Rapport de PFE

Maître de stage : Yann-Gaël GUÉHÉNEUC

Tuteur : Guy CAPLAT

Elève : Sébastien COLLADON

REMERCIEMENTS

Je tiens à remercier le professeur Yann-Gaël GUEHENEUC, pour m'avoir accepté au sein de son équipe, ainsi que tous les membres du laboratoire pour leur accueil chaleureux, leur gentillesse et leur disponibilité tout au long de mon séjour parmi eux.

Je remercie également Mr Elod EGYED-ZSIGMOND responsable des relations internationales du département Informatique de l'INSA de Lyon, Mme Annick MELO responsable de la mobilité internationale de l'INSA de Lyon ainsi que tous ceux qui contribuent à nous donner la chance de partir étudier à l'étranger.

Je remercie également toutes les personnes de l'administration de l'Ecole Polytechnique de Montréal pour leur soutien, leur disponibilité, leur efficacité et leurs conseils.

Je remercie également toutes les personnes qui m'ont donné des conseils, émis des avis critiques, corrigé mon rapport et plus généralement toutes celles qui m'ont aidé à réaliser ce PFE.

Enfin, merci à toutes les personnes que j'ai rencontré à Montréal, qui m'ont fait apprécier mon séjour et font partie des choses qui me donnent envie d'y retourner.

TABLE DES MATIERES

Remerciements 2
Table des matières 3
Introduction 5
Présentation du stage 6
1. **École polytechnique de Montréal** 6
2. **Laboratoires d'accueil** 6
3. **Déroulement du stage** 7
4. **Poste de travail** 7
Le Projet 8
1. **Prérequis de lecture** 8
2. **Contexte** 8
3. **Planification des tâches du projet** 9
4. **PADL** 9
   4.1. A quoi sert PADL ? 10
   4.2. Comment instancier un modèle PADL ? 10
   4.3. Comment utiliser un modèle PADL ? 11
5. **Expression du besoin** 11
   5.1. Présentation du problème 11
   5.2. Exigences fonctionnelles 12
   5.3. Contraintes imposées et faisabilité technologique 12
   5.4. Configuration cible 13
6. **Étude de faisabilité** 13
   6.1. Outils étudier 14
   6.2. Résultats de l'étude de faisabilité 15
7. **Spécification** 16
   7.1. Exigences opérationnelles 16
   7.2. Exigences techniques 17
   7.3. Interfaces 17

7.4. Exigences concernant la conception et la réalisation 17
7.5. Préparation de la livraison 18
8. Conception 18
8.1. Choix d'architecture 18
8.2. Modèle des tâches de l'utilisateur 19
8.3. Cas d'utilisation 20
8.4. Implantation des données 20
8.5. Décomposition modulaire et hiérarchique 20
8.6. Modifications apportées à PADL 21
8.7. Construction du modèle structurel de l'application 23
8.8. Modélisation du comportement de l'application 24
9. Développement 25
9.1. Gestion d'un projet C++ dans Eclipse 25
9.2. Algorithme de création du modèle PADL 25
10. Test 26
10.1. Tests de non régression 26
10.2. Tests de recouvrement 26
10.3. Tests fonctionnels 26
11. Déploiement 26
12. Etat du projet, suites possibles 27
13. Difficultés rencontrées 27
Conclusion 28
Annexes 29
1. Références Bibliographiques 29

INTRODUCTION

Afin de valider un diplôme d'ingénieur de l'INSA de Lyon spécialisé en Informatique, il est nécessaire de terminer la formation par un projet de fin d'étude (PFE). J'ai choisi d'effectuer mon PFE à Montréal au Canada, le pays qui m'a accueilli pendant ma dernière année de cycle ingénieur en échange universitaire, au sein du département de génie logiciel et génie informatique de l'École Polytechnique de Montréal.

Le projet qui m'a été confié était de réaliser un générateur de modèle PADL pour le langage C++ avec le langage Java. Actuellement, le laboratoire est capable de créer un modèle PADL à partir du byte code généré par le compilateur Java. Le laboratoire utilise les modèles PADL comme base pour beaucoup de leurs outils et donc pour leur recherches. La réussite de ce projet permettrait au laboratoire d'élargir ses horizons de recherche.

Tout d'abord, je vais présenter mon environnement de travail. Ensuite j'expliquerai les différentes étapes de la réalisation de ce PFE en commençant par le contexte de ce projet, je présenterai le métamodèle PADL, le recueil des besoins, l'étude faisabilité effectuée au niveau du choix de l'outil d'extraction des informations du C++, les spécifications de la solution, la conception de la solution, le développement de cette solution, la stratégie de test ainsi que la stratégie de déploiement de cette solution. J'aborderai également quelles sont les poursuites envisageable de ce PFE ainsi que les difficultés rencontrées lors de sa réalisation.

PRESENTATION DU STAGE

1. École polytechnique de Montréal

L'Université de Montréal est une université publique francophone, fondée en 1878. Elle fait partie des principales universités canadiennes : c'est la deuxième du pays au niveau du nombre d'étudiants, du budget alloué à la recherche, et c'est la plus grande université francophone du monde hors de France.

L'École Polytechnique de Montréal est affiliée à l'Université de Montréal. Elle a été fondée en 1873 et compte actuellement près de 6 300 étudiants. Ses missions sont la recherche en génie et l'enseignement de l'ingénierie dans toute la diversité de ses domaines d'applications.

2. Laboratoires d'accueil

Le département de génie informatique et génie logiciel se situe dans le pavillon Claudette MACKAY-LASSONDE, datant de 2005, qui offre un environnement de travail très agréable.

L'équipe Ptidej (Pattern *Trace Identification, Detection, and Enhancement in Java)* est dirigée par Yann-Gaël GUÉHÉNEUC. Son but est de développer des méthodes et des outils pour évaluer et améliorer la qualité de programmes codés en langages orientés objets, en particulier grâce à l'utilisation de motifs et de patrons de conception. Des travaux majeurs sont l'étude de leurs impacts sur la qualité du code et la création de la suite d'outils Ptidej.

J'étais également sous la co-supervision du professeur Giuliano ANTONIOL, qui dirige l'équipe Soccer (SOftware *Cost-effective Change and Evolution Research).* En effet, ces deux professeurs ont décidé de travailler ensemble, répartissant ainsi leurs étudiants dans leurs différents locaux suivant le diplôme préparé. Pour ma part, j'ai été installé dans le laboratoire des étudiants en *Ph.D.* où la langue de travail était principalement l'anglais, aucun d'eux n'étant natif de la province de Québec et étant le seul français parmi ces étudiants.

3. Déroulement du stage

Ce PFE a été effectué du 4 janvier au 4 mai de l'année 2012 (4 mois pleins, ce qui fais 85 jours ouvrés). Au début du PFE, le sujet n'était pas encore clairement défini car Yann-Gaël GUÉHÉNEUC avait plusieurs sujets à me proposer. J'avais le choix entre 2 sujets. Le premier sujet était le développement d'un générateur de modèle PADL pour le langage C++. Le deuxième sujet était le développement d'un générateur de modèle PADL pour le langage C#. J'ai choisi le premier car c'était le projet qui représentait pour moi le défi le plus intéressant en termes de complexité.

4. Poste de travail

Le laboratoire est équipé de station de travail avec deux écrans chacune sous CentOS qui avait la possibilité de booter sur le réseau et de charger le répertoire utilisateur (la partition /home) de chaque personne du laboratoire. Toutefois, mon maître de stage m'a conseillé au début de mon stage d'utiliser mon ordinateur personnel pour travailler, car cela me permettrai de pouvoir travailler de chez moi en cas de problème (tempête de neige, maladie, etc.) et cela me permettrai également de ne pas être limité au niveau de la taille de stockage pour les informations du projet.

