Nancy-Université
Université Henri Poincaré

Unité d'enseignement LCIN4U51
Projet de Synthèse
Licence Informatique
Juin 2010

Auteur Jan KEROMNES

Paléo

Rapport du Projet de Synthèse

Tiens, des pommes... miam !

Capture d'écran

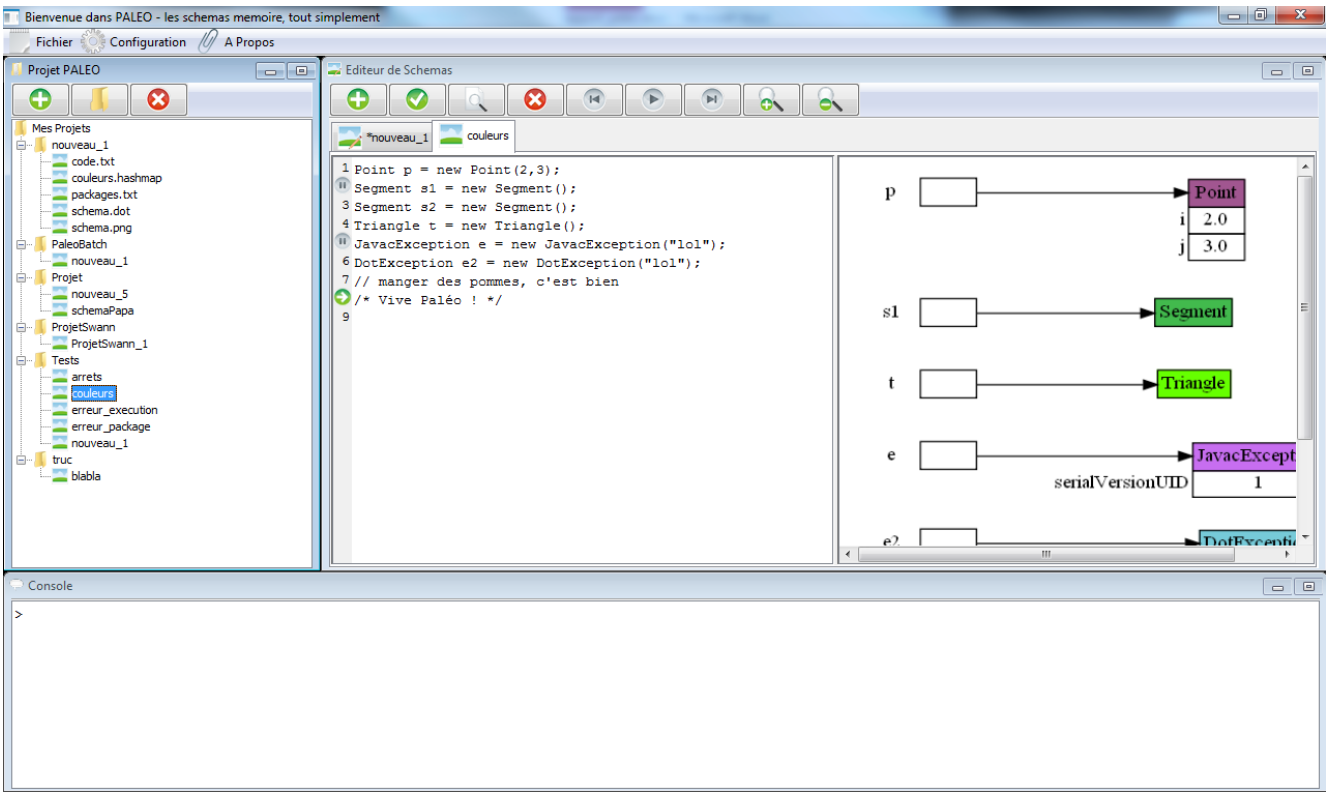


Table des Matières

1) Introduction

1.1) Définition

Le nom de code de ce projet de synthèse, « **PALÉO** », est un acronyme signifiant « **P**etite **A**pplication **L**ogicielle d'**É**tude des **O**bjets ».

Il s'agit d'un « pseudo-compilateur », qui analyse un code arbitraire simple (écrit en Java), et qui génère ensuite un « schéma mémoire », dessin représentant l'état de la mémoire pendant ou à la fin de l'exécution du code.

1.2) Intérêts et objectifs du projet

De part sa nature de projet de synthèse, l'intérêt premier de **Paléo** est de nous faire mettre en pratique les connaissances que nous aurons acquises au cours de ces deux premières années de Licence Informatique pour concevoir et réaliser un pseudo-compilateur de code Java.

L'objectif de l'unité d'enseignement « Projet de Synthèse » (LCIN4U51) était de nous faire aborder tout au long du Semestre 4 les différentes phases qui amènent un projet de la définition des spécifications à un produit conforme à ces dernières. Ceci constitue une introduction à la réalisation de projets informatiques, qui est une compétence réellement indispensable dans le monde de l'informatique, autant dans les milieux de la recherche que de l'ingénierie.

L'idée de pouvoir générer automatiquement un schéma mémoire à partir d'un code Java simple présente également un intérêt d'ordre pédagogique. En effet, mené à bien, **Paléo** permettra aux futurs étudiants de première année d'assimiler facilement les notions-clefs de la programmation, et plus précisément celles de la programmation par objets, à savoir des concepts comme la variable ; le type d'une variable, primitif ou objet ; les affectations ; les attributs, les méthodes, etc. En outre, l'approche de la découverte et de la résolution des erreurs de compilation/exécution sera facilitée par l'utilisation de messages d'erreurs intuitifs et traduits en français, permettant ainsi aux nouveaux étudiants d'acquérir rapidement des compétences d'analyse d'erreurs et de « debugging », ainsi que de bons réflexes de programmation en général, pour construire leur apprentissage sur des bases solides.

