Université de Nice
Sophia Antipolis
Année 2003-2004
Maîtrise d'Informatique

Travail d'étude et de Recherche

JavInspector

Inspecteur de classes Java

Christophe APPIETTO
Jean-Michel ARNAUD
Hassan EL MAHRATI

Encadrants :
Pierre Crescenzo
Philippe Lahire

SOMMAIRE

I. INTRODUCTION

Logiciel d'inspection d'application Java, JavInspector a été réalisé dans le but précis d'analyser et de visualiser les informations contenues dans les fichiers **.class**.

Un fichier **.class**, résultat de la compilation d'un programme Java, est interprété par la machine virtuelle Java lors de son exécution.
Après la compilation d'une classe, étant le seul fichier nécessaire à sa représentation, il contient l'ensemble des informations qui y étaient présentes.
JavInspector permet de visualiser ces informations.

Connaître les informations sur une classe et pouvoir les récupérer, sans disposer du code peut être un atout considérable et un luxe dans le domaine de la programmation : JavInspector a été réalisé dans cette optique.
Le but principal de cette application est d'extraire le maximum d'informations concernant une classe. En effet il est possible de visualiser les informations collectées et d'explorer les classes dont elle dépend à travers une interface claire et agréable.
Une fonctionnalité intéressante du logiciel est la capacité à sauvegarder l'analyse d'une classe sous la forme d'un fichier XML de manière à pouvoir la réutiliser dans une application externe.

Nous avons tenté de respecter au mieux les délais et contraintes imposés par les encadrants, pour fournir les différentes versions, mais après la rencontre de quelques problèmes de développement et l'ajout de quelques contributions à notre initiative, le planning prévu a dû évoluer.

Ainsi, nous allons vous présenter le travail que nous avons réalisé, en détaillant notre organisation, en précisant les choix que nous avons effectués et en fournissant le résultat de notre entreprise ainsi que les apports personnels.

II. CAHIER DES CHARGES SYNTHETIQUE

1. Résumé

JavInspector est un logiciel d'inspection d'application Java. La version 1.0 permet d'analyser des fichiers .class et de visualiser les informations retirées à l'aide de vues graphiques. Ainsi l'application est en mesure de donner des informations sur les constructeurs, les champs, les méthodes mais aussi sur les interfaces implémentées ainsi que sur l'héritage des classes. De plus, elle permet de rechercher des informations dans le fichier à l'aide d'un moteur de recherche.

La nouvelle version de JavInspector devra être en mesure d'étendre la palette des informations récupérées dans un « fichier class » et de les synthétiser dans un fichier XML afin d'offrir la capacité d'une mise en page spécifique ou d'une utilisation par un programme externe.

Le principal objectif des futures versions est une installation facile pour tout client, il sera donc appréciable de fournir des paquetages de type RPM ou DEB permettant une décompression et installation automatique ainsi qu'une gestion des conflits ou des versions déjà présentes sur le système. Le second objectif sera d'obtenir une application d'utilisation facile et appréciable, donc autant du point de vue de la convivialité que de la richesse des fonctionnalités. Aucun bogue ne devra être présent ni dans la version de départ qui servira de base, ni dans les fonctionnalités qui seront ajoutées.

2. Organisation du projet

2.1. Processus

La première étape consiste à comprendre et à analyser les versions précédentes de l'application JavInspector déjà disponibles.
Suite à cet audit de code, une liste d'améliorations et de modifications devra être réalisée. Cette tâche essentiellement réalisée lors de la première partie pourra s'exécuter durant la totalité du projet. Les modifications principales seront essentiellement :

- amélioration de la Javadoc
- amélioration et nettoyage du code
- modification des paquetages java pour des noms plus cohérents
- ajouts de nouvelles fonctionnalités

Une tâche importante sera la détection puis la résolution des nombreux bogues figurant dans les versions antérieures.
Cette première partie donnera lieu à la version 1.2 de JavInspector.

La seconde étape représente la partie la plus conséquente du projet : elle consistera à rechercher les moyens d'accéder aux informations concernant le corps des méthodes (variables lues/modifiées, appels d'autres méthodes, etc.). Suite à cette recherche nous incorporerons les résultats graphiquement dans l'application. Une fois cette partie terminée, nous aboutirons à la version 1.3 de JavInspector.

