Centre de Recherche Informatique de Montréal
550, rue Sherbrooke Ouest
Bureau 100
Montréal, QC
Canada H3A 1B9

ST50 – Projet de fin d'études

POM, un outil pour le calcul des métriques de qualité

Farouk ZAIDI - Département Génie Informatique
Filière ILC

Printemps 2004

Suiveur en entreprise : Mr Houari SAHRAOUI
Suiveur UTBM : Mr Abder KOUKAM

REMERCIEMENTS

Ce stage a été pour moi une expérience inoubliable. Les personnes que j'ai pu y rencontrer m'ont toujours aidé aussi bien dans le travail que dans mon intégration.

Je tiens tout d'abord à remercier Mr SAHRAOUI et Mr PETRENKO pour leur accueil et leur encadrement.

Je tiens ensuite à remercier Mr GUEHENEUC et Mr ROUATBI pour leur collaboration et leurs conseils. De même, j'adresse un grand remerciement à Mehdi, Kamal, Jean-François et Amine pour m'avoir mis en confiance dès le début et par la suite. C'était un plaisir de partager le bureau avec eux.

Je voudrais aussi remercier mes amis de l'UTBM avec qui j'ai gardé contact tout au long de mon stage.

Je tiens finalement à remercier Mr Abder KOUKAM pour la lecture de ce rapport. En tant que professeur de génie logiciel, il saura juger l'intérêt de ce projet de recherche et développement pour la communauté scientifique.

SOMMAIRE

GLOSSAIRE

UTBM : Université de Technologie de Belfort-Montbéliard

CRIM : Centre de Recherche Informatique de Montréal

Sun Microsystems : entreprise américaine dans le secteur de l'informatique – inventeur du langage Java

JDK (JAVA Development Toolkit): environnement de développement pour JAVA fourni par Sun. Celui-ci est très complet mais les fonctionnalités fournies, comme la compilation, se font en ligne de commande.

Machine virtuelle : programme assurant la traduction d'un programme JAVA en un code compréhensible par la machine. La machine virtuelle assure une exécution cohérente de programmes JAVA quelle que soit la plate-forme.

Eclipse : environnement de développement JAVA dont le code source est libre. Le projet a été initié par IBM. Chacun peut y ajouter des fonctionnalités en développant ou en utilisant des extensions commerciales ou libres.

IDE : abréviation pour environnement de développement

CVS : Concurrent Versions Systems, système de gestion concurrente de fichiers (codes sources, fichiers Word, etc.) pour le travail collaboratif

JUnit : framework de tests de regression, écrit par Erich Gamma et Kent Beck. Il est utilisé par les développeurs qui implantent des tests unitaires en JAVA. JUnit est fourni sous licence open-source.

SWT : Standard Widget Toolkit, API native permettant le développement d'interfaces graphiques pour la plate-forme Eclipse et d'une manière générale, indépendamment du système d'exploitation.

Métrique de qualité : métrique mesurant sur une échelle de valeurs numériques une propriété d'un programme

POM : Primitives, Operators, Metrics. API permettant le calcul de métriques sur des programmes. POM calcule des ensembles (de classes, interfaces, etc.) et applique des opérations sur ces ensembles.

PADL : API permettant de créer un modèle d'un programme.

CFParse : API permettant de faire le parsing de classes JAVA compilés. CFParse analyse le bytecode et crée un modèle du programme.

Bytecode : langage machine de la machine virtuelle. Le bytecode est composé de 256 instructions que la JVM utilise pour exécuter les programmes JAVA

Class files (ou fichiers .class) : fichiers JAVA compilés

Reverse engineering : Analyse des sources d'un programme pour recréer le modèle conceptuel associé.

eXtreme Programming (XP) : processus de développement, basé sur des cycles courts de développement. Le logiciel est délivré au fur et à mesure, avec des versions contenant uniquement les besoins nécessaires. XP privilégie l'interaction humaine, plutôt qu'un processus lourd de conception avec UML.

ANTLR : ANother Tool for Language Recognition, framework pour l'écriture de compilateurs ou de traducteurs à partir de grammaires de langages

INTRODUCTION

Le projet de fin d'études donne l'occasion à l'étudiant UTBM de réaliser un projet de niveau ingénieur en entreprise. Il lui permet aussi de mettre en pratique le savoir acquis pendant les trois ans de la formation d'ingénieur.

J'ai effectué mon stage ST50 au CRIM, le Centre de Recherche Informatique de Montréal. J'ai été accueilli dans l'équipe Analyse de systèmes distribués. J'y ai réalisé un projet sous la direction de Mr SAHRAOUI et Mr PETRENKO, dans le domaine de la qualité des logiciels objet. Ce projet a été effectué dans le cadre d'une collaboration entre le CRIM et le laboratoire de génie logiciel de l'université de Montréal dirigé par Mr SAHRAOUI, également chercheur associé au CRIM.

Faire de la qualité logicielle était considérée auparavant comme un luxe dans les développements informatiques. Aujourd'hui, les certifications ISO 9001 et 9126 sont devenues une valeur ajoutée pour les entreprises. Les propriétés des programmes comme leur fiabilité ou leur maintenabilité sont devenues des facteurs importants pour s'assurer du bon fonctionnement de systèmes informatiques devenant de plus en plus interconnectés.

Des travaux de recherche sur la qualité des systèmes à objets ont contribué à la définition de métriques de qualité. Il en existe plus d'une centaine actuellement. Devant ce nombre croissant, des travaux parallèles ont contribué à pouvoir les exprimer de manière simple, grâce à des primitives et des opérations ensemblistes.

Le but du projet que j'ai eu à réaliser est de fournir une implantation de ces travaux. Dans le but d'étendre ce projet, il a été prévu de fournir une extension qui viendrait s'intégrer à la plate-forme Eclipse.

Ce rapport a pour but de présenter l'organisme où j'ai effectué mon projet de fin d'études, le projet que l'on m'a confié et les résultats obtenus.

1.Le CRIM

1.1. Son rôle au Québec

Le Centre de Recherche Informatique de Montréal (CRIM) est un organisme à caractère semi-public et semi-privé. Son premier rôle au Québec est d'assurer le transfert de technologie des équipes de recherche universitaire vers les entreprises.

1.1.1. Une mission de transfert de technologie

Le transfert de technologie a toujours été une priorité pour chaque gouvernement. Cela est primordial pour améliorer la compétitivité des entreprises et favoriser la croissance d'un pays. Il existe souvent une personne ou un service chargé de cette mission dans les grandes entreprises ou les grandes universités. A eux de multiplier les contacts et ainsi étendre leur réseau de connaissances. Comment faire quand personne n'assure ce travail ou quand on manque de visibilité ? Le CRIM est un de ces établissements qui fait la liaison entre des universités et des entreprises qui n'ont pas toujours le temps de se créer des contacts.

Le CRIM compte actuellement 132 partenaires au total. Parmi les industriels, on peut citer Bombardier Transport, Bell, Microsoft ou SAP Labs. De même, il a des liens avec la majorité des établissements supérieurs du Québec. L'Université de McGill et l'Université de Montréal en font partie. Elles figurent aussi parmi les cinq premières universités du Canada.

Type de partenaire	Effectifs
Grandes entreprises	25
PME	73
Universités	10
Centres de recherche	2
Membres associés	19
Membres internationaux	2
Membres d'office	1

Doc. 1 : Partenaires du CRIM

A côté de ses activités en recherche & développement, le CRIM dispense des formations dans plusieurs domaines: génie logiciel, sécurité informatique, etc. Ainsi 4000 personnes du monde de l'industrie ont pu suivre des cours, des séminaires ou des conférences dans ses locaux en 2003.

1.1.2. Son financement

Le principal partenaire financier du CRIM est le ministère du Développement économique et régional du Québec, à hauteur de 49 %. Le gouvernement québécois est ainsi très impliqué. Suivant les orientations politiques, le budget alloué a subi des coupures. Il lui a fallu trouver d'autres partenaires et multiplier les contrats avec les industriels, tout en gardant sa mission de lien entre les universités et les entreprises. Il en ressort une nouvelle dynamique, dans des temps où l'informatique subit une restructuration et peine à redémarrer économiquement.

Revenus :

Nature	**Valeur en $CDN**	**Pourcentage**
Subvention de base	5 275 000	49
Cotisations	579 941	5
Recherche et développement	1 085 534	10
Formation	1 321 867	12
Centre de tests du logiciel	1 540 593	14
Entreprises partenaires	530 000	5
Autres	516 188	5
Total	**10 849 123**	**100**

Charges :

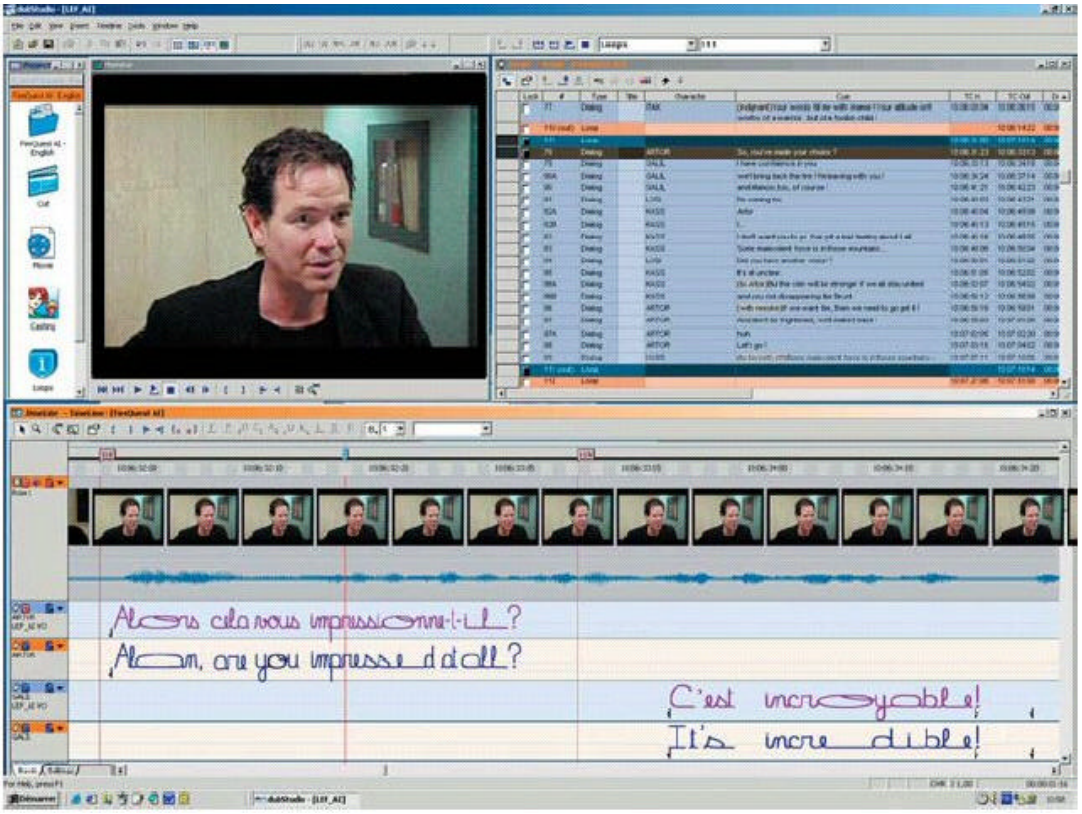
Nature	**Valeur en $CDN**	**Pourcentage**
Salaires et charges sociales	6 573 869	59
Honoraires et compléments de bourses	1 083 607	10
Charges d'exploitation	3 069 578	28
- Loyer, électricité et taxes	1 393 674	13
- Publicité et promotion	289 993	3
- Services professionnels administratifs	207 119	2
- Autres	1 178 792	10
Amortissement des immobilisations	339 105	3
Total	**11 066 159**	**100**