1.3) Spécifications techniques

Le projet « **PALÉO** » sera développé pour les **technologies Java Sun**, et effectuera l'analyse du code utilisateur grâce à l'**introspection** ainsi qu'aux bibliothèques **JFlex** et **Cup**. L'application se décomposera en deux parties :

- Un **mode Batch**, utilisable directement en ligne de commande, prenant comme arguments un fichier contenant du **code**, et un fichier contenant les **packages** utilisés dans ce code. L'image sera générée grâce à « **Dot** », un logiciel du package GraphViz, et affichée automatiquement par l'appel à une visionneuse d'images, comme par exemple « Eog » sous l'environnement Gnome (Linux Ubuntu).

- Une **Interface Graphique** (GUI), proposant à l'utilisateur un fonctionnement **intuitif** et une meilleure **ergonomie**, offrant un **accès rapide** à de nombreuses fonctionnalités supplémentaires, comme par exemple l'édition intelligente de codes, la gestion de projets, la compilation pas-à-pas avec points d'arrêts, etc.

2) Conception de l'application

Nous souhaitons procéder à la conception de l'application **Paléo**. Il s'agit ici d'identifier et d'analyser les différents problèmes qui se poseront lors de la programmation avant même d'écrire la moindre ligne de code. En effet, nous cherchons à travailler de manière méthodique, réfléchie, et efficace ; il ne faudrait surtout pas partir sur de mauvaises bases ! Tâchons donc de décomposer le processus de conception en plusieurs sous-problèmes, que nous nous efforcerons de traiter au mieux l'un après l'autre.

Le tout premier problème auquel nous nous retrouvons confrontés est assez simple à identifier, et découle directement des spécifications de **Paléo**…

2.1) Comment dessiner un schéma mémoire ?

Il s'agit peut-être ici de la fonction la plus importante de notre projet, et fait en quelque sorte office de clef de voûte de **Paléo**, sans laquelle l'application ne peut tout simplement pas exister. Etudions donc avec soin les tenants et les aboutissants de la génération d'un schéma mémoire…

Il est indiqué dans les spécifications du projet que nous utiliserons la commande « **Dot** » pour dessiner l'image. C'est un logiciel du package GraphViz, ensemble d'outils open source créés par le laboratoire de recherche AT&T, qu'il convient donc d'installer sur les machines sur lesquelles nous travaillerons.

2.1.1) Fonctionnement de « Dot »

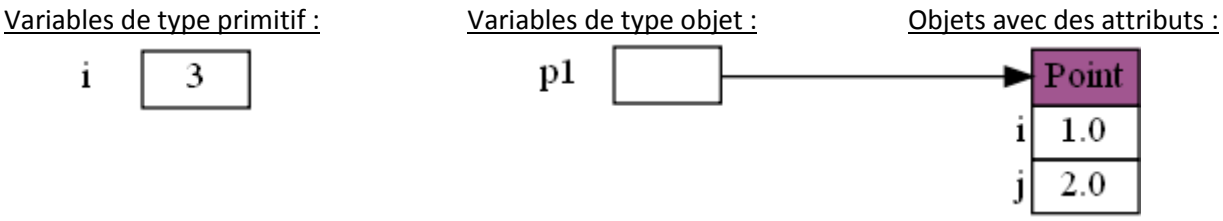
Il est toujours très important de bien s'approprier les outils avec lesquels on travaille, c'est pourquoi nous allons ici tâcher de comprendre le fonctionnement de Dot.

Dot est un outil permettant de dessiner des graphes orientés à partir de leur description en langage **dot**, et de les exporter dans les formats d'images classiques : JPEG, GIF, PNG ou même SVG :

Java ne supportant actuellement pas la technologie des images vectorielles offerte par le format **SVG**, nous nous contenterons d'utiliser le format **PNG**. Une autre solution aurait été d'implémenter la bibliothèque open source « **Batik** » afin d'importer la technologie vectorielle sous Java, mais par soucis de simplicité, nous nous contenterons de travailler avec un format d'images matricielles.

2.1.2) Eléments d'un schéma mémoire

Rappelons succinctement la définition d'un schéma mémoire : Il s'agit d'une représentation de l'état de la mémoire suite à l'exécution d'un programme quelconque. Dans **Paléo**, nous analyserons uniquement des programmes Java orientés Objet. Voici donc la liste des composants que nos schémas mémoires seront susceptibles de contenir :



Graphiquement, cela signifie que nous dessinerons des cases et des tableaux de cases. Une case sera identifiée par le nom de sa variable, et contiendra la valeur de cette variable si elle est de type primitif, ou indiquera par une flèche le tableau de cases correspondant à l'objet qu'elle référence. Un tableau de cases comportera le nom de la classe de son objet, sur un fond de couleur spécifique, et une liste d'attributs qui seront également des cases, primitives ou pointant vers de nouveaux objets.

2.1.3) Comment générer la description d'un schéma mémoire ?

Après avoir maîtrisé la syntaxe du langage dot, nous réaliserons une petite application Java nommée « **Paléo0** ». Cette application produira la description d'un graphe orienté dans un fichier texte, et fera ensuite appel à la commande **Dot** pour générer le fichier image correspondant.

Pour ce faire, nous allons modéliser les différents éléments du schéma mémoire au sein d'une structure arborescente, le « **Graphe Abstrait** ». Celui contiendra donc des « **Nœuds** » de type« **Case** » et de type « **Tableau** ». Les « Nœuds » de type « Case » seront différenciés en « **CaseValeur** », lorsqu'elles représenteront des variables primitives, nulles on non définies, et « **CasePointeur** », lorsqu'elles pointeront vers un « Tableau ». Les « Nœuds » de type « Tableau » contiendront une collection de sous-« Nœuds » de type « **Ligne** », séparés de façon similaire au cases en « **LigneValeur** » et « **LignePointeur** ». Une foi construit, le « Graphe Abstrait » fera appel à un « **Générateur Dot** » pour générer un code en langage Dot décrivant le schéma mémoire final. (Note : Ce générateur pourra s'occuper de générer une couleur différente unique pour chaque nom de classe du schéma mémoire, en convertissant par exemple le « hash code » du nom de classe en couleur hexadécimale.)

