Ptidej

Pattern Trace Identification,
Detection, and Enhancement
in Java

IFT3051 – Projet d'informatique

Intégration de l'interface graphique du Ptidej à la plate-forme Eclipse

Projet présenté par :
Driton Salihu et
Lulzim Laloshi

Responsable du projet :
Yann-Gaël Guéhéneuc

Université de Montréal

Département d'informatique et de recherche opérationnelle,
Université de Montréal
15 août 2004

TABLE DE MATIÈRES

1.1. Contexte

À ce jour et à notre connaissance, il n'existe aucun outil qui permet d'obtenir automatiquement un modèle abstrait précis à partir du code source d'un programme. Même les outils industriels comme Rational Rose sont incapables de retro-concevoir le code source d'un programme précisément et se contentent d'en fournir une représentation graphique.

Un ensemble de définitions a été développé et implanté dans un outil, Ptidej UI, qui permet d'analyser statiquement le code source d'un programme et d'y identifier des relations abstraites entre classes, telles que les relations d'association, d'agrégation et de composition. Cet outil permet aussi d'appeler un solveur de contraintes pour identifier les microarchitectures similaires à la solution d'un patron de conception donné.

Cependant, l'intégration de cet outil avec l'environnement de développement Eclipse est encore très limitée et ne permet pas son utilisation aisée habituelle.

1.2. Travail demandé

Le travail consiste à prendre l'implantation de l'interface graphique de Ptidej dans Eclipse et à la développer pour atteindre une intégration complète et pour implanter toutes les fonctionnalités disponibles (y compris l'ajout d'informations dynamiques et l'appel au solveur de contraintes). Cela veut dire que l'utilisateur peut à tout moment, lorsqu'un projet Eclipse est ouvert, choisir un fichier .jar ou .class et l'ajouter pour analyser sa structure et ses relations incluant la généralisation, l'association, l'agrégation et la composition. Tout cela peut se visualiser à l'aide d'un éditeur de diagrammes directement dans Eclipse.

1.3. Le but de cette présentation

Nous espérons fortement que cette présentation sera utile à ceux qui continueront à travailler sur ce projet comme une introduction rapide et efficace pendant ce voyage dans le monde de modules d'extensions d'Eclipse.

2. ENVIRONNEMENT DE TRAVAIL

L'environnement de travail utilisé, comme nous l'avons précisé dès le début du projet, est la plate-forme Eclipse 2.1 sous Windows XP. Néanmoins, le projet est tout à fait portable sur d'autres systèmes d'exploitation, en fait pour tous les OS pour lesquels Eclipse a été développé. Nous avons choisi de travailler sur cette version d'Eclipse du fait qu'elle était considérée comme la version la plus stable. Il faut souligner que nous l'avons testé avec la version 3 d'Eclipse et qu'il existe une incompatibilite que nous la traiterons ci-dessous dans les discussions.

3. CONCEPTS THÉORIQUES

3.1. Qu'est-ce que Eclipse ?

Eclipse est un environnement de modélisation et de développement générique, ouvert et extensible.

- Eclipse est générique puisqu'il permet le développement peu importe le langage utilisé (Java, C/C++, Cobol, XML/XSL, UML...) sur de nombreux systèmes d'exploitation (Linux, Windows, Solaris, QNX, AIX, HP-UX, Mac OSX) ;
- Eclipse est une plate-forme ouverte puisqu'elle est offerte sous licence *Common Public License*, c'est-à-dire que le code source est libre de redevance n'importe qui peut le redistribuer et les travaux dérivés sont autorisés ;
- Eclipse est également extensible puisque c'est un framework permettant de construire et d'intégrer des outils de développement de toute nature.

Eclipse vient avec un ensemble de modules et de bibliothèques servant à la gestion des ressources. On peut créer des projets, éditer et sauvegarder des fichiers, imprimer, partager des ressources, gérer les versions à l'aide d'une interface CVS (*Concurrent Versions System*) intégrée, etc. Eclipse offre aussi de façon standard un environnement de développement Java et des outils de développement de modules d'extension.

L'environnement de développement Java, qui n'est qu'un ensemble de modules d'extension (appelé JDT pour *Java Development Tooling*), offre un éditeur spécialisé, une compilation incrémentale, un débogueur et différents services tels que le *code completion*, des *code templates* et le *refactoring*.

Ce qui distingue Eclipse des autres IDE est l'extensibilité de son environnement. Eclipse a été conçu de manière à pouvoir facilement étendre ses fonctions à l'aide de modules d'extension tout en conservant une interface graphique cohérente. On peut ainsi charger différents modules dans Eclipse pour le développement en tout genre comme Java, C/C++, Cobol, XML/XSL, UML, etc.

Développé initialement par la compagnie IBM, Eclipse est maintenant pris en charge par un consortium de plusieurs grandes compagnies qui s'engagent à utiliser la plate-forme Eclipse, à construire des modules d'extension gratuits et commerciaux pour Eclipse et à contribuer au développement du projet Eclipse et le supporter publiquement. Les membres initiaux du projet (novembre 2001) sont Borland, IBM, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft et Webgain auxquels se sont ajoutés entres autres Oracle, SAP, HP, Hitachi, Fujitsu et Sybase.

3.2. L'environnement Eclipse

Eclipse fournit un modèle d'interface utilisateur commun pour l'utilisation des outils. Il est conçu pour s'exécuter sur plusieurs systèmes d'exploitation tout en procurant une intégration robuste au système d'exploitation sous-jacent. Il est, en effet, développé en Java, mais n'utilise ni Swing ni AWT, mais SWT, c'est-à-dire les widgets du système d'exploitation. En fait, SWT est une surcharge des widgets de l'OS, si pour un OS le widget existe alors le OS appelle le widget natif sinon SWT émule le widget manquant. Les modules d'extension peuvent se programmer sur les API portables d'Eclipse et s'exécuter sans changement sur les différents systèmes d'exploitation supportés.

