Nous tenons à remercier M. Fabrice BOUQUET, Maître de conférences habilité à l'université de Franche Comté, notre professeur de Compilation et qui a parfaitement tenu le rôle d'un client très patient et compréhensif.

Nous souhaiterions remercier également M. Stéphane DEBRICON qui nous a initiés aux joies de la méthode XP.

Pour créer un programme exécutable, on a recours à un compilateur qui va, à partir du langage dans lequel on a écrit le programme, analyser la structure du code et former un autre code interprétable par la machine.

C'est ce genre d'outil que nous avons développé en groupe dans le cadre d'une réalisation de Master 1re année. Ce projet était proposé par M. Bouquet qui s'est donné le rôle de client.

Par conséquent, notre application est un outil complémentaire à l'enseignement de Compilation, en fournissant une représentation concrète de notions manipulées par le cours sur les compilateurs de langages abstraits. Notre application peut être vue également comme un outil d'expérimentation pour les étudiants des années inferieurs pour apprendre à manipuler des concepts de grammaires abstraites.

Un état de l'art, composer d'un sujet détaillé, d'un support de cours et d'un grammaire abstraite, nous a permis de définir avec le client les objectifs du projet, en particulier de choisir les structures de données à implémenter et sous quelle forme les représenter, ainsi que les structures de contrôle du langage dans le compilateur.

Nous avons d'abord procéder à différents choix techniques concernant les outils et le langage de développement que nous allions utiliser.

Notre choix c'est porter sur l'écriture d'un projet Java à des fins de portabilité, et sur la création d'un IHM basée sur un RCP (*RICH CLIENT PLATFORM*) Eclipse.

Nous avons également décidé d'utiliser le compilateur de langage JavaCC que nous avions déjà utilisé dans notre cursus universitaire, ainsi que l'utilisation du patron de conception Visiteur de Java.

Nous commencerons ce rapport par une première partie présentant le concept de compilateur, puis nous détaillerons les technologies résultantes de nos choix techniques dans une seconde partie. Dans un troisième temps nous étudierons en détails l'implémentation de notre compilateur. Pour finir, nous conclurons par la présentation du bilan tant humain que technique de ce projet.

1 – Présentation du concept de compilateur

Un compilateur est tout d'abord un programme. C'est un programme qui satisfait les quelques propriétés suivantes.

Premièrement, un compilateur reçoit un code source en entrée, et produit du code objet ou code exécutable en sortie. Il indique également les erreurs éventuelles survenues lors de la compilation.

La compilation en elle-même est la transformation, ou conversion du code source en code objet. Cela a une influence beaucoup plus large que la programmation. Le principe initial de la compilation n'est pas que de produire un programme exécutable à partir de son source, c'est également transformer un fichier écrit dans un format en un autre fichier utilisant un autre format mais ayant une sémantique interprétable.

Les étapes de fonctionnement d'un compilateur sont :

Analyse lexicale : Dans cette phase, les caractères isolés qui constituent le texte source sont regroupées pour former des *unités lexicales*, qui sont les mots du langage. L'analyse lexicale opère sous le contrôle de l'analyse syntaxique ; elle apparait comme une sorte de fonction de lecture qui fournit un mot (ou token) lors de chaque appel.

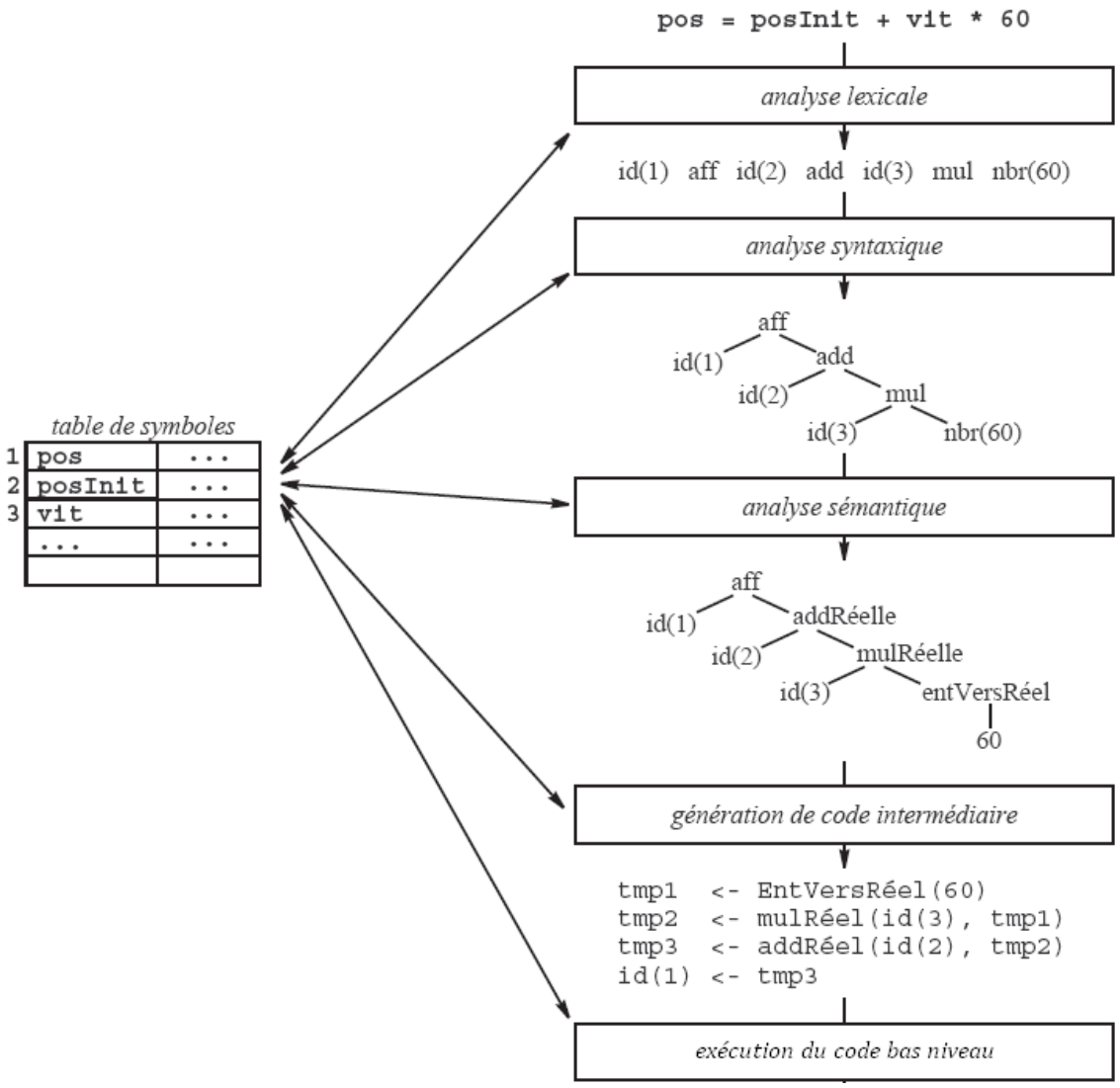
Analyse syntaxique : Alors que l'analyse lexicale reconnait les mots du langage, l'analyse syntaxique reconnait la constitution des phrases. Le rôle principal de cette phase est de dire si le texte source appartient au langage L(k) considéré, c'est-à-dire s'il est correct relativement à la grammaire de ce dernier.

Analyse sémantique (contrôle de type) : La structure du texte source étant correcte, il s'agit ici de vérifier certaines propriétés sémantiques, c'est-à-dire relatives à la signification de la phrase et de ses constituants en terme de structure de code :

- les identificateurs apparaissant dans les expressions ont-ils bien été déclarés ?
- les opérandes ont-ils les types requis par les opérateurs ?
- les opérandes sont-ils compatibles ? n'y a-t-il pas des conversions a insérer ?
- les arguments des appels de fonctions ont-ils le nombre et le type requis ?
- etc.

Génération de code intermédiaire : Après les phases d'analyse, certains compilateurs ne produisent pas directement le code attendu en sortie, mais une représentation intermédiaire, une sorte de code pour une machine abstraite. C'est le cas ici ou on peut choisir d'interpréter directement le langage sources, le Minijaja, ou générer le code intermédiaire, le Jajacode.

L'interprétation du programme : modification de la mémoire en fonction des instructions de bas niveau générées lors des étapes précédentes.



2 - Choix techniques

2.1. Eclipse

2.1.1. Introduction

Pour reprendre les mots de ses créateurs, Eclipse est « une sorte de plateforme universelle de développement d'outils, un environnement de développement ouvert pour tout et rien de particulier". Eclipse ne possède en effet presque aucune fonctionnalité personnelle. Sa valeur se situe plutôt dans la philosophie qu'elle encourage, à savoir un développement rapide de fonctionnalités intégrées à l'environnement sous forme de plug-ins.