Tous les projets utilisés sont centralisés sur un serveur SVN. Seules les personnes travaillant dans le laboratoire ont un accès en lecture et en écriture sur tous les projets contenus dans ce serveur. Certains étudiants ont également le droit de lecture et parfois le droit d'écriture sur certains projets.

Tous les vendredis après-midi, le laboratoire organise des séminaires internes au laboratoire afin que chaque personne puisse présenter son travail, l'état d'avancement de ses recherches et que chacun puisse profiter du travail des autres ou que le travail de chacun profite des remarques des autres.

LE PROJET

1. Prérequis de lecture

Les prérequis pour la lecture de cette partie du document sont, suivant les niveaux de lecture :

- 1er niveau de lecture : des notions de conception orienté objets, de recueil du besoin, une base de culture informatique, une connaissance de l'environnement Java.
- 2ème niveau de lecture : la connaissance du langage de modélisation UML.

2. Contexte

Le laboratoire Ptidej analyse les programmes afin de trouver des principes ou des lois et ainsi poser les fondements de la compréhension d'un programme. Pour ce faire, Yann-Gaël GUÉHÉNEUC a créé un métamodèle permettant de représenter un programme développé avec un langage objet. Ce métamodèle a été développé en Java et fourni une représentation statique du code (comme les patrons de conception du GoF). Au début, celui-ci était fait principalement pour le langage java (et ses conventions de nommages). Il existait à mon arrivée un seul parseur qui était pour le langage Java, à partir du byte code généré. Il y a actuellement un parseur en développement pour le Java à partir des fichiers sources utilisant l'AST de Eclipse JDT et pour le C# à partir des sources avec ANTLR.

Pour pouvoir analyser toujours plus de programmes et faire des comparaisons entre les langages, les programmes et les développeurs, le laboratoire a besoin de pouvoir analyser d'autres langages. C'est pourquoi il m'a été proposé de travailler sur le langage C++. En effet, le langage C++ représente une mine de programmes open source analysables et donc un énorme potentiel de découverte pour le laboratoire.

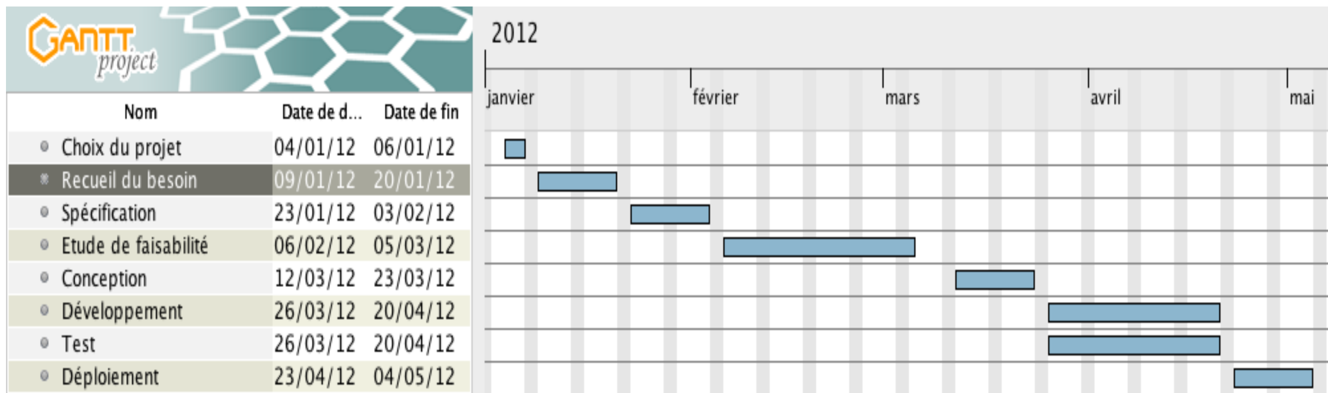
De plus, générer le modèle PADL pour les programmes écris en C++ représente un projet commencé en 2005. Ce projet a déjà été tenté par 4 équipes de travails différentes (dont mon maître de stage) sans jamais aboutir à une version exploitable. Etant donné que j'avais le choix entre effectuer un générateur de modèle PADL pour le langage C++ et un autre pour le langage C# (langage proche du java), j'ai opté pour ce qui selon moi représentait un défi en choisissant d'effectuer le générateur pour le langage C++.

De plus, créer un générateur de modèle PADL pour le langage C++ rentrait dans la continuité d'un cours que j'avais pris au premier semestre intitulé « Patron de conception pour la compréhension de programme ». En effet, nous devions lors des travaux pratiques de ce cours, réaliser un générateur de modèle PADL à partir d'un fichier XML généré avec un outil (chaque groupe

de travail utilisait un outil différent) qui prend en entrée un fichier source C++. La partie « Étude de faisabilité » de ce document explique pourquoi je n'ai pas continué l'un de ces projets.

3. Planification des tâches du projet

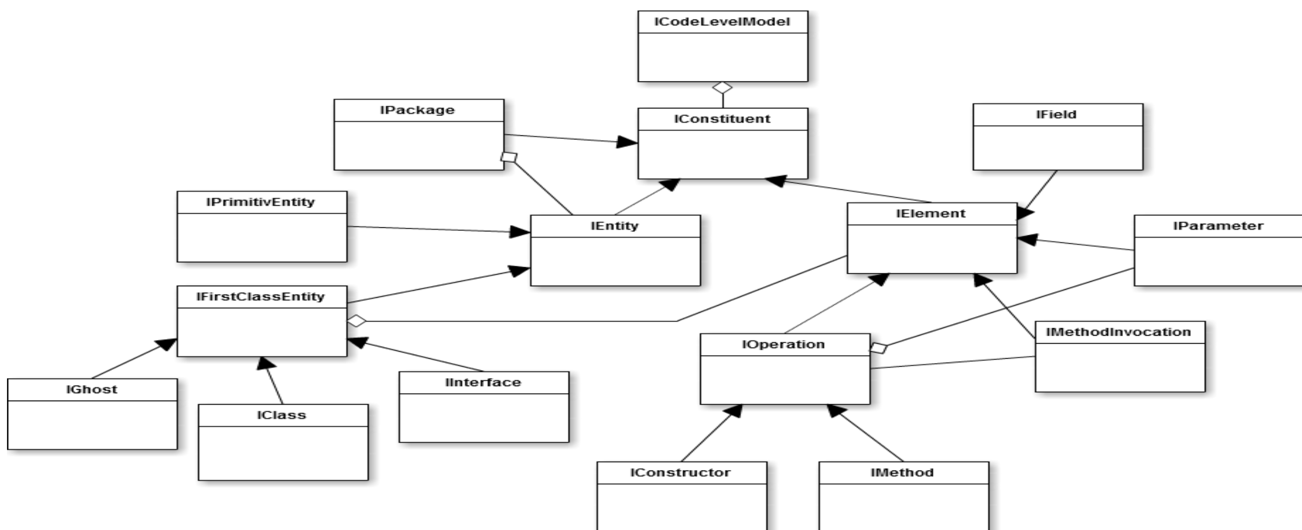
Voici le diagramme de Gantt effectif de la réalisation des tâches du projet :



4. PADL

PADL est un métamodèle de représentation d'un logiciel développé par un langage à objets. PADL est utilisé pour calculer certaines métriques sur un logiciel, détecter les motifs de patron de conception et d'autres choses. PADL est à la base de beaucoup d'articles scientifiques dans le laboratoire.