La troisième phase devra doter le logiciel d'une vision XML. Les menus d'aide détaillés ainsi que les documents annexes(manuel de l'utilisateur, …) seront rajoutés dans cette phase finale. Si toutes les phases sont réalisées avec succès et sans trop de difficulté l'équipe rajoutera d'autres fonctionnalités (internationalisation, possibilité de lancer l'application sous forme d'Applet, …).

Chacune des précédentes étapes devra être réalisées entièrement avant le commencement d'une nouvelle.

2.2. Limites et interfaces

Le logiciel développé ne devra en aucune manière modifier le fichier qu'il analyse: il devra être non intrusif. De plus, il s'agit d'une analyse d'un fichier class et en aucun cas d'une exécution (comme le font par exemple des débogueurs).
Cette application étant destiné surtout à des informaticiens, il serait envisageable que celle-ci puisse s'intégrer dans des IDE tels que JBuilder ou Eclipse sous forme de plugins par exemple.

3. Gestion

3.1. Objectifs

Une nouvelle version du logiciel doit être terminée, fonctionnelle et diffusable à la fin du projet même si les fonctionnalités sont limitées.
La version finale doit implémenter les nouvelles fonctionnalités déterminées par le client et l'équipe: Il sera possible de synthétiser les informations récupérées lors de l'analyse dans un fichier XML, l'équipe se contraindra bien sûr à fournir un schéma XML au client permettant de définir la grammaire XML utilisée dans le document. Si le temps en convient, l'équipe propose de reprendre et de mener à son terme la version 1.1 qui permettait de lancer l'application également sous la forme d'une Applet Java. L'équipe de programmeurs s'engage à écrire du code convenablement structuré, très lisible et abondamment commenté. Ces qualités permettront ainsi aux futures équipes de reprendre facilement le code de ce projet.

L'installation et l'utilisation du logiciel devront être portables et faciles. L'équipe devra respecter au mieux les délais de conceptions soumis par le client et devra le tenir au courant de l'évolution du projet via son site Internet. Toutes fonctionnalités rajoutées en plus de ce que le client propose seront les bienvenues.

3.2. Priorités

Le client a imposé un ordre dans le déroulement du projet, c'est à dire que chaque nouvelle étape ne peut être commencée sans que l'étape précédente ne soit entièrement terminée et validée par le client.
L'équipe s'attaquera en priorité au remaniement des paquetages Java (choix des noms de paquetage et répartition des sources de façon plus cohérente et respectant les conventions de nommage) ainsi qu'un travail sur la lisibilité et la documentation des sources. La suppression des bogues et l'amélioration de la structure du code sont également des tâches prioritaires et étalées sur toute la durée du projet. Une fois cette étape terminée, l'équipe pourra commencer à implémenter les fonctionnalités spécifiques à la version 1.3. L'étape concernant la sauvegarde des informations dans un fichier XML pourra donc débuter. En parallèle les menus d'aide et la documentation seront réalisés.
Si le projet arrive à une version opérationnelle sans avoir rencontré de problèmes majeurs, le client propose d'ajouter à l'application la capacité de la lancer sous forme d'Applet Java, ou la possibilité de choisir la langue (internationalisation).

3.3. Hypothèses, dépendances et contraintes

3.3.1. Hypothèses

L'appel d'offre initial formulé par le client est très précis: le client impose les choix technologiques. La seule liberté laissée à l'équipe du programmeur concerne le document XML ainsi que les fonctions propre à la convivialité de l'application. En effet l'équipe devra définir la grammaire utilisée pour ce document à l'aide d'un schéma XML. Il pourra alors utiliser les informations qui y seront synthétisées à d'autres fins.

3.3.2. Dépendances

Le projet étant une prolongation d'une application déjà existante, l'implémentation des nouvelles fonctionnalités dépend de la stabilité des versions déjà existantes (robustesse). Toute version en cours d'implémentation reposera essentiellement sur la version directement antérieure, d'où la nécessité de ne fournir une version qu'une fois celle-ci complètement achevée, stable et robuste.