Eclipse est un environnement open source développé autour d'une architecture permettant la découverte dynamique de plug-ins. La plateforme se contente de fournir à l'utilisateur un modèle de navigation standard et de s'occuper de la logistique de l'environnement de base. Chaque développeur de plug-in peut alors se concentrer sur les tâches à accomplir par le plug-in. Il n'y a aucune limite aux domaines d'application de ces tâches, si ce n'est l'imagination du développeur (et ses compétences bien sur).

2.1.2. Architecture

L'architecture générale d'Eclipse est décrite par la figure 3.2[1] . La plateforme est divisée en sous-systèmes. Chaque sous-système est lui-même implémenté sous forme de plug-ins. Ces sous-systèmes forment la couche supérieure de l'architecture. La couche inférieure est constituée du « Platform runtime ». C'est lui qui définit le modèle de plug-ins et des points d'extension. C'est lui qui est chargé du référencement des plug-ins et du maintien des informations relatives à ceux-ci dans un registre de la plateforme.

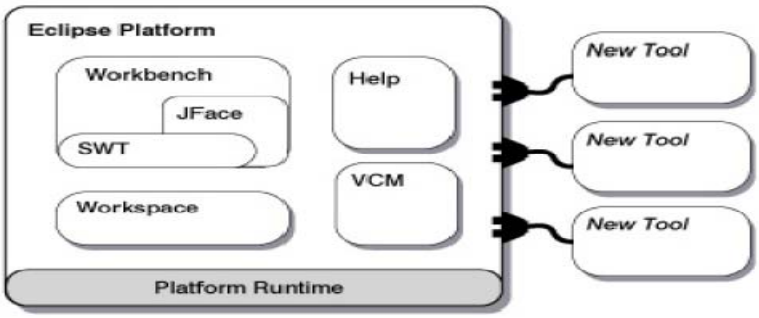


Fig. 3.2 – Architecture générale d'Eclipse

[1] extraite de http://www-106.ibm.com/developerworks/opensource/library/os-plat/

Voyons maintenant un à un chaque module de la couche supérieure de cette architecture :

- Workspace : c'est lui qui définit les API permettant de créer et de gérer les ressources (Projets, fichiers, dossiers) qui sont produites ou utilisées par les plug-ins et stockées dans le système de fichiers.
- Workbench : implémente l'interface utilisateur de la plateforme qui permettra de naviguer parmi les ressources et d'utiliser les autres outils. Il définit les points d'extension pour l'ajout de composants graphiques (comme les menus, vues, etc.). Il inclut des outils de conception d'interface utilisateurs comme SWT et JFace (cf. section 2.2.1.).
- Help : définit les points d'extension permettant aux plug-ins de créer leurs modules d'aide et de documentation.
- VCM : définit les modèles de programmation en équipe et de gestion de version des ressources.

Cette architecture purement modulaire (même le Platform runtime est composé de plug-ins) ne présente malheureusement pas que des qualités. Elle a aussi ses défauts.

L'un des plus importants problèmes de cette architecture est la taille du produit. Pour les premières versions d'Eclipse, les développeurs de l'environnement avaient basé leurs décisions sur l'estimation qu'un produit assez grand comporterait quelques centaines de plug-ins. De nos jours, certains produits de classe professionnelle développés sur Eclipse comprennent plus de mille plug-ins.

Les développeurs d'Eclipse ont dû revoir leurs prévisions et réadapter leur architecture.

La solution mise en place au sein d'Eclipse est assez simple. Aucun plug-in n'est chargé entièrement en mémoire au démarrage. Plutôt que de charger le plug-in, le platform runtime charge les « plug-in manifest ».

Ce fichier ne contient aucun code. C'est un fichier XML reprenant les fonctionnalités du plug-in, ses dépendances vis-à-vis d'autres plug-ins. L'utilisateur peut de cette façon accéder à la liste des fonctionnalités disponibles sans agrandir la zone de mémoire système allouée à Eclipse.

Dès que l'utilisateur décidera d'utiliser une fonctionnalité, Eclipse chargera en mémoire le code du plug-in correspondant. Ceci permet de restreindre la taille d'Eclipse dans la mémoire système tout en gardant un accès rapide à toutes les fonctionnalités proposées par les plug-ins.

2.2. JFACE

2.2.1. Présentation

JFace s'appuie sur la bibliothèque SWT pour fournir une API de développement plus évoluée et plus structurée.

Les principaux concepts proposés par JFace :

- une abstraction des composants natifs SWT
- une séparation de la partie modèle et de la vue (modèle MVC)
- des composants graphiques additionnels (Dialog, Preferences, …)
- une utilisation plus fine des ressources (Action, ImageDescriptor, …)

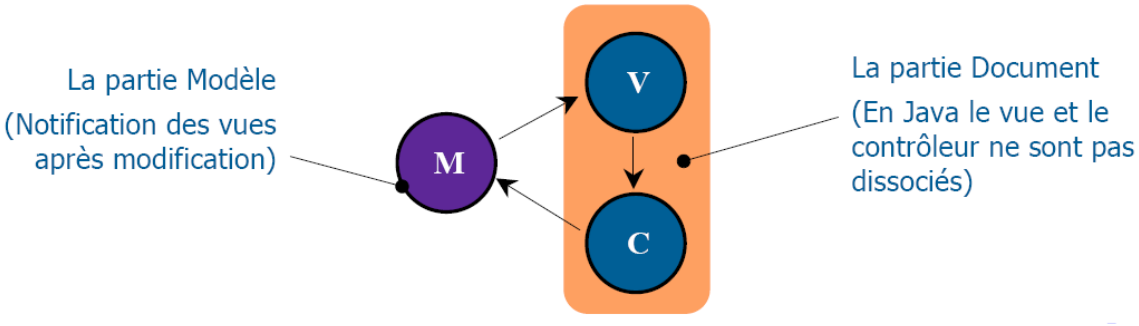
JFace a pour fonction de simplifier les développements en SWT, sans pour autant masquer totalement SWT. Combinée à SWT, JFace est utilisée pour le développement d'applications pour la plateforme Eclipse.

2.2.2. Utilisation

Typiquement en SWT, vous créez le composant, ajoutez des données et appelez des méthodes. De ce fait il devient difficile de mettre à jour proprement les données des composants.

Une approche MVC (Model, View, Control) est fournie par la surcouche JFace. Elle permet la séparation « stricte » entre le modèle de données et le modèle graphique. Cela à pour objectif de rendre plus facile l'ajout des écouteurs (listeners) sur le modèle de données pour notifier de ces changements d'états. Cela permet également de brancher plusieurs vues pour un même modèle et d'accéder aux autres modèles (sélection, édition, …) à partir de la vue.

On peu également «Customiser » le rendu des données en configurant des Interfaces Java de décorations des données.



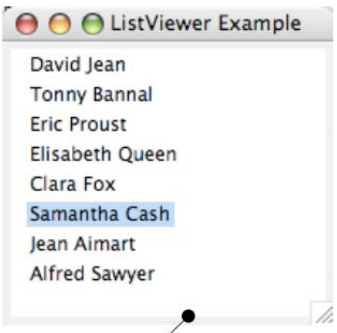
L'implémentation proposée par JFace de MVC se rapproche plus du Document/Vue ou Model/View (le contrôleur étant associé à la vue) comme on peut le voir sur le schéma ci-dessus.

2.2.3. Exemples de Widgets JFace

Le package org.eclipse.jfaces.viewers fournit un ensemble de classes pour l'encapsulation des composants SWT. Les composants de visualisation sont appelés des Viewers. Le nom des classes est composé du nom de l'encapsulation SWT suivi de « Viewer ».