Eclipse permet un développement rapide des dispositifs intégrés basés sur un modèle de modules d'extensions. Le corps d'exécution de la plate-forme implante le moteur d'exécution qui démarre la base de la plate-forme et découvre les modules d'extension de manière dynamique. En effet, Eclipse scanne à chaque démarrage le répertoire de modules d'extension pour voir si de nouveau y ont été ajoutés.

3.3. La terminologie d'Eclipse

3.3.1. Le module d'extension

Eclipse est conçu pour être un outil modulaire. De nombreux modules d'extension sont fournis avec Eclipse et il est aussi très facile d'en ajouter d'autres développés par la communauté ou des sociétés commerciales. L'installation d'un module d'extension doit être très simple car elle consiste à dézipper l'archive qui contient le module d'extension dans le répertoire des modules d'extension où est installé Eclipse. Un module d'extension est un composant structuré, se décrivant lui-même au système à l'aide d'un manifeste (plugin.xml).

Les modules d'extension contribuent à la fonctionnalité de la plate-forme et peuvent fournir un support pour l'édition ou la manipulation d'autres types de ressources, tels que des fichiers, des programmes, des documents ou des pages. La plate-forme conserve un registre des modules d'extension installés et des fonctions

qu'ils procurent. Les modules d'extension sont ainsi regroupés dans un fichier d'archive de modules d'extension, et décrits à l'aide de leur fichier manifeste.

3.3.2. Le manifeste

Après la création d'un module d'extension, l'éditeur de manifeste apparaît pour le nouveau module d'extension. Ce fichier peut contenir des définitions pour l'exécution du module d'extension, des définitions des modules d'extensions requis par celui-ci, des déclarations de tous les points d'extension qu'il offre aux autres modules d'extension et la configuration des extensions.

3.3.3. Le workspace

Le workspace est l'entité qui permet de conserver les projets et leur contenu. Physiquement c'est un répertoire du système d'exploitation qui contient une hiérarchie de fichiers et de répertoires. Il y a un répertoire pour chaque projet à la racine du workspace.

Dans Eclipse, on peut créer différents types d'entités qui seront stockées dans le workspace :

- des projets;
- des répertoires pour organiser les projets;
- des ressources de différents types qui sont des fichiers.

3.3.4. Le workbench, perspectives, vues et éditeurs

Le workbench est composé de perspectives dont plusieurs peuvent être ouvertes mais une seule est affichée en même temps. À l'ouverture par exemple, c'est la perspective Ressource qui est affichée (qui ouvre par défaut les vues Navigateur, Structure et Tâches, ainsi qu'un éditeur qui permet d'éditer une ressource sélectionnée dans la vue Navigateur). Une perspective est ainsi composée de sous fenêtres qui peuvent être de deux types :

- les vues;
- les éditeurs.

Une vue permet de visualiser et de sélectionner des éléments. Plusieurs vues différentes peuvent être assemblées dans une même sous fenêtre. L'accès à chaque vue se fait grâce à un onglet. Un éditeur permet de visualiser et de modifier le contenu d'une ressource. Un éditeur peut contenir plusieurs ressources, chacun étant identifié par un onglet.

3.3.5. PDE

L'environnement de développement de modules d'extension (PDE) est un outil conçu pour simplifier le développement de modules d'extension à partir du plan de travail de la plate-forme PDE fournit un ensemble d'extensions (vues, éditeurs, perspectives, etc.) qui rationalisent le processus de développement de modules d'extension à l'intérieur du plan de travail.

Un projet PDE est un projet qui "sait" qu'il héberge un module d'extension et peut effectuer des opérations spécifiques sur les modules d'extension. Même si la présence effective d'un manifeste suffit pour utiliser bon nombre de fonctions PDE, la tendance est en faveur des projets de module d'extension Java car la plupart des modules d'extension sont écrits en Java et PDE repose sur la plate-forme et le JDT. L'une des fonctions les plus utiles dans PDE est liée à la détermination et à la gestion du chemin d'accès aux classes Java pour les modules d'extension et aux fragments en cours de développement.

3.3.6. Les points d'extension

Une fonction est ajoutée au système à l'aide d'un modèle d'extension. Les points d'extension sont des points de fonction bien définis dans le système qui peuvent être étendus par des modules d'extension. Il s'agit pour simplifier des fonctions publiques que les autres modules d'extensions peuvent utiliser. Lorsqu'un module d'extension contribue à l'implantation d'un point d'extension, on dit qu'il contribue à la plate-forme Les modules d'extension peuvent définir leurs propres points d'extension, de telle sorte que d'autres modules d'extension puissent s'intégrer étroitement à eux. Les extensions sont toujours écrites en langage Java à l'aide des API de la plate-forme

La plate-forme Eclipse est structurée sous forme d'un moteur d'exécution et d'un ensemble de fonctionnalités supplémentaires installées en tant que modules d'extension de la plate-forme. Les modules d'extension contribuent à la fonctionnalité de la plate-forme en contribuant aux points d'extension prédéfinis.

Dans sa forme la plus simple, la déclaration d'un point d'extension se résume à quelques lignes. Elle définit l'entité du point d'extension. Toute autre information attendue par le point d'extension est spécifique à celui-ci et documentée ailleurs. Les extensions constituent le mécanisme qu'utilise un module d'extension pour ajouter de nouvelles fonctions à la plate-forme Les extensions ne peuvent être créées arbitrairement. Chaque extension doit se conformer à la spécification du point d'extension qu'elle étend.

3.3.7. SWT :

SWT (*Standard Widget Toolkit*) est un outil destiné aux développeurs Java qui fournit une bibliothèque graphique transférable commun et en l'implantant sur chaque plate-forme, chaque fois que possible à l'aide de widgets natifs. Une application développée en SWT n'utilise donc ni les composants AWT ni les composants Swing, mais ceux du système d'exploitation. Ceci permet au toolkit de refléter immédiatement les modifications apportées à la présentation de l'interface graphique du système d'exploitation sous-jacent, tout en maintenant un modèle de programmation cohérent sur toutes les plates-formes.