Doc. 2 : états financiers du CRIM en 2003

1.1.3. Les résultats du CRIM

Sur le plan de la recherche, le CRIM compte trois équipes de recherche. Celles-ci travaillent dans les domaines de la reconnaissance de la parole, la vision et l'analyse de systèmes distribués. En 2002-2003, elles ont réalisé 49 publications scientifiques. Plusieurs étudiants y réalisent actuellement des doctorats ou des maîtrises, en co-tutelle avec les universités du Québec.

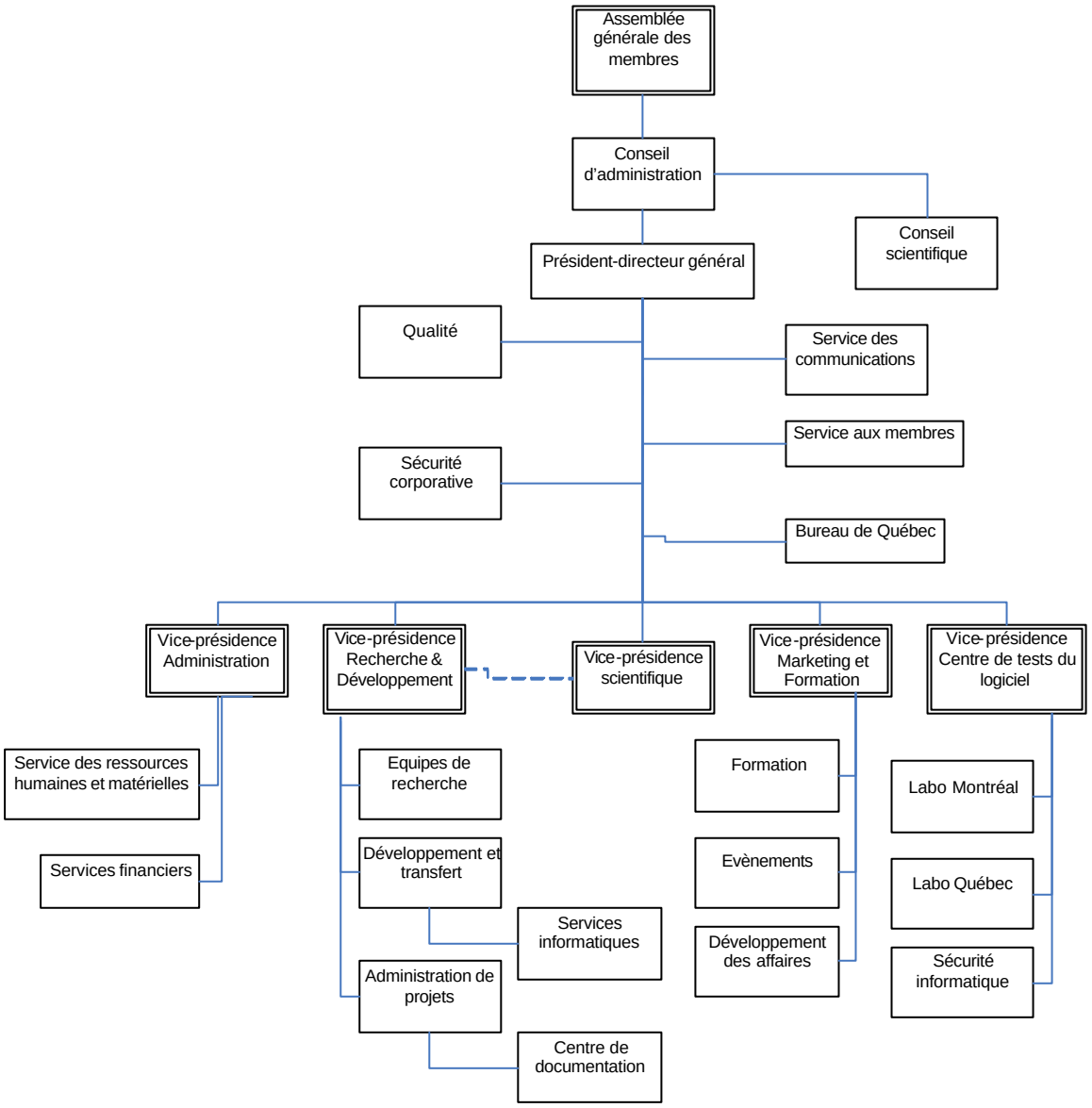
Qualifié de «moteur d'innovation pour le Québec », le CRIM voit son action concrétiser dans plusieurs réalisations avec ses partenaires industriels L'une des plus fameuses a été dubStudio. C'est un programme de postsynchronisation de doublage de la voix pour le cinéma et la télévision. Il utilise la technique de la reconnaissance de la parole. Réalisé en collaboration avec l'entreprise Ryshco Média, ce projet a reçu le prix OCTAS de l'Innovation Technologique.



Doc. 3 : projet DubStudio

1.2. Organisation

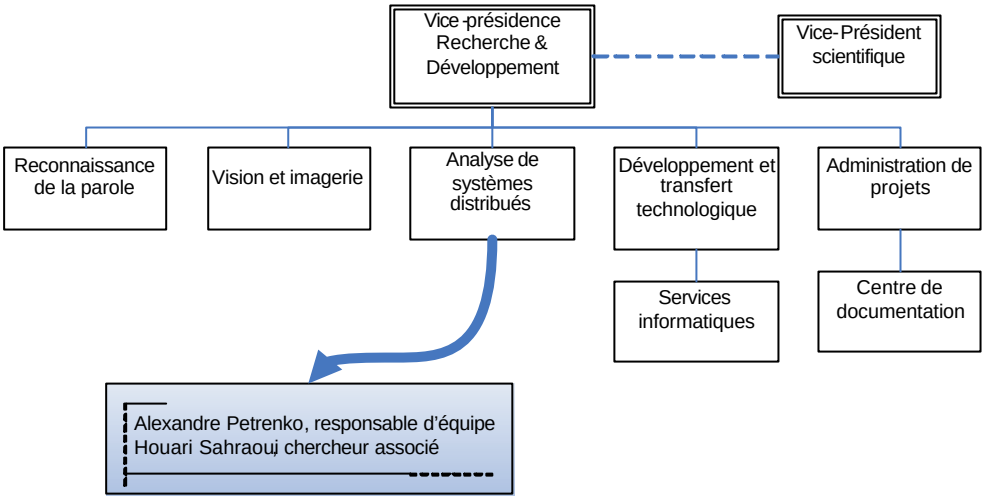
Le Conseil d'Administration du CRIM est composé principalement de membres issus des entreprises et des universités partenaires. Ceux-ci s'assurent de la cohérence de la direction, assumé actuellement par Yves Sanssouci. Sous sa direction, les vice-présidents sont chargés d'une des fonctions de l'entreprise.



Doc. 4 : organigramme du CRIM

1.3. L'équipe Analyse de systèmes distribués

En dessous de la vice-présidence Recherche & Développement se trouvent les équipes de recherche. L'équipe « Analyse de systèmes distribués » se trouve sous la responsabilité d'Alexandre Petrenko. Celui-ci encadre actuellement plusieurs étudiants en thèse et des stagiaires. Pendant mon stage, j'étais sous la co-direction de Mr Petrenko et Mr Sahraoui, actuellement chercheur associé au CRIM.