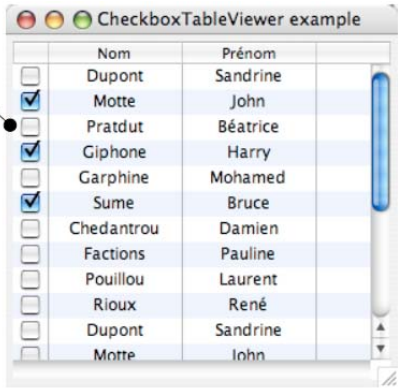
Les principaux composants sont :

- TreeViewer : un arbre
- TableViewer : un tableau
- ListViewer : une liste
- TableTreeViewer : un tableau avec un arbre sur la première colonne
- CheckboxTableViewer : un tableau avec des éléments à cocher
- CheckboxTreeViewer : un arbre avec des éléments à cocher
- ComboViewer : une boite à valeurs

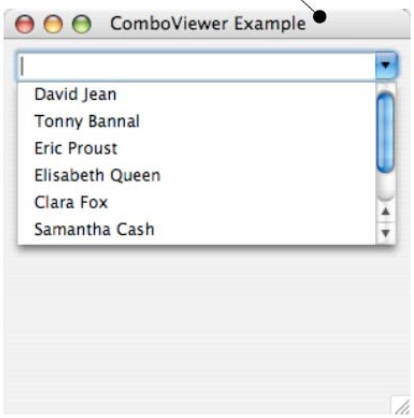


ListViewer

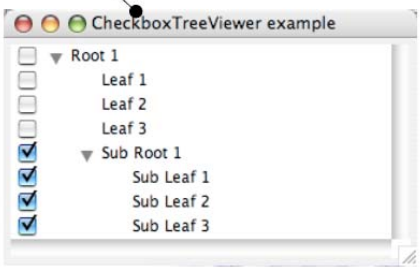
CheckboxTableViewer

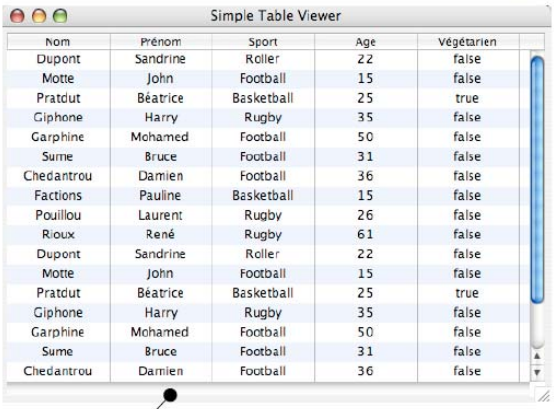


ComboViewer

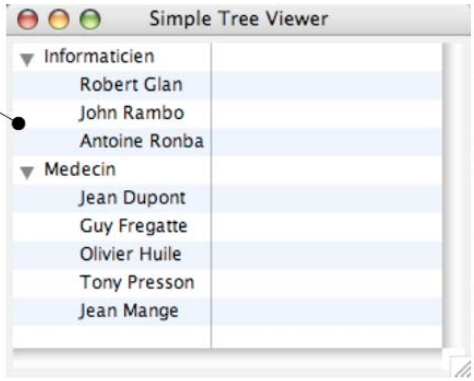


CheckboxTreeViewer

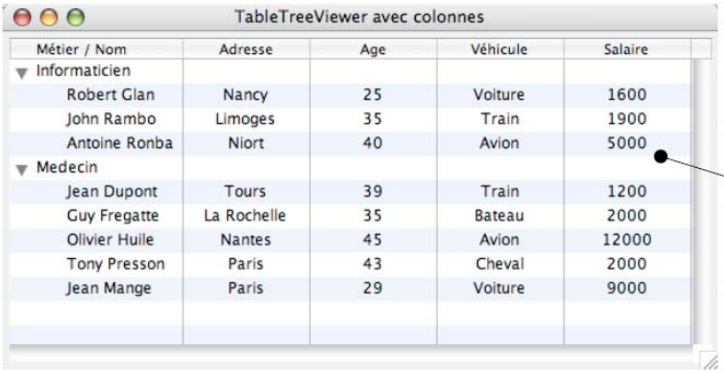




TreeViewer



TableViewer



TableTreeViewer (TreeViewer avec colonnes)

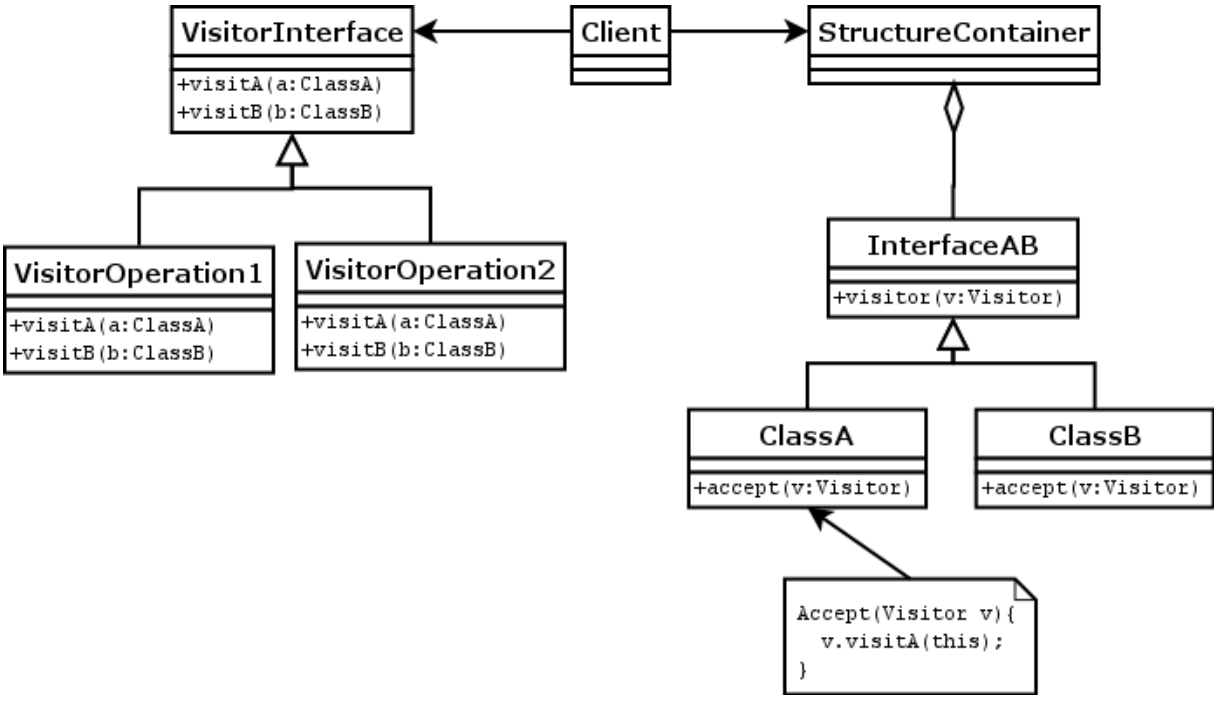
2.3. Le design Pattern Visiteur (motif de conception)

Le pattern visiteur est un concept Java associé au parcours de structure arborescentes. Nous l'avons utilisé pour les parcours et traitements des arbres de syntaxes abstraites du programme.

Dans le cadre de ce projet nous avons à utiliser des arbres pour différentes opérations : le contrôle de types, l'interprétation du MiniJaja, la compilation du code MiniJaja vers du JajaCode et l'interprétation du Jajacode. L'utilisation du patron de conception Visiteur permet de réaliser de manière claire et fonctionnelle les traitements associées a ces structures de données.

Un article de Wikipédia, l'encyclopédie libre, présenté ici, nous explique parfaitement son fonctionnement.

En programmation orientée objet, génie logiciel et accessoirement en MITIC, le motif de conception visiteur est une manière de séparer un algorithme de la structure d'un objet.



Ce modèle de conception permet à une classe externe d'accéder aux variables internes d'autres classes (allant à l'encontre des concepts de la POO). Ce modèle est utile lorsqu'on a un nombre raisonnable d'instances d'un petit nombre de classes et qu'on désire effectuer des opérations qui les impliquent tous (ou un bon nombre d'entre eux).

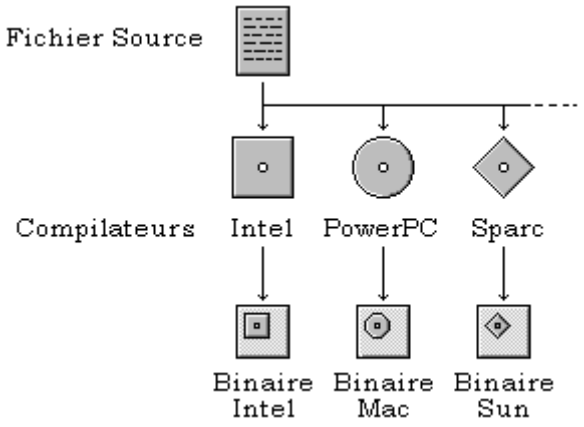
Si déporter des opérations contenues dans une classe vers une autre peut sembler mauvais au sens POO, il y a de bonnes raisons pour le faire. En effet, si ces opérations sont identiques pour chaque classe au lieu de dupliquer cette méthode il est préférable de mettre ces opérations dans un visiteur (centralisation de l'opération). Le visiteur utilisera ensuite les données internes de chaque objet pour effectuer l'opération demandée.