Le diagramme de classes simplifié de PADL est :



Le ICodeLevelModel est la classe conteneur de tout un programme, il contient les IPrimitiveEntity qui représentent les types primitifs d'un langage (ils sont stockés directement dans le code-level model pour éviter de les dupliquer dans les packages où ils sont rencontrés en paramètres de méthode ou en attribut d'une classe) et les IPackage (représentent les packages en Java, les namespace en C++) qui contiennent les IClass (classes en langages à objets), les IInterface (les interfaces en Java, les classes virtuelles pures en C++) et les IGhost. Un IGhost est une classe dont nous n'avons pas eu la déclaration, elle provient généralement d'une bibliothèque précompilée. Ensuite, les IField représentent les attributs d'une classe, les IParameter représentent les paramètres de méthodes et les IOperations représentent les méthodes (et constructeurs, destructeurs) d'une classe. Enfin, viennent les invocations de méthodes qui représentent les différents appels de méthode effectués dans une méthode (essentiel pour la détection de motifs de conception).

4.1. A quoi sert PADL ?

PADL est un métamodèle permettant d'avoir une représentation d'un logiciel développé avec un langage à objets. Cette représentation permet de détecter des métriques permettant de comparer des logiciels entre eux, de les classer et d'extraire des informations indépendantes de leurs fonctionnalités. Une des principales fonctionnalités du modèle PADL est de permettre une détection de motifs de conceptions[1] par le biais de Choco (un moteur de résolution de contraintes en Java). Nous n'expliquerons pas comment est utilisé Choco à partir de PADL pour trouver des motifs de conception car cela ne rentre pas dans le cadre du PFE.

4.2. Comment instancier un modèle PADL ?

Pour instancier un modèle PADL, il faut utiliser le langage Java. Un modèle PADL est représenté par un objet Java de type CodeLevelModel. Il est impossible de créer un objet CodeLevelModel directement, il faut utiliser l'usine (factory) fournie par le projet PADL. En effet, les classes sont privées, ce qui force à utiliser les interfaces (ICodeLevelModel dans ce cas). Ce procédé est le même pour tous les objets de PADL et permet de sous-classer facilement les objets de bases afin d'adapter le fonctionnement de ceux-ci à ses besoins en minimisant les changements dans le code concerné par ses changements.

Ensuite, il faut respecter certaines règles de créations pour le modèle. Les classes doivent être ajoutées au modèle par le biais d'un package, les méthodes et les attributs doivent être ajoutés à une classe. Une invocation de méthode doit être ajoutée à une méthode. Le modèle ayant été développé avec Java 1.2, celui-ci n'utilise pas les collections génériques ce qui implique que du mauvais code peut-être développé sans erreurs de compilation (ajouté une méthode a un package par exemple). Par contre, une vérification de type est effectuée à l'exécution et une exception est lancée si une des contraintes d'ajout d'un objet n'est pas respectée.

---

[1] Un motif de conception représente une instanciation d'un patron de conception. Cela signifie que le motif reprend les fonctionnalités de la solution préconisée par un patron de conception tout en étant possiblement moins strict sur les contraintes nécessaires pour que la solution soit validée. Ex : le patron de conception Composite avec l'héritage du type abstrait par une classe petite fille.

L'architecture représente très bien les programmes écrits en Java mais nécessite quelques ajustements pour pouvoir représenter les autres langages. Nous verrons lors de la conception que l'architecture de PADL à dû être assouplie pour pouvoir représenter le C++ plus justement.

4.3. Comment utiliser un modèle PADL ?

L'utilisation d'un modèle PADL est transparente pour l'utilisateur. Une fois le modèle obtenu, il suffit d'utiliser l'API fournie, en fournissant en argument le modèle PADL et en renseignant si nécessaire les autres paramètres, pour obtenir le résultat du traitement voulu sur le modèle. Lors de la phase de test, nous avons beaucoup utilisé le détecteur de patron de conception de cette manière.

5. Expression du besoin

5.1. Présentation du problème

5.1.1. But (éléments de motivations, intérêt du client, finalité), nature du logiciel, utilisateurs concernés

Cette application a pour but de créer le modèle PADL correspondant à n'importe quel programme écrit en C++. L'application doit permettre de récupérer à la fin du traitement le modèle PADL créé. Les utilisateurs seront essentiellement les personnes travaillant dans le laboratoire Ptidej.

5.1.2. Formulation des besoins, exploitation et ergonomie, expérience

Les besoins sont principalement liés aux fonctionnalités attendues pour le générateur de modèle PADL pour C++. Le générateur devra permettre un déploiement aisé sur n'importe quelle plateforme possédant une machine virtuelle Java.

5.1.3. Portée, développement, mise en œuvre

La portée du projet est assez grande car elle concerne le laboratoire d'accueil. Le développement doit être fait par l'étudiant. L'étudiant utilisera toutes ses connaissances acquises au niveau modélisation et savoir-faire technologique afin de mener à bien ce projet.

5.1.4. Limites

Le projet doit être achevé idéalement à la fin du PFE, ce qui correspond à la date du 4 mai 2012.

5.2. Exigences fonctionnelles

5.2.1. Fonctions de base, performances et aptitudes

Le générateur devra, à partir des sources d'un programme en langage C++, créer le modèle correspondant à ce programme.

5.2.2. Contraintes d'utilisation, norme de documentation

La contrainte d'utilisation est que le générateur doit pouvoir s'exécuter sur n'importe quelle machine possédant une machine virtuelle Java. La norme de documentation est la même que celle du laboratoire, c'est à dire un code source commenté selon les meilleures pratiques du laboratoire et un rapport expliquant toutes les étapes de la réalisation du projet.

5.2.3. Critères d'appréciation

Les critères d'appréciation identifiés sont les suivants :

- portabilité
- maintenabilité
- facilité de déploiement

5.2.4. Flexibilité, variation de coûts associée

Le générateur doit être flexible au niveau des modifications architecturales nécessaire à apporter au modèle PADL actuel.

5.3. Contraintes imposées et faisabilité technologique

5.3.1. Sûreté, planning, organisation, communication

La sûreté du projet devra être assurée par une relation étroite entre l'étudiant (Sébastien COLLADON) et son directeur de projet (Yann-Gaël GUÉHÉNEUC) afin de vérifier l'avancement du projet, permettre d'orienter l'étudiant en fonction des difficultés et de discuter des différentes évolutions du projet. Le directeur de projet assurera en quelque sorte le rôle de MOA (Maîtrise d'ouvrage) alors que l'étudiant assurera le rôle de MOE (Maîtrise d'œuvre).

La communication du projet a été effectuée par échange de courriels et par réunions d'avancement. Ces réunions d'avancements ont eu lieu les mardis de chaque semaine.

5.3.2. Complexité

Le projet présente plusieurs aspects qui ont une complexité différente :

- *Aspect traitement des données :* compliqué, dépend de l'outil utilisé
- *Aspect test :* plutôt compliqué dû à la difficulté de valider un modèle PADL

- *Aspect documentation :* simple, ce document ainsi que les commentaires et la Javadoc du projet

5.3.3. Compétences, moyens et règles

Le développement de ce générateur nécessite des compétences en programmation orientée objet, test, algorithmique et documentation.

Pour faire le développement, il sera nécessaire d'avoir un ordinateur avec une machine virtuelle Java installée.

5.4. Configuration cible

5.4.1. Matériel et logiciels

Le générateur devra pouvoir s'exécuter sur n'importe quelle machine. Le langage de programmation utilisé est Java pour pouvoir appeler l'API de PADL plus simplement.

5.4.2. Stabilité

Le générateur ne doit pas crasher suite à une erreur dans son fonctionnement. Si des erreurs sont rencontrées, cela doit provenir d'un problème au niveau des entrées et doit faire l'objet d'un message à l'utilisateur.

5.4.3. Interfaces

Le générateur ne possède aucune interface personne-machine.

6. Étude de faisabilité

L'étude de faisabilité menée n'était pas une étude de faisabilité du projet en lui-même mais une étude de faisabilité au niveau des outils permettant de récupérer les informations nécessaires à la création d'un modèle PADL à partir de C++. En effet, l'objectif pour le laboratoire était que le projet soit réalisé à la fin de mon stage et non pas que le projet soit seulement débuté. Cette étude de faisabilité cherche à trouver suivant certains critères quel outil utiliser.