Doc. 5 : organigramme de la fonction Recherche et Développement

2. Le contexte technique

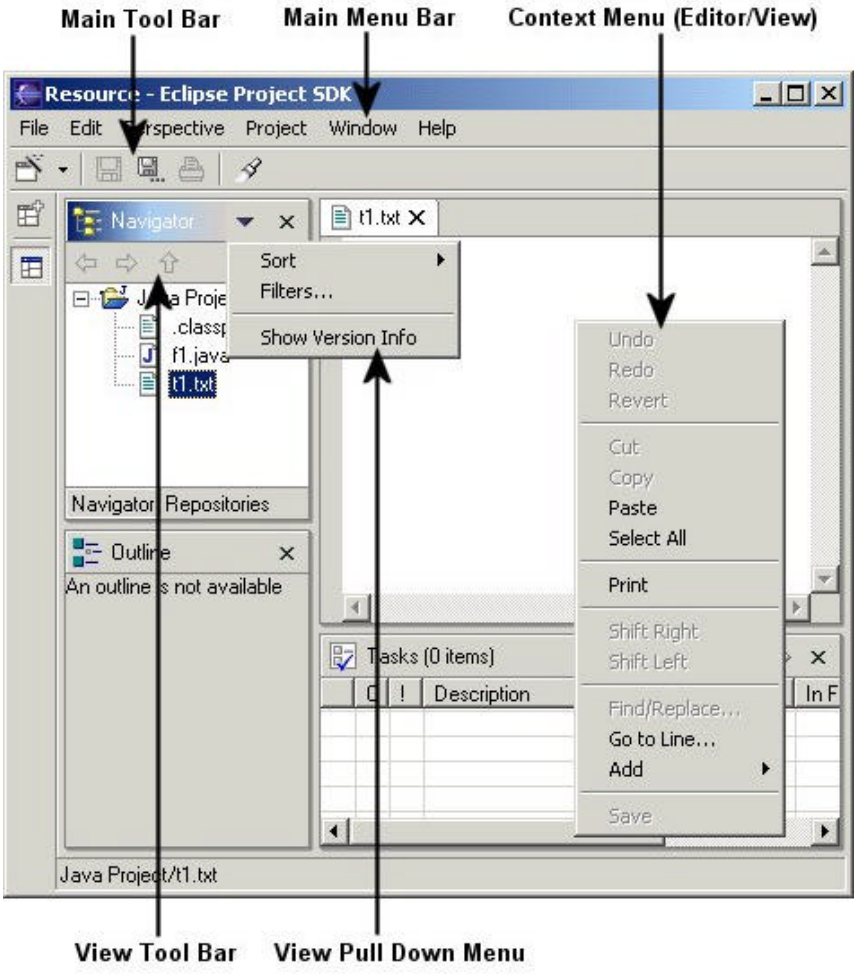
2.1. Eclipse

Eclipse est un projet open-source dont le but est de fournir un environnement de développement en JAVA. Ce projet a été initié par IBM qui a confié le développement à Object Technology International. Cette société s'est déjà illustré avec IBM Visual Age. En reprenant certains traits de ce dernier, Eclipse est devenu un environnement mature, un des leaders dans les ateliers de développement JAVA. Devant l'avancée de Borland sur le marché des IDE, IBM a décidé de réagir. Il s'illustre dans sa nouvelle orientation des logiciels open source. Néanmoins, il fournit aussi un logiciel payant Websphere Studio Application Developer. Il est basé sur le noyau d'Eclipse et les développements de ces deux plate-formes se font en parallèle, de manière centralisée pour assurer une cohérence. Le côté open source est un vecteur de diffusion pour IBM. La majorité des développeurs peuvent se former gratuitement aux produits d'IBM, en particulier dans les universités. Ceux-ci seront donc capables de maîtriser WSAD sans grande peine.

Eclipse est basé sur une architecture très originale qui offre un cadre très extensible. Le socle est composé du workspace et du workbench. Le workspace contient les données, c'est-à-dire les projets sur lequel le développeur travaille. Le workbench est l'interface graphique qui offre plusieurs perspectives. Il est possible de passer d'une perspective à un autre. Une perspective contient plusieurs éditeurs et vues. Il est facile d'en ajouter ou d'en retirer. Chacune des fonctionnalités est ajoutée avec des extensions. Par défaut, tous ceux qui sont nécessaires au développement JAVA sont présents.

Ainsi, Eclipse se démarque profondément des anciens environnements comme IBM Visual Age. C'était un outil de développement très fermé. Il était très compliqué de le faire interagir avec d'autres environnements. Eclipse change totalement d'orientation et mise sur un cadre extensible et le travail collaboratif. Par exemple, CVS offre une interface simple pour partager les ressources sur lesquels une personne travaille et permet de voir les différences entre les versions. La comparaison est basée sur une différence des fichiers source. Il n'est pas possible de faire la même chose pour des fichiers binaires.

Les extensions sont des composants que tout le monde peut développer. Actuellement, il en existe une multitude qui couvre la majorité des besoins pour les programmeurs: développements en J2EE, C++, UML. Ils sont gratuits ou commerciaux. Le noyau d'Eclipse gère l'ensemble des extensions, entre autres leur cycle de vie. Il est possible de charger un nombre important de extensions. L'environnement Eclipse, du fait de sa grande extensibilité, devient lourd. Le chargement d'un extension durant le runtime permet d'optimiser la gestion mémoire. Quand il n'est plus utile, il est déchargé de la mémoire. C'est ce que permet la dernière version d'Eclipse. Il existe aussi un ensemble de lignes de conduite pour la programmation de extensions. Il s'agit de les respecter pour assurer une meilleure cohabitation des extensions dans l'environnement. Ces « règles » prônent la réutilisation (interfaces graphiques, code source) ou la facilité d'ajout pour les autres contributeurs. Une partie de mon travail pendant ce stage a été de contribuer à Eclipse, en développant mon travail sous la forme d'une extension. Cela représente une opportunité de diffuser le travail accompli actuellement sur la qualité des programmes.



Doc. 6 : interface graphique d'Eclipse

2.2. Métriques de qualité

On peut résumer la théorie de la mesure par la citation suivante de Lord Kelvin :

« Lorsque vous pouvez mesurer ce dont vous parlez et l'exprimer en nombres, vous savez quelque chose à son sujet ; mais lorsque vous ne pouvez l'exprimer en nombres, votre connaissance est maigre et peu satisfaisante : c'est peut-être le début de la connaissance mais dans vos pensées, vous avez à peine progressé jusqu'au stade de la science. »

Pour juger de la qualité d'un programme, les programmeurs peuvent juger les performances ou la qualité du code source. C'est souvent ad-hoc, c'est-à-dire que le jugement est basé sur l'expérience. Comment faire sur des grands systèmes ? Il faut disposer de métriques clairement définies dont le calcul peut être automatisé.

Ainsi les métriques de qualité mesurent une propriété d'un programme. En objet, on peut compter les éléments d'un programme (classes, interfaces, etc.) et juger des attributs internes d'un programme comme le couplage ou la cohésion. La définition de métriques et d'une échelle de mesure est problématique, au niveau de la sémantique. Certaines propriétés sont faciles à déduire comme l'héritage. Pour d'autres, la difficulté se retrouve au niveau de la sémantique : que mesure-t-on ?.

Actuellement, il est possible de calculer automatiquement les attributs internes d'un programme, et cela grâce à la méta-modélisation. Elle permet de manipuler la représentation d'un programme. Pour un calcul pertinent, le méta-modèle utilisé doit être riche d'informations.

2.3. La méta-modélisation en PADL

Pour disposer de la représentation d'un programme, le méta-modèle PADL a été utilisé. Il nous évite le développement d'un analyseur syntaxique et la construction d'un modèle du système logiciel. Les informations présentes dans le méta-modèle sont très importantes. A partir de celles-ci, les primitives sont calculées. Si le modèle est incomplet, il est alors nécessaire de l'enrichir. PADL se base lui-même sur CPL et CFParse, une librairie IBM pour l'analyse des fichiers JAVA compilés.

2.3.1. CPL

CPL est une librairie JAVA qui permet le chargement de fichiers JAVA compilés. Dans le but de simplifier cette opération, CPL fournit un unique point d'entrée. Il constitue une couche au-dessus de CFParse.

CFParse est une librairie IBM destinée à la manipulation des classes JAVA compilées. Il est possible de lire les informations contenues dans un fichier (nom de la classe, attributs, méthodes, etc.) ou de le modifier. Un des intérêts de prendre le bytecode d'un programme comme base est de posséder le code qui sera exécuté. Le calcul des métriques se retrouve amélioré. Le « code mort » est supprimé par le compilateur lors d'optimisations (ex : suppression des tests inutiles). De même, il est possible d'analyser des logiciels propriétaires, sans pour autant avoir les codes sources.

CFParse passe par une étape de reverse engineering et crée un modèle assez pauvre du programme. Par exemple, les relations d'héritage ne figurent pas. Il est juste possible de connaître la composition d'une classe. Le modèle offre assez d'informations pour créer un modèle plus complet. Pour se faire, il est nécessaire de posséder un minimum de connaissance sur la machine virtuelle JAVA. J'ai eu l'occasion de lire la spécification la JVM. Le bytecode y est décrit avec précision, notamment toutes les instructions et leur action sur la pile mémoire. Sans se reporter à cette documentation, il est difficile de pouvoir aller plus loin dans l'analyse.

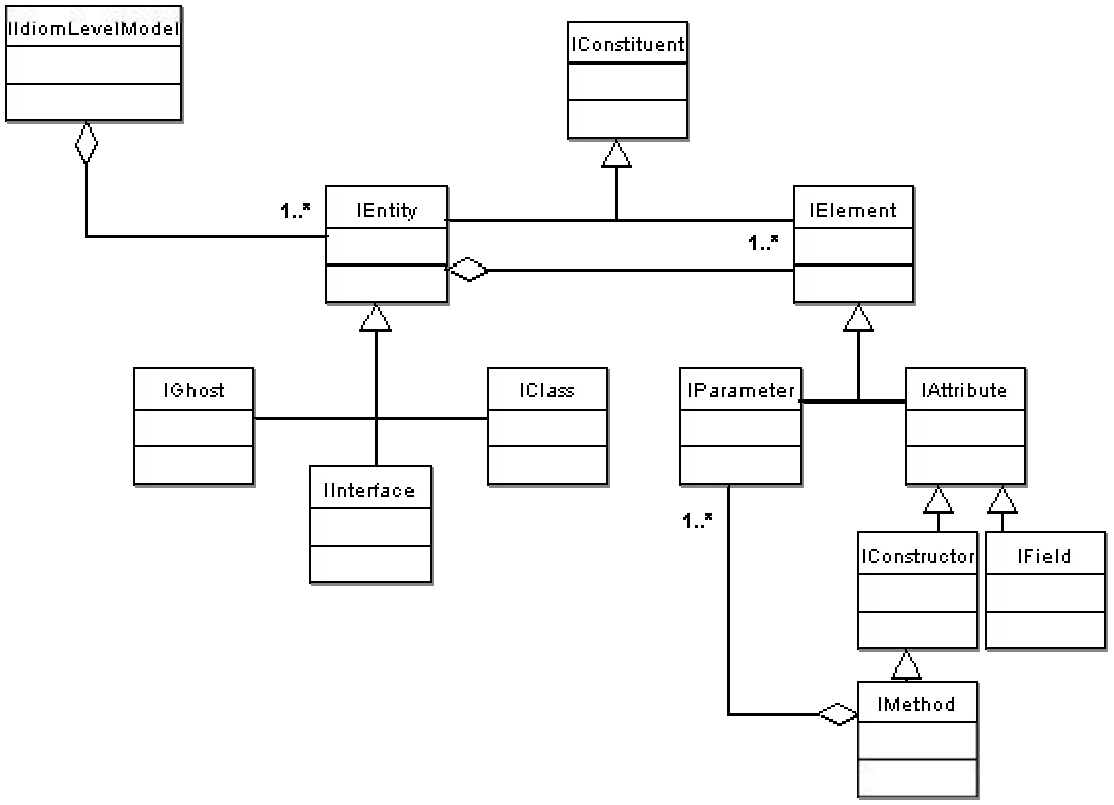
2.3.2. PADL

PADL est un méta-modèle développé par Yann-Gaël Guéhéneuc et Hervé Albin-Amiot. Son code source est libre. Il est donc possible de le modifier selon les besoins du projet.