En pratique, le modèle de conception visiteur est réalisé de la façon suivante : chaque classe pouvant être « visitée » doit mettre à disposition une méthode publique « accepter » prenant comme argument un objet du type « visiteur ». La méthode « accepter » appellera la méthode « visite » de l'objet du type « visiteur » avec pour argument l'objet visité. De cette manière, un objet visiteur pourra connaître la référence de l'objet visité et appeler ses méthodes publiques pour obtenir les données nécessaires au traitement à effectuer (calcul, génération de rapport, etc.)

2.4. Le langage de programmation

Avec les langages évolués courants (C++, Pascal, etc.) nous avons pris l'habitude de coder sur une machine identique à celle qui exécutera nos applications; la raison est fort simple : à de rares exceptions près les compilateurs ne sont pas multiplateformes et le code généré est spécifique à la machine qui doit l'accueillir.

Ainsi si nous souhaitons distribuer un produit sur plusieurs systèmes, nous devons le compiler pour chacun de ces systèmes :



Nous devons alors utiliser soit *n* compilateurs différents sur *n* machines, soit un ou plusieurs compilateurs multiplateformes. Outre l'investissement en matériels et logiciels, la connaissance de plusieurs systèmes et de leurs outils de développement (formation aux différents compilateurs et débuggeurs) représente un surcoût non négligeable.

Cette vision du développement logiciel s'accordait assez bien aux besoins tant que les applications produites disposaient d'une interface homme - machine (HIM) restreinte et que les sources étaient aisément portables d'un système à un autre. Les programmes fortran recouvrent bien cette définition.

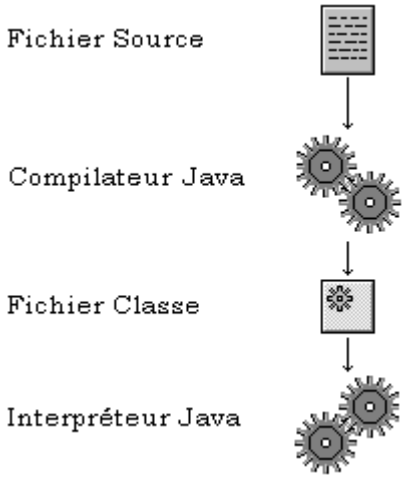
Aujourd'hui, la généralisation des interfaces graphiques et l'usage de langage plus évolués compliquent encore davantage le problème. Ainsi pour développer une application destinée à plusieurs systèmes, il nous faut non plus seulement connaître plusieurs compilateurs, mais également plusieurs systèmes d'exploitation avec ses différentes couches de librairies et d'interfaces; les API de ces interfaces étant toutes différentes.

Ainsi nos applications sont fortement dépendantes des ressources (y compris graphiques) du système hôte, dépendantes des API des interfaces utilisées, et le code produit ne peut s'exécuter que sur le système pour lequel il a été initialement produit.

La solution : JAVA

Tout d'abord, Java simplifie le processus de développement : quelle que soit la machine sur laquelle on code, le compilateur fournit le même code.

Ensuite, quel que soit le système utilisé cet unique code est directement opérationnel :



En effet, la compilation d'un source Java produit du pseudo-code Java qui sera exécuté par tout interpréteur Java sans aucune modification ou recompilation.

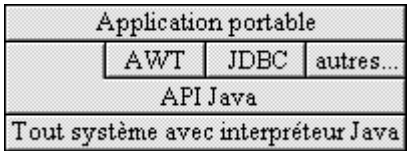
Cet "interpréteur" est couramment dénommé "machine virtuelle Java". De plus en plus, cette machine virtuelle utilise un compilateur JIT ("Just In Time") qui transforme, au moment de l'exécution, le pseudo-code en code natif afin d'obtenir la vitesse d'exécution maximale.

Le langage Java peut être utilisé pour créer des modules de code référencés au sein d'une page html et exécutés par un navigateur compatible Java (ou un simple interpréteur Java), on parle alors d'applets.

Ces modules de code ont rendu Java populaire car ils permettent à un créateur de site d'enrichir le contenu de son site de modules dynamiques et/ou interactifs qui tourneront à l'identique quelque soit la machine et le système utilisé par le visiteur de ce site.

Java permet également de créer des applications autonomes qui peuvent se substituer à des applications développés en langage compilé. Pour ces applications l'api Java apporte un ensemble très riche de classes répondant à de nombreux besoins et pouvant être étendue; cette unique api simplifie la création et le déploiement des applications, en effet cette application s'exécutera sur tout système en utilisant l'aspect visuel de ce système.

La construction d'une application Java répond au schéma :



Les applications ne sont plus dépendantes d'api propriétaires de leurs éditeurs (l'api Java est rendue publique par Sun); si le langage Java réussit à s'imposer, l'apprentissage de ce seul langage et d'une seule api permettra une certaine universalité aux applications débarrassées de leur tutelle propriétaire.

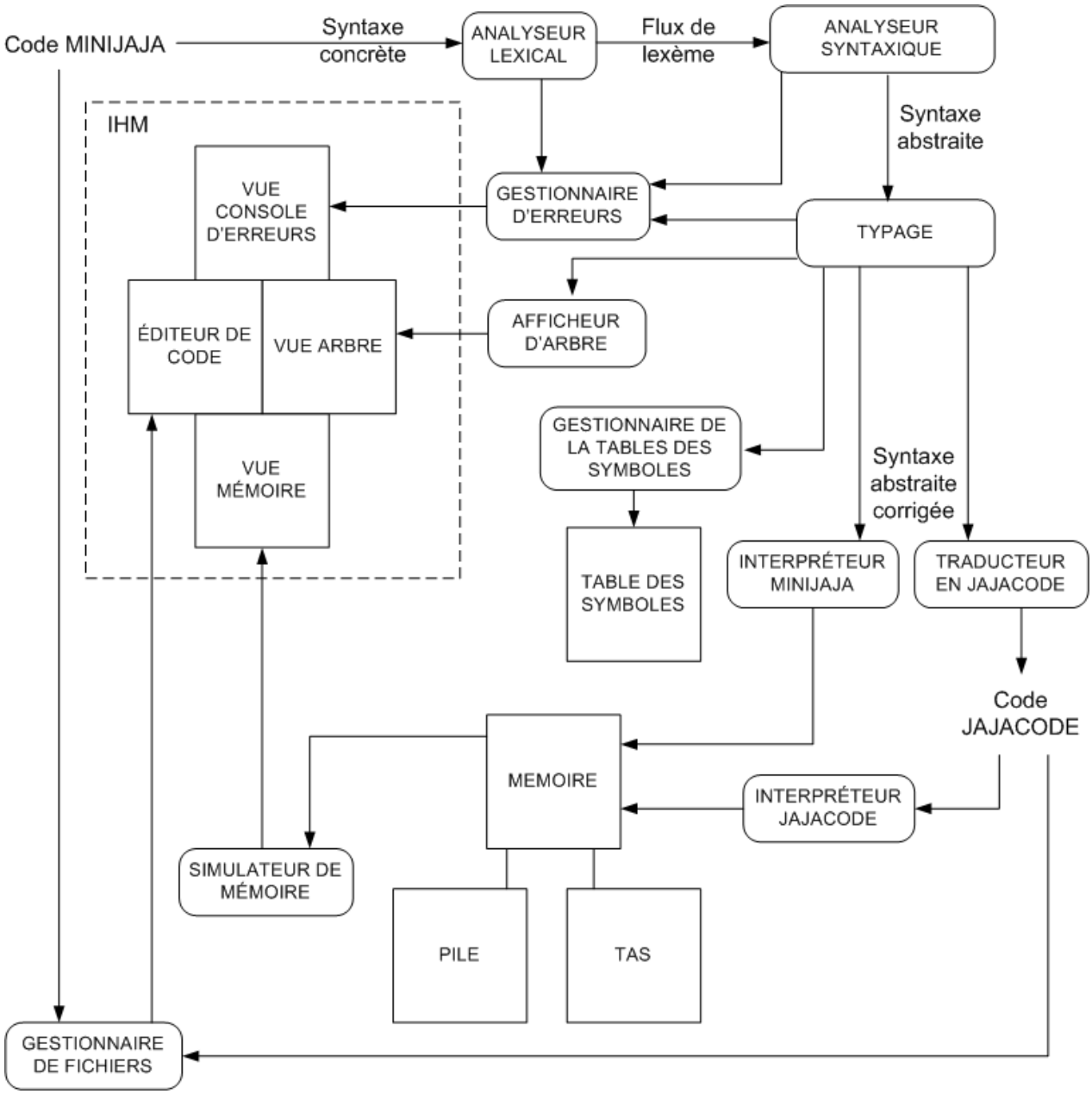
2.5. Java Compiler Compiler (JavaCC) - The Java Parser Generator

Java Compiler Compiler (JavaCC) est le plus utilisé des générateurs de parser pour Java. Un générateur de parseur est un outil qui lit les spécifications d'une grammaire et qui la convertit en program Java. En plus d'être un générateur de parseur, JavaCC fournit d'autres possibilités relatives à la génération de parser comme la construction d'arbre, le debugage, etc...