Dans cette étude de faisabilité, nous nous sommes intéressés à peu près à tous les outils permettant de représenter un programme C++. Nous nous sommes donc intéressés aux générateurs XML et aux parseur fournissant un AST. L'objectif est de trouver une solution facilement déployable sur n'importe quel type d'ordinateur, sans contraintes de système d'exploitation, étant suffisamment souple, documenté pour permettre à quelqu'un d'autre de reprendre le développement effectuer grâce à cet outils. Ce qui peut se résumer par les critères suivants classés par ordre d'importance : portabilité, maintenabilité, facilité de déploiement.

J'ai donc été amené à étudier 3 générateurs XML de C++, 3 parseurs C++ et 2 IDE. Nous allons voir par la suite quel a été mon choix.

6.1. Outils étudier

6.1.1. GCCKDM

Le premier parseur XML était GCCKDM, celui permet de représenter un fichier source C++ avec le modèle KDM. Il fonctionne comme un plugin pour GCC 4.5.x. Seulement ce parseur nécessite beaucoup de transformation des fichiers de sorties afin de n'avoir que les informations intéressantes et ne permet pas d'accéder aux invocations de méthodes dans les fonctions.

6.1.2. SRCML

Le deuxième parseur XML était SRCML. Celui-ci permet une représentation XML d'un fichier source C++ à l'identique, ce qui permet de retransformer un fichier XML généré par SRCML en fichier source C++ sans perte d'information (au commentaire près). Etant donné la grammaire ambigüe du C++, cela rendait la tâche presque aussi compliquée que de réaliser un parseur C++. En effet, beaucoup de cas particulier sont apparus à l'utilisation et aucun parseur XML ne semblait favoriser réellement l'utilisation de cet outil. L'outil est disponible pour les systèmes d'exploitation Linux et pour Windows. La compatibilité ascendante n'est pas assurée par cet outil étant donné qu'elle est fonction de la compatibilité ascendante du langage.

6.1.3. GCCXML

Le troisième parseur XML était GCCXML. Ce parseur permet de créer une représentation XML du code source vu par un compilateur. Les compilateurs qu'ils utilisent sont GCC et Visual C++. Ce parseur nécessite beaucoup de préparation à l'utilisation car il faut spécifier à l'outil quelles classes parser au lancement afin d'évider que celui-ci ne rentre trop profondément dans le code et n'analyse la bibliothèque Math, par exemple. De plus, sa représentation est XML et, malgré une table des symboles servant d'index, le parsing est rendu assez ardu à cause de la grammaire ambiguë du C++.

6.1.4. Xtext

Xtext est un cadriciel pour le développement de langages de programmation et de langages dédiés. Il permet de créer un AST et parseur pour obtenir l'AST. Le problème est qu'il n'existe pas de grammaire pour le C++ (au moment de l'étude de faisabilité) utilisable par Xtext. Xtext est surtout utile lorsque l'on doit créer un nouveau langage car il fourni tous les outils de test et de développement intégrés à Eclipse. De plus, après étude, il s'est avéré que Xtext n'est pas adapté au C++ à cause de la grammaire ambiguë du C++ et de la conception de Xtext très orientée langages dédiés.

6.1.5. ANTLR

ANTLR est un cadriciel libre de construction de compilateur. Un des projets précédent de générateur C++ n'ayant pas abouti utilisait ANTLR pour parseur le C++. Le code de ce projet était

vraiment compliqué et erroné, ce qui aurait coûté du temps (trop) pour comprendre le code et pour le déboguer. Les grammaires proposées sur le site d'ANTLR pour le C++ n'apportent aucune garantie au niveau de la gestion des erreurs ni une bonne documentation de l'utilisation de celle-ci De plus, mon maître de stage n'était vraiment pas favorable à cette solution. Il existe une grammaire développée par l'équipe de Netbeans en 2008 (n'inclue pas les dernières évolutions du C++). Cette grammaire est libre mais doit être citée par les travaux qui l'utilisent.

6.1.6. JavaCC

JavaCC est un générateur de parseur pour Java. JavaCC permet de parser le C++ sans toutefois permettre de parcourir un AST (un jjtree pour JavaCC). Il produit du code pur Java ce qui garantit une exécution sur n'importe quel type de système possédant une JVM.

6.1.7. Netbeans

Netbeans est un IDE. Cet IDE permet de créer des programmes C++ et apporte plusieurs fonctionnalités telles que l'auto-complétion des attributs et des méthodes. Pour effectuer cette auto-complétion, l'IDE doit maintenir une représentation du code écrit dans le projet. Malheureusement, il n'a pas été trouvé de moyen simple d'accéder, par n'importe quel biais que ce soit (plugin, API), à cette représentation.

6.1.8. Eclipse CDT

Eclipse CDT est un IDE qui IDE permet de créer des programmes C++ et apporte plusieurs fonctionnalités comme l'auto-complétion des attributs et des méthodes. De plus celui-ci permet, grâce à l'architecture « plugin » d'Eclipse d'accéder à l'AST créée par le parseur de l'IDE. Eclipse, étant développé en Java, est utilisable aussi bien sur Linux, Windows que Mac. Par contre, la documentation à propos de l'utilisation et de la récupération de l'AST est faible. Elle ne présente qu'une documentation des classes mais aucun exemple d'utilisation. De plus, les choix architecturaux ayant évolué avec les versions, beaucoup d'informations disponibles sur l'Internet s'avèrent erronées et inutilisables, ce qui complique son utilisation.

6.2. Résultats de l'étude de faisabilité

	Portabilité	Maintenabilité	Déploiement
GCCKDM	0	- (beaucoup de code juste pour le parsing. Communauté de développeur active)	- (nécessite GCC)
SRCML	+	- (beaucoup de code juste pour le parsing. Communauté de développeur peu active)	+ (nécessite un script d'appel de l'outil en fonction de la plateforme)
GCCXML	0	- (beaucoup de code juste pour le parsing. Communauté de développeur peu active)	- (nécessite GCC)

Xtext	? (outil non approprié)	? (outil non approprié)	? (outil non approprié)
ANTLR	+	- (nécessite de comprendre la grammaire et le parseur. Communauté de développeur active)	++ (code Java)
JavaCC	++	- (nécessite de comprendre la grammaire et le parseur. Communauté de développeur active)	++ (code Java)
Netbeans	? (inutilisable)	? (inutilisable)	? (inutilisable)
Eclipse CDT	++	+ (pas de paramétrage, bonne documentation mais peu d'exemple d'utilisation de l'AST. Communauté de développeur active)	+ (code Java mais indécouplable de la plateforme Eclipse)

Suite à cette Etude de faisabilité, il a été mis en lumière que l'outil à utiliser pour parser le C++ et réaliser le projet était Eclipse CDT.

7. Spécification

Cette partie détaille les spécifications logicielles de ce PFE. Cette partie répond à la partie précédente de recueil des besoins.

7.1. Exigences opérationnelles

7.1.1. Environnement

Le générateur est complètement autonome, il ne possède que des interactions avec le système d'exploitation et l'utilisateur.

7.1.1.1. Environnement matériel

Le matériel cible doit permettre de traiter de très gros programmes cibles. Par conséquent, il est conseillé d'exécuter l'application sur un environnement matériel performant.

7.1.2. Environnement logiciel

Afin de pouvoir s'exécuter, le générateur doit être lancé dans un environnement possédant une machine virtuelle Java. Le générateur ne tient pas compte des erreurs présentes dans les fichiers fournis en entrés.

7.1.3. Exigences de performance

Le générateur doit être la plus rapide possible sans que cela n'ait un impact sur le modèle obtenu.