CFParse effectue la première étape du travail en fournissant un modèle exploitable. PADL l'enrichit en effectuant une analyse plus approfondie des classes. Ce processus en deux étapes permet de découpler le parsing et la construction finale du méta-modèle. PADL peut être réutilisé pour représenter des programmes objet écrits en C++, SmallTalk ou ADA par exemple.

PADL fournit une représentation de plus haut niveau d'un programme, proche des diagrammes de classe UML au niveau de la conception. Il considère les relations d'agrégation, d'association ou d'héritage entre les classes. Ces relations sont nécessaires pour faire du reverse engineering et obtenir un modèle proche d'une conception de haut niveau. Par exemple, une cardinalité 1,n entre deux classes est souvent implémenté en utilisant une classe du package java.util. On retrouve ainsi les relations entre les classes à un niveau plus haut.

Le diagramme de classes ci-dessous permet de mieux comprendre la structure de PADL. Un modèle est défini par l'interface IIdiomLevelModel. Un modèle contient classes et des interfaces. La notion de Ghost a été introduite pour représenter des classes dont le fichier n'était pas disponible lors de l'analyse. Une entité (IEntity) possède une liste d'éléments génériques. Ce sont ses attributs et ses méthodes.



Doc. 7 : diagramme de classes de PADL

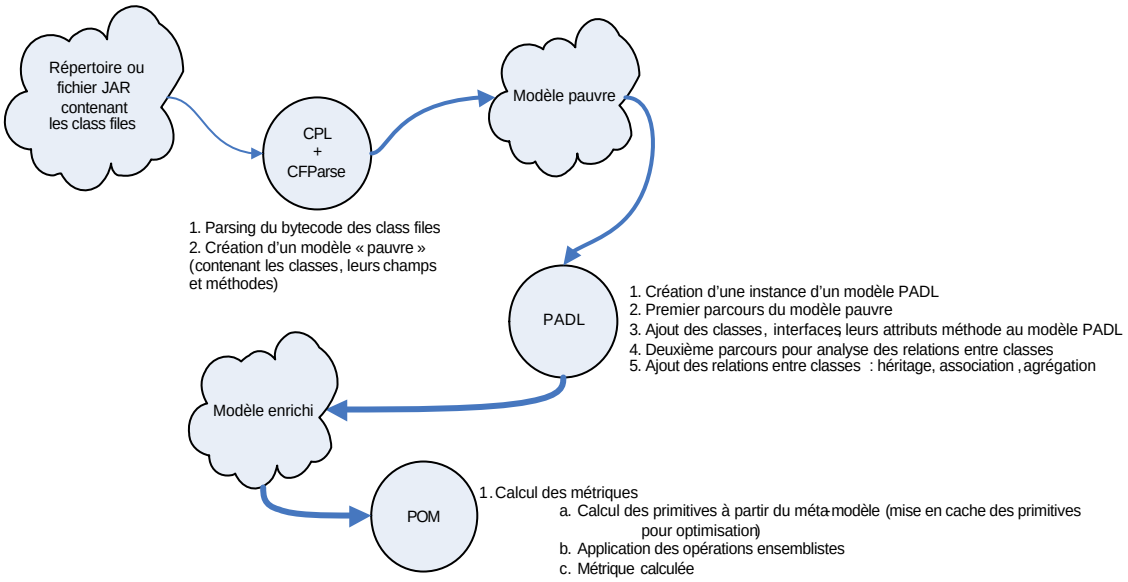
3. Le travail réalisé

3.1. Conception & Architecture

L'objet de mon stage consistait à créer un outil basé sur des travaux de recherche déjà avancés : le calcul de métriques à partir d'opérations ensemblistes . Un document de travail du CRIM a constitué la base pour commencer à implémenter les métriques et les opérations ensemblistes. Ce document est une compilation de travaux antécédents. Il décrit pour certaines métriques de qualité leur expression ensembliste et les primitives à calculer. Il présente aussi des pseudo-algorithmes pour calculer des ensembles, cela en fonction d'un méta-modèle donné. Le problème est que ce dernier n'était pas implémenté. De même, le document faisait des suppositions sur ce qui pouvait exister. Certains problèmes dans la méta-modélisation étaient donc mis délibérément de côté.

Deux possibilités s'offraient pour la réalisation de mon travail : implémenter le méta-modèle proposé dans la spécification ou en réutiliser un autre. A ce moment, j'ai eu la chance d'avoir rencontré Mr Guéhéneuc. Il m'a proposé PADL comme méta-modèle. Sa description satisfaisait la majorité des besoins. Sachant que Mr Guéhéneuc et Mr Sahraoui travaillent ensemble, il était aussi facile d'obtenir de l'aide rapidement et de se faire orienter. C'est là qu'a débuté notre collaboration en eXtreme Programming. Dans un premier temps, j'ai eu à maîtriser certains concepts dans la méta-modélisation. Dans un deuxième temps, j'ai appris à manipuler la librairie de PADL.

Le calcul des métriques se fait en trois étapes, chacune étant assurée par une bibliothèque. La première consiste à faire l'analyse de l'ensemble des programmes JAVA avec CPL et CFParse. A partir d'un premier méta-modèle, PADL crée les entités du modèle, leurs propriétés (champs et méthodes) et analyse les relations entre les classes. Le méta-modèle PADL dispose alors de plusieurs informations. POM va les utiliser pour calculer des primitives (ex : ancêtres d'une classe) et ensuite, à partir d'opérateurs sur les primitives, calcule les métriques.



Doc. 8 : calcul des métriques en 3 étapes

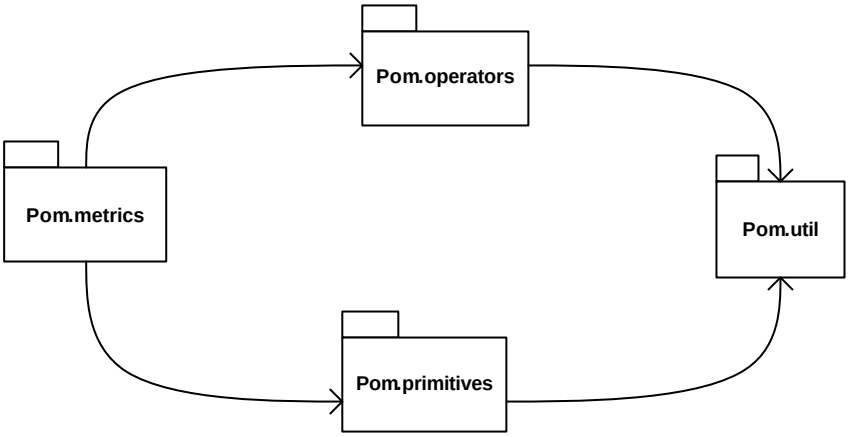
La décomposition en trois couches permet d'assurer un découplage entre les packages. Il est donc possible de substituer chacune des bibliothèques si une autre correspond plus à nos attentes. Par exemple, il est possible d'utiliser un méta-modèle autre que PADL. De même, si on veut analyser des programmes objet écrits dans un autre langage, il suffit dutiliser un parseur autre que CFParse. On peut donc calculer des métriques sur des programmes écrits en C#, SmallTalk-80, etc.

L'architecture interne de POM est composée de trois packages.

Le package pom.operators définit les opérations qu'il est possible d'appliquer sur des ensembles. Il contient par exemple l'union, l'intersection, la comparaison, les quantificateurs existentiels. Pour faire la correspondance en JAVA, un ensemble est représenté par une instance de la classe java.util.List

Le package pom.primitives définit les primitives que l'on extrait du méta-modèle. Les primitives pour une classe sont de trois types : classes ou interfaces, méthodes, champs. Par exemple, il est possible de connaître pour une classe ses ancêtres, ses descendants. Des primitives permettent de calculer les méthodes surchargées, les méthodes héritées, les nouvelles méthodes implantées ou les invocations de méthode d'une classe. Pour les champs, des primitives extraient les attributs que les méthodes de la classe utilisent. On peut imaginer un nombre illimité de primitives. Mais dans notre projet, seules les primitives permettant le calcul des métriques spécifiées nous intéressaient.

Le package pom.metrics contient l'implantation des métriques. Chacune des méthodes prend en paramètre un objet de type IEntity. Sur celui-ci sont calculés des primitives et le résultat de la métrique. Enfin, le package pom.util contient des méthodes utilitaires, utilisées par les autres packages.



Doc. 9 : packages de POM

3.2. Processus de développement

Le contact humain avec les autres membres de l'équipe est motivant et permet d'améliorer la qualité du projet. De même, le fait de produire des versions fonctionnelles durant des cycles courts aide à garder l'intérêt et le goût du travail. De plus, il est facile de mesurer l'avancée du projet. Le choix de faire le développement en eXtreme Programming (XP) devient alors naturel.