JavaCC prend comme entrée un fichier MaGrammaire.jj qui contient entre autre les descriptions des règles de la grammaire et produit un parser descendant (dans le fichier MaGrammaire.java). Une classe MaGrammaire est définie dans le fichier java. Elle implémente l'interface MaGrammaireConstants, définie dans MaGrammaireConstants.java et qui contient les définitions des mots clés de la grammaire.

3 – Détails de l'implémentation

3.1. Schéma général du compilateur Minijaja



3.2. La grammaire Minijaja sous forme BNF

```
/** Fichier jajacode.jjt
 * Auteurs : J.SCHOFFEN
 * Correction : L.RICHARD le 12/11/2007
 *
 * L'option MULTI permet de typer les
arbres par ASTNomArbre.
 * VISITOR=true permet l'utilisation des
visiteurs
 **/

options{
  MULTI=true;
  VISITOR=true;

VISITOR_EXCEPTION="minijaja.exception.MiniJ
ajaVisitorException";
}

/** --- Definition du parseur : --- */


PARSER_BEGIN(MiniJaja)
       package minijaja.jjtree;

       public class MiniJaja{
              public static void main
(String arg[]) throws ParseException {
                     MiniJaja mjj = new
MiniJaja(System.in) ;
              mjj.classe() ;
              }
       }
PARSER_END(MiniJaja)

/** ----------------- Regles lexicales ----
------------------- */

SKIP:{
  " "
  | "\t"
  | "\n"
  | "\r"
  | <"//"(~["\n", "\r"])*("\n"
  | "\r"
  | "\r\n")>
  |
<"/*"(~["*"])*"*"(~["/"](~["*"])*"*")*"/">
}
TOKEN:/* MINIJAJA RESERVED WORDS*/
{
    <BOOLEAN:"boolean">
  | <CLASS:"class">
  | <MAIN:"main">
  | <ELSE:"else">
  | <FALSE:"false">
  | <FINAL:"final">
  | <IF:"if">
  | <INT:"int">
  | <RETURN:"return">
  | <TRUE:"true">
  | <VOID:"void">
  | <WHILE:"while">
}

TOKEN:/* SEPARATORS */
{
    <LPAREN:"(">
  | <RPAREN:")">
  | <LBRACE:"{">
  | <RBRACE:"}">
  | <LBRACKET:"[">
  | <RBRACKET:"]">
  | <SEMICOLON:";">
  | <COMMA:",">
}
TOKEN:/* IDENTIFIERS */
{
       < IDENTIFIER: (["a"-"z","A"-"Z"])
(["a"-"z","A"-"Z","0"-"9","_"])* >
  | < NOMBRE: (["0"-"9"]) (["0"-"9"])* >
}
TOKEN:/* OPERATORS */
{
       <EQ:"==">
  | <ASSIGN:"=">
  | <BANG:"!">
  | <INCR:"++">
  | <PLUSASSIGN:"+=">
  | <PLUS:"+">
  | <MINUS:"-">
  | <STAR:"*">
  | <SLASH:"/">
  | <LT:"<">
  | <OR:"||">
  | <AND:"&&">
  | <GT:">">
}

/** -------------------- Grammaire  -------
------------------ */

/*
 * A ajouter dans ASTident
       private String ident;

       public void setValeur(String str){
              this.ident=str;
       }

       public String getValeur(){
              return this.ident;
       }
 * A ajouter dans ASTnbre
       private int nombre;

       public void setValeur(int nb){
              this.nombre=nb;
       }

       public int getValeur(){
              return this.nombre;
       }
 * A ajouter dans MiniJaja
       public JJTMiniJajaState getJJTree()
{
              return jjtree;
       }
 * A modifier dans JJTMiniJajaState
       public class JJTMiniJajaState {

       public Node rootNode() {
 */

void classe() #classe : {} {
       <CLASS> ident() <LBRACE> decls()
methmain() <RBRACE> <EOF>
}

void ident() #ident: {} {
       <IDENTIFIER> {
jjtThis.setValeur(token.image); }
}

void decls() #void : {} {
```

```
       decl() <SEMICOLON> decls() #decls(2)
  | {} #vnil
}

void decl() #void : {} {
       <FINAL> type() ident() vexp()
#cst(3)
  |    typemeth() ident() declII()
}

void declII() #void : {} {
       <LBRACKET> exp() <RBRACKET>
#tableau(3)
  | <LPAREN> entetes() <RPAREN> <LBRACE>
vars() instrs() <RBRACE> #methode(5)
  |  vexp() #var(3)
}

void vars() #void : {} {
    var() <SEMICOLON> vars() #vars(2)
  | {} #vnil
}

void var() #void : {} {
       <FINAL> type() ident() vexp()
#cst(3)
  | typemeth() ident() varII()
}

void varII() #void : {} {
    <LBRACKET> exp() <RBRACKET> #tableau(3)
  |  vexp() #var(3)
}

void vexp() #void : {} {
    <ASSIGN> exp()
  | {} #omega
}

void methmain() #main(2) : {} {
  <MAIN> <LBRACE> vars() instrs() <RBRACE>
}

void entetes() #void : {} {
    entete() entetes2() #entetes(2)
  | {} #enil
}

void entetes2() #void : {} {
       <COMMA> entete() entetes2()
#entetes(2)
  | {} #enil
}

void entete() #void : {} {
       type() ident() #entete(2)
}

void instrs() #void : {} {
       instr() <SEMICOLON> instrs()
#instrs(2)
  | {} #inil
}

void instr() #void : {} {
    <RETURN> exp() #retour
  | <WHILE> <LPAREN> exp() <RPAREN>
<LBRACE> instrs() <RBRACE> #tantque(2)
  | <IF> <LPAREN> exp() <RPAREN> <LBRACE>
instrs() <RBRACE> elseNode() #si(3)
  | ident() instrII()
}

void elseNode() #void : {} {
    <ELSE> <LBRACE> instrs() <RBRACE>
  | {} #inil
}

void instrII() #void : {} {
    identII() instrIII()
  | <LPAREN> listexp() <RPAREN> #appelI(2)
}

void instrIII() #void : {}{
    <ASSIGN> exp() #affectation(2)
  | <PLUSASSIGN> exp() #somme(2)
  | <INCR> #increment(1)
}

void identII() #void : {} {
       [ <LBRACKET> exp() <RBRACKET>
#tab(2) ]
}

void listexp() #void : {} {
    exp() listexpII() #listexp(2)
  | {} #exnil
}

void listexpII() #void : {} {
    <COMMA> exp() listexpII() #listexp(2)
  | {} #exnil
}

void exp() #void : {} {
    <BANG> exp1() #non
  | <MINUS> exp1() #neg
  | exp1() expII()
}

void expII() #void : {} {
  [
    <AND> exp1() expII() #et(2)
  | <OR> exp1() expII() #ou(2)
  ]
}

void exp1() #void : {} {
       exp2() exp1II()
}

void exp1II() #void : {} {
  [
    <EQ> exp2() #egal(2)
  | <LT> exp2() #inf(2)
  | <GT> exp2() #sup(2)
  ]
}

void exp2() #void : {} {
       terme() exp2II()
}

void exp2II() #void : {} {
  [
    <PLUS> exp2() #plus(2)
  | <MINUS> exp2() #moins(2)
  ]
}

void terme() #void : {} {
       fact() termeII()
}

void termeII() #void : {} {
  [
    <STAR> fact() termeII() #mult(2)
  | <SLASH> fact() termeII() #div(2)
  ]
}
```

```
void fact() #void : {} {

    <TRUE> #vrai
  | <FALSE> #faux
  | <LPAREN> exp() <RPAREN>
  | ident() factII()
  | nbre()
}
void factII() #void:{}{
       ( <LPAREN> listexp() <RPAREN> )
#appelE(2)
  | identII()
}

void nbre() #nbre : {int x;} {
       <NOMBRE> {
              try { x =
Integer.parseInt(token.image);
              } catch
(NumberFormatException ee) {
System.err.println("Error: " + token.image
+ " is not a number.");
                  x = 0;
              }
              jjtThis.setValeur(x);
       }
}

void typemeth() #void : {} {
       <VOID> #rien
  | type()
}

void type() #void:{}{
    <INT> #entier
  | <BOOLEAN> #booleen
}
```