3.4. CVS et Eclipse

CVS (*Concurrent Versions System*) est un outil libre de gestion des versions initialement développé pour Unix. Toutes les données sont stockées dans un référentiel. Chaque modification d'une ressource gérée par CVS associe à cette ressource un numéro de révision unique. Une version contient un ensemble de ressource, chacune ayant une révision particulière pour la version correspondante. CVS ne verrouille pas ces ressources. Deux développeurs peuvent créer chacun une révision d'une même ressource. La fusion des deux versions est à la charge d'un des développeurs. Eclipse propose une perspective pour utiliser CVS.

4. LA CONCEPTION DE PTIDEJ ET APPROCHES

4.1 La structure d'un module d'extension

La structure des modules d'extension que nous présentons est celle créée par défaut lorsque l'on utilise l'assistant de création de modules d'extension. Cette organisation semble être une bonne base qu'il est conseillé d'utiliser. De plus, certains éléments y sont obligatoires.

4.1.2. Dossier bin

Ce dossier contient les fichiers *.class et *.jar. C'est-à-dire le module d'extension compilé.

4.1.3. Dossier src

Ce dossier contient les fichiers sources *.java du module d'extension.

4.1.4. Dossier schéma

Ce dossier contient les schémas de point d'extension.

4.1.5. Éléments du module d'extension à la racine

Les fichiers liés au projet tels que les icônes, les fichiers d'aide et autres se trouvent à la racine du dossier contenant le module d'extension. Les trois fichiers suivants s'y trouvent également.

4.1.6. Le fichier .template

Ce fichier contient les informations qui s'affichent sur la page bienvenue de l'éditeur de manifeste.

4.1.7. Le fichier build.properties

Ce fichier contient les informations nécessaires à la compilation

4.1.8. Le fichier plugin.xml

C'est le fichier le plus important du module d'extension. C'est dans celui-ci que sont définies toutes les informations concernant le module d'extension ainsi que les points d'extensions qu'il utilise et qu'il fournit. Éclipse fournit une interface graphique permettant d'éditer ce fichier sans avoir à manipuler directement son code source (interface décrite dans le chapitre suivant). Il est néanmoins intéressant d'en connaître la structure et le contenu :

la première ligne est l'entête XML classique :

```
<?xml version="1.0" encoding="UTF-8"?>
```

une balise plugin :

```
   <plugin
          id="idduplugin"
          name="nom du plugin"
          version="1.0.0"
          provider-name="Université de Montréal"
           class="test.testPlugin">
```

la librairie d'exécution

```
<runtime>
      <library name="molj.jar"/>
</runtime>
```

les dépendances

```
<requires>
      <import plugin="org.eclipse.core.resources"/>
      <import plugin="org.eclipse.ui"/>
</requires>
```

Ensuite sont déclarées toutes les extensions
Voici par exemple comment est déclaré un éditeur de fichier *.e.

```
<extension point="org.eclipse.ui.editors">
      <editor
          name="Exemple d'éditeur XML"
          icon="icons/sample.gif"
          extensions="e"
          ContributorClass="org.eclipse.ui.texteditor.
               BasicTextEditorActionContributor"
          class="test.editors.XMLEditor"
          id="test.editors.XMLEditor">
     </editor>
</extension>
```

enfin une balise ferme le fichier

```
</plugin>
```

4.2. La conception de Ptidej

Vu la nature de ce projet (développement d'un module d'extensions) et ses caractéristiques particulières nous avons suivi une démarche particulière adaptée à la situation. Au lieu de faire la conception traditionnelle avec des diagrammes de classes UML, nous avons défini les classes à modifier et à implanter tout en expérimentant le développement des modules d'extensions sous Eclipse et pour Eclipse. Malgré que cette méthode soit peu conseillée dans le développement, nous avons utilisé cette approche car l'environnement Eclipse et le développement des modules d'extensions en particulier nous étaient inconnus.

Pour développer un module d'extension en général il faut d'abord identifier les points d'extension qu'on va utiliser pour l'implantation de notre module d'extension. Ainsi dans cette section nous décrirons les bibliothèques graphiques offertes par la plate-forme Eclipse utilisée dans notre projet. (Une description très détaillée se trouve dans l'article *Plugin Developer Guide* sur le site www.eclipse.com/)

4.2.1. Les vues

Le point d'extension org.eclipse.ui.views permet au module d'extension d'ajouter des vues au plan de travail. L'interface destinée aux vues est définie dans IViewPart, mais les modules d'extensions peuvent choisir d'étendre la classe ViewPart (comme dans notre cas)

plutôt que d'implanter une classe IViewPart à partir de rien. Les modules d'extension ajoutant une vue doivent l'enregistrer dans leur fichier plugin.xml et fournir des informations sur la configuration de cette dernière, telle que sa classe d'implantation, la catégorie (ou groupe) de vues à laquelle elle appartient, ainsi que le nom et l'icône à utiliser pour la décrire dans les menus et les libellés. En voici un exemple qui fait partie de notre fichier *plugin.xml* :

```
<extension
      point="org.eclipse.ui.views">
   <category
         name="Ptidej"
         id="ptidej.viewer">
   </category>
   <view
         name="Ptidej Diagram Control"
         icon="icons/Ptidej.gif"
         category="ptidej.viewer"
         class="ptidej.viewer.control.ControlView"
         id="ptidej.viewer.control.ControlView">
   </view>
</extension>
```

4.2.2. Les éditeurs

Un éditeur est une partie du plan de travail qui permet à un utilisateur d'éditer une ressource (dans notre cas c'est les fichiers *.class et *.jar en les analysant et en créant le diagramme de classes avec tous les relations entre les entités contenues dans ces fichiers). L'interface pour les éditeurs est définie dans IEditorPart, mais les modules d'extension peuvent choisir d'étendre la classe EditorPart plutôt que d'implanter IEditorPart à partir de rien. La classe du programme de contribution ajoute des actions liées à l'éditeur aux menus et à la barre d'outils du plan de travail. Elle doit implanter l'interface IEditorActionBarContributor.