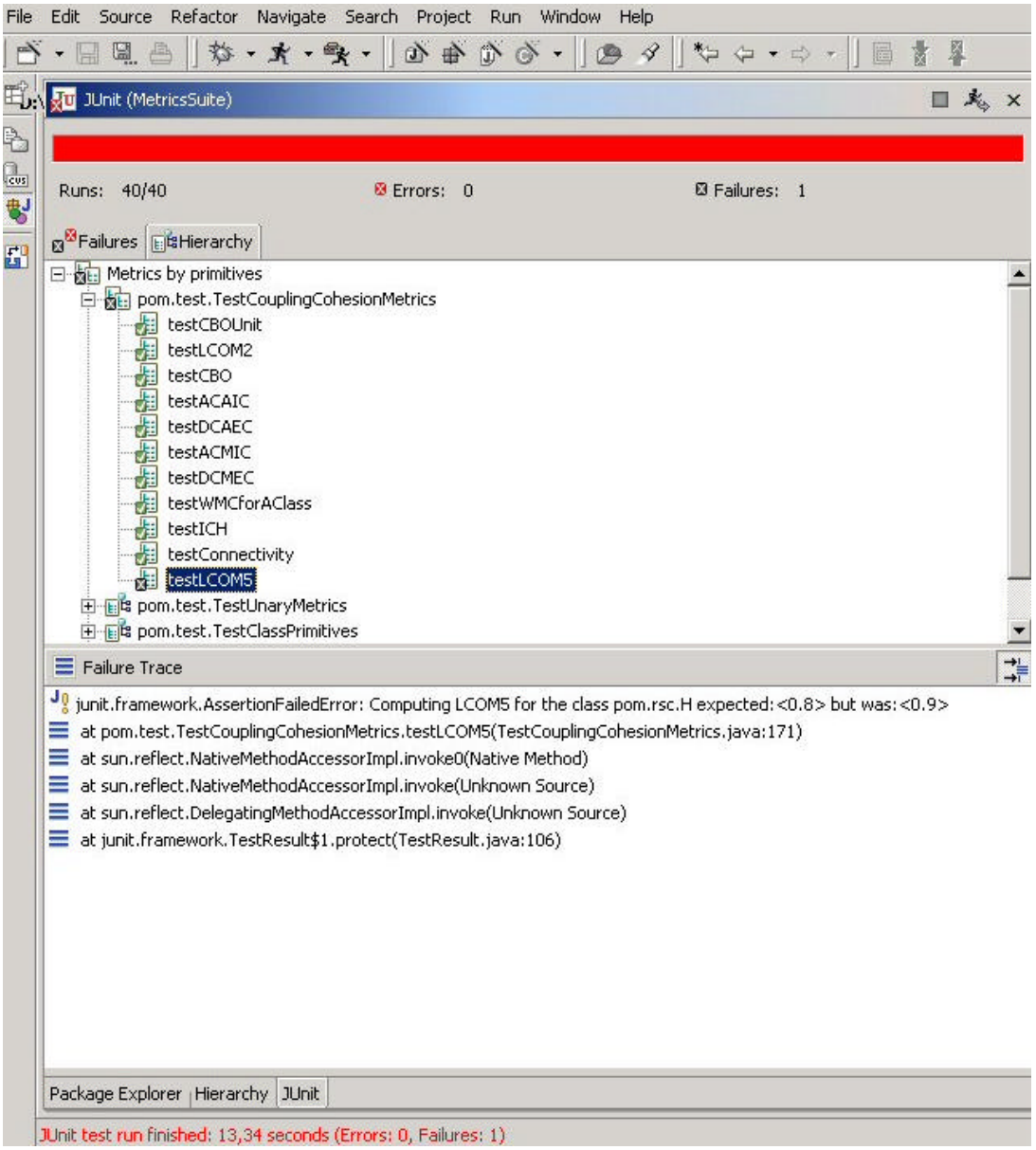
La méthode XP donne la priorité aux interactions entre les développeurs et les utilisateurs, pour capturer les réels besoins tout au long du projet. Au départ, ils n'étaient pas clairement définis. Aussi, plusieurs personnes devaient se servir de mon travail. Pendant six mois, j'ai donc principalement collaboré avec Mr Guéhéneuc pour les développements, surtout en ce qui concernait les ajouts à PADL. J'ai aussi bénéficié de ses conseils sur l'utilisation des patrons de conception dans la conception.

Pour mesurer l'avancée du projet, les tests JUnit représentent une bonne solution. Un test valide une fonctionnalité ou une nouvelle métrique. Si le test ne passe pas, il est facile de détecter l'erreur. Nous avons suivi la description d'un cycle court donnée par XP :

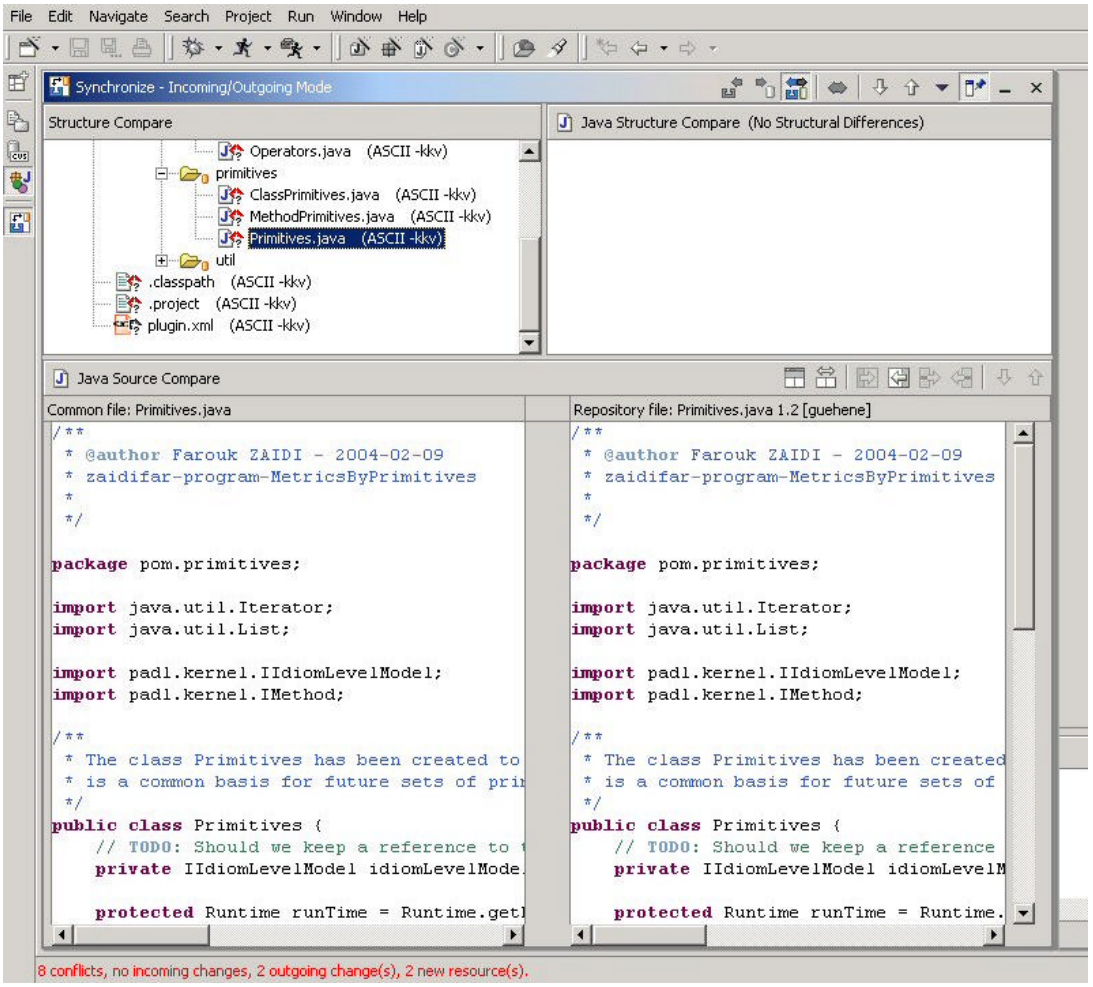
1. Le besoin est décrit sous forme textuelle.
2. Un test JUnit qui permet de valider le résultat attendu est écrit.
3. Le test doit être documenté.
4. Le besoin est implanté, selon le code déjà existant.
5. Le code doit être documenté.
6. Si le test ne passe pas, le code est corrigé.

Cette méthode oblige à écrire des tests, ce qui n'est pas toujours la priorité des développeurs. L'exécution des tests permet aussi de prévenir les impacts de changements dans le projet. Si un ou des tests ne sont plus validés, alors il doit être possible de retrouver l'erreur assez rapidement.

Doc. 10 : Interface JUnit



Eclipse facilite le travail collaboratif grâce à CVS. Pour les utilisateurs, nous avons créé des versions stables des projets avec lesquels il pouvait travailler au fur et à mesure. Vu que nous étions plusieurs développeurs parfois à utiliser les mêmes projets, il était important de se tenir au courant des changements. La communication verbale est nécessaire. De plus, Eclipse permet de comparer les fichiers sources, de manière très pertinente. Avec tous ces outils, il était assez facile d'avancer dans le projet sans pour autant bloquer une autre personne.



Doc. 11 : interface du plug-in CVS d'Eclipse

3.3. Implémentation

3.3.1. Implantation des ensembles et des métriques

A l'aide du premier document de travail, les ensembles nécessaires aux métriques ont été implantés. Ci-dessous sont donnés quelques exemples d'ensembles très utilisés. Le pseudo-code est utilisé ici pour s'abstraire du langage et mettre en évidence l'utilisation des ensembles.

Ancestor : ensemble des super-classes de c

```
Set ancestor(Class c)
{
  Set S = emptySet;
  Set p = parents(c) ;
  S = p ;
  for (i = 0 ; i < c.nbreParents ; i++)
  {
    Class oneClass = p.next() ;
    S = S UNION ancestor(oneClass) ;
  }
  return S ;
}
```

Deux catégories de méthodes ont été faites selon la sémantique de l'objet, par deux formalismes. Ainsi l'ensemble des méthodes d'une classe a été exprimé de deux manières :

Formalisme 1: M(c) est l'union de deux ensembles disjoints ; l'ensemble des méthodes « déclarées » et l'ensemble des méthodes « implémentées »

$$M(c) = M_I(c) \cup M_D(c) \text{ et } M_D(c) \cap M_I(c) = \varnothing$$

Formalisme 2 :

- Minh(c) : ensemble des méthodes héritées
- Movr(c) : ensemble des méthodes surchargées
- Mnew(c) : ensemble des nouvelles méthodes définies dans c, ou bien non héritées et non surchargées.

$$M(c) = \text{Minh(c)} \cup \text{Movr(c)} \cup \text{Mnew(c)}$$

L'ensemble M_D est défini par les méthodes héritées de la classe c qui ne sont pas redéfinies, ou les méthodes virtuelles de la classe c. Le pseudo-code est donné ci-après :

```
Set methodDefined(Class c)
{
  Set S = emptySet;
  for (i = 0 ; i < c.nbreFeature; i++)
  {
    Feature oneFeature = c.features[i] ;
    boolean ok = false ;
    if ( ! oneFeature instanceof Attribut)
    {
      if (oneFeature.parentFeature == null)
      {
          if (oneFeature.isVirtuel()) ok = true ;
       }
       else {
      {
        FInheritance fInh = oneFeature.parentFeature ;
        if (fInh.status ==  INHERITANCE) ok = true ;
      }
      if (ok)  S = S UNION {oneFeature} ;
    }
  }
  return S ;
}
```

Le pseudo-code montre qu'il est possible de s'abstraire facilement du méta-modèle puisque l'on ne considère que des éléments au sens objet (classes, méthodes, champs). Mais la manière de considérer certaines notions reste relative au langage parsé. La différence de l'héritage entre SmallTalk-80 et C++ le prouve.

