRAD

Directeur de Thèse

M. Djamel MESLATI, Professeur, Université Badji Mokhtar-Annaba

Composition du jury :

Président :	**Hassina SERIDI**	Prof.	Université Badji Mokhtar-Annaba
Examinateurs :	**Abdelkrim AMIRAT**	Prof.	Université Mohammed Chérif Messaâdia - Souk Ahras
	Nora BOUNOUR	MCA	Université Badji Mokhtar-Annaba

بسم الله الرحمن الرحيم

والصلاة والسلام على محمـــد رســـول الله

In the name of God, the most gracious, the most merciful

And peace and blessings be upon Muhammed the messenger of God

REMERCIEMENTS

« Nous sommes des nains juchés sur les épaules de géants »
***Isaac Newton** (Proverbe).*

ENFIN, l'heure est venue de rédiger ces fameux remerciements, but ultime de chaque doctorant puisqu'il s'agit bel et bien de l'événement qui marque la fin de la thèse. C'est avec un grand plaisir que je réserve cette page en signe de gratitude et de reconnaissance à tous ceux qui ont participé à ce travail. Bien qu'il soit difficile de reconnaître et de remercier ici toutes les personnes (nombreuses) grâce à qui j'ai pu mener à bien toutes ces années de recherche; une période de rencontre et d'échange qui ne peut se résumer à une simple liste de noms.

El-Hamdou Li ALLAH, je tiens premièrement à me prosterner remerciant mon Dieu ALLAH le tout puissant de m'avoir donné la force morale et physique pour achever cette thèse. Tout d'abord, je remercie le Professeur *Djamel MESLATI*, pour m'avoir honoré par son encadrement, sa disponibilité indéfectible, ses conseils précieux, ses nobles valeurs humaines pendant les moments difficiles de ma vie de doctorant ainsi que pour la confiance qu'il m'a témoignée jusqu'à l'aboutissement de ce travail. Avec patience et pédagogie, il m'a fait découvrir plusieurs facettes de l'activité de chercheur.

Mes remerciements vont également aux membres de jury : Prof. *Hassina Seridi*, Prof. *Abdelkrim AMIRAT* et Dr. *Nora BOUNOUR*, qui me font le grand honneur de juger mon travail. Je les remercie également pour les discussions que nous avons pu avoir et qui m'ont enrichi.

Un grand merci va également à l'ensemble des membres du ***L**aboratoire d'**I**ngénierie des **S**ystèmes **CO**mplexes (**LISCO**),* avec lesquelles j'ai pu avoir de nombreux échanges scientifiques, et avec qui je partage bien plus qu'un lieu de travail, je nomme les enseignants-chercheurs *BENOUHIBA Toufik, HARIATI Mehdi, SOUICI-MESLATI Labiba, BOUNOUR Nora, ATIL Fadila* et *SAHEB Faiza,* et qui m'ont fait découvrir l'informatique et m'ont donné goût au génie logiciel.

Je tiens aussi à exprimer toute ma gratitude à tous les enseignants du département d'informatique qui ont contribué à ma formation et pour leur soutien moral et leurs encouragements, un grand merci à toutes les personnes qui, de près ou de loin, m'ont apporté leur soutien technique, scientifique ou moral pendant les moments difficiles.

Je ne pourrais oublier mes amis que je remercie pour leurs encouragements permanents dans les moments difficiles durant mon doctorat.

Bien sûr, je remercie mes parents, qui avec leur amour, leur esprit de sacrifice et leur soutien sans fin, m'avaient toujours guidé et encouragé, et qui ont tout fait pour que je puisse réussir dans ma vie. Merci pour tous et que le bon Dieu, vous garde et vous protège.

SASSI BENTRAD

DÉDICACES

À mes chers parents,

À ma famille,

À mes amis.

DÉCLARATION

JE déclare sur l'honneur que ce manuscrit représente l'ensemble des travaux de recherche que j'ai effectué avec un effort personnel durant les cinq années de ma thèse de *DOCTORAT* dans l'équipe *"**IPARAD**" (**I**ntégration de **PARAD**igmes pour le développement de systèmes complexes)* au ***L**aboratoire d'**I**ngénierie des **S**ystèmes **CO**mplexes (**LISCO**)* de l'Université Badji Mokhtar-Annaba (UBMA).

SASSI BENTRAD
5 juillet 2015

RÉSUMÉ

LE secteur économique est aujourd'hui exigeant en termes de systèmes logiciels dont le développement est de plus en plus complexe et la qualité un facteur décisif. La qualité étant considérée du point de vue utilisation, maintenabilité et réutilisabilité. Les logiciels sont par nature intangibles et abstraits et ne sont pas des artéfacts figés. Le logiciel évolue et change, ce qui implique que sa conception est révisée et améliorée de façon continue. A cet effet, les développeurs et les personnes chargées de la maintenance qui doivent le construire et le maintenir, doivent le comprendre en premier lieu. La maîtrise de la compréhension s'avère donc indispensable et pour la simplifier nous avons besoin d'atteindre un certain niveau d'abstraction du logiciel. Cette dernière peut être atteinte à travers la visualisation qui devient une aide précieuse lors du développement et maintenance. Le premier axe de recherche de notre thèse s'intéresse à la visualisation des grands systèmes logiciels orientés-aspects afin de faciliter efficacement la tâche de compréhension. Dans ce contexte, nous proposons une approche d'analyse et de visualisation bidimensionnelle (2D) et tridimensionnelle (3D) pour des propriétés quantitatives et plus particulièrement le volet statique (c.-à-d. code source) des programmes ASPECTJ.

D'un autre côté, bien que la séparation des préoccupations apporte de multiples avantages pour la communauté du génie logiciel, les développeurs éprouvent des difficultés à concevoir et à implémenter des programmes incluant des préoccupations. Le second axe de notre thèse cherche à atténuer ces difficultés lors de la programmation. Nous proposons une nouvelle méthodologie de codage hybride en incorporant les deux styles de codage textuel et visuel évitant ainsi les écueils de l'approche conventionnelle (à base de texte) en gérant mieux les diverses préoccupations.

Nos propositions sont mises en œuvre à travers des implémentations prototypes sous la plateforme ECLIPSE et évaluées par le biais d'expérimentations pratiques. De ces expérimentations, nous constatons que nos contributions sont prometteuses et ouvrent la voie vers des améliorations visuelles de l'ensemble des activités liées à l'ingénierie des logiciels.

MOTS-CLÉS : *ASPECTJ, Programmation orientée-aspects, Programmation visuelle, Séparation des préoccupations, Visualisation des logiciels.*

ABSTRACT

THE economic sector is now demanding in terms of software systems whose development is becoming more and more complex and their quality a decisive factor. Quality is considered from the point of view of use, maintainability and reusability. Softwares are intangible and abstract by their nature and are not frozen artifacts. The software evolves and changes, which means that its design is revised and improved continuously. To this end, the developers and mainteners that must build and maintain it, need to first understand it. So mastering the understanding is essential and in order to simplify it we need to reach a certain abstraction level of the software. This can be achieved through the visualization which is a valuable aid in the development and maintenance. The first research direction of our thesis is interested in visualizing aspect oriented softwares in order to facilitate effectively the task of their understanding. In this context, we propose a visualization and analysis approach in two and three dimensions for the quantitative properties of large aspects oriented softwares, especially the static property of ASPECTJ programs.

On the other hand, although the separation of concerns brings multiple benefits to the community of software engineering, developers find it difficult to design and implement programs including concerns. The second focus of our thesis seeks to mitigate these difficulties at the implementation level. We propose a new hybrid coding methodology incorporating both coding styles textual and visual thus avoiding the pitfalls of conventional textual approach.

Our proposals are implemented through prototypes under the ECLIPSE platform and evaluated through practical experiments. From these experiments, we find that our contributions are promising and open the door to visual enhancements of all activities related to software engineering.

***KEYWORDS**: ASPECTJ, Aspect-oriented programming, Separation of concerns, Software visualization, Visual programming.*

ملخص

حاليا القطاع الصناعي متشدد بخصوص البرمجيات حيث أصبح تطويرها متزايد التعقيد وجودتها عاملا حاسما. وتأخذ الجودة هنا من وجهة نظر الاستخدام، والصيانة وإعادة الاستخدام. وتكون البرمجيات بطبيعتها غير ملموسة ومجردة ولا تعد نتاجا اصطناعيا ثابتا. البرمجية تتطور وتتغير ويستلزم هذا مراجعة تصميمها وتحسينه باستمرار. وتحقيقا لهذه الغاية، فإن على المطورين وعمال الصيانة المكلفين ببنائها وصيانتها فهمها أولا. وبالتالي فإن الفهم ضروري ومن أجل تبسيط ذلك فإننا نحتاج للوصول إلى نضرة للبرمجية ذات مستوى تجريدي معين. يمكن تحقيق هذا من خلال العرض البصري للبرمجيات فهو مساعدة قيمة عند تطويرها وصيانتها. يهتم المحور الأول لأطروحتنا بالعرض البصري للبرمجيات الجانبية المنحى وذلك لتسهيل فهمها بنجاعة. في هذا المجال، نقترح مقاربة للتحليل والعرض البصري الثنائي والثلاثي الأبعاد لخصائص كمية للبرمجيات الكبيرة والجانبية المنحى وخصوصا الصفة الثابتة لبرامج أسبكت ج (ASPECTJ).

من ناحية أخرى، مع أن فصل الشواغل العرضية يجلب منافع متعددة لمجتمع هندسة البرمجيات، فان مطوري البرامج يجدون صعوبات في تصميم وتنفيذ البرامج الحاملة للشواغل العرضية. يسعى المحور الثاني من أطروحتنا لتخفيف الصعوبات على مستوى الترميز. حيث نقترح منهجية ترميز هجينة جديدة بدمج أسلوبا الترميز النصي والبصري لتجنب عقبات الطريقة النصية التقليدية للترميز.

يتم تنفيذ مقترحاتنا من خلال نماذج في قاعدة اكليبس (ECLIPSE) ومن ثم تقييمها من خلال التجارب العملية. من هذه التجارب، نلاحظ أن مساهماتنا واعدة وتفتح الباب أمام تحسينات بصرية لجميع الأنشطة المتصلة بهندسة البرمجيات.

الكلمات الدالة المفتاحية: البرمجة المرئية، العرض البصري للبرمجيات، الفصل للشواغل العرضية، البرمجة جانبية المنحى، اسبكت ج.

TABLE DES MATIÈRES

REMERCIEMENTS III

DEDICACES IV

DECLARATION V

RESUME VI

ABSTRACT VII

ملخص VIII

TABLE DES MATIERES IX

LISTE DES FIGURES XII

LISTE DES TABLEAUX XIV

LISTE DES ABREVIATIONS ET SIGLES XV

CHAPITRE 1 INTRODUCTION GENERALE 1

1.1 CONTEXTE DE RECHERCHE ET PROBLEMATIQUE 2

1.2 MOTIVATIONS 3

1.3 OBJECTIFS 5

1.4 ORGANISATION DU MEMOIRE 5

CHAPITRE 2 SEPARATION AVANCEE DES PREOCCUPATIONS 9

2.1 INTRODUCTION 10

2.1.1 SEPARATION AVANCEE DES PREOCCUPATIONS 11

2.1.2 PROBLEMES RECURRENTS ET IMPLICATIONS 12

2.1.3 PRINCIPALES APPROCHES ACTUELLES 14

2.2 L'APPROCHE ORIENTEE-ASPECTS 14

2.2.1 PRINCIPE ET CONCEPTS DE BASE 15

2.2.2 ASPECTJ, UNE IMPLEMENTATION REFERENCE 16

2.3 CONCLUSION 18

CHAPITRE 3 VISUALISATION DES PROGRAMMES ET PROGRAMMATION VISUELLE 19

3.1 INTRODUCTION 20

3.2 VISUALISATION DE LOGICIELS 21

3.2.1 REPRESENTATION GRAPHIQUE DE CONCEPTS ABSTRAITS 22

3.2.2 TECHNIQUES DE VISUALISATION 23

3.2.3 TRAVAUX CONNEXES 27

3.3 PROGRAMMATION VISUELLE 30

3.3.1 INTRODUCTION ET TERMINOLOGIE 30

3.3.2 AVANTAGES ET INCONVENIENTS 37

3.3.3 CLASSIFICATIONS 38

3.3.4 TRAVAUX CONNEXES 39

3.4 CONCLUSION 40

CHAPITRE 4 CONTRIBUTION À LA VISUALISATION 2D ET 3D 41

4.1 INTRODUCTION 42

4.2 VUE DETAILLEE DE L'APPROCHE 44

4.2.1 SYSTEME DE VISUALISATION 2D & 3D ORIENTEE-METRIQUES 44
4.2.2 PARAMETRES IMPORTANTS DE VISUALISATION 47
4.2.3 METRIQUES CONSIDEREES EN VISUALISATION 48

4.3 MISE EN ŒUVRE DE L'APPROCHE 55

4.3.1 VIZZASPECTJ-2D 55
4.3.2 VIZZASPECTJ-3D 61

4.4 CONCLUSION 65

CHAPITRE 5 VERS UNE METHODOLOGIE DE CODAGE HYBRIDE 67

5.1 INTRODUCTION 68

5.2 VUE DETAILLEE DE L'APPROCHE 70

5.2.1. VERS UNE APPROCHE HYBRIDE DE CODAGE 70
5.2.2. APPROCHE PROPOSEE 73

5.3 MISE EN ŒUVRE DE L'APPROCHE 77

5.3.1 ARCHITECTURE 78
5.3.2 IMPLEMENTATION 81

5.4 CONCLUSION 88

CHAPITRE 6 ÉVALUATION ET DISCUSSION 89

6.1 PROJETS SELECTIONNES POUR L'EVALUATION 90

6.2 ÉVALUATION DU SYSTEME DE VISUALISATION 91

6.2.1 ÉTUDE DE CAS SUR "VIZZASPECTJ-2D" 91
6.2.2 ÉTUDE DE CAS SUR "VIZZASPECTJ-3D" 93
6.2.3 DISCUSSION 96

6.3 ÉVALUATION DE L'APPROCHE HYBRIDE DE CODAGE "HM4AOP" 97

6.3.1 ÉTUDES DE CAS PRELIMINAIRES 97
6.3.2 EXPERIMENTATION DETAILLEE 104

6.4 CONCLUSION 114

CHAPITRE 7 CONCLUSION GENERALE 115

7.1 RAPPEL DU CADRE ET DES OBJECTIFS DE LA THESE 116

7.2 BILAN DES CONTRIBUTIONS 116

7.3 PERSPECTIVES DE RECHERCHE ET TRAVAUX FUTURS 118

REFERENCES 123

1. REFERENCES BIBLIOGRAPHIQUES 123
2. REFERENCES WEB (TECHNIQUES) 132

ANNEXE A CONSTRUCTIONS ET NOTATIONS GRAPHIQUES PROPOSEES 133

ANNEXE B ÉVALUATION PRELIMINAIRE DU PROTOTYPE HCODELESSAJ 139

1. ÉVALUATION EMPIRIQUE ET ANALYSE (PRETEST) 139
2. SONDAGE BASÉ-QUESTIONNAIRE (POSTTEST) 142

ANNEXE C GLOSSAIRE DE TERMINOLOGIE DU GENIE LOGICIEL 147

1. TABLEAU DES TERMINOLOGIES 147
2. REFERENCES DES TERMINOLOGIES 150

ANNEXE D A PROPOS DE L'AUTEUR 151

1. BIOGRAPHIE DE L'AUTEUR 151
2. CONTRIBUTIONS SCIENTIFIQUES 152

LISTE DES FIGURES

FIGURE 1. 1 SCHEMA D'ORGANISATION DE LA THESE. 7

FIGURE 1. 2 PLAN DE LECTURE DE LA THESE. 8

FIGURE 2. 1 PRINCIPE DE FONCTIONNEMENT DE LA PROGRAMMATION ORIENTEE-ASPECTS 15

FIGURE 3. 1 VUE POLYMETRIQUE DE VISUALISATION. 25

FIGURE 3. 2 VISUALISATION A L'AIDE D'UNE METAPHORE DE VILLE. 27

FIGURE 3. 3 EXEMPLE DE LA VISUALISATION DANS CODECITY. 28

FIGURE 3. 4 EXEMPLE DE LA VISUALISATION DANS VERSO. 29

FIGURE 3. 5 EXEMPLE DE LA VISUALISATION 3D HEB. 29

FIGURE 3. 6 PROGRAMMATION VISUELLE VERSUS VISUALISATION DES PROGRAMMES. 36

FIGURE 3. 7 LA CLASSIFICATION DE SHU. 39

FIGURE 4. 1 EXEMPLE D'UNE REPRESENTATION ARBORESCENCE. 46

FIGURE 4. 2 METRIQUES APPLIQUEES DANS LA VUE « COMPLEXITE DU SYSTEME » POUR UN CODE JAVA. 49

FIGURE 4. 3 METRIQUES APPLIQUEES DANS LES DEUX VUES "DEPENDANCES DES CLASSES" ET "DEPENDANCES DES PACKAGES". 50

FIGURE 4. 4 METRIQUES APPLIQUEES DANS LA VUE "COMPLEXITE DU SYSTEME" POUR UN CODE ASPECT. 51

FIGURE 4. 5 METRIQUES APPLIQUEES DANS LES VUES (A): "DEPENDANCES DES ASPECTS" ET (B) "DEPENDANCES DES PACKAGES". 51

FIGURE 4. 6 VISUALISATION TRIDIMENSIONNELLE (3D) AVANT ET APRES LE TISSAGE STATIQUE 54

FIGURE 4. 7 VUE SIMPLIFIEE DE LA REPRESENTATION INTERNE DU CODE SOUS "VIZZASPECTJ-2D". 56

FIGURE 4. 8 DIAGRAMME DE CLASSES DE LA REPRESENTATION INTERNE DU CODE SOUS "VIZZASPECTJ-2D". 57

FIGURE 4. 9 CHAINE DE PRODUCTION DES VUES POLYMETRIQUES SOUS "VIZZASPECTJ-2D". 58

FIGURE 4. 10 CYCLE DE CONSTRUCTION DES VUES SOUS "VIZZASPECTJ-2D" 59

FIGURE 4. 11 UNE VUE DE L'OUTIL "VIZZASPECTJ-2D" SOUS LA PLATEFORME ECLIPSE. 60

FIGURE 4. 12 DIAGRAMME DE CLASSES UML SIMPLIFIE DU MODELE DE VILLE SOUS "VIZZASPECTJ-3D". 62

FIGURE 4. 13 CHAINE DE PRODUCTION D'UNE VUE DE VILLE SOUS "VIZZASPECTJ-3D". 62

FIGURE 4. 14 INTERACTIONS ENTRE "VIZZASPECTJ-3D" ET "VIZZASPECTJ-2D" SOUS LA PLATEFORME ECLIPSE. .. 63

FIGURE 4. 15 ALGORITHME DE CONSTRUCTION DE LA HIERARCHIE DES PACKAGES. 64

FIGURE 4. 16 UNE CAPTURE D'ECRAN DE L'OUTIL "VIZZASPECTJ-3D" SOUS LA PLATEFORME ECLIPSE 65

FIGURE 5. 1 PROCESSUS GLOBALE D'UNE CONSTRUCTION HYBRIDE DES PROGRAMMES ORIENTE-ASPECTS SELON L'APPROCHE "HM4AOP"..... 72

FIGURE 5. 2 VUE GLOBALE DE L'APPROCHE PROPOSEE. 74

FIGURE 5. 3 ÉTAPES DU PROCESSUS DE CODAGE..... 76

FIGURE 5. 4 ARCHITECTURE STRUCTURELLE DE L'OUTIL SUPPORT DE L'APPROCHE "HM4AOP". 78

FIGURE 5. 5 ARCHITECTURE FONCTIONNELLE DE L''OUTIL SUPPORT DE L'APPROCHE "HM4AOP"..... 80

FIGURE 5. 6 UNE VUE GLOBALE DU PROCESSUS ET TECHNOLOGIES ADOPTEES POUR L'IMPLEMENTATION DE L'OUTIL SUPPORT DE L'APPROCHE "HM4AOP". 82

FIGURE 5. 7 UNE CAPTURE D'ECRAN DU PROTOTYPE "HCODELESSAJ" SOUS LA PLATEFORME ECLIPSE. 84

FIGURE 5. 8 VUE D'ENSEMBLE DU PROCESS DE TRANSFORMATION ET DE GENERATION DES TEMPLATES DE CODE SOUS LE PROTOTYPE "HCODELESSAJ". 87

FIGURE 6. 1 VUE "COMPLEXITE DU SYSTEME" VISUALISANT LA HIERARCHIE D'HERITAGE. 92

FIGURE 6. 2 VISUALISATION DES DEPENDANCES VIA "VIZZASPECTJ-2D": (A) "DEPENDANCES DES CLASSES & ASPECTS" ET (B) "DEPENDANCES DES PACKAGES". 93

FIGURE 6. 3 VISUALISATION TRIDIMENSIONNELLE (3D) DU CODE SOURCE DE L'OUTIL "VIZZASPECTJ-3D"..... 94

FIGURE 6. 4 VUE DE VILLE EN 3D DU PROJET "AJHOTDRAW"..... 95

FIGURE 6. 5 VUE DE VILLE EN 3D DU PROJET "JDK1.6". 95

FIGURE 6. 6 PROTOTYPAGE DES ASPECTS CANDIDATS SUR LE DIAGRAMME DE CLASSE UML..... 98

FIGURE 6. 7 UN APERÇU DU MODELE VISUEL DU PROGRAMME CIBLE "TELECOM". 99

FIGURE 6. 8 UN EXTRAIT AU FORMAT XMI ILLUSTRANT LE MODELE VISUEL DU PROGRAMME CIBLE "TELECOM"..... 99

FIGURE 6. 9 UN APERÇU DU MODELE VISUEL DU PROGRAMME CIBLE "FIBONACCI". 101

FIGURE 6. 10 UN EXTRAIT AU FORMAT XMI ILLUSTRANT LE MODELE VISUEL DU PROGRAMME CIBLE "FIBONACCI". 102

FIGURE 6. 11 VUE DES REPRESENTATIONS GRAPHIQUES RESUMANT LES RESULTATS DE L'ETUDE COMPARATIVE QUANTITATIVE : "HCODELESSAJ" VS. "AJDT". 108

FIGURE 6. 12 VUE GRAPHIQUE RESUMANT LES RESULTATS DE L'ETUDE COMPARATIVE QUALITATIVE : "HCODELESSAJ" VS. "AJDT". 109

FIGURE 7. 1 UNE VUE GLOBALE SUR L'ARCHITECTURE CONCEPTUELLE DU FRAMEWORK "HM4AOP". 121

LISTE DES TABLEAUX

TABLE 4. 1 METRIQUES SELECTIONNEES POUR UNE VISUALISATION EFFICACE. 47

TABLE 4. 2 DESCRIPTION DES METRIQUES APPLIQUES DANS LA VUE « COMPLEXITE DU SYSTEME » POUR UN CODE JAVA. 48

TABLE 4. 3 DESCRIPTION DES METRIQUES APPLIQUES DANS LES VUES « DEPENDANCES DES CLASSES & PACKAGES ». 49

TABLE 4. 4 DESCRIPTION DES METRIQUES APPLIQUEES DANS LA VUE "COMPLEXITE DU SYSTEME" POUR UN CODE ASPECT. 50

TABLE 4. 5 DESCRIPTION DES METRIQUES APPLIQUEES DANS LES VUES "DEPENDANCES DES ASPECTS ET DES PACKAGES". 51

TABLE 4. 6 METRIQUES D'ARETES DANS LES DEUX VUES "DEPENDANCES DES CLASSES" ET "DEPENDANCES DES PACKAGES". 52

TABLE 5. 1 CODAGE CONVENTIONNEL VS. METHODOLOGIE DE CODAGE HYBRIDE. 73

TABLE 5. 2 EXTRAIT DE LA TABLE DES NOTATIONS GRAPHIQUES PROPOSEES AVEC LES REGLES DE TRANSFORMATIONS ACCELEO. 85

TABLE 6. 1 UNE LISTE DETAILLEE DES PROGRAMMES ASPECTJ SELECTIONNES POUR L'EVALUATION. 90

TABLE 6. 2 LES PROGRAMMES ASPECTJ SELECTIONNES POUR L'EXPERIMENTATION. 105

TABLE 6. 3 RECAPITULATIF DES RESULTATS DE COMPARAISON QUANTITATIVE : "HCODELESSAJ" VS. "AJDT". 107

TABLE 6. 4 RECAPITULATIF DES RESULTATS DE COMPARAISON QUALITATIVE : "HCODELESSAJ" VS. "AJDT". 108

TABLE 6. 5 RECAPITULATIF DES REPONSES AU QUESTIONNAIRE. 113

LISTE DES ABREVIATIONS ET SIGLES

LA signification des acronymes utilisés dans ce manuscrit est, en règle générale, précisée lors de leur première utilisation. Ci-après nous donnons tous ces acronymes, leur signification en anglais et (ou) une équivalence en français lorsque nécessaire.

API	Application Programming Interface
AJDT	AspectJ Development Tools
AOP	Aspect Oriented Programming ; *Programmation Orientée Aspects (**POA**)*
AOSD	Aspect-Oriented Software Development
ASoC	Advanced Separation of Concerns ; *Séparation Avancée des Préoccupations (**SAP**)*
CIM	Computer Independent Model
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
GUI	Graphical User Interface
HM4AOP	**H**ybrid **M**ethodology **for A**spect-**O**riented **P**rogramming
HCodelessAJ	**H**ybrid **Codeless** Programming Methodology for **A**spect**J**
IDE	Integrated Development Environment
IHM	Interfaces Homme-Machine
JDK	Java Development Kit
MDA	Model Driven Architecture ; *Architecture dirigée par les modèles*
MDD	Model Driven Development
MDE	Model-Driven Engineering ; *Ingénierie Dirigée par les Modèles (**IDM**)*
M2T	Model-to-Text transformation
MIT	Massachusetts Institute of Technology
OMG	Object Management Group
OOP	Object Oriented Programming; *Programmation Orientée Objets (**POO**)*
PIM	Platform Independent Model ; *Modèles indépendants des plateformes*
PSM	Platform Specific Model ; *Modèles liés aux plateformes*
PDE	Plug-in Development Environment
SoC	Separation of Concerns ; *Séparation des préoccupations*
SWEBOK	Software Engineering Body of Knowledge
T2M	Text-to-Model transformation
UML	Unified Modeling Language
XMI	XML Meta-data Interchange ; *Format XML d'échange de méta-données*
XML	eXtensible Markup Language

CHAPITRE 1

INTRODUCTION GÉNÉRALE

SOMMAIRE

1.1 CONTEXTE DE RECHERCHE ET PROBLEMATIQUE 2
1.2 MOTIVATIONS 3
1.3 OBJECTIFS 5
1.4 ORGANISATION DU MEMOIRE 5

Ce chapitre est une introduction du manuscrit de cette thèse. Il présente successivement le contexte de recherche de la problématique étudiée et les motivations du sujet, la démarche de travail et son analyse, puis les objectifs que nous voulons atteindre face à la problématique posée. Nous terminons cette introduction en exposant l'organisation de la thèse ainsi que son plan de lecture.

1.1 Contexte de Recherche et Problématique

De nos jours, l'évolution des systèmes logiciels est devenue un problème central dans l'industrie du logiciel. Le secteur économique est aujourd'hui exigeant en termes de logiciels dont le développement est de plus en plus complexe et la qualité un facteur décisif. La qualité étant considérée du point de vue de l'utilisation, la maintenance et la réutilisation.

La complexité des logiciels se répercute sur leur processus de maintenance qui est également de plus en plus complexe. Une proportion substantielle des ressources dépensées dans les l'industries logicielles est versée dans les activités de maintenance. Le coût de maintenance tourne souvent autour de 50 % à 75 % du coût total du logiciel tout au long de son cycle de vie. Il a été établi que ces activités expliquent, aujourd'hui, le coût plus grand dans le développement logiciel [Dag 03, Som 04].

La maintenance d'un système logiciel et le temps de développement de nouvelles fonctionnalités sont des éléments cruciaux en termes de gestion de projets, car le non-respect des délais, parfois très courts, peut conduire à un surcoût imprévu, voire à l'échec du projet [Pig 96]. On estime que plus de la moitié des efforts de maintenance sont consacrés à la compréhension du logiciel lui-même [Cor 89]. Ce constat montre que la compréhension est un enjeu crucial, présent tout au long la durée de vie du logiciel, et conditionne, même, la réussite ou l'échec de la maintenance à la fin.

Le logiciel est par nature intangible et abstrait et généralement complexe. Cette complexité se traduit par un nombre énorme d'entités logicielles (objets ou artéfacts) liées par différents types de dépendances (en anglais : *Relationships*). De plus, le logiciel n'est pas figé. Il évolue et change, ce qui implique que sa conception et son implémentation sont à réviser et à améliorer de façon continue. Pour cela les développeurs et les personnes chargées de la maintenance qui doivent l'étendre et le maintenir, doivent bien le comprendre en premier lieu.

La compréhension est une première étape importante à réaliser avant de pouvoir effectuer des tâches de maintenance sur un logiciel. Ceci est une évidence en soi, car, avant de pouvoir apporter un changement ou une correction, il faut tout d'abord déterminer les parties concernées. De plus, dans un cas comme dans l'autre, il faut aussi pouvoir déterminer quel sera l'impact des modifications.

En situation réelle, les mainteneurs doivent sans cesse redécouvrir l'implémentation de certaines fonctionnalités du logiciel, au prix d'efforts importants et d'une perte de temps. Suites à des évolutions multiples, le logiciel devient de plus en plus complexe, difficile à analyser et à comprendre en lisant juste le code source. En effet, cette lecture prend un temps considérable et ne donne pas une idée globale sur les divers aspects du logiciel. Cette problématique est souvent liée à un code non structuré, ou à une documentation incomplète ou inexistante. Par conséquent, la compréhension basée sur la lecture du code source deviendra difficile voire impossible, plus particulièrement pour les grands systèmes logiciels.

C'est à ce niveau que se situe le premier axe de recherche de notre thèse, avec pour objectif de répondre, au moins partiellement, à la question : Quelle est la manière la plus efficace de présenter un logiciel pour faciliter sa compréhension ? C'est une question à la fois simple et vaste qui peut être abordée de différentes façons.

D'autre part, l'augmentation de la complexité et de la taille des systèmes logiciels actuels met en évidence certaines limites de l'approche orientée-objets. Face à cette complexité la communauté du génie logiciel propose diverses approches et paradigmes qu'il est primordial de maîtriser. La séparation avancée des préoccupations (ASoC, *Advanced Separation of Concerns*) apporte de multiples avantages pour le développement moyennant de nouveaux concepts et mécanismes [Lop 95]. Le développement orientés-aspects de logiciels (AOSD, *Aspect-Oriented Software Development*), initialement apparu comme une technique d'implémentation de ces concepts, est devenu récemment un sujet de recherches actives dans la communauté du génie logiciel [Rob 04, W25]. Particulièrement, un grand intérêt pour l'approche de programmation orientée-aspects (AOP, *Aspect-Oriented Programming*) a émergé. Les nouveaux concepts de l'AOP permettent aux développeurs de contrôler l'exécution des programmes de base en agissant sur leurs flux de contrôle et leurs flux de données. Une action judicieuse sur les deux flux permet de mettre en œuvre de nombreuses préoccupations allant de la gestion de la concurrence à la persistance des données, voire l'optimisation des calculs.

Malheureusement, face à ces atouts, les développeurs éprouvent des difficultés à concevoir et plus particulièrement à implémenter des programmes incluant des préoccupations. De ce fait, la problématique majeure soulevée par cette thèse (second axe) est résumée par la question de recherche suivante : comment peut-on faire face aux difficultés qu'éprouvent les développeurs à gérer mieux les préoccupations ? Cette question représente l'ossature de notre travail et de celle-ci se dégagent plusieurs lignes directrices qui nous ont guidés.

1.2 Motivations

La compréhension des logiciels orientés aspects constitue un des problèmes majeurs auquels les programmeurs et les mainteneurs doivent faire face. Parce que les logiciels en question sont trop complexes et trop précieux pour être remplacé ou être réécrit à partir de zéro, la compréhension devient une tâche impossibles sans autres alternatives. Beaucoup de travaux ont abordé l'automatisation de la compréhension. Cependant, elle reste pour une grande part une tâche semi-automatique qui fournit une assistance intelligente au processus de maintenance.

Pour mieux comprendre le code source, nous avons besoin d'atteindre un certain niveau d'abstraction, cela peut être fait à travers les diverses techniques de visualisation, car l'être humain est souvent plus efficace pour comprendre une information lorsque celle-ci est

représentée concrètement. La visualisation des programmes représente une aide précieuse lors du développement et la maintenance. Elle permet de présenter un programme (ou autres artéfacts) sous une forme synthétique abstraite qui donne rapidement une idée sur le contenu, la logique, les liens entre entités, les points représentant un problème, etc.

Les motivations de cette thèse sont nombreuses, nous en citons, dans ce qui suit, deux facteurs :

Le **premier facteur** qui nous a motivés à chercher une solution à la problématique est l'introduction de plus d'interactivité au processus de développement à travers l'aspect visuel, qui est un moyen pour améliorer les diverses tâches en étant simple et éfficace. Le développement des systèmes logiciels orientés aspects de grandes tailles se caractérise par la grande capacité de recueil de données, mais également une grande puissance des techniques adoptées. En effet, la volumétrie et la complexité sont de plus en plus hors de portée des capacités humaines. C'est pour ces raisons que les techniques visuelles, en plein essor, s'imposent. Pendant des années, les chercheurs ont essayé d'améliorer les processus de programmation en y ajoutant des techniques visuelles. Certains travaillent pour faciliter la compréhension des paradigmes de la programmation conventionnelle en utilisant des représentations visuelles (visualisation des programmes). D'autres utilisent des paradigmes de programmation différents qui exploitent les représentations visuelles (programmation visuelle).

Le **second facteur** est le développement orienté-aspects. L'approche orientée-aspects est une technologie relativement jeune et semble une voie prometteuse pour contourner le phénomène inhérent à la décomposition. La naissance d'une approche nécessite de nouvelles méthodes et l'implantation de nouveaux outils supports permettant, d'une part, d'assurer son intégration dans le milieu industriel et, d'autre part, de donner les moyens nécessaires à la communauté du génie logiciel de l'étudier et de l'appréhender. Pour cette raison, différents travaux de recherche ont émergé suite à l'apparition de cette approche, en particulier lors de la phase de programmation. En outre, la vaste adoption du langage JAVA dans la communauté nous pousse à choisir le langage ASPECTJ, en sa qualité d'implémentation référence de l'approche orientée-aspects appliquée à JAVA, comme un support pour la mise en œuvre de nos contributions.

Nous pensons que l'introduction de l'approche visuelle (visualisation et programmation visuelle) facilite, d'une part, le codage et la documentation en permettant d'éviter certains écueils de l'approche conventionnelle textuelle afin de mieux gérer les diverses préoccupations et, d'autre part, la compréhension et l'amélioration des programmes en les rendant plus modulaires et plus faciles à faire évoluer.

1.3 Objectifs

Cette thèse s'inscrit dans le domaine d'ingénierie des logiciels complexes. Elle est consacrée principalement à la prise en compte de l'approche visuelle lors de la phase d'implémentation et la phase de maintenance des systèmes utulisant une séparation avancée des préoccupations, et plus précisément les systèmes orientés aspects.

L'idée principale de notre travail est de faire appel aux diverses techniques d'interaction visuelles, de visualisation et de programmation visuelle simultanément pour offrir une approche unifiée pouvant contribuer à simplifier les tâches de compréhension et de construction de code.

Nous annonçons les objectifs de ce travail de la façon suivante :

- **Dans un premier temps**, notre travail aura à proposer une approche d'analyse et de visualisation bidimensionnelle (2D) et tridimensionnelle (3D) pour des aspects (ou propriétés) quantitatives des grands systèmes logiciels orientés aspects et plus particulièrement le volet statique (c.-à-d. le code source) plutôt que sur le volet dynamique (c.-à-d. l'exécution) des programmes ASPECTJ, dans le but d'effectuer des analyses qualitatives. L'approche se basera sur des vues polymétriques et des métaphores visuelles pour rendre les analyses plus intuitives et plus efficaces. Les outils supports sont sensé être puissants et offrir la possibilité de donner des vues interactives avec un haut niveau d'abstraction permettant d'obtenir rapidement une idée sur le contenu, la logique, les liens entre entités, les points représentant un problème, etc.

- **Dans un second temps**, on vise l'introduction de l'approche visuelle durant la phase d'implémentation. Cette approche est supposée être mise en pratique à travers un prototype sous la plateforme ECLIPSE. Ce style de codage a pour objectif de rendre l'approche orientée-aspects attractive pour les développeurs débutants, en leur permettant d'être plus productifs et créatifs, en se concentrant sur les innovations et les solutions, plutôt que sur les formalismes syntaxiques qui sont souvent complexes.

1.4 Organisation du mémoire

Ce manuscrit est organisé en trois grandes parties selon le schéma de description montré à la **Figure 1.1**. La **Figure 1.2** donne un guide de lecture.

La **première partie**, contenant deux chapitres, elle traite de la séparation avancée des préoccupations en premier lieu, puis la visualisation des programmes et la programmation visuelle. Elle est répartie en deux chapitres suivant les concepts clés du sujet. Dans chacun, nous introduisons les grandes lignes et les principaux concepts référencés et essentiels à la lecture du manuscrit.

- Le **chapitre 2** expose la séparation avancée des préoccupations, et plus particulièrement l'une de ses approches les plus prometteuses, à savoir la programmation orientée-aspects. On met l'accent notamment sur le langage ASPECTJ comme une implémentation de référence.
- Le **chapitre 3** présente la visualisation des programmes en premier lieu, puis la programmation visuelle. On y trouve également une vue globale sur les différentes techniques visuelles considérées avec les principaux travaux connexes dans chaque champ de recherche.

La **deuxième partie**, comprenant le quatrième et le cinquième chapitre, est consacrée à décrire en détail nos contributions au domaine de visualisation et programmation visuelle. Ces deux chapitres répondent aux objectifs principaux de la thèse.

- Le **chapitre 4** décrit d'abord en détail notre première contribution concernant une approche d'analyse et de visualisation bidimensionnelle (2D) et tridimensionnelle (3D) pour des propriétés quantitatives des grands logiciels orientés aspects et plus particulièrement le volet statique des programmes ASPECTJ. Nous présentons également les implantations prototypes des outils supports réalisés dans les deux volets : conceptuel et implémentation.
- Le **chapitre 5** expose notre deuxième contribution concernant l'introduction de l'approche visuelle durant la phase d'implémentation. Dans un premier temps, nous y décrivons une méthodologie de codage hybride en particulier pour la programmation orientée-aspects, et ensuite nous détaillons aussi la conception et l'implémentation de l'outil support proposé.

Dans la **troisième partie**, représentée par le **chapitre 6**, nous abordons de manière détaillée l'évaluation de nos contributions ainsi que les résultats obtenus par les outils supports proposés.

Cette partie vise à évaluer nos propositions en analysant leurs possibilités en termes d'applicabilité et d'extensibilité à travers des expérimentations détaillées. Ces dernières ont été mises en place sur les implantations prototypes afin de montrer les aspects faisabilité et efficacité puis valider nos contributions. Cette description est accompagnée d'une discussion autour des résultats obtenus de nos expérimentations et également de certaines fonctionnalités. Enfin, nous détaillons quelques problèmes rencontrés lors de l'implémentation.

Finalement, le **chapitre 7** donne une conclusion et dresse un bilan récapitulant les idées principales de notre thèse et nos contributions. Nous y précisons les principaux apports, les objectifs atteints et les limites actuelles de nos travaux. On y trouve également des éléments

de réflexion et on y dresse explicitement les extensions possibles et également les principales perspectives dégagées à l'issue de nos travaux.

La fin de ce manuscrit contient des références bibliographiques et techniques suivies par des annexes complétant ce qui a déjà été traité dans les différents chapitres. **L'annexe A** présente un tableau descriptif des représentations graphiques considérées dans la deuxième contribution. **L'annexe B**, donne un rapport sur les expérimentations. **L'annexe C** est un glossaire des termes techniques employés, et **l'annexe D** présente un aperçu à propos de l'auteur et liste la production scientifique faite dans le cadre de cette thèse.

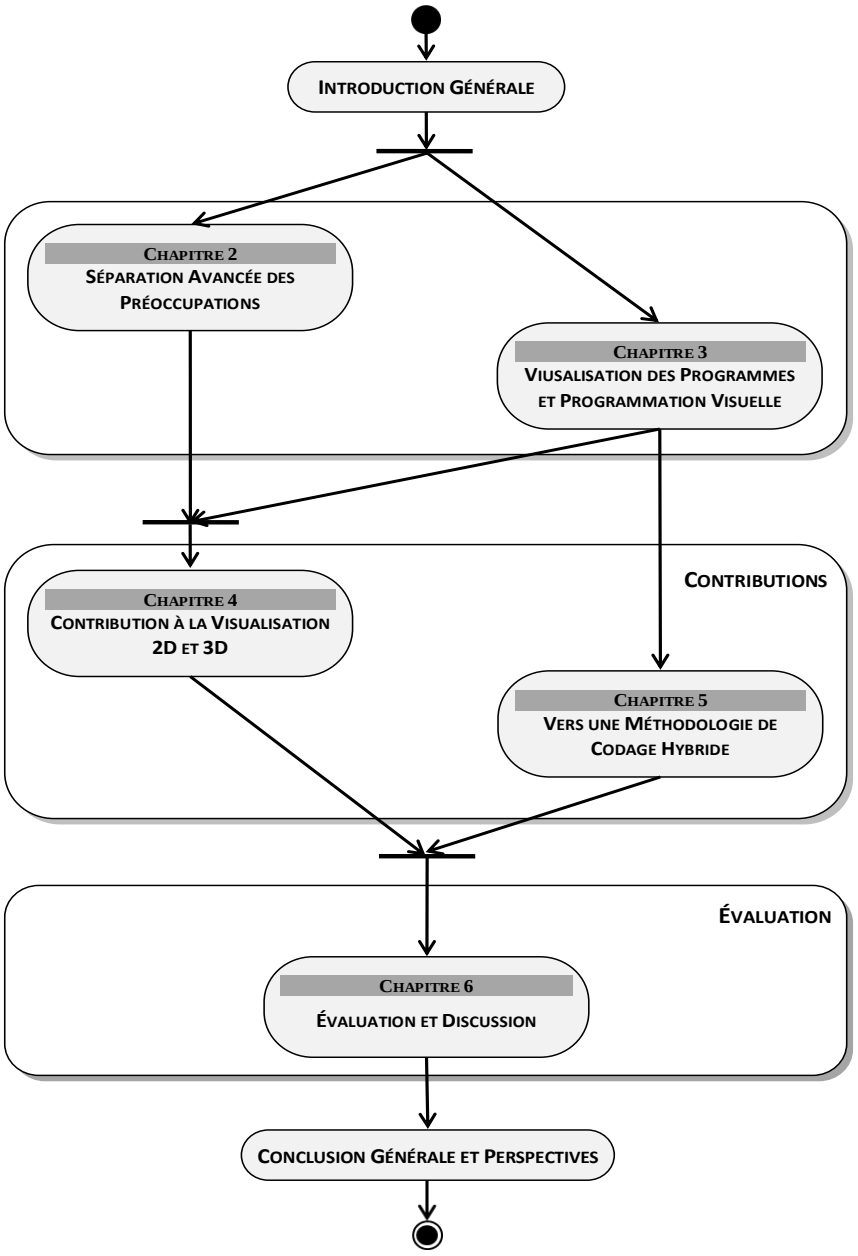


Figure 1. 1 Schéma d'organisation de la thèse.

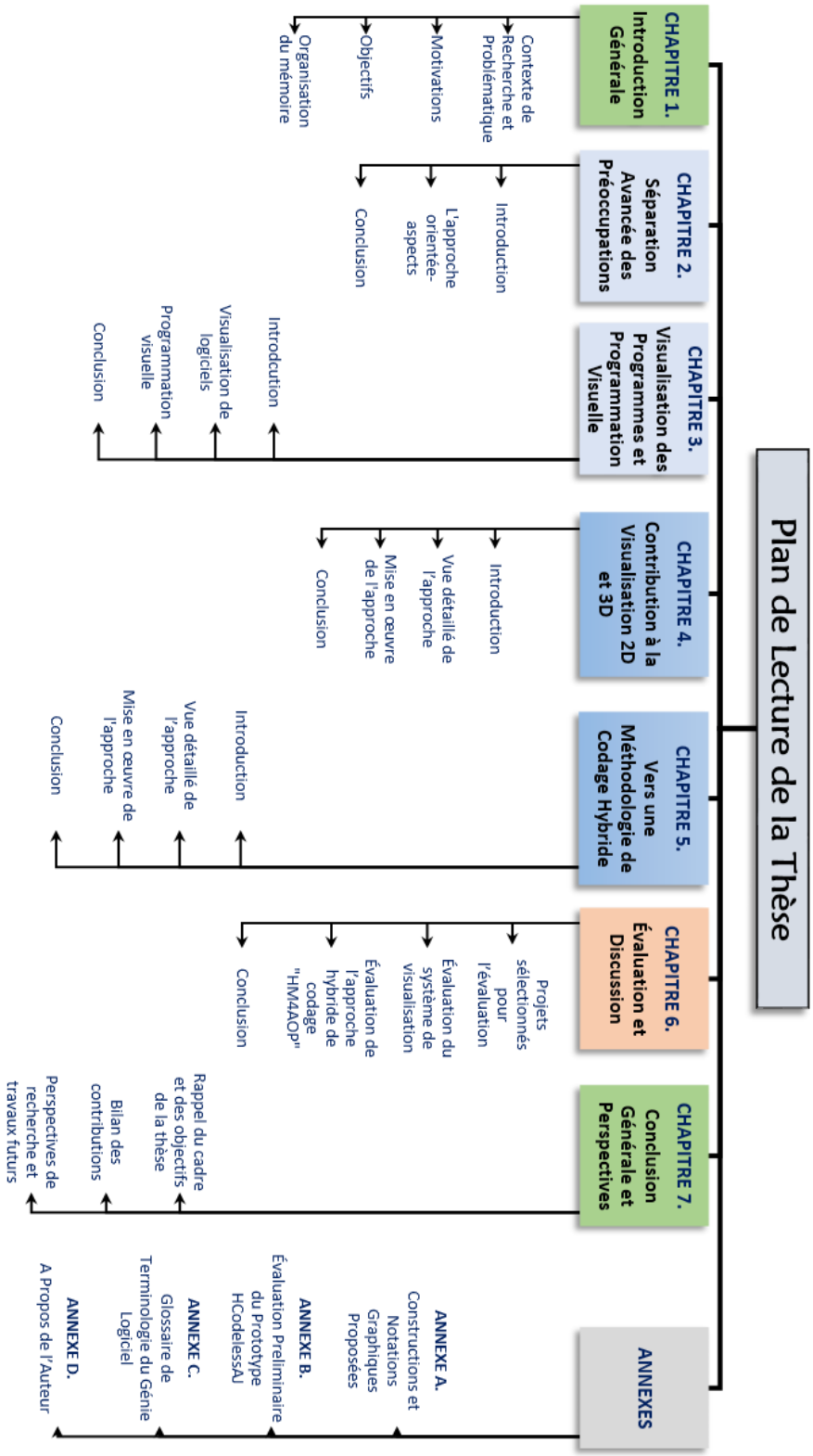


Figure 1. 2 Plan de lecture de la thèse.

CHAPITRE 2

SÉPARATION AVANCÉE DES PRÉOCCUPATIONS

SOMMAIRE

2.1 INTRODUCTION 10

2.1.1 SEPARATION AVANCEE DES PREOCCUPATIONS 11
2.1.2 PROBLEMES RECURRENTS ET IMPLICATIONS 12
2.1.3 PRINCIPALES APPROCHES ACTUELLES 14

2.2 L'APPROCHE ORIENTEE-ASPECTS 14

2.2.1 PRINCIPE ET CONCEPTS DE BASE 15
2.2.2 ASPECTJ, UNE IMPLEMENTATION REFERENCE 16

2.3 CONCLUSION 18

Dans ce chapitre, nous donnons une vue d'ensemble sur la séparation avancée des préoccupations (ASoC, Advanced Separation of Concerns), en se concentrant sur l'approche de programmation par aspects (AOP, Aspect-Oriented Programming) en tant que domaine d'application de notre travail et plus particulièrement sur son implémentation référence appliquée au langage JAVA—le langage ASPECTJ. On s'intéressera tout d'abord à quelques généralités théoriques et notions préliminaires essentielles à la lecture du manuscrit, ainsi que les champs de recherche actuels, puis on présentera certains des travaux connexes les plus prometteurs. Pour plus de détails, le lecteur peut se référer aux références citées dans les deux chapitres.

2.1 Introduction

De nos jours, les systèmes logiciels intègrent de plus en plus de fonctionnalités afin de faire face à des problèmes toujours plus nombreux. L'introduction de ces fonctionnalités alourdit le développement ; par exemple : la concurrence, la distribution, les contraintes temps réel, la persistance, la tolérance aux pannes, et les interfaces homme-machine (IHM). Leur implémentation peut être soumise à des changements qui ne sont pas forcément prévus dès l'origine et leur interdépendance n'est donc pas nécessairement prise en compte. Les logiciels deviennent donc de plus en plus complexes et difficiles à faire évoluer. Les fonctionnalités précédemment citées peuvent aussi être partagées entre plusieurs logiciels, et donc leur réutilisation est rendue difficile à cause des interdépendances [Qui 04].

Un des buts principaux de l'ingénierie des logiciels est de faire face à ces problèmes et de garantir le développement de logiciels de qualité. Les approches traditionnelles de développement telle que l'approche orientée-objets (POO, *Programmation Orientée-Objets*) fournissant un support important de décomposition permettant d'offrir la réutilisation, et améliorer la qualité logicielle, grâce à de nombreux atouts (ex. encapsulation, polymorphisme, modularité...). Cette dernière constitue indéniablement une technologie importante pour aider à l'implémentation des systèmes complexes. En effet, elle permet ainsi la réduction des délais de développement et de maintenance. Pourtant, elle ne prend pas en compte les fonctionnalités entrecoupantes dont l'implémentation se trouve souvent éparpillées dans les différentes entités du système. De ce fait, la réutilisation et l'évolution de ces fonctionnalités demandent des transformations importantes et répétitives lorsqu'elles sont extraites du cadre de leur utilisation première. C'est l'une des raisons pour laquelle la communauté du génie logiciel vise à améliorer cette approche. Elle constitue une motivation première pour organiser et décomposer un logiciel en un ensemble d'éléments compréhensibles et facilement manipulables [Mes 07].

Ce besoin a donné naissance à un nouveau paradigme : "la séparation des préoccupations" (SoC, *Separation of Concerns*) qui est considérée comme l'une des approches les plus prometteuses du génie logiciel [Lop 95]. Son principe préconise le découpage d'un logiciel en entités de taille réduite et aussi indépendantes que possible les unes des autres afin d'améliorer la compréhension, la réutilisation et l'évolution du code [Par 72, Lop 95]. Une préoccupation (*Concerns*) est un concept générique décrivant une entité homogène pour désigner une fonctionnalité. Le principe est que ces préoccupations doivent être séparées les unes des autres pour permettre leur réutilisation dans différents contextes. La difficulté est de s'affranchir des problèmes de couplage inter-préoccupations.

Ce paradigme prône la séparation des différentes préoccupations à la fois lors de la phase de conception et lors de la phase d'implémentation. Cette séparation est justifiée par les trois tendances [Qui 04] :

- Les logiciels sont composés de différentes préoccupations enchevêtrées,
- La complexité est de plus en plus croissante, et
- Le nombre de préoccupations requis est aussi en augmentation.

En effet, moyennant de nouveaux concepts, le paradigme résout deux problèmes épineux: l'enchevêtrement et l'éparpillement de code. Ces derniers sont considérés par la communauté du génie logiciel comme la source de nombreux problèmes et donc leur élimination ouvre la voie vers des améliorations et des issues prometteuses pour l'ensemble des activités liées à l'ingénierie des logiciels. En particulier, une modélisation plus fidèle de la réalité et une meilleure maîtrise de la complexité de logiciels, ainsi qu'une réutilisation plus facile des artefacts produits [Mes 07].

Il est important que cette séparation soit réalisée dans les différentes étapes du cycle de développement. Ce paradigme permet d'identifier les préoccupations qui composent un logiciel puis les sépare lors de la conception (voire des phases en amont). Cette séparation doit ensuite être répercutée lors de la phase d'implémentation [Qui 04].

Le principe est de dissocier la programmation des différentes préoccupations composant un logiciel et de les décrire avec leurs interrelations de manière à rendre leur code source plus modulaire, plus lisible et plus compréhensible. De ce fait, le logiciel n'est plus abordée dans sa globalité, mais par parties. Ce principe est au cœur de tout développement en ingénierie moderne. Il s'agit d'un principe clé pour la réduction de la complexité apparente de développement de grands systèmes logiciels. En effet, il améliore la compréhension de tels systèmes, facilite leur maintenance et leur évolution, et augmente leur réutilisation [Kic 92, Aks 94, Lop 94, Oka 94, Hac 06].

2.1.1 Séparation avancée des préoccupations

Sous l'appellation séparation des préoccupations, nous regroupons un ensemble de nouvelles approches qui partagent un point commun : celui d'identifier les différentes facettes d'un système logiciel comme : des parties fonctionnelles (code métier ou encore code de base) qui fait référence aux tâches de base que le système réalise et pour lesquelles il a été conçu (c.-à-d. structures et comportements correspondant à la partie métier) et des parties non-fonctionnelles (ou code technique). Ces dernières dites préoccupations transversales (en anglais : *Crosscutting Concerns*) telles que la synchronisation, la sécurité, la gestion de la persistance, la gestion des exceptions, la gestion des transactions, l'optimisation, etc. [Qui 04, Mes 07]. Cette distinction entre les différentes catégories de préoccupations, va dans le sens d'une conception et d'une implémentation simplifiée, une meilleure compréhension, une diminution du couplage entre préoccupations et plus généralement, une réutilisation accrue [Qui 04].