3.3.3. Contraintes

Le client impose le langage de programmation. Le programme devra être écrit en utilisant la version du j2sdk 1.4.2 de Java. De plus, notamment pour l'implémentation des nouvelles fonctionnalités, seules les bibliothèques standard de la version 1.4.2 de Java pourront être utilisées. Le logiciel doit aussi être portable c'est à dire tourner indifféremment sur un système Linux ou Windows.

3.4. Gestion du risque

Nous avons identifié deux risques majeures : tout d'abord l'équipe n'est pas assurée de résoudre tous les bogues de la version 1.0 de JavInspector qui pourraient influencer l'exécution de la nouvelle version. Lorsqu'un bogue sera détecté lors de l'exécution d'une nouvelle fonctionnalité, il faudra alors détecter la source de l'erreur, comprendre le problème et essayer de le résoudre. La première phase concernant la compréhension de la version 1.0 et des modèles informatiques utilisés est donc essentielle. En effet, elle va déterminer la stabilité de la version 1.2 qui sera la première version fournie par l'équipe et qui constituera la réelle base de travail.
Le deuxième problème que l'équipe a mis en valeur concerne l'implémentation des nouvelles fonctionnalités de l'application. Le raffinement des fonctionnalités demande un réel travail de recherche sachant qu'elle ne peut utiliser d'autres bibliothèques Java.

3.5. Moyens de contrôle

L'équipe est contrainte par le client de livrer régulièrement des versions finies de JavInspector. Il pourra ainsi tester les versions, relever les faiblesses éventuelles et en débattre lors des réunions. Les réunions entre le client et l'équipe se tiendront au moins tous les 15 jours et l'équipe devra établir un compte-rendu électronique hebdomadaire au client répertoriant tout le travail qu'elle a réalisé au cours de la semaine.
L'équipe a décidé d'établir des fichiers tests permettant de vérifier le bon fonctionnement du logiciel. Afin de vérifier la validité des analyses, nous comparerons le contenu des fichiers sources aux résultats retournés. De plus les différentes versions de JavInspector étant disponibles sur Sourceforge, elles pourront être testées par d'éventuels utilisateurs. Grâce au forum de notre site, ils pourront ainsi nous faire part de leurs commentaires ou des éventuelles améliorations.

4. Technique

4.1. Méthodes

Lors de la conception d'un produit, le respect des délais permet de satisfaire le client et surtout de minimiser les coûts. Un travail rapide et efficace permettra une meilleure conception et augmentera de façon considérable la qualité de l'application par ses finitions et de nombreux tests (qui sont à cause d'un retard bien trop souvent négligés).

C'est pour cela que l'équipe a pris le temps de bien réfléchir aux méthodes de conception, développement et organisation du projet.

Tout le travail et tous les choix ont été basés sur une organisation optimale, tant en matière de mise à jour par exemple du site Internet, qu'en matière de conception ou de développement. Le travail à réaliser pour les premières versions, est un travail non ciblé, il concerne la totalité de la plate forme de travail (version 1.0 qui est la base du travail). C'est pour cela qu'il a fallu déterminer des méthodes de travail pour que les membres de l'équipe puissent travailler simultanément et non pas en alternance.
Une spécification a donc été mise en place pour permettre une attribution précise, mais pas stricte des tâches à réaliser en fonction des préférences et des compétences de chacun. Comme toute application formée d'un grand nombre de classes dépendantes les unes des autres, ceci n'aurait pas suffit pour maximiser le travail fournit. En effet, certaines tâches ne pouvant s'effectuer qu'une fois le travail d'un autre membre terminé, l'équipe a donc du se munir de différents outils spécialisés.

En ce qui concerne la distribution, les rapports de bogues, toutes informations ou tout autre besoin du client, un site Internet à été développé pour l'application. Il permet de récupérer l'ensemble des versions, d'y trouver un grand nombre d'informations et même d'interagir grâce à un Forum et une Mailing List avec les autres membres. Un suivi en temps réel du travail y est également disponible.