7.2. Exigences techniques

Le générateur doit utiliser, lorsque c'est nécessaire, des patrons de conception afin de garantir des performances optimales et une facilité de maintenance.

7.3. Interfaces

7.3.1. Interfaces personne-machine

Aucune interface personne-machine

7.3.2. Interfaces avec des matériels

Aucune interface avec un matériel hormis un ordinateur.

7.3.3. Interfaces avec d'autres logiciels

Suite à l'étude de faisabilité, une interface avec Eclipse CDT est envisagée pour récupérer les informations du parsing du code source C++.

7.3.4. Interfaces avec des fichiers ou des bases de données

Le générateur utilise les fichiers sources C++ à analyser pour créer le modèle PADL. Il fournit un modèle PADL, i.e., un objet conforme à l'interface ICodeLevelModel.

7.4. Exigences concernant la conception et la réalisation

7.4.1. Exigences d'adaptation vis-à-vis du système

Le langage de programmation utilisé est le Java. Eclipse est nécessaire au moins pour la phase de parsing du code source C++.

7.4.2. Exigences de programmation

Le guide style du laboratoire doit être utilisé et le code doit être rigoureusement commenté selon afin de permettre une maintenance aisée.

7.4.3. Exigences envers les outils de développement

Afin d'aider au développement du générateur et de se conformer aux fonctionnements du laboratoire, Eclipse doit être utilisé ainsi que SVN pour la gestion des versions.

7.4.4. Exigences particulières de sécurité

Aucune exigence particulière.

7.4.5. Traçabilité des exigences

Étant donné que je suis tout seul à développer le projet, la traçabilité des exigences ne comportera que sa composante temporelle.

7.5. Préparation de la livraison

Afin de permettre un déploiement facilité et sécurisé, le générateur devra être installable via une procédure claire. Suite aux choix effectuer dans l'étude de faisabilité, l'environnement cible doit posséder une machine virtuelle Java et une installation d'Eclipse avec le plugin CDT de la même version que celle utilisée lors du développement.

8. Conception

La conception de la solution est intrinsèquement liée et influencée par l'architecture de l'outil de récupération des informations nécessaire à l'instanciation du modèle PADL. La conception est également liée au modèle PADL. Nous allons voir quels ajouts ont été apportés au modèle PADL afin que celui-ci puisse représenter un programme C++ de façon plus complète. Nous verrons ensuite quelle architecture a été mise en place pour réaliser la solution ainsi que le diagramme de classe de la solution.

8.1. Choix d'architecture

Une fois ces modifications apportées, il faut récupérer les informations qui nous intéressent avec Eclipse CDT pour pouvoir instancier le modèle PADL représentant le programme. Pour ce faire j'ai opté pour l'architecture plugin d'Eclipse qui offre une grande souplesse, assure une inclusion des dépendances efficaces et permet de s'interconnecter facilement avec les projets du laboratoire puisque chaque projet du laboratoire est un plugin Eclipse. De plus, il n'est pas possible de découpler

le parseur C++ du plugin CDT de la plateforme d'Eclipse car ce parseur est très couplé à la notion de projet d'Eclipse. Le parseur doit être lancé sur les sources d'un projet Eclipse de type C++. Or, il n'est pas possible de créer ou de récupérer un projet Eclipse de type C++ sans lancer la plateforme Eclipse car un workspace doit être chargé (entre autres dépendances). J'ai passé de temps sur cet aspect du PFE afin d'être sûr que la solution envisagée ne pouvait être simplifiée.

Parmi la liste des extensions réalisables, j'ai choisi de faire un plugin qui n'utilise pas de système graphique car nous n'en avions pas besoin. J'ai donc opté pour une extension de type Application. Cette extension a l'avantage de lancer une nouvelle application Eclipse paramétrée selon nos besoins. Ce qui permet de lancer le traitement directement au démarrage de cette nouvelle instance d'Eclipse. Cela permet également de lancer Eclipse en mode Headless ce qui signifie que Eclipse ne charge pas l'interface graphique. Cela permet de gagner énormément de temps et de mémoire au lancement du générateur et de faciliter les tests ainsi que le déploiement.

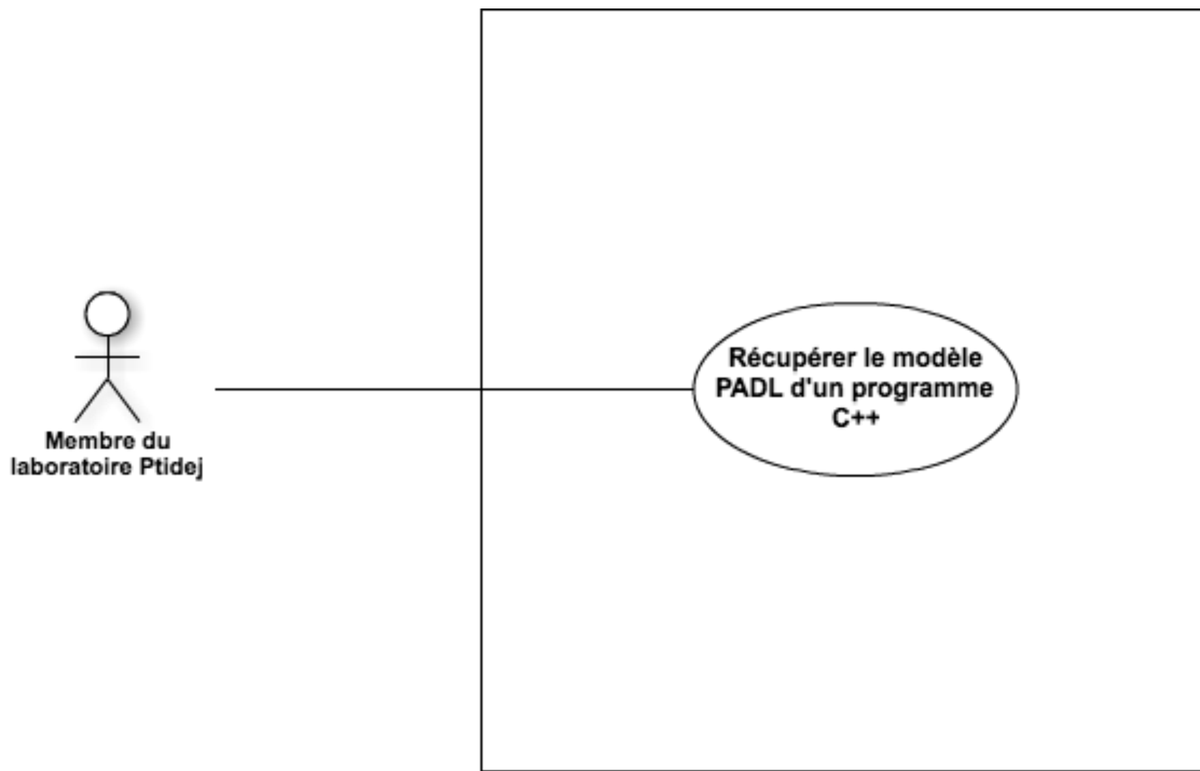
8.2. Modèle des tâches de l'utilisateur

8.2.1. Description des types d'utilisateurs

Les utilisateurs sont essentiellement les membres du laboratoire qui travaillent sur des logiciels développés en C++. Il est également possible que d'autres utilisateurs utilisent le générateur par le biais d'une plateforme intégrant cet outil. Ces utilisateurs ne sont pas pris en compte dans cette conception.

8.3. Cas d'utilisation

8.3.1. Cas d'utilisation générale :



Comme on peut le remarquer, le cas d'utilisation est simple, le générateur ne fait qu'une seule action. Elle crée un modèle PADL à partir de fichiers sources C++.