Les approches de séparation des préoccupations prônent la séparation des préoccupations des parties fonctionnelles comme une approche qui permet une meilleure modularité et ayant une influence bénéfique sur les processus de développement et de maintenance [Mes 07]. Actuellement, il est communément admis qu'une bonne séparation des préoccupations permet d'une part de réduire le travail et l'expertise demandés au programmeur, et d'autre part, de mieux réutiliser la partie fonctionnelle dans d'autres environnements et d'autres logiciels. Pour cela, les classes de base ne doivent contenir que la partie fonctionnelle et leur adaptation doit être effectuée à l'extérieur de ces classes [Kic 97, Aks 98, Oss 99].

Dans sa forme la plus générale, la séparation est multidimensionnelle dans le sens où le système logiciel n'est pas partagé en partie fonctionnelle et partie non fonctionnelle, mais la séparation peut être faite simultanément selon des dimensions multiples et de type arbitraire. Par conséquent, les préoccupations elles-mêmes interagissent et se recouvrent [Oss 99, Tar 99, Mes 07].

La mise en œuvre des préoccupations transversales dans le cadre des approches traditionnelles de développement (ex. la programmation orientée-objets, à base de composants, générique, méta-programmation) conduit, le plus souvent, à plusieurs problèmes. Nous montrons brièvement, dans ce qui suit, les problèmes récurrents et leurs implications.

2.1.2 Problèmes récurrents et implications

Les principaux problèmes engendrés par les préoccupations transversales découlent d'une manière générale de deux problèmes de fond récurrents en développement traditionnel : les problèmes d'enchevêtrement du code (en anglais : *Code Tangling*) et d'éparpillement ou dispersion du code (en anglais : *Code Scattering*) [Hac 06].

✪ *Enchevêtrement du code*

Une préoccupation est l'abstraction d'un but particulier ou une unité modulaire ayant un certain intérêt dans le système logiciel. Ces unités de décomposition peuvent interagir simultanément avec plusieurs préoccupations transversales. Il est fréquent, par exemple, que les développeurs aient à penser de manière simultanée à la synchronisation, à la persistance ou encore à la sécurité d'accès aux données d'une même préoccupation fonctionnelle de base. Ceci conduit à la présence d'éléments de définition de plus d'une préoccupation transversale dans la définition d'une seule préoccupation de base, donnant lieu ainsi à un code enchevêtré [Hac 06].

✪ *Eparpillement du code*

Les préoccupations transversales concernent, en général, une ou plusieurs préoccupations fonctionnelles. Leurs définitions se trouvent ainsi dispersées au travers de l'ensemble des unités de modularisation représentant les préoccupations fonctionnelles qu'elles affectent. La définition d'une préoccupation transversale se trouve ainsi dispersée dans plusieurs unités modulaires (voir **Figure 2.1**). Par exemple, si on considère la sécurité, on constate que chaque classe du code fonctionnel contiendra une partie du code relatif à cette préoccupation technique. On parle alors d'éparpillement ou dispersion du code de cette dernière, ce qui rend la maintenance et l'évolution du code une tâche fastidieuse.

Combinés ensemble, ces deux problèmes majeurs contrarient le principe de séparation des préoccupations et affectent ainsi, la conception et l'implémentation de plusieurs façons, que nous énumérons dans ce qui suit [Fer 04, Hac 06].

- **Code non lisible et difficile à comprendre.** L'enchevêtrement (respectivement, l'éparpillement) du code d'une ou de plusieurs préoccupations transversales avec celui (respectivement, dans celui) relatif à l'ensemble des préoccupations fonctionnelles, produit souvent un code final mélangeant des préoccupations dissimulées. Ceci diminue la lisibilité du code qui devient moins compréhensible.
- **Mauvaise traçabilité.** Les préoccupations transversales dont le code est éparpillé dans celui du reste du logiciel, sont difficiles à localiser dans le code final et sont donc non traçables, nuisant à la compréhension et par conséquent à la réutilisation et à l'évolution du code.
- **Maintenabilité et évolution difficiles.** Le code étant non lisible et moins compréhensible, il est très difficile de localiser et faire évoluer les différentes préoccupations. En effet, toute modification ou évolution d'une préoccupation transversale devient très complexe dès lors qu'elle affecte la majorité des préoccupations fonctionnelles concernées.
- **Faible efficacité et productivité.** La définition de plusieurs préoccupations transversales simultanément en même temps et dans une même unité modulaire, éloigne l'attention du développeur du but final du logiciel, vers des besoins annexes, limitant d'autant son efficacité et sa productivité.
- **Faible réutilisation.** Le code des différentes préoccupations transversales étant éparpillé, il est non réutilisable. En effet, le code relatif aux préoccupations fonctionnelles étant non propre (mélangé avec celui des préoccupations transversales affectant ces dernières), il est difficilement réutilisable dans d'autres contextes d'utilisation [Hac 06].

2.1.3 Principales approches actuelles

Les limites des approches traditionnelles de développement vis-à-vis de la séparation des préoccupations transversales ont poussé les chercheurs à préconiser une réelle séparation qui permette, entre autres, une réduction de la complexité de conception, de réalisation, mais aussi de maintenance, une meilleure modularité qui améliore la compréhension, la réutilisation et l'évolution du code. Ce souci a donné lieu à de nombreux travaux sur la séparation dans la communauté du génie logiciel. De nos jours, une large part de ces travaux traite de trois principales approches : l'approche orientée-aspect (AOP, *Aspect Oriented Programming*) [Kic 97], l'approche composition de filtres (CF, *Composition Filters*) [Aks 98] et l'approche de séparation multidimensionnelle des préoccupations (MDSOC, *Multi-Dimensional Separation of Concerns*) [Oss 99]. Bien que les auteurs de ces approches reconnaissent l'intérêt de séparer les préoccupations importantes d'un logiciel, ils ne s'entendent pas sur les concepts et mécanismes à utiliser. La philosophie de chaque approche est différente et les concepts utilisés sont variés et ad hoc [Mes 04]. Dans la suite, nous présentons l'approche orientée-aspects.

2.2 L'approche orientée-aspects

Les approches traditionnelles présentent, le plus souvent, certaines limites et problèmes conséquents vis-à-vis de la séparation et de la représentation explicite et modulaire des préoccupations transversales (en anglais : *Crosscutting Concerns*). De la même façon ils n'offrent pas de solutions efficaces pour la composition et l'adaptation de telles préoccupations. L'approche orientée-objets a permis d'atteindre un certain degré d'indépendance sans pour autant casser totalement les liens entre les préoccupations. Ainsi, les préoccupations fonctionnelles restent encore dépendantes des préoccupations techniques. De nos jours, ces observations sont au cœur de nombreux travaux de recherche, dont le but principal est d'offrir des modèles permettant une meilleure séparation et composition de tout type de préoccupations [Hac 06].

Cette vision a donné naissance actuellement à de nouveaux modèles et langages de programmation associés. Ces modèles sont connus sous la désignation plus générale d'approche orientée-aspects. Ils prônent une décomposition des programmes non seulement en unités modulaires représentant les préoccupations fonctionnelles de base, mais aussi en unités dédiées à la représentation des préoccupations transversales. De plus, ils offrent des solutions adéquates de composition de préoccupations, afin de construire des systèmes logiciels efficaces [Hac 06].

Gregor Kiczales et *al.* [Kic 97] ont pris comme point de départ les deux problèmes précédemment cités de l'approche traditionnelle et prônent la séparation des préoccupations comme un moyen d'éliminer leurs implications et inconvénients. La programmation orientée-aspects (POA) ou (AOP, *Aspect-Oriented Programming*), est une approche de

programmation relativement récente, dont les fondations ont été définies suite aux travaux de *Kiczales* et de son équipe au centre de recherche Xerox PARC (*Palo Alto Research Center*) en Californie au milieu des années 1990 [Kic 97, Kic 01a, Ren 04].

L'approche aspects est l'une des techniques novatrices les plus étudiées à l'heure actuelle. Elle a émergé suite à nombreux travaux de recherche, et a été originellement proposée comme une extension de l'approche objets dont l'objectif était d'offrir une meilleure modularité par le fait que le code source est réduit, moins enchevêtré et plus proche de la perception du monde réel. Par conséquent, cette approche améliore la réutilisation, l'évolution et la maintenance.

2.2.1 Principe et concepts de base

Une solution aux problèmes d'enchevêtrement et de dispersion du code des propriétés non-fonctionnelles rencontrés avec la programmation orientée-objets (POO), consiste à séparer et découpler leurs définitions comme le veut le principe de la séparation des préoccupations [Lop 95, Fer 04]. Partant de ce principe, l'approche de programmation par aspects (POA) consiste à structurer l'application logicielle en unités modulaires indépendantes représentant les définitions de préoccupations transversales. Ces modules étant tous indépendants les uns des autres, il devient beaucoup plus facile de les implémenter, lors du développement, mais aussi pour les maintenir et les réutiliser par la suite [Kic 97, Fer 04].

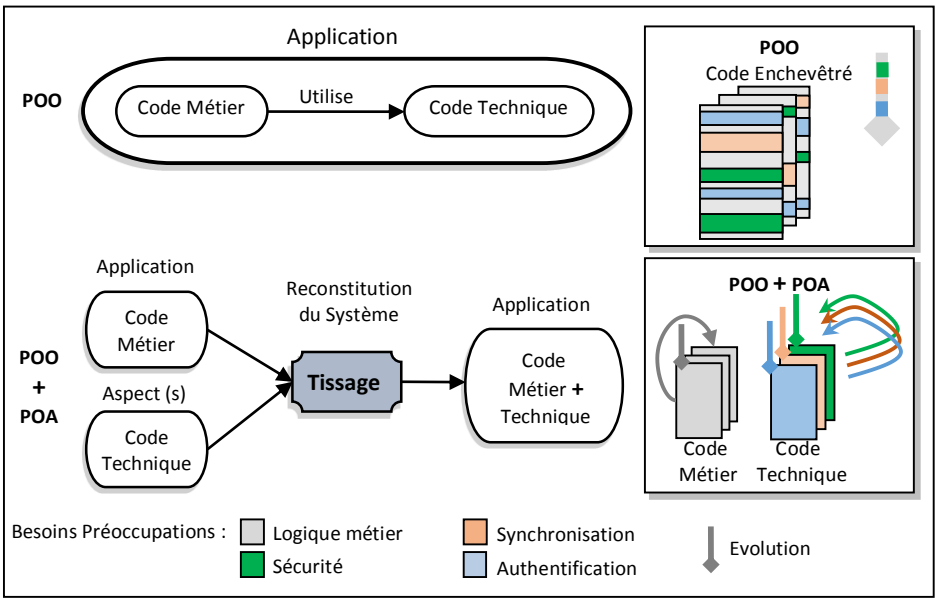


Figure 2. 1 Principe de fonctionnement de la programmation orientée-aspects

La **Figure 2.1** montre le principe général de fonctionnement. Une application construite en utilisant l'approche de POO possède une architecture pouvant se résumer à du "code

métier" qui utilise certaines fonctionnalités secondaires "code technique" (ou "non-métier", telles que logging, authentification, caching, transactions, journalisation, etc.). Par exemple, la journalisation est typiquement une préoccupation transversale qui nécessite que le code métier, fasse explicitement appel au code technique de l'opération journalisation pour chaque élément nécessitant d'être journalisé.

La POA se fonde sur une séparation claire entre les préoccupations fonctionnelles et non-fonctionnelles présentes dans les systèmes logiciels. Il est important de souligner qu'elle utilise la POO dans un cadre qui en conserve les bénéfices tout en palliant les limitations suscitées.

L'approche aspects propose d'encapsuler le code technique dans de nouvelles unités modulaires appelées "aspects" pour remédier à l'entrelacement (appelé aussi "entrecoupage", en anglais : *Crosscutting*) des préoccupations qui sont mal prises en compte par l'approche objets. Un "aspect" est une abstraction d'une préoccupation, dont la particularité est de s'appliquer à un ensemble de classes entrecoupantes pour les classes fonctionnelles. L'implémentation d'un aspect va s'entrelacer avec le reste de l'implémentation. L'approche objets est par contre toujours utilisée pour modéliser les comportements qui sont naturellement hiérarchisés [Kic 01a].

Le code technique est composé statiquement ou dynamiquement avec le code métier (c.-à-d. greffer le code des aspects dans le code des méthodes des classes) —lequel ne contient aucune référence explicite au code technique— afin de former le comportement global du programme final. Cette composition, appelée tissage d'aspects (en anglais : *Aspect Weaving*), a le plus souvent lieu à la compilation, mais peut avoir lieu (selon les langages et les outils supports) à l'exécution. Même si cela signifie une régénération (reconstitution) de l'enchevêtrement et l'éparpillement du code, le tissage n'élimine pas les avantages de l'approche, car le code source demeure un code orienté-aspects. Ainsi le code n'est pas mélangé et il est plus facile de comprendre le code de chaque préoccupation qui n'est présent qu'en un seul exemplaire, et sa mise au point ou son évolution s'en trouvent facilités [Kic 01a, Mes 07].

Ainsi, la vision motivante de l'approche aspects est que, lors de la conception et l'implémentation, on puisse fournir des spécifications indépendantes pour chaque préoccupation et chaque fonctionnalité, ce qui permet de bénéficier des avantages de la modularité qui garantit un code plus simple, plus facile à développer et à maintenir et qui a un fort potentiel de réutilisation.

2.2.2 ASPECTJ, une implémentation référence

Les modèles et langages de programmation associés, introduits dans le cadre de l'approche aspects, sont définis le plus souvent comme étant des extensions des modèles et langages "généraux" existants tels que JAVA ou C++, appelés langages de base et avec

lesquels sont exprimées les préoccupations de base du programme [Hac 06]. Un langage orienté-aspects est utilisé principalement lors de la programmation et, à un degré moindre, à des niveaux plus abstraits. Nous citons à titre d'exemple quelques extensions appliquées au langage JAVA : ASPECTJ [W1], AspectWerkz [W17], CaesarJ [W18], JBoss AOP [W19], JAC [W20], etc.

Cependant, rien de la POA n'est spécifique à la POO en général, ou à JAVA en particulier. On trouve ainsi des outils supports de POA plus ou moins avancés en C, C++, C# ou Smalltalk. Bien que le domaine de la POA commence à devenir mature, des solutions abouties pour les étapes amont de la conception ainsi qu'un processus de développement orienté-aspects de logiciels (AOSD, *Aspect-Oriented Software Development*) restent à définir [Rob 04, W25].

Pour illustrer un exemple, nous avons choisi de présenter ASPECTJ comme une principale implémentation, et l'une des expérimentations les plus abouties des langages orientés-aspects [Kic 01a, Lad 03, W1]. Développé en 1996 par *Gregor Kiczales* et son équipe du centre de recherche Xerox PARC (*Palo Alto Research Center*), et dont les premières versions ont été disponibles en 1998. Au-delà d'ASPECTJ, la POA rencontre depuis cette date un engouement important dans le milieu de la recherche. Cela a donné lieu à l'émergence de nombreux autres langages et outils supports.

ASPECTJ [W1] est une extension orientée-aspects, à usages multiples, de JAVA. Elle apporte des formalismes syntaxiques très expressifs avec de nouveaux mots-clefs permettant d'une part, de définir dans des aspects le code relatif aux préoccupations transversales, et de préciser, d'autre part, la manière (i.e. où, quand et comment) dont ce code sera composé avec celui des classes de base. La sémantique propose de voir les aspects comme des classes, et donc de profiter de l'ensemble des apports de l'approche objets en JAVA [Kic 01a, Gra 03]. *Kiczales* qualifie ASPECTJ d'extension «compatible» du langage JAVA dans la mesure où [Kic 01a, Lad 03] :

- Chaque programme valide JAVA est un programme valide ASPECTJ,
- Tous les programmes valides ASPECTJ sont capables, une fois compilés, de s'exécuter sur n'importe quelle machine virtuelle JAVA,
- Il est possible d'étendre l'ensemble des outils JAVA pour supporter ASPECTJ, ceci inclut les IDEs, les outils de documentation et de conception,
- Le style de programmation avec ASPECTJ est très proche de celui de la programmation avec JAVA.

Le langage JAVA a été étendu tout en supportant la modularisation, dans des aspects, de deux types d'implémentation de transversalité (ou d'entrecroisement de préoccupations transversales avec les préoccupations de base). La première implémentation assure la définition des perfectionnements, qui doivent être exécutées à des points bien définis, du flot

d'exécution de l'ensemble des composants constituant le système. Il s'agit ici, d'un entrecroisement dynamique (en anglais : *Dynamic Crosscutting*). Il permet la définition d'éléments destinés à modifier le comportement (les opérations et méthodes) des classes de base. La deuxième implémentation, permet de définir des nouvelles structures et méthodes sur des types déjà existants. Il s'agit ici, d'un entrecroisement statique (en anglais : *Static Crosscutting*) du fait qu'elles affectent les signatures statiques des différents composants du système. Il est destiné à augmenter la définition des classes de base, tout en leur ajoutant de nouvelles propriétés structurelles et/ou comportementales tels que les attributs et les méthodes [Hac 06].

Pour ce faire, en plus des concepts fondamentaux de l'approche objets (i.e. classe, interface, attribut, opération, méthode, association, généralisation…), ASPECTJ introduit un certain nombre de nouveaux concepts et mécanismes dont, les points de jointures (*Join Points*), les points de coupure (*Pointcuts*) et les consignes (*Advices*) pour l'implémentation des transversalités dynamiques, et les introductions (*Inter-Type Declarations*) pour l'implémentation des transversalités statiques [Kic 01a, Hac 06].

2.3 Conclusion

La séparation des préoccupations fournit un support méthodologique de modélisation et de programmation. Elle doit, bien sûr, être accompagnée d'un processus d'intégration (dit de composition ou de tissage) des différents artefacts générés pour les différentes préoccupations. Ce support permet de gérer plus naturellement, et d'une façon modulaire, des intégrations complexes. Comme énoncé dans [Mes 06], il est communément admis qu'une bonne séparation des préoccupations réduit la complexité des systèmes logiciels, facilite la réutilisation, améliore la compréhension et simplifie l'intégration des artefacts.

Nous nous sommes limités, dans ce chapitre, à rappeler les grandes lignes de la séparation des préoccupations. Nous avons aussi décrit les principes et les fondements de la programmation par aspects avec ASPECTJ comme une implémentation référence dans ce domaine et un bon moyen d'illustrer cette nouvelle approche. Cette présentation est synthétique et ne couvre pas tout le domaine, ni tout le langage. Par rapport à l'approche objets, la POA est une technologie relativement jeune. C'est un paradigme clé des applications logicielles d'aujourd'hui et de demain. Elle présente des avantages indéniables, et elle a acquis une grande popularité et l'acceptation de la communauté des développeurs JAVA [Ngu 05]. Pour un état de l'art détaillé se référer à [Bou 01].

Comme notre travail se place dans le vaste contexte de la visualisation de programmes et la programmation visuelle, le chapitre suivant est consacré à donner une vue d'ensemble du domaine.

CHAPITRE 3

VISUALISATION DES PROGRAMMES ET PROGRAMMATION VISUELLE

SOMMAIRE

3.1 INTRODUCTION 20

3.2 VISUALISATION DE LOGICIELS 21

3.2.1 REPRESENTATION GRAPHIQUE DE CONCEPTS ABSTRAITS 22
3.2.2 TECHNIQUES DE VISUALISATION 23
3.2.3 TRAVAUX CONNEXES 27

3.3 PROGRAMMATION VISUELLE 30

3.3.1 INTRODUCTION ET TERMINOLOGIE 30
3.3.2 AVANTAGES ET INCONVENIENTS 37
3.3.3 CLASSIFICATIONS 38
3.3.4 TRAVAUX CONNEXES 39

3.4 CONCLUSION 40

Dans ce chapitre, nous donnons une vue d'ensemble sur la visualisation des programmes et la programmation visuelle. Dans un premier temps, on se focalise sur la visualisation qui permet de montrer les diverses propriétés d'un système logiciel et qui fait l'objet de nombreux travaux de recherche. Puis, on donnera un aperçu sur la programmation visuelle. À la fin, nous exposerons brièvement les travaux courants les plus prometteurs qui prônent l'introduction de l'approche visuelle durant le processus d'ingénierie des logiciels et en particulier lors de la phase de programmation et la phase de maintenance. Pour plus de détails, le lecteur peut se référer aux références citées dans les deux chapitres.

3.1 Introduction

L'être humain est doté de capacités très développées pour visualiser des informations complexes, ce qui joue un rôle majeur dans ses processus cognitifs de compréhension et de mémorisation (reconnaissance rapide de motifs, couleurs, formes, etc.). Afin de diminuer grandement la charge cognitive nécessaire à la compréhension, il utilise des approches visuelles (ou graphiques) afin de mieux appréhender des notions abstraites ou pour représenter le monde qui l'entoure. En effet, l'être humain a plus d'efficacité à comprendre l'information lorsque celle-ci est représentée de manière concrète (c.-à-d. d'une façon graphique) plutôt que théorique ou virtuelle. Le rôle le plus important que la visualisation joue dans le raisonnement humain en général et plus particulièrement dans le progrès scientifique a été souligné aussi par les philosophes à travers les siècles [Has 01].

Gershon [Ger 94] définit la visualisation comme suit : "la visualisation est plus qu'une méthode de calcul. C'est le processus de transformation des informations en des formes visuelles, permettant leur observation et leur manipulation. La présentation graphique résultante permet de percevoir visuellement des fonctionnalités et des informations qui sont cachées dans les données, mais néanmoins avec l'exigence d'explorer et analyser ces données". Il s'agit donc de fournir à l'utilisateur une compréhension qualitative du contenu de l'information.

La visualisation de l'information est définie comme l'usage de représentations graphiques et interactives supportées par l'ordinateur afin de représenter d'une façon efficace toute sorte d'informations (c.-à-d. des données souvent abstraites) à celui qui l'utilise. Ses représentations nécessitent de manipuler des objets graphiques avec leurs attributs (ou indices) visuels (tels que : forme, taille, position, couleur, mouvement, etc.) d'une manière interactive avec différents niveaux d'abstraction afin de produire des vues différentes pour la même information [And 02, Sol 06].

La visualisation d'information consiste à représenter les données sous forme graphique en utilisant au maximum les capacités de perception visuelle des utilisateurs et les différentes dimensions perceptuelles fournies par les représentations graphiques [Has 07]. Il s'agit d'afficher un maximum de données dans un espace restreint tout en conservant une bonne lisibilité pour qu'elles puissent être analysées. La représentation graphique permet de refléter les propriétés structurales des données par l'arrangement spatial des objets, ou encore de refléter les propriétés intrinsèques de ces données par les attributs visuels des objets graphiques.

La visualisation d'information nous permet: (1) d'exploiter les caractéristiques du système visuel humain pour faciliter la manipulation et l'interprétation de données variées, (2) de faciliter la découverte de connaissances grâce à des représentations visuelles issue d'une tâche d'analyse, et (3) de communiquer efficacement les informations à travers ces représentations.

Dans ce chapitre, nous nous intéressons à l'utilisation de la visualisation dans le contexte d'ingénierie des logiciels pour stimuler la compréhension et la perspicacité, puis la programmation visuelle. Ces deux champs présentent en effet des intersections et concernent respectivement la représentation et la manipulation d'autres représentations. Dans ce qui suit, nous donnons une vue d'ensemble des généralités théoriques et connaissances techniques de base au sujet de chaque champ. Tout d'abord, nous nous intéresserons aux techniques de visualisation les plus couramment utilisées, ensuite, nous terminerons par une synthèse des principaux travaux existants. La seconde partie est consacrée à présenter des concepts du domaine de programmation visuelle en donnant un aperçu de quelques travaux actuels ainsi que les principaux problèmes liés au domaine.

3.2 Visualisation de logiciels

La visualisation de logiciels est une façon efficace de comprendre un système logiciel et d'explorer visuellement les informations extraites de celui-ci. Une de ses grandes forces est de pouvoir représenter sous des formes familières, souvent en utilisant des métaphores du monde réel, des données intangibles qui ne possèdent pas forcément une représentation naturelle. Les vues de visualisations produites permettent ainsi de tirer profit de la puissance du système de perception visuelle humaine afin d'accélérer le processus de compréhension [Die 07].

Dans ce contexte, de nombreux outils de visualisation ont été proposés ces dernières années. Ces outils représentent une aide précieuse lors du développement et maintenance en permettant aux développeurs d'élever le niveau d'abstraction ce qui facilite grandement la tâche de compréhension de logiciels de grandes tailles en présentant des informations relatives au code source sous forme claire et concise. Cependant, certains types d'information demeurent difficiles à représenter de manière efficace [Lan 05b, Lan 06].

Price [Pri 98], a défini la visualisation de logiciels d'une manière générale comme suit: "Software visualization is the use of the crafts of typography, graphic design, animation and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software". Que nous traduisons par : "La visualisation de logiciels est l'utilisation de l'art de la typographie, la conception graphique, l'animation et la cinématographie avec les techniques modernes d'interaction homme-machine et des outils graphiques afin de faciliter à la fois, la compréhension humaine et l'usage efficace des logiciels".

La visualisation de logiciels ou de programmes (PV, *Program Visualization*) est largement utilisée dans l'industrie. C'est un domaine d'intérêt grandissant qui vise à réduire la complexité de maintenance des grands systèmes logiciels [Eic 98]. Elle est vue comme un procédé d'abstraction des concepts et d'entités logicielles (tels que : classes, aspects, packages et leurs dépendances ou relations) et de production d'une représentation concrète

(graphique) et compréhensible. Cette représentation visuelle présente quelques propriétés (ou aspects) du logiciel.

En général, la visualisation est exigée dont le but d'exprimer et de simplifier un concept abstrait ou un objet réel d'une manière que l'utilisateur de cette visualisation puisse comprendre facilement ce qu'il voit et puisse découvrir facilement, par exemple, les anomalies de conception et éventuellement les localiser en explorant simultanément le code source avec les diverses vues. C'est une question de représentation multidimensionnelle statique ou dynamique en utilisant des objets graphiques et des animations pour présenter les données, les entités logicielles du code source et leurs relations, etc. Comme le code source est une sorte d'informations, nous considérons la visualisation de logiciels comme une spécialisation de la visualisation d'information dont l'objectif est d'afficher approximativement tout type de données abstraites, par contre dans le domaine d'ingénierie des logiciels, la concentration est uniquement sur la visualisation des programmes [Die 07].

Selon la définition étroite de *John T. Stasko* [Sta 98], "la visualisation des programmes est le procédé d'obtention des vues tangibles (graphiques) du code source ou des structures de données sous une forme statique ou dynamique". Selon une définition large de *Stephan Diehl* [Die 07], "C'est une visualisation d'artéfacts en relation avec le logiciel et son processus de développement en permettant une compréhension plus rapide et plus précise de ses fonctionnalités et ses diverses propriétés (c.-à-d. structure, comportement, évolution)".

Les experts du domaine focalisent leurs recherches sur les techniques de représentations graphiques de diverses propriétés (ou aspects) du logiciel [Die 07] :

- La **structure** fait référence aux parties statiques et relations du logiciel, c.-à-d. celles qui peuvent être mesurées ou être inférées sans exécution. Cela inclut le code source, les structures de données, le graphe d'appel statique et l'organisation modulaire du logiciel.
- Le **comportement** fait référence à l'exécution du logiciel avec des données réelles et abstraites.
- L'**évolution** fait référence au processus de développement et en particulier au changement du code source pendant l'ajout de nouvelles fonctionnalités ou la correction des bogues.

3.2.1 Représentation graphique de concepts abstraits

Le premier problème rencontré pour représenter un logiciel est l'absence de forme concrète de ce dernier. Le code source n'a pas de forme si ce n'est que le texte qui le représente, ce qui rend sa représentation très difficile. De plus, le code a une sémantique qui existe uniquement dans le but d'être comprise par le programmeur et la machine et n'a pas de réalité en dehors de cet usage. Étant donné cette problématique, nous sommes obligés de

choisir des formes arbitraires pour le représenter. Il est possible alors d'utiliser des formes déjà existantes ou de les emprunter à d'autres domaines [Die 07].

Les caractéristiques visuelles des éléments composant une représentation graphique sont la base de tout type de visualisation. Leur choix doit se faire à la fois dans un souci d'efficacité pour une meilleure compréhension; un souci d'efficacité en terme d'affichage par la machine et d'esthétisme pour l'analyste. La justesse de ces caractéristiques fait la différence entre les bons et les mauvais outils de visualisation [Die 07].

Un concept abstrait du programme est une connaissance conceptuelle à un haut niveau d'abstraction. La représentation des objets abstraits est difficile parce qu'elle ne concerne pas des objets tangibles qui ont déjà des représentations graphiques dans nos esprits. Cette représentation présente une construction logique plutôt qu'une construction physique. Une question se pose alors, comment transformer les entités logicielles et leurs relations en des représentations graphiques significatives ?

Face à la diversité et au nombre grandissant de problèmes traités, les objets mathématiques (p. ex. les graphes, les automates, etc.) permettant un bon niveau d'abstraction sont devenus incontournables. La théorie des graphes permet de simplifier un problème donné pour mieux étudier les relations de ses éléments. Par exemple, un graphe est défini par un ensemble d'éléments appelés noeuds, un nœud est une abstraction sur un objet.

Le domaine de visualisation ne peut être traité sans aborder les techniques d'interaction afin de permettre une exploitation réelle des vues produites. En effet, la perception est indissociable de l'action d'interaction ; l'être humain est capable d'extraire des informations d'une interface s'il peut agir directement et rapidement sur cette interface. L'interaction sera donc mise en avant dans les diverses approches de visualisation [Sol 06, Die 07].

La possibilité d'explorer rapidement un système logiciel de grande taille et de le quantifier, à l'aide des vues interactives, ou d'avoir la possibilité de comprendre sa forme, afin de découvrir rapidement ses entités et leurs dépendances, représente un critère important pour une meilleure compréhension durant la phase de maintenance [Lan 06, Die 07].

3.2.2 Techniques de visualisation

Un système logiciel est par nature un produit intangible. Seule sa représentation peut être communiquée entre les personnes, et entre les personnes et les machines [Cha 96]. La visualisation existe non seulement pour l'analyse de celui-ci, mais aussi pour qu'il soit plus facile de le comprendre. En même temps, il est reconnu qu'une visualisation d'un système logiciel de grande taille sous des formes classiques (p.ex. diagrammes, graphes, etc.) est à toute fin pratique inefficace [Kni 00]. En ce sens, de nombreux travaux utilisent les métaphores visuelles (en anglais: *Visual Metaphor*s) comme moyen efficace de représentation.

Les métaphores sont déjà utilisées en informatique depuis longtemps. Une métaphore est une analogie qui est une représentation graphique d'une entité abstraite ou d'un concept dans le but de transférer des propriétés du domaine de la représentation graphique à celui de l'entité abstraite ou du concept [Dos 02]. Les visualisations qui s'appuient sur des métaphores du monde réel permettent d'exploiter les ressources des systèmes visuel et cognitif humains afin d'extraire de ces représentations les régularités et les discontinuités qui sont les éléments de base de toute analyse qualitative [Lan 05a, Lan 05c].

Dans le domaine d'ingénierie des logiciels, la visualisation métaphorique consiste à représenter souvent le logiciel dans un contexte familier, en utilisant des formes graphiques que l'utilisateur reconnait automatiquement. Ceci permet à l'utilisateur de comprendre plus rapidement l'information grâce à son rapport à la réalité [Gre 93, Dos 00, Kni 00, Plo 02].

D'après *Dos Santos* et *al.* [Rus 00], une technique métaphorique permet une reconnaissance plus rapide de la structure globale du logiciel et une meilleure compréhension des situations plus complexes, ce qui évite un des plus gros problèmes des visualisations en trois dimensions, à savoir la désorientation dans l'espace 3D. De plus, les visualisations utilisant des métaphores du monde réel se basent sur la compréhension naturelle et intuitive de l'être humain du monde qui l'entoure, et donc une meilleure orientation et navigation au sein de la visualisation.

Dans un tel monde métaphorique, les métaphores peuvent rendre l'interface plus intuitive et plus attrayante. Cependant elles peuvent aussi ajouter un bruit visuel inutile. L'aspect attractif de ce genre de visualisation est intéressant pour les utilisateurs et notamment les développeurs.

✪ *Vue Polymétrique*

La vue polymétrique (en anglais : *Polymetric View*) crée par *Lanza*, *Ducasse* et *Demeyer* [Dem 99, Lan 03b, Lan 03c], est une visualisation qui exploite des métriques différentes pour montrer un ensemble d'entités logicielles. Elle utilise des formes géométriques simples dont la taille et la couleur peuvent être modifiées en fonction des valeurs de métrique ou d'une caractéristique particulière.

Cette technique de visualisation est implantée dans l'outil *CodeCrawler* [Lan 03a]. C'est une représentation de graphes où la forme et la taille de chaque élément est configurable afin de pouvoir créer plusieurs vues différentes en fonction des associations entre les caractéristiques visuelles et des métriques logicielles directes (qui ne dépendent pas d'autres métriques) pour avoir des métriques faciles à interpréter, précisément définies et qui quantifient une seule caractéristique.

Notons néanmoins que ce type de visualisation permet une mise à l'échelle pour des systèmes relativement grands [Gre 05]. Ceci est en partie dû aux formes géométriques simples qui sont utilisées pour représenter l'information. Elle dispose d'une sémantique

intuitive. Par exemple, dans le graphe d'un arbre d'héritage, les nœuds représentent les entités logicielles (tels que : packages, classes, aspects, etc.), et les arêtes entre ces nœuds expriment les relations d'héritage. Par sa nature, elle est enrichie par six métriques (hauteur, largeur, couleur, deux positions horizontale et verticale, et l'épaisseur des arêtes), reflétant des informations tirées de la structure des entités [Lan 03c].

- **Taille.** La hauteur et la largeur d'un nœud correspondent à deux métriques.
- **Couleur.** La couleur exprime une métrique sémantique (le type d'entité logicielle : classe abstraite, classe concrète, interface, aspect abstrait, aspect concret, etc.).
- **Position.** La position d'un nœud est exprimée par ces coordonnées (métriques) et elle est dépendante de la nature du graphe (ex. graphe ou arbre d'héritage).
- **Épaisseur de l'arête.** Dans certains graphes, des arêtes relient les nœuds entre eux. L'épaisseur de ces arêtes peut être utilisée pour exprimer une métrique entre deux entités (ex. dans un arbre d'héritage les arêtes représentent les relations d'héritage).

Toutes les caractéristiques visuelles de ce type de graphe dépendent des valeurs de ces métriques. Cette liberté au niveau de l'association métrique-représentation permet de créer une vue appropriée en fonction des besoins de l'utilisateur.

La **Figure 3.1** montre un exemple simple de cette visualisation. Cette figure est partiellement inspirée de [Lan 03c].

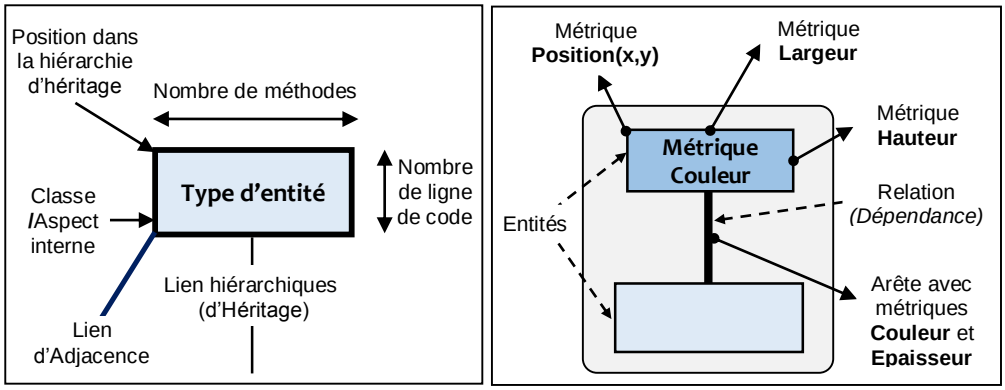


Figure 3. 1 Vue polymétrique de visualisation.

✪ *Métaphore de la Ville*

Une métaphore de plus en plus privilégiée est celle de la ville (en anglais : *City Metaphor*). Ce genre de visualisation permet une compréhension à grande échelle du système tout entier grâce à l'analogie qui est faite avec le monde réel. En 1993, *Dieberger* propose de représenter l'information en utilisant une métaphore de ville pour résoudre les problèmes de navigation

qui surviennent dans un monde 3D [Die 94, Die 98]. *Knight* et *Munro* ont fait partie des premiers à représenter les logiciels en utilisant une métaphore de ville appelée *SoftwareWorld* [Kni 00, Cha 02].

Bien qu'il existe plusieurs études qui ont été faites à l'aide de cette métaphore [Die 98, Kni 00], certaines restent superficielles en se contentant d'avoir une correspondance graphique entre les éléments. D'autres ne sont pas orientées sur la qualité du logiciel mais plutôt sur la représentation des entités logicielles [Lan 05c].

La contrainte principale de la métaphore de la ville est que cette dernière est représentée sur un seul plan [Lan 08], ce qui est problématique pour la représentation des relations entre éléments. Même si certains travaux tentent de donner de la hauteur à la métaphore de la ville, comme par exemple la métaphore des iles et des villes où les iles sont représentées à différents niveaux d'élévation [Pan 07]. Cette métaphore de ville reste assez plate et même si la visualisation de la ville n'est pas forcement totalement réaliste, nous considérons, comme [Yan 03], que certains écarts avec la réalité ne gênent pas la compréhension. Au contraire, nous pensons que simplifier les formes géométriques et retirer des détails inutiles permet de se focaliser sur les informations importantes et de diminuer la charge cognitive.

La métaphore de la ville a été très largement étudiée et a donné lieu à beaucoup de publications scientifiques [Lan 05a, Wet 07, Ala 07, Pan 07, Dha 08, Wet 08b]. Des expérimentations sur l'utilisation d'un outil de visualisation CodeCity utilisant ce type de métaphore montrent l'amélioration des performances des tâches réalisées sur le logiciel étudié par (+24%) ainsi que le temps utilisé pour réaliser ces tâches par (-12%), par rapport à l'utilisation classique à travers l'environnement de développement intégré ECLIPSE pour explorer le code source et avec l'outil Excel pour explorer les métriques [Wet 11]. D'autres expérimentations avec l'outil VERSO montrent aussi que l'utilisation de ce type de métaphore aide à détecter les anomalies de conception [Lan 07, Dha 07].

La métaphore de la ville n'a pas forcément besoin de correspondre exactement à la réalité, quelques écarts sont permis et même souhaitables [Yan 03]. En effet, nous considérons qu'une visualisation simplifiée de la ville aide à mieux se focaliser sur les informations importantes, sans être distrait par des objets visuels réalistes qui sont sans valeur ajoutée pour les vues. Une capture d'écran d'une métaphore de ville sous l'outil CodeCity [Wet 07] est montrée à la **Figure 3.2**.

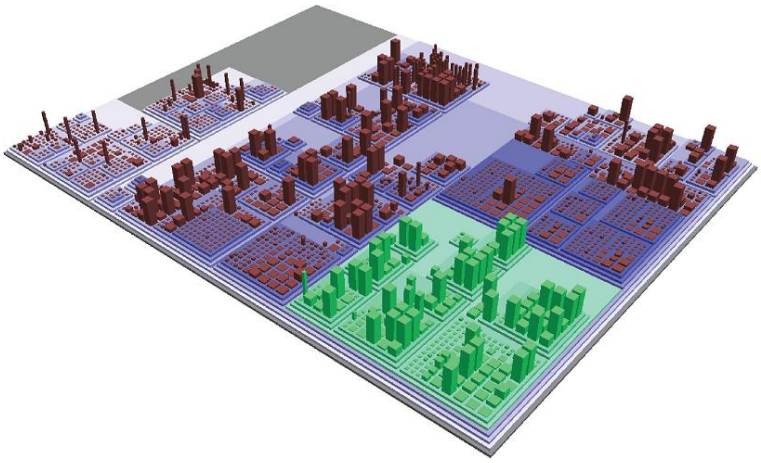


Figure 3. 2 Visualisation à l'aide d'une métaphore de ville.

Dans ce qui suit, nous présentons un aperçu de quelques travaux antérieurs que nous avons sélectionnés comme des références dont l'inspiration de nos travaux et dont les fondements théoriques sont essentiels à notre proposition.

3.2.3 Travaux Connexes

Il existe aujourd'hui un grand nombre de travaux dans le domaine de la visualisation des logiciels ce qui montre que c'est un domaine très actif et qu'il existe un réel engouement de la part des chercheurs mais également du monde industriel en ce qui concerne les techniques et systèmes de visualisation destinés aux grands systèmes logiciels [Cas 12].

Ce domaine a beaucoup évolué ces dernières années, ce qui a permis d'apporter aux développeurs de nouveaux outils supportant l'analyse, la compréhension et le développement des systèmes complexes. Même si de nouveaux outils ont atteint le niveau d'outil commercial comme **SolidFX** [W16] (une visualisation qui aide à la re-conception des programmes C et C++), la visualisation n'est pas encore largement répandue au niveau de l'industrie. Ils ont souvent un gros problème de passage à l'échelle lorsqu'il s'agit de conquérir le monde industriel où il y a des logiciels de grande taille. En effet, la représentation par exemple grâce aux techniques basé-graphes n'est pas une chose facile car ces graphes deviennent illisibles, à cause du grand nombre de nœuds et d'arêtes [Cas 12].

Des solutions substantielles sont souvent nécessaires pour transformer les outils prototypes en produits commerciaux. En effets, les chercheurs utilisent très souvent des programmes open sources disponibles sur internet pour tester les techniques de visualisation de ces prototypes. Alors qu'en pratique, les systèmes industriels peuvent être de grande taille et plus complexes et donc bien plus difficiles à analyser et visualiser. Pour contourner ces problèmes, chaque année de nouvelles techniques sont proposées, tels que le filtrage par suppression de nœud [Ril 05] et l'agrégation au niveau des entités logicielles [Hol 07], ainsi que d'autres styles d'interaction qui permettent d'interagir plus efficacement avec les diverses

vues. Ces dernières années, une tendance de recherche se dégage pour l'exploitation d'une troisième dimension. Même si la navigation reste un peu difficile au sein d'un espace 3D, des résultats prometteurs ont été obtenus notamment à travers l'utilisation des métaphores du monde réel. Il y a donc de grandes opportunités de recherche, et d'industrialisation, dans ce domaine [Cas 12].

Nous avons choisi de présenter ici quelques outils qui sont représentatifs soit de ce qui est actuellement utilisé soit d'idées qui nous semblent caractéristiques de ce qui sera sans doute conservé dans les outils éventuels au futur.

CodeCity [W15], est un outil de visualisation en 3D développé par *Richard Wettel* et *Michael Lanza*. Comme son nom l'indique, cette visualisation s'appuie sur la métaphore de la ville. Les paquets (en anglais : *Packages*) sont des districts (ou quartiers) représentés grâce à un *Treemap* (un type de visualisation d'une donnée hiérarchique) et les classes sont représentées par des constructions (en anglais : *Buildings*) placés dans le district correspondant à leur paquet. Ces constructions sont caractérisées par cinq attributs visuels (dimensions, position, couleur, saturation, et transparence). Chaque attribut visuel correspond à une métrique logicielle. Une capture d'écran d'une vue en 3D sous cet outil est montrée par la **Figure 3.3** [Wet 08a].

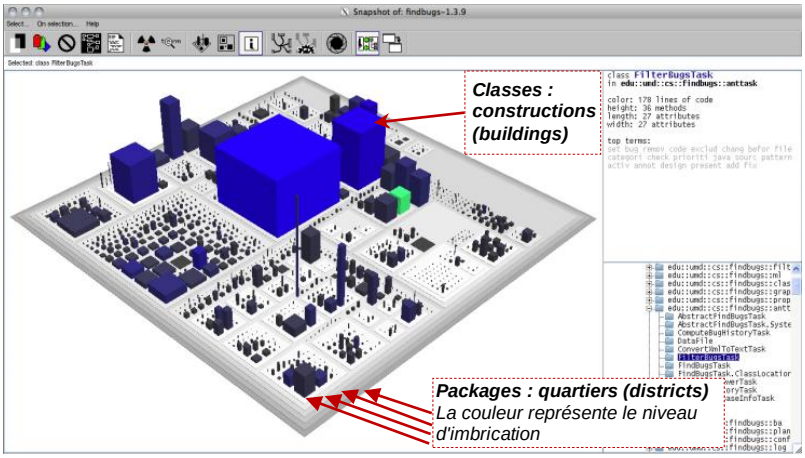


Figure 3. 3 Exemple de la visualisation dans CodeCity.

Citylyzer [Bia 07, W14] est un outil de visualisation en 3D pour les programmes JAVA ; développé par *Andria Biaggia* sous forme de plug-ins ECLIPSE. Il s'appuie sur la métaphore de la ville d'une manière un peu similaire à celle utilisée sous l'outil CodeCity.

VERSO (Visualisation pour l'Évaluation et la Rétro-ingénierie du SOftware), développé par *Langelier* et *al.* [Lan 05b, Lan 06], est un système qui permet de visualiser dans un espace en 3D la hiérarchie logique de modules (paquetages, classes, etc.) d'un logiciel en utilisant la visualisation *Treemap* ainsi que la valeur de certaines métriques calculées sur les classes. La **Figure 3.4** montre un exemple de cette visualisation.

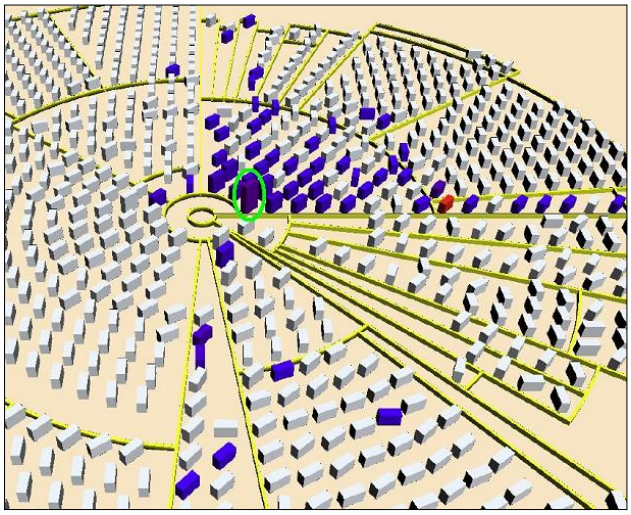


Figure 3. 4 Exemple de la visualisation dans VERSO.

Une des visualisations récentes, est **3D-HEB** (3D Hierarchical Edge Bundles) développé par *Caserta* et *al.* [Cas 11]. Cette technique affiche simultanément, dans un espace en trois dimensions, la hiérarchie logique et les liens statiques de dépendances (appelés d'adjacence) qui existent entre les classes. La structure hiérarchique est affichée en utilisant la métaphore de la ville, comme dans les outils VERSO et CodeCity. Les classes sont représentées par des boîtes et la structure hiérarchique est affichée à l'aide d'une représentation par inclusion. Dans leur visualisation, deux placements sont offerts pour représenter la hiérarchie d'un logiciel : un placement par imbrication et un placement nommé Street Layout, qui représente la hiérarchie de paquetages (*Packages*) comme des rues perpendiculaires. Les liens d'adjacence sont affichés en intégrant la technique HEB (Hierarchical Edge Bundles) dans chacun de leur placement, et ils utilisent la métrique couleur pour quantifier les liens ainsi que pour afficher leur direction. La **Figure 3.5** montre un exemple de cette visualisation.

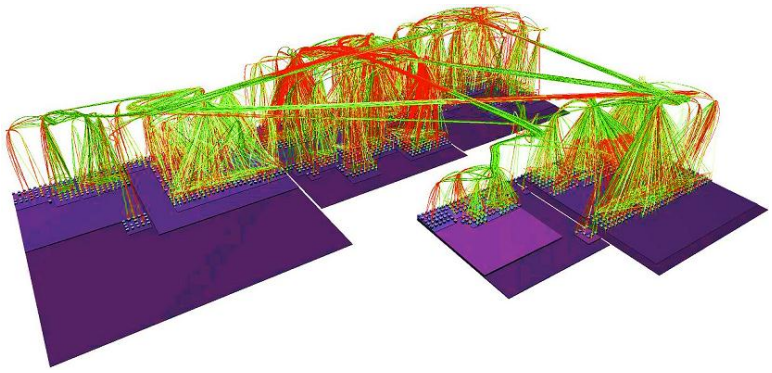


Figure 3. 5 Exemple de la visualisation 3D-HEB.

Finalement, on peut dire qu'aucun outil de visualisation n'était capable de réaliser toutes les tâches de compréhension. C'est le niveau d'abstraction des vues produites qui affecte

grandement le succès de la réalisation des tâches de compréhension. Pour plus de détails sur d'autres travaux, des informations complémentaires pourront être trouvées par exemple dans [Die 07], ouvrage de référence dédie au domaine, ou encore dans [W26], mais également dans les références données dans la bibliographie présente à la fin dans ce manuscrit.

La section suivante est consacrée à la programmation visuelle. On présente tout d'abord, les notions du domaine en donnant quelques généralités théoriques que nous utiliserons par la suite. Nous terminons par un survol des principaux travaux existants.

3.3 Programmation visuelle

3.3.1 Introduction et terminologie

Pour l'être humain, l'utilisation des images était et reste toujours la façon la plus simple pour l'expression et la communication. Dès l'enfance, il apprend la vie par les images et durant toute sa vie il pense en formant des images dans son esprit. Les recherches dans le domaine de la programmation visuelle considèrent cette perception visuelle humaine pour concevoir des systèmes et des langages de programmation qui utilisent des images, des objets simples ou des constructions graphiques. Ces recherches sont favorisées par l'avancement technologique du matériel en particulier au niveuau des écrans graphiques (couleurs, résolution,...) [Rav 02].

De nos jours, ce développement doit s'accompagner d'un développement des outils avancés de programmation. Ceci nécessite, d'un côté, une évolution des interfaces homme-machine qui doivent être compréhensibles par un grand nombre d'usagers (rendant ainsi, la programmation de l'outil la plus claire possible). De l'autre côté, une évolution de la programmation est nécessaire. En effet, on a vu se développer les outils supports visant à simplifier les diverses tâches de programmation qui étaient longtemps réservée à des spécialistes. Il est clair que notre environnement a changé, et aujourd'hui un nombre sans cesse croissant de machines doivent être paramétrées et même programmées. La question qui se pose est : comment rendre la programmation accessible à une large gamme d'usagers, y compris les personnes moins spécialisés ou débutants ?

De ce fait, l'idée principale est d'explorer de nouveaux paradigmes afin de rendre la tâche de programmation plus accessible. Mais aussi afin de construire des langages et outils supports avancés permettant d'améliorer la vitesse de spécification et construction des programmes en utilisant les avantages de l'approche visuelle (c.-à-d. syntaxe moins lourde, sémantique plus riche des primitives visuelles, feedback visuel immédiat à travers des techniques d'animation ...) réduisant ainsi l'écart entre le processus mental de résolution du

problème et la programmation effective. Dans ce contexte, les domaines d'interface homme-machine et des langages visuels apportent des éléments promoteurs de réponse.

Nous abordons donc un vaste domaine, et pour le présenter nous allons au préalable préciser la signification des termes : langage visuel, programmation visuelle et visualisation des programmes. Nous poursuivrons par la discussion de quelques avantages et problèmes liés au domaine. Ensuite, nous présenterons des classifications tentant de le structurer.

Il existe certaines confusions terminologiques entre : langage visuel (VL*, Visual Language*), programmation visuelle (VP*, Visual Programming*) et visualisation des logiciels ou des programmes (PV*, Program Visualization*).

✪ *Langage visuel (VL)*

Plusieurs chercheurs (p. ex. *Glinert* [Gli 85] et *Scanlan* [Sca 89]) ont établi d'une manière empirique, l'avantage cognitif des techniques visuelles (ou graphiques) par rapport aux techniques textuelles. Le domaine de programmation visuelle regroupe des recherches sur les langages de programmation, les systèmes graphiques, et les techniques d'interaction homme-machine. La classification d'un langage visuel est possible après des études en se basant sur différents points tels ques : les représentations visuelles, les paradigmes, les caractéristiques du langage et son utilisation [Rav 02].

Myers [Mye 90] définit le langage visuel comme un langage à base de représentations graphiques. Cela inclut donc la programmation visuelle et les systèmes de visualisation des programmes. Selon cette définition, on constate qu'il y a deux catégories de langages visuels : langages de programmation visuelle et langages supportant la visualisation (ou langages de visualisation). Certains langages font partie de ces deux catégories à la fois. La différence entre ces deux catégories peut être basée sur le moment où les éléments visuels (ou graphiques) interviennent. C'est lors de la phase de spécification et construction pour les langages de programmation visuelle et à la phase d'exécution pour les langages de visualisation [Rav 02].

✪ *Programmation visuelle (VP)*

La programmation visuelle est un vaste domaine qui est relativement jeune par rapport à la programmation conventionnelle textuelle. La programmation visuelle a été rendue possible seulement depuis l'apparition des interfaces graphiques (GUI, *Graphical User Interface*) performantes supportant des résolutions capables d'afficher simultanément un nombre significatif d'objets et différentes vues. Son objectif principal est de mettre en valeur la compréhensibilité des programmes et de simplifier le processus de programmation lui-même. Un nombre de chercheurs ont abordé ce domaine en suivant des approches différentes. Dans tous les cas, il semble difficile de trouver une définition faisant l'unanimité dans ce domaine. Cependant dans ce qui suit nous en distinguons certaines qui semblent être admises par un grand nombre de chercheurs.

La définition probablement la plus générale de la programmation visuelle ou graphique est celle donnée par *Shi-Kuo Chang* et reprise dans [Cha 90] : c'est l'utilisation de notations ou d'expressions graphiques significatives (ex. icônes, dessins, entrées gestuelles, ...) manipulables interactivement pour construire des programmes. C'est la représentation visuelle d'entités conceptuelles et d'opérations durant le processus de programmation [Cha 95]. Cette définition est adoptée par d'autres chercheurs [Koe 92, Rav 02]. Le but est de simplifier les divers tâches de programmation et de mettre en valeur la compréhensibilité des programmes. Les outils supports de programmation visuelle fournissent donc des éléments graphiques qui peuvent être manipulés d'une manière interactive par l'utilisateur en formant des expressions visuelles en fonction d'une grammaire spatiale spécifique pour définir et composer des programmes. L'objectif est de mettre en valeur la compréhensibilité des programmes et de simplifier la programmation elle-même [Leb 99].