Voici un example d'une telle extension:

```
<extension
   point="org.eclipse.ui.editors">
   <editor
      name="Ptidej class diagram"
      default="true"
      icon="icons/Ptidej.gif"
      extensions="ptidej"
      contributorClass="ptidej.viewer.editor.DiagramEditorContributor"
      class="ptidej.viewer.editor.DiagramEditor"
      id="ptidej.viewer.editor.DiagramEditor">
   </editor>
</extension>
```

4.2.3. Les extensions aux perspectives

Les modules d'extension peuvent ajouter leurs propres jeux d'actions, vues et raccourcis à des perspectives existantes avec le point d'extension org.eclipse.ui.perspectiveExtensions.

```
<extension
      point="org.eclipse.ui.perspectiveExtensions">
   <perspectiveExtension
         targetID="org.eclipse.ui.resourcePerspective">
      <view
            ratio="0.7"
            relative="org.eclipse.ui.views.TaskList"
            relationship="right"
            id="ptidej.viewer.control.ControlView">
      </view>
   </perspectiveExtension>
</extension>
```

4.2.4. Les menus contextuels

Le point d'extension org.eclipse.ui.popupMenus permet à un module d'extension de contribuer aux menus en incrustation des autres vues et éditeurs. Les classes qui définissent les actions de menus contextuels doivent implanter l'interface IObjectActionDelegate qui à son tour étend IActionDelegate. Nous avons implanté trois contributions pour afficher le menu contextuel sur les actions appropriées.

```
<extension
      id="org.eclipse.popAction.jarContribution"
      name="org.eclipse.ui.popupMenus"
      point="org.eclipse.ui.popupMenus">
   <objectContribution
         objectClass="org.eclipse.core.resources.IFile"
         nameFilter="*.class"
         id="Ptidej UI Viewer Eclipse.objectContribution1">
      <menu
            label="Ptidej"
            path="additions"
            id="Ptidej UI Viewer Eclipse.menu1">
         <separator
               name="Ptidej UI Viewer Eclipse.separator1">
         </separator>
      </menu>

      <action
            label="Add for analysis"
            class="ptidej.viewer.action.PopActionClass"
            menubarPath="Ptidej UI Viewer Eclipse.menu1/Ptidej UI
                        Viewer Eclipse.separator1"
            enablesFor="1"
            id="Ptidej UI Viewer Eclipse.action1">
      </action>
   </objectContribution>
   ....
</extension>
```

4.2.5. Les pages de préférences

Le point d'extension org.eclipse.ui.preferencePages permet d'ajouter des pages à la boîte de dialogue des préférences du plan de travail (Fenêtre/Préférences). La boîte de dialogue des préférences présente une liste hiérarchique des entrées des préférences utilisateur. Une fois sélectionnée, chaque entrée affiche la page de préférences correspondante.

La classe contenant la page de préférences doit implanter l'interface IWorkbenchPreferencePage. Le plan de travail utilise PreferenceManager pour maintenir une liste de tous les noeuds dans l'arborescence des préférences et leurs pages correspondantes.

```
<extension
      point="org.eclipse.ui.preferencePages">
   <page
         name="Ptidej Preferences"
         class="ptidej.viewer.preference.PreferencePage"
         id="ptidej.viewer.preference.PreferencePage">
   </page>
</extension>
```

5. LES RESULTATS OBTENUS

5.1. Les fonctions

Nous avons implanté des fonctions qui sont responsables de traiter les fichiers .class, .jar, .java et .ptidej ainsi que les repertoires et les paquetages et de les analyser avec l'outil Ptidej.

La première fonction consiste à choisir un fichier .class et de l'analyser avec l'outil Ptidej :

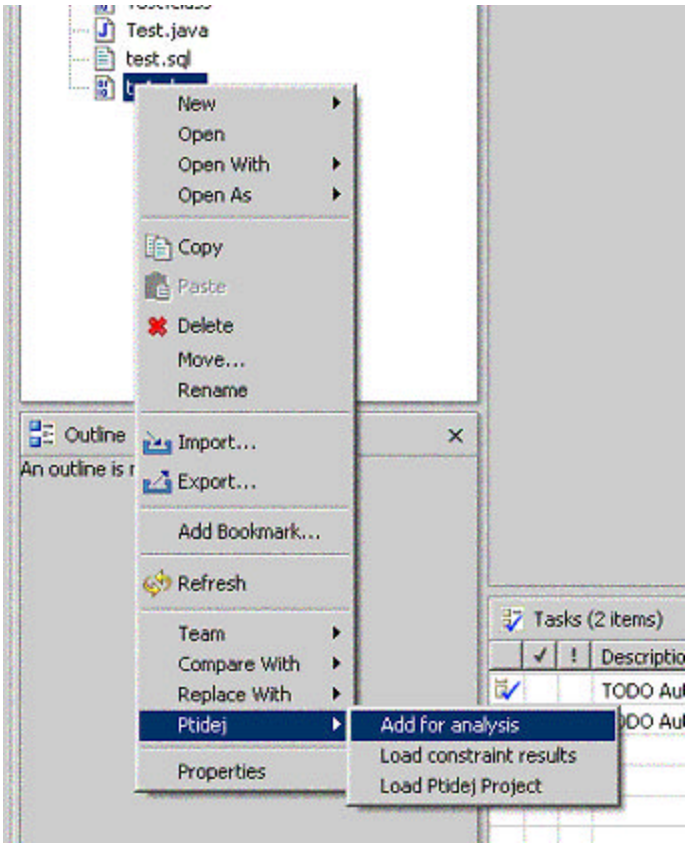

Fig. 1. Illustration de la fonction de l'analyse des fichiers de classes java.

1. Dans l'explorateur de paquetages sélectionner un fichier class du projet et puis cliquer avec le bouton droit de la souris.
2. Dans le menu contextuel choisir *Ptidej*
3. Sous *Ptidej* choisir *Add for analysis*

Le diagramme correspondant à la classe sera ouvert dans l'éditeur.

La deuxième fonction consiste à choisir un fichier .jar et à l'ajouter au projet à analyser. L'image ci-dessous illustre l'utilisation de cette fonction.