3.2. La représentation de la mémoire

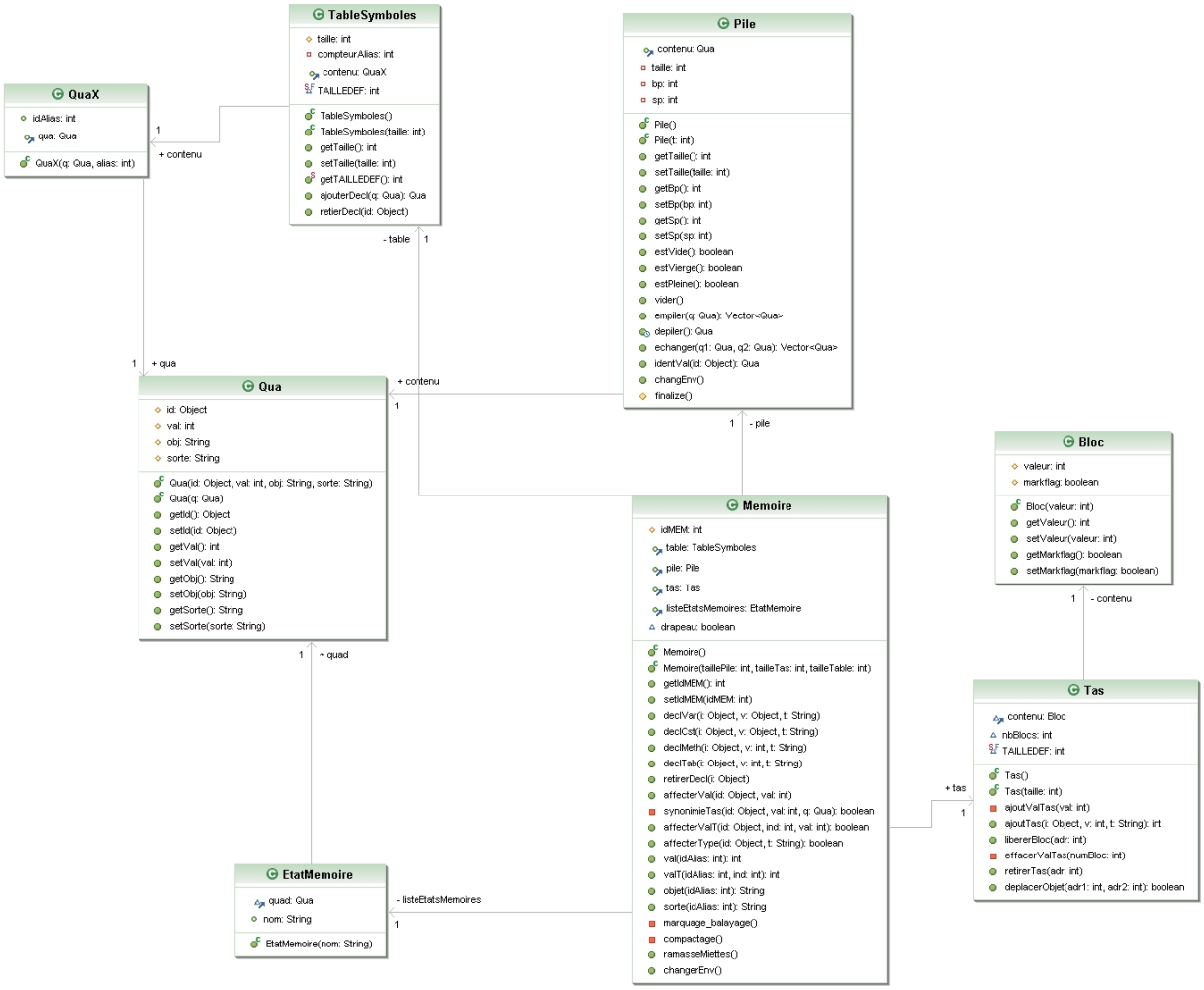
3.2.1. Définitions

Durant l'exécution d'un programme, les éléments manipulés par le programme sont stockés dans différents types de mémoire.

La pile d'exécution contient des éléments dont la durée de vie n'excède pas la durée d'un appel de fonction. La pile contient justement, en général, les différents éléments relatifs à un tel appel : les arguments, les variables locales, les adresses de retour, etc.

La mémoire dite dynamique va, quant à elle, être utilisée pour représenter toutes les données dont la création a été demandée par le programme durant son exécution, et dont la durée de vie ne peut pas être a priori bornée. La zone d'allocation mémoire dynamique est appelée le tas.

3.2.2. Représentation UML

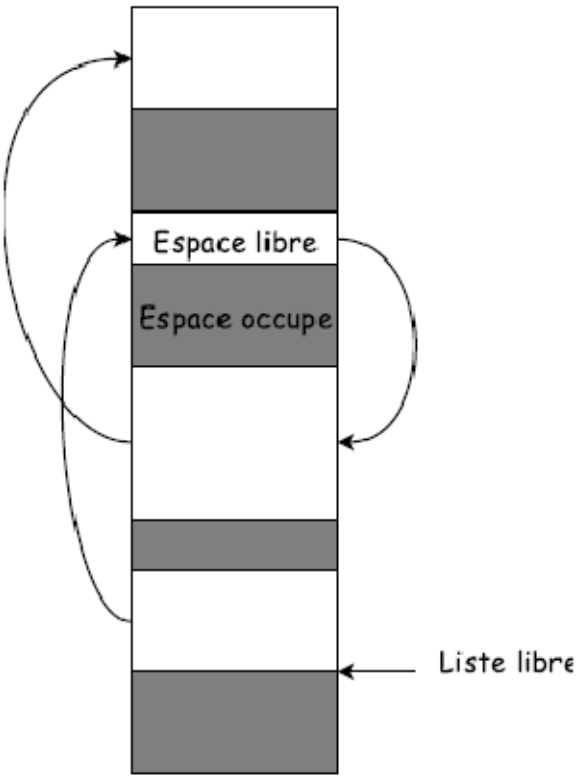


3.2.3. La gestion des espaces libres

La méthode dite « de la liste libre » consiste à chainer entre eux les blocs de mémoire libre, de sorte que chaque bloc contienne un pointeur vers le bloc suivant. La liste ainsi créée est appelée la liste libre. Quand un bloc mémoire est nécessaire, il est choisi dedans et extrait de la liste libre. A l'inverse, lorsqu'un bloc est recyclé, il est inséré dans la liste libre.

Le premier désavantage de cette représentation est le temps qui peut être nécessaire pour trouver un bloc de la bonne taille (ou bien un bloc de taille supérieure, qu'il faudra « casser » pour en extraire une partie de la taille souhaitée).

Un autre problème – probablement le plus important – est le fait qu'à force d'allouer et de recycler de la mémoire de la sorte, la mémoire occupée finit par être fragmentée, c'est-`a-dire que des blocs pointent sur d'autres blocs qui sont physiquement éloignés d'eux, ce qui a un coût à l'exécution, puisque les accès successifs à ces blocs peuvent entrainer des relectures de segments de mémoire physiquement distinct (même si ici le problème reste virtuel).



3.2.4. Le Garbage Collector

Lorsque le tas est plein et qu'il n'y a donc plus de mémoire disponibles, les GC de ce groupe vont parcourir la mémoire afin de déterminer l'ensemble des objets (blocs) effectivement utilisés, qu'on appelle aussi les objets vivants, par opposition aux objets devenus inutiles, dont on dira qu'ils sont morts.

Déterminer les objets vivants détermine aussi les objets morts, qui forment le complémentaire des premiers dans le tas.

On dira qu'un objet est vivant si :

- son adresse est la valeur d'une variable (un pointeur) qui est stockée de façon temporaire dans la pile;
- il y a un objet alloué dans le tas (tableau) qui est lui-même vivant et qui contient un pointeur vers cet objet.