Nan C. Shu [Shu 88, Shu 89], précise qu'il y a plusieurs aspects au sein du processus de programmation et que la programmation visuelle peut s'appliquer à chacun. Les différents aspects considérés sont [Leb 99]:

- Le langage et l'environnement utilisés
- L'écriture même du programme
- La détermination si oui ou non l'ordinateur a effectué ce qui était prévu
- L'affichage des données lors de l'exécution.

Le point de vue de *Myers* [Mye 90] est différent. La programmation visuelle se rapporte à tout système contient différentes approches graphiques permettant de spécifier les programmes dans un mode multidimensionnel au moyen au moins de deux dimensions. Bien que cette définition soit très large, les langages textuels conventionnels (TPLs, *Textual Programming Languages*) manipulant des données visuelles, ainsi que les systèmes qui ne gèrent qu'une visualisation des programmes ou des données ont été exclus, puisque ceux-ci sont définis de façon linéaire (unidimensionnelle) [Leb 99, Rav 02].

Différentes propositions de définition de ce qu'est un langage de programmation visuelle (VPLs, *Visual Programming Languages*) ont été faites :

- Selon *Golin* [Gol 90], le langage visuel manipule l'information visuelle ou supporte l'interaction visuelle, ou bien encore s'appuie sur les manipulations d'éléments syntaxiques visuels (ou d'expressions visuelles) pour exprimer un programme (ou une spécification). Cette dernière possibilité est considérée comme étant la définition d'un langage de programmation visuelle.
- *Lakin* [Lak 86], considère que le langage visuel est un ensemble d'arrangements spatiaux de symboles textuels et graphiques avec une interprétation sémantique utilisée pour exécuter des actions de communication.

- Pour *Burnett* [Bur 89], les langages visuels ont des expressions visuelles naturelles pour lesquelles il n'y a pas d'équivalents textuels évidents.
- Selon *Newton* [New 90], les langages de programmation visuelle sont ceux qui utilisent des éléments graphiques pour représenter directement la structure ou la logique du programme. Le programme est écrit dans un langage d'objets et de constructions graphiques. Ce n'est pas le cas pour les outils qui permettent de visualiser graphiquement la structure des programmes écrits dans des langages traditionnels non visuels, et même pour les outils qui sont conçus comme support d'aide au développement d'interface graphique (GUI).
- Selon *Burnett* et *McIntyre* [Bur 89], la programmation visuelle est définie comme l'utilisation d'expressions visuelles significatives en mode interactif dans le processus de programmation. Ces expressions peuvent être utilisées dans des environnements de programmation comme les interfaces graphiques pour les langages de programmation textuelle, ou dans des présentations graphiques du comportement ou de la structure d'un programme. Elles peuvent aussi être utilisées pour spécifier la syntaxe de nouveaux langages de programmation visuelle conduisant à de nouveaux paradigmes (ex. programmation par démonstration).

Ainsi, il n'est pas facile de donner une définition générale de ce qu'est un langage de programmation visuelle, ce terme étant parfois utilisé de manière abusive pour désigner des approches très différentes. La proposition de *Lakin* ainsi que celle de *Burnett* et *McIntyre* ont été retenu, par contre les autres paraissant trop floues ou ne donnant pas de véritable définition, mais les moyens de les différencier des langages traditionnels de programmation textuelle [Pie 02].

Les langages de programmation visuelle (VPLs) se différencient des langages traditionnels (TPLs) par deux caractéristiques principales:

- **Le système de représentation des éléments de base du langage.** Il autorise l'utilisation plus d'une dimension perceptuelle (c.-à-d. paramètres physiques perçus directement par humain) que la forme pour coder un élément de l'alphabet du langage, telles que la couleur et la taille des objets formant les éléments de base (les équivalents des unités lexicales d'un TPL) ainsi que les relations spatiales liant ces objets (ex. inclusion, proximité, placement…) ou encore le temps, contrairement aux langages textuels qui n'utilisent que la forme des glyphes pour différencier les éléments. Ces derniers peuvent donc être les glyphes de caractères, mais aussi des formes géométriques comme les cercles, les triangles ou des entités (ou objets) iconiques simples ou des constructions graphiques plus complexes [Lak 86, Pie 02].

- **La multi-dimensionnalité.** Dans les langages textuels, il existe des contraintes fortes sur la manière de placer les éléments de l'alphabet pour former les unités lexicales et les expressions du langage. Ils doivent être juxtaposés suivant une direction donnée, formant ainsi un flot unidimensionnel de caractères. Les langages visuels s'affranchissent de cette contrainte (c'est la différence essentielle avec les langages textuels), permettant ainsi la création d'expressions visuelles utilisant plus d'une dimension et en fonction d'une grammaire spatiale spécifique pour coder la sémantique [Lak 86, Pie 02].

De plus, il ne faut pas confondre les langages de programmation visuelle avec les langages de programmation pour le traitement d'informations visuelles, qui sont souvent des langages textuels permettant la manipulation des informations ayants des représentations graphiques inhérentes [Pie 02].

Un langage de programmation visuelle a les propriétés suivantes [Rav 02]:

- Il décrit des objets spécifiques et leurs comportements (ex. structures de données, objets graphiques, etc.),
- Il utilise des éléments lexicaux graphiques manipulables en mode interactif et éventuellement des éléments textuels,
- Il a une syntaxe spatiale et au moins une partie de la sémantique est déterminée par la position et les relations spatiales des éléments graphiques.

Les langages de programmation visuelle (VPLs) sont souvent confondus avec les environnements visuels de programmation (VPEs, *Visual Programming Environments*), ces derniers fournissent une interface graphique et des éléments visuels aidant à la construction de programmes exprimés dans des langages traditionnels de programmation textuelle, simplifiant les tâches de développement et facilitant notamment la spécification d'interfaces utilisateurs (GUI). On peut citer comme exemples : *Visual Basic*, *Visual C++*, *VisualWorks*, etc. Ils sont nommés "visuels" pour des raisons commerciales, bien qu'ils bénéficient des avancées venant de la recherche dans le domaine [Pie 02, Bos 04].

De nos jours, les VPEs représentent une part très importante des environnements commerciaux. Ils peuvent être considérés comme une étape intermédiaire entre les langages de programmation textuelle (TPLs) et visuelle (VPLs), apportant aux langages textuels certains avantages des interfaces graphiques, comme l'exécution de programme pas à pas avec un feedback visuel immédiat, la construction de l'interface graphique du programme lui-même par manipulation directe des composants, ou encore la possibilité d'accéder facilement aux fichiers d'aide. Les VPEs ne sont pas considérés comme de véritables langages de programmation visuelle, mais font office de canal de transfert pour une partie de la recherche sur les langages visuels [Pie 02, Zha 07].

Les VPLs sont habituellement employés avec des environnements visuels. Le but principal de la recherche sur les VPLs est d'améliorer le processus de programmation tout en réduisant les efforts mentaux du programmeur. Il s'agit de [Pie 02] :

- Rendre la programmation plus accessible aux personnes ayant des connaissances dans le domaine du problème à résoudre, et qui ne sont pas forcément des professionnels ou experts ;
- Améliorer la justesse de la programmation ;
- Améliorer la vitesse à laquelle la tâche de programmation est accomplie (productivité).

Selon *Burnett* [Bur 99], pour arriver à ces fins, dans n'importe quelle stratégie de mise en œuvre, un VPL doive se caractériser par certaines des caractéristiques communes entre sa syntaxe et l'environnement support [Pie 02].

- **Description concrète.** C'est-à-dire essayer d'exprimer certains aspects (ou propriétés) du programme au moyen d'instances ;
- **Simplicité (être direct).** L'usager doit avoir l'impression de manipuler directement les objets visuels par les actions associées ;
- **Description explicite.** Un aspect de la sémantique est explicite dans l'environnement s'il est déclaré directement d'une façon textuelle ou visuelle, sans que l'usager n'ait à l'inférer ;
- **Feedback visuel immédiat.** Afficher automatiquement les conséquences de l'édition de parties du programme. Le terme "liveness" [Tan 90] catégorise la réponse immédiate avec laquelle un retour sémantique visuel est automatiquement fourni pendant l'édition.

Puisque les VPLs ont moins de restrictions sur la façon d'exprimer les programmes d'un point de vue syntaxique, il devient possible d'explorer de nouveaux mécanismes de programmation qui n'étaient pas envisageables dans le passé, mais qui sont devenus possibles aujourd'hui par la montée en puissance de la technologie matérielle offrant des interfaces graphiques de haute-résolution et des environnements supportant le multi-fenêtrage [Mye 96].

✪ *Visualisation des programmes vs. Programmation visuelle*

Les langages de programmation visuelle (VPLs) doivent être distingués des environnements de visualisation (visualisation des programmes et visualisation des données). *Shi-Kuo Chang* [Cha 90], définit la visualisation comme l'utilisation de représentations visuelles

(ex. images, graphiques, animations) pour illustrer la structure (les données), le comportement et l'évolution des programmes. En d'autres termes, la visualisation des programmes (PV) est une représentation graphique des aspects d'un programme qui a pu avoir été écrit dans un langage conventionnel de programmation textuelle (TPL). La visualisation des données offre une vue des données manipulées par le programme [Rav 02].

Myers [Mye 90], considère que la programmation visuelle (VP) et la visualisation des programmes (PV) sont deux concepts totalement différents. En programmation visuelle, les représentations graphiques sont utilisées pour créer des programmes, alors que la visualisation utilise ces représentations pour illustrer certains aspects du programme, ou son exécution [Leb 99].

La **Figure 3.6** (cette figure est tirée de [Die 07]) illustre le champ de programmation visuelle et celui de la visualisation des programmes. La visualisation génère des vues relatives aux caractéristiques des programmes, alors que la programmation visuelle produit des programmes avec des caractéristiques visuelles (programme visuel).

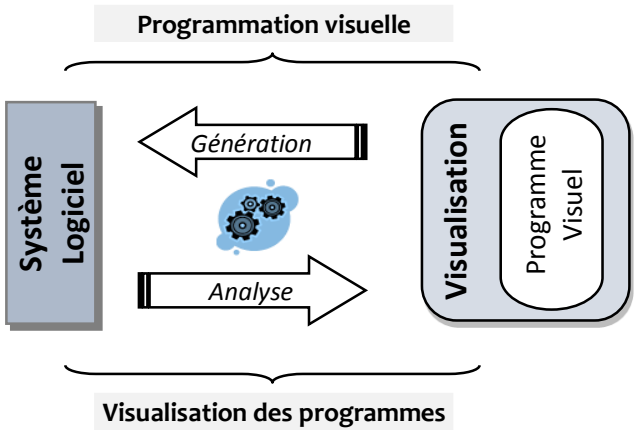


Figure 3. 6 Programmation visuelle versus visualisation des programmes.

Stephan Diehl [Die 07], considère la combinaison de ces deux approches comme une ébauche de recherche pour faire émerger une nouvelle génération d'environnements de développement (c.-à-d. *a Round-Trip Visual Engineering System*). Par exemple, produire une représentation visuelle du code source d'un programme (PV), changer cette vue (VP), et produire un autre code du programme.

3.3.2 Avantages et inconvénients

Nous récapitulons les principales avantages de la programmation visuelle comme suit [Rav 02].

- **Un raisonnement visuel.** la plupart des activités cognitives est accompagnée d'une visualisation mentale. Les VPLs peuvent aider les usagers à penser visuellement, par exemple en voyant des chemins plutôt qu'en les exprimant. Ils peuvent servir de moyen de communication pour le résultat d'un tel raisonnement.
- **Une ouverture vers des usagers moins spécialisés.** Les spécialistes d'un domaine applicatif ont rarement des compétences informatiques et plus particulièrement en programmation. Les techniques visuelles facilitent l'interaction homme-machine et aussi la création de programmes—tâche complexe réservée traditionnellement aux spécialistes.
- **Amélioration de la qualité des programmes.** La programmation visuelle permet aux programmeurs professionnels et experts de mieux se concentrer sur la résolution du problème et sur les innovations plutôt que sur les formalismes syntaxiques durant le développement du code source.
- **Enlèvement de la barrière linguistique.** Les expressions visuelles seront utiles pour ceux qui ont des problèmes avec l'anglais en général, puisque la plupart des langages conventionnels de nos jours tendent à être basés sur cette langue.

Selon *Burnett* [Bur 99], les buts spécifiques les plus communs des recherches dans le domaine de programmation visuelle sont : (1) rendre la programmation plus accessible à certaines catégories particulières, et (2) améliorer l'exactitude et la vitesse avec lesquelles les usagers accomplissent leurs tâches de programmation. Malgré un certain attrait pour leur côté intuitif, les VPLs présentent quelques inconvénients qui posent aujourd'hui des problèmes importants [Rav 02] :

- Difficultés d'abstraction et de représentation de certaines données ou structures de contrôle. Par rapport aux VPLs, les TPLs restent plus généraux et plus capables d'exprimer des abstractions. On peut discuter de tout, mais on ne peut pas tout visualiser, par contre ce qu'on peut voir est mieux compris.
- Les VPLs sont bien adaptés pour la description des objets manipulés, mais ils ont plus de difficultés à exprimer le comportement. Ceci devrait être représenté par d'autres objets visuels abstraits.
- Il y a aussi des représentations graphiques qui dépendent de la culture. Donc, il faut se mettre d'accord sur la façon de représenter les choses. La normalisation pourrait remédier à ce problème.

- La résistance au facteur d'échelle (*Scalability*), c'est l'un des plus importants problèmes des VPLs. C'est la possibilité d'utiliser d'écrire des programmes de taille importante. La plupart des langages se comportent très bien avec de petits programmes de démonstration. Mais dès que la taille augmente, des problèmes de représentation, de navigation et de compréhension apparaissent.

  La programmation visuelle a besoin d'un large espace sur écran pour créer un programme. Ceci peut être résolu partiellement par l'utilisation de défilement dans une fenêtre et/ou par des mécanismes d'abstraction comme la représentation hiérarchique. Mais la compréhension du code peut être affectée par une visualisation partielle, car avec chaque élément invisible on perd aussi ses relations avec le reste des objets. Comme solution pour ce problème, la hiérarchisation couplée avec des représentations iconiques exige la définition des représentations pour des objets de plus en plus généraux et abstraits. Les recherches visant à trouver de nouveaux moyens d'adresser le problème de l'espace d'écran sont ainsi de plus en plus utiles.

3.3.3 Classifications

À la lumière des définitions précédentes, les approches de programmation visuelle sont nombreux et différentes, vraisemblablement à cause de la grande variété des systèmes existants. Afin de les structurer, nous en présentons deux classifications. La première s'attache aux langages visuels (classification de *Shi-Kuo Chang*), alors que la seconde propose un découpage de l'ensemble de processus de programmation (classification de *Nan C. Shu*) [Leb 99].

✪ *Classification de Shi-Kuo Chang*

Chang [Cha 87], propose une classification légèrement différente des langages visuels, basée sur deux critères : (1) la représentation des objets manipulés, et (2) les constructions du langage. Il considère deux types d'objets : les objets logiques auxquels on rattache une représentation visuelle et les objets avec une représentation visuelle inhérente auxquels on rattache une interprétation logique.

✪ *Classification de Nan C. Shu*

Selon *Nan C. Shu* [Shu 88, Shu 89], l'évolution de la programmation visuelle se fait selon deux axes : le premier correspond à l'utilisation des techniques graphiques et des systèmes de conception pour fournir un environnement visuel permettant de : (1) créer et exécuter des programmes, (2) retrouver et afficher des informations ou des données, et (3) concevoir et comprendre des programmes. Le second axe de développement étant la conception de langages visuels afin de : (1) traiter des informations visuelles, (2) supporter des interactions

visuelles, et (3) programmer avec des expressions visuelles significatives (objets et constructions graphiques) [Leb 99].

La **Figure 3.7** présente toutes ces tendances. Pour plus de détails et précisions sur chacun de ces sous-domaines se référer à [Mye 90, Leb 99].

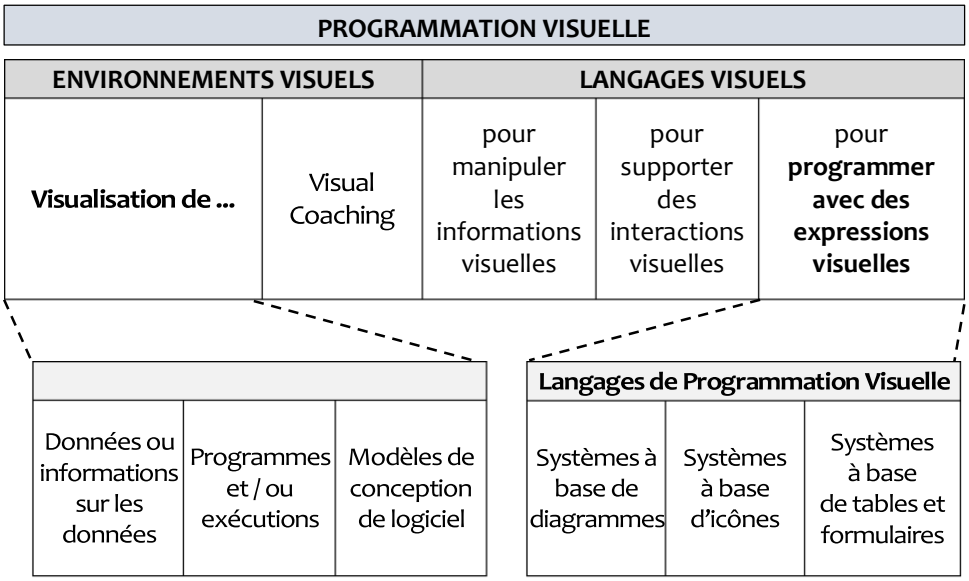


Figure 3. 7 La Classification de Shu [Shu 88].

Dans la section suivante, nous présentons brièvement les principaux travaux relatifs à notre axe de recherche et autour des techniques de construction visuelle des programmes et les outils supports qui sont liés à notre proposition.

3.3.4 Travaux Connexes

Depuis longtemps, l'introduction des techniques graphiques couvre un grand nombre de domaines qui considèrent l'aspect visuel sous différents angles et avec des perspectives variées. Il existe aujourd'hui un nombre important de travaux de recherche qui ont été portés sur la création de nouveaux outils supports permettant aux développeurs d'effectuer leurs tâches avec plus de souplesse et d'efficacité, et par conséquent avec moins d'efforts [San 06].

De nos jours, ils ont été introduits avec succès à des domaines d'application particuliers (DSLs, *Domain-Specific Languages and tools*), ainsi que dans le contexte éducatif comme des outils et assistants pédagogiques à la programmation. *Alice* [Kel 02, W4], est un outil pédagogique puissant permettant d'empêcher les erreurs syntaxiques à travers une interface graphique sur laquelle les programmeurs peuvent modifier directement l'arborescence de la syntaxe. *Limnor Studio* [W5] et *Tersus* [W6] sont les deux outils les plus matures qui emploient

le style d'interaction "Glisser-Déposer" (*Drag-and-Drop*) sous un éditeur visuel de code, mais pour des applications limitées.

Cependant, peu d'outils à usage général sont disponibles actuellement [Whi 02]. Certains d'entre eux sont destinés au paradigme "orienté-objets" tels que *Prograph* [Cox 89, Sco 95], *Visual Zero* [Per 08] et *Larch Environment* [Fre 13, W28]. Ce dernier est un environnement de programmation en langage Python qui utilise une approche hybride de programmation textuelle et visuelle en permettant à des objets visuels et interactifs d'être intégrés au sein du code source textuel, et même des segments de code pourraient être intégrés au sein de ces objets. *BioPro System* [Tak 05] et *Tersus* [W6] sont des exemples d'environnements de programmation visuelle qui ont prouvé leur succès dans le développement web.

La méthodologie de développement "*Codeless*" tels qu'elle est sous les deux projets *Barista* [Mye 06] et *Limnor Studio* [W5] représente une direction prometteuse pour un prototypage rapide des programmes avec un haut niveau d'interactivité. C'est un moyen plus pratique qui tente de ne pas se focaliser sur les formalismes syntaxiques en explorant l'idée d'une nouvelle génération d'éditeurs de code (*Code Structure Editors*), et en préservant la puissance de programmation traditionnelle. Cette méthodologie innovatrice représente une ébauche de recherche attrayante vers une cinquième génération des approches de programmation. Ces deux outils ont été sources d'inspiration pour notre proposition : on vise à imiter le style de développement "*Codeless*", mais d'une nouvelle façon.

A notre connaissance, bien qu'un nombre important de travaux se base sur l'approche visuelle, il n'y a aucun qui supporte les diverses implémentations orientées aspects existantes en exploitant les capacités des techniques interactives et visuelles durant la phase de codage.

Pour plus de détails sur d'autres travaux, des informations complémentaires pourront être trouvées par exemple dans [W27], cite de référence dédie au domaine, mais également dans les références données dans la bibliographie présente à la fin dans ce manuscrit.

3.4 Conclusion

Nous nous sommes limités, dans ce chapitre, à présenter une vue d'ensemble du domaine de visualisation des programmes et de programmation visuelle. Nous avons aussi résumé quelques techniques et travaux connexes à l'échelle industrielle et académique (issus des laboratoires de recherche). Cette présentation est synthétique et ne couvre pas tout le domaine, ni toutes les travaux actuels. Pour un état de l'art détaillé se référer à [Die 07, W26].

La deuxième partie de la thèse est réservée à décrire en détail nos approches et leurs contributions à la visualisation et à la programmation visuelle dans un contexte orienté-aspects.

CHAPITRE 4

CONTRIBUTION À LA VISUALISATION 2D ET 3D

SOMMAIRE

4.1 INTRODUCTION 42

4.2 VUE DETAILLEE DE L'APPROCHE 44

4.2.1 SYSTEME DE VISUALISATION 2D & 3D ORIENTEE-METRIQUES 44
4.2.2 PARAMETRES IMPORTANTS DE VISUALISATION 47
4.2.3 METRIQUES CONSIDEREES EN VISUALISATION 48

4.3 MISE EN ŒUVRE DE L'APPROCHE 55

4.3.1 VIZZASPECTJ-2D 55
4.3.2 VIZZASPECTJ-3D 61

4.4 CONCLUSION 65

Dans ce chapitre, on présente en détail notre première contribution. Elle concerne une approche d'analyse et de visualisation bidimensionnelle (2D) et tridimensionnelle (3D) des propriétés quantitatives des codes sources de grands systèmes logiciels orientés aspects. L'objectif étant d'effectuer des analyses qualitatives à partir de ces visualisations. Notre approche se base sur des vues polymétriques et des métaphores visuelles rendant les analyses plus intuitives et plus efficaces. Elle est mise en pratique à travers deux outils supports pour la plateforme ECLIPSE: "VizzAspectJ-2D" et "VizzAspectJ-3D" permettant respectivement une visualisation en deux dimensions et en trois dimensions.

4.1 Introduction

L'analyse approfondie d'un système logiciel est le seule moyen permettant de recueillir de nombreuses informations difficiles, voire impossibles, à obtenir autrement. Ces informations sont souvent complexes ce qui nous a poussé à travailler sur des techniques qui en permettent une meilleure assimilation [Cas 12]. Notre contribution se focalise principalement sur deux volets : l'analyse du code source et les techniques de visualisation bidimensionnelle (2D) et tridimensionnelle (3D) facilitant le processus cognitif de compréhension.

Dans notre approche de visualisation nous nous sommes focalisés sur le volet statique du logiciel en occurrence son code source plutôt que sur l'exécution [Ben 13]. Cette dernière représente le volet dynamique et constitue une des perspectives majeures de nos travaux futurs. Nous proposons donc une approche d'analyse et de visualisation en 2D et 3D pour des propriétés quantitatives des grands logiciels orientés aspects.

La compréhension de la structure organisant les entités d'un logiciel est une première étape indispensable dans la résolution de tâches d'analyse et de maintenance de celui-ci [Has 07, Bou 11]. Les logiciels orientés objets et orientés aspects organisent leurs entités de façon hiérarchique, par exemple, l'arbre d'héritage, l'architecture en paquetage de JAVA. Donc, la compréhension globale d'un logiciel commence avant tout par la compréhension de son organisation hiérarchique.

Afin d'être plus efficaces, les outils d'aide à la compréhension et à l'évaluation de logiciels qui exploitent des techniques de visualisation doivent être en mesure d'afficher de façon claire et sans ambiguïté la structure hiérarchique des entités logicielles. Ceci permet ainsi à un utilisateur non seulement d'extraire rapidement une vue d'ensemble des liens hiérarchiques entre les entités mais aussi de mettre en contexte le rôle d'une entité en particulier par rapport au reste du logiciel. En plus des liens d'héritage qui définissent la structure hiérarchique, il existe toute une gamme de liens exprimant les diverses dépendances existantes entre les entités logicielles que nous appelons liens d'adjacence. Ces liens sont indépendants de la structure hiérarchique et représentent par exemple les invocations entre méthodes [Bou 11].

Bien que l'affichage de la structure hiérarchique soit un bon début et une première étape importante pour une compréhension globale d'un logiciel, elle n'est pas suffisante pour être plus efficace en effectuant les diverses tâches de maintenance. Comprendre comment les éléments constituants le logiciel sont inter-reliées et comment ils communiquent entre eux sont tout aussi importants. Donc, on doit afficher de façon concurrente la structure hiérarchique organisant les entités logicielles visualisées ainsi que les relations de dépendances existants entre celles-ci afin que les vues de visualisation puissent fournir des informations utiles à l'accomplissement d'une bonne partie des tâches qu'une personne chargée de la maintenance voudrait effectuer sur le logiciel [Bou 11, Cas 12].

Peu de méthodes, jusqu'à présent, permettent de représenter simultanément ces deux types de relations, et encore moins de façon claire et concise lorsqu'utilisées pour visualiser des logiciels de grandes tailles. Récemment, Hierarchical Edge Bundles (HEB), est une technique de visualisation des relations en 2D en limitant les croisements d'arrêtes (ou liens) dans une structure hiérarchique. Elle s'est avérée efficace et a été utilisée depuis dans plusieurs outils. Par contre, son adaptation demeure approximative particulièrement quand les nombreuses propriétés des entités logicielles doivent être également représentées [Hol 06, Bou 11].

D'autre part, la structure hiérarchique d'un logiciel correspond à un graphe hiérarchique, qui est un cas particulier de graphe orienté. La relation parent-enfant entre les entités (nœuds) oriente les liens et apporte des contraintes supplémentaires qui permettent de réorganiser de façon systématique l'emplacement des entités afin de clarifier la visualisation des liens existants entre eux. L'arborescence est l'une des méthodes les plus connues pour faire cela. [Her 00, Noa 05, Bou 11].

Une compréhension globale des grands logiciels doit donc tenir compte de tous les types de dépendances (liens hiérarchiques et de dépendances) ce qui revient à visualiser un très grand graphe avec beaucoup d'arêtes de différentes natures. Sans des techniques efficaces pour gérer le positionnement des nœuds ainsi que pour la représentation des arêtes, le graphe produit ne serait pas compréhensible à cause du chevauchement des arêtes qui recouvriraient les nœuds, rendant ainsi le grpahe très confus et donc la vue de visualisation peu efficace pour fournir des informations utiles [Bou 11].

De plus, une telle représentation bidimensionnelle (2D) rend presque impossible l'investigation d'une seule dépendance en particulier. Une solution qui aide à éviter ce problème en offrant une liberté au niveau du positionnement des nœuds est d'exploiter une dimension supplémentaire en représentant le graphe dans un espace en 3D [Fro 06]. L'utilisateur peut ensuite naviguer au sein d'un environnement 3D pour trouver une vue sans encombrement qui lui permet d'examiner et de comprendre facilement les diverses dépendances entre certains constituants [Cas 12].

La première problématique que nous cherchons à résoudre dans cette thèse est donc de pouvoir visualiser, de façon claire et efficace, à la fois la structure hiérarchique d'un programme orienté-aspects, les propriétés de ses entités et les diverses dépendances qui existent entre ces dernières. Il est aussi important que notre système de visualisation fonctionne avec des grands systèmes logiciels et offre diverses méthodes d'interaction afin de permettre à l'utilisateur de naviguer efficacement à travers les données qui lui sont présentées [Bou 11].

4.2 Vue détaillée de l'approche

4.2.1 Système de visualisation 2D & 3D orientée-métriques

Une des exigences initiales de notre approche est de permettre à un développeur ou un mainteneur de visualiser un très grand nombre d'entités d'un logiciel ou un échantillon provenant de plusieurs logiciels sans que son système visuel ne soit surchargé par un grand nombre d'éléments graphiques et de relations [Lan 05c].

Un système de visualisation doit être intuitif et facile à interpréter. Cependant, la plupart des systèmes actuels sont inefficaces dès que le nombre d'entités est de l'ordre de centaines [Lan 05c]. Intuitivement, il suffit d'associer une forme graphique (p. ex. une boîte rectangulaire) à une entité logicielle (package, classe, interface, aspect) et d'affecter les attributs visuels correspondants (taille, couleur, position, etc.) aux différentes métriques logicielles calculées de ces entités. Dans ce qui suit, nous donnons une description synthétique de notre système de visualisation proposé.

Dans la littérature, il existe plusieurs centaines de métriques logicielles. La représentation de chacune de ces métriques nécessite de trouver un attribut graphique à lui associer. Donc, nous devons retenir un nombre raisonnable (le plus grand possible) d'attributs graphiques qui préserve l'efficacité de la perception. Ce nombre est toutefois limité car de nombreux travaux ont montré qu'au-delà de quatre attributs graphiques (incluant la position), la performance de la perception chute dramatiquement [Hea 98, Lan 05c].

Une métrique logicielle (en anglais : *Software Metric*) est une mesure issue des propriétés techniques ou fonctionnelles du logiciel [Fen 03]. Son but est de quantifier une caractéristique particulière du logiciel telle que la complexité, la structure, le nombre de ressources utilisées ou la stabilité du système. Les métriques logicielles sont intéressantes parce qu'elles apportent des informations sur la qualité de la conception et contribuent ainsi à la gestion de cette qualité durant le processus de développement [Kan 02, Wet 11, Cas 12].

Les visualisations permettent de transformer les valeurs numériques des métriques en caractéristiques visuelles. La jointure des vues avec ces métriques permet d'éliminer le besoin des tables métriques énormes pour les tâches d'interprétation. De plus, ceci permet à l'utilisateur d'avoir une meilleure perception et donc une compréhension plus rapide de l'information [Cas 12]. Cependant, afin de faciliter le processus cognitif, le principal défi est de trouver une association efficace entre la métrique et caractéristique visuelle (c.-à-d. visualisation orientée-métriques) [Irw 03].

L'ensemble de métriques permettant de mesurer la qualité des logiciels peuvent être divisées en trois sous-catégories [Fen 00, Xen 00, Cas 12] :

- **Métrique produit** : Elle est calculée à partir du code source et quantifie certains aspects du logiciel tels que la taille, la complexité, le couplage, etc.

- **Métrique processus** : Elle est liée au processus de développement et donne des informations sur certains aspects du processus de développement tels que le temps, l'effort investi, etc.
- **Métrique projet** : Elle est liée au projet lui-même tels que les métriques d'ordonnancement, de coût, du nombre de ressources humaines, etc.

En générale, ces métriques aident les développeurs et les personnes chargées de la maintenance à mieux comprendre, contrôler, gérer, prévoir et améliorer le processus de développement et de maintenance, notamment pour les systèmes logiciels complexes. Les métriques liées au produit lui-même sont particulièrement intéressantes du point de vue des développeurs parce qu'elles sont calculées à partir d'une analyse du code source ou de l'exécution du logiciel. Ces métriques apportent donc des informations plus précises sur l'état du code source et de son exécution [Cas 12].

Pour une analyse qualitative, nous nous intéressons à la visualisation des propriétés quantitatives des entités logicielles telles que les classes, les interfaces, les aspects, etc. Pour représenter ces entités nous avons donc opté pour des objets graphiques simples comme par exemple des boîtes rectangulaires, des constructions, etc. Outre le fait que ce sont des formes simples dont l'affichage est moins coûteux, les attributs visuels expérimentés, et plus spécifiquement leurs variations, sont faciles à distinguer et leur association avec certaines métriques est intuitive. Par exemple, la métrique hauteur est efficace pour représenter une mesure de la taille d'une entité.

La visualisation de l'architecture est l'un des sujets les plus étudies dans le domaine de la visualisation des logiciels. Beaucoup de techniques traitent ce type de visualisation [Den 02, Hat 04, Car 08, Cas 12]. Un logiciel orienté aspects est structuré hiérarchiquement, avec des paquets (Packages) qui contiennent récursivement des sous-paquets, puis des entités logicielles (classes, interfaces et aspects) qui contiennent des méthodes, attributs, points de coupure et consignes. Il est donc logique de vouloir utiliser cette structuration pour représenter les constituants du logiciel. De plus, les relations de dépendances (Relationships) telles que les liens d'héritage, d'appel de méthodes, etc. représentent des caractéristiques importantes de l'architecture. De ce fait, la visualisation de l'architecture consiste à visualiser cette structure hiérarchique et les relations entre les divers constituants [Wis 99, Cas 12].

Dans notre système de visualisation, nous nous focalisons sur les techniques de visualisation qui s'intéressent à visualiser trois propriétés différentes du logiciel. La première propriété est l'organisation hiérarchique globale d'entités constituantes (packages, classes, interfaces, aspects, etc.). La deuxième est relative aux liens de dépendances entre les entités, qu'elles soient de type héritage ou graphe d'appel. La troisième concerne les métriques logicielles.

✪ *Visualisation de l'arborescence*

L'arborescence d'un logiciel est l'organisation des répertoires (paquets), des différents fichiers dans les répertoires (classes, interfaces et aspects), des méthodes, etc. Ce type de visualisation suit une certaine logique d'organisation, regroupant les constituants censés réaliser la même fonctionnalité. Il est donc intéressant d'utiliser ce regroupement pour visualiser et comprendre le fonctionnement du logiciel [Cas 12]. La structure de données la plus appropriée pour représenter cette arborescence est l'arbre. On commence par placer la racine dans le haut de la vue, puis on place ses enfants directement en-dessous d'elle, et ainsi de suite de façon récursive pour chacun de ses enfants. Ceci donne des couches successives d'éléments, espacées également, où chaque couche représente un niveau de la hiérarchie [Bou 11] (**Figure 4.1**). La vue "Complexité du Système" montre un exemple de visualisation de la structure hiérarchique (arborescence) permettant de montrer l'arbre d'héritage complet du logiciel. Nous la verrons en détail par la suite.

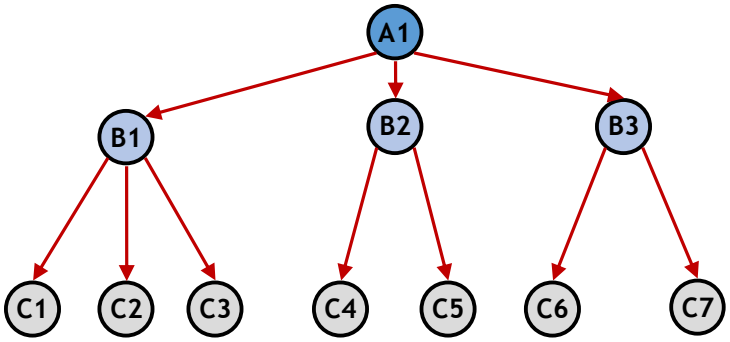


Figure 4. 1 Exemple d'une représentation arborescence.

✪ *Visualisation des dépendances*

La visualisation des diverses liens de dépendances entre les entités logicielles est une tâche plus compliquée que celle de la visualisation de l'arborescence. En effet, ces liens peuvent être de différentes natures telles que les liens d'héritage, les appels de méthodes, etc. Pour représenter toutes les dépendances, la représentation en arbre n'est pas adaptée. Ce sont les représentations de graphe qui permettent de modéliser au mieux ces dépendances. En effet, les graphes ont toutes les caractéristiques nécessaires pour représenter les diverses relations entre les entités, en associant les entités aux nœuds du graphe et les relations aux arêtes [Her 00, Lew 03, Noa 05, Sch 06, Cas 12]. Nous verrons en détail par la suite les vues : "Dépendances des Classes & Aspects" et "Dépendances des Packages" montrant la visualisation des dépendances selon notre approche.

Visualiser toutes les liens de dépendances d'un logiciel de grande taille revient donc à visualiser un très grand graphe avec beaucoup d'arêtes qui se chevaucheraient et qui recouvriraient ces nœuds. Donc, le graphe ne serait pas compréhensible. Une solution permettant d'aider à éviter cette problématique est de représenter le graphe dans un espace en 3D à l'aide d'une métaphore du monde réel (ex. Métaphore de la ville) [Fro 06, Cas 12].

4.2.2 Paramètres importants de visualisation

Dans notre approche, nous avons traité uniquement des métriques "produits" et plus précisément celles qui caractérisent et quantifient un côté précis du code source parmi celles proposées dans la littérature. À cet effet, on a effectué une étude détaillée sur les programmes orientés aspects. Dans la suite de cette section, on va présenter quelques métriques qu'on a trouvées pertinentes et qui sont utilisées couramment dans les outils de mesure de la qualité du code source, et d'autres qu'on a déterminées comme des paramètres indispensables pour produire une visualisation efficace en deux et trois dimensions. La **Table 4.1** suivante montre une liste des métriques sélectionnées.

Acronyme	Nom	Description
N_MD_c	Number of Methods Declared in class	Nombre de méthodes de la classe **c**
N_AD_c	Number of Attributes Declared in class	Nombre d'attributs de la classe **c**
N_MD_a	Number of Methods Declared in aspect	Nombre de méthodes au niveau de l'aspect **a**
N_AD_a	Number of Attributes Declared in aspect	Nombre d'attributs au niveau de l'aspect **a**
N_ITMD_c	Number of Methods Introduced in class	Nombre de méthodes introduites par différents aspects au niveau de la classe **c**
N_ITAD_c	Number of Attributes Introduced in class	Nombre d'attributs introduits par différents aspects au niveau de la classe **c**
$N\_POINTCUT_a$	Number of Pointcuts Declared in aspect	Nombre de points de coupure associés à l'aspect **a**
N_ADVICE_a	Number of Advices Declared in aspect	Nombre de consignes associées à l'aspect **a**
NLOC_CLS	Number of lines of code of the class	Nombre de lignes de code par classe
NLOC_ASP	Number of lines of code of the aspect	Nombre de lignes de code par aspect
NLOC_PKG	Number of lines of code of the package	Nombre de lignes de code par package
NLOC_PRO	Number of lines of code of the project	Nombre de lignes de code du projet
NPKG _PRO	Number of packages of the project	Nombre de packages du projet
NASP _PRO	Number of aspects of the project	Nombre d'aspects du projet
NCLS _PRO	Number of classes of the project	Nombre de classes du projet
NASP_PKG	Number of aspects of the package	Nombre d'aspects par package
NCLS_PKG	Number of classes of the package	Nombre de classes par package
NPJ_PCUT	Number of Join Point by Pointcut	Nombre de points de jointure par point de coupure
NAD_PCUT	Number of Advices associated to Pointcut	Nombre de consignes associés au point de coupure

Table 4. 1 Métriques sélectionnées pour une visualisation efficace.

4.2.3 Métriques considérées en visualisation

Notre intérêt pour les programmes ASPECTJ nous a conduit à se focaliser sur la vue polymétrique pour une visualisation en deux dimensions (2D) d'une part, et d'une métaphore de ville pour une visualisation en trois dimensions (3D) d'autre part. Nous verrons dans ce qui suit comment les métriques statiques sont utilisées sous les diverses vues de visualisation en 2D et 3D respectivement.

4.2.3.1 Visualisation bidimensionnelle (2D)

Pour une représentation graphique en 2D, l'utilisation des métriques offre la possibilité de modéliser des entités logicielles (telles que les classes, les interfaces, les aspects et les packages) sous une forme plus abstraite comme par exemple des rectangles. Les liens de dépendances telles que la hiérarchie d'héritage entre ces entités sont modélisées sous forme d'arêtes. Ce choix, même s'il peut paraître simple, représente un bon compromis entre la simplicité et la quantité d'informations pouvant être exprimées. Pour un code JAVA, on a sélectionné d'une part les différentes métriques utilisées sous l'outil X-Ray [W13], et de l'autre part on a adapté ces métriques pour le cas d'un code ASPECTJ en leur adjoignant d'autres métriques spécifiques à l'orienté aspects.

✪ *Pour un code JAVA*

La vue "Complexité du Système" montre une visualisation en 2D de la hiérarchie d'héritage. C'est une vue d'ensemble de l'arbre d'héritage du système logiciel en entier. Cette vue peut donner des indications sur la complexité et la structure du système (le nombre d'entités logicielles, etc.), ainsi que des informations sur l'utilisation de l'héritage dans le système (voir les profondeurs des hiérarchies et en général si le système est plat ou profond). De plus, elle permet de détecter facilement les anomalies de conception telle que des classes très grandes, très petites ou vides.

La **Table 4.2** présente les métriques appliquées dans la vue "Complexité du Système" pour un code JAVA.

Métrique	Description
Position	Calculée suivant la disposition de l'arbre (orientée top-down). (les classes dans les plus hautes positions sont des parents pour les classes de positions inférieures).
Couleur	Type de classe : vert (classe externe au projet), blanc (interface), bleu (classe concrète), bleu clair (classe abstraite)
Largeur	Nombre de méthodes
Hauteur	Nombre de lignes de code (LOC, Ligne Of Code)
Frontière	Frontière noir pour une classe autonome ou orange pour une classe interne.
Arêtes	Héritage entre classes (les dépendances de la hiérarchie d'héritage)

Table 4. 2 Description des métriques appliquées dans la vue « Complexité du Système » pour un code Java.

La **Figure 4.2** montre graphiquement les métriques utilisées dans la vue "Complexité du Système".

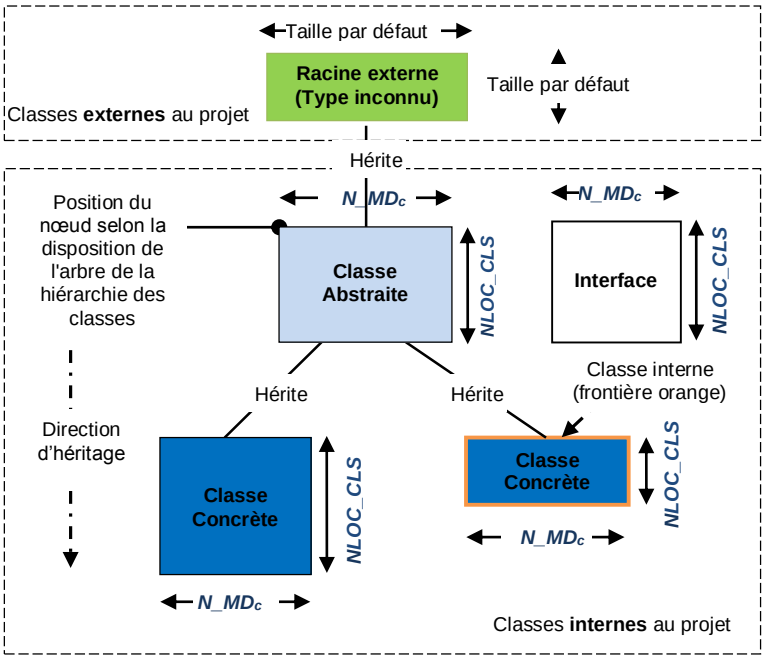


Figure 4. 2 Métriques appliquées dans la vue « Complexité du Système » pour un code Java.

La **Table 4.3** présente les métriques appliquées sous les deux vues : "Dépendances des Classes" et "Dépendances des Packages" pour un code JAVA.

Métrique	Description
Couleur	Packages: marron, Classes: blanc (interface), bleu (classe concrète), bleu clair (classe abstraite).
Arêtes	Liens de dépendances entre entités logicielles avec deux métriques : couleur et épaisseur différentes pour distinguer les poids des dépendances comme le montre la **Table 4.6**.

Table 4. 3 Description des métriques appliquées dans les vues « Dépendances des Classes & Packages ».

La **Figure 4.2** montre graphiquement les métriques utilisées sous les deux vues de dépendances (*Circle Graph*). Cette illustration montre une vue de dépendances des entités logicielles de type Classe. Elles ont la même taille et leurs couleurs reflètent leurs types (une classe abstraite ou concrète).

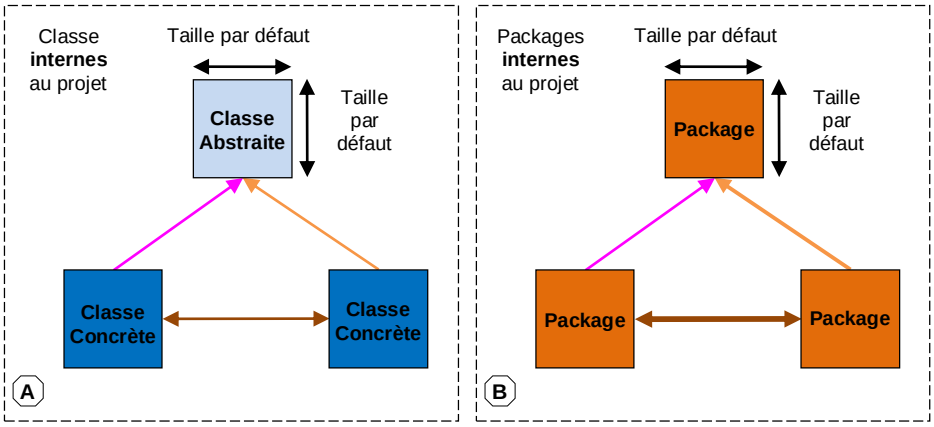


Figure 4. 3 Métriques appliquées dans les deux vues "Dépendances des Classes" et "Dépendances des Packages".

✪ *Pour un code Aspect*

La **Table 4.4** présente les métriques appliquées dans la vue "Complexité du Système" pour un code Aspect.

Métrique	Description
Position	Calculée suivant la disposition d'arbre (orienté top-down). Les classes ou les aspects dans les plus hautes positions sont des parents pour les classes ou les aspects de positions inférieures.
Couleur	Type d'aspect : orange (aspect externe au projet), rose (aspect concret), violet (aspect abstrait).
Largeur	Nombre d'attributs et méthodes.
Hauteur	Nombre de points de coupure et des consignes.
Frontière	Frontière noire pour un aspect autonome ou bleu pour un aspect interne.
Arêtes	Héritage entre classes et aspects (les dépendances de la hiérarchie d'héritage : dépendance aspect-aspect, dépendance aspect-classes).

Table 4. 4 Description des métriques appliquées dans la vue "Complexité du Système" pour un code Aspect.

La **Figure 4.4** montre graphiquement les métriques utilisées dans la vue "Complexité du Système".

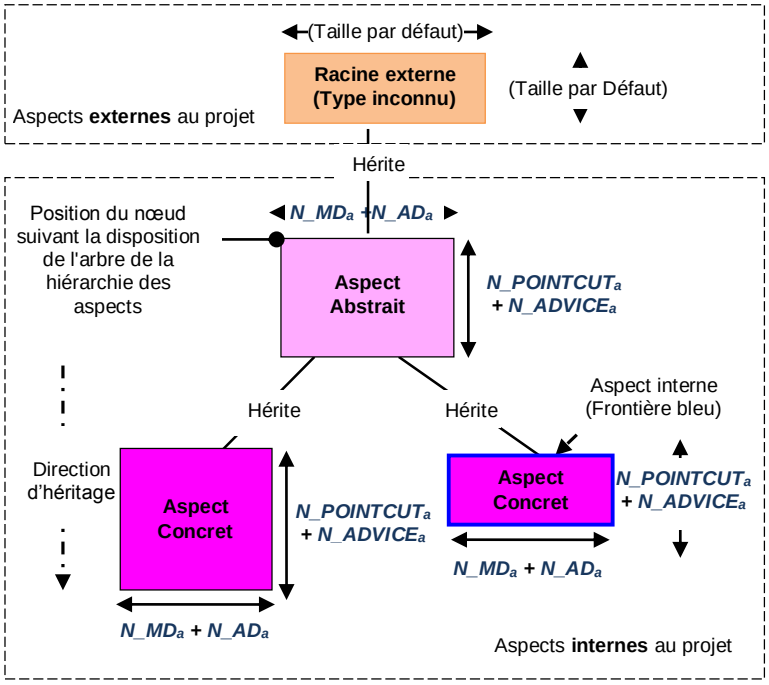


Figure 4. 4 Métriques appliquées dans la vue "Complexité du système" pour un code Aspect.

La **Table 4.5** présente les métriques appliquées sous les deux vues "Dépendances des Aspects" et "Dépendances des Packages" pour un code Aspect.

Métrique	Description
Couleur	Packages : marron, Aspects : rose (aspect concret), violet (aspect abstrait).
Arêtes	Liens de dépendances entre entités logicielles avec deux métriques : couleur et épaisseur différentes pour distinguer les poids des dépendances comme le montre la **Table 4.6**.

Table 4. 5 Description des métriques appliquées dans les vues "Dépendances des Aspects et des Packages".

La **Figure 4.5** montre graphiquement les métriques utilisées sous les deux vues de dépendances.

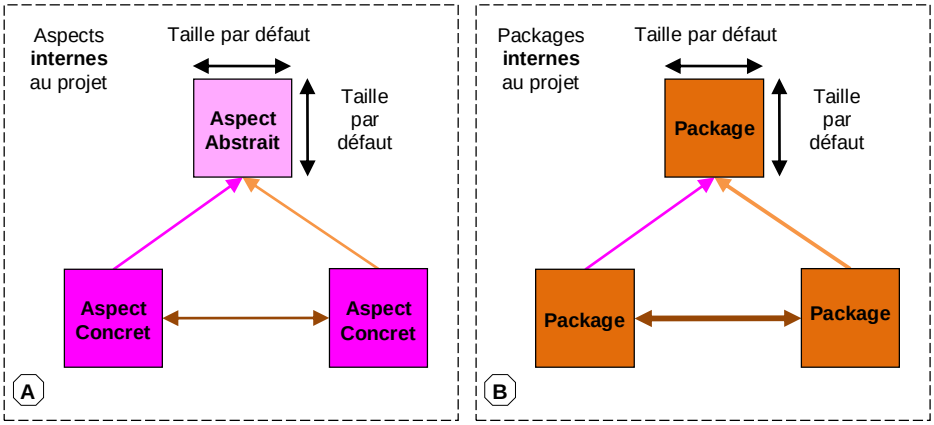


Figure 4. 5 Métriques appliquées dans les vues (A): "Dépendances des Aspects" et (B) "Dépendances des Packages".

Les liens de dépendances sont visualisés par des arêtes avec deux métriques "couleur" et "épaisseur" pour quantifier leurs poids. Selon l'association métrique-représentation, pour de nombreuses entités logicielles et avec des formes différentes, les vues produites peuvent créer une vraie problématique d'encombrement visuel. Les recherches de *Gestalt* ont montré que l'être humain n'est capable de reconnaitre et distinguer de manière efficace que jusqu'à six tailles différentes d'un même objet. Comme le cas de l'outil X-Ray [W13], on a donc décidé de limiter le nombre de représentations avec ces deux métriques pour afficher les liens à six niveaux afin de réduire grandement la charge cognitive et la complexité visuelle, et donc rendre la visualisation plus facile pour naviguer.

La **Table 4.6** montre les six niveaux déterminés pour différencier les différents poids de dépendances.

Niveau	Nb d'appels externes	Couleur	Épaisseur
1	1-2	Violet	Par défaut, taille mince (2 pixels)
2	3-4	Rose	Augmenté par un facteur de 2 (4 pixels)
3	5-9	Orange	Augmenté par un facteur de 3 (6 pixels)
4	10-19	Marron Clair	Augmenté par un facteur de 4 (8 pixels)
5	20-49	Marron	Augmenté par un facteur de 5 (10 pixels)
6	>50	Noir	Augmenté par un facteur de 6 (12 pixels)

Table 4. 6 Métriques d'arêtes dans les deux vues "Dépendances des Classes" et "Dépendances des Packages".

4.2.3.2 Visualisation tridimensionnelle (3D)

Ces dernières années, une tendance de recherche se dégage pour les techniques de visualisation tridimensionnelle. Ce genre de visualisation pourrait mener à une meilleure utilisation de l'espace vu que nous pouvons afficher plus d'entités en exploitant la troisième dimension. La visualisation en 3D vise à condenser les diverses vues, à représenter plus de relations, ainsi qu'à avoir la possibilité de faire de meilleures interactions. De plus, elle réduit grandement la complexité visuelle et rend les vues plus réalistes et familières et donc plus facile pour naviguer [Lew 03, Gre 05, Fro 06].

Notre idée est de faire une visualisation en 3D de la structure hiérarchique, basée sur la métaphore de la ville avec une combinaison de quelques techniques d'interaction et de métriques logicielles. Cette métaphore est comme celle proposée par *Richard Wettel* et *Michael Lanza* et est très semblable à celle utilisée sous l'outil CodeCity [W15] et sous le système VERSO, développé par *Langelier* et *al.* [Lan 05b, Lan 06].

Dans la métaphore de la ville, la représentation d'un système logiciel ressemble beaucoup à une vue aérienne d'une ville. La structure de la ville du point de vue complexité et interactions entre les constituants est à notre avis équivalente à celle d'un système logiciel de taille similaire. Mais contrairement au système logiciel, les concepts tels que le type de construction (ou édifice), la nature des quartiers (ou districts) et le type des villes sont très familiers et ont un sens consensuels pour les observateurs. En effet, un coup d'œil sur une vue aérienne d'une ville ou d'un quartier peut évoquer une foule d'informations dont l'équivalent en logiciel nécessiterait un long effort d'analyse.

Cependant, pour profiter de cet avantage, il faut que l'équivalence entre les concepts du logiciel et ceux de la ville soit la plus directe et la moins ambiguë possible. La facilité et la rapidité d'interprétation des informations dérivées par observation dépendent grandement de la nature de cette équivalence (c.-à-d. les équivalences entre les constructions et les entités logicielles (ex. classes, interfaces et aspects), les districts et les packages, et enfin les villes et systèmes logiciels) et la structure de la hiérarchie est affichée à l'aide d'une représentation par inclusion.

Dans notre approche de visualisation tridimensionnelle, nous avons tenu compte du tissage (ou Entrecoupage) d'aspects (CS, Crosscutting Statique) pour distinguer deux états de visualisation : avant le tissage et après le tissage. De plus, afin de quantifier le changement qui est prévu au niveau de la partie du code JAVA après ce tissage, nous avons déterminé d'autres métriques appropriées.