Nous allons donc créer notre premier package, « **paleo.graphe** », et les éléments écrits en gras dans le paragraphe précédent en seront les classes. La méthode « main » de **Paléo0** ajoutera des éléments de schéma mémoire au « Graphe Abstrait », lui fera générer la description du schéma et lancera Dot pour produire une image PNG, qui sera ensuite affichée à l'écran grâce à la commande « eog ».

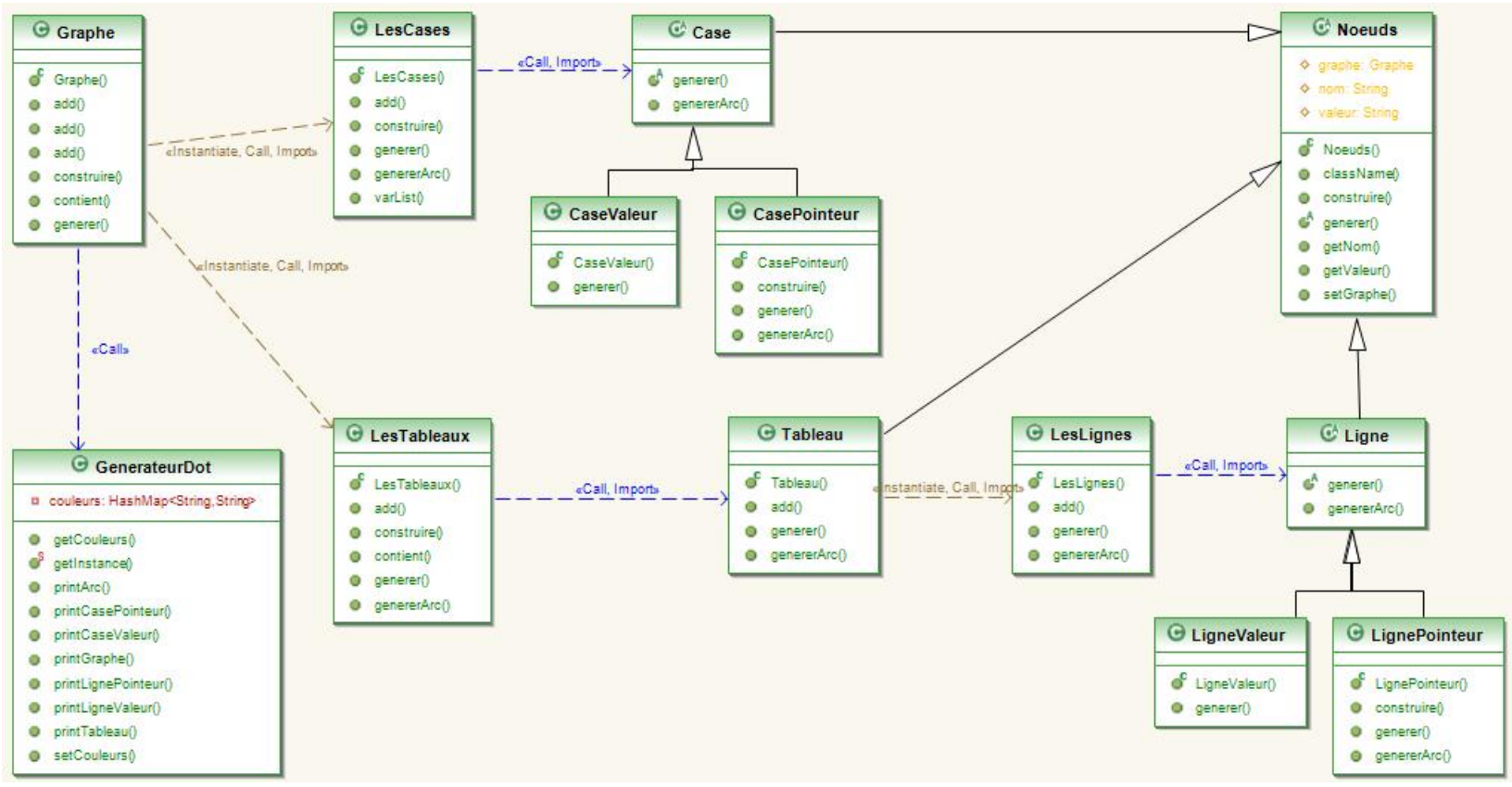
2.1.3) L'introspection, un premier pas vers l'automatisation

Une fois l'application **Paléo0** rendue fonctionnelle, elle est capable de générer un schéma mémoire à partir d'un Graphe Abstrait, que nous aurons au préalable pris soin de garnir manuellement de « cases » et de « tableaux ». Grâce à la réflexion offerte par Java, cette garnison du graphe devient en partie automatisable.

En **POO** (**P**rogrammation **O**rientée **O**bjets), l'**introspection** est la capacité d'un programme à examiner ses structures internes, comme par exemple ses **objets** et leurs **classes**. Dans le cadre de notre application, cela nous permettrait de récupérer automatiquement les informations utiles pour une variable objet, afin de construire à partir de sa « case pointeur » le « tableau » vers lequel elle pointe. En effet, il est possible de récupérer le nom de la classe d'un objet, ainsi que la liste de ses attributs. De façon récursive, les « lignes pointeur » correspondantes aux attributs pourront être construites de la même manière. Il ne resterait donc plus qu'à indiquer au graphe les variables déclarées dans le code utilisateur, et ce dernier se débrouillerait pour construire tout seul le reste de ses éléments.

Pour effectuer ce premier pas vers une génération de schéma mémoire automatisée, nous allons faire évoluer notre application « **Paléo0** » en « **Paléo1** », en implémentant l'utilisation de l'introspection.