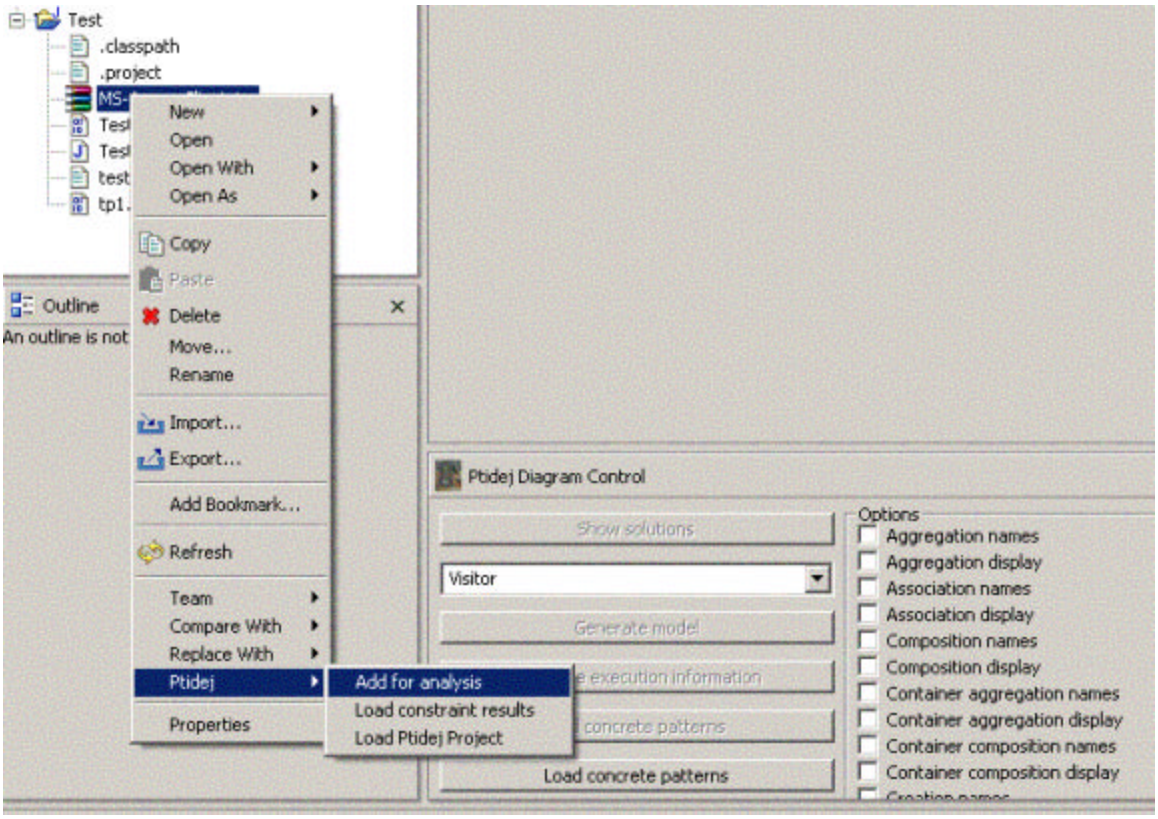

Fig. 2. Illustration de la fonction de l'analyse des jars

1. Dans l'explorateur de paquetages sélectionner un fichier jar du projet et puis cliquer avec le bouton droit de la souris
2. Dans le menu contextuel choisir *Ptidej*
3. Sous *Ptidej* choisir *Add for analysis*
4. Le diagramme correspondant au fichier jar sera affiché dans l'éditeur

La troisième fonction consiste à choisir un paquetage d'un projet ouvert dans l'explorateur de paquetages et à l'analyser avec Ptidej. Le mode d'utilisation de cette fonctionnalité est similaire à celle précédente.

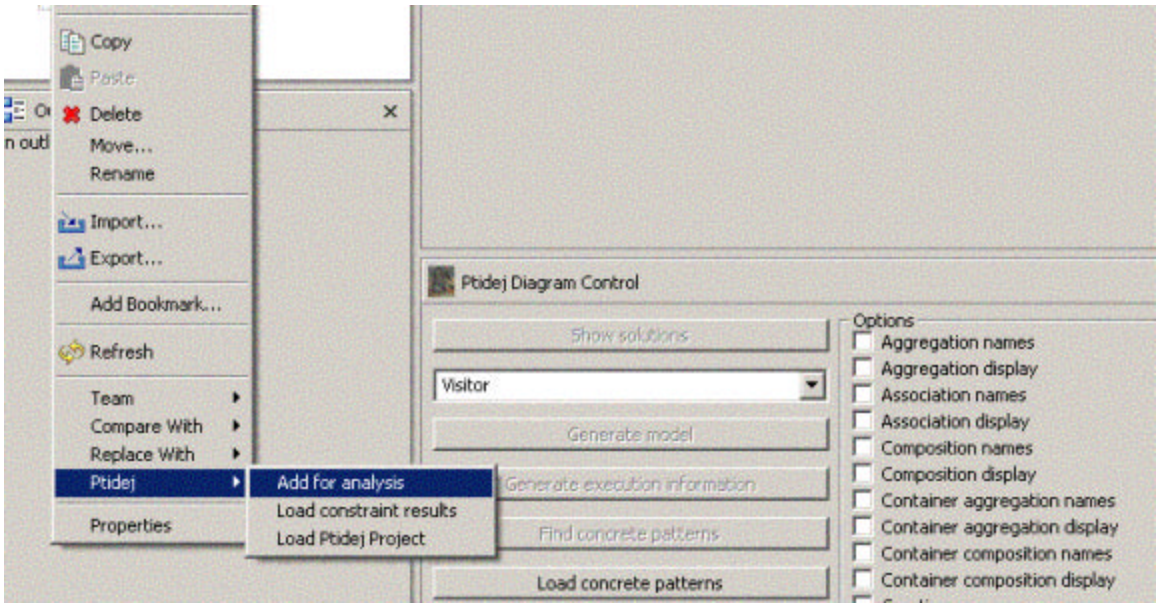

Fig. 3. Illustration de l'analyse des paquetages

1. Dans l'explorateur de paquetages sélectionner un paquetage du projet et puis cliquer avec le bouton droit de la souris.
2. Dans le menu contextuel choisir *Ptidej*
3. Sous *Ptidej* choisir *Add for analysis*
4. Le diagramme correspondant au contenu du paquetage sera affiché dans l'éditeur

La quatrième fonction traite les projets Ptidej qui sont des chemins d'accès vers un dossier de fichiers .class et .jar.