La **Figure 4.6** présente en détail les métriques sélectionnées, exprimées sous deux vues, avant et après l'opération de tissage.

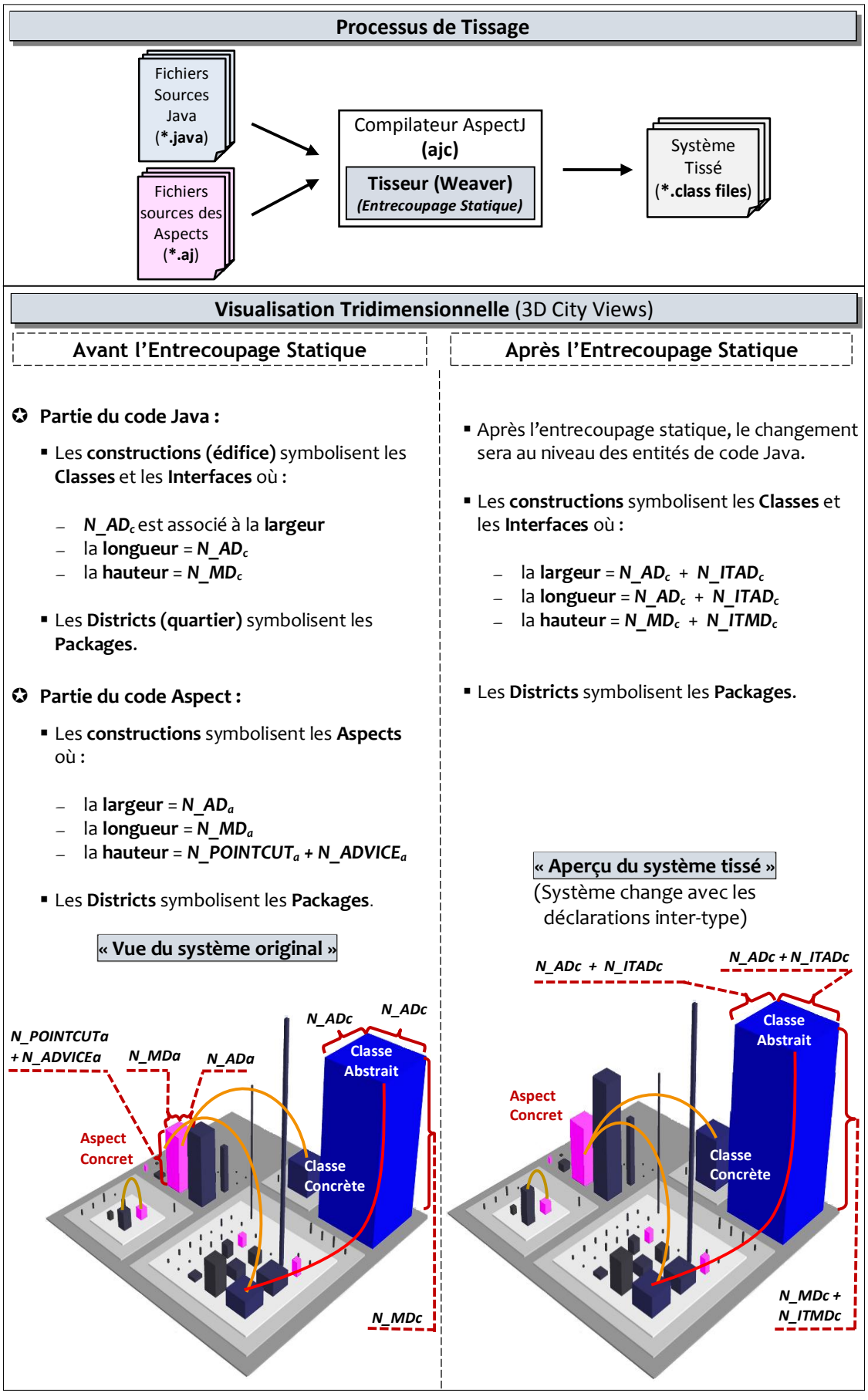


Figure 4. 6 Visualisation tridimensionnelle (3D) avant et après le tissage statique.

Avant le tissage d'aspects, pour le code JAVA, le nombre de méthodes (*N_MD*c) par classe est représenté par la taille (hauteur) de la construction et le nombre d'attributs (*N_AD*c) par classe est symbolisé par la largeur de la construction. Pour le code ASPECTJ, le nombre de points de coupure et de consignes associées (*N_POINTCUT*a + *N_ADVICE*a) par aspect est représenté par la hauteur de la construction et le nombre d'attributs (*N_AD*a) (respectivement le nombre de méthodes (*N_MD*a)) par aspect est symbolisé par la largeur (respectivement la longueur) de la construction. Cela permet donc de montrer la quantité de fonctionnalités pour chaque entité logicielle en fonction du volume de la construction correspondante.

Avec ces deux vues de visualisation, l'utilisateur sera capable d'apercevoir, localiser et de quantifier approximativement les changements qui auront lieu après le tissage en fonction des variations en volumes des constructions.

4.3 Mise en œuvre de l'approche

Dans cette section, nous donnons une description détaillée de la mise en œuvre de notre approche décrite précédemment, selon le volet conceptuel et le volet implémentation. Pour ce faire, nous présenterons les outils supports implémentés; en premier lieu, l'outil "VizzAspectJ-2D" permettant une visualisation bidimensionnelle (2D) et l'outil "VizzAspectJ-3D" pour une visualisation tridimensionnelle (3D) dans un second lieu.

Au lieu de créer notre système de visualisation de toutes pièces, nous avons plutôt utilisé, sous la plateforme ECLIPSE, l'outil X-Ray [W13] et Citylyzer [W14] comme point de départ respectivement pour les deux types de visualisation en 2D et en 3D. Chaque outil offre déjà un environnement interactif bien construit. Nous profitons donc des principales fonctionnalités qui existaient déjà, non seulement au niveau des méthodes de navigation, mais aussi au niveau de la représentation des propriétés des entités logicielles. De plus, nous avons ajouté d'autres fonctionnalités, ainsi que modifié plusieurs aspects qui étaient déjà en place afin d'offrir des vues qui s'adaptent avec un code orienté-aspects.

4.3.1 VizzAspectJ-2D

4.3.1.1 Conception

La **Figure 4.7** montre une vue simplifiée du diagramme de classes UML modélisant le noyau de l'outil "VizzAspectJ-2D". Afin de simplifier la compréhension de la modélisation, on ne visualise ni les méthodes ni les attributs.

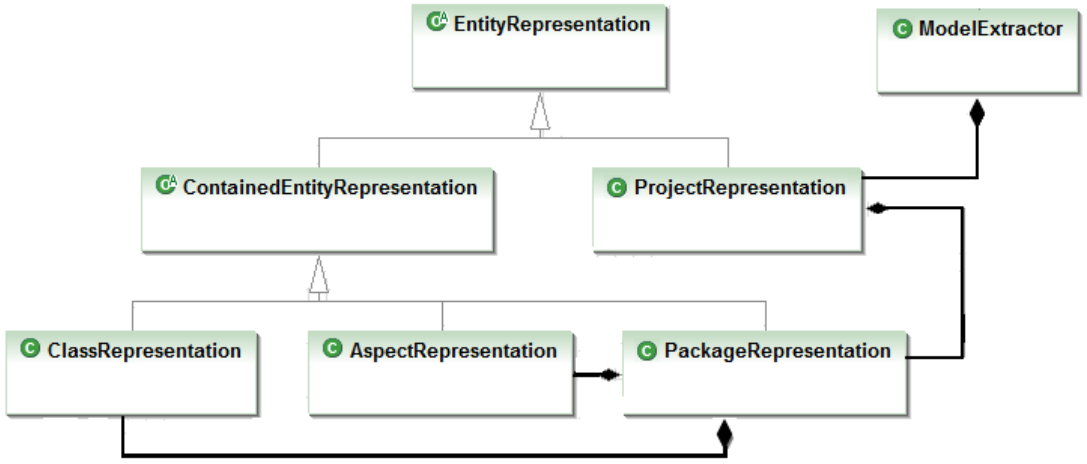


Figure 4. 7 Vue simplifiée de la représentation interne du code sous "VizzAspectJ-2D".

L'arbre d'héritage montré via cette vue peut être aussi exposé dans la vue "Complexité du Système" produite par cet outil, en visualisant le code source de l'outil lui-même. La partie la plus importante est son noyau qui est responsable de la construction d'une représentation interne du code (ICR, *Internal Code Representation*) comme une étape initiale dans le processus de production des vues.

A. Représentation interne du code

À la première étape du processus de production, la création d'une représentation interne du code (ICR) reflétant le code source du projet à visualiser est assurée directement par le noyau.

La **Figure 4.8** montre une modélisation de cette représentation ICR avec plus de détails où chaque entité logicielle (package, classe, interface et aspect) est modélisée par la classe *« EntityRepresentation »*.

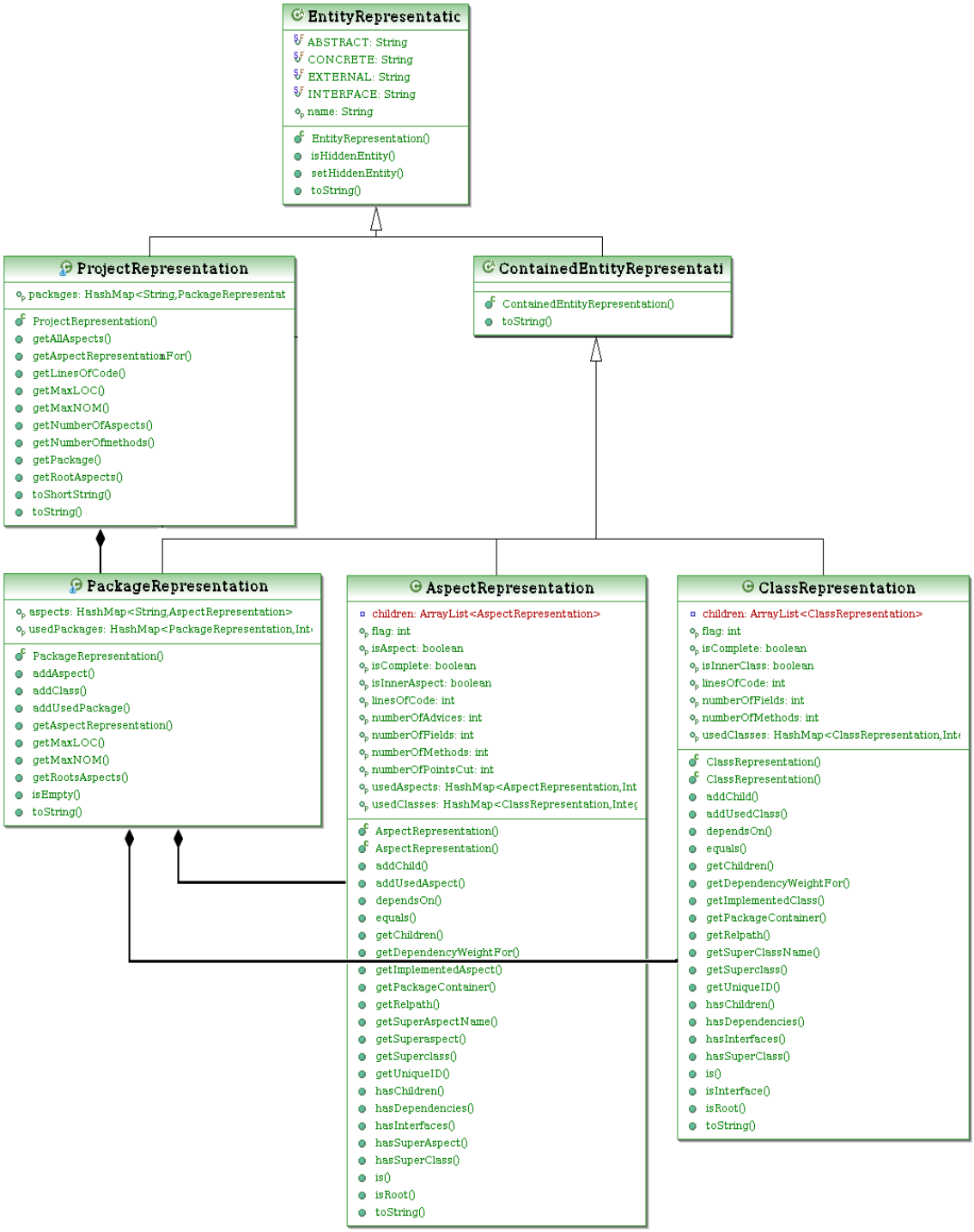


Figure 4. 8 Diagramme de classes de la représentation interne du code sous "VizzAspectJ-2D".

B. Chaîne de production des vues

La **Figure 4.9** montre la chaine de construction des diverses vues polymétriques de visualisation en deux dimensions au sein de l'outil "VizzAspectJ-2D".

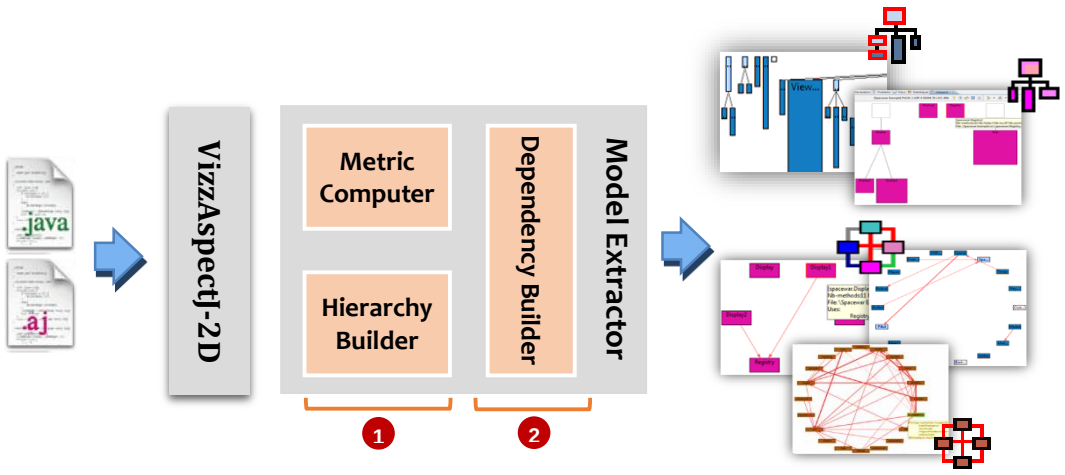


Figure 4. 9 Chaîne de production des vues polymétriques sous "VizzAspectJ-2D".

C. Extracteur du modèle « Model Extractor »

C'est le responsable de la création d'un modèle du code source du projet à visualiser après une étape d'analyse qui s'exécute immédiatement par le noyau suite à l'initialisation. La classe « *ModelExtractor* » modélise cet extracteur.

Le processus d'extraction est décomposé en deux phases : la première est assurée par deux composants : Constructeur de la Hiérarchie « *Hierarchy Builder* » et Evaluateur des Métriques « *Metric Computer* », et la seconde est assurée par le Constructeur des Dépendances « *Dependency Builder* » (Voir la **Figure 4.10**).

✪ *Constructeur de la Hiérarchie*

Ce constructeur analyse le projet et réunit des informations concernant la hiérarchie d'héritage de chaque entité logicielle (classe, interface et aspect). Une fois cette tâche exécutée, le projet à visualiser, précédemment vu comme un ensemble de répertoires et de fichiers sources, prend une autre forme.

✪ *Evaluateur des Métriques*

Cet évaluateur est responsable de recueillir des informations concernant les métriques requises pour la création de la vue "Complexité du Système". Il détermine le nombre de méthodes et d'attributs, le nombre de lignes de code, etc. Dès que cette seconde tâche est achevée, l'extracteur du modèle aura rempli la représentation interne du code par toute l'information requise pour finaliser et générer la représentation graphique de la vue "Complexité du Système".

D. Constructeur des dépendances « Dependency Builder »

Ce constructeur explore le code source et réunit des informations concernant les relations de dépendances existantes entre les diverses entités logicielles (packages, classes, interface et aspects). Ces informations sont requises pour la représentation graphique sous forme d'arêtes de ces dépendances sous les trois vues de dépendances.

D'un point de vue technique, cette tâche est souvent lourde (en temps et espace mémoire) dès que le système logiciel à visualiser est de grande taille. De ce fait, une extension de ce constructeur (*Incremental Dependency Builder*) a été ajoutée permettant de contrôler l'ensemble d'entités logicielles à visualiser (c.-à-d. exécution sur demande) dont le but d'améliorer les performances d'exécution et même faciliter l'utilisation de l'outil "VizzAspectJ-2D".

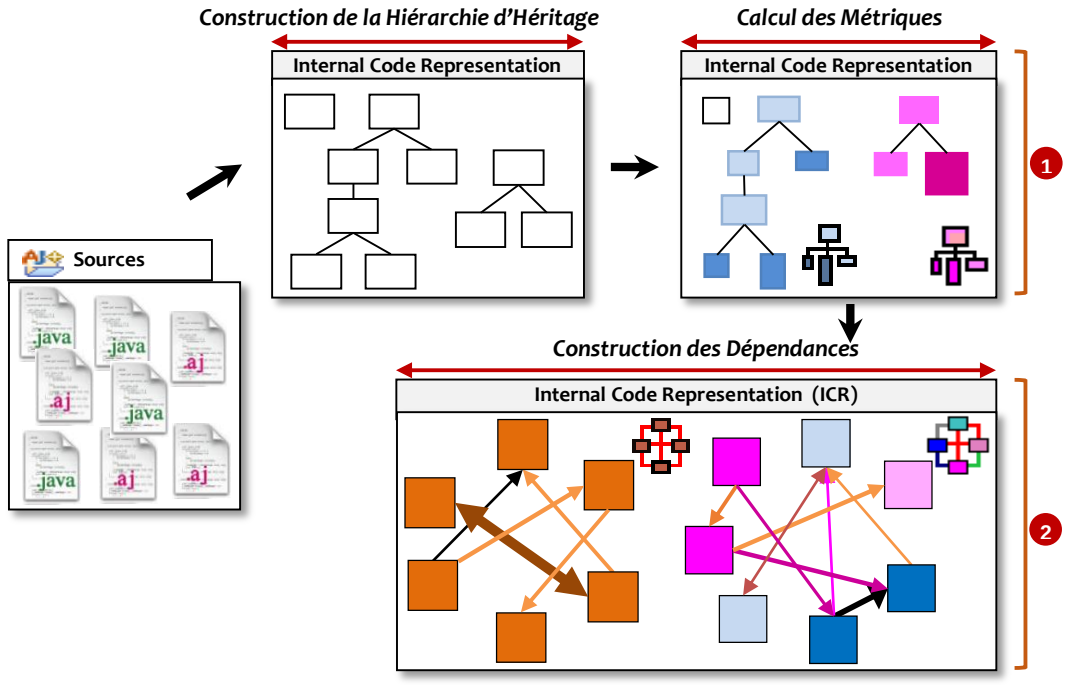


Figure 4. 10 Cycle de construction des vues sous "VizzAspectJ-2D".

4.3.1.2 Implémentation

✪ *Choix du support de développement*

Presque tous les outils d'analyse et visualisation sont des applications autonomes (en anglais : standalone applications), comme par exemple l'outil CodeCity [W15]. Cela force les utilisateurs à basculer entre ces outils et leurs éditeurs préférés de codage (entre différents fenêtres et contextes). Ce changement de contexte pose un problème durant l'interaction en termes de temps et d'effort et même force les utilisateurs à utiliser ces outils en dehors de

leurs éditeurs préférés. De ce fait, le développement des outils d'analyse et de visualisation open-source sous la forme de plug-ins ECLIPSE représente un pas considérable en vue de rapprocher ce genre d'outils d'assistance indispensables au processus de maintenance où les utilisateurs perçoivent et analysent les diverses vues pendant l'écriture du code source. Notre choix de l'environnement de développement ECLIPSE a été guidé par le souci de réunir divers outils de manières naturelles.

✪ *Interface Utilisateur*

La **Figure 4.11** présente une capture d'écran de l'outil "VizzAspectJ-2D" sous la plateforme ECLIPSE. Le menu de l'interface est constitué de deux parties : la première donne une brève description textuelle sur les entités logicielles du projet analysé en termes de quelques métriques (nombre de lignes de code, packages, classes, aspects, etc.). La seconde partie représente une palette de petites icônes représentant les diverses actions qu'on peut entreprendre.

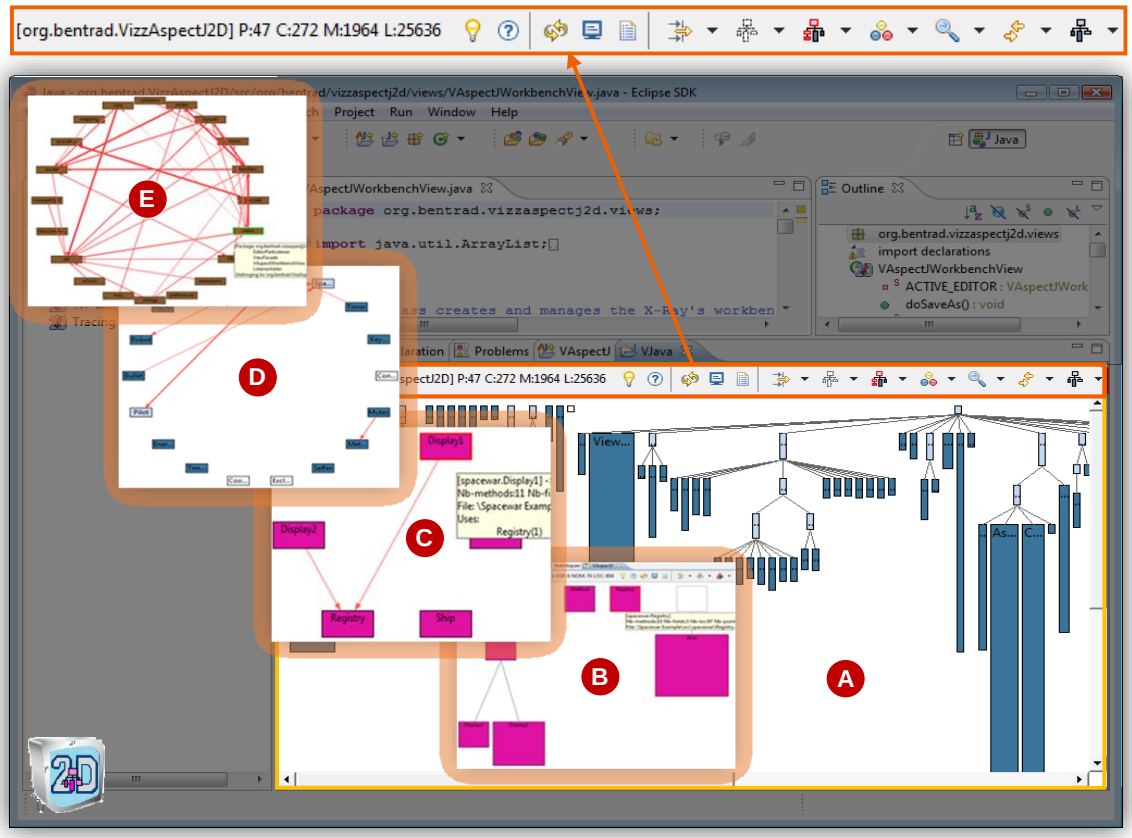


Figure 4. 11 Une vue de l'outil "VizzAspectJ-2D" sous la plateforme ECLIPSE.

✪ *Visualisation par les vues*

Dans un système logiciel, les mêmes entités logicielles peuvent être représentées de nombreuses façons. Selon le point de vue sous lequel l'utilisateur souhaite voir le système, les différentes abstractions avec les métriques appliquées devront être utilisées afin de diminuer la complexité des entités exposées et de faciliter les tâches d'analyse et de navigation. Le système de visualisation offert via l'outil "VizzAspectJ-2D" fournit deux catégories de vues polymétriques déjà mentionnées précédemment :

- La vue "**Complexité du Système**". C'est une vue polymétrique qui expose une visualisation de l'arborescence des entités logicielles. Une capture d'écran de cette vue est montrée dans la **Figure 4.11** : (**A**) et (**B**).

- Les **Vues de dépendances**. Pour les diverses entités logicielles (Packages, Classes et Aspects), le système de visualisation propose respectivement les vues suivantes : "Dépendances des Packages" et "Dépendances des Classes & Aspects". Voir les captures d'écran correspondantes (**C**), (**D**) et (**E**) dans la **Figure 4.11**.

4.3.2 VizzAspectJ-3D

4.3.2.1 Conception

La **Figure 4.12** suivante montre un diagramme de classe UML simplifié modélisant la métaphore de la ville sous l'outil "VizzAspectJ-3D". D'autres modélisations techniques ne sont pas décrites ici telle que la méthode de navigation sur la ville moyennant une caméra.

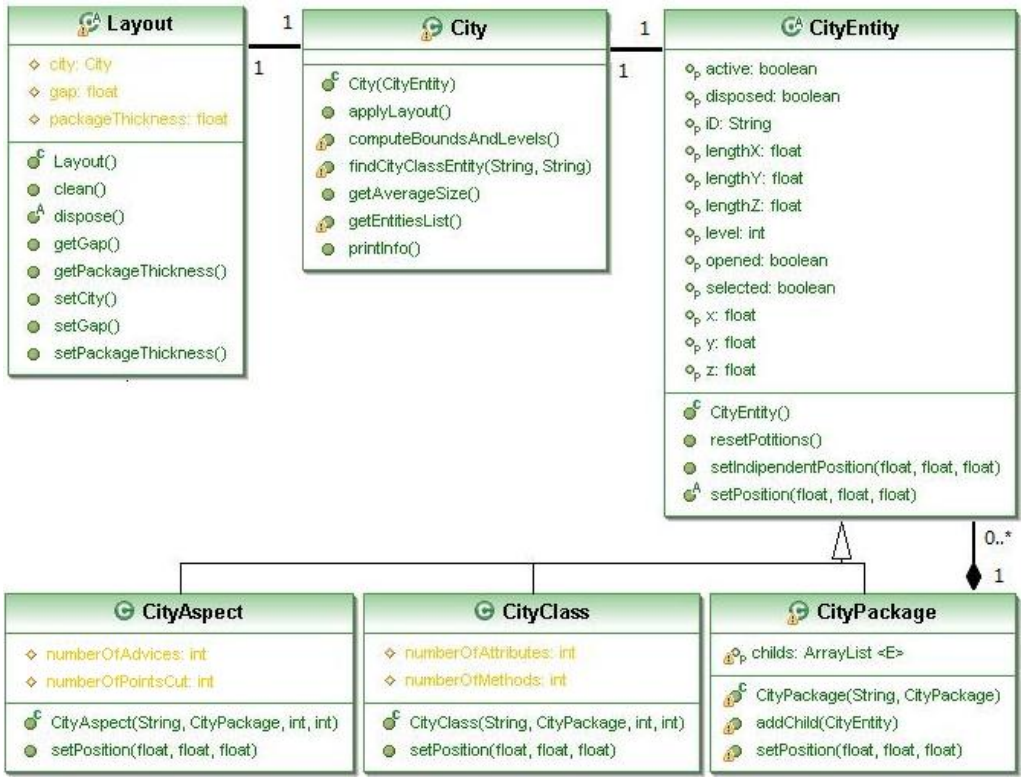


Figure 4. 12 Diagramme de classes UML simplifié du modèle de ville sous "VizzAspectJ-3D".

✪ *Chaîne de production des vues*

La **Figure 4.13** montre la chaine de construction d'une vue de ville en trois dimensions au sein de l'outil "VizzAspectJ-3D".

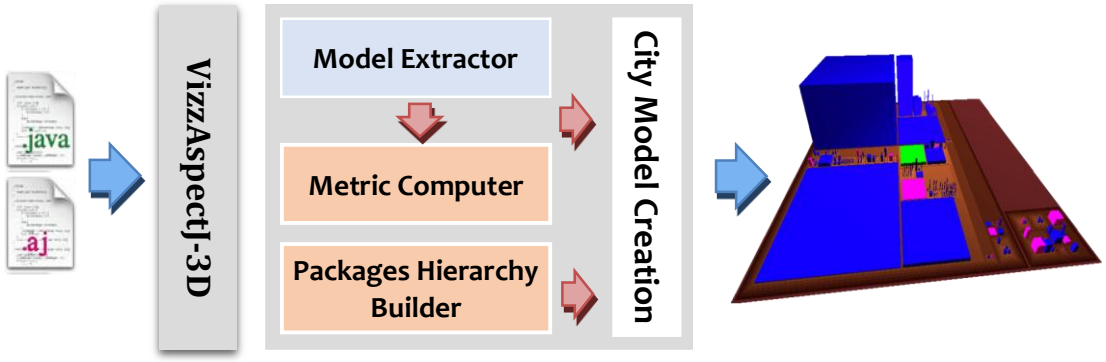


Figure 4. 13 Chaîne de production d'une vue de ville sous "VizzAspectJ-3D".

✪ *Création de la ville*

Sous l'outil "VizzAspectJ-3D", la construction d'une vue de ville requière certaines interactions (ou dépendances de communications) entre son noyau et celui de l'outil "VizzAspectJ-2D" d'une part, et avec la bibliothèque OpenGL [W24] et la plateforme ECLIPSE d'autre part. Le fonctionnent de ce processus est décrit brièvement via la **Figure 4.14**.

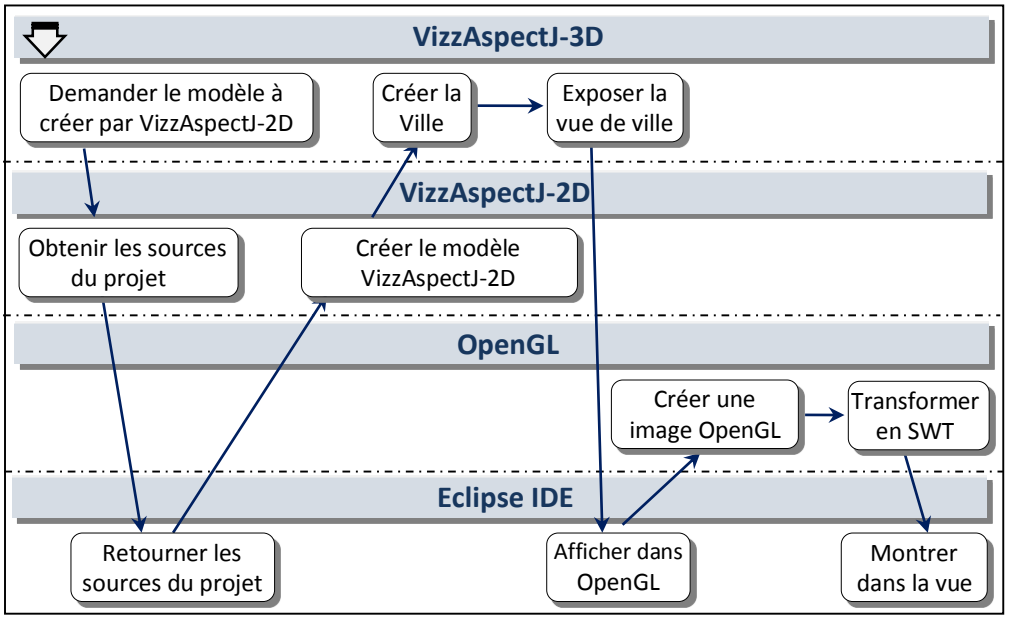


Figure 4. 14 Interactions entre "VizzAspectJ-3D" et "VizzAspectJ-2D" sous la plateforme ECLIPSE.

Au début, à partir du modèle fourni via "VizzAspectJ-2D", nous pouvons obtenir une liste de tous les packages existants. Leur organisation hiérarchique (c.-à-d. la hiérarchie des packages) est faite selon l'algorithme de *Andrea Biaggi* qui a implémenté dans son outil de visualisation Citylyzer [W14].

Une vue de déroulement de cet algorithme sur les trois packages (org.vizzaspectj3d.model.class; org.vizzaspectj3d.view ; org.vizzaspectj3d.model.package) est aussi exposée dans la **Figure 4.15** (**(A)** Algorithme de construction de la hiérarchie des packages et **(B)** Vue de déroulement de l'algorithme).

La représentation graphique de cette hiérarchie est faite par une transformation où chaque niveau correspond à un district. Les entités (classes, interface et aspects) sont présentées sur le niveau de leurs parents, et les paquetages sont de tailles décroissantes de l'extérieur vers l'intérieur.

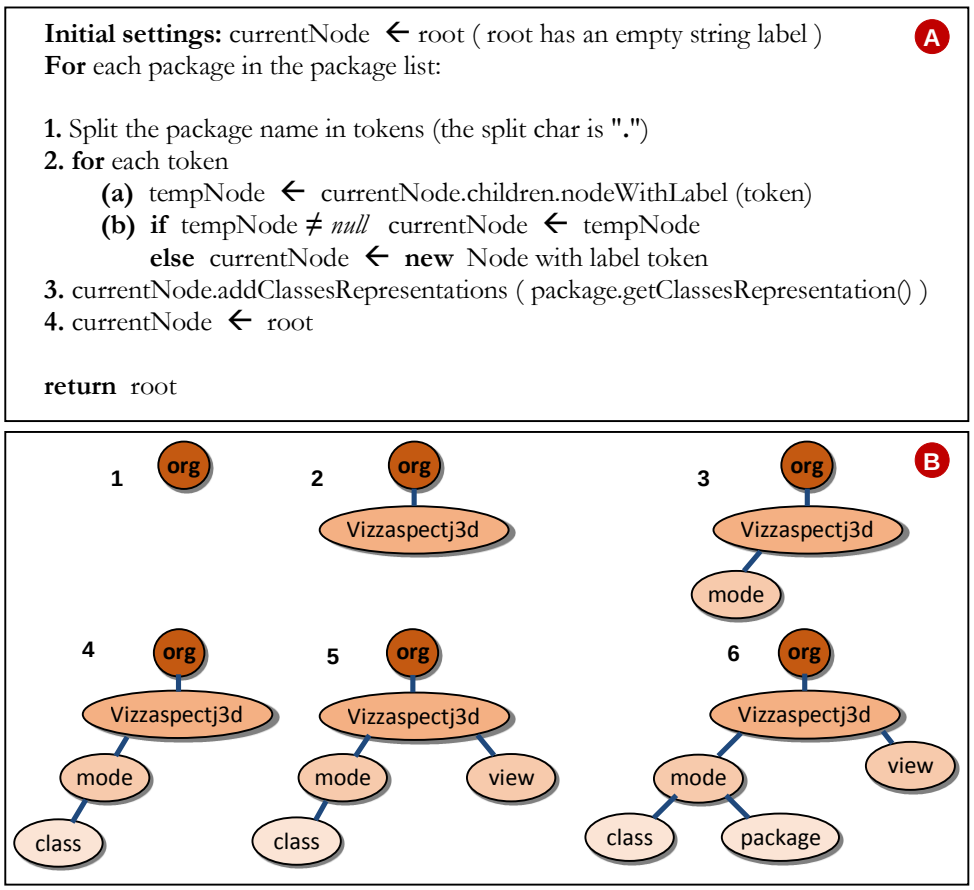


Figure 4. 15 Algorithme de construction de la hiérarchie des packages.

4.3.2.2 Implémentation

La **Figure 4.16** présente une capture d'écran de l'outil "VizzAspectJ-3D" sous la plateforme ECLIPSE. L'interface offre une palette constituant un ensemble actions permettant des diverses façons d'interaction telles que :

- La navigabilité dans un espace 3D au sein de la ville qui est assurée via une caméra permettant de survoler d'une manière souple la ville, de zoomer, de sélectionner des constructions et d'interagir avec les fichiers sources des entités logicielles correspondantes.
- La spécification de quelques paramètres sur la vue (ex. l'espace entre les constructions, l'épaisseur des districts, etc.).
- De plus, la possibilité de naviguer dans la ville à travers des combinaisons de touches du clavier.

Le système de navigation apporte à l'utilisateur une plus grande liberté dans l'exploration. Il permet de s'approcher d'un point (zoom in), de s'en éloigner (zoom out) et de voir de petites entités logicielles qui peuvent être cachées par les plus grandes, il suffit de changer l'angle de vue de la caméra. Il est également possible d'obtenir plus de détails sur une entité en cliquant sur sa représentation graphique. Jusqu'à présent, en quelques secondes, l'outil support "VizzAspectJ-3D" a réussi à visualiser et à analyser efficacement les éléments clés de systèmes logiciels de près de 10 000 entités.

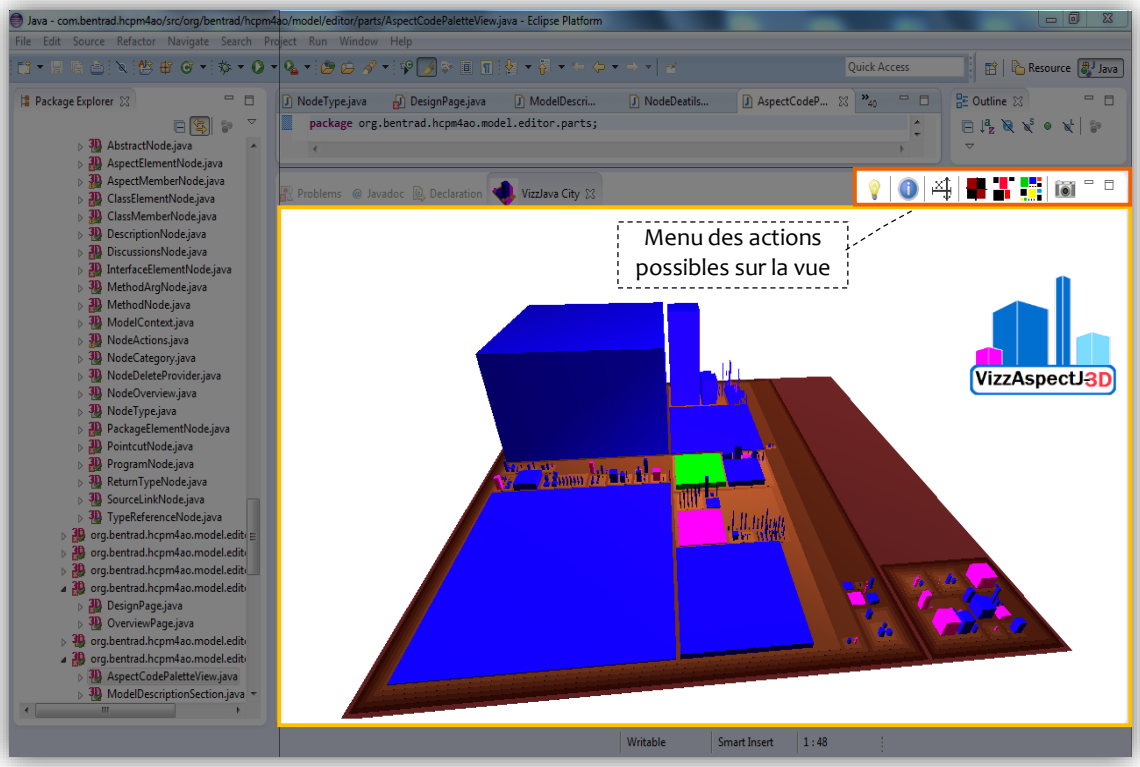


Figure 4. 16 Une capture d'écran de l'outil "VizzAspectJ-3D" sous la plateforme ECLIPSE.

4.4 Conclusion

Dans ce chapitre, nous avons présenté en détail notre première proposition concernant une approche d'analyse et de visualisation bidimensionnelle (2D) et tridimensionnelle (3D) permettant de représenter efficacement des propriétés quantitatives des grands logiciels orientés-aspects et plus particulièrement le volet statique des programmes ASPECTJ dans le but d'effectuer des analyses qualitatives à travers l'utilisation de métriques.

Nous avons concrétisé cette proposition par la mise en œuvre d'un système de visualisation qui se base sur deux outils prototypes pour la plateforme ECLIPSE : "VizzAspectJ-2D" et "VizzAspectJ-3D" permettant une visualisation en 2D et 3D en utilisant

respectivement des vues polymétriques et la métaphore de la ville. Les vues de visualisation produites servent à visualiser la structure hiérarchique et les liens de dépendances. Nos critères dans l'élaboration de ces vues étaient que la façon de l'affichage ne génère pas trop d'encombrement visuel, et qu'elles offrent à l'utilisateur diverses façons d'interagir afin de filtrer les informations à visualiser selon le besoin.

Comme mentionné précédemment, l'utilisation des vues polymétriques et la métaphore de la ville n'est pas une chose nouvelle. Cependant à notre connaissance, aucun travail existant n'a visé simultanément la visualisation et l'analyse qualitative d'un code orienté-aspects de grande taille contrairement aux codes orientés-objets.

CHAPITRE 5

VERS UNE MÉTHODOLOGIE DE CODAGE HYBRIDE

SOMMAIRE

5.1 INTRODUCTION 68

5.2 VUE DETAILLE DE L'APPROCHE 70

5.2.1. VERS UNE APPROCHE HYBRIDE DE CODAGE 70
5.2.2. APPROCHE PROPOSEE 73

5.3 MISE EN ŒUVRE DE L'APPROCHE 77

5.3.1 ARCHITECTURE 78
5.3.2 IMPLEMENTATION 81

5.4 CONCLUSION 88

Nous abordons là notre deuxième axe de recherche qui est la programmation visuelle. Ce chapitre présente en détail une contribution concernant l'introduction de l'approche visuelle durant la phase d'implémentation. Notre objectif est de proposer une nouvelle méthodologie de codage hybride en intégrant les deux styles de codage : textuel et visuel, pour le paradigme "orienté-aspects" qui facilite le codage en diminuant l'effort et le temps requis pour l'expression des diverses préoccupations. Cette méthodologie est mise en pratique à travers un prototype de l'outil support sous la plateforme ECLIPSE.

5.1 Introduction

Dans le cycle de vie de développement, la méthode adoptée lors de la phase d'implémentation (appelée "programmation" ou "codage") repose souvent sur l'écriture de code source à la main selon une syntaxe spécifique relative au langage de programmation [W21]. Les programmeurs utilisent des outils supports qui dépendent fortement du style basé texte et sont souvent confrontés à des obstacles pour faire évoluer leurs codes. Au cours du processus de compréhension, ils doivent exécuter tout un ensemble de tâches (lecture, recherche, réflexion, traduction, rappel et modélisation mentale), ce qui rend la mise au point sur des problèmes spécifiques très difficile [Sto 05].

Les langages de programmation sont les moyens indispensables pour soutenir les pratiques du génie logiciel. Il est important qu'il soient bien conçues et mis en œuvre au sein d'outils supports solides. Ces derniers doivent offrir des éditeurs de code avec un haut degré de flexibilité et d'efficacité simultanément, afin de rendre le processus de programmation plus efficace. Un effort important a été dédié à relever ce défi. Des éditeurs modernes de codage viennent avec des fonctionnalités utiles comme la coloration syntaxique, la vérification et l'autocomplétion de code, etc. afin de rendre le style traditionnel moins décourageant et ennuyeux, particulièrement pour les débutants. Cependant, généralement la plupart des abstractions qui sont significatives en phase de conception peuvent être perdues lors de la phase de codage [Cai 99]. En outre, de nombreux travaux cherchent à apporter d'autres améliorations au processus de codage en explorant la capacité de l'interface graphique (GUI) à travers d'autres techniques comme les "templates", "générateurs de code", et les "assistants" dans presque tous les environnements modernes de développement intégré (IDEs, *Integrated Development Environments*) [Bur 01, Pla 13].

Cependant, face à ces atouts, la tâche de codage reste fastidieuse et les programmeurs doivent encore consacrer leurs efforts et temps en se focalisant sur les détails d'implémentation. Elle peut conduire à un stress répétitif fréquemment dû aux formalismes syntaxiques, ce qui affecte négativement la capacité du programmeur et sa créativité.

Des travaux de recherche visent à fournir un niveau supérieur d'abstraction en exploitant des techniques graphiques différentes avec plus d'interactivité et souplesse au cours du processus de développement pour assurer l'élaboration, l'exécution et la visualisation des programmes [Fer 02, Bos 04, Zha 07]. L'abstraction représente l'un des principaux avantages des langages de programmation de haut niveau. La tendance actuelle est d'élever le niveau d'abstraction à travers des améliorations apportées aux éditeurs de code afin d'éviter certains écueils de l'approche conventionnelle à base de texte. Ceci permet d'accélérer le codage en minimisant les efforts et par conséquent, d'augmenter la productivité [Yen 96, Zha 08].

Malgré l'évolution des paradigmes de programmation, le style traditionnel est toujours décourageant et fastidieux. Les programmeurs souffrent souvent de certaines limites techniques dues à la méthode de saisie adoptée qui requiert très souvent une bonne maîtrise des formalismes syntaxiques. Cette dernière est considéré comme l'un des facteurs

susceptibles de réduire la productivité et la qualité de code et même l'adoption du paradigme. *Gail C. Murphy* a défini le coût d'implémentation et la complexité comme deux principaux facteurs à prendre en considération lors de l'évaluation d'une nouvelle méthodologie [Mur 99].

En 2001, le MIT (*Massachusetts Institute of Technology*) a annoncé l'approche aspects comme une technologie clé dans la prochaine décennie. Aujourd'hui, quatorze ans plus tard, l'approche n'a pas décroché l'adoption attendue par rapport à l'approche objets, le paradigme de développement le plus populaire actuellement [Fre 09]. Parmi les raisons qui ont entravé sa large adoption on trouve : (1) la prise de conscience (c'est encore moins facile à utiliser), (2) l'absence d'un support universelle, et (3) son adoption relativement moindre par les experts et les professionnels [Rog 02, Des 05, Mad 07]. Le succès de ce nouveau paradigme s'appuie fortement sur la disponibilité d'outils supports solides permettant son intégration à l'échelle académique et industrielle.

Nous convenons que dans le contexte pédagogique, la compréhension du modèle abstrait d'un langage de programmation est plus cruciale que sa syntaxe. Les débutants perdent souvent beaucoup de temps et d'efforts pour l'apprentissage des formalismes en suivant des règles strictes afin d'assurer l'exécution de leurs programmes. Ceci, diminue relativement la motivation pour la programmation et la compréhension de l'essence du paradigme adopté et même entrave leur créativité [Ste 06].

Pour le cas du paradigme "orienté-aspects", un grand défi (challenge) se pose pour l'édition de code (précisément lors de l'expression des préoccupations transversales), où généralement la plupart des abstractions qui sont significatives lors de la phase de conception peuvent-être perdue lors de l'implémentation en raison du manque d'outils supports solides et appropriés. De plus, les programmeurs et principalement les débutants éprouvent quelques difficultés avec les formalismes syntaxiques de certains concepts et mécanismes offerts par ce paradigme [Wal 99, Ste 06, Muh 10]. Une étude détaillée sur les stratégies de mise en œuvre pour ce paradigme est fournie dans [Zhe 11].

De ce fait, afin de diminuer ces difficultés, il est devenu nécessaire de proposer une méthodologie adéquate pour les implémentations orientées aspects. Il y a une nécessité de prendre le style de codage hors du style conventionnel en introduisant plus d'interactivité et flexibilité.

Vis-à-vis ce qui précède, nous énonçons nos objectifs comme suit :

- **Objectif à long terme :** nous visons à fournir une nouvelle méthodologie de programmation orientée-aspects *(AO-specific programming facility)*, permettant aux programmeurs, d'une part, d'exprimer les préoccupations transversales en manipulant les divers concepts et mécanismes d'une manière interactive à l'aide des représentations graphiques (*Visual Code Development*) et d'autre part, rendre les entités

de code source, et leurs dépendances, explicitement visibles et accessibles en construisant des vues de visualisation (*Graphical Code Illustration*).

Il est plus approprié de parler d'une génération avancée d'IDEs avec une interface de programmation hybride qui vise à améliorer la manière d'interagir visuellement avec un code orienté-aspects en particulier pour les grands systèmes logiciels. Cela permet de faciliter la compréhension pendant les activités de développement et de maintenance. Nous considérons ce travail comme une ébauche de recherche pour faire émerger une nouvelle approche visant à mettre l'accent sur les innovations, plutôt que sur des détails d'implémentation.

- **Objectif à court terme :** notre travail initial porte principalement sur le contexte pédagogique, où nous offrons un outil support préliminaire aidant à la programmation orientée-aspects en ASPECTJ. L'objectif principal est d'apporter des techniques visuelles permettant de réduire l'utilisation des formalismes syntaxiques afin de simplifier le codage et la compréhension en termes de temps et d'effort.

  Nous visons à : (1) accroître la motivation de programmer en orienté-aspects, et (2) améliorer l'efficacité des tâches pédagogiques, de former les débutants et les soutenir à se familiariser avec ce nouveau paradigme afin d'accroître son adoption.

Dans les sections suivantes, nous nous donnons une description détaillée de la méthodologie proposée ainsi que sa mise en œuvre.

5.2 Vue détaillée de l'approche

5.2.1. Vers une approche hybride de codage

Les implémentations du paradigme "orienté-aspects" introduisent une nouvelle dimension à la programmation en combinant deux méta-niveaux lors du processus de codage (pour le code métier et le code technique). Cela, engendre une complexité et une éventuelle résistance principalement chez les débutants.

Nous proposons une méthodologie hybride pour la programmation orientée-aspects "**HM4AOP**" (le nom est l'acronyme de : **H**ybrid **M**ethodology **for** **A**spect-**O**riented **P**rogramming). C'est une tentative de soutenir le programmeur durant le processus de codage en introduisant plus d'interactivité avec un certain degré de flexibilité entre les deux méta-niveaux de l'AOP. L'objectif principal est de concentrer le temps et l'effort du programmeur sur les innovations tout en cachant les détails d'implémentation sous-jacents afin de minimiser l'influence des formalismes syntaxiques.

Dans notre proposition, nous retenons les avantages du style textuel, tout en permettant l'introduction de l'aspect visuel dans le code source où il est bénéfique. Nous distinguons alors deux styles de programmation. Le premier de bas niveau correspondant au langage conventionnel de programmation (en anglais : *Host-Language*). Le second style est de haut niveau (en anglais : *Domain-Specific Language*) présenté à travers l'aspect visuel introduit pour exprimer les nouveaux concepts et mécanismes du paradigme "orienté-aspects".

En agissant ainsi, il est important de donner la possibilité au programmeur de basculer entre les deux styles. Par conséquent, les techniques visuelles et textuelles viennent se compléter les unes les autres, et on peut tirer profit des avantages de chacune et d'éviter leurs limites.

La figure suivante (**Figure 5.1**) donne une vue simplifiée de notre proposition. On décrit le processus de programmation orientée-aspects selon notre méthodologie conjointement avec le style conventionnel adopté au codage textuel et qui comprend les étapes suivantes :

- Implémentation des abstractions primaires (code métier).
- Identifier les préoccupations qui entrecoupent (crossccut) les abstractions primaires.
- Définir les aspects qui encapsulent chaque préoccupation.
- Tisser les aspects dans les abstractions primaires. Le tissage est un processus spécial de transformation de programme qui est utilisé pour convertir un code orienté-aspects en un code orienté-objets avec les aspects intégrés dans le code de base.
- Compiler et exécuter le programme résultant.

Selon ce codage hybride, nous pouvons distinguer deux niveaux : une programmation de bas-niveau et l'autre de haut-niveau (*Low-level* & *High-level Programming*). La première est présentée par un langage conventionnel textuel (TPL) qui exige de l'utilisateur de connaître d'une manière approfondie les concepts de programmation et leurs formalismes syntaxiques, par exemple l'utilisation du langage ASPECTJ à travers l'outil support "AJDT" (AspectJ Development Tools) [W2]. Tandis que la programmation de haut-niveau est effectuée selon la méthodologie proposée, l'utilisateur n'est pas obligé de se concentrer profondément sur les détails syntaxiques. L'outil fournit une interface avancée permettant de spécifier et d'exprimer visuellement un modèle du programme cible. En outre, ce modèle peut être exporté comme un simple fichier qui sera utilisé ultérieurement pour générer des templates textuelles du code source au moyen d'un générateur approprié.

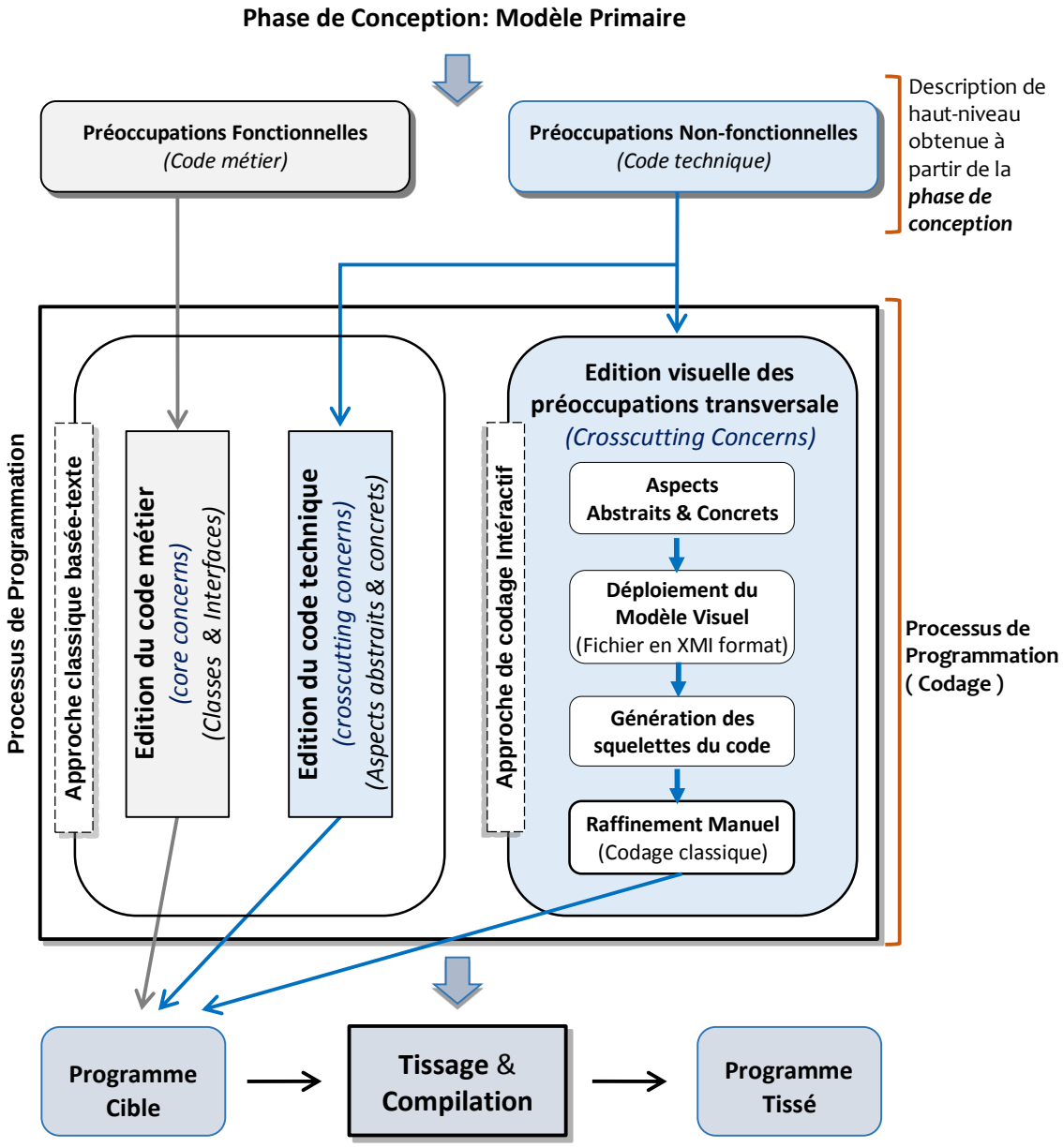


Figure 5. 1 Processus globale d'une construction hybride des programmes orientés aspects selon l'approche "HM4AOP".