3.3.2. Ajout des invocations de méthode dans PADL

Certains ensembles nécessitent l'analyse des corps des méthodes. Or le méta-modèle PADL était dépourvu de ces analyses et moyens de les représenter. Le problème est très complexe car il faut faire le lien entre le code JAVA et le bytecode. CPL nous permet pour un bytecode, de retrouver la méthode invoquée, mais il ne permet pas de la rattacher à l'appelant, ni de connaître les paramètres de la méthode appelée.

L'algorithme qui a été utilisé essaie de reproduire la logique de la machine virtuelle qui est une machine à pile. Certains bytecodes empilent ou dépilent des données sur la pile mémoire. Les invocations de méthode sont données par les quatre micro-instructions suivantes :

- **invokeinterface** : invoque une méthode héritée
- **invokespecial** : invoque un constructeur
- **invokestatic** : invoque une méthode statique
- **invokevirtual** : invoque une méthode d'instance

En s'appuyant sur la spécification de la machine virtuelle JAVA, il est possible de décomposer le corps d'une méthode en instructions JAVA, c'est-à-dire une instruction qui se termine par un «; ». Par la suite, il s'agit de pouvoir de retrouver les appels imbriqués. La machine à pile est construite à ce moment là.

Ci-dessous est donné un exemple de code JAVA. Le bytecode correspondant est donné après, avec une reconstruction partielle des invocations de méthode.

```
package examples.cfparse;

public class MethodDump implements MyInterface {

  private int myMethod(String s,int i) throws Exception {
            float f = 3.3f;
            MyInterface mi = (MyInterface)this;
            mi.doNothing();
            System.out.println(f+" ");
            return 0;
  }

  public void doNothing(){
            return;
  }
}

interface MyInterface {
  public void doNothing();
}
```

Bytecode de la méthode myMethod(String, int)

```
//Début du corps
0: ldc F3.3          //Met une donnée de type float et valant 3.3 sur la pile
1: fstore_3          //Affecte la variable f avec 3.3F
2: aload_0           //Place la variable this sur la pile
3: astore v4         //Affecte this à la variable locale mi
4: aload v4          //Met la variable mi sur la pile
5: invokeinterface void cfparse.tutorial.MyInterface.doNothing() #1 #0
                     //Invoque la méthode doNothing
6: getstatic java.io.PrintStream java.lang.System.out
                     //Met la variable statique out de type PrintStream sur la pile
7: new java.lang.StringBuffer
                     //Appel au constructeur de la classe StringBuffer
8: dup
9: fload_3
                     //Place la variable locale f sur la pile
10: invokestatic java.lang.String java.lang.String.valueOf(float)
11: invokespecial void java.lang.StringBuffer.<init>(java.lang.String)
                     //Construit la chaîne de caractères à afficher
12: ldc " "
                     //Place la chaîne de caractères " " sur la pile
13: invokevirtual java.lang.StringBuffer
        java.lang.StringBuffer.append(java.lang.String)
                     //Invoque append qui prend les deux derniers arguments de la pile
14: invokevirtual java.lang.String java.lang.StringBuffer.toString()
15: invokevirtual void java.io.PrintStream.println(java.lang.String)
                     // Invoque la méthode println
16: iconst_0
                     //Place la valeur 0 sur la pile
17: ireturn
                     //Retourne la valeur entière
```

Dans la majorité des cas testés et rencontrés, il a été possible de retrouver la méthode appelée et l'appelant, d'identifier le type de méthode (statique ou d'instance), l'ordre d'appel des méthodes et les champs de la classe utilisés. Cette partie a demandé un mois à plein temps.

3.3.3. Problèmes de performance

Lors de tests sur des grands systèmes, des problèmes de performances sont apparus. Sur des programmes de l'ordre de 3000 fichiers, les calculs étaient lents. JProfiler est un outil qui permet de voir les ressources utilisées par un programme, tout au long de son exécution. Il a permis de retrouver quelles étaient les méthodes les plus appelées, l'utilisation de la mémoire et du processeur à chaque instant. Le problème venait des structures de données manipulées. La création de grands ensembles et l'application d'opérations sur ceux-ci sont très coûteuses en temps mémoire et processeur. Le garbage collector de la machine virtuelle devient inefficace.

Deux solutions sont alors possibles : augmenter la mémoire physique de la machine ou mieux gérer les ensembles créés. La première solution est possible par exemple, si on exécute le programme sur un cluster. Néanmoins, le problème réapparaîtrait sur des systèmes plus grands. La deuxième solution est la plus viable. La gestion de la mémoire est laissée à notre soin. Pour cela, une technique est de calculer à l'avance le nombre de listes que l'on peut stocker, en fonction de la mémoire allouée à la machine virtuelle. Par exemple, pour 100 Mo alloués, il sera possible de stocker 10 000 listes. Une fois dépassé ce nombre, toutes les listes stockées sont supprimées explicitement et un appel au garbage collector permettrait de défragmenter la mémoire.

Au niveau de la conception, avant de calculer un ensemble, il est d'abord recherché dans un tableau associatif qui contient tous les ensembles. S'il n'est pas présent, l'ensemble est calculé et stocké dans le tableau. Avant chaque ajout dans le tableau, un test indique si sa taille est toujours inférieure au nombre maximum de listes possibles à stocker.

Cette solution est très efficace. Elle a permis, avec le même PC, de calculer des métriques sur NetBeans. Il contient à peu près 6 000 classes JAVA.

3.3.4. ANTLR

Au bout de quelques mois, une première version de POM était opérationnelle. Une des orientations du projet a été de faire un langage permettant d'écrire des métriques simplement. Par exemple, la métrique CBO (Coupling Between Objects) s'écrit de la manière suivante :

CBO(c)=|{d ? C\{c}:uses(c,d) OR uses(d,c) }|

Définition: soit un ensemble où il existe d appartenant à l'ensemble C (toutes les classes du système) exclus c, tel que c invoque d OU d invoque c. CBO(c) est la cardinalité de cet ensemble.

Grâce à un éditeur d'expressions, calculer des métriques ne nécessiterait que très peu de connaissances de JAVA. A partir de son expression « mathématique », le code JAVA est automatiquement généré.

Au départ de cette idée, il faut d'abord écrire la grammaire de ce langage. Puis j'ai utilisé ANTLR, un outil très puissant qui sert à générer des parseurs. A partir de la grammaire, il est possible de générer un parseur en JAVA. Un des avantages est qu'il existe un plug-in pour Eclipse. Le travail est donc simplifié. Cependant, être familier avec la théorie des langages est important. Le code ci-dessous est le début de la grammaire. Pour écrire un parseur avec ANTLR, il faut d'abord décrire les mots réservés (tokens) du langage. Pour écrire les règles, il faut le faire d'une manière descendante. L'élément le plus complexe est décrit, les éléments les plus simples sont décrits à la fin (ex : metric, declareMetric, pour terminer vers expr).

```
tokens{
        FOREACH="foreach";
        EXISTS="exists";
        METRIC="metric";
        IF="if";
        ….
}

metric: declareMetric EQUAL^ (parenthesisExpr|expr|ifThenElseExpr);

declareMetric : METRIC^ d=declarativePart;

declarativePart : IDENT^ LEFTPARENTHESIS! arguments RIGHTPARENTHESIS!;

arguments : argument (COMMA! arguments)?;

argument : identifier;

identifier : IDENT ;

arithmeticExpr:  (multiDivExpr ( (MINUS^|PLUS^) arithmeticExpr)?);

multiDivExpr :  (parenthesisExpr ( (DIVIDE^|MULTIPLY^) multiDivExpr)?)        ;
```

```
parenthesisExpr : (LEFTPARENTHESIS!  listComputeExpr RIGHTPARENTHESIS! );

listComputeExpr : (a=arithmeticExpr|m=multiDivExpr) (lst=listComputeExpr)?;

expr: countExprOnSet | functionCall | NOMBRE | identifier;
```

Il s'agit ensuite d'intégrer les actions sémantiques attachées à chacune des règles de la grammaire. Elles permettent de faire la génération JAVA. Pour chaque règle, on donne le code permettant de générer le code attendu. La transformation associée à la règle « metric » est donnée ci-après :

```
metric returns [String str=""] // Variable qui sera retourné
        {
                String d, e="", ifThenElse=""; // Initialisation de variables locales
        }
        :
/**
        La definition de la règle «metric »est réécrite pour affecter
        les valeurs des tokens aux variables locales.
*/
d = declareMetric EQUAL^
        (e=parenthesisExpr|e=expr
        | ifThenElse=ifThenElseExpr)
                /**
                        La transformation vers JAVA est écrite
                        en JAVA pour générer explicitement
                        du JAVA par ANTLR.
                */
                {
                        str +=d+"{"+"double metricValue=0;";
                        if(!e.equals("")){
                                str+="metricValue="+e+";"+"return metricValue;"+"}";
                        }
                        if(!ifThenElse.equals("")){
                                str+=ifThenElse;
                                str+="}";
                        }
                };
```

Pour la règle « metric », le code généré ressemble à celui-ci-dessus. Des bouts de code sont omis pour améliorer à la compréhension.

```
        public final String  metric(){
                String str=""; //Déclaration de la variable qui sera retournée

                returnAST = null;
                ASTPair currentAST = new ASTPair();
                AST metric_AST = null;

                String d, e="", ifThenElse=""; // Déclaration des variables locales
                                               // explicitement déclarés dans la
```

```
                                        // grammaire

                d=declareMetric();    // Déclenchement de la règle declareMetric
                                      // et récupération du retour du résultat
                match(EQUAL);         // Vérifie si le token est EQUAL
                {
                        if(…) {
                                e=parenthesisExpr();// Déclenchement de
                                                    // la règle parenthesisExpr
                                                    // et récupération du résultat
                                                    // dans e
                }
                else
                        if (…) {
                                e=expr();
                        }
                else if (…) {
                        ifThenElse=ifThenElseExpr();
                                                }
                else {
                        throw new NoViableAltException(LT(1), getFilename());
                }

                }


                /**
                        Le code associé à la transformation est inséré
                        après les étapes d'analyse syntaxique,
                        soit le travail assuré par ANTLR.
                */
                str +=d+"{"+"double metricValue=0;";
                if(!e.equals("")){
                        str+="metricValue="+e+";"+"return metricValue;"+"}";
                }
                if(!ifThenElse.equals("")){
                        str+=ifThenElse;
                        str+="}";
                }
        ….
        return str;
}
```

La grammaire était aboutie. La première version permettait de générer des métriques complexes. Ci-dessous, on peut retrouver l'expression de la métrique DCMEC définie par le langage que j'ai imaginé :

```
metric DCMEC(c) = card(
                        foreach c1 in descendents(c) :
```

```
foreach m in MNEW(c1) :
        exists a in Par(m) | T(a)=c
)
```

Cette expression calcule la cardinalité d'un ensemble tel que pour chacune des méthodes nouvellement implantées dans les classes descendantes c1 de la classe c, il existe a appartenant à l'ensemble des paramètres d'une méthode et tel que son type est du type de la classe c.