8.4. Implantation des données

8.4.1. MAJ

Les mises à jour se feront par le biais de l'utilitaire d'exportation d'Eclipse. Il suffit d'exporter le plugin et de remplacer l'ancien plugin dans le répertoire Eclipse de la machine cible.

8.5. Décomposition modulaire et hiérarchique

Le générateur peut être décomposée en 3 grands modules :

- ajout de fonctionnalité à PADL

- développement du générateur
- mise en place de la stratégie de déploiement

8.6. Modifications apportées à PADL

Afin de pouvoir représenter le C++, il a fallu ajouter certaines classes au métamodèle PADL. En effet, le C++ est beaucoup plus souple que le langage Java et permet une plus grande variété de représentation des données.

Les classes qui ont été ajoutées héritent des classes de base de PADL. PADL utilise un système de patron de conception visiteur pour parcourir un modèle. Ce mécanisme est mis en place pour permettre à n'importe quelle application de spécifier sa façon de parcourir un modèle et surtout d'appliquer un traitement à celui-ci sans surcharger le modèle de cette implémentation. Il a donc fallu créer un visiteur particulier pour prendre en compte les classes ajoutées.

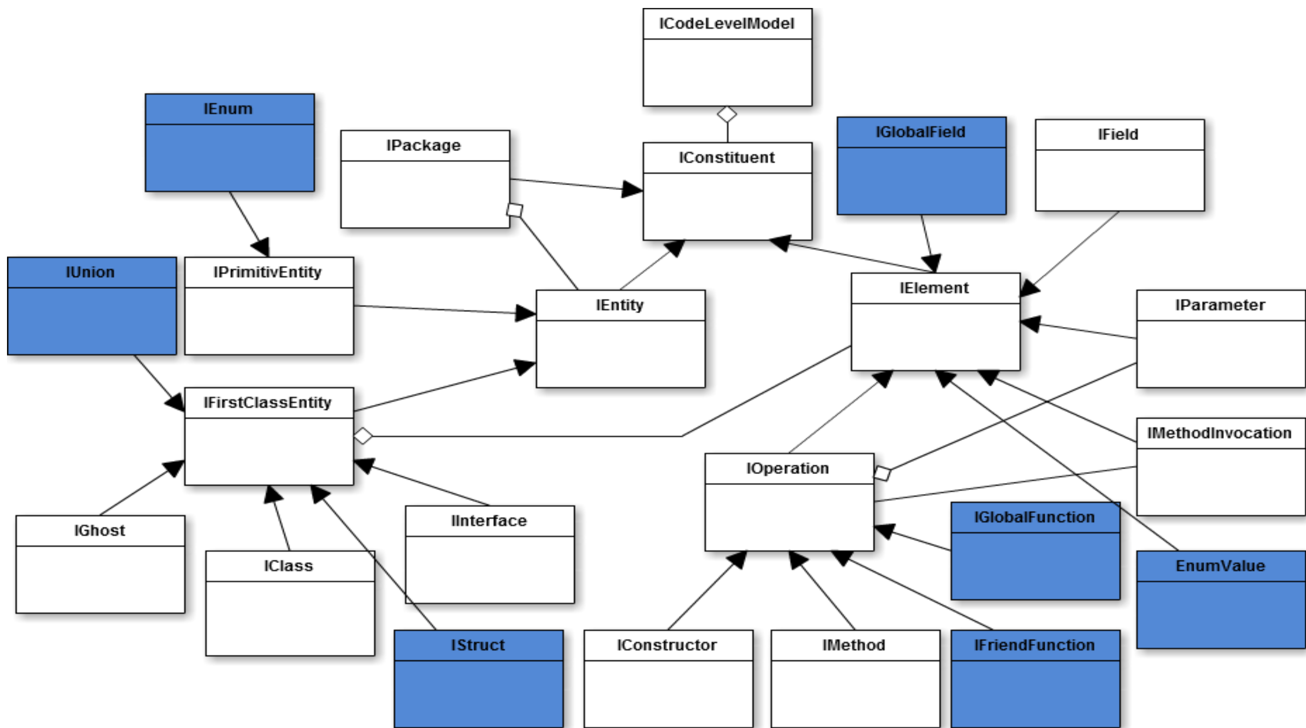
Le visiteur ajouté hérite du visiteur utilisé à l'origine et ne fais que spécifier le comportement des classes ajoutées. Ce comportement est simplement le même que la classe mère. Il suffit d'appeler la méthode du visiteur sur les objets ajoutés en effectuant un cast de ces objets en leur type mère. Cela garanti que les objets se comporteront comme leur type mère.

De plus, ce nouveau visiteur peut servir de visiteur de base pour tous les nouveaux traitements à effectuer sur un modèle PADL C++. Il suffit d'hériter du visiteur créé et de spécifier un traitement à effectuer pour les classes voulues sans oublier d'appeler les méthodes correspondantes dans la classe mère (avant ou après le traitement en fonction des besoins).

Cela procure deux avantages. Le premier est que l'ajout de sémantique dans le modèle est transparent pour les applications qui ne sont pas capable de traiter cet ajout. Le deuxième est que cela permet un raffinement des outils si un jour il est décidé de prendre en compte ces informations.

Les ajouts apportés au modèle PADL permettent de garantir le fonctionnement précédent tout en permettant une évolution des informations traitées. Cette flexibilité de l'architecture permet une évolution simple des anciens traitements afin d'être plus performant avec les modèles pour le langage C++

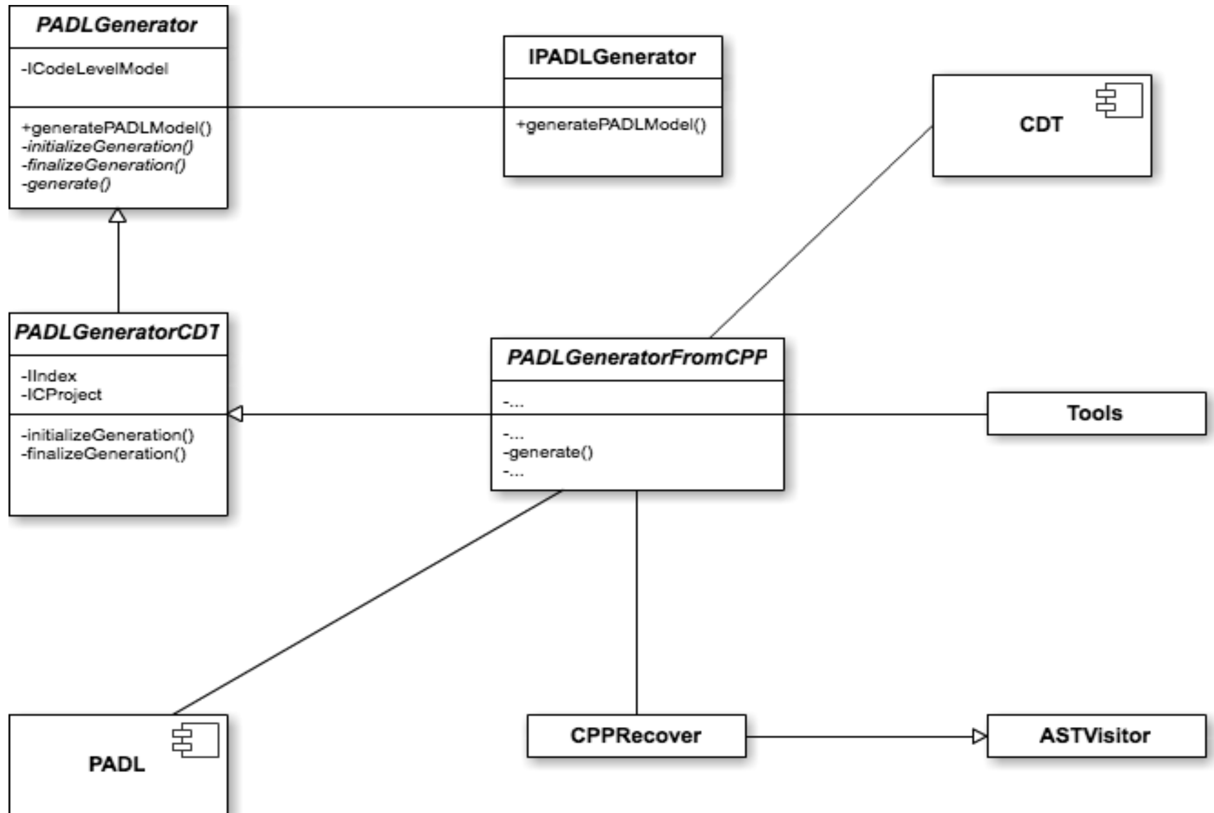
Diagramme de classes de PADL modifié :