5.2.2. Approche proposée

En partant de l'introduction de l'aspect visuel, la méthodologie que nous allons proposer doit, d'une part, définir une interaction de haut niveau, et, d'autre part, préserver la puissance de la méthodologie classique à base de texte et faire en sorte que ces deux méthodologies se complètent les unes les autres, et le choix le plus pertinent de l'une ou de l'autre dépend du contexte. Quand la situation nécessite un pouvoir d'expression accessible par l'une et l'autre, elles peuvent être utilisées conjointement.

La **Table 5.1** donne une brève comparaison entre la méthodologie conventionnelle de codage et notre proposition. Pour chacune, on décrit les informations clés, ainsi que les éléments requis au sein de l'outil support correspondant.

Table 5. 1 Codage Conventionnel *vs.* Méthodologie de Codage Hybride.

	Méthodologie Classique de Codage (Text-based code editing)	**Méthodologie de Codage Hybride** (General-purpose partially-visual code editing)
Outil support de l'édition	Editeur conventionnel de code (CE, *Code Editor*)	Concepteur ou éditeur visuel de la structure de code source (VMD, *Visual Model Designer*)
Élément de codage	Codage textuel (*Code-oriented method*)	Combinaison flexible du codage textuel et des constructions graphiques (*Interactive Code Editing*).
Règles de construction	Formalismes syntaxiques du langage de programmation	Représentations visuelles & spécifications pour construire un modèle visuel (VM, *Visual Model*) du programme cible (description et spécification de haut niveau de la structure ou le squelette du code source).
Style de codage	Codage entière sous un éditeur textuel	Techniques d'interaction via une interface graphique de programmation hybride ; par exemple la technique "Glisser-Déposer" (*Drag-and-Drop*).
Outils supports post-édition	Compilateur conventionnel (p. ex. *ajc, AspectJ Compiler*)	Composants de programmation visuelle : ① **Constructeur du modèle** (VMD) dont le but est de construire un modèle visuel du programme cible. ② **Générateur du code orienté-aspects** (*AOP Code-Generation Engine*) pour assurer la transformation du modèle visuel en des templates textuelles du code source (*Code Skeletons Templates*).

La **Figure 5.1** montre un aperçu de l'approche proposée (**HM4AOP**, **H**ybrid **M**ethodology **for A**spect-**O**riented **P**rogramming).

Notre objectif est d'introduire une nouvelle génération d'environnements pour rendre le processus de développement plus attractif et plus souple. Un environnement HM4AOP

dispose d'une une interface graphique de programmation hybride, efficace et évolutive (*Scalable Interface*) ce qui améliore la capacité de visualiser, de modifier et d'interagir avec le code source.

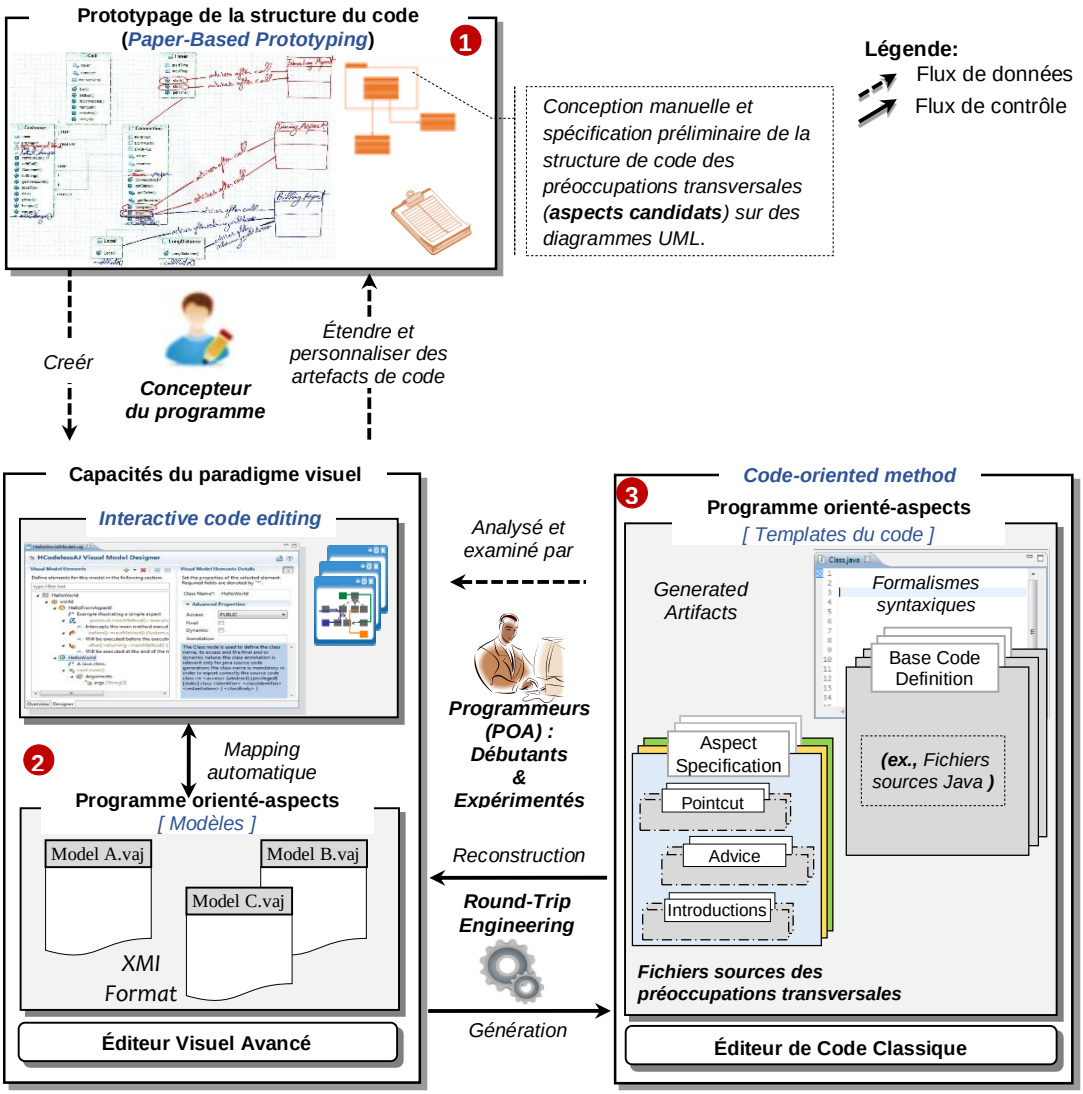


Figure 5. 2 Vue globale de l'approche proposée.

La **Figure 5.2** montre une vue globale du nouveau processus de codage. On le récapitule dans les trois étapes suivantes.

La première étape (marqué ❶ dans la **Figure 5.2**), un prototypage rapide de code (*Rapid Code Prototyping*), exige du programmeur de concevoir et spécifier à la main des templates préliminaires de la structure de code des préoccupations transversales (aspects candidats) ainsi que ses entités et leurs dépendances sur des diagrammes UML. Ce concepteur de code utilise le langage naturel conventionnel comme un simple langage de script avec un ensemble

d'artefacts expressifs sans aucunes définitions structurales ou formalismes syntaxiques. Ce prototypage sur papier (en anglais : *Paper-Based Prototyping*) est considéré comme une implémentation initiale (en anglais : *Crosscutting Code's Design*) aidant à découvrir les anomalies de conception plus particulièrement pour le code non-fonctionnel (aspects) juste avant l'étape de codage.

À la deuxième étape (marqué ❷ dans la **Figure 5.2**), une édition interactive de code (en anglais : *Interactive Code Editing*), le programmeur exprime la structure de code du programme cible et certains comportements selon les templates prototypés sur les diagrammes UML en termes d'éléments visuels au sein d'un éditeur visuel. Une fois cette construction graphique achevée, le modèle visuel résultant peut être facilement examiné, contrôlé manuellement et discuté en utilisant des vues interactives du code source.

Ces deux étapes constituent un méthode agile pour les programmeurs afin de concevoir des templates préliminaires du squelette de code source, mais aussi une méthode clé permettant aux programmeurs de:

- Editer des templates préliminaires du code des préoccupations à une étape précoce,
- Raffiner et apporter des changements significatifs rapidement dans le code,
- Impliquer les membres de l'équipe, au sien d'un environnement collaboratif, pour obtenir rapidement des feedbacks,
- Etre plus créatif — expérimenter beaucoup d'idées sans s'engager dans codage.
- Élever la productivité — épargner l'effort et le temps de programmation en résolvant les principaux problèmes dès le début. Les changements imprévus peuvent être extrêmement coûteux à mettre en œuvre lorsqu'ils sont découverts plus tard lors de la phase de codage. A l'inverse, l'identification précoce de ce qui est vraiment nécessaire permet d'obtenir avec moins d'efforts un code de haute qualité en particulier pour les systèmes logiciels de grande taille.

Afin de simplifier l'interchangeabilité du code visuel entre les développeurs et leurs outils de développements préférés, le modèle visuel est sérialisé à l'aide d'un fichier de déploiement XMI (*XML Metadata Interchange*) [W23]. Ce fichier de description peut être ensuite utilisé par un générateur pour produire en sortie des templates de code orienté-aspects.

À la troisième étape (marqué ❸ dans la **Figure 5.2**), c'est le codage textuel classique où le programmeur n'a alors plus qu'à raffiner et rééditer les templates générés en complétant les différents endroits du code qui lui sont clairement indiqués par des commentaires jusqu'à l'obtention d'un code final opérationnel pour le programme cible.

Suite à ce nouveau processus de codage, on décrit, dans la **Figure 5.3**, le flux des principales activités à l'aide d'un exemple démonstratif pour l'implémentation ASPECTJ.

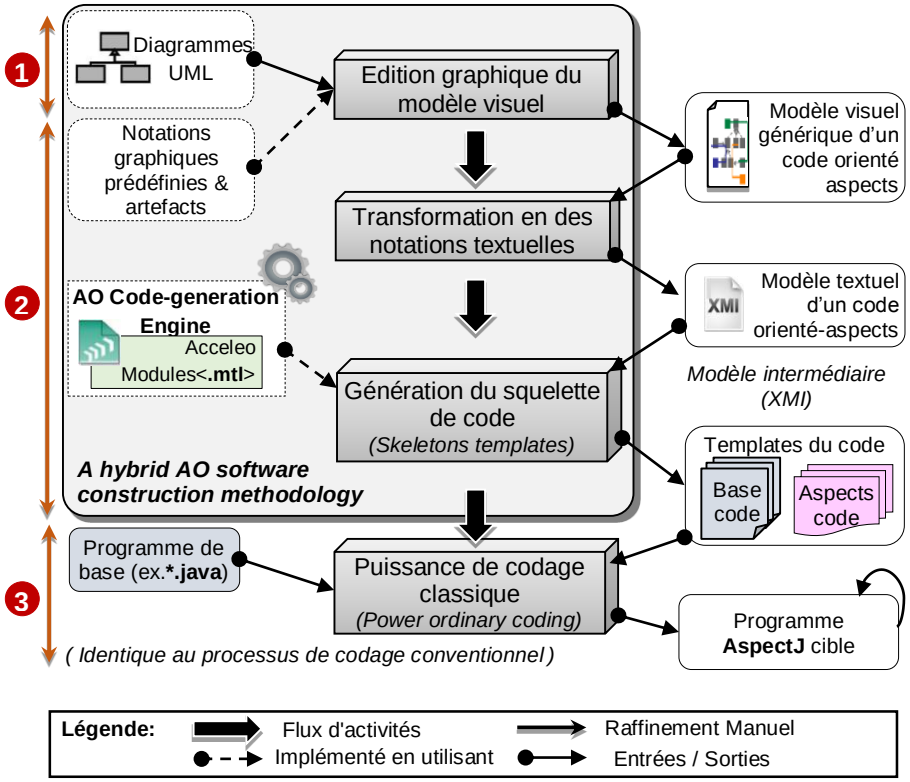


Figure 5. 3 Étapes du processus de codage.

Notre idée est de s'éloigner de la méthode conventionnelle à base de texte à travers la proposition d'une nouvelle méthodologie en particulier pour la programmation orientée-aspects. Concevoir interactivement le squelette d'un programme (la structure du code), et spécifier de manière hybride les divers concepts et mécanismes programmatiques de l'orienté-aspects à l'aide des représentations graphiques prédéfinies représente une idée prometteuse pour aller au-delà des restrictions de la méthode textuelle et même de remédier aux difficultés posées souvent par les formalismes syntaxiques complexes.

C'est un nouveau style de codage hybride "codeless" en incorporant des techniques visuelles et textuelles durant la phase de codage. Nous imitons la technique conventionnelle "*Drag-and-Drop*" utilisée au sein de la majorité des éditeurs graphiques mais d'une nouvelle façon, afin de diminuer la saisie du code à la main. L'objectif principal est de permettre aux développeurs de se concentrer sur la conception de leurs programmes et sur les innovations au lieu de perdre plus d'efforts sur les détails d'implémentation tel que le formalisme syntaxique.

Selon notre méthodologie (HM4AOP), l'éditeur visuel proposé (VMD) est complémentaire et non concurrent à l'éditeur conventionnel de code. En premier lieu, au sein de cet éditeur, le programmeur crée un squelette visuel de son programme cible avec des spécifications en utilisant une collection de constructions graphiques avec une interactivité de haut-niveau. Ensuite, dès que l'étape de génération des templates de code est

achevée, il peut les rééditer sous l'éditeur classique en ajoutant le reste du code essentiel pour finaliser le programme. De cette façon, les efforts de concentration sur les formalismes syntaxiques seront réduits, et le programmeur sera plus productif et même plus créatif. Une fois l'étape de conception et spécification sous le VMD achevée, des templates textuelles de code peuvent être générées à partir du modèle visuel crée avec une possibilité de l'exporter directement pour une utilisation ultérieure.

Dans la section suivante, nous donnons une description détaillée de la mise en œuvre de notre approche, selon le volet conceptuel et le volet implémentation. Pour ce faire, nous présenterons un prototype de l'outil support.

5.3 Mise en œuvre de l'approche

Afin de montrer la faisabilité pratique de notre approche, nous avons conçu et implémenté un outil support préliminaire sous la plateforme ECLIPSE. Nous avons choisi le langage ASPECTJ pour être notre première implémentation cible car c'est une référence de l'approche orientée-aspects et il est le plus couramment utilisé. L'outil "AJDT" d'ECLIPSE est sans doute le support le plus complet, et le plus mature pour ASPECTJ. Il est considéré dans la communauté comme un représentant Open source pour l'implémentation ASPECTJ [Col 04, W3].

Nous avons développé jusqu'à présent un prototype, appelé **"HCodelessAJ"** (le nom est l'acronyme de : **H**ybrid **Codeless** Methodology for **A**spect**J**). Plus précisément au niveau technique, c'est un outil de codage iconique à usage général (general-purpose iconic-based coding tool) qui s'appuie sur "AJDT" et comprend une implémentation initiale sous la plateforme ECLIPSE. D'autres extensions prévues sont mentionnées comme des perspectives dans le **Chapitre 7**.

L'outil que nous proposons est la première expérimentation concernant ASPECTJ qui introduit l'aspect visuel en programmation pour supporter les innovations de ce langage, plutôt que les détails d'implémentation. Notre objectif est d'offrir aux programmeurs une capacité de codage dans un haut niveau d'interactivité en évitant les écueils de l'approche textuelle (less hand-typing) et sans la nécessité de se concentrer profondément sur les formalismes syntaxiques.

Il offre une interface de programmation hybride avec les caractéristiques suivantes :

- Les utilisateurs peuvent présenter la structure du programme et certains comportements dans une combinaison flexible du code textuel et des objets graphiques dans un éditeur visuel (*Visual Structured Code Editor*).
- Les représentations visuelles considérées sont couplées avec des techniques d'interaction afin de simplifier la navigation et la compréhension rapide du code. Les utilisateurs peuvent facilement vérifier la complétude (*Completeness*) des entités

logicielles qui composent le programme ainsi que l'uniformité (*Consistency*) de ces dépendances.

- Contrôle facile des codes qui ont été mis au point durant les étapes de développement. Cela est crucial en ingénierie des logiciels complexes où les développeurs doivent faire face à des programmes de grande taille et souffrent souvent de surcharge cognitive.

5.3.1 Architecture

Dans ce qui suit, nous allons tout d'abord décrire l'architecture actuelle du prototype implémenté et ses principaux composants. Ce prototype est en pleine évolution comme une collection de plug-ins au sein de la plateforme ECLIPSE.

La **Figure 5.4** montre une vue de haut-niveau de l'architecture structurelle. C'est une architecture ouverte, ce qui signifie qu'il n'y a pas de limites pour l'extensibilité tant interne qu'externe.

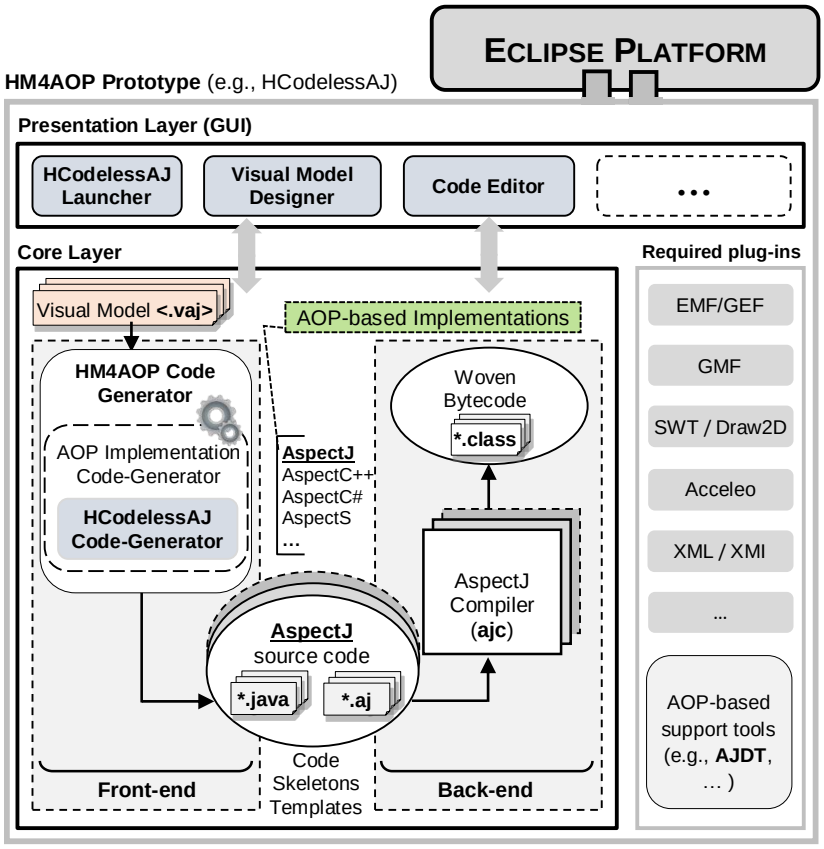


Figure 5. 4 Architecture structurelle de l'outil support de l'approche "HM4AOP".

Elle se compose de deux parties principales : le noyau "*Core Layer*" et l'interface utilisateur "*Presentation Layer (GUI)*". La partie noyau est constituée également de deux composants : un générateur spécifique de code pour une implémentation orientée-aspects cible (p.ex. *HCodelessAJ Code Generator*), absolument implémenté comme un front-end pour le noyau. Le back-end pour le noyau est le compilateur conventionnel relatif à l'implémentation cible (p.ex. ajc, *AspectJ Compiler*). La partie interface est constitué de trois composants principaux : l'activateur courant de l'outil "*HCodelessAJ Launcher*", l'éditeur visuel (VMD, *Visual Model Designer*) et l'éditeur de code classique (CE, *Code Editor*).

Dans ce qui suit, nous exposons les principaux composants (ou modules) au sein de architecture actuelle.

✪ *Concepteur de modèle visuel (VMD, Visual Model Designer)*

Le concepteur de modèle visuel est le composant fondamental dans l'outil. C'est un éditeur assistant de codage visuel, où l'utilisateur peut manipuler des notations et constructions visuelles interactives pour éditer un modèle visuel (*.vaj) pour le squelette du code du programme cible. C'est un éditeur graphique de la structure du code (*Graphical Editor of Code Structure*), offrant une vue interactive permettant d'obtenir un aperçu rapide sur les divers préoccupations à prendre en considération à partir de la spécification préliminaire du modèle. Le VMD est un assistant complémentaire et non concurrent à l'éditeur classique de codage (CE, *Code Editor*). On gère les entités et les concepts de programmation du code source d'une manière interactive à l'aide des éléments graphiques associés du modèle visuel (MEs, *Model Elements*).

Ce concepteur est vu comme une nouvelle génération d'éditeurs de code hautement interactif offrant les caractéristiques suivantes :

- Réduire la saisie du code (hand-typing) à l'aide d'un style hybride.
- Contrôler et modifier la structure du programme et ses entités avec un haut-niveau de flexibilité et sans avoir besoin d'examiner plus profondément le code source. Le programmeur peut interagir à l'aide de plusieurs actions avec n'importe quel élément du modèle avant de régénérer les templates de code correspondants à la fin.
- Importer et exporter les modèles visuels créés pour une réutilisation ultérieure.

✪ *Générateur de code (HM4AOP Code Generator)*

Comme montré précédemment dans la **Figure 5.4**, pour le cas de l'outil "HCodelessAJ", nous avons prévu un composant spécifique à l'implémentation orientée-aspects cible (*HCodelessAJ Code Generator*).

C'est un extracteur de code ASPECTJ (*Template-based ASPECTJ Code Extractor*) basé sur la technologie ACCELEO [W9]. Il prend, comme entrée, le modèle visuel (*.vaj) construit auparavant sous le concepteur visuel (VMD), et produit en sortie des templates textuelles montrant les squelettes du code pour différentes entités du programme cible. Nous expliquerons en détail le processus de transformation plus loin dans ce chapitre.

✪ *HCodelessAJ Launcher*

C'est le module responsable de l'activation du noyau "*HCodelessAJ Core"* de l'outil avec les principales fonctionnalités. Il contrôle aussi l'interface avec les diverses ressources offertes par la plateforme ECLIPSE.

La **Figure 5.5** montre un aperçu de l'architecture fonctionnelle du prototype "HCodelessAJ". C'est un arrangement de fonctionnalités avec les interfaces définissant la séquence des flux de contrôle et de données durant le processus de codage. Les flèches descendantes représentent le flux ou les étapes des fonctionnalités au sein de l'outil. Les flèches horizontales pointillées montrent le flux de données, alors que les autres montrent le flux de contrôle.

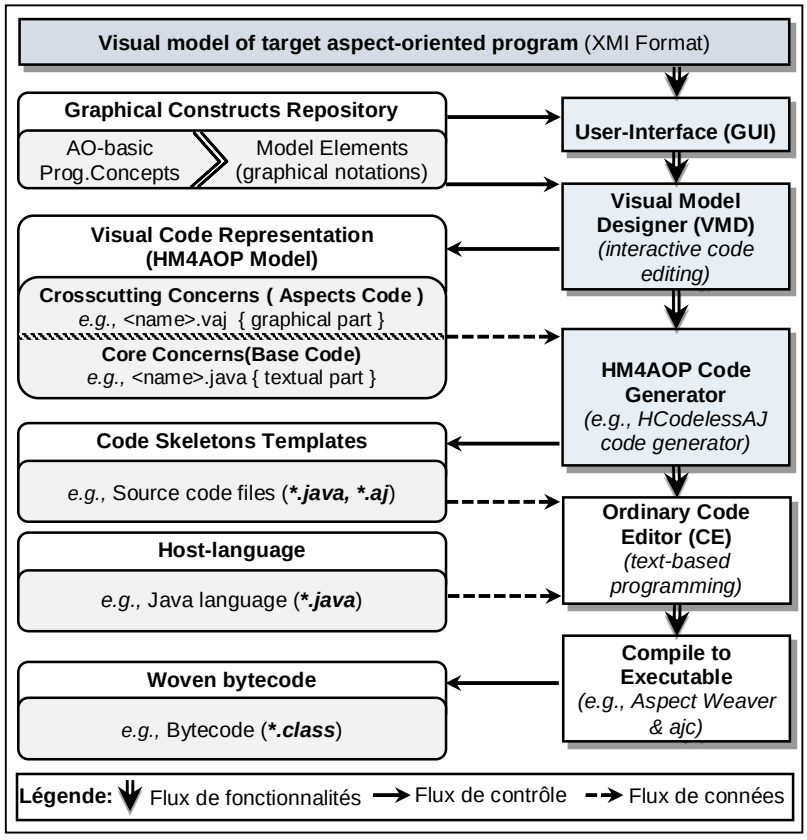


Figure 5. 5 Architecture fonctionnelle de l''outil support de l'approche "HM4AOP".

5.3.2 Implémentation

Dans cette section, nous décrivons successivement: (1) le processus à suivre pour mettre en œuvre l'approche proposée, (2) les frameworks et les technologies à base de l'ingénierie dirigée par les modèles (IDM) adoptées, (3) le prototype de l'outil support implémenté et (4) une description détaillée du processus de transformation et de génération de code.

5.3.2.1 Étapes de mise en œuvre de l'approche

Le processus à suivre pour mettre en œuvre notre proposition est constitué de trois principales étapes décrites dans la partie droite de la **Figure 5.6** et énumérées ci-dessous :

✪ *Définition du méta-modèle pour un langage orienté-aspects*

Après la sélection de l'implémentation cible (*AOP-based implementation language*), on définit le méta-modèle correspondant (p. ex. "*AspectJ.ecore*" un modèle Ecore pour le langage ASPECTJ) capable de représenter le modèle visuel de l'implémentation ASPECTJ cible.

Un modèle Ecore (*.ecore) est un méta-modèle EMF du langage cible, généré à partir de son profile UML (*.uml). Il est utilisé pour garantir la conformité des templates de code générés au langage (c.-à-d. la syntaxe du code généré doit être définie selon ce méta-modèle). De plus, un modèle du générateur (*.genmodel, Generator model), associé au modèle Ecore déjà définis, est indispensable pour la génération de code (p. ex. "*AspectJ.genmodel*").

A l'inverse du modèle Ecore qui définit seulement un modèle indépendant des plateformes dits PIM (*Platform-Independent Model*), le modèle du générateur définit un modèle lié aux plateformes dits PSM (*Platform-Specific Model*).

✪ *Développement d'un éditeur de codage graphique pour le méta-modèle EMF*

Un éditeur graphique a été développé afin de permettre au programmeur de créer, visualiser et modifier les modèles visuels construits et qui représentent des instances de méta-modèle EMF. C'est un éditeur avancé de codage (VMD, *Visual Model Designer*) qui prend en charge diverses techniques d'interaction tels que "*Drag-and-Drop*", "*Cut-and-Paste*", etc.

✪ *Développement d'un générateur de code*

Vu que notre proposition est une approche basée-modèle (a template-based approach), le mécanisme de transformation (le mapping model-to-code) repose principalement sur la

technique ACCELEO (M2T, *Model-to-Text transformation*) comme un standard et technique alternative pour la génération de code [W9].

Ce mapping est effectuée par l'application des règles de transformation définies dans des modules ACCELEO (*.mtl) permettant de transformer d'une façon automatique le modèle visuel en des templates textuelles montrant le squelette d'un code orienté-aspects conformes à la l'implémentation cible.

Le moteur de génération de code (automatic code-generation engine) est formé d'une collection de modules ACCELEO. Chacun est défini par un ensemble de templates ACCELEO de génération (c.-à-d. des scripts pour personnaliser le générateur avec précision) donnant les règles requises pour la transformation. Ces templates sont implémentées en utilisant le langage ACCELEO au sein d'un outil support basé-EMF comme ObeoDesigner [W22]. Ces templates doivent être définis et validés avant de passer au processus de génération de code.

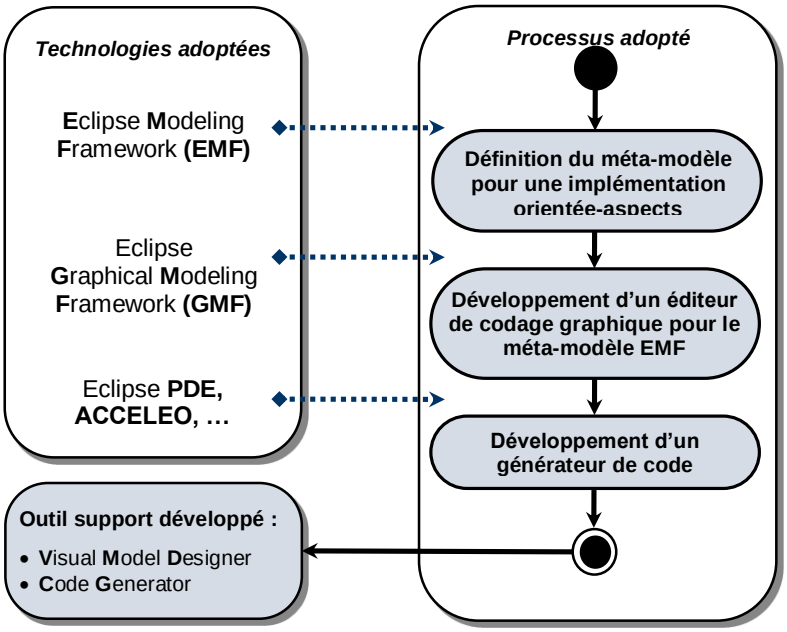


Figure 5. 6 Une vue globale du processus et technologies adoptées pour l'implémentation de l'outil support de l'approche "HM4AOP".

5.3.2.2 Technologies et framework adoptées

La technologie sous-jacente, que nous avons utilisé pour l'implémentation de l'outil support est la plateforme universelle ECLIPSE, une véritable plateforme de développement extensible grâce à son mécanisme de plug-ins. Un assistant spécifique (PDE, *Plug-in Development Environment*), et une architecture ouverte et extensible pour l'intégration de multiples outils de développement conduit à un développement totalement autonome.

En outre, la mise en œuvre de l'approche se base sur l'ingénierie dirigée par les modèles (MDE, *Model-Driven Engineering*) et des frameworks associées au sein du projet ECLIPSE à savoir EMF (*Eclipse Modeling Framework*) [W7], GEF (*Graphical Editing Framework*) [W8], GMF (*Eclipse Graphical Modeling Framework*) [W12] et la technologie ACCELEO [W9], dont le but de concevoir et créer l'éditeur visuel avec le générateur de code associé.

La **Figure 5.6** résume ces frameworks et technologies utilisées en implémentation. Pour définir un méta-modèle pour une implémentation cible (p. ex. ASPECTJ) nous avons utilisé EMF, et pour développer l'éditeur graphique (VMD), nous avons utilisé GMF.

Nous avons choisi ACCELEO comme un outil de définition de la transformation M2T (*Model-to-Text transformation*) car c'est un très bon soutien pour les meta-modèles EMF et pour la construction des générateurs de code. ACCELEO définit un langage à base de templates utilisé dans un outil support basé-EMF comme ObeoDesigner [W22], permettant la transformation des modèles instances conformes aux méta-modèles EMF en des templates textuelles (ex. code source). Ce langage est vu comme une alternative standard de génération de code [W9].

5.3.2.3 HCodelessAJ, Prototype de l'outil support

La **Figure 5.7** montre une capture d'écran du prototype "HCodelessAJ", développé en JAVA sous la plateforme ECLIPSE.

L'implémentation actuelle offre un support préliminaire de programmation orientée-aspects en ASPECTJ. Elle offre un éditeur simple pour une édition visuelle du code (partially-visual code editing) en illustrant les représentations graphiques considérées avec quelques techniques d'interactions prises en charge par l'interface graphique. C'est une implémentation prototype illustrant la faisabilité de l'approche proposée.

Nous récapitulons les principales caractéristiques de la version actuelle comme suit: (1) Aspect visuel simple au sein de l'éditeur graphique (VMD) ; (2) Manipulation directe des concepts de programmation et des données à l'aide des éléments visuels du modèle; (3) Combinaison souple des deux méthodes graphique et textuelle; et (4) Une forte intégration des principaux composants : l'éditeur de modèle visuel (VMD), l'éditeur classique de code, le générateur de code (*HCodelessAJ Code Generator*) et "AJDT" sous la plateforme ECLIPSE.

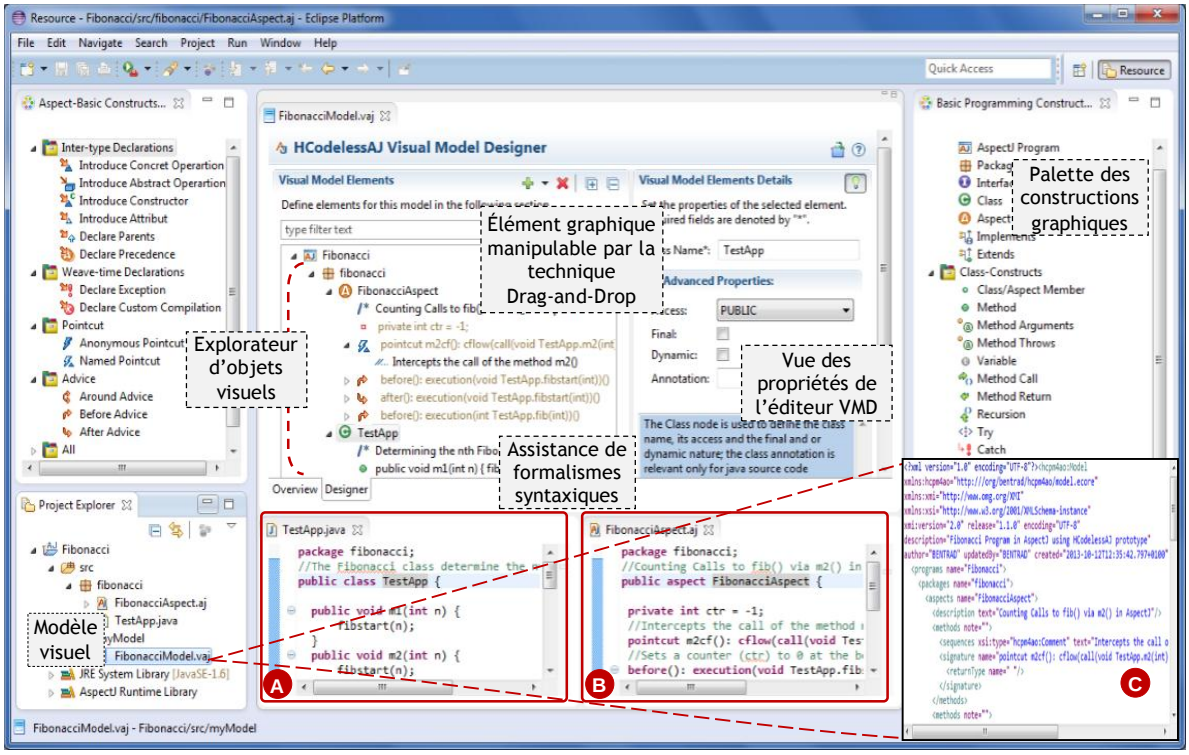


Figure 5. 7 Une capture d'écran du prototype "HCodelessAJ" sous la plateforme ECLIPSE.

(A) & (B) extraits de l'implémentation du programme Fibonacci respectivement dans "VMD" et l'éditeur conventionnel de codage; (C) Représentation XMI du modèle visuel créé.

Cette interface se présente sous la forme de vues ECLIPSE et fournit un style d'interaction selon la méthode "*Drag-and-Drop*". Elle pourrait être améliorée en offrant une possibilité de navigation entre les éléments du modèle et ceux qui leur correspondent comme templates générés au sein de l'éditeur de code.

La palette de constructions graphiques est une bibliothèque de représentations interactives explicites et intuitives, associées aux différents concepts de programmation du langage d'implémentation cible (ex. ASPECTJ) arrangées en catégories : *Structured-Constructs* (package, class, interface, aspect, loops, conditions, etc.) ; *Class-Constructs* (method, attribute, etc.) ; *Aspect-Constructs* (pointcut, advice, inter-type declarations, and weave-time declarations) ; etc. Dans cette implémentation prototype, nous avons considérés des représentations iconiques (*Visual iconic-based tool*). Notre choix reste assez simple ; il pourrait être intéressant d'investiguer d'autres représentations afin d'améliorer l'expressivité (expressiveness) de l'aspect visuel de l'outil pour les divers concepts et mécanismes du paradigme "orienté-aspects".

La **Table 5.2** résume un ensemble de notations graphiques proposées avec les implémentations textuelles correspondantes (les règles de transformations ACCELEO pour la génération des templates de code). Ces notations sont conçues pour représenter les concepts fondamentaux et mécanismes offerts par le paradigme "orienté-aspects"

implémentés en JAVA & ASPECTJ ainsi que leurs formalismes syntaxiques. L'**Annexe A** présente ce tableau avec plus de détails.

Table 5. 2 Extrait de la table des notations graphiques proposées avec les règles de transformations ACCELEO.

AO-Basic Constructs & Features		Graphical Notations (iconic elements)	Corresponding Syntactic Formalisms of Static AO-Basic Constructs and Features (e.g., ASPECTJ) **ACCELEO Rules of Model Transformation** (i.e., ACCELEO templates)
Aspect Constructs	Advice	Before Advice Around Advice After Advice	**aspect ::= <access>** [abstract] [privileged] [static] aspect **<identifier> <classIdentifier> <instantiation> { <aspectBody> }** **<access>** ::= [PUBLIC | PROTECTED | PRIVATE] **<identifier>** ::= letter {letter | digit} **<classIdentifier>** ::= [DOMINATES] [EXTENDS] [IMPLEMENTS] **<instantiation>** ::= [ISSINGLETON | PERTHIS | PERTARGET | PERCFLOW | PERFLOWBELOW] **<aspectBody>** ::= {<aspectFeature>} **<aspectFeature>**::= <features> | <introduction> | <declareParents> | <declarePrecedence> | <pointcut> | <advice> | <declareSoft> | <customCompilation> **<advice> ::= [returnType] <adviceType> "(" [formals] ")" [afterQualifier] [throwsClause] ":" <attachedpointcut> "{"<adviceBody>"}"** **returnType ::=** TypeOrPrimitive ; (applies only to around advice) **<adviceType> ::=** before | after | around **afterQualifier ::=** throwing | returning (applies only to after advice) **[formals]** as a Java parameter list **<attachedPointcut> ::=** <pointcut> | <namedPointcut> | <abstractNamedPointcut> **<adviceBody>** as a Java method body (with a specific method: proceed()) ...
	Pointcut	Anonymous Pointcut Named Pointcut	
	Inter-Type Declaration of members	Introduce Concrete Method Introduce Abstract Method Introduce Constructor Introduce Field Declare Parents Declare Precedence Declare Exception Declare Custom Compilation	

Class Constructs & Structured Statements	Structure	Program
		Package
		Class
		Interface
		Aspect
			[template public getAspectBody(aAspect : AspectElement)]
			[if(aAspect.members->notEmpty())]
			[for (aMember : AspectMember | aAspect.members)]
			[getComment(aMember.description,aMember.note)/][getAnnotation(aMember.annotation)/][getAccess(aMember.access)/][getMemberDeclaration(aMember.declaration)/]
			[getBaseType(aMember.type)/][getMemberName(aMember)/];
			[/for]
			[/if]
			[for (aMethod : Method | aAspect.methods)]
			[getComment(aMethod.description,aMethod.note)/][getAnnotation(aMethod.annotation)/][getOverride(aMethod.signature.declaration)/][getAccess(aMethod.signature.access)/]
			[getDeclaration(aMethod.signature.declaration)/][getReturnType(aMethod.signature)/] [getMethodName(aMethod)/]([getArgsElement(aMethod)/]) {
			[for (s : Sequence | aMethod.sequences)]
			[getSequence(s)/]
			[/for]}
			[/for]
			[for (aAdvice : Advice | aAspect.advices)]
			[getComment(aAdvice.description,aAdvice.note)/][getAnnotation(aAdvice.annotation)/][getOverride(aAdvice.signature.declaration)/][getAccess(aAdvice.signature.access)/][getDeclaration(aAdvice.signature.declaration)/][getReturnType(aAdvice.signature)/] [getAdviceName(aAdvice)/]([getArgsElement(aAdvice)/])
			{
			[for (s : Sequence | aAdvice.sequences)]
			[getSequence(s)/]
			[/for]}
			[/for]
			...
			[/template]

	Conditions, Loops & Inheritance	Switch Case If Else Else if Iteration Implements Extends ...	
	Documenting	Description Discussion Comment Source Link Group ...	

```
[template public genAspectElement(aPackageName:String, aAspect :
AspectElement)]
[if ((not aAspect.name.oclIsUndefined()) and aAspect.name.trim().size()>0)]
[file (aPackageName.substituteAll('.',
'/').concat('/').concat(aAspect.name.trim()).concat('.aj'), false)]
[genPackageName(aPackageName)/]
[getImportBlock(aAspect)/]
[getComment(aAspect.description,null)/]
[getAnnotation(aAspect.annotation)/]
[getAccess(aAspect.access)/][if(aAspect.final)] final [/if]
aspect [aAspect.name/][if (aAspect.extendsElements->notEmpty())]
extends [genAspectExtendsElement(aAspect)/][/if]
[if (aAspect.implementsElements->notEmpty())]
implements [genAspectImplementsElement(aAspect)/][/if] {
       [getAspectBody(aAspect)/] }
[/file][/if]
[/template]
                                        ...
```

Le modèle visuel créé décrit le squelette du programme cible avec des spécifications du comportement. Dans sa forme la plus simple, c'est une visualisation interactive permettant d'avoir rapidement une vue globale sur le code source.

La spécification complète des données de ce modèle est exportée dans un fichier de taille réduite définis sous le format XMI avec l'extension (*.vaj). Afin d'élaborer plus de détails, ce modèle peut contenir également des artefacts importants comme la documentation en différents niveaux d'abstraction (p. ex. modèles UML, graphes, tableaux, descriptions textuelles, enregistrements, etc.). Avoir ces données supplémentaires ensemble dans un seul entrepôt permet de créer des liens utiles entre les templates de code généré et ces artefacts. De plus, Il est intéressant de mettre la source du programme sous forme d'un document visuel interactif [Fre 14].

5.3.2.4 Processus de transformation

Plusieurs propositions ont été identifiées sur les transformations de modèles et la génération de code. Les plus pertinents pour le paradigme orienté-aspect sont introduits dans [Ben 10], par *J. Bennett, K. Cooper* et *L. Dai.* Ils ont proposé une approche de génération d'un code ASPECTJ. La transformation entre les modèles s'effectue au moyen de spécifications XML et meta-modèles de XML et du langage ASPECTJ.

Dans [Abi 13a, Abi 13b], *Abid Mehmood* et *Dayang N. A. Jawawi* ont effectué une étude systématique des travaux de recherches dans le domaine de la génération de code orienté-aspects à base de modèles (aspect-oriented model-driven code generation). Les méthodes existantes s'appuient sur les modèles UML avec juste la possibilité de générer des squelettes de code. Le travail le plus lié au mécanisme que nous adopte est le travail de *Hyun Seung* [Hyu 13]. Il a proposé une méthode basée sur la technique d'ACCELEO pour générer un code JAVA plus sophistiqué pour la plateforme Android à partir des diagrammes de classe et de séquence d'UML. Jusqu'à présent, notre mécanisme de transformation s'occupe de la

génération automatique des templates code ASPECTJ/JAVA à partir des modèles visuels, déjà construits et déployés en format XMI. Pour cela, nous utilisons des règles de transformation ACCELEO prédéfinies.

La **Figure 5.8** donne une vue simplifiée sur le processus de génération de code. En particulier, la figure met en évidence les technologies adoptées à chaque étape du processus. La partie **(A)** montre le processus MDA (*Model-Driven Architecture*), comme décrit dans [Kle 03]. La partie **(B)** illustre le processus de transformation d'un modèle visuel en des templates textuelles pour le programme cible.

Le processus globale s'effectue en deux phases principales. En premier lieu, il prend comme entrée le modèle visuel créé, conformément au méta-modèle EMF (p. ex. *AspectJ.ecore*) et enregistré dans un fichier de déploiement de format XMI (*.vaj). La seconde étape, est l'analyse et la génération des templates des squelettes de code source (contenant la structure du code et certaines spécifications du comportement) selon le langage d'implémentation cible (p. ex. temlates de code en ASPECTJ).

Dans le cas de notre première implémentation prototype de l'outil support "HCodelessAJ" (voir la **Figure 5.4**), le générateur de code approprié (*HM4AOP Code Generator*) est un composant générateur spécifique (*HCodelessAJ Code-Generator*) utilisant les modules ACCELEO et le modèle du générateur associé au méta-modèle EMF du langage cible (ex. *AspectJ.genmodel*) pour générer un code ASPECTJ. Les templates générés peuvent être réédités en complétant les différents endroits du code qui sont clairement indiqués par des commentaires jusqu'à l'obtention d'un code final opérationnel pour le programme cible.

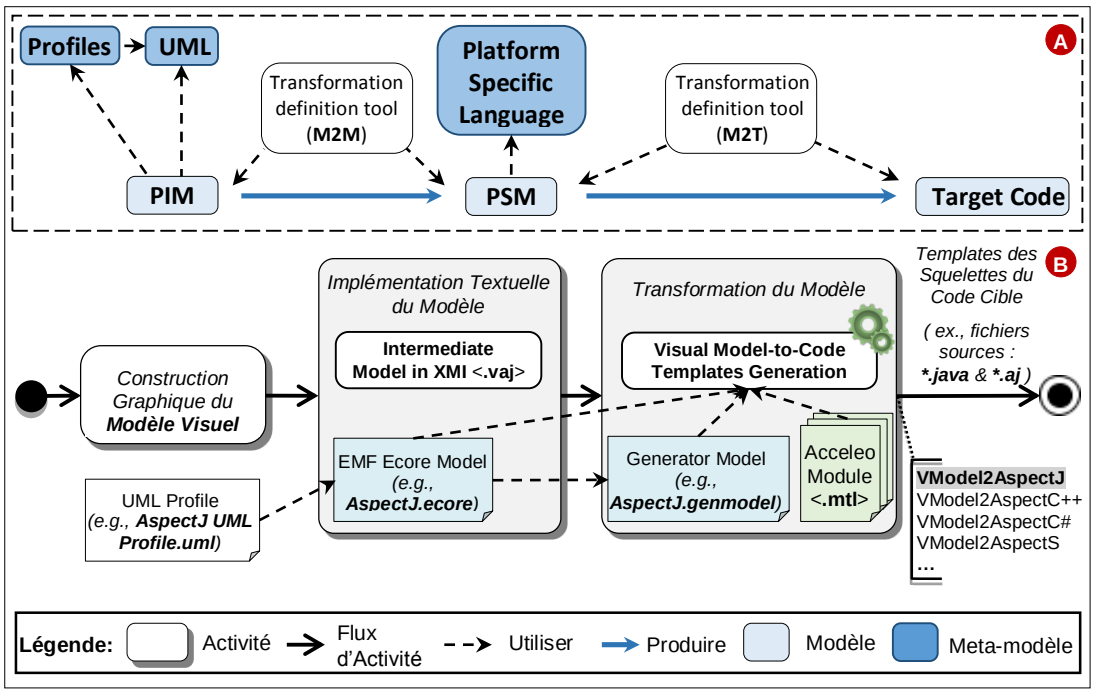


Figure 5. 8 Vue d'ensemble du processus de transformation et de génération des templates de code sous le prototype "HCodelessAJ".

5.4 Conclusion

Nous avons présenté notre deuxième contribution dans ce chapitre. Il s'agit d'une appoche de codage orienté-aspects basée sur une technique hybride en combinant les avantages de l'approche visuelle et la méthode conventionnelle basée texte.

Pour démontrer l'aspect faisabilité, nous avons concrétisé cette proposition par la mise en œuvre d'un prototype de son outil support "HCodelessAJ" sous la plateforme ECLIPSE. Cette mise en œuvre est considéré comme un premier pas pour éviter les écueils de la méthode textuelle afin de gérer au mieux les diverses préoccupations.

Nous pensons que cette contribution constitue une approche prometteuse. L'aspect visuel introduit reste toujours bien complémentaire et non compétitif à la méthode conventionnelle. De plus, nous prédisons qu'il aura un impact majeur sur les outils supports d'autres implémentations orientées aspects.

CHAPITRE 6

ÉVALUATION ET DISCUSSION

SOMMAIRE

6.1 PROJETS SELECTIONNES POUR L'EVALUATION 90

6.2 ÉVALUATION DU SYSTEME DE VISUALISATION 91

6.2.1 ÉTUDE DE CAS SUR "VIZZASPECTJ-2D" 91
6.2.2 ÉTUDE DE CAS SUR "VIZZASPECTJ-3D" 93
6.2.3 DISCUSSION 96

6.3 ÉVALUATION DE L'APPROCHE HYBRIDE DE CODAGE "HM4AOP" 97

6.3.1 ÉTUDES DE CAS PRELIMINAIRES 97
6.3.2 EXPERIMENTATION DETAILLEE 104

6.4 CONCLUSION 114

Ce chapitre donne une description détaillée de nos évaluations, suivie également des discussions. Dans un premier temps, nous décrirvons les expérimentations effectuées sur les outils supports proposés dans le cadre de notre première contribution. On les validera à travers des études de cas sur quelques exemples de benchmarks. Dans un second temps, nous nous intéressons à la validation de notre deuxième contribution concernant l'approche de codage hybride à travers une expérimentation détaillée que nous avons mise en place afin de démontrer les aspects faisabilité et efficacité.

6.1 Projets sélectionnés pour l'évaluation

Nous venons de présenter dans les deux chapitres précédents, nos principales contributions d'un point de vue concepts et mécanismes proposés et les outils supports implémentés. Nous allons maintenant nous intéresser aux expérimentations effectuées.

Des exemples suffisamment variés de programmes benchmarks et des projets en ASPECTJ sont considérés dont l'objectif d'effectuer une évaluation de nos contributions et leurs outils supports. La **Table 6.1** expose en détail une liste de ces programmes. Il s'agit de projets de grandes tailles librement disponibles sur Internet et les autres sont des exemples de programmes (p. ex. "*Introduction*", "*Bean*", et "*Telecom*") distribués avec l'implémentation ASPECTJ et son outil support "AJDT".

Table 6. 1 Une liste détaillée des programmes AspectJ sélectionnés pour l'évaluation.

Nom du Projet	Description	Source	Version (LOC)	#Classes	#Aspects
AspectJ Examples URL : http://www.eclipse.org/aspectj/	Examples of the AspectJ distribution	Eclipse Project	AJ5 (**2 878**)	56	27
AspectJ Design Patterns URL : http://www.cs.ubc.ca/~jan/AODPs/	AspectJ Implementation of the GoF Patterns	University of British Columbia	v1.1 (**2 344**)	104	40
AspectJ Hot Draw URL : http://sourceforge.net/projects/ajhotdraw/	An AspectJ version of the JHotDraw 2D graphics framework	Open Source Project	v0.3 (**22 104**)	294	31
AJHSQLDB URL : http://sourceforge.net/projects/ajhsqldb/	SQL relational database engine	University of Passau	V.18 (**75,556**)	250	31
aTrack URL : https://atrack.dev.java.net/	Bug tracking application		CVSHead (**2 221**)	53	28
Jakarta Cactus URL : http://jakarta.apache.org/cactus/	Test framework for server-side java code		v1.3 (**5 244**)	93	1
Glassbox URL : http://www.glassbox.com/	Troubleshooting agent for Java applications		v1.0a2 (**1 562**)	28	24
GTalkWap URL : http://sourceforge.net/projects/gtalkwap	GoogleTalk access from WAP-enabled devices		v1.0b (**1 013**)	25	2
Infra Red URL : http://sourceforge.net/projects/infrared	Performance monitoring tool for Java/J2EE		v2.3 (**13 888**)	158	11
My SQL Connector J URL : http://www.mysql.com/products/connector	MySQL native Java driver		v5.0 (**40 755**)	149	1
Surrogate URL : http://sourceforge.net/projects/surrogate	Unit testing framework		v1.0 RC1 (**806**)	19	3

6.2 Évaluation du système de visualisation

Dont le but de valider notre première contribution, nous avons mené une évaluation d'envergure du système de visualisation présenté précédemment (c.-à-d. les deux outils supports implémentés : "VizzAspectJ-2D" et "VizzAspectJ-3D") en le testant à travers des études de cas avec plusieurs benchmarks en ASPECTJ sous la plateforme ECLIPSE.

En premier lieu, nous avons évalué "VizzAspectJ-2D", l'outil de visualisation 2D, puis l'outil de visualisation 3D "VizzAspectJ-3D" dans un second lieu. Ces expérimentations ainsi que les résultats obtenus sont discutés dans cette section.

Pour chaque outil, nous illustrons les fonctionnalités offertes pour interagir d'une manière efficace avec les vues de visualisation produites. Nous présentons des captures d'écrans de ces vues, et nous discutons leurs apports particulièrement sur la compréhension des systèmes logiciels de grandes tailles.