Clairement, tous les objets vivants dans le tas peuvent être identifiés par un parcours du graphe mémoire, dont les points de départ sont les valeurs de quadruplets stockées dans la pile. On appelle ces points de départ des racines et tout objet qui n'est pas accessible depuis ces racines sera considéré comme mort et pourra être recyclé.

```
/**
 * Algorithme de Marquage-balayage :
 *     - A chaque bloc alloue  est associe un marqueur (ou drapeau, ou mark flag).
 *     - Si le bloc fait partie d'un objet refference (pointe) par la Pile, on le
 *       marque (vivant).
 *     - Enfin, on balaye le tas en dï¿½salouant les blocs non marque (les morts).
 * @author Richard Ludovic
 */

private void marquage_balayage(){

	int i,j, adr, decal;

	// inversion du drapeau :
	this.drapeau = ! this.drapeau;

	// --- Marquage :

	// parcours de la Pile (racine) :
	for(i=this.pile.getBp();i<this.pile.getSp();i++){

		// si on trouve une declaration de tableau :
		if(pile.contenu.get(i).obj == "tab"){

			// marquage des vivants dans le tas :
			adr = Integer.parseInt(pile.contenu.get(i).val.toString());
				decal = this.tas.contenu[(adr-1)].valeur;
				for(j=adr-1; j<=adr+decal; j++){
					this.tas.contenu[j].setMarkflag(this.drapeau);
				}
			}
		}
```

```
		// --- balayage :

		// on parcours le tas et on supprime les morts :
		for(i=0; i<this.tas.contenu.length; i++){
			if(this.tas.contenu[i].markflag != this.drapeau){
				this.tas.libererBloc(i);
			}
		}
	}
```

Une fois les éléments morts éliminés du tas, il faut déplacer les objets vivants vers le début du tas en les regroupant pour faciliter l'insertion de nouveaux objets. C'est le compactage.

```
/**
 * Algorithme de compactage :
 *   - deplacement (copie) des objets vers le debut du tas en remplissant les
 *     blocs vides
 *   - liberation des blocs recopies
 *   - au final tous les blocs vides sont a la fin du tas (plus de fragmentation).
 *   @author Richard Ludovic
 */
private void compactage(){

  int i, cptBlcFree;
  cptBlcFree = 0;

  for(i=0; i<this.tas.contenu.length; i++){//parcours du tas.

	if(this.tas.contenu[i] == null){

		cptBlcFree++; // compte les blocs vides avant un objet.

	}else{

	if(cptBlcFree != 0 ){// on décale :

	this.tas.contenu[(i - cptBlcFree)].valeur = this.tas.contenu[i].valeur;
	this.tas.libererBloc(i);
	cptBlcFree--;
	}
	}
   }
}

/**
 * Execute le garbage colector sur le tas.
 * Lorsque ramasse-miettes se dï¿½clenche:
 *     - phase de marquage: trouver les vivants
 *     - phase de balayage: recycler les morts
 *     - phase de compactage pour rassembler les blocs vides.
 *@author Richard Ludovic
 */

public void ramasseMiettes(){
	this.marquage_balayage();
	this.compactage();
}
```

3.2.5. Le jeux d'essais

```
/* test*/
public static void main(String[] args) throws MemoireException{

	Memoire mem = new Memoire(1000);
	String trace ="";

	System.out.println("début du test mémoire");
	System.out.println("etat initial de la mémoire :");
	System.out.println(mem.toString());

	mem.declCst("CONSTANTE", null, "entier");
	mem.declMeth("f","150", "boolean");
	mem.declVar("x", 10, "entier");
	mem.declVar("y", true,"boolean");
	mem.declTab("tab1", 20, "tab");
	mem.affecterVal("CONSTANTE", 1000);
	mem.affecterType("y", "entier");
	for(int k=0; k<20; k++){
		trace += mem.affecterValT("tab1", k, (k+1));
	}


	System.out.println("etat final de la mémoire :");
	System.out.println(mem.toString());

	System.out.println("fin du test mémoire !");
	System.out.println("trace :");
	System.out.println(trace);
	}
```

On crée une mémoire, on empile des quadruplets et on déclare des tableaux puis on affecte une valeur à une constante, on change le type d'une variable et on affecte des valeurs aux cellules d'un tableau.

On test ainsi les différentes fonctions mémoires.

Le résultat obtenu s'affiche dans la console :

```
Debut du test memoire
etat initial de la memoire :
   > Pile : []
   > Tas :
[NULL|NULL][NULL|NULL][NULL|NULL][NULL|NULL][NULL|NULL][NULL|NULL][NULL|NULL][NULL|
NULL][NULL|NULL][NULL|NULL][NULL|NULL]
[NULL|NULL][NULL|NULL][NULL|NULL][NULL|NULL][NULL|NULL][NULL|NULL][NULL|NULL][NULL|
NULL][NULL|NULL][NULL|NULL]
[NULL|NULL][NULL|NULL][NULL|NULL][NULL|NULL][NULL|NULL][NULL|NULL][NULL|NULL][NULL|
NULL][NULL|NULL]
etat final de la memoire :
   > Pile :
<tab1,0,tab,tab>.<y,true,var,entier>.<x,10,var,entier>.<f,150,meth,boolean>.<CONSTA
NTE,1000,cst,entier>
   > Tas :
[1|1][2|2][3|3][4|4][5|5][6|6][7|7][8|8][9|9][10|10][11|11]
[12|12][13|13][14|14][15|15][16|16][17|17][18|18][19|19][20|-1][NULL|NULL]
[NULL|NULL][NULL|NULL][NULL|NULL][NULL|NULL][NULL|NULL][NULL|NULL][NULL|NULL][NULL|
NULL][NULL|NULL]
fin du test memoire !
```

3.3. Le contrôle de type

Il existe deux sortes de contrôle de type :

- L'un est destiné à vérifier le typage de tous les variables, méthodes, constantes et tableaux et leurs appels au sein du programme MiniJaja.
- Le second se charge de vérifier le type des quadruplets lors de l'interprétation du programme JajaCode.

3.3.1. Contrôleur de type MiniJaja

Ce contrôle s'effectue sur l'arbre obtenu après l'analyse syntaxique. Le processus de vérification a été implémenté via un Visitor. Un package a été spécialement créer pour les classes concernant ce contrôle de type (minijaja.controleurtype). Le visitor prend en paramètre une table des symboles qui se remplit au fur et à mesure du parcours.

La table des symboles permet de vérifier si un symbole existe déjà ou alors de retrouver le type d'un symbole.

Des informations sont donc enregistrer pour chaque symbole :

- Son identifiant
- Son type d'objet (var, cst, meth, tab)
- Sa sorte (entier ou booleen)
- Le profil de ses paramètres (s'il s'agit d'un objet de type meth)
- Sa portée (il s'agit en fait de l'endroit ou il a été déclaré (fonction ou classe))

La table des symboles est construite avec une table de hachage afin d'accélérer les temps d'accès aux données des symboles.

Elle comporte ainsi ses fonctions permettant de retrouver la sorte, le profil ou la portée des symboles. Ses fonctions sont utilisées par le Visitor.

Voici donc comment fonctionne le Visitor :

A chaque nœud que le Visitor parcours, il va vérifier son propre type et appeler la vérification de ses nœud fils.

Suivant le type de nœud la fonction visit() ne renvoie pas la même chose :

- Les nœuds non typés : ce sont les nœuds tels que « instr », « decl », « classe »... qui ne sont pas affecté à un type. Dans ce cas, la fonction renvoi simplement true, si le contrôle de type des fils n'a pas renvoyé d'erreur, sinon des exceptions sont générer suivant l'erreur.
- Les nœuds typés : ce sont les nœuds tels que « exp », « et », « plus » dont le nœud représente en lui même un type. Dans ce cas, la fonction visit renvoi une chaines de caractère si sera analysé par le nœud père afin de vérifié que les types sont correcte. Certains nœuds peuvent aussi renvoyer un profil de paramètres de fonction. Ces nœuds verifie aussi le typage de leur fils. Des exceptions peuvent aussi être levées.

Ce contrôle de type a est utilisé directement pas le compilateur dans la méthode ci dessous :

```
public void controleType() throws MiniJajaControleurTypeException {
	ControleurTypeData ctdata = new ControleurTypeData();
	try{

	    boolean ret = (Boolean)this.executeVisitor(new
ControleurTypeMiniJajaVisitor(),ctdata);
	}catch(MiniJajaVisitorException e){
		throw new MiniJajaControleurTypeException(e.toString());
	}
}
```

3.3.2. Contrôleur de type Jajacode

Ce contrôle vérifie pour chaque opération effectuer par l'interpréteur JajaCode et la mémoire, que les quadruplets sont conforme a ceux attendu par chaque opération. Par exemple, le «neg» ne peut être effectué que sur une constante ayant une valeur entière. Le contrôle se fait dans les fonctions d'interprétation du JajaCode mais aussi dans les fonctions qui agissent sur la mémoire (la pile et le tas). En cas, d'erreur des exceptions sont levés.