Figure 1. **Diagramme UML** de synthèse du package « **paleo.graphe** »



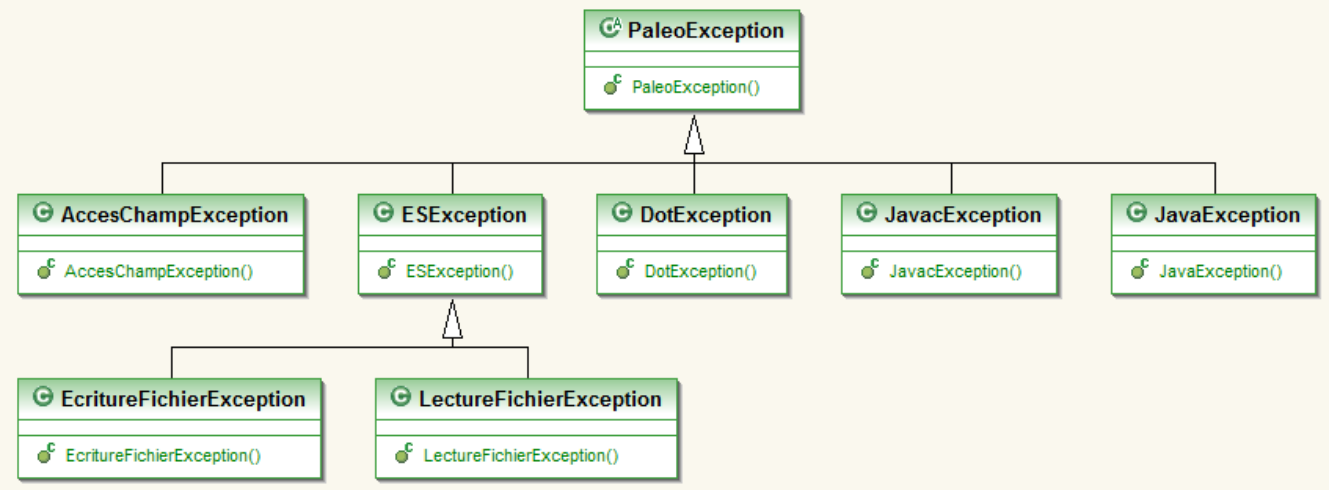
2.2) Au cœur de la compilation, l'analyse du code

Avec **Paléo1**, le dernier pas à franchir pour parvenir à une génération de schémas mémoire complètement automatique est de trouver un moyen pour fournir à son « graphe abstrait » les « cases valeur » et les « cases pointeur » correspondantes aux variables déclarées dans le code utilisateur. Pour ce faire, nous utiliserons des outils d'analyse lexicale et syntaxique, respectivement **JFlex** et **Cup**, pour garder en mémoire l'état des variables pendant l'exécution du code, et nous stockeront cet état dans une nouvelle structure arborescente, l'« **Arbre Abstrait** ». Nous serons alors en mesure de faire générer à cet arbre le code Java permettant à **Paléo1** de compléter son « graphe abstrait », que nous pourrons ensuite compiler et exécuter.

2.2.1) Des exceptions personnalisées

Notre nouvelle application « **Paléo2** », implantant à la fois l'**analyse** du code utilisateur et notre ancienne application **Paléo1**, comportera une chaine de compilation assez conséquente. En effet, il faudra commencer par compiler et exécuter le code utilisateur, pour vérifier qu'il ne comporte aucune erreur, puis importer le résultat de l'analyse de ce code dans **Paléo1**, qu'il faudra également compiler et exécuter. Plusieurs types de dysfonctionnements peuvent se produire au cours de ce processus, et nous allons gérer ce problème en levant, propageant et gérant des exceptions Java personnalisées. Les principaux dysfonctionnements que nous devrons gérer sont les suivants : Les erreurs de compilation Java, d'exécution Java, de génération Dot, et les erreurs d'entrée/sortie. Ajoutons donc à notre projet un nouveau package, « **paleo.exceptions** », contenant toutes les exceptions dont nous aurons besoin.

Figure 2. **Diagramme UML** des **exceptions** utilisées dans Paléo



2.2.2) Des outils performants pour plus d'efficacité

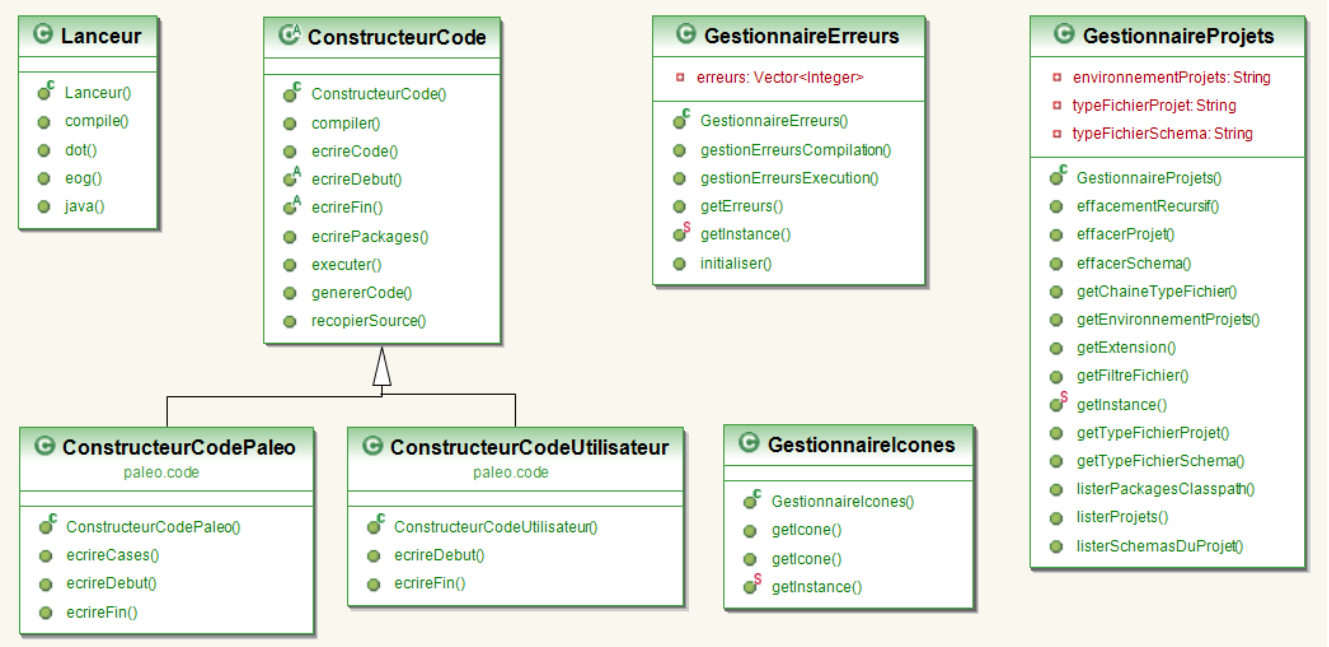
Comme indiqué dans le paragraphe précédent, la chaine de compilation de **Paléo2** sera longue, et fera appel à de diverses fonctions. Nous pouvons factoriser intelligemment notre code en gérant ces fonctions au sein d'un package « **paleo.outils** ».