Des démonstrations vidéo montrant les principales caractéristiques et l'utilisation du système de visualisation sur quelques études de cas sont également disponibles sur notre site web à l'adresse : http://www.bentrad-sassi.sitew.com/.

6.2.1 Étude de cas sur "VizzAspectJ-2D"

En premier lieu, nous avons expérimenté notre approche de visualisation en 2D. Dans ce qui suit, nous présentons des études de cas effectuées à travers son outil support "VizzAspectJ-2D".

✪ *Visualisation de la hiérarchie d'héritage*

La vue "Complexité du Système" offre une vue de l'arborescence globale des différentes hiérarchies d'héritage des entités logicielles. L'interaction est possible afin d'explorer plus d'informations sur chaque entité visualisée. Nous devons déplacer tout simplement le pointeur de la souris au-dessus ou à proximité du nœud correspond et attendre que l'infobulle d'informations apparaîsse. Une autre possibilité d'interaction permet de montrer les divers liens de dépendances existants pour une ou plusieurs entités. La **Figure 6.1** montre une capture d'écran de cette vue bidimensionnelle.

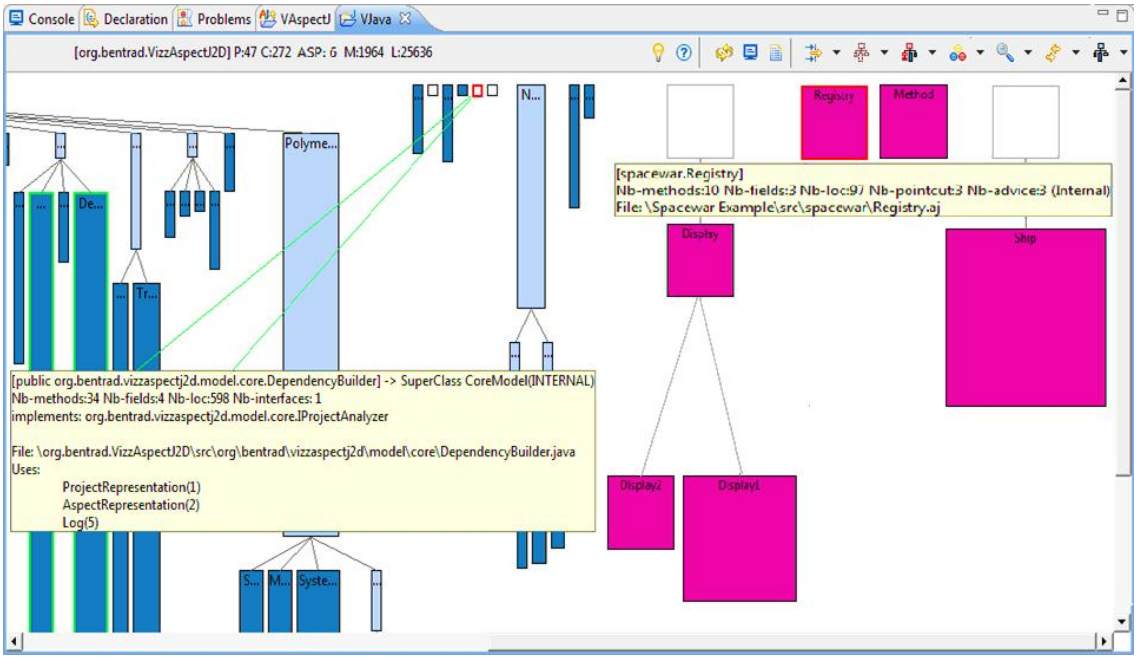


Figure 6. 1 Vue "Complexité du Système" visualisant la hiérarchie d'héritage.

Cette vue montre la hiérarchie d'héritage du code source de l'outil "VizzAspectJ-2D" lui-même. L'analyse effectuée sur la version actuelle donne les informations suivantes : 47 packages, 272 classes, 1964 méthodes et 25 636 lignes de code (LOC).

De plus, l'utilisateur a la possibilité d'interagir avec souplesse pour basculer entre l'entité sélectionnée dans la vue et le code source correspondant au sein de l'éditeur de code, faire modifier ce dernier et voir directement l'impact de ce changement sur la vue (reconstitution de la visualisation) après une opération simple d'actualisation.

La vue "Complexité du Système" fournit une aide précieuse pour montrer une vue globale sur tout le code source d'un système logiciel de n'importe quelle taille en explorant rapidement sa structure éventuellement complexe dès que le nombre de ses entités devient important. De plus, les informations détaillées de chaque entité sont obtenues selon le besoin. Cet outil support reste un prototype, et son implémentation peut être améliorée avec d'autres fonctionnalités et techniques d'interaction avancées.

✪ *Visualisation des dépendances*

Au sein des deux vues "Dépendances des Packages" et "Dépendances des Classes & Aspects", les épaisseurs des liens de dépendances entre entités logicielles avec la métrique couleur nous ont permis de quantifier leurs poids. De plus, ce genre de visualisation met en valeur les anomalies de conception, en fournissant des informations concernant le couplage et la cohésion ainsi que la détermination des entités qui ont beaucoup de responsabilités. La **Figure 6.2** montre deux exemples de cette visualisation bidimensionnelle.

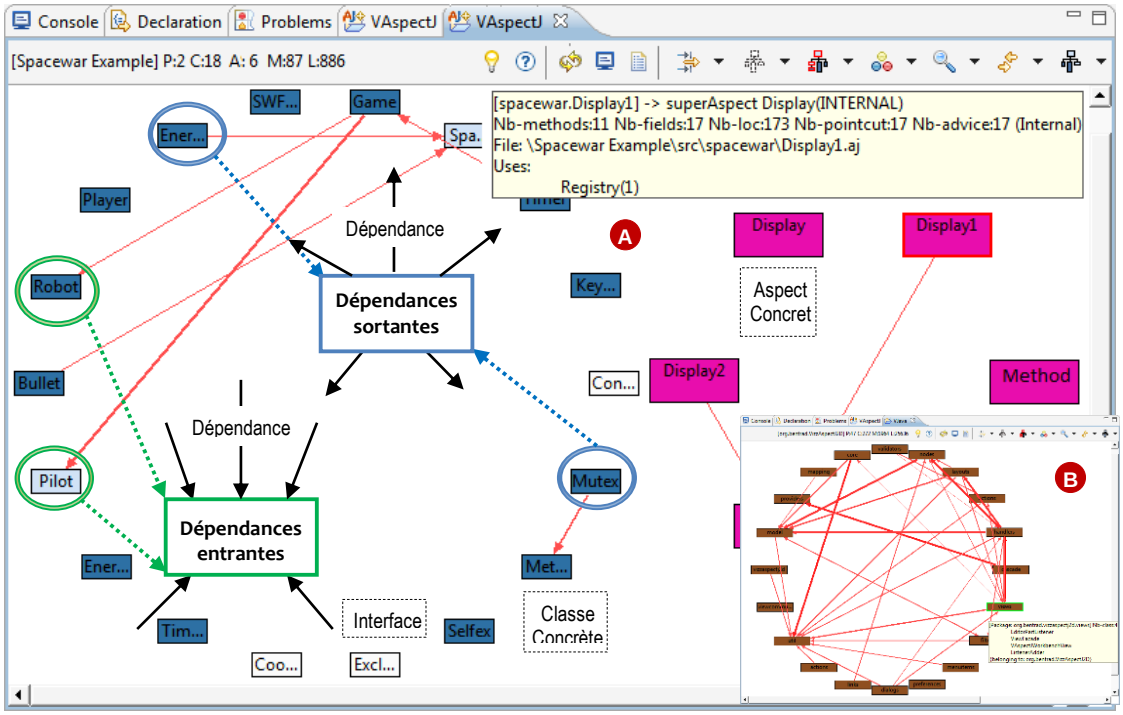
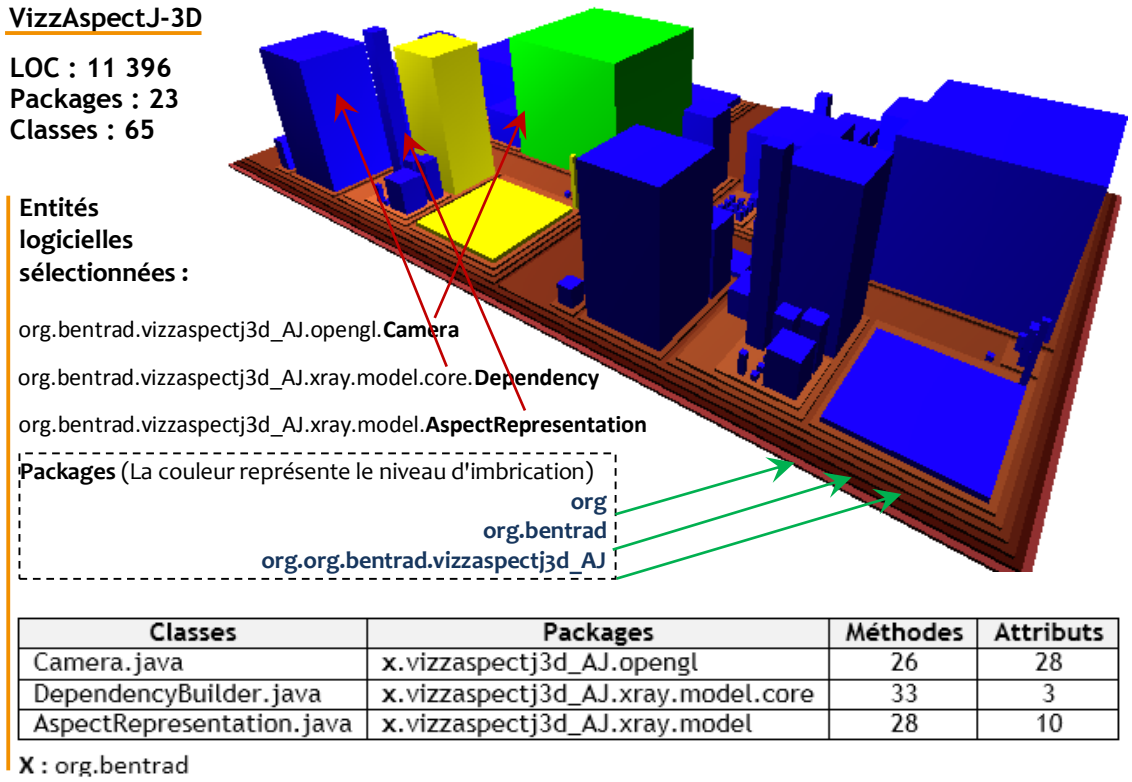


Figure 6. 2 Visualisation des dépendances via "VizzAspectJ-2D":
(A) "Dépendances des Classes & Aspects" et (B) "Dépendances des Packages".

6.2.2 Étude de cas sur "VizzAspectJ-3D"

Dans ce qui suit, nous présentons des études de cas expérimentées de notre approche de visualisation tridimensionnelle. Le principal objectif étant d'évaluer l'utilisabilité (*Usability*) de l'outil support "VizzAspectJ-3D".

En premier lieu, on visualise le code source de "VizzAspectJ-3D" lui-même. La **Figure 6.3** montre une vue en 3D de la version actuelle ce prototype.



Classes	Packages	Méthodes	Attributs
Camera.java	x.vizzaspectj3d_AJ.opengl	26	28
DependencyBuilder.java	x.vizzaspectj3d_AJ.xray.model.core	33	3
AspectRepresentation.java	x.vizzaspectj3d_AJ.xray.model	28	10

X : org.bentrad

Figure 6. 3 Visualisation tridimensionnelle (3D) du code source de l'outil "VizzAspectJ-3D".

Nous avons présenté les entités (classes, interfaces et aspects) sur le niveau de leur parent, et les paquetages sont de tailles décroissantes de l'extérieur vers l'intérieur. Les possibilités d'interaction sont toujours offertes. L'utilisateur a la possibilité d'interagir avec souplesse (le survole sur l'espace 3D est à l'aide d'une caméra dans les quatre directions) pour faire basculer entre l'entité sélectionnée dans la vue et le code source correspondant au sein de l'éditeur de code ; faire modifier ce dernier et voir directement l'impact de ce changement sur la vue (reconstitution de la visualisation) après une opération simple d'actualisation.

Afin d'évaluer et prouver la montée à l'échelle (*Scalability*) de l'outil support, nous avons sélectionné des programmes de grandes tailles tels que : "AJHotDraw" (v 1.5 ; LOC: 22 104 ; Packages : 72; Classes : 294 ; Aspects : 31) et "JDK" (v 1.6 ; LOC: 160 000, Packages : 137; Classes : 4 715), etc.

La **Figure 6.4** montre une visualisation tridimensionnelle du code source du projet "AJHotDraw 1.5".

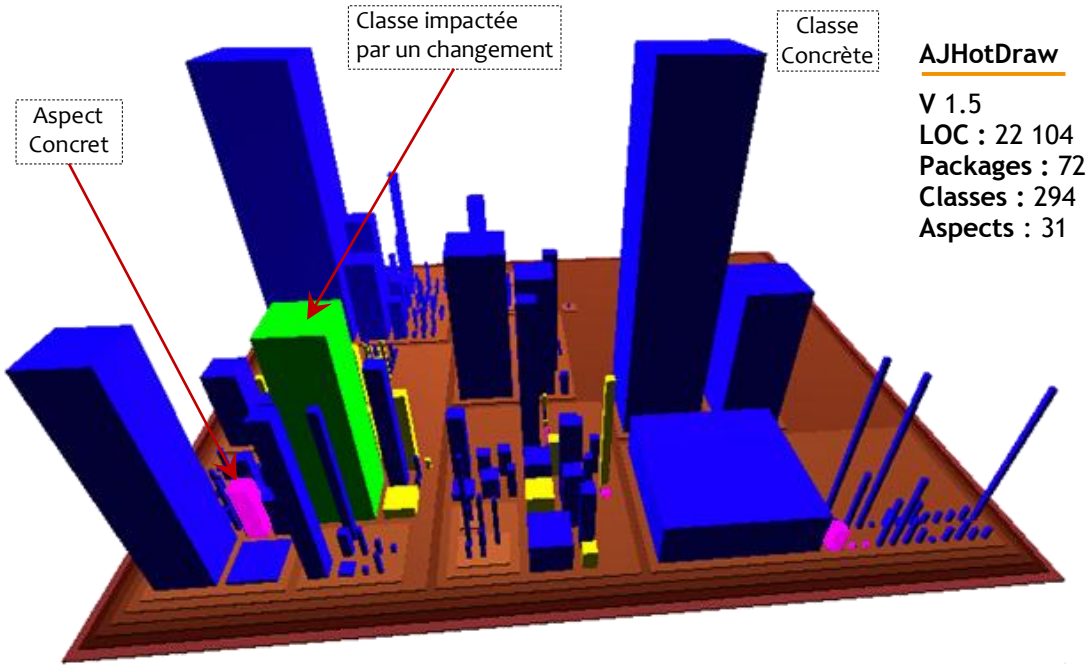


Figure 6. 4 Vue de ville en 3D du projet "AJHotDraw".

La **Figure 6.5** montre des captures d'écran d'une visualisation tridimensionnelle du code source du projet "JDK 1.6" ; les deux vues (A) et (B) représentent deux dispositions de la même visualisation.

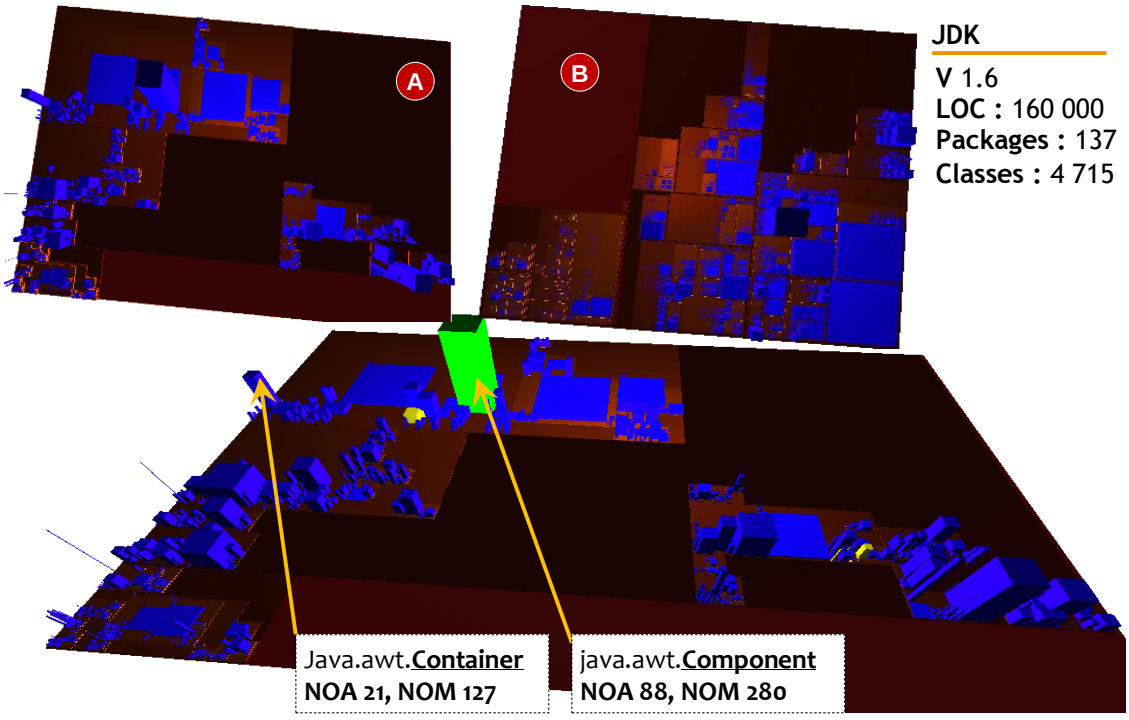


Figure 6. 5 Vue de ville en 3D du projet "JDK".

6.2.3 Discussion

Nous avons fait une évaluation de notre système de visualisation en testant les deux outils supports "VizzAspectJ-2D" et "VizzAspectJ-3D" avec des programmes ASPECTJ. Les résultats obtenus par l'analyse exploratoire de ces études de cas nous ont permis de faire certaines remarques intéressantes et de vérifier que celui-ci s'adapte bien à des programmes ASPECTJ de différentes tailles.

Il est évident que la capacité d'afficher l'ensemble d'entités et des liens simultanément est l'une des fonctionnalités les plus importantes que notre visualisation offre. Des infobulles sont affichées en passant tout simplement le pointeur de la souris au-dessus des liens indiquant la direction de la dépendance (en s'assurant de faire ressortir clairement le point de départ et d'arrivée) et même ressortir des informations sur leurs poids et les composants de chaque entité.

De plus, afin de réduire la complexité des vues et de diminuer l'encombrement visuel causé par l'affichage simultané de plusieurs types de liens, les deux outils offrent une fonctionnalité puissante de filtrage, et ce tant au niveau de la hiérarchie des entités que des diverses dépendances. Aussi, la méthode de navigation à l'aide d'une caméra permet à un utilisateur de bouger librement dans un espace 3D au sein d'une masse d'information.

Ce genre de mécanismes a pour but d'éviter de compliquer et troubler la compréhension, ce qui permet à l'utilisateur de se concentrer sur l'information pouvant l'aider dans ses tâches de codage et de compréhension.

D'autre part, les expérimentations confirment que la détection des anomalies de conception lorsqu'elle est faite après une inspection humaine assistée par ce genre de vues est bien plus rapide et plus efficace que d'inspecter le code source manuellement [Dha 07].

Finalement, on peut signaler qu'aucun outil de visualisation n'est capable de réaliser toutes les tâches de compréhension. C'est le niveau d'abstraction des vues produites qui affecte grandement le succès des tâches de compréhension.

6.3 Évaluation de l'approche hybride de codage "HM4AOP"

Cette section est dédiée à l'évaluation et la validation de notre approche de codage orienté-aspects (HM4AOP) vue dans le chapitre précédent.

Dans un premier temps, afin de prouver la faisabilité (*Feasibility*) de notre proposition, nous avons effectué des études de cas préliminaires sur son outil support "HCodelessAJ".

Dans un second temps, nous voulons aussi évaluer et étudier empiriquement l'efficacité (*Effectiveness*) de cette approche. Pour cela, nous ferons en parallèle une étude comparative avec "AJDT", l'outil support conventionnel de programmation orientée-aspects en ASPECTJ sous la plateforme ECLIPSE.

6.3.1 Études de cas préliminaires

En premier lieu, nous cherchons à prouver la faisabilité de l'approche proposée à travers des études de cas en utilisant l'outil prototype proposé pour les programmes ASPECTJ, en même temps nous voulons aussi expérimenter ses fonctionnalités.

Plusieurs programmes benchmarks distribués avec ASPECTJ sont considérés pour cet objectif. Nous présentons ici quelques études de cas sur des programmes simples ("*Introduction*", "*Bean*", "*Telecom*" et "*Fibonacci*") tirés de la distribution académique de l'outil support "AJDT" sous la plateforme ECLIPSE, illustrant la manière d'utiliser ce prototype.

Des démonstrations vidéo, montrant les principales caractéristiques et fonctionnalités ainsi que l'utilisation de la version actuelle du prototype "HCodelessAJ" en quelques expérimentations, sont également disponibles sur notre site web à l'adresse : http://www.bentrad-sassi.sitew.com/.

6.3.1.1 Première étude de cas

Le petit projet "Telecom" simule une communication téléphonique et gère les communications locales et de longue distance entre les clients, ainsi que les conversations.

Dans un premier temps, un "prototypage rapide de code" (en anglais : *Rapid Code Prototyping*) donne les classes essentielles. Nous complétons le diagramme de classes UML en montrant comment les aspects interviennent dans le code métier de l'application : quels sont les consignes, les déclarations inter-type de membres, etc.

Le diagramme de classes UML est montré dans la **Figure 6.6**. Le code source de ce projet est constitué des six classes : *Call*, *Timer*, *Customer*, *Connection*, *Local* et *LongDistance*.

- *Call* : soutenir le processus d'un client tentant de se connecter à d'autres.
- *Timer* : sert à enregistrer le temps écoulé.

- *Customer* : Le client a un identificateur unique (son nom dans ce cas pour des fins didactiques, mais il pourrait être le numéro de téléphone) et l'indicatif de la région. Ils ont aussi un protocole de gestion des appels.
- *Connection* : les communications sont des circuits établis entre les clients. Il existe deux types : une communication locale et communication longue distance (sous-classes).

Nous ajoutons à ce digramme les trois aspects suivants :

- *Timerlong Aspect* : se préoccupe du temps de communication enregistrée.
- *Timing Aspect* : se préoccupe par la durée de communication et avec le temps de communication cumulé du client.
- *Billing Aspect* : générer une facture en fonction du type de l'appel et la durée de communication correspondante.

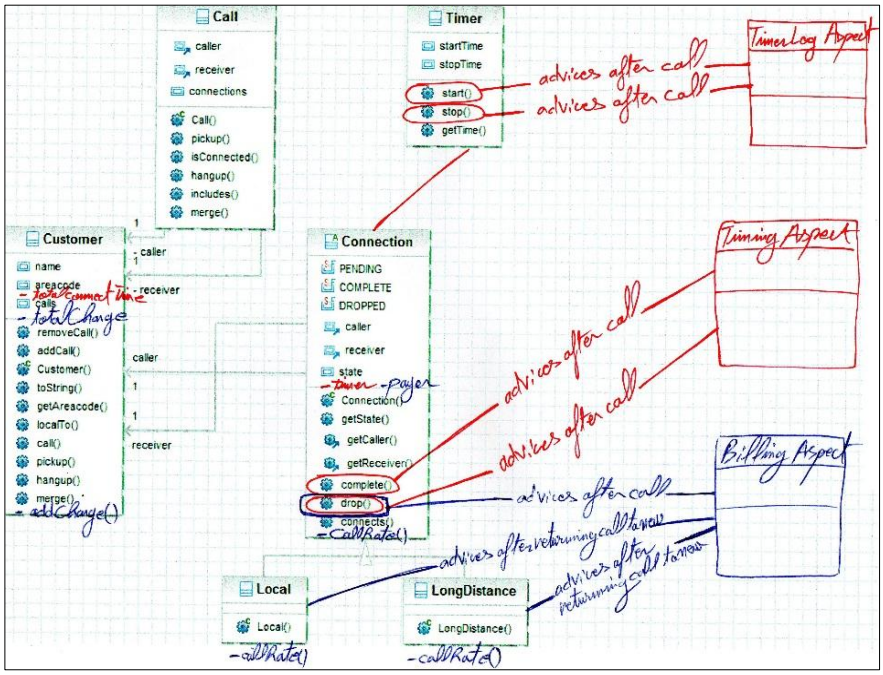


Figure 6. 6 Prototypage des aspects candidats sur le diagramme de classe UML.

Puis, c'est "l'édition interactive de code" (en anglais : *Interactive Code Editing*), on exprime d'une manière graphique la structure du code et avec quelques portions du comportement du programme cible en utilisant l'éditeur visuel (VMD).

Les deux **Figure 6.7** et **6.8** montrent, respectivement, un aperçu du modèle visuel créé et sauvegardé dans un fichier XMI (.vaj).

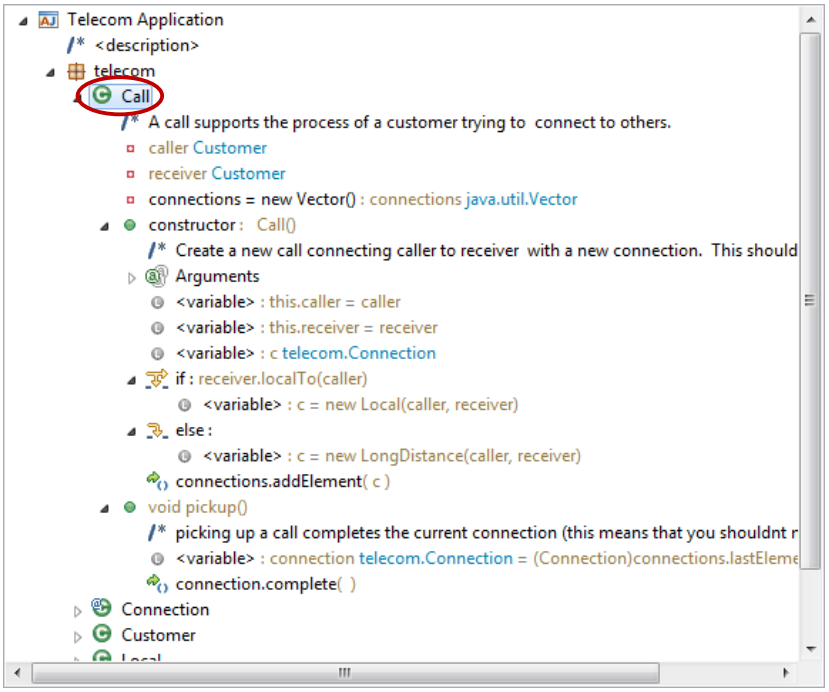


Figure 6. 7 Un aperçu du modèle visuel du programme cible "Telecom".

```
<?xml version="1.0" encoding="UTF-8"?>
<hcpm4ao:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xmlns:hcpm4ao="http:///org/bentrad/hcpm4ao/model.ecore"
release="1.1.0" encoding="UTF-8" author="BENTRAD" updatedBy="BENTRAD" created=
"2014-09-24T17:13:05.536+0100" updated="2014-09-29T22:47:14.411+0100">
  <programs name="Telecom Application">
    <description/>
    <packages name="telecom">
      <classes name="Call" annotation="">
        <description text="A call supports the process of a customer trying to &#xD;&#xA;connect to
others.&#xD;&#xA; "/>
        <members access="PRIVATE" name="caller">
          <type name="Customer"/>
        </members>
        <members access="PRIVATE" name="receiver">
        <members note="connections = new Vector()" access="PRIVATE" annotation="" name="connections">
        <methods note="constructor" annotation="">
          <description text="Create a new call connecting caller to receiver &#xD;&#xA;with a new
connection. &#xD;&#xA;This should really only be called by Customer.call(..)"/>
          <sequences xsi:type="hcpm4ao:Variable" name="this.caller" value="caller"/>
          <sequences xsi:type="hcpm4ao:Variable" name="this.receiver" value="receiver"/>
          <sequences xsi:type="hcpm4ao:Variable" name="c" value="">
          <sequences xsi:type="hcpm4ao:ControlSequence">
          <sequences xsi:type="hcpm4ao:MethodCall" name="connections.addElement" arguments="c"/>
          <signature name="Call"/>
          <arguments name="caller">
          <arguments name="receiver">
        </methods>
        <methods>
      </classes>
      <classes name="Connection" annotation="abstract class">
      <classes name="Customer">
      <classes name="Local">
      <classes name="LocalDistance">
      <classes name="Timer">
    </packages>
  </programs>
</hcpm4ao:Model>
```

Figure 6. 8 Un extrait au format XMI illustrant le modèle visuel du programme cible "Telecom".

Le **Listing 6.1** montre un extrait du résultat de mapping du modèle visuel (présenté dans la **Figure 6.7**) en des templates textuelles de squelette du code selon le langage d'implémentation cible (p. ex. ASPECTJ).

```
1 package telecom;
2 import telecom.Connection;
3 import java.util.Vector;
4 import telecom.Customer;
5 /**
6  * A call supports the process of a customer trying to connect to others.
7  */
8 Public class Call{
9 private Customer caller;
10 private Customer receiver;
11 /**
12  * NOTE: connections = new Vector()
13  */
14 private Vector connections;
15 /**
16     * NOTE: constructor
17     * Create a new call connecting caller to receiver with a new connection.
18     * This should really only be called by Customer.call(..)
19     */
20 Public void Call(Customer caller, Customer... receiver)
21 {
22     /*type*/ this.caller = caller;
23     /*type*/ this.receiver = receiver;
24     Connection c = null;
25 if (receiver.localTo(caller)) {
26     /*type*/ c = new Local(caller, receiver);
27     } else {
28     /*type*/ c = new LongDistance(caller, receiver);
29     }
30 connections.addElement(c);
31 }
32 /**
33   * picking up a call completes the current
34   * connection (this means that you shouldnt merge
35   * calls until they are completed)
36   */
37 Public void pickup() {
38     Connection connection =
39     (Connection) connections.lastElement();
40 connection.complete();
41 }

                                    ...

1 package telecom;
2 public aspect TimerLog {
3
4   after(Timer t): target(t) && call(* Timer.start())  {
5      System.err.println("Timer started: " + t.startTime);
6   }
7
8   after(Timer t): target(t) && call(* Timer.stop()) {
9      System.err.println("Timer stopped: " + t.stopTime);
10  }
11 }
```

Listing 6. 1 Un extrait des templates de code AspectJ générées du programme cible "Telecom".

6.3.1.2 Deuxième étude de cas

Pour cette étude de cas, une implémentation récursive déterminant le Nième nombre de "Fibonacci" est donnée comme un deuxième exemple illustratif.

L'aspect "*FibonacciAspect.aj* " est utilisée pour déterminer le nombre des appels de la méthode *fib()* lors de l'exécution de la méthode *fibstart()*, lorsque ce dernier se produit dans le flux de contrôle d'un appel à la méthode *m2()*. L'aspect initialise un compteur (ctr) à 0 au début de l'exécution de la méthode *fibstart()*, incrémente ce compteur par 1 au début de chaque exécution de la méthode *fib()*, et imprime sa valeur après exécution de la méthode *fibstart()*.

La **Figure 6.9** montre un aperçu du modèle visuel créé sous l'éditeur visuel (VMD) pour l'implémentation cible de son code source.

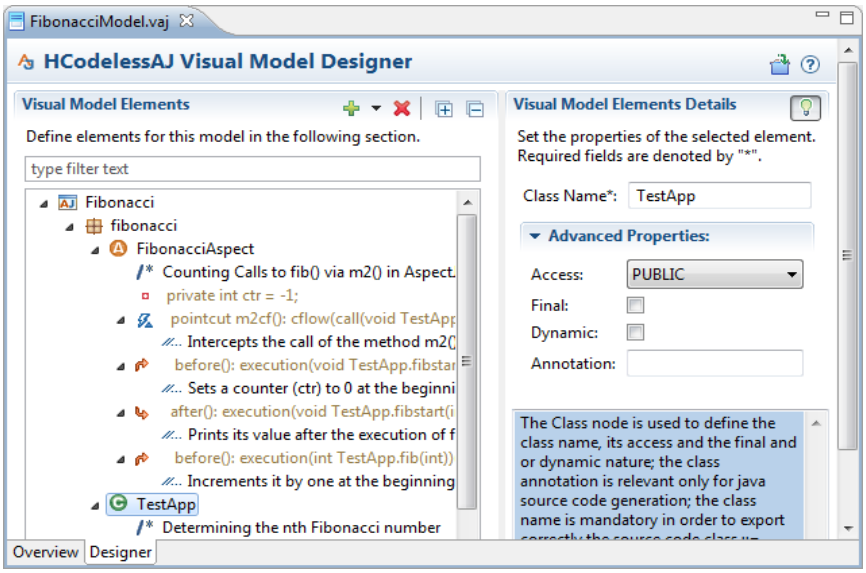
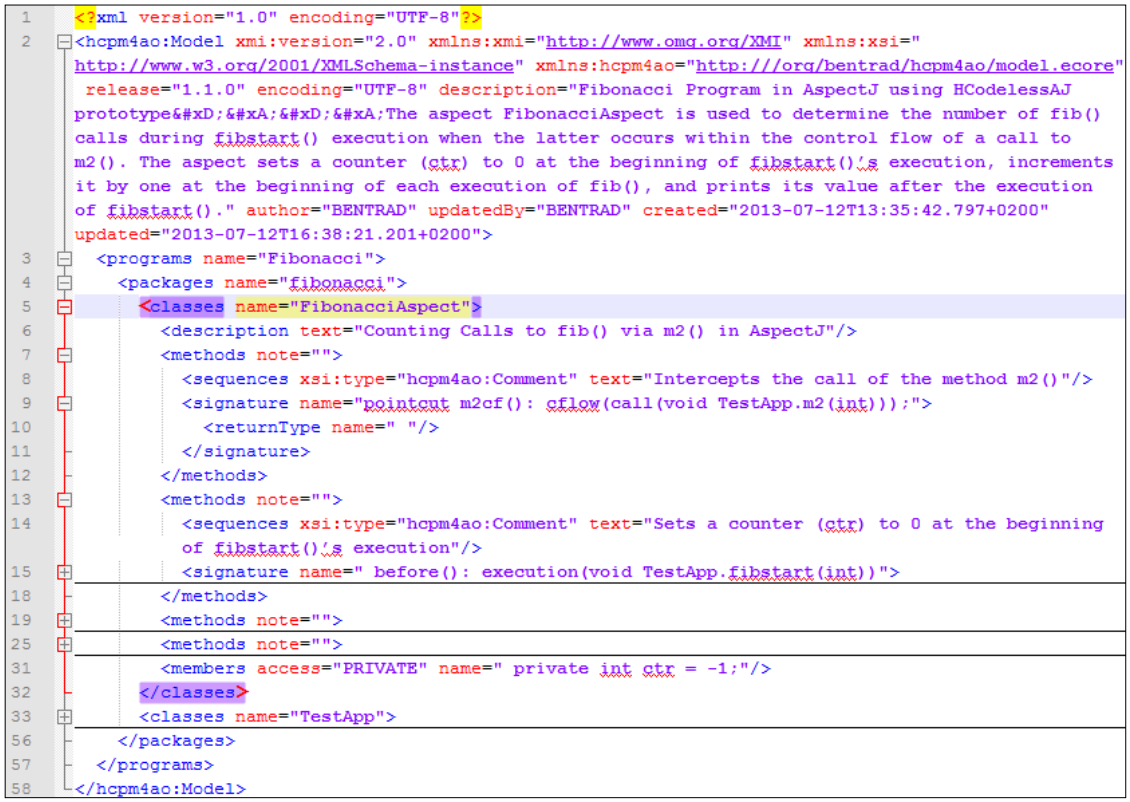


Figure 6. 9 Un aperçu du modèle visuel du programme cible "Fibonacci".



```
<?xml version="1.0" encoding="UTF-8"?>
<hcpm4ao:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance" xmlns:hcpm4ao="http:///org/bentrad/hcpm4ao/model.ecore"
 release="1.1.0" encoding="UTF-8" description="Fibonacci Program in AspectJ using HCodelessAJ
prototype&#xD;&#xA;&#xD;&#xA;The aspect FibonacciAspect is used to determine the number of fib()
calls during fibstart() execution when the latter occurs within the control flow of a call to
m2(). The aspect sets a counter (ctr) to 0 at the beginning of fibstart()'s execution, increments
it by one at the beginning of each execution of fib(), and prints its value after the execution
of fibstart()." author="BENTRAD" updatedBy="BENTRAD" created="2013-07-12T13:35:42.797+0200"
updated="2013-07-12T16:38:21.201+0200">
  <programs name="Fibonacci">
    <packages name="fibonacci">
      <classes name="FibonacciAspect">
        <description text="Counting Calls to fib() via m2() in AspectJ"/>
        <methods note="">
          <sequences xsi:type="hcpm4ao:Comment" text="Intercepts the call of the method m2()"/>
          <signature name="pointcut m2cf(): cflow(call(void TestApp.m2(int)));">
            <returnType name=" "/>
          </signature>
        </methods>
        <methods note="">
          <sequences xsi:type="hcpm4ao:Comment" text="Sets a counter (ctr) to 0 at the beginning
          of fibstart()'s execution"/>
          <signature name=" before(): execution(void TestApp.fibstart(int))">
        </methods>
        <methods note="">
        <methods note="">
          <members access="PRIVATE" name=" private int ctr = -1;"/>
      </classes>
      <classes name="TestApp">
    </packages>
  </programs>
</hcpm4ao:Model>
```

Figure 6. 10 Un extrait au format XMI illustrant le modèle visuel du programme cible "Fibonacci".

Le **Listing 6.2** montre le résultat de mapping du modèle visuel en templates de squelette du code source ("*FibonacciAspect.aj* " et "*TestApp.java* ") du programme cible.

```
1  package fibonacci;
2  //Counting Calls to fib() via m2() in AspectJ
3  public aspect FibonacciAspect {
4   private int ctr = -1;
5   //Intercepts the call of the method m2()
6   pointcut m2cf(): cflow(call(void TestApp.m2(int)));
7   //Sets a counter (ctr) to 0 at the beginning of fibstart()'s execution
8   before(): execution(void TestApp.fibstart(int))
9         && m2cf() { ctr = 0; }
10  //Prints its value after the execution of fibstart()
11  after(): execution(void TestApp.fibstart(int))
12        && m2cf() { System.out.println(ctr); }
13  //Increments it by one at the beginning of each execution of fib()
14  before(): execution(int TestApp.fib(int))
15        && m2cf() { ctr++; }
16 }
```

```
1  package fibonacci;
2  //Determining the nth Fibonacci number
3  public class TestApp {
4   public void m1(int n)
5                       { fibstart(n); }
6   public void m2(int n)
7                       { fibstart(n); }
8   public void fibstart(int n)
9                       { fib(n); } }
10 public int fib(int k) {
11        return (k > 1) ? fib(k-1)+fib(k-2) : k;
12 }
                                    ...
```

Listing 6. 2 Un extrait des templates de code AspectJ générées du programme cible "Fibonacci".

6.3.1.3 Discussion

Kiczales [Kic 97] a mentionné ceci : "It is extremely difficult to quantify the benefits of using AOP without a large experimental study, involving multiple programmers using both AOP and traditional techniques to develop and maintain different applications". Que nous traduisons par : c'est extrêmement difficile de quantifier les avantages du paradigme "orienté-aspects" (POA) sans une large étude expérimentale, impliquant plusieurs programmeurs utilisant à la fois la POA et les techniques traditionnelles pour développer et maintenir des applications différentes".

Étant donné que ces études de cas sont à but exploratoire, et servent à illustrer et évaluer l'efficacité de notre proposition ; l'analyse est principalement qualitative.

Bien que l'outil support implémenté soit dans sa première version et est encore en phase expérimentale, les essais préliminaires démontrent la faisabilité et l'efficacité de l'approche proposée et son idée innovatrice de codage. Au sein de l'éditeur visuel (VMD), la nouvelle représentation du code source rend explicite ses entités avec leurs liens de dépendances telles

que l'héritage, les hiérarchies de packages, de classes et de préoccupations transversales, etc. qui sont implicites et difficiles à localiser à travers les descriptions textuelles au sein des éditeurs textuels classiques. Ceci permet d'avoir rapidement une vue globale sur le squelette du code et, par conséquent, facilite la compréhension, la maintenance et l'éxtension ultérieures.

Le taux de génération est près de 70 % du code source de chaque projet. En outre, les templates générés sont de haute qualité par apport au code source original, ce qui donne un pas considérable pour améliorer les principaux facteurs de qualité tels que l'extensibilité, la maintenabilité et la réutilisabilité.

Le mécanisme de génération est automatisé grâce à la technique de transformation ACCELEO qui donne la possibilité de générer les spécifications du comportement, la réduction des coûts et l'exactitude des templates générées.

ACCELEO est un langage de génération de code basé modèle qui n'a pas de restrictions sur le type du code ; il n'y a qu'une seule règle : "*If you can write it, ACCELEO can generate it* ". Que nous traduisons par : si vous pouvez l'écrire, ACCELEO peut le générer. De cette façon, notre approche pourrait être adaptée pour fonctionner avec diverses implémentations du paradigme "orienté-aspects". D'un autre point de vue, pour les diverses tâches de maintenance, les développeurs devront mettre à jour leurs modèles visuels déjà construits, réutiliser des codes existants, et régénérer de nouveaux templates de haute qualité.

Cette contribution est considérée comme un premier pas vers une nouvelle génération d'outils supports pour un codage orienté-aspects. Il y a d'autres extensions possibles, nous les décrivons par la suite comme des perspectives de recherche et travaux futurs.

6.3.2 Expérimentation détaillée

Cette section est dédiée à une étude expérimentale détaillée du prototype "HCodelessAJ" sur plusieurs programmes ASPECTJ. Nous décrivons les différentes étapes de l'expérimentation ainsi que les résultats obtenus. Dans un premier temps, nous effectuons une évaluation quantitative, puis une autre qualitative. Avec cette étude empirique, nous avons mené une comparaison entre ce prototype et "AJDT". Puis, nous voulons voir aussi les observations et les commentaires de chaque participant à travers un sondage basé sur un questionnaire et un interview, afin d'en fournir une étude critique. À la fin, une analyse et discussion générale est donnée.

6.3.2.1 Description

Nous montrons un aperçu sur les deux évaluations qualitative et quantitative que nous avons effectuées avec un groupe d'étudiants afin d'évaluer l'efficacité de l'approche

proposée. En outre, les deux évaluations sont consacrées à faire une étude comparative impartiale avec l'outil support de l'implémentation référence d'ASPECTJ sous la plateforme ECLIPSE, "AJDT".

✪ *Outils supports*

- **AspectJ Development Tools (AJDT)**. C'est l'outil support le plus mature pour la programmation orientée-aspects en ASPECTJ. "AJDT" s'intègre étroitement avec la plateforme ECLIPSE et propose diverses installations pour comprendre les interactions entre les aspects et le code métier. Site officiel du projet : http://www.eclipse.org/ajdt/.

- **HCodelessAJ**. Un prototype de l'outil support proposé pour programmer en ASPECTJ sous la plateforme ECLIPSE.

✪ *Programmes sujets d'étude (subject programs)*

Nous avons sélectionné quatre programmes ASPECTJ de taille raisonnable. La **Table 6.3** résume quelques informations sur ces programmes (nom du programme, taille (LOC, lines of code), nombre de classes et nombre d'aspects).

Table 6. 2 Les programmes AspectJ sélectionnés pour l'expérimentation.

Program	LOC	# Classes	# Aspects	Description
Bean	176	2	1	ASPECTJ Benchmark Examples from AJDT Distribution for Eclipse IDE
Introduction	247	1	3	
Observer	184	6	2	
TJP	97	1	1	

✪ *Participants*

Le groupe de participants à l'expérimentation est constitue de seize (16) étudiants de Master 2 en "Ingénierie des Logiciels Complexes" à l'Université de Badji Mokhtar-Annaba (UBMA, Algérie). Ces étudiants ayant des bases convenables en programmation orientée-aspects en général et en langage ASPECTJ en particulier (ont suivi et réussi le cours "Séparation Avancée des Préoccupations"). De plus, ils sont familiarisés avec "AJDT", l'outil support conventionnel d'ASPECTJ sous la plateforme ECLIPSE.

Afin de réaliser une analyse empirique et comparative équitable entre le prototype "HCodelessAJ" et "AJDT" l'étude expérimentale a été effectuée en trois étapes, et la satisfaction des participants a été également examinée.

Tout d'abord, les participants ont reçu une description courte et formelle sur l'étude avec des démonstrations illustrant les deux outils supports : "HCodelessAJ" et "AJDT". Après, chaque participant devient normalement prêt à faire ses premiers pas en programmation orientée-aspects.

Dans la deuxième étape (Pretest : *Controlled Experiment*), les participants ont été répartis aléatoirement en deux groupes équilibrés. Chaque participant devait taper le code source de certains programmes sélectionnés en utilisant séparément ces deux outils. Un groupe a commencé avec "AJDT", tandis que l'autre groupe devait utiliser le prototype "HCodelessAJ". Durant l'étude, avec chaque participant et pour chaque programme, des données ont été notées par rapport à certains critères.

Dans la troisième étape (Posttest : *Questionnaire-Based Survey*) et afin de recueillir des informations sur la satisfaction, les opinions et les commentaires de chaque groupe, chaque participant devait remplir un questionnaire.

Durant l'expérimentation, on n'a fourni aucune aide aux participants pour ne pas biaiser les résultats. On prend seulement des commentaires et on note les difficultés rencontrées. Plus de détails sur cette évaluation figurent dans l'**Annexe B**.

6.3.2.2 Évaluation empirique et analyse (Pretest : controlled experiment)

Dans cette section, nous décrivons et discutons les critères retenus pour l'évaluation, le récapitulatif des tableaux d'évaluation qui montre les résultats obtenus en plus de ses représentations graphiques.

✪ *Evaluation Quantitative*

Les participants ont été invités à comparer les deux outils selon les critères suivants :

- **Durée d'apprentissage** (en anglais ***Training time : in day***) : la durée nécessaire pour se familiariser avec la programmation ASPECTJ selon une seule méthodologie de codage et à l'aide de l'outil support correspondant.
- **Durée de développement** (en anglais ***Development time : in minutes***) : le temps requis pour effectuer la tâche de codage pour chaque programme.
- **Durée de compilation** (en anglais ***Compilation time : in seconds***) : le temps écoulé pour compiler le programme créé.
- **Pourcentage d'erreurs** (en anglais ***Miss-typing percentage : %***) : la proportion d'erreurs de frappe et d'erreurs syntaxiques du code source.

Un résumé des tables d'évaluation montrant les résultats obtenus est donné à la **Table 6.3**.

Table 6. 3 Récapitulatif des résultats de comparaison quantitative : "HCodelessAJ" vs. "AJDT".

Test Programs	AJDT				HCodelessAJ			
	T1	T2	T3	P	T1	T2	T3	P
Bean	13	51m 0s	3s, 16ms	20 %	8	33m 0s	3s, 16ms	9 %
Introduction		46m 0s	0s, 39ms	19 %		30m 0s	0s, 39ms	7 %
Observer		35m 0s	4s, 40ms	17 %		23m 0s	4s, 40ms	6 %
TJP		15m 0s	0s, 12ms	15 %		8m 0s	0s, 12ms	5 %
Average	**13**	**36m 45s**	**2s, 12ms**	**18 %**	**8**	**23m 30s**	**2s, 12ms**	**7 %**

Legend:
T1: Training time (in day)
T2: Development time (in minutes, m)
T3: Compilation time (in seconds, ms)
P: Miss-typing percentage (%)
NB: In this table, we show the average of results obtained from all participants.

✪ *Evaluation Qualitative*

En plus de l'évaluation quantitative, on mesure les performances et la contribution du prototype par rapport à certains critères (indices de performance). Les indices les plus importants suggérés sont les suivants :

- **Expressivité** (en anglais : ***Expressiveness***) : les restrictions possibles imposées à l'étape de codage.
- **Utilisabilité** (en anglais : ***Usability***) : définie comme la mesure avec laquelle un outil peut être utilisé pour atteindre les objectifs dans un contexte d'utilisation spécifique, avec un minimum de prérequis en informations techniques et de temps pour accomplir une tâche particulière (easy to use and to program). Elle est définie selon *Nielsen* avec cinq aspects principaux : "*Efficiency*", "*Learnability*", "*Memorability*", "*Error Handling*", et "*User Satisfaction*".
- **Performance de calcul** (en anglais : ***Computational Performance***) : le temps nécessaire pour la compilation et l'exécution.
- **Productivité** (en anglais : ***Productivity***) : l'effort requis pour développer et déployer les programmes. Elle est définie avec cinq aspects principaux : "*Flexibility*", "*Scalability*", "*Reusability*", "*Comprehensibility*" et "*Maintainability*".

✪ *Analyse et interprétation des résultats*

Dans cette section, nous discutons l'étude comparative de ces deux outils en suivant les critères déjà sélectionnés. Le résumé des résultats d'évaluation obtenus est montré

graphiquement via des diagrammes en bâtons (en anglais : *Histogram*) et boîtes à moustaches (en anglais : *Box plot*) à la **Figure 6.11** et **6.12** respectivement.

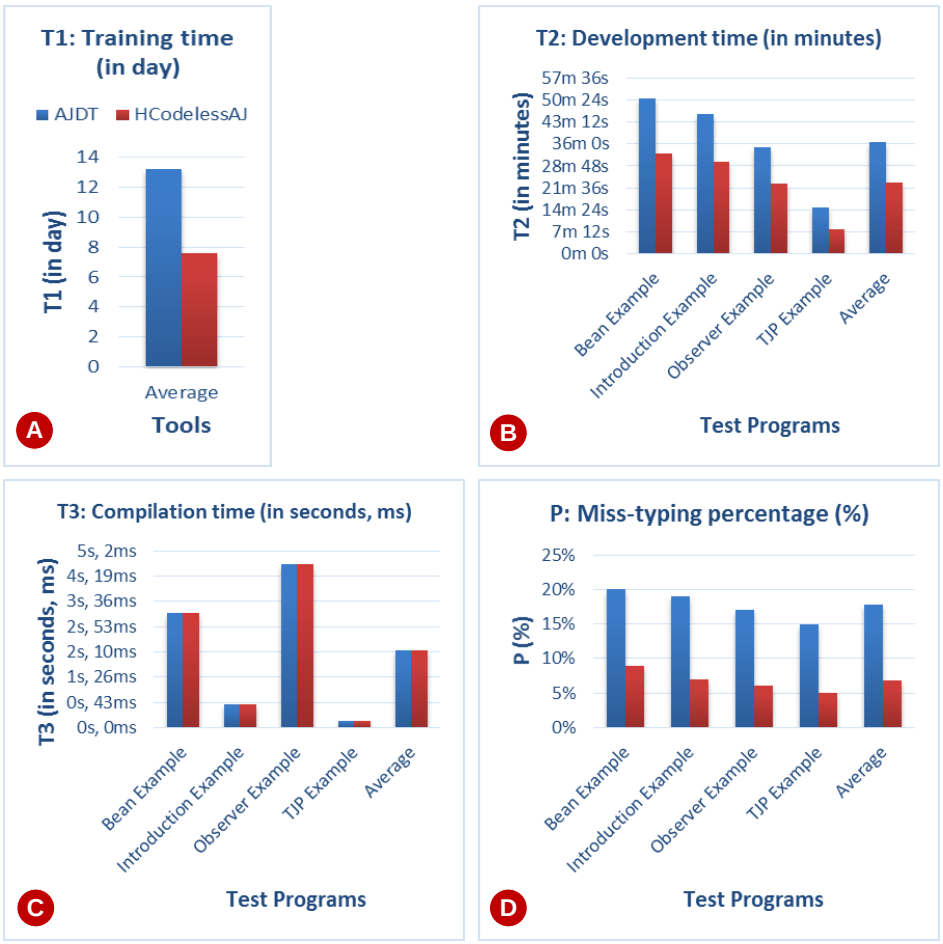


Figure 6. 11 Vue des représentations graphiques résumant les résultats de l'étude comparative quantitative : "HCodelessAJ" vs. "AJDT".

Les notations attribuées aux énoncés (indices de performance) sont entre zéro point pour "nothing recognizable at all" et quatre points pour "right statement". Les résultats de l'évaluation qualitative sont récapitulés dans la **Table 6.4**.

Table 6. 4 Récapitulatif des résultats de comparaison qualitative : "HCodelessAJ" vs. "AJDT".

Aspects Considered / AspectJ Tool Support	**Expressiveness**	**Usability** Efficiency, Learnability, Memorability, Error Handling, User Satisfaction	**Computational Performance**	**Productivity** Flexibility, Scalability, Reusability, Comprehensibility, Maintainability
AJDT	**2** / *(4 points)*	**2** / *(4 points)*	**4** / *(4 points)*	**2** / *(4 points)*
HCodelessAJ	**3,5**/*(4 points)*	**3** / *(4 points)*	**4** / *(4 points)*	**3,5** / *(4 points)*

Nous avons adopté les boîtes à moustaches afin de résumer et représenter graphiquement les données de réponses en utilisant les valeurs minimales et maximales des points qui ont été attribués par les participants aux indices de performances pour les deux outils.

Ce diagramme en boîte est un moyen rapide, utilisé principalement dans l'analyse exploratoire d'une série statistique de données, pour comparer un même indice dans deux populations de tailles différentes.

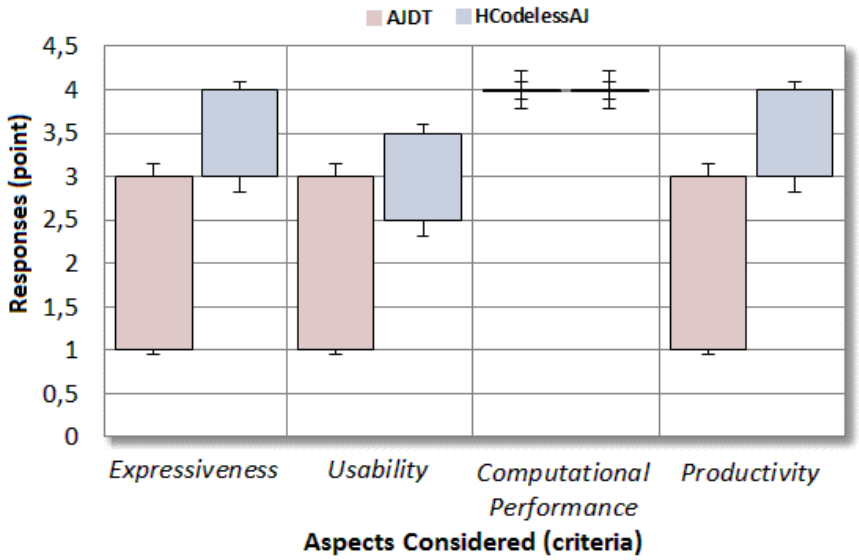


Figure 6. 12 Vue graphique résumant les résultats de l'étude comparative qualitative : "HCodelessAJ" vs. "AJDT".