Les ajouts au modèle sont en bleu. Ont été ajoutés la notion d'énumération comme étant un type primitif, les notions d'union et de structure comme étant des entités conteneurs d'éléments, la notion de variable globale comme étant un élément, les notions de fonction globales et de fonction amie comme étant des opérations, la notion de valeur d'énumération comme étant un élément.

Afin de pouvoir instancier chacun des objets ajoutés aux modèles tout en respectant les choix architecturaux de PADL, une factory spéciale pour les modèles C++ a été créée. Cette nouvelle factory hérite de l'ancienne et spécifie simplement comment créer les nouveaux objets ajoutés au métamodèle.

8.7. Construction du modèle structurel de l'application



Toutes les classes hormis les composants PADL et CDT ainsi que la classe ASTVisitor ont été créé pour la réalisation de ce PFE. Un signe plus devant une méthode signifie que cette méthode est publique. Un signe moins devant une méthode ou un attribut signifie une portée protégée (transmise par héritage). Les classes ou les méthodes qui ont leur nom en italique sont abstraites. IPADLGenerator est une interface qui décrit le protocole pour créer un modèle PADL de manière générale. Pour créer un modèle, il est obligatoire d'utilise un objet de type IPADLGenerator.

La classe abstraite PADLGenerator spécifie le comportement de la méthode generatePADLModel qui consiste en l'initialisation de la génération, la génération du modèle PADL et la finalisation de la génération. Elle permet d'obtenir le modèle PADL final.

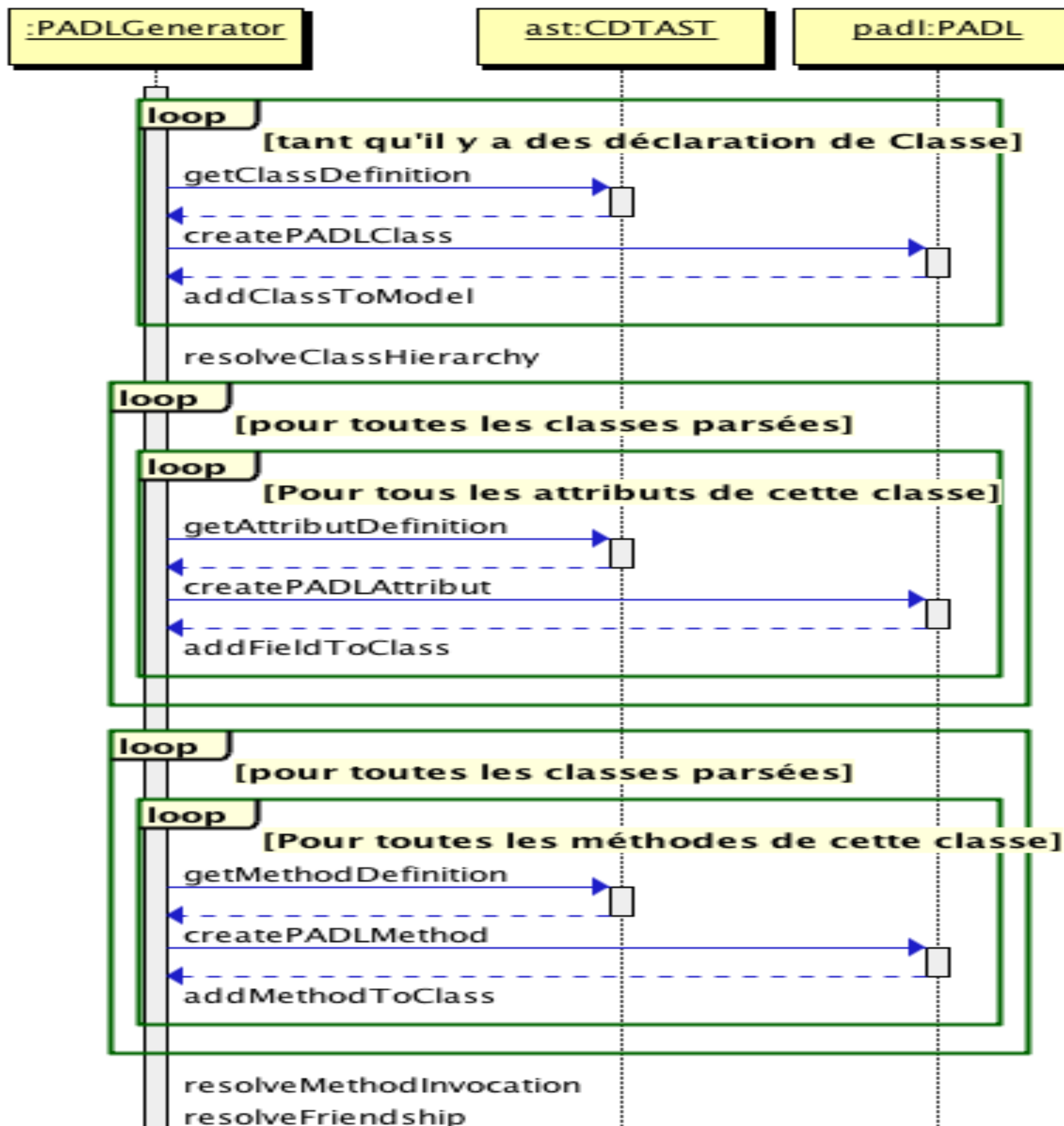
La classe abstraite PADLGeneratorCDT spécifie l'initialisation et la finalisation pour une génération à partir du plugin CDT. L'initialisation s'occupe de rafraichir le contenu du projet cible, de réserver l'accès à l'index de l'AST et de rafraichir l'index de l'AST. La finalisation s'occupe de rendre l'accès à l'index possible pour les autres processus.

La classe concrète PADLGeneratorFromCpp spécifie le comportement de la fonction de génération dont l'algorithme est décrit plus bas. Cette classe utilise une classe statique nommée Tools permettant de faire certaine opération sur des objets de l'AST afin de récupérer des informations. Elle utilise également un visiteur pour l'AST afin de récupérer les déclarations de classe, unions et structures. Elle utilise également directement l'API de CDT pour pouvoir parcourir les autres déclarations et connaître tous les types nécessaires à la récupération des informations pour créer un modèle PADL. Enfin, PADL est utilisé pour créer le modèle PADL du logiciel C++ cible.

8.8. Modélisation du comportement de l'application

8.8.1. Modèle dynamique de l'application

Voici le diagramme de séquence de création du modèle PADL pour un logiciel C++. Ce diagramme est simplifié et ne rentre pas dans les détails. Il permet de comprendre l'algorithme mis en place pour construire le modèle PADL



Le parsing des méthodes d'une classe se fait dans une autre boucle que dans celle du parsing des attributs. Ce choix a été fait pour faciliter le débogage et les tests. Il serait tout à fait possible, une fois les méthodes rigoureusement testées, d'incorporer les deux traitements pour chaque classe dans la même boucle.