L'action que nous allons répéter le plus souvent est l'appel d'une commande (compilation, exécution, dot, eog,…), c'est pourquoi nous allons créer une classe « **Lanceur** », qui comportera plusieurs méthodes, chacune effectuant l'appel d'une commande différente. Si l'appel d'une de ces commandes échoue, le « Lanceur » lèvera l'une des exceptions créées dans la partie précédente.

Nous aurons également besoin de construire des fichiers de code Java d'une forme particulière, c'est pourquoi nous ajouterons également une classe « **ConstructeurCode** » à notre package.

Nous ajouterons en outre trois gestionnaires, qui seront des classes uniques tout comme le « GenerateurDot ». Le rôle du « **GestionnaireErreurs** » sera de chargera de traduire les messages d'erreurs provenant des exceptions de compilation et d'exécution (correction du nom de fichier et de la ligne de l'erreur) ; le « **GestionnaireIcones** » s'occupera de centraliser le chargement et le stockage en mémoire des icônes de notre future interface graphique ; et la manipulation des projets du « workspace » de **Paléo** sera effectuée par le « **GestionnaireProjets** ».

Figure 3. **Diagramme UML** de synthèse des **outils** de Paléo

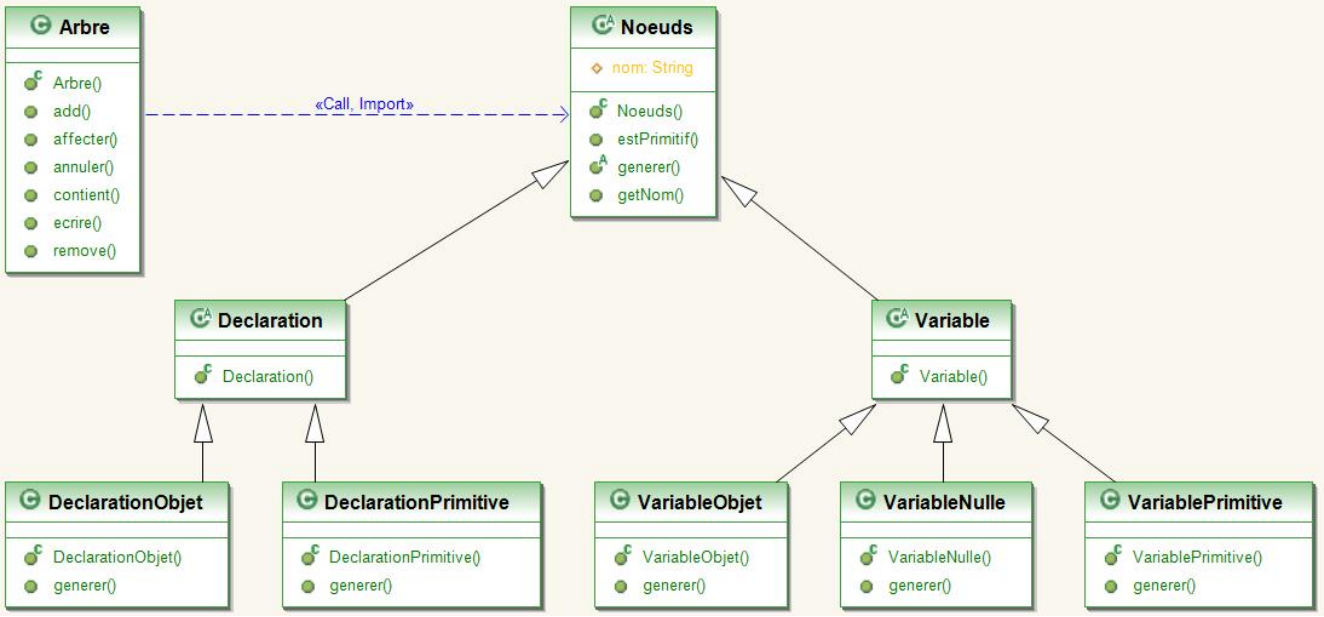


2.2.3) Un structure arborescente modélisant l'état des variables

Comme décrit précédemment, le rôle de l'analyseur de **Paléo** sera d'examiner étape par étape les instructions du code utilisateur, et de garder en mémoire tout au long de ce processus l'état des variables déclarées, affectées ou annulées. Nous devons donc lui fournir une structure adaptée, que nous nommerons l'« **Arbre Abstrait** ». A cet « Arbre Abstrait », l'analyseur pourra ajouter ce que nous appellerons des « **Declarations** », à savoir des variables déclarées mais pas encore affectées à une valeur, ainsi que des « **Variables** », chacun de ces éléments héritant d'une classe « **Nœud** », possédant un nom unique, qui pourra être utilisé ensuite par le Graphe pour récupérer les valeurs de chacune des variables. L'Arbre pourra également mettre à jour l'état