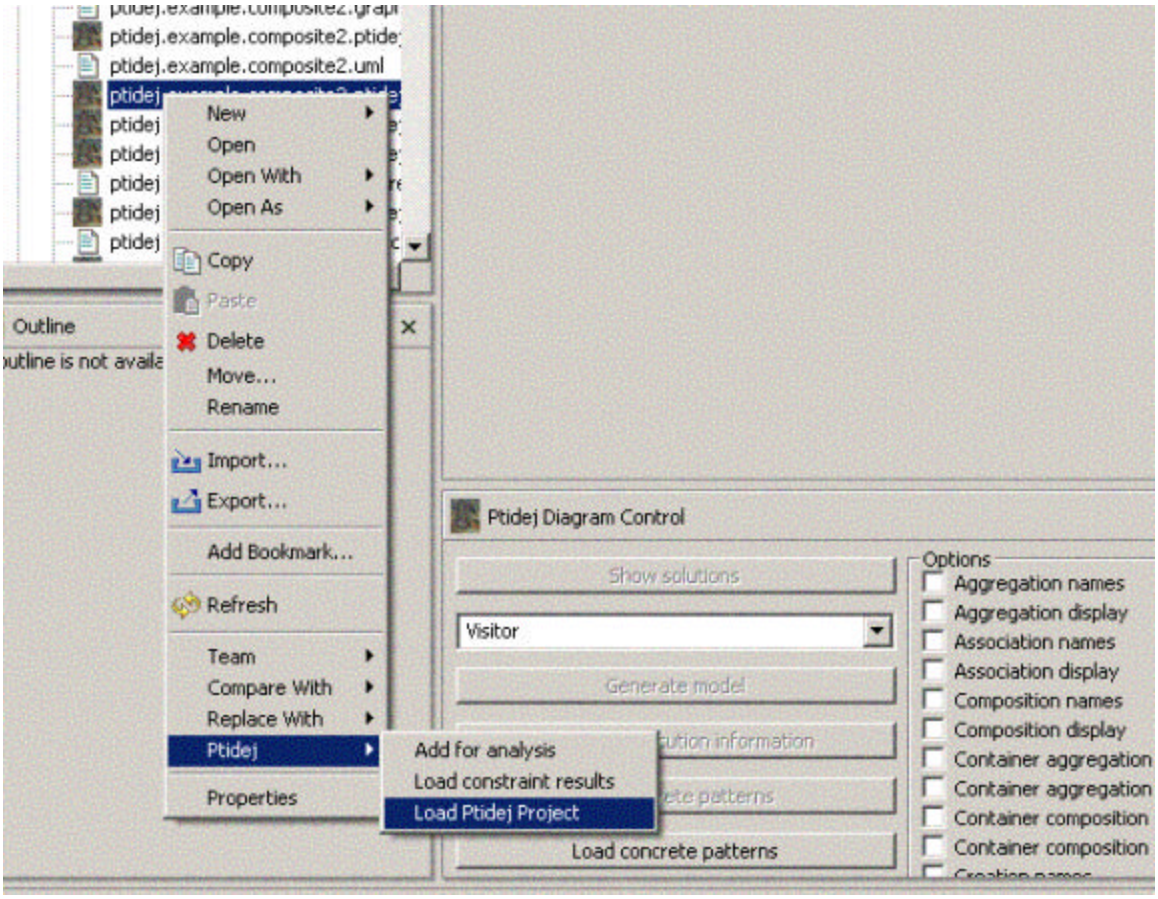

Fig. 4. Illustration de la quatrième fonction

1. Dans l'explorateur de paquetages sélectionner un fichier avec l'extension .ptidej du projet et puis cliquer avec le bouton droit de la souris.
5. Dans le menu contextuel choisir *Ptidej*
6. Sous *Ptidej* choisir *Load Ptidej Project*
7. Le diagramme correspondant au projet ptidej sera affiché dans l'éditeur

Lorsqu'une des fonctions décrites ci-dessus est exécutée, le modèle est crée et affiché dans l 'éditeur comme montré sur la figure suivante :

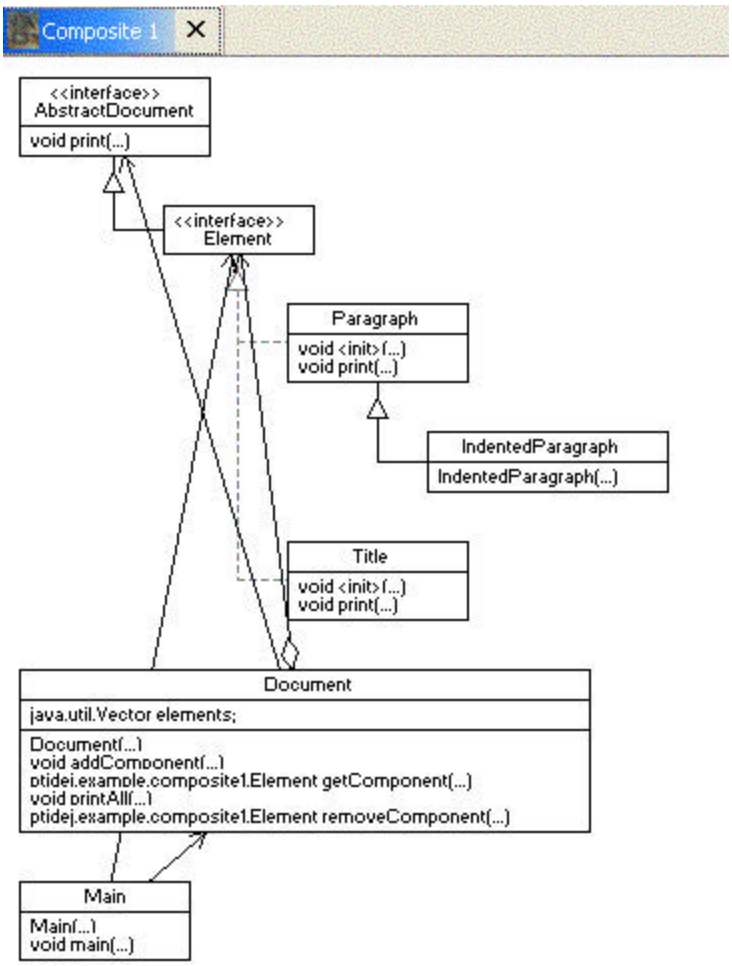

Fig. 5. La vue d'un projet ouvert avec *Ptidej*.