9. Développement

Le développement de ce projet s'est réalisé de manière incrémentale. C'est à dire que chaque fonctionnalités du projet ont été incluses petit à petit afin d'être sûr de ne rien casser. À la fin du développement d'une fonctionnalité, des tests de non régression étaient créés. La façon dont ces tests ont été pensés est expliquée dans la partie Test de ce document.

9.1. Gestion d'un projet C++ dans Eclipse

Afin de faciliter le développement, j'ai commencé par tout ce qui faisait référence au projet C++ dans Eclipse. Le générateur instancie un Eclipse qui possède un projet C++ par défaut dans lequel il faut mettre les sources du programme dont l'on souhaite obtenir un modèle PADL. Au lancement du générateur, le programme rafraichit ce projet C++ afin de pouvoir traiter le programme voulu car un projet C++ Eclipse garde en mémoire des références vers les fichiers sources qui étaient précédemment présents à la précédente fermeture de Eclipse.

9.2. Algorithme de création du modèle PADL

Cette partie explique le diagramme de séquence système de la partie conception.

Le programme construit ensuite l'AST de chaque fichier source et l'index correspondant à chaque déclaration. Le programme parcours ensuite pour chaque Translation Unit, les définitions de classes, de structures, d'unions, d'énumérations, de fonctions globales et de variables globales.

Une fois cela effectué, le générateur récupère pour chaque classe les attributs, les méthodes, les constructeurs et les destructeurs. Il faut faire cela après les déclarations de classes car si une classe contient un attribut ou champ de méthode qui utilise une classe que nous n'avons pas encore parsée, cela devient difficile d'obtenir les informations car il faut maintenir un cache des références.

Ensuite nous renseignons les héritages et les relations d'amitiés aux niveaux des classes et aux niveaux des fonctions.

Une fois cela effectué, nous pouvons renseigner les invocations de méthodes dans chaque méthode. Pour faire ceci, nous maintenons un dictionnaire des méthodes PADL avec leur équivalence dans l'AST. Pour chaque méthode, nous récupérons les appels de méthodes et nous vérifions si nous ne possédons pas cette méthode dans PADL. Si c'est le cas, nous ajoutons cette invocation de méthode dans la méthode courante.

Enfin, nous renseignons toutes les relations d'amitié. Pour ce faire, nous récupérons les relations d'amitié dans l'AST pour chaque classe puis nous recherchons l'équivalent dans le modèle PADL. Nous ajoutons ce qui a été trouvé à la classe cible comme étant une classe ou une méthode amie en fonction du type

10. Test

Cette partie présente la stratégie de test mise en place pour permettre de valider l'avancement et l'état du projet.

10.1. Tests de non régression

Afin de tester le générateur, j'ai réalisé des tests de non-régression au niveau des fonctionnalités permettant de m'assurer que lors d'un ajout de fonctionnalité, cet ajout n'avait pas cassé le fonctionnement validé lors de l'ajout des fonctionnalités précédentes. Ces tests de non-régressions consistaient en la création d'une classe C++ et du modèle PADL correspondant (créé à la main) pour comparer ensuite le résultat de la génération à partir des sources.

10.2. Tests de recouvrement

Pour tester les fonctionnalités de la partie du générateur permettant de récupérer des informations de l'AST ou de l'index, un test permettant de s'assurer que la fonctionnalité était bien réalisée était créé. Des tests de recouvrement (boite blanche) ont également été créés afin de s'assurer que certaines parties du code étaient bien atteintes lorsqu'elles le devaient.

10.3. Tests fonctionnels

Pour tester le modèle PADL créé, un projet C++ contenant un programme pour chaque patron de conception du GoF a été créé. Une fois le modèle PADL généré, celui-ci est passé aux détecteurs de patron de conception afin de s'assurer que le programme fonctionne correctement. Cela m'a beaucoup aidé pour détecter les fonctionnalités manquantes. Cette partie n'était pas facile. Il aurait été plus simple de générer le modèle PADL puis de générer à partir de celui-ci le code source et comparer les deux fichiers. Seulement PADL ne permet pas de créer du code source « représentatif » à partir d'un modèle PADL.

11. Déploiement

Pour faciliter le déploiement, nous avons créé une petite procédure. Nous exportions le plugin application en incluant toutes ses dépendances (ce qui a mon étonnement n'étais pas si lourd que ça : 500ko environs). Ensuite, il suffit de placer ce plugin dans le répertoire plugin de l'installation Eclipse cible. Ensuite, il faut créer un projet C++ vide avec comme nom « C++ » dans le workspace par défaut de l'installation d'Eclipse cible. Ensuite, il suffit de lancer une commande Java qui lance le lanceur de Eclipse avec comme application de base le plugin développé. Si le plugin se trouve dans le

bon répertoire (plugin de Eclipse) et que Eclipse possède une installation de CDT, le programme est alors déployé.

Cette étape n'a pas été réalisée concrètement. Elle a seulement été créée et testée sur la machine de développement.

12. Etat du projet, suites possibles

Le générateur est stable et utilisable. Il permet d'avoir un modèle PADL permettant de retrouver la plupart des patrons de conception contenu dans un programme source C++. Il faut maintenant adapter un peu plus PADL au langage C++ afin de permettre à PADL de tirer des informations des relations d'amitié, de l'utilisation de fonctions globales et de variables globales.

Yann Gaël GUÉHÉNEUC m'a proposé de faire un doctorat dans son laboratoire afin de travailler sur ce projet. Il m'a aussi proposé de faire un parseur pour le langage C# ou pour toutes application développé avec le cadriciel C# (VB.NET et autres).

13. Difficultés rencontrées

Enormément de difficultés ont été rencontrées pour comprendre comment fonctionne l'API de Eclipse CDT car il n'y aucune documentation claire et précise expliquant comment utiliser cette API. L'API a subi d'énormes évolutions, ce qui fait qu'il est possible de trouver des tutoriels sur l'Internet mais il n'y en a aucun qui est viable pour la dernière version de CDT.

Beaucoup de difficultés ont été rencontrées avec PADL, il a fallu modifier son code source à plusieurs reprises afin de rendre son architecture plus souple et modulaire.

CONCLUSION

Ce projet de fin d'étude a été pour moi une expérience très intéressante. Il m'a permis d'allier plusieurs aspirations parmi lesquelles travailler en milieux anglophone et travailler dans le milieu de la recherche.

Le projet est actuellement réalisé, fonctionnel. Il permet de créer un modèle PADL à partir de code source C++ en utilisant l'AST fourni par la plateforme Eclipse CDT. De plus, les fonctionnalités ajoutées à PADL pour représenter correctement le C++ permettent une plus grande souplesse du métamodèle et une évolutivité des outils du laboratoire utilisant un modèle PADL.

Le fait d'avoir réussi à mener à bien un projet qui n'avait pas abouti autant de fois auparavant est pour moi quelque chose dont je suis fier, gage de savoir-faire et me permet par la même occasion de représenter correctement l'INSA de Lyon, mon école, à l'étranger.

Grâce à ce projet, J'ai énormément appris sur les patrons de conception ce qui m'a permis de perfectionner mes connaissances en architecture logiciel.

Ce projet permet actuellement au laboratoire de pouvoir élargir ses horizons de recherche en accédant aux nombreux logiciels réalisés en C++.

ANNEXES

1. Références Bibliographiques

CDT, E. (2012, Mai 20). *Eclipse CDT documentation*. Consulté le Mai 20, 2012, sur Eclipse CDT documentation: http://www.eclipse.org/cdt/documentation.php

Erich Gamma, R. H. *Design Pattern.* Boston: Vuibert.

GUEHENEUC, Y.-G. (2012, mai 20). *Ptidej*. Consulté le mai 20, 2012, sur Web of the Ptidej Team: http://www.ptidej.net/