d'affectation ou d'annulation des variables, et nous séparerons ici aussi les nœuds « ***Primitifs** » des nœuds « ***Objets** ». Le cas d'une « **VariableNulle** » sera traité à part, puisqu'il aura pour effet d'ajouter une « CaseValeur » au Graphe sans pour autant correspondre à une « VariablePrimitive ». Une fois l'analyse du code terminée, l'Arbre Abstrait sera capable d'**écrire** la partie manquante du code de Paléo1, et nous procéderons ensuite à sa compilation et à son exécution.

Figure 4. **Diagramme UML** de synthèse de l'**Arbre Abstrait** de Paléo



2.2.4) L'analyse lexicale et syntaxique du code utilisateur

Après avoir effectué toutes les préparations nécessaires, nous pouvons enfin entrer dans le vif du sujet de cette partie sur l'analyse de code : Nous allons développer, grâce aux bibliothèques **JFlex** et **Cup**, un analyseur lexical et syntaxique pour notre code utilisateur. Cet analyseur possèdera une instance d'« Arbre Abstrait », parcourera le code instruction après instruction, et s'occupera de tenir au fait les éléments de l'Arbre Abstrait.

Ecrivons un fichier « **AnalyseurLexical.jflex** », qui comportera un ensemble d'**expressions régulières** définissant les séquences de caractères possibles pour former des **tokens** (ou **léxèmes**), qui seront ensuite « consommés » par un analyseur syntaxique. Ce fichier jflex donnera lieu à la génération de la classe « **AnalyseurLexical** ».

Ecrivons également un fichier « **Grammaire.cup** », qui définira les règles d'un langage formel simple et proche du langage Java, et qui donnera lieu à la génération de la classe « **AnalyseurSyntaxique** ».

Muni de ces deux analyseurs et de la structure de l'Arbre Abstrait, nous sommes désormais en mesure d'effectuer une analyse sommaire du code utilisateur, afin d'optenir les informations nécessaire pour compléter le code de l'application **Paléo1**, et donc de générer au final un schéma mémoire de façon entièrement automatiquement.

2.2.5) Bilan : Fonctionnement du « mode Batch »

Ayant réuni au cours de cette partie tous les éléments nécessaires à la conception du « **mode Batch** », nous en avons terminé avec la chaîne de compilation. L'étape de finition restante est de bien regrouper et lier tous les éléments, ainsi que d'écrire la classe « **PaleoBatch** » qui contiendra la méthode « **main** », appelée avec 2 paramètres : un fichier de code et un fichier de packages, exécutant la chaîne de compilation, et appelant en dernier une commande d'affichage sur le fichier image produit. Pour illustrer ce fonctionnement, un **diagramme de séquence** du mode Batch a été réalisé pour un appel sur fichier code contenant l'instruction « `Point p1 = new Point(10., 11.) ;` » et sur un fichier package contenant « `geometrie` ».

Ce diagramme étant de taille trop importante pour figurer dans ce rapport, il peut cependant être retrouvé sous le répertoire « `paleo/sequences/batch` » de l'archive du projet.

2.3) Une Interface Graphique pour Paléo

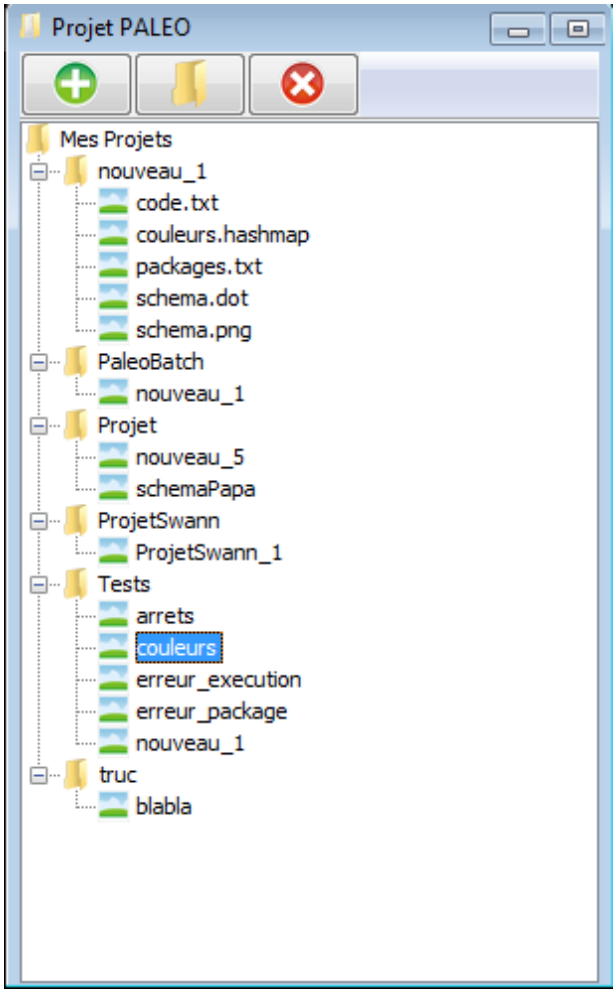
Après avoir terminé la chaîne de compilation et le mode Batch, nous allons maintenant nous attaquer au développement d'une Interface Graphique. Utilisant la bibliothèque « **javax.swing** », son rôle sera d'offrir à l'utilisateur un accès simplifié aux fonctions principales de Paléo, ainsi qu'à nombre de nouvelles fonctionnalités.