5.2. La description des classes

Les classes implantées sont responsables des actions présentées. Voici leur principe et leur implantation.

PopActionClass, PopActionJava et PopActionPtidej

Définissent les actions survenant lorsqu'on veut analyser un fichier *.class, *.java ou *.jar. Elles sont responsables de l'interaction directe avec les fichiers .class, .java et .jar. Lorsque l'option *Add for analysis* est choisie le chemin d'accès du fichier sélectionné est récupéré et l'éditeur de diagrammes dessine le diagramme correspondent au fichier sélectionné. Si un éditeur de diagrammes est déjà ouvert c'est dans celui-ci que le nouveau diagramme est dessiné en prennent en compte les changements apportés.

PopActionNewClass, PopActionNewJava et PopActionNewPtidej

Les classes précédentes analysent les ressources sélectionnées dans le même éditeur de diagrammes pour voir les relations qui existent entre déférents projets tandis que les classes ci hautes ouvre un nouveau projet (.class, .jar, .java ou .ptidej) dans un nouvel éditeur de diagrammes. Pour ce faire il faut choisir l'option *Add for new analysis* .

PopActionPackage et PopActionJavaPackage

Ces deux classes sont responsables pour l'interaction directe avec les répertoires contenant les fichiers .class et .java respectivement dans le navigateur. Elles rajoutent les fichiers sélectionnés dans l'éditeur de diagrammes actif. Si aucun éditeur de diagrammes n'est actif elle ouvre un nouveau.

PopActionNewPackage et PopActionNewJavaPackage

Comme les classes précédentes sauf qu'à chaque ajout de fichiers pour analyse on ouvre un nouvel éditeur.

AddFolderAction et AddJDTPackageAction

Ces deux classes gérèrent l'interaction directe avec les répertoires et les packages respectivement dans le navigateur (*Resource Perspective*). Lorsque l'utilisateur choisit d'analyser un répertoire, tous les fichiers .class contenus dans ce répertoire et dans les sous répertoires sont analysés. Tandis que pour les paquetages seulement le fragment de paquetage sélectionné est analysé.

AddNewFolderAction et AddNewJDTPackageAction

Ce sont deux classes qui gèrent les mêmes événements sauf que le répertoire ou le paquetage est analysé et ouvert dans un nouvel éditeur de diagrammes.

AddJDTPackage et AddNewJDTPackage

Elles servent d'analyser un paquetage java dans la perspective de ressources c'est-à-dire un répertoire de fichiers source java.

DiagramEditor

C'est l'éditeur des diagrammes qui s'occupe à aller chercher les ressources et d'en afficher la structure en forme de diagrammes de classes.

PtidejDiagramEditor

Fourni Ptidej avec les fonctionnalités qui le rendent compatible avec sa version antérieure Standalone. Il ne traite que les fichiers *.ptidej.

PtidejDiagramEditorCotributor et DiagramEditorContributor

Les contributeurs des éditeurs correspondants.

Explorer

Elle contient les méthodes nécessaires pour chercher les fichiers .class correspondants aux fichiers .java sélectionnés dans le navigateur des paquetages. Après avoir trouver le chemin d'accès du projet dans lequel est bcalisé le fichier .java, on parcours le système de fichiers en prenant comme racine le répertoire du projet jusqu'à ce qu'on trouve le fichier .class correspondant.

Util

C'est une classe qui contient des méthodes pour manipuler et traiter les chemins d'accès des ressources sélectionnés.

ClassFileFilter et ClassJarDirFilter

Des classes implantant l'interface java.io.FileFilter qui servent pour recuperer seulement les fichiers .class et/ou .jar lors de la recherche de fichiers .class correspondants aux fichiers java.

6. DISCUSSIONS

6.1. Les obstacles et les difficultés rencontrées

Dès le début du projet, nous nous sommes rapidement rendu compte que le développement de modules d'extension Eclipse n'était pas aussi facile qu'il paraissait. En effet, il faut tout d'abord prendre en main Eclipse, ce qui demande un certain temps d'adaptation. Mais, la difficulté principale réside ailleurs: s'il est facile de créer un module d'extension simple de type *Hello World*, faire un module d'extension plus avancé est une tache ardue à cause de la documentation Eclipse. Nous ne savons pas où chercher les composants à utiliser dans les bibliothèques d'Eclipse, ni comment faire interagir notre module d'extension avec l'environnement Eclipse.

Nous allons évoquer les soucis rencontrés au cours du développement de notre module d'extension qui nous ont empêchés dans l'avancement de notre module d'extension.

6.1.1. La recherche des documents pertinents sur les modules d'extensions

Cette tâche représente une consommatrice énorme de temps car ce n'est pas du tout facile de trouver des exemples et de la documentation sur le sujet. Nous avons étudié une panoplie des modules d'extensions simples qui comparé à Ptidej étaient trop simples, et également un module d'extension très compliqué qui requis beaucoup de temps d'analyse. Pour en nommer quelques uns : le fameux module d'extension Hello World, un éditeur de texte SQL et le code source du module d'extension UML d'Omondo pour Eclipse. Il y a une ressemblance fonctionnelle entre ce dernier et Ptidej, mais leurs implantations sont très différentes.

6.1.2. Le problème de chargement des jars

Nous avons consacré beaucoup de temps à ce problème sans pour autant trouver la solution définitive ce qui nous a retardé plusieurs semaines. Pour trouver cela il faut une connaissance parfaite de chaque ligne du projet ptdej ce qui difficile étant donné que nous n'étions pas les auteurs du code.