Il existe encore des inconsistances lors du passage vers JAVA dans certains cas, notamment au niveau du typage des variables. En considérant les contraintes de temps, nous avons jugé que l'éditeur d'expressions devait être mis en suspens. Ainsi, il restait assez de temps pour intégrer POM à un outil graphique.

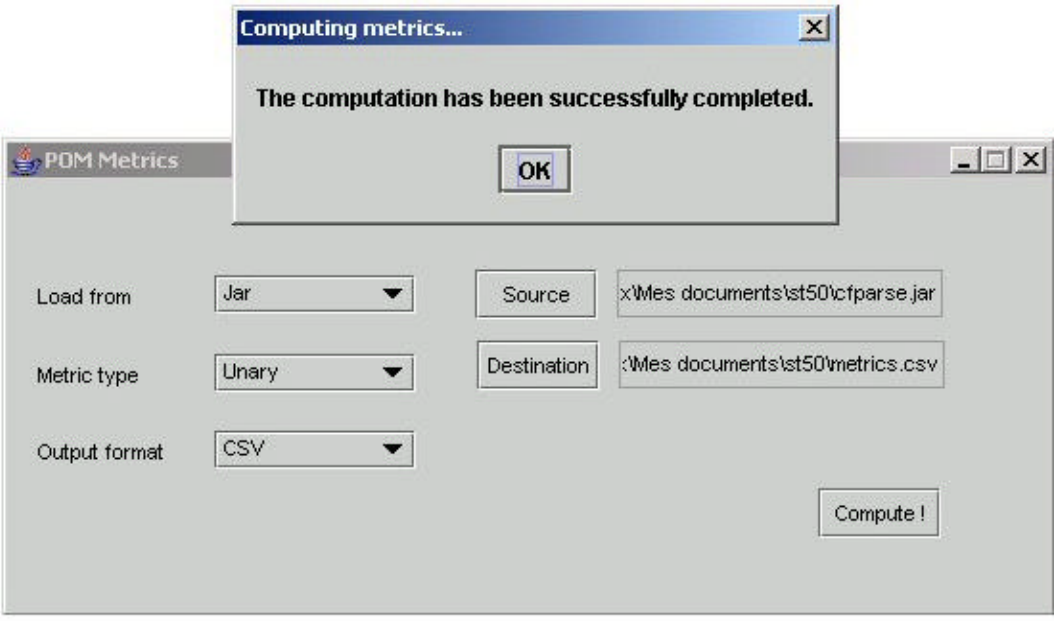
3.3.5. POM Stand-alone

Pour diffuser POM aux plus grands nombres d'utilisateurs et de développeurs, il était important de faire une interface utilisateur.

Une version stand-alone permet d'utiliser simplement POM pour réaliser des calculs. L'utilisateur doit spécifier le répertoire source et le fichier destination des résultats ainsi que le type des métriques. Enfin, il choisit le format de sortie. Nous avons choisi les formats XML et CSV. XML est un standard qui permet de réutiliser facilement les résultats par tout autre programme. CSV est un format où les données sont séparées par des «; », l'avantage est qu'il est possible de l'exploiter avec des tableurs, notamment Excel.

Au niveau de l'implantation, les nouvelles métriques qui pourraient être ajoutées sont prises automatiquement en compte. En effet, pour appeler la méthode adéquate, la réflexion dans JAVA (package java.lang.reflect) est utilisée. Toutes les métriques sont contenues dans la classe pom.metrics.Metrics. Il est possible de connaître et d'invoquer les méthodes publiques possédant un ou deux paramètres de type IEntity. La réflexion est un moyen puissant pour éviter d'écrire du code.

```
for(int h=0;h<listOfMetrics.length;h++){
   if((listOfMetrics[h].getModifiers() & Modifier.PUBLIC) //méthode public ???
                == Modifier.PUBLIC
                && listOfMetrics[h].getParameterTypes()[0] //Premier parameter
                                                          // de type IEntity ?
                      ==IEntity.class&&
                      listOfMetrics[h].getParameterTypes().length==1) {
                                                  //Méthode avec un  seul paramètre
```



Doc. 12 : POM en version stand-alone

Pour le format XML, il a été décidé de réutiliser celui de Mohamed Rouatbi qui en avait déjà proposé. Pour chaque classe, il est calculé un ensemble de métriques dont une description peut être donnée. Ci-dessous est donnée une partie qui aide à comprendre le format XML.

```
<Objects>
	<Object>
		<Name>com.ibm.toad.cfparse.ClassFile</Name>
		<ObjectProperties>

			<Prop>
				<PropName>CBO</PropName>
				<ProValue>18.0</ProValue>
				<Description>
					Coupling Between Objects
				</Description>
			</Prop>
			<Prop>
				<PropName>DIT</PropName>
				<ProValue>1.0</ProValue>
				<Description>
					Depth Inheritance Tree
				</Description>
			</Prop>
			<Prop>
				<PropName>LCOM1</PropName>
				<ProValue>508.0</ProValue>
				<Description>
					Lack Of Cohesion In Methods Version 1
				</Description>
			</Prop>
```

```
                <Prop>
                        <PropName>LCOM2</PropName>
                        <ProValue>73.0</ProValue>
                        <Description>
                                Lack Of Cohesion In Methods Version 2
                        </Description>
                <Prop>
                        <PropName>WMC</PropName>
                        <ProValue>232.0</ProValue>
                        <Description>
                                Weight Method per Class
                        </Description>
                </Prop>
                        ….
        </ObjectProperties>
</Object>
<Object>
        …….
```

Le format CSV n'est disponible que pour les métriques unaires, soit la majorité des métriques présentes. C'est un tableau à double entrée, le format le plus simple pour retravailler les résultats avec sur un tableur. La première colonne contient le nom des classes, la première ligne contient le nom des métriques qui ont été calculées. La figure suivante est un exemple d'un fichier CSV. Les métriques calculées sont SIX, WMC et cohesionAttributes sur les classes AttrUtils, BadJavaError, ByteArray, CFUtils, CPUtils et InstrUtils.

```
Entities;SIX;WMC;cohesionAttributes;
AttrUtils;0.0;12.0;0;
BadJavaError;0.0;3.0;0;
ByteArray;0.0;9.0;0;
CFUtils;0.0;229.0;0.0;
CPUtils;0.0;221.0;0.23529411764705882;
InstrUtils;0.0;192.0;0.008333333333333333;
```

Le programme stand-alone se veut très simple pour l'utilisateur, mais très puissant. Il devient facile de calculer des métriques sur des grands systèmes et de retravailler avec les données après.

3.3.6. L'extension Eclipse

Pour l'intégration de POM à Eclipse, le travail a été plus conséquent. Le problème n'était pas au niveau de l'implantation. Il ne faut que très peu de code. Le problème d'Eclipse est qu'il offre beaucoup de choses. Eclipse est très complet au niveau des composants, mais un néophyte peut s'y perdre très vite

Travailler avec Eclipse implique des règles à apprendre et à respecter. Pour cela, il faut lire la référence suivante : Erich Gamma, Kent Beck - Contributing to Eclipse: Practices, Plug-Ins, Pattern ». Ce livre commence par le fait suivant : « Avant de pouvoir contribuer à Eclipse, vous allez d'abord lire six heures par jour, puis coder pendant une heure. ».

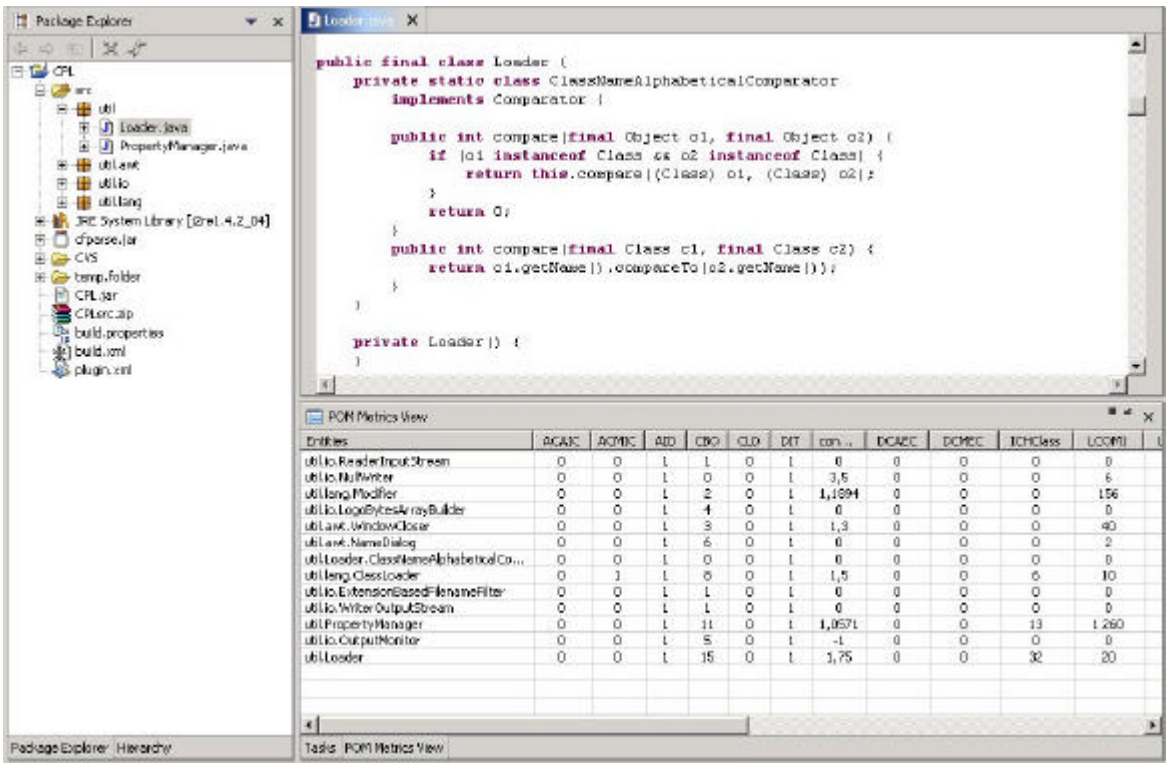
Eclipse utilise la bibliothèque graphique SWT pour dessiner les composants graphiques et les fenêtres. La philosophie est différente de la bibliothèque standard Swing, disponible dans le toolkit de développement de JAVA.