Respecant le schéma de programmation MVC (Modèle-Vue-Contrôleur), elle est composée de plusieurs « **Vues** », puisant leurs informations dans le « **ModelePaleo** » et se mettant à jour à chaque modification.

Comme précisé lors de la **démonstration**, son environnement de travail se décompose en trois sous-fenêtres.

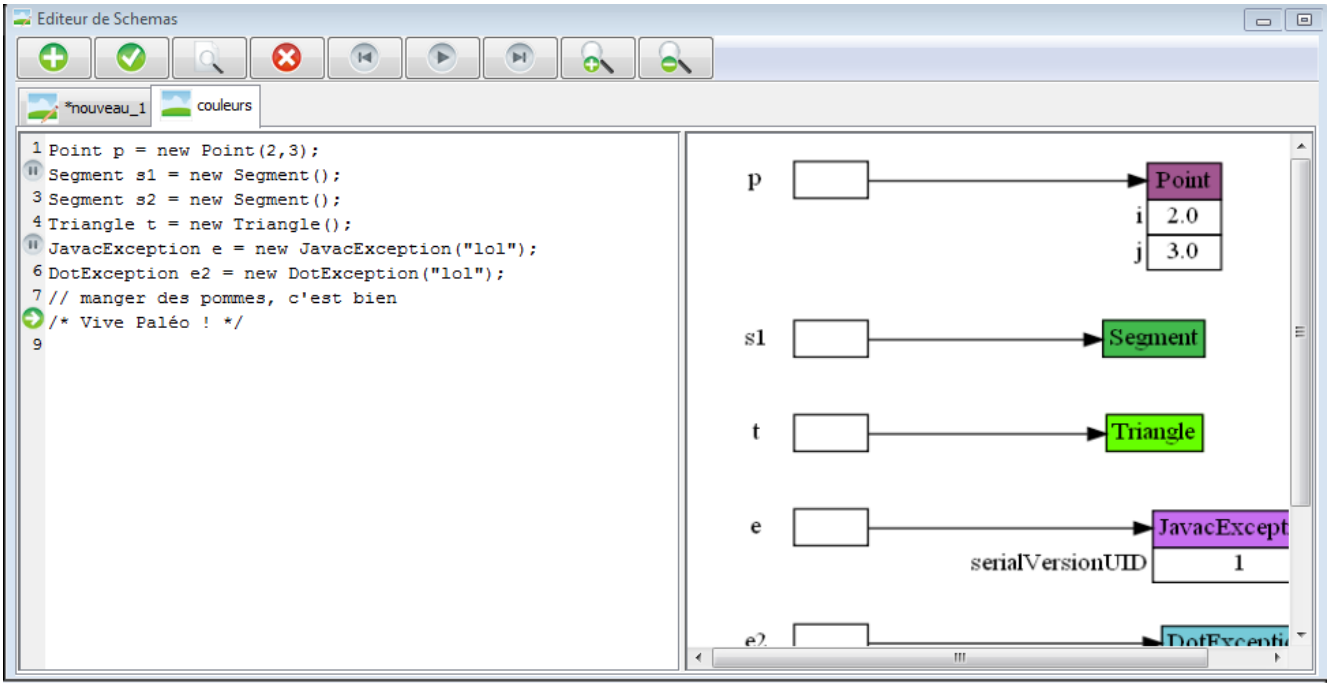
2.3.1) L'arborescence du « workspace »

Notre interface graphique proposera aux utilisateurs d'enregistrer leurs schémas et leurs projets dans une **arborescence interne** à Paléo, proposant des raccourcis comme la création ou la suppression de projets, ainsi que l'ouverture des schémas afin de pouvoir les éditer.



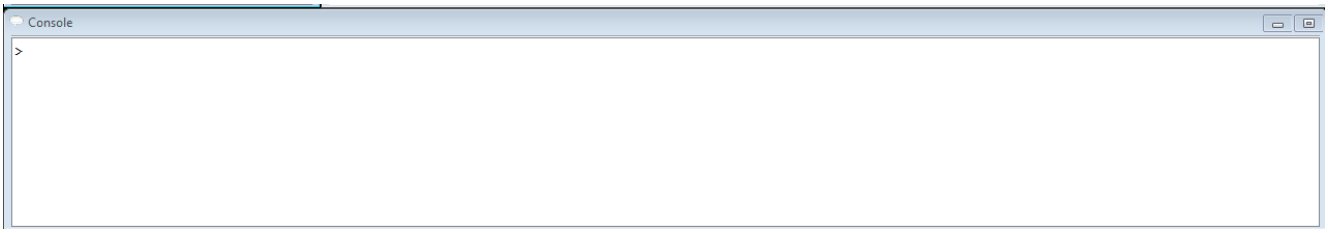
2.3.2) L'éditeur de schémas

Regroupant tous les raccourcis et les fonctionnalités utiles dans un seul menu, **l'éditeur** permet notamment de travailler sur plusieurs schémas à la fois, et de les compiler rapidement grâce à un bouton de lancement unique.



2.3.3) La Console

Affichant les messages d'erreurs simplifiés du modèle, elle permet de comprendre et de corriger rapidement les erreurs éventuelles du code.