6.1.3. Le manque de temps

L'obstacle principal que nous avons dû affronter est le manque de temps pour réaliser les fonctionnalités demandées. Nous considérons que le temps fixé pour la réalisation de ce projet était insuffisant, étant donné que nous avons dû d'abord nous familiariser avec Eclipse et ensuite avec le développement des modules d'extensions en même temps que la planification du projet et l'établissement d'un échéancier raisonnable. Pour bien les maîtriser, il faut plus de temps et d'expérience avec de tels projets.

6.2. Les bénéfices apportés

Étant donné que la plate-forme Eclipse est devenue l'environnement de développement intégré de référence dans l'industrie et dans les milieux universitaires, les bénéfices tirés de ce projet sont significatifs et nous sommes certains que cela va nous aider dans les futurs développements envisagés. Malgré les contraintes de temps, nous avons réussi d'analyser le code et la structure des outils offerts par Eclipse et également le code source d'Eclipse lui même, ce qui nous a permis d'acquérir des nouvelles connaissances et de meilleures habitudes de programmation qui sont une base pour une bonne programmation. D'ailleurs c'est la première fois qu'on participe dans un projet d'une telle envergure et qu'on ait accès au code source d'un outil aussi avancé et extensible que Eclipse.

Après avoir travaillé sur ce projet, nous avons maintenant une meilleure approche de la résolution des problèmes divers concernant les applications aussi grandes, une meilleure planification, organisation et contrôle de développement du

projet, une expérience et une préparation pour les travaux futurs dans le développement et la maintenance des logiciels.

Ceci était également une occasion pour apprendre de nouvelles (bonnes) façons de programmer, d'améliorer et de pratiquer les bonnes méthodes de programmation en java comme la documentation du code, l'utilisation des normes pour les noms des interfaces par rapport aux classes, la déclaration des variables (final lorsque cela est requis) et bien sur l'utilisation de super-types et des interfaces pour déclarer les variables.

Personnellement, j'ai trouvé très intéressante et même importante la norme utilisée par Eclipse pour nommer les interfaces de classes qui doivent commencer par la lettre 'I' majuscule.

6.3. Suggestion

Comme mentionné dans le dernier paragraphe de la section précédente la norme utilisée dans l'Eclipse pour nommer les interfaces est une très bonne façon pour améliorer la facilité d'analyse et de modification du code, une norme qui sans aucun doute a une influence positive sur la qualité du logiciel et en particulier sur la maintenance.

Également, les autres règles d'organisation d'un workspace (la structure des paquetages et l'emplacement du code source et du code exécutable) seraient d'une grande utilité si elles étaient appliquées et enseignées par des professeurs de notre université surtout ceux qui enseignent la programmation.

6.4. Améliorations

Le projet est sujet d'améliorations possibles dans le futur en ce qui concerne l'intégration et le choix de certains algorithmes de recherche de classes correspondantes aux fichiers source java.

L'algorithme de recherche de fichiers .class correspondants aux fichiers contenus dans un paquetage suppose que la structure du projet à analyser suit certaines normes définis et acceptés dans l'environnement de développement Eclipse; cela veux dire que le code source d'un projet est localisé dans un répertoire nommé **src** et le code binaire se trouve dans un répertoire nommé **bin**. Cela est une structure bien connue dans le développement des logiciels à code source libre, mais cela n'étant pas obligatoire il se peut que d'autres structures sont utilisées pour certains projets.

6.5. Compatibilité

Une importance particulière concerne la compatibilité de Ptidej avec la version 3 d'Eclipse. Comme mentionne ci-dessus Ptidej a été développé dans Eclipse 2.1 et pour Eclipse 2.1, il existe une incompatibilité avec l'Eclipse 3 à cause des changements fait au méthodes qui servent d'ouvrir les éditeurs dans l'interface IEditorPart. Pourtant cette incompatibilité n'affecte que le développement c'est-à-dire que cela n'affecte pas la façon dont Eclipse 3 exécute les modules d'extension en forme exécutable. Eclipse 3 possède un mécanisme de «*binary runtime compatiblilty*» qui permet aux modules d'extension existants qui utilisent les méthodes supprimées openEditor ou openSystemEditor de continuer leur fonctionnement normal malgré ces changements dans l'API.

Nous avons réussi également de résoudre le problème de cette incompatibilité en effectuant des petits changements dans les classes qui gèrent les actions de l'utilisateur. Nous allons vous montrer comment éviter ce problème lors des développements futurs de Ptidej dans Eclipse 3. Le problème consiste dans les méthodes openEditor auparavant déclarées dans IEditorPart et que maintenant n'y sont plus présentes. Mais ces méthodes ont été définies dans une classe IDE comme des méthodes statiques avec une petite différence que nous pouvons les utiliser directement pour ouvrir un éditeur. En effet, dans Eclipse 2.1 un éditeur doit s'ouvrir de façon suivante :

```
IEditorPart editorPart = activePage.openEditor(
                                IFile fileToOpen,
                                String editorID)
```

Tandis que dans Eclipse 3 :

```
IEditorPart editorPart = IDE.openEditor(
                            IWorkbenchPage activePage,
                            IFile fileToOpen,
                            String editorID)
```

7. REMERCIEMENTS

Nous souhaitons remercier Yann-Gaël Guéhéneuc qui nous a permis de travailler sur ce projet et nous a accordé une grande autonomie.

8. BIBLIOGRAPHIE ET RÉFÉRENCES

Ouvrages :

- The Java Developers Guide to Eclipse, un groupe d'auteurs, Addison-Wesley
- Exemples sur le CD-ROM qui accompagne ce livre

Documentation sur internet :

- http://www.eclipse.org
  Site officiel d'Eclipse
- http://www.beyondcode.org/articles/eclipse-plugins.html
  Documentation sur les modules d'extensions Eclipse
- http://drdb.fsa.ulaval.ca/sujets/boisvertAlain/html/bk03.html
  Preséntation d'Eclipse
- http://www.eclipse-plugins.info
  Site officiel regroupant les modules d'extensions Eclipse