Comme montrée dans le diagramme comparatif, l'analyse qualitative révèle qu'il y avait une différence significative entre les deux outils sur les indices : "*Expressiveness*", "*Usability*" et "*Productivity*".

Par exemple, la moyenne des notations attribuées à l'indice "*Expressiveness*" est 2 points pour "AJDT" et 3,5 points pour le prototype "HCodelessAJ" ; (AJDT = 2 points, HCodelessAJ = 3 points) pour l'indice "*Usability*" et (AJDT = 2 points, HCodelessAJ = 3,5 points) pour l'indice " *Productivity*".

✪ *Discussion*

Pour une évaluation empirique, les participants devaient évaluer les deux outils selon les critères définis précédemment. Nous discuterons ici comment le prototype "HCodelessAJ" satisfait ces critères en mettant l'accent principalement sur la comparaison avec "AJDT".

- **Formation de courte durée.** Tous les participants ont constaté que le degré de facilité d'utilisation (ease to use) est acceptable. Dès le début à l'aide de certaines interactions on arrive à construire un modèle visuel pour le programme cible et puis ajout du code

requis sur les templates générés où la plupart des détails d'implémentation (certains formalismes syntaxiques) ont été manipulés avec transparence et visibilité immédiate. Cette dernière caractéristique, offerte par ce prototype, apparaît très utile. En outre d'une manière quantitative, il y a une réduction de 30 à 40 % de temps par rapport à "AJDT" et elle sera probablement plus élevée au futur.

- **Moins de temps et effort de développement.** La technique "*Drag-and-Drop*", adoptée au sein du prototype offre la possibilité de coder avec un haut niveau d'interactivité et souplesse. Tous les participants ont ressenti la réduction significative du temps et d'effort en codage. En outre, les erreurs syntaxiques sont moins fréquentes, en raison de la réduction du typage des formalismes syntaxiques. Cependant, pour un programmeur professionnel qui est déjà familiarisé avec l'approche classique, probablement autant de temps sera requis pour s'adapter avec la nouvelle méthodologie de codage.
- **Performances de calcul raisonnable.** Le prototype proposé est un aide à la construction et la spécification graphique du code, implémenté en se basant sur l'outil "AJDT" sous la plateforme ECLIPSE. De même, le compilateur "ajc" a été utilisé. Nous remarquons les mêmes temps de compilation en négligeant le temps de transformation et de génération des templates.
- **Maintenance facile.** La facilité de maintenance est considérée comme un des apports les plus importants des outils visuels de programmation. Au sein d'un code visuel, il est facile de comprendre en un coup d'œil sa structure et les divers flux, ce qui améliore considérablement la maintenabilité. Sous les outils conventionnels, la structure d'un programme et les dépendances de ses entités doivent être explorées à l'aide d'autres outils de visualisation et d'analyse.
- **Interchangeabilité.** Le modèle visuel du programme cible peut être exporté sous forme d'un fichier XMI. L'interchangeabilité éventuelle d'un code orienté-aspects entre les développeurs et leurs IDEs préférés peut se faire de façon plus simple.

6.3.2.3 Sondage basé-questionnaire (Posttest : survey)

Comme une seconde procédure dans l'évaluation, un sondage par questionnaire a été effectué dans le but de clarifier la contribution de l'approche proposée aux programmeurs lors de la phase de codage. Nous décrivons dans ce qui suit ce sondage, puis nous présentons et interprétons les résultats obtenus.

✪ *Description*

Ce sondage a été conçu pour recueillir les observations des participants. Notre objectif principal est de voir comment les différents aspects et caractéristiques du prototype

"HCodelessAJ" affectent leur capacité de programmer en orienté-aspects par rapport à l'approche conventionnelle à travers "AJDT". En outre, nous visons à évaluer la facilité d'utilisation du prototype. Les réponses nous ont permis d'identifier et de remédier à certains lacunes.

Dans ce questionnaire, figurent vingt-six questions qui sont arrangées en sept catégories : "*Subject Experience*", "*Methodology of Coding and HCodelessAJ Prototype Satisfaction*", "*Performed Task*", "*Suitability for the Task*", "*Suitability for Learning*", "*Error Tolerance*" et "*Comments*".

Tout d'abord, on doit savoir, pour chaque participant, le niveau et les compétences de programmation avec "AJDT" (c.-à-d. "*Subject Experience*"). Après l'évaluation empirique, chaque participant doit répondre au questionnaire. Enfin, un interview informel est tenu pour déterminer quelles sont les difficultés rencontrées et même d'extraire en plus leurs opinions et commentaires.

La **Table 6.5** est un récapitulatif des résultats obtenus. Dans cette table chaque valeur est indiquée comme un pourcentage. Ce dernier était calculé en divisant la fréquence de la réponse par le nombre de participants.

ID	Questions and Responses

Subject Experience (Background Information)				
Q1	How do you judge your programming experience?	**Novice**	**Medium**	**Advanced**
		68,75 % (11/16)	31,25 % (5/16)	0 % (0/16)
Q2	How do you judge your AOP/AspectJ programming experience through "AJDT"?	**Novice**	**Medium**	**Advanced**
		68,75 % (11/16)	31,25 % (5/16)	0 % (0/16)
Q3	How do you judge your Eclipse IDE experience?	**Novice**	**Medium**	**Advanced**
		0 % (0/16)	100 % (16/16)	0 % (0/16)

Methodology of Coding and "HCodelessAJ" Prototype Satisfaction					
Q4	What do you think about the use of intuitive graphical icons in this novel coding methodology?				
	Highly Interesting	**Interesting**	**Little of Interest**	**Not Interest**	**No Opinion**
	0 % (0/16)	68,75 % (11/16)	31,25 % (5/16)	0 % (0/16)	0 % (0/16)
Q5	Were the prototype functionalities easy to use?				
	Yes, Highly	**Yes, Somewhat**		**No**	
	50 % (8/16)	50 % (8/16)		0 % (0/16)	
Q6	Is the user interface friendly, clear and easy to understand?				
	Excellent	**Good**	**Satisfactory**	**Needs Improvement**	**No**
	12,5 % (2/16)	25 % (4/16)	37,5 % (6/16)	25 % (4/16)	0 % (0/16)
Q7	Did you need any external help during the experimentation?				
	Yes		**No**		
	37,5 % (6/16)		62,5 % (10/16)		
Q8	How would you rate this prototype in terms of functionalities?				
	Excellent	**Good**	**Satisfactory**	**Needs Improvement**	**No Opinion**
	12,5 % (2/16)	37,5 % (6/16)	25 % (4/16)	25 % (4/16)	0 % (0/16)

Q9	How would you rate this prototype as compared to AJDT?				
	Excellent	**Good**	**Satisfactory**	**Needs Improvement**	**No Opinion**
	12,5 % (2/16)	37,5 % (6/16)	25 % (4/16)	12,5 % (2/16)	12,5 % (2/16)
Q10	Would you recommend this prototype for a training purpose to students?				
	Yes, Highly	**Yes, Somewhat**		**No**	
	62,5 % (10/16)	37,5 % (6/16)		0 % (0/16)	
Q11	How do you assess the usefulness of this methodology for the future of AOP?				
	Very Useful	**Somewhat Useful**	**Not Useful**	**No Opinion**	
	25 % (4/16)	43,75 % (7/16)	0 % (0/16)	31,25 % (5/16)	

Performed Coding Task				
Q12	Was the task of constructing program visual model simple?			
	Excellent	**Good**	**Needs Improvement**	**No Opinion**
	25 % (4/16)	37,5 % (6/16)	37,5 % (6/16)	0 % (0/16)
Q13	Is the number of operations (steps) to achieve the task appropriate?			
	Agree	**Disagree**	**Strongly Disagree**	**No Opinion**
	62,5 % (10/16)	12,5 % (2/16)	0 % (0/16)	25 % (4/16)
Q14	Were the composition and connection between the visual objects and the parameters easy to perform?			
	Agree	**Disagree**	**Strongly Disagree**	**No Opinion**
	62,5 % (10/16)	0 % (0/16)	0 % (0/16)	37,5 % (6/16)
Q15	Are there any difficulties to find the needed visual objects (i.e. graphical icons) for each concept?			
	Yes, Absolutely	**Somewhat**		**No, Strongly**
	68,75 % (11/16)	31,25 % (5/16)		0 % (0/16)

Suitability for the Coding Task					
Q16	How would you rate the effect of the arrangement of the concepts on views on the acceleration of the coding process?				
	Excellent	**Good**	**Satisfactory**	**Needs Improvement**	**No**
	0 % (0/16)	25 % (4/16)	37,5 % (6/16)	37,5 % (6/16)	0 % (0/16)
Q17	Did you find all the information and commands needed at coding process in a given view?				
	Excellent	**Good**	**Satisfactory**	**Needs Improvement**	**No**
	0 % (0/16)	31,25 % (5/16)	31,25 % (5/16)	18,75 % (3/16)	18,75 % (3/16)
Q18	How would you rate the easiness of adapting the visual model created for building a new model?				
	Excellent	**Good**	**Satisfactory**	**Needs Improvement**	**No**
	6,25 % (1/16)	25 % (4/16)	25 % (4/16)	31,25 % (5/16)	12,5 % (2/16)
Q19	How would you rate the easiness of generating templates of code, importing and exporting the visual models created?				
	Excellent	**Good**	**Satisfactory**	**Needs Improvement**	**No**
	0 % (0/16)	43,75% (7/16)	25 % (4/16)	31,25 % (5/16)	0 % (0/16)
Q20	What do you think about the moving back and forth during the coding process within the visual model designer?				
	Excellent	**Good**	**Satisfactory**	**Needs Improvement**	**No**
	12,5 % (2/16)	37,5 % (6/16)	31,25% (5/16)	18,75 % (3/16)	0 % (0/16)

Suitability for Learning			
Q21	Is the time to learn the functionalities of the prototype appropriate as compared to AJDT?		
	Decreased	**Stayed the Same**	**Increased**
	56,25 % (9/16)	43,75 % (7/16)	0 % (0/16)
Q22	Do you think the terms and concepts used are clear and unambiguous? (i.e. graphical icons have a clear meaning)		
	Yes	**Somewhat**	**No**
	81,25 % (13/16)	18,75 % (3/16)	0 % (0/16)
Q23	Did you need to remember numerous details for using the prototype properly?		
	Yes, Highly	**Somewhat**	**No**
	18,75 % (3/16)	31,25 % (5/16)	50 % (8/16)

Error Tolerance				
Q24	How would you rate the errors of system and Eclipse IDE (e.g. crashes) that occurred during the coding process?			
	Often	**Sometimes**	**Rarely**	**Not at all**
	0 % (0/16)	6,25 % (1/16)	12,5 % (2/16)	81,75 % (13/16)
Q25	How would you rate the effort dedicated to correcting mistakes? (e.g. Easily undo the last operation when making mistakes).			
	Yes, Important	**Medium**		**Negligible**
	0 % (0/16)	18,75 % (3/16)		81,75 % (13/16)
Q26	Do you perceive the error messages as helpful?			
	Very Helpful	**Somewhat Helpful**		**Not Help at All**
	0 % (0/16)	68,75 % (11/16)		31,25 % (5/16)

Comments
- What is your overall opinion of "HCodelessAJ" prototype ? - Do you have any comments about how HCodelessAJ prototype affects the coding process ? - ...

Table 6. 5 Récapitulatif des réponses au questionnaire.

✪ *Interprétation des résultats*

Ce sondage donne quelques réponses qui apparaissent suffisantes en général pour tirer des conclusions. Bien que tous les participants ne soient pas totalement familiarisés et qu'ils aient peu d'expérience sur l'utilisation du prototype "HCodelessAJ" par apport à leur familiarisation avec "AJDT", ce sondage a montré en général une perception positive dans l'ensemble et les réponses concordent également avec les résultats de l'étude quantitative précédente.

Les résultats ont révélé un degré de satisfaction acceptable de tous les participants, alors que l'analyse empirique a mis en évidence la faisabilité et l'efficacité de l'approche proposée. On a constaté qu'ils ont donné de nombreux commentaires positifs concernant la facilité d'utilisation ; cependant, ils ont également révélé la nécessité de certaines améliorations mises en évidences par des commentaires sur les lacunes.

On remarque aussi que les participants recommandent fortement d'introduire ce prototype comme un soutien pour les débutants. Il y a aussi une unanimité que cette contribution représente une alternative intéressante et utile au contexte pédagogique pour l'enseignement du paradigme "orienté-aspects" et même une façon prometteuse pour promouvoir son adoption. Cependant, des défis majeurs sont posés pour faire améliorer la génération de code, l'implantation de la "rétro-ingénierie" et l'élaboration des représentations visuelles pour différents concepts et mécanismes offerts en orienté-aspects.

6.4 Conclusion

Dans ce chapitre, nous avons décrit et discuté les expérimentations que nous avons effectuées sur les outils supports implémentés pour les deux contributions. Dans un premier temps, nous avons décrit les expérimentations sur le système de visualisation en deux et trois dimensions, puis, dans un second temps, les expérimentations sur le prototype de codage hybride. Nous avons ainsi donné pour chaque contribution, une analyse empirique accompagnée d'une discussion des résultats obtenus.

Nos contributions ont été évaluées avec succès. Nous avons montré la faisabilité et l'efficacité de chaque approche. De plus, afin de bien examiner et montrer les avantages et les limites, chaque évaluation peut être reproduite dans d'autres conditions, c'est-à-dire des conditions non contrôlées (hors laboratoire), il est aussi intéressant de faire appel à d'autres participants et collecter d'autres données (perspectives expérimentales).

CHAPITRE 7

CONCLUSION GÉNÉRALE

SOMMAIRE

7.1 RAPPEL DU CADRE ET DES OBJECTIFS DE LA THESE 116

7.2 BILAN DES CONTRIBUTIONS 116

7.3 PERSPECTIVES DE RECHERCHE ET TRAVAUX FUTURS 118

Dans ce qui suit, nous concluons ce manuscrit en récapitulant les idées principales qui ont eu lieu dans cette thèse. Puis, nous dressons un bilan synthétique de ce travail en soulignant très précisément nos contributions et en donnant une vision sur les objectifs atteints. Finalement, nous présentons les travaux futurs possibles ainsi que les principales perspectives de recherche qui en découlent.

7.1 Rappel du cadre et des objectifs de la thèse

Dans cette thèse, nous nous sommes concentrés sur la simplification des tâches de compréhension et de construction de code à travers le développement d'une nouvelle approche unifiée pouvant être utilisée en combinaison avec les principales approches de séparation des préoccupations et plus particulièrement les plus prometteuses, à savoir l'approche orientée-aspects.

Notre travail s'inscrit dans le cadre général de la combinaison des techniques graphiques (techniques de visualisation et de construction graphiques et d'interactions) lors de la phase d'implémentation et de maintenance, et concerne plus précisément l'implémentation référence (ASPECTJ) de l'approche orientée-aspects appliquée au langage JAVA.

L'objectif principal est d'apporter des techniques facilitant efficacement la compréhension des grands systèmes logiciels afin de réduire la complexité de maintenance; et ainsi, d'introduire plus de flexibilité et d'interactivité durant le processus de développement. Certaines tâches sont simplifiées, particulièrement lors de la programmation, afin d'aboutir à une meilleure séparation des préoccupations avec moins de coûts en termes de temps et d'effort.

7.2 Bilan des contributions

Notre état de l'art nous a amenés à orienter nos recherches et à définir deux grands axes de recherche à suivre et qui constituent le plan de travail de notre thèse. Dans chaque axe, nous sommes arrivés à apporter une proposition originale. Nous récapitulons les principales contributions comme suit :

La **première contribution** de cette thèse est une approche de visualisation bidimensionnelle (2D) et tridimensionnelle (3D) pouvant analyser et afficher simultanément des propriétés quantitatives et des liens de dépendances des grands systèmes logiciels orientés aspects et plus particulièrement les programmes ASPECTJ dans le but d'effectuer des analyses qualitatives. Notre approche se base sur des vues polymétriques et métaphores visuelles intéressantes pour rendre les analyses plus intuitives et plus efficaces. Dans un premier temps, nous nous focalisons sur le volet statique du logiciel en occurrence son code source plutôt que sur l'exécution.

Nous avons développé à l'occasion de nos travaux de recherche dans ce premier axe un système offrant une visualisation dédiée en 2D et 3D sous la plateforme ECLIPSE. En premier lieu, nous avons proposé "VizzAspectJ-2D", un outil de visualisation en 2D servant à afficher à la fois la hiérarchie d'héritage d'un programme ASPECTJ et les liens de dépendances existants entre ses éléments en utilisant la vue polymétrique proposée par *Michael Lanza* [Lan 03c]. En second lieu pour une visualisation en 3D, nous avons proposé

l'outil "VizzAspectJ-3D" permettant d'afficher la structure hiérarchique en utilisant la métaphore de la ville, comme dans CodeCity et VERSO.

Nous avons également apporté un grand soin dans le choix de l'architecture pour chaque outil support afin de garantir sa pérennité, son extensibilité, mais aussi sa facilité d'utilisation et son support des besoins et des actions de l'utilisateur, qu'il soit débutant ou expert. Cette contribution a fait l'objet d'une publication déjà parue dans un journal international spécialisé [Ben 13].

Nos principes dans l'élaboration des vues étaient que la façon d'afficher simultanément divers liens de dépendances et de la hiérarchie d'héritage ou de la structure hiérarchique respectivement dans un espace 2D et 3D ne génère pas trop d'encombrement visuel, que les vues produites soient extensibles et qu'elles offrent aussi à l'utilisateur toute une gamme de méthodes d'interaction afin de mieux filtrer les informations visualisées selon le besoin. En outre, cette contribution a été évaluée avec succès sur des systèmes logiciels réels de grande taille et des résultats encourageants ont été obtenus.

La **seconde contribution** de cette thèse est l'introduction de l'approche visuelle durant la phase de codage. C'est une nouvelle méthodologie hybride "**HM4AOP**" (**H**ybrid **M**ethodology **for A**spect-**O**riented **P**rogramming) intégrant les deux styles de codage, textuel et visuel, pour l'approche orientée-aspects. C'est une alternative complémentaire et non concurrente à la méthode conventionnelle (à base de texte), permettant de coder d'une façon plus attrayante et facile en mettant l'accent sur les innovations d'AOP, plutôt que sur des détails d'implémentation dont le but de gérer au mieux les diverses préoccupations. Cette contribution a fait l'objet d'un article soumis pour publication dans une revue internationale spécialisée [Ben 15].

Nous avons concrétisé cette proposition par la réalisation d'un prototype de son outil support "**HCodelessAJ**" (**H**ybrid **Codeless** Programming Methodology for **A**spect**J**) sous la plateforme ECLIPSE, en étendant l'outil support "AJDT".

Cette proposition est, à notre avis, une tentative réussie pour réduire l'utilisation de la méthode textuelle lors de l'introduction de diverses implémentations orientées aspects. En résumé, les principaux apports substantiels sont les suivants :

- Eviter certains écueils de l'approche conventionnelle à base de texte (p. ex. les formalismes syntaxiques).
- Simplifier, en termes de temps et d'effort, les tâches d'enseignement et d'apprentissage des concepts et mécanismes puissants de l'orienté-aspects dans un contexte hybride en incorporant les techniques graphiques et textuelles.
- Permettre aux programmeurs de coder dans un haut niveau de flexibilité grâce à des possibilités d'interactivité, ce qui, par conséquent, permet de : (1) empêcher les diverses erreurs syntaxiques et d'améliorer la qualité du code, (2) apercevoir la structure du code

rapidement en particulier pour les grands systèmes logiciels, et (3) rendre le code plus compréhensible pour les tâches de maintenance ultérieures (Meilleure efficacité dans la reconnaîssance et la modification des concepts et mécanismes de programmation), etc.

Nous avons fait une évaluation de notre outil prototype **"HCodelessAJ"** en l'utilisant quelques programmes benchmarks en ASPECTJ. Avec cette évaluation, nous avons montré la faisabilité de notre approche, et son efficacité en tant que une idée innovatrice de codage, couvrant particulièrement la programmation orientée-aspects en ASPECTJ. Nous avons aussi montré comment l'utiliser pour éditer, maintenir, et déployer un code orienté-aspects facilement. Cette évaluation nous a permis de tirer certaines remarques.

Dans ce qui suit, nous discutons les perspectives qui émergent de notre travail et expérimentations.

7.3 Perspectives de recherche et travaux futurs

Nous pourrions diviser les travaux futurs comme des perspectives expérimentales et de recherche en au moins deux directions, correspondant aux deux principales contributions de cette thèse :

Tout d'abord, pour la **première direction**, nos travaux autour de la visualisation méritent d'être poursuivis comme suit:

- Premièrement, dans le cadre d'une visualisation tridimensionnelle, la métaphore de ville ouvre de nombreuses perspectives pour une analyse efficace et une compréhension rapide. Une des plus importantes est l'approfondissement de la métaphore et de son impact sur l'analyse en visualisant le code au niveau de la méthode, consigne, point de coupure, etc. On peut aussi utiliser une métaphore de briques, composant chaque construction de la ville, et une métrique couleur pour plus de détail d'informations. Nous souhaitons aussi améliorer la méthode de navigation en utilisant notamment de nouvelles techniques avancées d'interaction.

- L'utilisation d'autres métaphores avec des sémantiques plus fortes peut permettre d'envisager des modèles de simulation de certains phénomènes liés au logiciel. Par exemple, l'utilisation de la théorie de propagation de la chaleur, peut permettre avec quelques adaptations de simuler la propagation des effets de changements dans les logiciels.

- Au sujet des anomalies de conception, qui font baisser la qualité globale du logiciel, l'implémentation d'une analyse automatique des métriques permettant la détection de celles-ci peut sembler complexe. Nous pensons que la recherche dans cette voie à des

limites et l'automatisation ne sera pas toujours fiable, et qu'une recherche semi-automatique et interactive peut aider à mieux comprendre quelles sont les raisons des anomalies trouvés [Dha 07].

- Comme perspective expérimentale, l'un de nos objectifs principaux est de tester empiriquement la compréhension avec des utilisateurs de différents niveaux d'expérience professionnelle dans le développement de logiciels. Le but serait de mesurer à quel point l'utilisation de nos outils supports permet de faciliter la tâche de compréhension.
- Dans un premier temps, nous nous sommes focalisés exclusivement sur le volet statique, mais nous pensons que l'analyse du volet dynamique (c.-à-d. l'exécution) doit également être prise en compte pour étudier le logiciel de façon précise.
- À plus long terme, nous envisageons d'introduire des actions graphiques permettant à l'utilisateur de modifier le code source directement grâce à la manipulation des vues produites. Nous pensons que les outils de visualisation sont tout à fait adaptés à servir une tâche de re-conception et que ce genre d'actions peuvent être reportées directement dans le code ce qui facilite grandement les tâches de développement et de maintenance.

Pour la **deuxième direction**, des travaux restent à faire dans le domaine de la programmation visuelle dans certains champs que nous résumons comme suit :

- Tout d'abord, nous sommes intéressés par l'extensibilité de l'implémentation actuelle du prototype : (1) intégrer d'autres fonctionnalités, (2) envisager de nouvelles constructions graphiques, et (3) introduire d'autres vues de visualisation y compris les possibilités d'interaction à l'aide de techniques avancées qui permettent à l'utilisateur de naviguer facilement (marqué **(A)** dans la **Figure 7.1**).
- Deuxièmement, il serait bénéfique d'élargir notre domaine d'application pour d'autres implémentations orientées aspects (tels que : *AspectC++, AspectC#, AspectS*, etc.), ce qui aidera à consolider l'adoption de celles-ci dans la communauté du génie logiciel.

  Une éventuelle extension de l'éditeur visuel (VMD) conjointement avec le générateur de code "*HM4AOP Code Generator*" devrait être facile. Seulement les règles de transformation (mapping) pour la génération de code devront être spécifiées (nous pouvons intégrer plusieurs méta-modèles dans un module ACCELEO, chacun d'eux sera spécifique à une implémentation). La génération de templates de code en une implémentation préférée à partir d'un modèle visuel générique sera effectuée à l'aide d'un composant approprié dans ce générateur (marqué **(B)** dans la **Figure 7.1**).

Cette éventuelle extension peut contribuer à :

- Augmenter l'aspect réutilisabilité d'un code orienté-aspects en exportant son modèle visuel pour une réutilisation ultérieure au sein de l'environnement de développement préféré.
- Assurer une interchangeabilité facile du modèle visuel (un code orienté-aspect visuel) entre différentes implémentations et IDEs supports (*Towards a neutral exchange based on XMI format among AOSD tools*).

- Enfin, troisièmement, comme le mécanisme de génération implémenté repose principalement sur la technologie ACCELEO (M2T, *Model-to-Text transformation*) pour transformer les constructions constituantes le modèle visuel en des templates textuelles, il sera intéressant d'implémenter l'opération inverse, c.-à-d. la rétro-ingénierie (T2M, *Text-to-Model transformation*) pour une reconstruction de modèles visuels initiaux à partir du code source (c.-à-d. *Back transformation of ASPECTJ code*).

Ces extensions semblent être faisables et utiles pour le paradigme "orienté-aspects" où le programmeur aura toujours la possibilité d'évoluer en toute transparence selon deux voies (*Bottom-Up* et *Top-Down*) pour transformer la source d'un programme dans une implémentation (en anglais : *Low-level representation*) à une autre en créant des modèles visuels (en anglais : *Higher-level models and artifacts*) et sans la nécessité d'effectuer de modifications majeures dans le code.

À plus long terme, nous projetons de mettre en place un outil de développement supportant le "*Round-Trip Engineering*" d'artefacts produits, afin que les caractérisations (ou spécifications) manuelles des templates de code soient prises en compte et fusionnées par le générateur (marqué **(C)** dans la **Figure 7.1**).

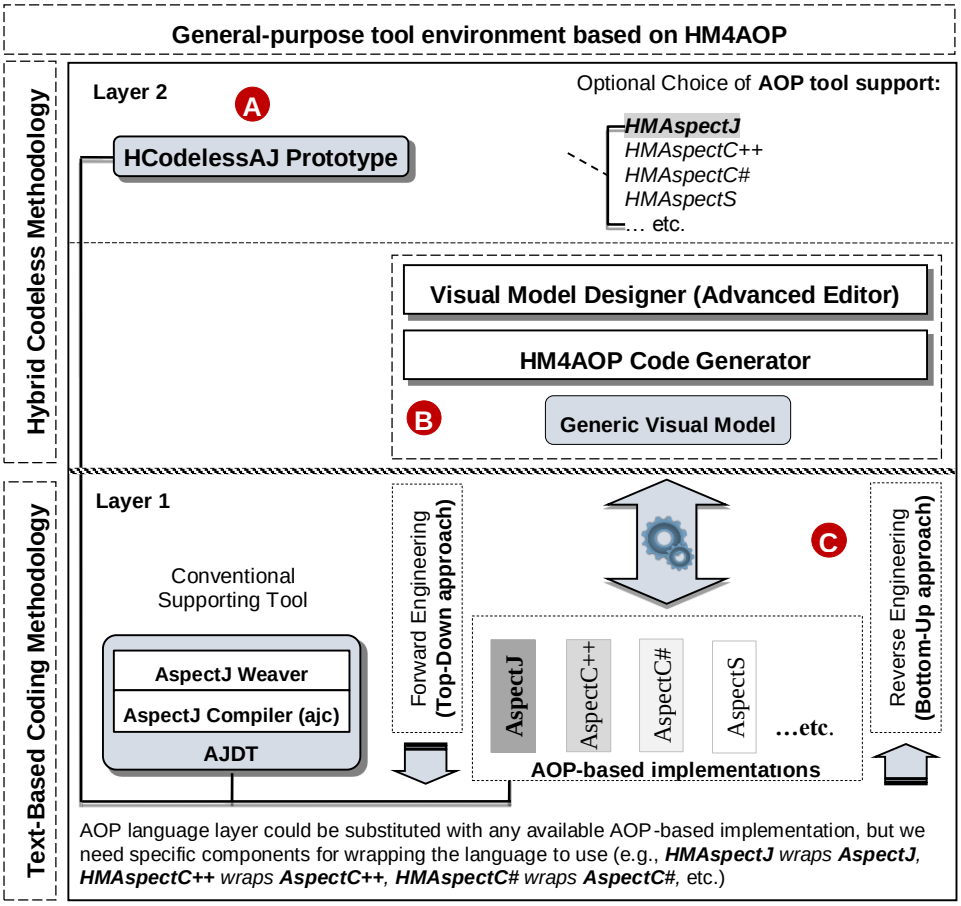


Figure 7. 1 Une vue globale sur l'architecture conceptuelle du Framework "HM4AOP".

Il est plus approprié de parler d'une génération avancée d'environnements avec une interface de programmation hybride qui vise à améliorer la manière d'interagir visuellement avec un code orienté-aspects de grande taille. La **Figure 7.1** représente un aperçu général sur l'architecture conceptuelle de cet environnement reposant sur l'approche "**HM4AOP**" et implantant ces trois perspectives sous la plateforme ECLIPSE. Comme montré, nous avons divisé l'architecture en deux couches principales afin d'obtenir une certaine flexibilité pour d'autres extensions.

1. Références Bibliographiques

[Abi 13a] M. Abid et Jawawi Dayang N. A., "Aspect-oriented code generation for integration of aspect-orientation and model-driven engineering", *International Journal of Software Engineering and Its Applications* (IJSEIA), Vol. 7, No. 2, 2013.

[Abi 13b] M. Abid et Jawawi Dayang N. A., "Aspect-oriented model-driven code generation: A systematic mapping study", *Information and Software Technology* (IST), Vol. 55, No. 2, pp. 395–411, 2013.

[Aks 94] M. Aksit, J. Bosch, W. van der Sterren et L. Bergmans, "Real-Time Specification Inheritance Anomalies and Real-Time Filters", In: Proceedings of *the 8th European Conference on Object-Oriented Programming* (ECOOP), Lecture Notes in Computer Science, Vol. 821, pp. 386–407, Springer-Verlag, Bologne, Italie, Juillet 1994.

[Aks 98] M. Aksit et B. Tekinerdogan, "Aspect-Oriented Programming Using Composition Filters", *ECOOP'98 Workshop Reader*, Springer Verlag, pp. 435, July 1998.

[Ala 07] S. Alam et P. Dugerdil, "Evospaces Visualization Tool: Exploring Software Architecture in 3D", In Proceedings of *the 14th Conf. on Reverse Engineering*, 2007.

[And 02] K. Andrews, "Information Visualisation", 2002.

[Ben 10] J. Bennett, K. Cooper et L. Dai, "Aspect-Oriented Model-Driven Skeleton Code Generation: A Graph-Based Transformation Approach", *Science of Computer Programming*, Elseiver, Vol. 75, No. 8, pp. 689–725, 2010.

[Ben 11a] Sassi Bentrad et Djamel Meslati, "2D and 3D Visualization of AspectJ Programs", In Proceedings of *the 10th Int'l Symposium on Programming and Systems* (ISPS), Algeria, pp. 183–190, 2011.

[Ben 11b] Sassi Bentrad et Djamel Meslati, "Visual Programming and Program Visualization –Towards an Ideal Visual Software Engineering System–", *ACEEE International Journal on Information Technology* (IJIT), Vol. 1, No. 3, DOI: 01.IJIT.1.3.22, pp. 56–62, 2011.

[Ben 13] Sassi Bentrad et Djamel Meslati, "Visualizing and Analyzing the Structure of AspectJ Software under the Eclipse Platform", *International Journal of Software Engineering and Its Applications* (IJSEIA), Vol. 7, No. 3, pp. 353–376, May 2013.

[Ben 15] Sassi Bentrad et Djamel Meslati, "Toward a Hybrid Approach to Coding Aspect-Oriented Programs", *Computing and Informatics Journal* (CAI), ISSN: 1335-9150, 2015.

[Bia 07] Andrea Biaggi, "Citylyzer- A 3D Visualization Plug-in for Eclipse", 2007.

[Bos 04] M. Boshernitsan et M. Downes, "Visual programming languages: a survey", University of California, Berkeley, California 94720, Tech. Rep., December 2004.

[Bou 01] N. M. N. Bouraqadi-Saâdani et T. Ledoux, "Le point sur la programmation par aspects", Technique et science informatiques, Vol. 20, pp. 505–528, 2001.

[Bou 11] Simon Bouvier, "Utilisation de la visualisation interactive pour l'analyse des dépendances dans les logiciels", Université de Montréal, Faculté des arts et des sciences, Août 2011.

[Bur 89] M. M. Burnett et A. L. Ambler, "Influence of visual technology on the evolution of language environments", *IEEE Computer*, Vol. 6, No. 2, pp. 9–22, October 1989.

[Bur 99] M. M. Burnett, "Visual Programming", In "*Encyclopedia of Electrical and Electronics Engineering*" (John G. Webster, ed.), John Wiley & Sons Inc., New York, 1999.

[Bur 01] M. M. Burnett, "Software engineering for visual programming languages", In: *Handbook of Software Engineering and Knowledge Engineering*, Vol. 2, 2001.

[Cai 99] J. W. Cain et R.J. McCrindle, "Software visualisation using c++ lenses", In: Proceeding of *the 7th IEEE International Workshop on Program Comprehension*, pp. 20–26, 1999.

[Car 08] Yaser Carpendale et Sheelagh Ghanam, "A Survey Paper on Software Architecture Visualization", Technical report, Univ. of Calgary, 2008.

[Cas 11] P. Caserta, O. Zendra et D. Bodénès, "3D Hierarchical Edge Bundles to Visualize Relations in a Software City Metaphor", In: Proceedings of *the 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis* (VISSOFT), USA, Septembre 2011.

[Cas 12] Caserta Pierre, "Analyse statique et dynamique de code et visualisation des logiciels via la métaphore de la ville : contribution à l'aide à la compréhension des programmes", Thèse de Doctorat, Université de Lorraine, France, Décembre 2012.

[Cha 87] Shi-Kuo Chang, "Visual languages: A tutorial and survey", *IEEE software*, pp. 29–39, January 1987.

[Cha 90] Shi-Kuo Chang, "Principles of Visual Programming Systems", Prentice-Hall International Editions, ISBN: 0-13-710765-X, 1990.

[Cha 95] Shi-Kuo Chang et al., "Visual language system for user interface", *IEEE Software*, Vol. 12, No. 2, pp. 33–44, ISSN: 0740-7459, 1995.

[Cha 96] N. Chapin et T.S. Lau, "Effective Size: An Example of Use from Legacy Systems", *Journal of Software Maintenance: Research and Practice*, Vol. 8, 1996.

[Cha 02] S.M. Charters, C. Knight, N. Thomas et M. Munro, "Visualisation for Informed Decision Making; from Code to Components", In: Proceeding of *the 14th Int'l Conf. on Software Engineering and Knowledge Engineering* (SEKE), 2002.

[Col 04] A. Colyer, A. Clement, G. Harley et M. Webster, "Eclipse AspectJ: aspect-oriented programming with AspectJ and the Eclipse AspectJ Development Tools", Addison-Wesley Professional, 2004.

[Cor 89] T. A. Corbi, "Program understanding: challenge for the 1990's", *IBM System Journal*, Vol. 28, No. 2, pp. 294–306, ISSN: 0018-8670, 1989.

[Cox 89] P.T. Cox, F. R. Giles et T. Pietrzykowski, "Prograph: a step towards liberating programming from textual conditioning", In: Proceeding of *the IEEE Workshop on Visual Languages*, pp. 150–156, October 1989.

[Den 02] M. Denford, T. O'Neill et J. Leaney, "Architecture-Based Visualisation of Computer Based Systems", In: Proceedings of *the IEEE Int'l Conf. on Engineering of Computer-Based Systems*, Vol. 2, 2002.

[Dem 99] S. Demeyer, S. Ducasse et M. Lanza, "A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualisation", In: Proceedings of *the 6th Working Conf. on Reverse Engineering*, pp. 175–186, 1999.

[Des 05] I. Despi et L. Lucay, "Aspect Oriented Programming Challenges", *Anale Seria Informatica*, Vol. 2, No. 1, pp. 65–70, 2005.

[Dha 07] Karim Dhambri, "Détection visuelle d'anomalies de conception dans les programmes orientés objet", Mémoire de maîtrise, Département d'Informatique et Recherche Opérationnelle, Université de Montréal, Décembre 2007.

[Dha 08] K. Dhambri, H. A. Sahraoui et P. Poulin, "Visual Detection of Design Anomalies", In: Proceedings of *the 12th Euro. Conf. on Software Maintenance and Reengineering*, pp. 279–283, 2008.

[Die 07] Diehl Stephan, "Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software", Springer Verlag, Inc., 2007.

[Die 94] A. Dieberger, "Navigation in Textual Virtual Environments Using a City Metaphor", Ph.D. thesis, Vienna Univ. of Tech., 1994.

[Die 98] A. Dieberger et A.U. Frank, "A City Metaphor to Support Navigation in Complex Information Spaces", *Journal of Visual Languages and Computing* (JVLC), Vol. 9, No. 6, pp. 597–622, 1998.

[Dos 00] C.R. Dos Santos et al., "Mapping Information onto 3D Virtual Worlds", In: Proceedings of *the Int'l Conf. on Information Visualization*, pp. 19–21, 2000.

[Dos 02] C. Dos Santos, "Visualisation métaphorique tridimensionnelle de l'information", Ph.D. thesis, Ecole nationale supérieure des télécommunications (PARIS-ENST), 2002.

[Eic 98] S. Eick "Maintenance of Large Systems", in Software Visualization, Stasko, J., Dominque, J., Brown, M., and Price, B., Eds., London MIT Press, 1998, pp. 315–328.

[Fen 00] Norman E. Fenton et Martin Neil., "Software Metrics: Roadmap", In: Proceedings of *the ACM Conference on the Future of Software Engineering* (ICSE), pp. 357–370, 2000.

[Fen 03] N. Fenton et S. L. Peeger, "Software Metrics: A Rigorous and Practical Approach", Boston, MA, USA: PWS Publishing Co., 2nd edition, 2003.

[Fer 02] F. Ferruci, G. Tortora et G. Vitello, "Exploiting visual languages in software engineering", In: Chang S. K., *Handbook of software engineering and knowledge engineering*. River Edge, NJ: Singapore World Scientific, 2002.

[Fer 04] Ferut Térence et Leroy Sébastien, "La programmation orientée-aspects", Juin 2004.

[Fre 09] Munoz Freddy, Baudry Benoit, Delamare Romain et Yves Le Traon, "Inquiring the Usage of Aspect-Oriented Programming: An Empirical Study", pp. 2–4, 2009.

[Fre 13] G.W. French, "The Larch Environment - Python programs as visual, interactive literature", Master of Science Thesis, School of Computing Science - University of East Anglia, 2013.

[Fre 14] G. W. French, J. R. Kennaway et A. M. Day, "Programs as visual, interactive documents", *Software: Practice and Experience* (SPE), Vol. 44, No. 8, DOI: 10.1002/spe.2182, pp. 911–930, 2014.

[Fro 06] A. Fronk, A. Bruckhoff et M. Kern, "3D Visualisation of Code Structures in Java Software Systems", In: Proceedings of *the Symposium on Software Visualization*, 2006.

[Ger 94] Nahum D. Gershon, "From perception to visualization", In L. Rosenblum, R.A. Earnshaw, J. Encarnacao, H. Hagen, A. Kaufman, S. Klimenko, G. Nielson, F. Post, and D. Thalmann, editors, *Scientific Visualization: Advances and Challenges*. Academic Press, 1994.

[Gli 85] E.P. Glinert, "PICT: Experiments in the Design of Interactive, Graphical Programming Environments", Ph.D. Dissertation, University of Washington, 1985.

[Gol 90] E. J. Golin et S. P. Reiss, "The specification of visual language syntax", *Journal of Visual Languages and Computing* (JVLC), Vol. 1, No. 2, pp. 141–157, 1990.

[Gra 03] J. D. Gradecki et N. Lesiecki, "Mastering AspectJ: aspect-oriented programming in Java", Wiley Publishing, Inc.: New York, NY, USA, 2003.

[Gre 05] O. Greevy, M. Lanza et C. Wysseier, "Visualizing feature interaction in 3D", In: Proceedings of *the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis* (VISSOFT), pp. 1–6, 2005.

[Gre 93] Mark Green et Jun Rekimoto: "The Information Cube: Using Transparency in 3D Information Visualization", In: Proceedings of *the 3rd Workshop on Information Technology and Systems*, pp. 125–132, 1993.

[Gui 06] G. Langelier, H. A. Sahraoui et P. Poulin, "Visualisation du logiciel et de son évolution", Dans Acte *Atelier sur l'évolution du logiciel* (AEL), 2006.

[Hac 06] Ouafa Hachani, "Patrons de conception à base d'aspects pour l'ingénierie des systèmes d'information par réutilisation", Thèse de Doctorat, Université de Joseph Fourier-Grenoble I, Juillet 2006.

[Has 01] Mountaz Hascoët et Michel Beaudouin-Lafon, "Visualisation interactive d'information", Laboratoire de Recherche en Informatique, UMR8623 du CNRS LRI, Université Paris-Sud, 91405 Orsay Cedex, CEPAD, 2001.

[Has 07] Salima Hassaine, "Un système d'aide à la visualisation interactive de logiciels", Mémoire de maîtrise, Département d'Informatique et Recherche Opérationnelle, Université de Montréal, Décembre, 2007.

[Hat 04] A. Hatch, "Software Architecture Visualisation", Ph.D. thesis, Univ. of Durham, 2004.

[Hea 98] C.G. Healey, "Applications of Visual Perception in Computer Graphics", In: Proceedings of *SIGGRAPH: Applications of Visual Perception in Computer Graphics*, 1998.

[Her 00] I. Herman, M.S. Marshall et G. Melan¸con., "Graph Visualization and Navigation in Information Visualization: A Survey", *IEEE Transactions on Visualization and Computer Graphics*, 2000.

[Hol 07] D. Holten, B. Cornelissen et J.J. van Wijk, "Trace visualization using hierarchical edge bundles and massive sequence views", In: Proceedings of *the 4th IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis* (VISSOFT), pp. 47–54, June 2007.

[Hol 06] Danny Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data", *IEEE Transactions on Visualization and Computer Graphics*, Vol. 12, pp. 741–748, 2006.

[Hyu 13] Hyun Seung, et al., "MOF based Code Generation Method for Android Platform", *International Journal of Software Engineering and Its Applications* (IJSEIA), Vol. 7, No. 3, 2013.

[Irw 03] W. Irwin et N. Churcher, "Object-Oriented Metrics: Precision Tools and Configurable Visualisations", In: Proceedings of *the 9th IEEE Int'l Software Metrics Symposium*, pp. 112–123, 2003.

[Jer 03] T. Jeremy et Bradley, "An Examination of Aspect-Oriented Programming in Industry", Technical Report, Colorado State University, USA, May 2003.

[Kan 02] S.H. Kan, "Metrics and Models in Software Quality Engineering", Addison-Wesley Longman Publishing Co., 2002.

[Kel 02] C. Kelleher et al., "Alice2: programming without syntax errors", in: Proceeding of *the 15th annual ACM symposium on User Interface Software and Technology* (UIST), 2002.

[Kic 01a] G. Kiczales et al., "An Overview of AspectJ", In: Proceedings of the *15th European Conference on Object-Oriented Programming* (ECOOP), Springer, pp. 327–353, 2001.

[Kic 01b] G. Kiczales et al., "Discussing aspects of AOP", *Communications of the ACM*, Vol. 44, No. 10, pp. 33–38, Octobre 2001.

[Kic 92] G. Kiczales, "Towards a New Model of Abstraction in the Engineering of Software", In: Proceedings *of the Workshop on Refection and Meta-Level Architecture* (IMSA), pp. 113–153, 1992.

[Kic 97] G. Kiczales et al., "Aspect-oriented programming", In: Proceedings of *the 11th European Conference on Object-Oriented Programming* (ECOOP), Lecture Notes in Computer Science, Springer, Vol. 1241, pp. 220–242, Juin 1997.

[Kle 03] Kleppe, J. Warmer et W. Bast., "MDA Explained, The Model-Driven Architecture: Practice and Promise", Addison Wesley, 2003.

[Kni 00] C. Knight et M. Munro, "Virtual But Visible Software", In: Proceedings of *the 4th IEEE International Conference on Information Visualisation*, pp. 198–205, 2000.

[Koe 92] D. Koelma, R. van Balen et A. Smeulders: "SCIL-VP: a Multi-Purpose Visual Programming Environment", In: Proceedings of *the ACM/SIGAPP Symposium on Applied Computing*, pp. 1188–1198, 1992.

[Lad 03] R. Laddad, "AspectJ in Action: Practical Aspect-Oriented Programming", Greenwich, CT, USA: Manning Publications Co., 2003.

[Lad 09] R. Laddad, "AspectJ in Action: Enterprise AOP with Spring Applications", Manning Publications, 2nd edition, 2009.

[Lak 86] F. Lakin, "Spatial parsing for visual languages", In S.K. Chang, T. Ichikawa, and P. Ligomenides, editors, Visual Languages, pp. 35–85. Plenum Press, New York, 1986.

[Lan 03a] Michele Lanza, "CodeCrawler - Lessons Learned in Building a Software Visualization Tool", In: Proceedings of *the IEEE Euro. Conf. on Software Maintenance and Reengineering*, 2003.

[Lan 03b] Michele Lanza, "Object-Oriented Reverse Engineering - Coarse-grained, Finegrained, and Evolutionary Software Visualization", Ph.D. thesis, Univ. of Bern, 2003.

[Lan 03c] M. Lanza et S. Ducasse, "Polymetric Views - a Lightweight Visual Approach to Reverse Engineering", *IEEE Computer Society Trans. Software Engineering*, Vol. 29, No. 9, pp. 782–795, 2003.

[Lan 05a] G. Langelier, H. A. Sahraoui et P. Poulin, "Visualization-Based Analysis of Quality for Large-Scale Software Systems", In: Proceedings of *the 20th IEEE/ACM Int'l Conf. on Automated Software Engineering*, pp. 214–223, 2005.

[Lan 05b] G. Langelier, H. A. Sahraoui et P. Poulin, "Visualisation and analysis of software quantitative data", In: Proceedings of *the 9th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering* (QAOOSE), 2005.

[Lan 05c] G. Langelier, H. A. Sahraoui et P. Poulin, "Visualisation et analyse de logiciels de grande taille", L'OBJET, Vol. 11, No. 1-2, pp. 159–173, Département d'Informatique et Recherche Opérationnelle, Université de Montréal, Québec, Canada, Mars 2005.

[Lan 06] Guillaume Langelier, "Visualisation de la qualité des logiciels de grandes tailles", M.sc. thesis, Département d'Informatique et Recherche Opérationnelle, Université de Montréal, Décembre 2006.

[Lan 07] G. Langelier et K. Dhambri, "Visual analysis of azureus using VERSO", In: Proceedings of *the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis* (VISSOFT), pp. 163–164, 2007.

[Lan 08] G. Langelier, H. A. Sahraoui et P. Poulin, "Exploring the Evolution of Software Quality with Animated Visualization", In: Proceedings of *the IEEE Symp. on Visual Language and Human-Centric Computing*, 2008.

[Leb 99] Frédéric LEBLANC, "Programmation visuelle pour la description du comportement d'un robot autonome", Septembre 1999.

[Lew 03] C. Lewerentz et A. Noack, "CrocoCosmos - 3D Visualization of Large Object-Oriented Programs", Graph Drawing Software. Springer-Verlag, 2003.

[Lop 94] C. Lopes et K. J. Lieberherr, "Abstracting Process-to-Process Relations in concurrent Object-Oriented Applications", In: Proceedings of *the 8th European Conference on Object-Oriented Programming* (ECOOP), Lecture Notes in Computer Science Vol. 821, Springer-Verlag, pp. 81–99, Bologne, Italie, Juillet 1994.

[Lop 95] C. Lopes et W. Hursch, "Separation of Concerns", Technical rapport, College of Computer Science, Northeastern University, Boston, MA, Etats-Unis, Février 1995.

[Mad 07] L. Madeyski, L. Szala, "Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study", *IET Software*, pp. 180–187, 2007.

[Mes 04] Djamel Meslati et al., "La composition de filtres et AspectJ : Une Comparaison conceptuelle", In: Proceedings of *the 8th Maghrebian Conference on Software Engineering and Artificial Intelligence*, Sousse, Tunisie, pp. 9–12, Mai 2004.

[Mes 06] Djamel Meslati, "MAGE : Une Approche Ontogénétique de l'Evolution dans les Systèmes Logiciels Critiques et Embarquées", Thèse de Doctorat d'état, Université Badji Mokhtar-Annaba, Algérie, 2006.

[Mes 07] Djamel Meslati, "Ingénierie des Logiciels : Les Nouveaux Paradigmes", Support de cours du module SAP, Master 2, Université Badji Mokhtar-Annaba, Algérie, Février 2007.

[Muh 10] Muhammad Sarmad Ali et al., "A systematic review of comparative evidence of aspect-oriented programming", *Information and Software Technology* (IST), Vol.52, No.9, pp. 871–887, 2010.

[Mur 99] G.C. Murphy, R.J. Walker, and E.L.A. Baniassad, "Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-Oriented Programming", August 1999.

[Mye 90] Brad A. Myers, "Taxonomies of visual programming and program visualization", *Journal of Visual Languages and Computing* (JVLC), Vol. 1, No. 1, pp. 97–123, March 1990.

[Mye 96] Brad A. Myers et al., "Strategic directions in human-computer interaction", *ACM Computing Surveys*, Vol. 28, No. 4, 1996.

[Mye 06] Brad A. Myers et A. J. Ko, "Barista: an Implementation Framework for Enabling New Tools", In: Proceedings of *Interaction Techniques and Views in Code Editors* (CHI), 2006.

[New 90] Peter Newton, "Visual Programming and Parallel Computing", Prentice-Hall International Editions, ISBN: 0-13-710765-X, 1990.

[Ngu 05] Nguyen Manh Tien, "Programmation Orientée Aspect", Juillet 2005.

[Noa 05] A. Noack et C. Lewerentz, "A Space of Layout Styles for Hierarchical Graph Models of Software Systems", In: Proceedings of *the ACM Symp. on Software Visualization*, pp. 155–164, 2005.

[Oka 94] H. Okamura et Y. Ishikawa, "Object Location Control Using Meta-level Programming", In: Proceedings of *the 8th European Conference on Object-Oriented Programming* (ECOOP), Lecture Notes in Computer Science, Springer-Verlag, Bologne, Italie, Vol. 821, pp. 299–319, Juillet 1994.

[Oss 99] H. Ossher et P. Tarr, "Multi-dimensional Separation of Concerns using Hyperspace", IBM Research Report 21452, IBM T.J. Watson Research Center, April, 1999.

[Pan 07] T. Panas, T. Epperly, D. Quinlan, A. Saebjornsen et R. Vuduc, "Communicating Software Architecture using a Unified Single-View Visualization", In: Proceedings of *the 12th IEEE Int'l Conf. on Engineering Complex Computer Systems*, pp. 217–228, 2007.

[Par 72] D.L. Parnas, "On the criteria to be used in decomposing systems into modules", *Communications of the ACM*, Vol. 15, No. 12, pp. 1053–1058, December 1972.

[Per 08] JBG. Perez-Schofield, EG; Rosell, FO. Soler et MP. Cota, "Visual Zero: a persistent and interactive object-oriented programming environment", *Journal of Visual Languages and Computing* (JVLC), Vol. 19, No. 3, pp. 380–398, June 2008.

[Pie 02] E. Pietriga, "Environnements et langages de programmation visuels pour le traitement de documents structurés", Thèse de Doctorat, Institut National Polytechnique de Grenoble, Grenoble, Novembre 2002.

[Pig 96] Thomas M. Pigoski, "Practical Software Maintenance: Best Practices for Managing Your Software Investment", Wiley, New York, 1996.

[Pla 13] I. Plauska et R. Damasevicius, "Usability analysis of visual programming languages using computational metrics", In: Proceedings of *the IADIS International Conference on Interfaces and Human-Computer Interaction*, Prague, Chech Republic, pp. 63–70, July 2013.

[Plo 02] D. Ploix., "Analogical Representations of Programs", In: Proceedings of *the 1st Int'l Workshop on Vis. Soft. for Understanding and Analysis*, 2002.

[Pri 98] B. A. Price, R. M. Baecker et I. S. Small, "An Introduction to Software Visualization", In: *Software Visualization*, Stasko, J., Dominque, J., Brown, M., and Price, B., Eds., London, England MIT Press, pp. 4–26, 1998.

[Qui 04] Laurent Quintian, "JADAPT : un modèle pour améliorer la réutilisation des préoccupations dans le paradigme objet", Thèse de Doctorat, Université de Nice Sophia-Antipolis, Juillet 2004.

[Rav 02] H.R. Ravaosolo, "Intégration de flot de contrôle et de flot de données dans un langage de programmation visuelle", Thèse de Doctorat, Université de Genève, 2002.

[Ren 04] P. Renaud, J. Phillipe et L. Seinturier, "Programmation orientée aspect pour Java/J2EE", Eyrolles, Paris, 2004.

[Ril 05] J. Rilling et S. P. Mudur, "3D visualization techniques to support slicing-based program comprehension", *Computers and Graphics*, Vol. 29, No. 3, pp. 311–329, 2005.

[Rob 91] George G. Robertson, Jock D. Mackinlay et Stuart K. Card, "Cone trees: animated 3D visualizations of hierarchical information", In: Proceedings of *the SIGCHI conference on Human factors in computing systems: Reaching through technology* (CHI), ACM, pp. 189–194, USA, 1991.

[Rob 02] Roger T. Alexander, James M. Bieman, "Challenges of Aspect Oriented Technology", In: Proceedings of *the ICSE Workshop on Software Quality*, Florida, 2002.