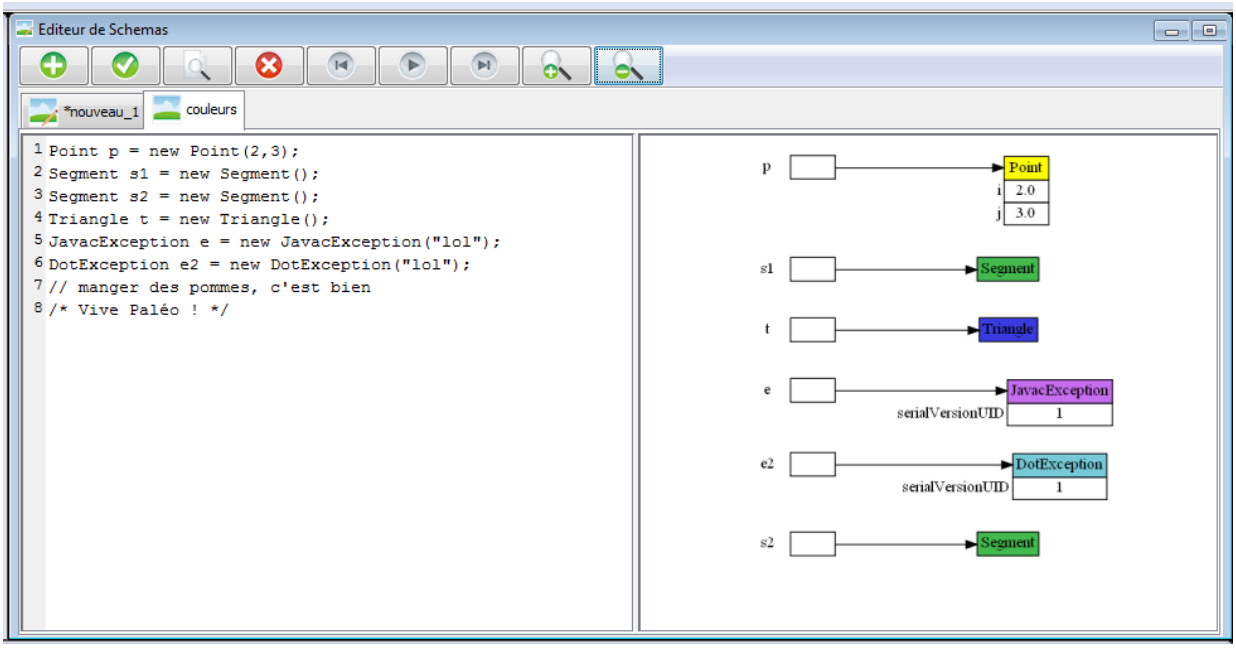
2.3.4) Bilan : Fonctionnement de l'Interface Graphique

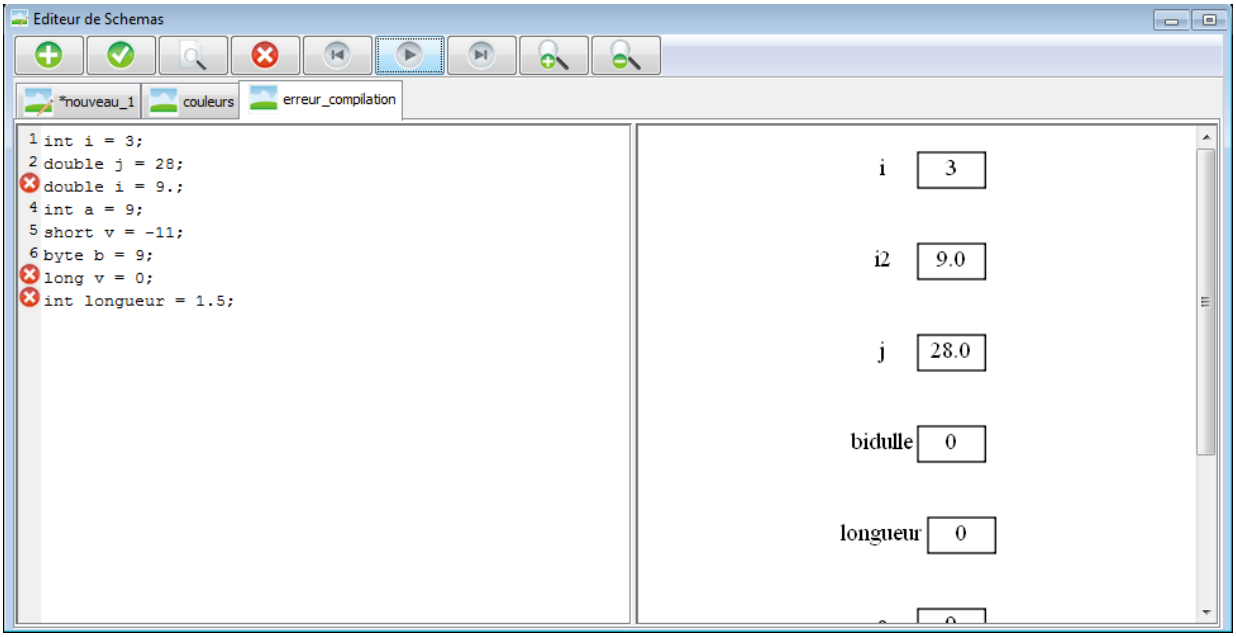
Comme pour le mode Batch, un **diagramme de séquence** a été réalisé pour détailler le fonctionnement et la chaine de compilation de l'Interface Graphique.

3) Conclusion

3.1) Tests

Dans le but de tester et d'expliciter le fonctionnement de l'application finale **Paléo**, plusieurs tests ont été mis au point et se trouvent dans le projet « Tests » du workspace. Ils sont accessibles rapidement depuis l'Interface Graphique, et démontrent le bon fonctionnement de Paléo, conforme aux spécifications imparties, ainsi que ses limitations.





3.2) Bilan de Paléo

Tout au long de ce semestre, nous aurons pu aborder toutes les étapes de la création d'un projet, et avons amené Paléo de la phase de réflexion aboutissant aux spécifications jusqu'au produit final, qui se trouve dans l'archive « **Paleo.tar** » jointe à ce rapport. Nous avons appris beaucoup de choses, rencontré et surmonté les difficultés inévitables de la réalisation d'un projet, et je suis prêt, au-delà de la remise et de la notation de ce projet, de continuer à travailler dessus, pour offrir à nos futurs étudiants de Licence Informatique un Paléo complet, avec une grammaire plus fournie et plus robuste, une interface encore meilleure, et en général un projet qui restera gravé dans l'histoire de la Licence Informatique de l'Université Henri Poincaré de Nancy.