Exemple de contrôle de type dans pour l'operation « neg » :

```
Qua q = idata.memoire.depiler();
if(q.getId()!=null){
	throw new InvalidQuaForInstructionException(instr + " : Impossible avec le Qua " +
q +  " car Qua.id n'est pas <null>" );
}else if(q.getVal()==null){
	throw new InvalidQuaForInstructionException(instr + " : Impossible avec le Qua " +
q +  " car Qua.val est <null>" );
```

```
}else if(!q.getObj().equals("cst")){
	throw new InvalidQuaForInstructionException(instr + " : Impossible avec le Qua " +
q +  " car Qua.obj n'est pas <cst>" );
}else if(q.getSorte()!=null){
	throw new InvalidQuaForInstructionException(instr + " : Impossible avec le Qua " +
q +  " car Qua.sorte n'est pas <null>" );
}else{
	if(!q.getVal().getClass().getSimpleName().equals("Integer")){
		throw new InvalidQuaForInstructionException(instr + " : Impossible avec le
Qua " + q +  " car Qua.val<" + q.getVal().getClass().getSimpleName() + "> n'est pas du
type <Integer>" );
	}else{
		idata.memoire.empiler(new Qua(null,-
((Integer)q.getVal()).intValue(),"cst",null));
		idata.appendAdresse(1);
	}
}
```

3.4. Le générateur de Jajacode

La génération du JajaCode du code à partir du fichier MiniJaja est la dernière étape du compilateur après le contrôle de type. Les classes correspondant au compilateur sont stocké dans le package « minijaja.compilateur ».

Cette génération se fait à l'aide d'un Visitor. Le Visitor parcourt l'ensemble de l'arbre d'analyse, et applique les règles de compilation de chaque nœud donné dans le cours. Lorsque que la fonction arrive sur un nœud, elle génère le JajaCode du nœud et celui des fils de celui-ci.

Un objet est passé en paramètre au Visitor, il permet de mémoriser certaines valeurs utiles lors que la compilation (Objet CompilateurData) :

- l'adresse de la prochaine instruction JajaCode généré
- le numéro de l'entête de fonction dont on veut générer le JajaCode
- un flag pour savoir s'il l'on doit générer le JajaCode de ce nœud, ou alors le code des retraits à lui effectuer

Le Visitor renvoi au final une chaine de caractère contenant l'ensemble du JajaCode. Puis cette chaine de caractère est enregistrée dans le fichier texte renseigné par l'utilisateur.

Voici comment est appelé le Visitor effectuant la compilation :

```
try{
	String compil = (String)this.executeVisitor(new CompileMiniJajaVisitor(),new
CompilateurData());
}catch(MiniJajaVisitorException e){
	throw new MiniJajaCompilateurException(e.toString());
}
```

Exemple : compilation du noeud neg

```
CompilateurData compdata = (CompilateurData)data;
String ret = "";
// e
ret += node.jjtGetChild(0).jjtAccept(this, data);
// neg
ret += compdata.getAdresse();
ret += " neg;¥n";
compdata.appendAdresse(1);
return ret;
```

3.5. L'interpréteur de jajacode

L'interpréteur JajaCode permet d'exécuter un fichier contenant des instructions JajaCode. Il est stocké dans le package « jajacode.interpreteur ».

L'interprétation s'effectue en deux étapes :

- L'analyse syntaxique du programme JajaCode avec JJTree.
- L'exécution du programme sur l'arbre obtenu.

Une grammaire a été défini pour parser le JajaCode et obtenir un arbre du programme.

Une fois l'arbre obtenu, un Visitor est chargé d'interpréter les nœuds de l'arbre. Un objet est transmis au Visitor (Objet InterpreteurJajaCodeData). Cet objet contient une mémoire (pile et tas) sur laquelle on doit effectuer l'interprétation mais aussi une adresse qui correspond à l'adresse de la prochaine l'instruction à exécuter.

L'appel du Visitor l'arbre va pas interpréter l'ensemble des instructions contenu dans l'arbre, mais seulement celle à l'adresse indiqué. Ce choix a été fait afin de facilité l'intégration du de l'interprétation en mode pas à pas.

Voici comment est appelée l'interprétation du programme complet :

```
public void interpreteJajaCode() throws JajaCodeInstructionException {
	while(this.hasNextInstructionToInterperte()){
		this.interpreteNextInstruction();
	}
}

public void interpreteNextInstruction() throws JajaCodeInstructionException {
	try{
		System.out.println(this.executeVisitor(new
ToStringJajaCodeInstrVisitor(),new Integer(idata.getAdresse())));
		this.executeVisitor(new InterpreteJajaCodeVisitor(),idata);
		System.out.println(idata.memoire);
	}catch(JajaCodeVisitorException e){
		throw new JajaCodeInstructionException(e.toString());
	}
}
```

L'interpréteur effectue des contrôles pour chaque opération sur les types des quadruplets utilisés par celles-ci.

A noter que l'interprétation d'un programme contenant des tableaux ne marche que partiellement. Ceci est du à un problème de gestion avec le tas.

Voici comment s'effectue l'interprétation du nœud de l'instruction « neg » :

```
InterpreteurJajaCodeData idata = (InterpreteurJajaCodeData)data;
String instr = "Instruction <" + (String)node.jjtAccept(new
ToStringJajaCodeInstrVisitor(),data) + "> a la ligne " + idata.getAdresse();
try{
	Qua q = idata.memoire.depiler();
	if(q.getId()!=null){
		throw new InvalidQuaForInstructionException(instr + " : Impossible avec le
Qua " + q +  " car Qua.id n'est pas <null>" );
	}else if(q.getVal()==null){
		throw new InvalidQuaForInstructionException(instr + " : Impossible avec le
Qua " + q +  " car Qua.val est <null>" );
	}else if(!q.getObj().equals("cst")){
		throw new InvalidQuaForInstructionException(instr + " : Impossible avec le
Qua " + q +  " car Qua.obj n'est pas <cst>" );
	}else if(q.getSorte()!=null){
		throw new InvalidQuaForInstructionException(instr + " : Impossible avec le
Qua " + q +  " car Qua.sorte n'est pas <null>" );
	}else{
		if(!q.getVal().getClass().getSimpleName().equals("Integer")){
			throw new InvalidQuaForInstructionException(instr + " : Impossible
```

```
avec le Qua " + q +  " car Qua.val<" + q.getVal().getClass().getSimpleName() + "> n'est
pas du type <Integer>" );
                }else{
                        idata.memoire.empiler(new Qua(null,-
((Integer)q.getVal()).intValue(),"cst",null));
                        idata.appendAdresse(1);
                }
        }
}catch(MemoireException e){
                throw new JajaCodeMemoireException(instr + " : " + e.toString());
}
return true;
```

Les fonctions du Visitor renvoie true si l'interpretation s'est bien déroulé sinon des exception sont levé pour expliqué l'erreur.

3.6. L'interpréteur de MiniJaja

Partie du compilateur qui interprète directement le programme Minijaja à partir de règles de manipulation des états mémoires décrite dans le support de cours. Dans ce cas de figure, le compilateur ne passe pas par la génération du code intermédiaire Jajacode.

Malheureusement, faute de temps et de moyens, cette partie n'a pu être finalisée dans les délais impartis.

4. Bilan et perspectives

Dans son état actuel le compilateur est opérationnel en passant par la génération du code intermédiaire, le jajacode. Le programme utilise une interface type lignes de commandes et permet l'exécution d'un programme pas à pas pour le débogage.

Cependant suite à des problèmes organisationnels et des baisses de productivités répétées de certains membres de l'équipe, des points du projet n'ont pu être mené à terme, comme l'interpréteur Minijaja, ou n'ont tous simplement été écartés de la phase de développement afin de ne pas alourdir une charge de travail déjà importante. Ce fut le cas par exemple des diverses optimisations.

Concernant l'interface, le projet était initialement prévu pour tourner sur un Eclipse RCP et implémenter un système d'interfaces graphiques de type JFace. Or, les nombreux problèmes de stabilité de l'application rencontrés tous au long du développement, la complexité de l'approche JFace, le caractère novateur de cette technologie (donc peu documenté) et le crash total de l'application suite au rajout de la coloration syntaxique une semaine avant la fin du projet, nous ont contraint à ne pas présenter de rendu final basé sur cette approche, essentiellement par soucis de fiabilité. Cela mais en évidence une erreur de choix technologique importante qui s'est répercutée sur l'avancement du projet.

La structure particulière de notre groupe et la difficulté à mettre en place une organisation efficace, outre les problèmes évidents de productivité que cela implique, nous ont également empêchées de mettre en place une méthode de suivi de projet efficace, notamment la méthode XP.

Sur un plan purement humain cette fois, l'ensemble des membres du projet ont eu la possibilité de tirer d'important enseignement de cette situation. Bien que pour la plupart encore novice dans la réalisation d'une tache de cette importance, chacun avait au démarrage de ce projet une conception bien arrêtée de la manière dont il devait ce dérouler. Les contraintes rencontrés, les réussites obtenues par un redoublement des efforts, mais également les échecs nous ont appris à appréhender avec un regard neuf, plus empreint d'expérience, ce que pourrai être notre travail futur, avec ses avantages bien sur, mais également avec ses inconvénient et son lots de défis à relever.