L'une des règles à suivre avec Eclipse est d'étendre et non pas de refaire ce qui a été déjà fait. En suivant ce principe, nous avons repris un plug-in existant pour l'étendre et ainsi y ajouter POM comme contribution. Il existe un projet sur le site SourceForge.net et qui permet de calculer des métriques de qualité. Notre contribution est que POM est plus extensible et demande très peu de code pour ajouter une métrique.

La démarche adoptée a été d'analyser ce plug-in. Comprendre sa structure est déterminante. Quelques jours à lire la documentation, les codes sources suffisent en principe. Sachant que l'intérêt est d'ajouter seulement des interfaces graphiques, le package net.sourceforge.metrics.ui sera le seul concerné.

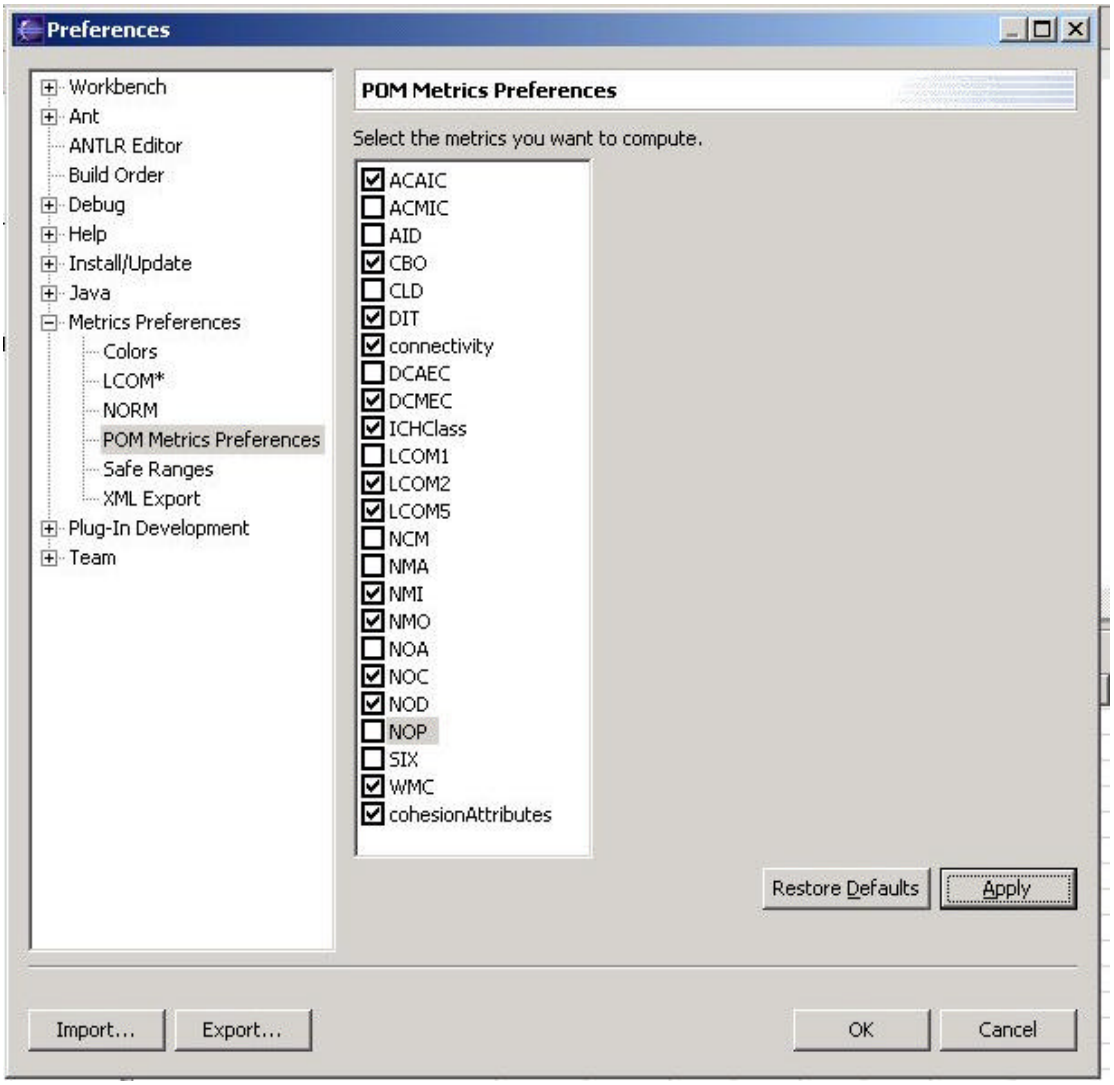
Pour le développement de plug-ins Eclipse, il est à noter que la plate-forme dispose d'un environnement de test: le run-time Workbench. Il est possible d'y charger ou d'y débugger des plug-ins en cours de développement. L'un de ses défauts est qu'il est lourd à lancer. Si le code source venait à changer, il faut obligatoirement le relancer pour prendre en considération les changements.

Utiliser le tableau à double entrée est un bon moyen de visualiser les données.



Doc. 13 : présentation des données sous Eclipse

Un autre écran consiste à choisir les métriques que l'on veut calculer. Celui-ci s'intègre dans les préférences d'Eclipse.



Doc. 14 : préférences pour le choix des métriques

Utilisation du plug-in POM :

Le plug-in calcule uniquement les métriques sur un des projets JAVA présents dans Eclipse. Pour l'utiliser, l'utilisateur doit activer le calcul des métriques pour un projet donné. Il peut à tout moment choisir les métriques à calculer. Une fois celles-ci choisies, il lance le calcul à l'aide d'un bouton présent dans la vue POM Metrics. L'étape prend de quelques secondes à une heure, selon la taille du projet considéré. Puis les résultats sont affichés. Il est alors possible de les exporter dans le même format XML décrit précédemment.

BILAN DU PROJET

L'architecture en trois couches est très extensible. Chaque composant peut facilement être changé avec un autre. Un nombre important de métriques est disponible. POM n'est pas qu'une API. C'est un outil disponible en deux versions : stand-alone et extension à Eclipse.

A ses débuts, POM a servi pour représenter des programmes objets sous forme d'un modèle physique. Ce travail, effectué par Mohamed Rouatbi, s'inscrit dans le cadre de sa maîtrise à l'Université de Montréal. Dans le modèle, chacune des classes est un objet 3D. Elles sont reliées par des ressorts. Leur tension varie en fonction du couplage entre les deux classes. D'autres paramètres, liés au domaine de la mécanique, sont représentés en considérant les métriques. L'ensemble de ce travail a été présenté à la conférence ECOOP 2004 à Oslo.

La seconde utilisation de POM entre dans le cadre dans la détection de design patterns dans des programmes. Après calcul d'un ensemble de métriques sur des grands systèmes, des algorithmes d'apprentissage sont utilisés pour découvrir des patterns dans les valeurs. On peut ainsi caractériser un design pattern par un ensemble de valeurs. Une publication a été soumise à ce sujet, co-signé par Yann-Gaël Guéhéneuc, Houari Sahraoui et moi.

Le projet est à un stade avancé. Mais il reste du travail pour l'améliorer. L'une des premières choses à faire serait de créer un plug-in Eclipse à part. En effet, le manque de temps a contraint à opter pour la solution de facilité, soit la réutilisation d'un plug-in déjà existant. Dans le futur, il est préférable d'avoir un plug-in qui évolue selon nos choix.

Le temps fait défaut à tous les projets. Il a été nécessaire de fournir des fonctionnalités simples à l'utilisateur. On peut penser à des fonctions plus avancées par la suite, notamment intégrer certains travaux de recherche sur la qualité. Nous pouvons citer par exemple l'interprétation des résultats obtenus après calcul. Ceci s'écarte un peu du cadre du projet.

Il existe des centaines de métriques. D'autres viendront. Il ne faut pas qu'elles restent un état de l'art. Les utilisateurs doivent pouvoir exprimer leurs métriques simplement. C'est la partie sur l'éditeur d'expressions qui a été mise de côté. Enfin dans le même sens, le méta-modèle PADL qui est utilisé est disponible en open source. Il est donc possible de le compléter par des informations pertinentes dont la granularité varie en fonction des besoins. Par exemple, la complexité cyclomatique n'est pas calculable à l'heure actuelle.

BILAN HUMAIN

Tout d'abord, il est gratifiant de voir que l'on a participé à un projet en équipe et que celui-ci est actuellement utilisé. La transmission de savoir n'était pas la première chose à laquelle je pensais en venant en stage. Mais il est toujours agréable de communiquer des connaissances à d'autres personnes, et de plus dans un domaine que l'on adore.

J'avais déjà effectué un stage dans un laboratoire universitaire. Dans ce stage, j'ai pris plus de temps pour découvrir la recherche. J'ai découvert aussi certains aspects de la qualité logicielle. C'est un domaine encore jeune, mais plein d'avenir.

Enfin, ce que je retiendrai le plus cette année sera Montréal, d'abord passée à l'Université de Montréal, puis en stage au CRIM. C'est une ville exceptionnelle, où les gens vous donnent l'envie de travailler, mais aussi de vivre. Je dédie donc une courte partie de ce rapport à cette ville qui restera dans mon cœur ;-).

BIBLIOGRAPHIE

Sites Internet :

Centre de Recherche Informatique de Montréal : http://www.crim.ca

Association d'entraide des développeurs francophones – www.developpez.com

JAVA – http://java.sun.com

Document interne du CRIM :

El Hachemi Alikacem – Modélisation des métriques pour un modèle générique

Publications

Lionel Briand, John W. Daly, and Jürgen Wüst- A Unified Framework for Coupling

Measurement in Object-Oriented Systems

Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi - Fingerprinting Design Motif Roles

Livres :

Au cœur de JAVA 2 – Fonctions avancées – Cay S. Horstmann & Gary Cornell

Modélisation objet avec UML – Pierre-Allain Muller & Nathalie Gaertner

JVM Specification version 2

INDEX DES DOCUMENTS