[Rob 04] Robert E. Filman SC. Tzilla Elrad et Aksit M., "Aspect-Oriented Software Development", Addison-Wesley Professional, ISBN: 0-321-21976-7, October 2004.

[Rus 00] C. Russo Dos Santos et al., "Metaphor-Aware 3D Navigation", In: Proceedings of *the IEEE Symposium on Information Visualization*, pp. 155–165, 2000.

[San 06] K. Sanders and A. van Dam, "Object-oriented programming in Java: a graphical approach", Addison Wesley, 2006.

[Sca 89] David Scanlan, "Structure Flowcharts Outperform Pseudocode: An Experimental Comparison", *IEEE Software*, Vol. 6, No. 4, pp. 28–36, September 1989.

[Sch 06] H. J. Schulz et H. Schumann, "Visualizing Graphs - a Generalized View", In: Proceedings of *the IEEE Conf. on Information Visualization*, pp. 166–173, 2006.

[Sco 95] B. Scott, Steinman et G. Carver. Kevin, "Visual programming with Prograph CPX", Manning Publications Co., Greenwich, CT, USA, 1995.

[Sev 01] G. Sevitsky, W. De Pauw, et R. Konuru, "An information exploration tool for performance analysis of java programs", In: Proceedings of *the Int'l Conf. on Technology of Object-Oriented Languages and Systems* (TOOLS), IEEE, Vol. 38, pp. 85–101, 2001.

[Shu 88] Nan C. Shu, "Visual programming", Van Nostrand Reinhold, 1988.

[Shu 89] Nan C. Shu, "Visual programming: Pespectives and Approaches", *IBM System Journal*, Vol. 28, No. 4, 1989.

[Som 04] Ian Sommerville, "Software Engineering", 7th edition, Pearson, Addison-Wesley, 2004,

[Sol 06] Solveig Vidal, "Visualisation de l'information : un panorama d'outils et de méthodes", CNRS. Publication, Mai 2006.

[Sta 98] J. T. Stasko, J. Domingue, M. H. Brown et B. A. Price, "Software Visualization - Programming as a Multimedia Experience", MIT Press, 1998.

[Ste 06] F. Steimann, "The paradoxical success of aspect-oriented programming", In: Proceedings of *the 21st Annual ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA), pp.481–497, 2006.

[Sto 05] M. A. Storey, "Theories, methods and tools in program comprehension", In: Proceeding of *the 13th International Workshop on Program Comprehension* (IWPC), pp. 181–191, 2005.

[Tak 05] Shimomura Takao, "Visual design and programming for Web applications", *Journal of Visual Languages and Computing* (JVLC), Vol. 16, No. 3, pp. 213–230, 2005.

[Tan 90] S. Tanimoto, "VIVA: a visual language for image processing", *Journal of Visual Languages and Computing* (JVLC), Vol. 2, No. 2, pp. 127–139, June 1990.

[Tarr 99] P. Tarr, H. Ossher et S.M. Sutton, "N Degrees of Separation: Multidimensional Separation of Concerns", In: Proceedings of *the Int'l Conf. on Software Engineering* (ICSE), Mai 1999.

[Wal 99] R. J. Walker, E. L. A. Baniassad, G. C. Murphy, "An initial assessment of aspect-oriented programming", in: Proceedings of *the 21st Int'l Conf. on Software Engineering* (ICSE'99), IEEE Computer Society Press, pp. 120–130, 1999.

[Wet 07] R. Wettel et M. Lanza, "Visualizing software systems as cities", In: Proceedings of *the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis* (VISSOFT), IEEE Computer Society, pp. 92–99, 2007.

[Wet 08a] R. Wettel et M. Lanza, "Codecity: 3D visualization of large-scale software", In: Proceedings of *ICSE Companion'08: Companion of the 30th ACM/IEEE Int'l Conf. on Software Engineering*, pp. 921–922, 2008.

[Wet 08b] R. Wettel et M. Lanza, "Visually Localizing Design Problems with Disharmony Maps", In: Proceedings of *the 4th ACM Symposium on Software Visualization*, 2008.

[Wet 11] R. Wettel, M. Lanza et R. Robbes. Software systems as cities, "A controlled experiment", In: Proceedings of *the Int'l Conf. on Software Engineering* (ICSE), Vol. 11, 2011.

[Whi 02] K. N. Whitley, "Visual programming languages and the empirical evidence for and against", *Journal of Visual Languages and Computing* (JVLC), Vol. 8, pp. 109–142, 2002.

[Wis 99] U. Wiss et D.A. Carr, "An Empirical Study of Task Support in 3D Information Visualizations", In: Proceedings of *the Int'l Conf. on Information Visualization*, 1999.

[Xen 00] M. Xenos, D. Stavrinoudis, K. Zikouli et D. Christodoulakis, "Object-Oriented Metrics: A Survey", In: Proceedings of *Federation of European Software Measurement Associations*, pp. 1–10, 2000.

[Yan 03] H.Y. Yang et H. Graham, "Software Metrics and Visualisation", Technical report, Univ. of Auckland, 2003.

[Yen 96] N. Ye et G. Salvendy, "Expert-novice knowledge of computer programming at different levels of abstraction", *Ergonomics*, Vol. 39, No. 3, pp. 461–481, 1996.

[Zha 07] K. Zhang, "Visual languages and applications", Springer-Verlag US, 2007.

[Zha 08] K. Zhang, J. Kong et J. Cao, "Visual software engineering", Wiley Encyclopaedia of Computer Science and Engineering, 2008.

[Zhe 11] D. Zhengyan, "Aspect Oriented Programming Technology and the Strategy of its Implementation", In: Proceedings of *the Int'l Conf. on Intelligence Science and Information Engineering* (ISIE), pp. 457–460, 2011.

2. Références Web (Techniques)

[W1] AspectJ Project home site, URL: http://www.aspectj.org/

[W2] AspectJ Development Tools (AJDT) home site, URL: http://www.eclipse.org/ajdt

[W3] AspectJ Team. 2004. AspectJ Programming Guide, URL: http://www.eclipse.org/aspectj

[W4] Alice Project home site, URL: http://www.alice.org/

[W5] Limnor Studio home site, URL: http://www.limnor.com/

[W6] Tersus Project home site, URL: http://www.tersus.com/

[W7] Eclipse Modeling Framework (EMF) site, URL: http://www.eclipse.org/modeling/emf/

[W8] Graphical Editing Framework (GEF) site, URL: http://www.eclipse.org/gef/

[W9] Acceleo Project home site, URL: http://www.eclipse.org/acceleo

[W10] Eclipse Project home site, URL: http://www.eclipse.org

[W11] JSON Project home site, URL: http://www.json.org

[W12] Graphical Modeling Framework (GMF) site, URL: http://www.eclipse.org/gmf/

[W13] X-Ray tool, URL: http://xray.inf.usi.ch

[W14] Citylyzer tool, URL: http://atelier.inf.unisi.ch/~biaggia/citylyzer/

[W15] CodeCity, URL: http://www.inf.unisi.ch/phd/wettel/codecity.html

[W16] SolidFX, URL: http://www.solidsourceit.com/products/SolidFX-static-code-analysis.html

[W17] AspectWerkz, URL: http://aspectwerkz.codehaus.org/

[W18] CaesarJ, URL : http://caesarj.org/

[W19] JBoss AOP, URL : http://jbossaop.jboss.org/

[W20] Java Aspect Components (JAC), URL : http://jac.ow2.org/

[W21] Software Engineering Body of Knowledge (SWEBOK) Project home site, URL: http://www.swebok.org/

[W22] Obeo, Acceleo User Guide, URL: http://www.acceleo.org/

[W23] Object Management Group (OMG), XMI specification, URL: http://www.omg.org/spec/XMI/

[W24] OpenGL, URL: http://www.eclipse.org/swt/opengl/

[W25] AOSD community home page, URL: http://aosd.net/

[W26] Software Visualization, URL: http://softvis.wordpress.com

[W27] Visual Language Research Bibliography, URL: http://web.engr.oregonstate.edu/~burnett/vpl.html

[W28] The Larch Environment, URL: http://www.larchenvironment.com/

ANNEXE A

CONSTRUCTIONS ET NOTATIONS GRAPHIQUES PROPOSÉES

Cette annexe présente les notations graphiques proposées avec les implémentations textuelles (les règles de transformations ACCELEO pour la génération des templates de code) correspondantes. Ces notations sont conçues pour représenter les concepts fondamentaux et mécanismes offerts par le paradigme "orienté-aspects" implémentés en JAVA & ASPECTJ avec leurs formalismes syntaxiques.

The ASPECTJ language consists of the ordinary JAVA constructs as well as the ASPECTJ crosscutting constructs.

- **Static Aspect-oriented language features:**
  - ***Aspect constructs*** (i.e. Advice, pointcut, introduction, declaration..),
  - ***Class constructs*** (i.e. Method variable try catch...),
  - ***Inheritance*** (implements, extends),
  - ***Structured statements*** (i.e.
    - Structure: Program Class Interface Aspect Resource ...
    - Conditions: switch case
    - Loops:
    - Documenting:
  - ...),

- **AspectJ crosscutting constructs:**
  - ***Common crosscutting constructs*** (join point, pointcut, and aspect),
  - ***Dynamic crosscutting construct*** (advice), and
  - ***Static crosscutting constructs*** (inter-type declarations and weave-time declarations).
    - Inter-type declaration (ITD) (also referred to as introduction) : member introduction, method introduction…
    - Weave-time declarations (weave-time warnings and errors)

AO-Basic Constructs & Features		Graphical Notations (iconic elements)	Syntactic Formalisms of Static AO-Basic Programming Constructs and Features (e.g., ASPECTJ) ACCELEO Rules of Model Transformation (i.e., ACCELEO templates)
		Before Advice	`<advice> ::= [returnType] <adviceType> "(" [formals] ")" [afterQualifier] [throwsClause] ":" <attachedpointcut> "{"<adviceBody>"}"` **returnType** ::= TypeOrPrimitive ; (applies only to around advice) **<adviceType>** ::= before | after | around **afterQualifier** ::= throwing | returning (applies only to after advice) **[formals]** as a Java parameter list **<attachedPointcut>** ::= <pointcut> | <namedPointcut> | <abstractNamedPointcut> **<adviceBody>** as a Java method body (with a specific method: proceed())

Aspect Constructs

Advice

```
[template public genAspectElement(aPackageName:String, aAspect :
AspectElement)]
[if((notaAspect.name.oclIsUndefined())andaAspect.name.trim().size()>0)]
[file
(aPackageName.substituteAll('.','/').concat('/').concat(aAspect.name.trim()).c
oncat('.aj'),false)]
[genPackageName(aPackageName)/]
[getImportBlock(aAspect)/]
[getComment(aAspect.description,null)/]
[getAnnotation(aAspect.annotation)/]
[getAccess(aAspect.access)/][if(aAspect.final)] final[/if]
aspect [aAspect.name/][if(aAspect.extendsElements->notEmpty())]
extends[genAspectExtendsElement(aAspect)/][/if]
[if(aAspect.implementsElements->notEmpty())]
implements [genAspectImplementsElement(aAspect)/][/if]{
[getAspectBody(aAspect)/]}
[/file][/if]
[/template]
```

...

```
BeforeAdvice ::= before "("[Formals]")"
                   [throws TypeList] ":"
                   Pointcut "{" [adviceBody] "}"
```

<typeList> ::= Type {"," Type (Java class types)}

Around Advice

```
AroundAdvice ::= around "("[Formals]")"
                   [throws TypeList] ":"
                 Pointcut "{" [adviceBody] "}"
```

<typeList> ::= Type {"," Type (Java class types)}

After Advice

```
AfterAdvice ::= after "("[Formals]")"
                   [throws TypeList] ":"
                 Pointcut "{" [adviceBody] "}"
```

<typeList> ::= Type {"," Type (Java class types)}

Special Forms

thisJoinPoint

reflective information about the join point.

thisJoinPointStaticPart

the equivalent of thisJoinPoint.getStaticPart(), but may use fewer resources.

thisEnclosingJoinPointStaticPart

the static part of the join point enclosing this one.

proceed (Arguments)

only available in around advice. The Arguments must be of the same number and type as the parameters of the advice.

Pointcut

Anonymous Pointcut

<pointcut> ::= <designator>{"&&" | "||" <designator> }";"

<designator> ::= <designatorIdentifier> (<signaturePattern> | <typePattern> | <pointcut> | <namedPointcut>)
<designatorIdentifier> ::= call | execution | initialisation | handler | get | set | this | target | args | cflow | cflowbelow | staticInitialization | within | withincode |if | adviceexecution | perinitialization
<signaturePattern> as Java operation signatures

...

Named Pointcut

```
<namedPointcut> ::= [<accesType>] pointcut <pointcutName> "("
[formals] ")" ":" <pointcut> abstractNamedPointcut ::= abstract
[<accesType>] pointcut <pointcutName> "(" [formals] ")" ";"
```

<accessType> ::= [public | protected | private]
<pointcutName> ::= { <identifier> }
<parameters> ::= { <identifier> <type> }
<identifier> ::= letter { letter | digit }
<type> ::= defined valid Java type
formals ::= {formal (as Java formal parameter) [","]}

...

<table>
<tr><td rowspan="6"></td><td rowspan="2"></td><td rowspan="2">Primitive Pointcuts</td><td>General form:
call(MethodPat)
call(ConstructorPat)
execution(MethodPat)
execution(ConstructorPat)
initialization(ConstructorPat)
preinitialization(ConstructorPat)
staticinitialization(TypePat)
get(FieldPat)
set(FieldPat)
handler(TypePat)
adviceexecution()
within(TypePat)
withincode(MethodPat)
withincode(ConstructorPat)
cflow(Pointcut)
cflowbelow(Pointcut)
if(Expression)
this(Type | Var)
target(Type | Var)
args(Type | Var , ...)</td><td>Where:

MethodPat is :
[ModifiersPat] TypePat [TypePat .] IdPat (TypePat | .., ...)
[throws ThrowsPat]

ConstructorPat is :
[ModifiersPat] [TypePat .] new (TypePat | .. , ...)
[throws ThrowsPat]

FieldPat is :
[ModifiersPat] TypePat [TypePat .] IdPat

TypePat is one of :
IdPat [+] [[] ...]
! TypePat
TypePat && TypePat
TypePat || TypePat
(TypePat)</td></tr>
<tr><td></td></tr>
<tr><td rowspan="6">Introduction</td><td rowspan="2">Introduce Concrete Method</td><td>ConcreteInterTypeMethod ::= <modifiers> <returnType>
<targetType>"."<id>"
("[formals]^')"[throwsClause]"{"<methodBody>"}"

<modifiers> ::= [public | protected | private] (as Java modifiers)
<returnType> any legal AspectJ type. (as Java)
<targetType> any legal AspectJ type. Defines the type to add the feature / member to.
<id> ::= letter {letter | digits} (as a Java identifier. Defines the name of the added method.)
formals ::= {formal (as Java formal parameters) [","]}
throwsClause (as Java throws clause)
<methodBody> (as Java method body)</td></tr>
<tr><td>...</td></tr>
<tr><td>Introduce Abstract Method</td><td>AbstractInterTypeMethod ::= abstract <modifiers> <returnType>
<abstractTargetType>"."<id>"("[formals]^')"[throwsClause] ";"

<abstractTargetType> any legal AspectJ type that must be an interface or an abstract class. Defines the type to add the member to.</td></tr>
<tr><td>...</td></tr>
<tr><td>Introduce Constructor</td><td>InterTypeConstructor ::= <modifiers><constructorTargetType>"." New
"("[formals]'^'JthrowsClause]"{"<constructorBody>"}''

<constructorTargetType> any concrete or abstract class</td></tr>
<tr><td>...</td></tr>
<tr><td rowspan="2">Introduce Field</td><td>InterTypeField ::= <modifiers><typeOfField><targetType>"." <id>
[fieldInitialization] ";"

<typeOfField> any any legal AspectJ type.
fieldInitialization ::= "=" Expression ";"</td></tr>
<tr><td>...</td></tr>
<tr><td></td><td>Declare Parents</td><td>Déclaration d'héritage :
SuperclassDeclaration ::= declare parents ":" <typePattem> extends
<typeList> ";"

<typePattern> AspectJ type pattern (Java class types)
<typeList> ::= Type {"," Type (Java class types)}

Déclaration de réalisation :
InterfaceDeclaration ::= declare parents ":" <typePattern>
implements <typeList> ";"</td></tr>
</table>

<table>
<tr><td rowspan="7"></td><td rowspan="7">Declaration</td><td></td><td>...</td></tr>
<tr><td>Declare Precedence</td><td>PrecedenceDeclaration ::= declare precedence ":" <typePatternList> ";"
<typePatternList> ::= typePattern{","typePattern (AspectJ aspect type)}</td></tr>
<tr><td></td><td>...</td></tr>
<tr><td>Declare Exception</td><td>SoftenDeclaration ::= declare soft ":" <exceptionType> ":" <staticallyDeteminablePointcut> ";"
declare soft : Exception : somePointcut();
<exceptionType> as java exception type
<staticallyDeteminablePointcut> ::= <pointcut> | <namedPointcut> (pointcuts with designators this, target, args, cflow, cflowbelow and if are non static) .</td></tr>
<tr><td></td><td>...</td></tr>
<tr><td>Declare Custom Compilation</td><td>MessageDeclaration ::= declare [error | warning] ":"<staticallyDeteminablePointcut> ":" <message> ";"
<staticallyDeterminablePointcut> ::= ; (a statically determinable pointcut as described in the previous section)
<message> ::= ;(a Java string literal)</td></tr>
<tr><td></td><td>...</td></tr>
<tr><td rowspan="3">Class Constructs</td><td colspan="2">Method | Constructor</td><td>Variable</td></tr>
<tr><td colspan="2">Try | Catch</td><td></td></tr>
<tr><td colspan="3">NB: For detailed description see The Java Language Specification ...</td></tr>
<tr><td rowspan="4">Structured Statements</td><td rowspan="4">Structure</td><td>Program</td><td>...</td></tr>
<tr><td>Package</td><td>...</td></tr>
<tr><td>Class</td><td>class ::= <access> [abstract] [privileged] [static] class <identifier> <classIdentifier> <instantiation>
{ <classBody> }
<access> ::= [PUBLIC | abstract | FINAL] legal for a Java Class.
<identifier> ::= letter {letter | digit} is a type that is the name of the class.
<classIdentifier> ::= [DOMINATES] [EXTENDS] [IMPLEMENTS]
<instantiation> ::= [SUPERCLASS | INTERFACES]
Superclass is a type which is the name of a class.
Interfaces is a set of types each of which is the name of an interface.
<classBody> ::= {<classFeature>}
<classFeature>::= <features> | <Attributes > | <Methods> | <Constructors>
Attributes is a set of attributes.
Methods is a set of methods.
Constructors is a set of constructors.</td></tr>
<tr><td></td><td>...</td></tr>
</table>

<table>
<tr><td rowspan="5"></td><td rowspan="3"></td><td>Interface</td><td>

```
interface ::= <access> [abstract] [privileged] [static] interface
<identifier> <interfaceIdentifier> <instantiation>

{ <interfaceBody> }
```

<identifier> ::= letter {letter | digit} is a type that is the name of the interface.
<interfaceIdentifier> ::= [DOMINATES] [EXTENDS] [IMPLEMENTS]
<instantiation> ::= [SUPERCLASS | INTERFACES]
Superclass is a type which is the name of an interface.
<interfaceBody> ::= {<interfaceFeature>}
<interfaceFeature>::= <features> | < Attributes > | <Methods>
Attributes is a set of attributes.
Methods is a set of methods such that ∀ M ∈ Methods, “abstract” ∈ Modifier(M)</td></tr>
<tr><td colspan="2">…</td></tr>
<tr><td>Aspect</td><td>

```
aspect ::= <access>[abstract] [privileged] [static] aspect
<identifier> <classIdentifier> <instantiation>
{ <aspectBody> }
```

<access> ::= [PUBLIC | PROTECTED | PRIVATE]
<identifier> ::= letter {letter | digit}
<classIdentifier> ::= [DOMINATES] [EXTENDS] [IMPLEMENTS]
<instantiation> ::= [ISSINGLETON | PERTHIS | PERTARGET | PERCFLOW | PERFLOWBELOW]
<aspectBody> ::= {<aspectFeature>}
<aspectFeature> ::= <features> | <introduction> | <declareParents> | <declarePrecedence> | <pointcut> | <advice> | <declareSoft> | <customCompilation>
<features> représente une propriété classique : un attribut, une opération ou une méthode.</td></tr>
<tr><td>

```
[template public getAspectBody(aAspect : AspectElement)]
[if(aAspect.members->notEmpty())]
[for (aMember : AspectMember|aAspect.members)]
[getComment(aMember.description,aMember.note)/][getAnnotation(aMember.annotation)/][getAccess(aMember.access)/][getMemberDeclaration(aMember.declaration)/]
[getBaseType(aMember.type)/][getMemberName(aMember)/];
[/for]
[/if]
[for (aMethod : Method|aAspect.methods)]
[getComment(aMethod.description,aMethod.note)/][getAnnotation(aMethod.annotation)/][getOverride(aMethod.signature.declaration)/][getAccess(aMethod.signature.access)/][getDeclaration(aMethod.signature.declaration)/][getReturnType(aMethod.signature)/][getMethodName(aMethod)/]([getArgsElement(aMethod)/]) {
[for (s : Sequence|aMethod.sequences)]
[getSequence(s)/]
[/for]}
[/for]
[for (aAdvice : Advice|aAspect.advices)]
[getComment(aAdvice.description,aAdvice.note)/][getAnnotation(aAdvice.annotation)/][getOverride(aAdvice.signature.declaration)/][getAccess(aAdvice.signature.access)/][getDeclaration(aAdvice.signature.declaration)/][getReturnType(aAdvice.signature)/][getAdviceName(aAdvice)/]([getArgsElement(aAdvice)/]) {
[for (s : Sequence|aAdvice.sequences)]
[getSequence(s)/]
[/for]}
[/for]
...
[/template]
```

…</td></tr>
<tr><td rowspan="3">Conditions</td><td colspan="2">Switch | Case | Default</td></tr>
<tr><td colspan="2">If | Else | Else if</td></tr>
<tr><td colspan="2">…</td></tr>
<tr><td>Loops</td><td colspan="2">Iteration
…</td></tr>
</table>

<table>
<tr><td rowspan="3"></td><td rowspan="3">Documenting</td><td>Description /*</td><td>Discussion</td><td>Comment</td></tr>
<tr><td>Group</td><td>Source Link</td><td>…</td></tr>
<tr><td colspan="3">…</td></tr>
<tr><td rowspan="2">Inheritance</td><td>Implements</td><td colspan="3">…</td></tr>
<tr><td>Extends</td><td colspan="3">…</td></tr>
<tr><td colspan="5">…</td></tr>
</table>

ANNEXE B

ÉVALUATION PRELIMINAIRE DU PROTOTYPE HCODELESSAJ

Cette annexe consiste à présenter un rapport sur l'évaluation quantitative et qualitative de l'outil prototype "HcodelessAJ", ainsi qu'une étude comparative avec l'outil "AJDT" (AspectJ Development Tools).

1. Évaluation Empirique et Analyse (Pretest)

1.1 Evaluation Quantitative

Quantitative Assessment
The participants were asked to compare both tools used (AJDT & HCodelessAJ prototype) according to the following criteria: • **Training time (day):** the training period required to familiarize with AspectJ coding according only to one methodology: Text-Based or Hybrid Codeless method by means of its corresponding support tool. • **Development time (minutes):** the time that is required to perform the task of coding for each program. • **Compilation time (seconds, ms):** the required time to compile the created program. • **Miss-typing percentage (%):** the percentage of errors due to miss-typing the source-code. The general summary of assessment tables that show the obtained results besides its graphical representationsas is presented

Quantitative Assessment Tables & Graphical Representations

Training Time Table

Training Time Table *(in day)*

Participants	AJDT	HCodelessAJ
Participant n° 1	14	7
Participant n° 2	15	8
Participant n° 3	12	6
Participant n° 4	13	8
Participant n° 5	16	9
Participant n° 6	11	6
Participant n° 7	15	7
Participant n° 8	14	8
Participant n° 9	12	7
Participant n° 10	13	8
Participant n° 11	11	7
Participant n° 12	15	10
Participant n° 13	13	9
Participant n° 14	11	6
Participant n° 15	12	6
Participant n° 16	14	9
Average	**13**	**8**

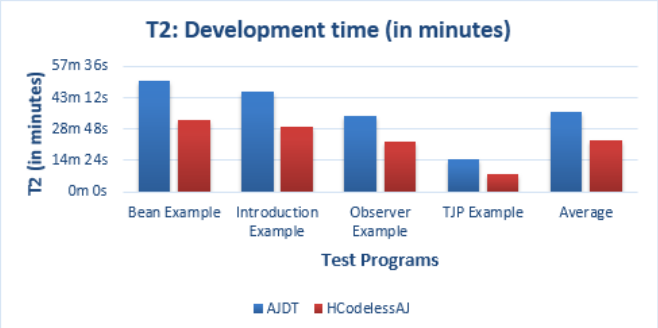
Graphical Representation

Development Time Table

Development Time Table *(in minutes)*

AspectJ Programs	AJDT	HCodelessAJ
Bean Example	51m 0s	33m 0s
Introduction Example	46m 0s	30m 0s
Observer Example	35m 0s	23m 0s
TJP Example	15m 0s	8m 0s
Average	**36s, 45ms**	**23m 30s**

Graphical Representation

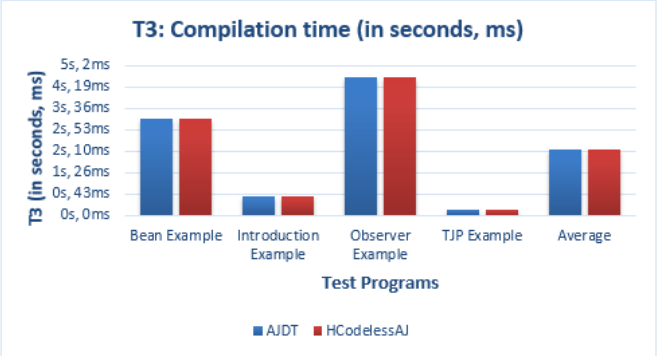


Compilation Time Table

Compilation Time Table *(in seconds, ms)*

AspectJ Programs	AJDT	HCodelessAJ
Bean Example	3s, 16ms	3s, 16ms
Introduction Example	0s, 39ms	0s, 39ms
Observer Example	4s, 40ms	4s, 40ms
TJP Example	0s, 12ms	0s, 12ms
Average	**2s, 12ms**	**2s, 12ms**

Graphical Representation

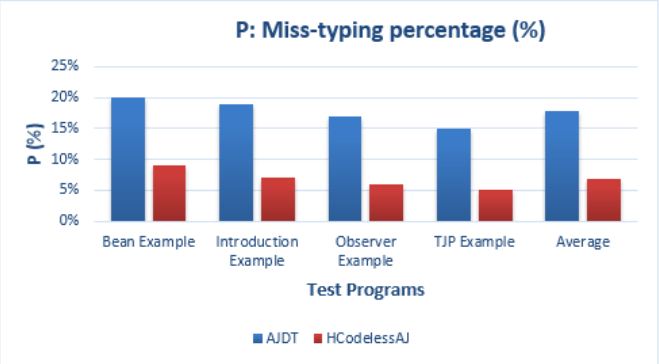


Miss-typing Percentage Table

Miss-typing percentage Table *(%)*

AspectJ Programs	AJDT	HCodelessAJ
Bean Example	20%	9%
Introduction Example	19%	7%
Observer Example	17%	6%
TJP Example	15%	5%
Average	**18%**	**7%**

Graphical Representation



1.2 Evaluation Qualitative

Qualitative Assessment: Measure of Performance of the Prototype "HCodelessAJ"

In addition to the quantitative assessment, we measured the performance and contribution of the prototype relative to some criteria. The practical experience suggests that the following are the most important requirements:

- **Expressiveness:** we should impose few restrictions as possible on the step of coding when we express programs.
- **Usability:** defined as the extent to which a tool can be used by specified users to achieve specified goals in a specified context of use, with a little of technical background and training, as well as less time to accomplish a particular task (easy to use and to program). It is defined with five main attributes according to Nielsen's model: *Efficiency* , *Learnability* , *Memorability* , *Error Handling* , and *User Satisfaction* .
- **Computational Performance:** the time required to compile and execute the program code.
- **Productivity:** effort required to develop and deploy programs. It is defined with five main attributes: *Flexibility, Scalability, Reusability, Comprehensibility* and *Maintainability* .

In the rating of statements, the answers were awarded between **0 and 4 points** for how close they came to the right statement. The questions were answered on a scale of ***zero points*** for ***nothing recognizable at all*** to ***four points*** for exactly ***right statement*** .

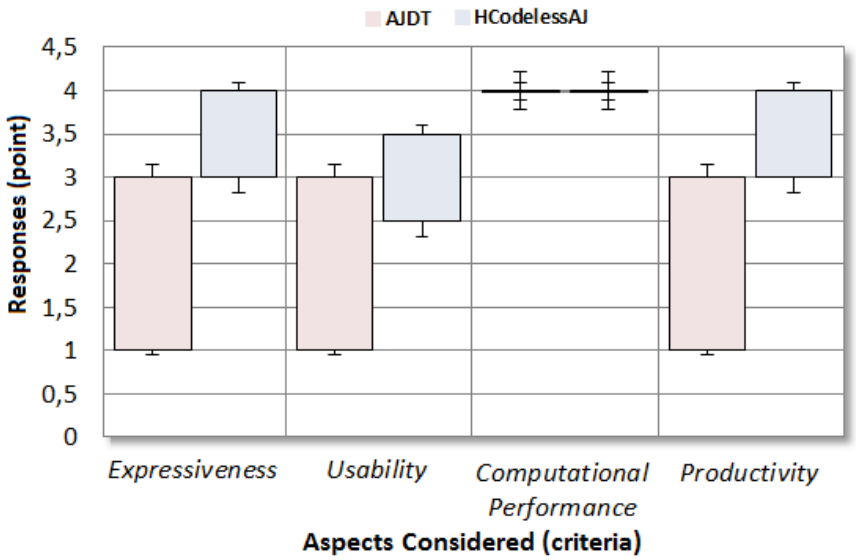
A comparison of "AJDT" and "HCodelessAJ" prototype according to the criteria defined

Aspects Considered (criteria)	AspectJ-Based Tool Support	
	AJDT *(point)*	**HCodelessAJ** *(point)*
Expressiveness	0 / 1 / 2 / 3 / 4	
Usability Efficiency, Learnability, Memorability, Error Handling, User Satisfaction		
Computational Performance		
Productivity Flexibility, Scalability, Reusability, Comprehensibility, Maintainability		

Summary of Qualitative Assessment Responses & Graphical Representations

The results of the survey are summarized in the following table.

Aspects Considered (criteria)	AspectJ-Based Tool Support			
	AJDT *(point)*		HCodelessAJ *(point)*	
	Min	Max	Min	Max
Expressiveness	**1**	**3**	**3**	**4**
Usability	**1**	**3**	**2,5**	**3,5**
Computational Performance	**4**	**4**	**4**	**4**
Productivity	**1**	**3**	**3**	**4**



2. Sondage Basé-Questionnaire (Posttest)

POST-EXPERIMENT QUESTIONNAIRE

Participant Number : 1
Name & Surname :

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

Dear study participant

The purpose of thi eriment questionnaire is to assess the usability of HCodelessAJ prototype. By complet re enabling us to identify and remedy any shortcomings, with the aim of enhancing it.

All the questions e sed answers according to some statements. For each of these statements, please che g of your choice.

The questionnaire this study contained 26 questions, shown in the table below, arranged in categorie *Experience, Methodology of Coding and HCodelessAJ Prototype Satisfaction, Performed Task, Suitability for the Task, Suitability for Learning, Error Tolerance* and *Comments.*

Note that your answers to these questions would help us greatly in our analysis of this questionnaire-based survey.

The Questionnaire-Based Survey

ID	Questions

Subject Experience (Background Information)				
Q1	How do you judge your programming experience?	**Novice**	**Medium**	**Advanced**
		○	○	○
Q2	How do you judge your AOP/AspectJ programming experience through AJDT?	**Novice**	**Medium**	**Advanced**
		○	○	○
Q3	How do you judge your Eclipse IDE experience?	**Novice**	**Medium**	**Advanced**
		○	○	○

Methodology of Coding and HCodelessAJ Prototype Satisfaction					
Q4	What do you think about the use of intuitive graphical icons in this novel coding methodology?				
	Highly Interesting	**Interesting**	**Little of Interest**	**Not Interest**	**No Opinion**
	○	○	○	○	○
Q5	Were the prototype functionalities easy to use?				
	Yes, Highly		**Yes, Somewhat**		**No**
	○		○		○
Q6	Is the user interface friendly, clear and easy to understand?				
	Excellent	**Good**	**Satisfactory**	**Needs Improvement**	**No Opinion**
	○	○	○	○	○
Q7	Did you need any external help during the experimentation?				
	Yes			**No**	
	○			○	
Q8	How would you rate this prototype in terms of functionalities?				
	Excellent	**Good**	**Satisfactory**	**Needs Improvement**	**No Opinion**
	○	○	○	○	○
Q9	How would you rate this prototype as compared to AJDT?				
	Excellent	**Good**	**Satisfactory**	**Needs Improvement**	**No Opinion**
	○	○	○	○	○
Q10	Would you recommend this prototype for a training purpose to students?				
	Yes, Highly		**Yes, Somewhat**		**No**
	○		○		○
Q11	How do you assess the usefulness of this methodology for the future of AOP?				
	Very Useful	**Somewhat Useful**		**Not Useful**	**No Opinion**
	○	○		○	○

Performed Coding Task				
Q12	Was the task of constructing program visual model simple?			
	Excellent	**Good**	**Needs Improvement**	**No Opinion**
	○	○	○	○
Q13	Is the number of operations (steps) to achieve the task appropriate?			
	Agree	**Disagree**	**Strongly Disagree**	**No Opinion**
	○	○	○	○
Q14	Were the composition and connection between the visual objects and the parameters easy to perform?			
	Agree	**Disagree**	**Strongly Disagree**	**No Opinion**
	○	○	○	○
Q15	Are there any difficulties to find the needed visual objects (i.e. graphical icons) for each concept?			
	Yes, Absolutely	**Somewhat**	**No, Strongly**	
	○	○	○	

Suitability for the Coding Task					
Q16	How would you rate the effect of the arrangement of the concepts on views on the acceleration of the coding process?				
	Excellent	**Good**	**Satisfactory**	**Needs Improvement**	**No**
	○	○	○	○	○
Q17	Did you find all the information and commands needed at coding process in a given view?				
	Excellent	**Good**	**Satisfactory**	**Needs Improvement**	**No**
	○	○	○	○	○
Q18	How would you rate the easiness of adapting the visual model created for building a new model?				
	Excellent	**Good**	**Satisfactory**	**Needs Improvement**	**No**
	○	○	○	○	○
Q19	How would you rate the easiness of generating templates of code, importing and exporting the visual models created?				
	Excellent	**Good**	**Satisfactory**	**Needs Improvement**	**No**
	○	○	○	○	○
Q20	What do you think about the moving back and forth during the coding process within the visual model designer?				
	Excellent	**Good**	**Satisfactory**	**Needs Improvement**	**No**
	○	○	○	○	○

Suitability for Learning			
Q21	Is the time to learn the functionalities of the prototype appropriate as compared to AJDT?		
	Decreased	**Stayed the Same**	**Increased**
	○	○	○
Q22	Do you think the terms and concepts used are clear and unambiguous? (i.e. graphical icons have a clear meaning)		
	Yes	**Somewhat**	**No**
	○	○	○
Q23	Did you need to remember numerous details for using the prototype properly?		
	Yes, Highly	**Somewhat**	**No**
	○	○	○

Error Tolerance				
Q24	How would you rate the errors of system and Eclipse IDE (e.g. crashes) that occurred during the coding process?			
	Often	**Sometimes**	**Rarely**	**Not at all**
	○	○	○	○
Q25	How would you rate the effort dedicated to correcting mistakes? (e.g. Easily undo the last operation when making mistakes).			
	Yes, Important	**Medium**	**Negligible**	
	○	○	○	
Q26	Do you perceive the error messages as helpful?			
	Very Helpful	**Somewhat Helpful**	**Not Help at All**	
	○	○	○	

Comments
▪ What is your overall opinion of HCodelessAJ prototype? ▪ Do you have any comments about how HCodelessAJ prototype affects the coding process?

Thank you very much for your help.

ANNEXE C

GLOSSAIRE DE TERMINOLOGIE DU GENIE LOGICIEL

L'explication des termes techniques employés dans ce document est regroupé par ordre alphabétique dans cette annexe. On donne la terminologie, leur signification en français et /ou une équivalence en anglais lorsque jugée nécessaire.

1. Tableau des Terminologies

Artifact

Un **Artefact** est le résultat de toute activité dans le cycle de vie d'un logiciel tels que les besoins, modèle d'architecture, spécifications de conception, le code source et scripts de test. Un artefact est un élément d'information utilisé ou produit par le processus de développement de logiciels.

Aspect

Un **Aspect** est une entité logicielle, une abstraction qui encapsule une préoccupation (fonctionnalité) transversale à une application ; dont la particularité est de s'appliquer à un ensemble de classes entrecoupantes pour les classes fonctionnelles. Un aspect est composé d'une expression de point de coupure (pointcut) et d'un greffon ou consigne (advice). L'implémentation des aspects va s'entrelacer avec le reste de l'implémentation **[Kic 01a].**

Benchmark

Un **Test de performance**, c'est un ensemble de tests utilisés pour comparer les performances des outils, méthodes ou techniques.

(1) A standard against which measurements or comparisons can be made. (2) A problem, procedure, or test that can be used to compare systems or components to each other or to a standard as in (1) **[IEEE STD 1471-00].**

Case study

Une **étude de cas**, c'est une technique de recherche où vous identifier les facteurs clés susceptibles d'influencer le résultat d'une activité, puis documenter l'activité : ses entrées, contraintes, ressources et sorties.

Case studies usually look at a typical project, rather than trying to capture information about all possible cases; these can be thought of a "research in the typical". Formal experiments, case studies and surveys are three key components of empirical investigation in software engineering **[Fenton&Pfleeger 96].**

Experiment

Une **expérimentation** est un procès qui est menée afin de vérifier une hypothèse définie dans un milieu contrôlé où les facteurs les plus critiques peuvent être contrôlés ou examinés **[ISERN].**

Forward Engineering

Le **redéveloppement** est le processus traditionnel permettant de passer des abstractions de haut niveau et de la conception logique indépendante de l'implémentation à la mise en œuvre physique d'un système **[Chikofsky&Cross 90].**

Framework

Un **cadre d'applications** ou **cadriciel**, c'est un espace de travail modulaire. C'est un ensemble de bibliothèques et de conventions permettant le développement rapide de tout ou d'une partie d'un système logiciel (ou application). Il fournit suffisamment de

briques (fonctions) logicielles et impose suffisamment de rigueur pour pouvoir produire une application aboutie et facile à maintenir.

A framework is a reusable design of all or part of a software system described by a set of abstract classes and the way instances of those classes collaborate **[Roberts&Johnson 96]**.

A framework is anything that can be adapted or extended via systematic extension or configuration **[Stahl&Volter 06]**.

Program Comprehension

La **compréhension des programmes** est la tâche de reconstruction de la conception abstraite du système, une partie ou la totalité, à partir de son code source. Elle peut être vue comme le processus qui extrait, à partir du programme, une connaissance conceptuelle à un haut niveau d'abstraction appelée les concepts abstraits du programme. Cette connaissance est utilisée par plusieurs activités de redéveloppement de logiciel tels que la maintenance, le débogage, la réutilisation, la migration, etc.

Metaphor

Une **métaphore** est une analogie qui est une représentation graphique d'une entité abstraite ou d'un concept dans le but de transférer des propriétés du domaine de la représentation graphique à celui de l'entité abstraite ou du concept **[Dos 02].**

Model-Driven Architecture (MDA)

MDA est une approche basée sur les modèles et un ensemble de standards (UML, MOF, CWM, XML et IDL) du consortium **OMG (Object Management Group) [OMG 04] [Kle 03] [Béz 05].**

Model-Driven Engineering (MDE)

L'**Ingénierie Dirigée par les Modèles (IDM)**, est une nouvelle approche en pleine expansion qui suscite un intérêt grandissant dans la communauté du génie logiciel. A l'origine, l'initiative **MDA** faite par le consortium **OMG** en novembre 2000. Il s'agissait alors de se baser sur **UML** pour rendre la description des systèmes indépendante des plateformes, conduisant ainsi à une meilleure portabilité, interopérabilité et réutilisation.

C'une démarche ouverte plus générale que **MDA**, prend en charge plusieurs autres espaces technologiques afin de les harmoniser. **MDA** est donc un cas particulier de **MDE**. **MDE** considère l'existence de modèles dans le sens large (les métaméta-modèles et les méta-modèles sont aussi des modèles) sur lesquels des opérations spécifiques peuvent être réalisées **[OMG 04] [Kle 03] [Béz 05].**

Model

A model is a simplified representation of a system or phenomenon with any hypotheses required to describe the system or explain the phenomenon, often mathematically. It is an abstraction of reality emphasizing those aspects that are of interest to someone. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality **[ISERN]**.

Model Transformation

The automatic generation of a target model from a source model, according to a set of **transformation rules**. A **transformation rule** is a description of how one or more constructs of the source model in the source language can be transformed into one or more constructs in the target language **[Kle 03].**

Paradigm

A point of view in which some principles, approaches, concepts, and even theories, have been stated uniformly. A set of assumptions about reality that, when applied to a particular situation, can be used as a guide for action. For example, the Quality Improvement Paradigm **[ISERN]**.

Platform

A set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented [Kle 03]

Reengineering

La **réingénierie** est l'examen et la modification du système logiciel pour corriger les fautes, améliorer la conception ou les performances, et modifier le produit pour satisfaire des exigences d'amélioration ou de changement **[Chikofsky&Cross 90].**

C'est l'altération d'un système logiciel dans le but de le reconstruire sous une nouvelle forme censée être meilleure. Même si elle peut être utilisée pour des changements mineurs, elle est souvent utilisée pour remplacer les logiciels âgés (Legacy systems).

Généralement, la **réingénierie** comprend certaine forme de **rétro-ingénierie** (afin de parvenir à une description plus abstraite), suivie par une certaine forme de **redéveloppement** ou de restructuration. Cela peut inclure des modifications en matière de nouvelles exigences non rencontrées par le système d'origine **[Chikofsky&Cross 90].**

Reverse Engineering

La **rétro-ingénierie** est une technique partant du produit fini et remontant progressivement toutes les méthodes de sa conception afin d'atteindre sa source pour l'analyser. C'est le processus d'analyse d'un système logiciel pour identifier ses composants et ses relations internes et créer des représentations dans d'autres formes, souvent à un plus haut niveau d'abstraction. Les résultats attendus de la rétro-ingénierie peuvent être : la reconstitution (voir la création) d'une documentation pour le logiciel, et la construction d'un modèle conceptuel **[IEEE STD 1471-00].**

Round-Trip Engineering (RTE)

The seamless integration between design and source code, between modeling and implementation. With round-trip engineering a programmer generates code from a design, changes that code in a separate development environment, and recreates the adapted design diagram back from the source code **[Demeyer et al. 99]**.

An iteration between modelling, generating code, changing that code and mapping this code back to the original model **[Demeyer et al. 00]**.

Software Maintenance

La **maintenance** est la modification d'un logiciel après son entrée en production, afin de corriger ses erreurs, d'améliorer ses performances et autres attributs, ou pour adapter le produit à son environnement **[IEEE STD 1471-00].**

Software Metric

Une **métrique logicielle** est une mesure issue des propriétés techniques ou fonctionnelles du logiciel **[Fen 03]**. Son but est de quantifier une caractéristique particulière telle que la complexité, la structure, le nombre de ressources utilisées ou la stabilité du système. Les métriques logicielles sont intéressantes parce qu'elles apportent des informations sur la qualité de la conception et contribuent ainsi à la gestion de cette qualité durant le processus de développement **[Kan 02] [IEEE STD 1471-00].**

Survey

A survey is a retrospective study of a situation to try to document relationships and outcomes. A survey is always done after an event has occurred. When performing a survey, you have no control over the situation at hand. That is, because it is a retrospective study, you can record a situation and compare it with similar ones. But you cannot manipulate variables as you do with case studies and experiments. Surveys try to poll what is happening broadly over large groups of projects: "research in the large" **[Fenton&Pfleeger 96]**.

SWEBOK

Software Engineering Body of Knowledge, est le corpus (la base) de connaissance du génie logiciel, chapeauté par l'organisme de standardisation des logiciels **IEEE** (Institute of Electrical and Electronic Engineer).

Site officiel du projet, URL : http://www.swebok.org/

2. Références des Terminologies

[Chikofsky&Cross 90] E.J. Chikofsky, J.H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software Engineering Journal, pp. 13-17, Jan. 1990.

[Fenton&Pfleeger 96] Norman E. Fenton, Shari Lawrence Pfleeger. Software Metrics: A Rigorous and Practical Approach. Thomson Computer Press, 1996.

[Roberts&Johnson 96] Don Roberts and Ralph Johnson: Evolving Frameworks: A Pattern-language for Developing Object-oriented Frameworks. PLoP '96 Proceedings, 1996.

[Demeyer et al. 99] S. Demeyer, S. Ducasse, S. Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. Proc. Int. Conf. UML, Springer-Verlag, 1999.

[Demeyer et al. 00] S. Demeyer, S. Ducasse, O. Nierstrasz. Finding Refactorings via Change Metrics. Proc. Int. Conf. OOPSLA 2000, ACM Press, October 2000.

[IEEE STD 1471-00] IEEE Standards Board. Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE STD 1471-00. September 2000.

[Dos 02] C. Dos Santos, Visualisation métaphorique tridimensionnelle de l'information, Ph.D. thesis, Ecole nationale supérieure des télécommunications (PARIS-ENST), 2002.

[Béz 04] J. Bézivin, Sur les principes de base de l'ingénierie des modèles, L'objet, 10/2004, pp. 145-156.

[OMG 04] Model-Driven Architecture, the official homepage,
URL: www.omg.org/MDA

[Béz 05] Bézivin J., Blay M., Bouzeghoub M., Estublier J., Favre J. M., Rapport de synthèse, Action Spécifique CNRS sur le MDA, janvier 2005.
URL: http://www.planetmde.org/as/rapport/AS-MDA-IDM-IDM-Synthese-1.1.pdf

[Stahl&Volter 06] T. Stahl, M. Volter: Model-Driven Software Development. Wiley, 2006.

[ISERN] International Software Engineering Research Network,
URL: http://www.iese.fhg.de/ISERN/

ANNEXE D

A PROPOS DE L'AUTEUR

Cette annexe regroupe les contributions scientifiques (publications et communications) produites par l'auteur de cette thèse. Elles sont classées par ordre chronologique inverse. Certaines de ces contributions sont disponibles à l'adresse : http://www.bentrad-sassi.sitew.com/.

1. Biographie de l'Auteur

M. Sassi BENTRAD est né en 1986 à Annaba (nord-est de l'Algérie). Il a obtenu son Baccalauréat (Option : Sciences de la Nature et de la Vie) en 2004. Il rejoint l'université de Badji Mokhtar-Annaba (UBMA) la même année pour suivre une formation LMD en Informatique (Option : ***I**ngénierie des **L**ogiciels **C**omplexes "**ILC**"*). En 2009, il a également obtenu son diplôme de Master 2 Recherche, avec mention très bien, puis accède, à partir de 2010, aux études de poste-graduation (3ième Cycle LMD) pour préparer une thèse de DOCTORAT en "*Ingénierie des Logiciels Complexes*" sous la direction du **Prof. Djamel MESLATI**.

Dans le cadre de ses activités scientifiques, il a participé à plusieurs conférences nationales et internationales (*ISPS*'11, *CIT*'11, *ACIT*'11, *JNIAK*'2012, *ICACIS*'12) et à plusieurs formations dans le domaine du génie logiciel.

À l'heure actuelle, il est membre de l'équipe **IPARAD** (**I**ntégration de **PARAD**igmes pour le développement de systèmes complexes) au sein du **L**aboratoire d'**I**ngénierie des **S**ystèmes **CO**mplexes (**LISCO**) de l'université de **Badji Mokhtar-Annaba** (**UBMA**).

✪ **Contact Information:**

Ph.D. Student **Sassi BENTRAD**
University of Badji Mokhtar-Annaba (UBMA)
(http://www.univ-annaba.dz/)
Computer Science Department, LISCO Laboratory
(http://lisco.univ-annaba.dz/)
P.O. Box 12, El-Hadjar, 23200 Annaba, Algeria
☎ :(+213) 5 54 03 78 18
✉ : sassi_bentrad@hotmail.fr
: http://www.bentrad-sassi.sitew.com/

ORCID: 0000-0002-7458-8121
(http://www.orcid.org/0000-0002-7458-8121)
ResearcherID: A-9442-2013
(http://www.researcherid.com/rid/A-9442-2013)
Scopus Author ID: 44461052600
(http://www.scopus.com/authid/detail.url?authorId=44461052600)

✪ **Objectifs ciblés par l'équipe:**

L'équipe **IPARAD** (***Intégration de PARADigmes pour le développement de systèmes complexes***) vise à étudier, maîtriser puis intégrer les divers paradigmes récents qui émanent du génie logiciel et des approches bio-inspirées. Ces paradigmes ont connu, durant ces dernières années, une évolution considérable qui promet un apport important dans le développement des systèmes complexes. C'est ainsi que de grandes firmes, comme IBM, investissent actuellement sur des approches d'intégration visant à rendre les systèmes autonomes et des chercheurs de divers horizons s'inspirent du vivant pour réduire la complexité des systèmes en les dotant de capacités leur permettant de s'adapter à leurs environnements. L'équipe investit le créneau des systèmes auto-évolutifs sous l'angle de l'ontogénèse biologique et tentera de l'utiliser dans des applications pratiques.

2. Contributions Scientifiques

2.1. Revues internationales avec peer-review

[J1] Bentrad S., Meslati D.: **"Visualizing and Analyzing the Structure of AspectJ Software under the Eclipse Platform"**, *IJSEIA - International Journal of Software Engineering and Its Applications*, Vol. 7, No. 3, May, 2013, pp. 353 – 376.
URL: http://www.sersc.org/journals/IJSEIA/vol7_no3_2013/33.pdf

ISSN: 1738-9984
Processus de reviewing: peer reviewed
Publisher: Science & Engineering Research Support society (SERSC)
Site Web: http://www.sersc.org/journals/IJSEIA/
Indexé par: SCOPUS, EBSCO, ProQuest, ULRICH, DOAJ, OpenJ-Gate, Cabell
SJR (SCImago Journal Rankings): 0,25 (2013)
SNIP (Source Normalized Impact per Paper): 1,532 (2013)

[J2] Bentrad S., Meslati D.: **"Visual Programming and Program Visualization – Towards an Ideal Visual Software Engineering System–"**, *ACEEE-IJIT - ACEEE International Journal on Information Technology*, vol. 1, no. 3, 2011, DOI: 01.IJIT.1.3.22, pp. 56 – 62.
URL: http://searchdl.org/index.php/journals/view/189

ISSN: 2158-012X (print); ISSN 2158-0138 (online)
Processus de reviewing: peer reviewed
Publisher: (ACEEE)
Site Web: http://ijit.theaceee.org , http://ijit.searchdl.org/
Indexé par : IET Inspec, EBSCO, ProQuest, DBLP - Computer Science Bibliography, …
SJR (SCImago Journal Rankings):
SNIP (Source Normalized Impact per Paper):

[J3] Bentrad S., Meslati D.: **"Toward a Hybrid Approach to Coding Aspect-Oriented Programs"**, *CAI - Computing and Informatics Journal*, ISSN 1335-9150.
URL : http://www.cai.sk/ojs/index.php/cai/index [**En cours de Reviewing**]

ISSN: 1335-9150
Processus de reviewing: peer reviewed
Publisher: Slovak Academy of Sciences (Institute of Informatics)
Site Web: http://www.cai.sk/ojs/index.php/cai/index
Indexé par: SCOPUS, ISI Current Contents® - Engineering, Computing and Technology, INSPEC, DBLP, Elsevier's Bibliographic Databases, SciSearch®, Research Alert®, CompuMath Citation Index®
SJR (SCImago Journal Rankings): 0,34 (2013)
SNIP (Source Normalized Impact per Paper): …

2.2. Conférences internationales avec comité de lecture

[C1] Bentrad S., Meslati D.: **"A Way to Introduce and Reduce Difficulties of Aspect-Oriented Coding through AspectJ under Eclipse Platform"**, In the *International Conference on Advanced Communication and Information Systems (ICACIS)*, Batna, Algeria, December 12-13, 2012.
URL: http://fac-sciences.univ-batna.dz/cs/icacis12/images/icacis2012%20program.pdf

[C2] Bentrad S., Meslati D.: **"Visual Programming and Program Visualization"**, In the *International Conference on Advances in Communication and Information Technology (CIT)*, Amsterdam, Netherlands, December 01-02, 2011, DOI: 02.CIT.2011.01.22, pp. 43 – 49, January 2012.
URL: http://searchdl.org/index.php/conference/view/287

[C3] Bentrad S., Meslati D.: **"Towards A New Way for Aspect-oriented Software Programming – VisPLAJ, a Pedagogic Visual Programming Language for AspectJ –"**, In the *International Arab Conference on Information Technology (ACIT)*, Riyadh, Saudi Arabia, December 11-14, 2011.
URL: http://www.nauss.edu.sa/acit/PDFs/f3177.pdf

[C4] Bentrad S., Meslati D.: **"2D and 3D Visualization of AspectJ Programs"**, In the *10th International Symposium on Programming and Systems (ISPS)*, 25-27 April 2011, Algiers, Algeria, ISBN: 978-1-4577-0905-0, DOI: 10.1109/ISPS.2011.5898888, pp. 183 – 190, June 2011.
URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5898888

2.3. Conférences nationales avec comité de lecture

[C5] Bentrad S., Meslati D.: **"Teaching and Learning Introductory Programming Concepts of AspectJ under Eclipse"**, *Premières Journées Nationales sur l'Informatique et ses Applications Khenchela (JNIAK)*, Khenchela, Algeria 29-30 Avril 2012.