4.2. Outils employés

Il a été constaté que l'utilisation d'un logiciel tel que JBuilder ou Eclipse augmentait considérablement la vitesse d'implémentation. Pour cela, l'outil JBuilder a été choisi par les membres de l'équipe qui l'apprécient et le connaissent assez bien. Ce choix n'est pas décisif mais permet un certain confort des programmeurs qui n'est pas à négliger car il maintient une certaine motivation et donc qualité de programmation. De même JCreator sera utilisé pour sa rapidité d'exécution et Emacs pour des changements ne nécessitant pas l'aide d'un véritable IDE.

La contrainte majeure lors de la réalisation de JavInspector est de fournir une nouvelle version par tâche importante réalisée. Cette contrainte a obligé l'équipe à trouver une bonne façon pour gérer ces différentes versions. L'utilisation de CVS s'est imposée d'elle-même car comme son nom l'indique « Concurrent Version System », ce système répond tout au besoin des programmeurs. Il va également fournir une très bonne sécurité d'implémentation car il gère des sauvegardes des versions antérieures, et enfin, il va permettre un travail tout à fait parallèle. En effet grâce à CVS, tous les membres de l'équipe pourraient travailler sur la même classe sans risquer d'écraser le travail d'un autre membre.

La gestion du site Internet et de la mise à jour des versions est importante, surtout si les changements y sont fréquents. Le site qui doit être interactif et faciliter aussi le travail de l'équipe est réalisé en Php avec EasyPhp. La mise à jour des différentes versions disponibles et l'utilisation de CVS seront réalisées grâce à Sourceforge qui héberge le site, les sources et permet une modification facile des données. En effet, divers outils seront mis à la disposition du client et des programmeurs comme Php (site Internet), Mysql (Forum), SCP (pour la mise en ligne des versions terminées) ou FTP (pour récupérer les données).

4.3. Documentation

La documentation relative au code sera générée par Javadoc.
La phase de documentation à été établie dans la phase finale :

- Le site web comportera toutes les informations relatives à l'installation.
- Nous fournirons aussi un « manuel de l'utilisateur »
- Le logiciel comportera un menu conséquent d'aide
- Les astuces du jour seront réalisées tout au long du projet.

5. Fonctions du produit

5.1. JavInspector version 1.0

- **Analyse de classe**

La fonctionnalité principale de la version 1.0 est l'analyse de classe. Les informations retirées de cette analyse sont modélisées sous deux formes de représentations bien distinctes. Une vue favorisant la vision de l'héritage et des interfaces implémentées, et une vue disponible sous deux formats, détaillée et globale, pour afficher la structure de la classe. Cette dernière vue détaillée permet de sélectionner les affichages suivants :

- les variables d'instance
- les variables statiques
- l'entête des constructeurs
- l'entête des méthodes d'instance (distinction entre les méthodes d'accès, les méthodes de modification, les méthodes abstraites, les méthodes redéfinies et les autres)
- l'entête des méthodes statiques
- les classes internes
- La vue globale quant à elle permet de visionner:
- tous les types de paramètre et de retour des méthodes ainsi que les éventuelles exceptions lancées
- tous les types de paramètre pris par les constructeurs ainsi que les éventuelles exceptions lancées
- tous les types des variables

Chacune de ces deux vues permet une exploration à travers toutes les classes présentes dans l'analyse pouvant donner suite à une nouvelle analyse.
Un historique des différentes analyses effectuées dans une session est disponible permettant de les consulter à tout moment.
Différentes méthodes s'offrent au client en ce qui concerne la consultation des classes. Il peut effectuer celle-ci soit en spécifiant le chemin en parcourant les répertoires, ou encore en indiquant un jar et le nom complet de la classe.
Cette fonctionnalité présente cependant quelques bogues : l'analyse de certaines classes par le logiciel est parfois impossible.

- **Recherche**

Un moteur de recherche a été implémenté spécialement pour permettre de rechercher les différentes occurrences d'une expression qui lui est fournie.

Il est possible d'effectuer cette recherche dans toutes les classes liées à l'analyse ou alors de restreindre celle-ci à une classe particulière. Une recherche avancée est également disponible permettant à l'utilisateur de spécifier des critères plus précis de recherche.
Cette fonctionnalité semble fonctionner correctement.

5.2. JavInspector version 1.1 (ou JavaWebInspector)

- **Intégration d'une nouvelle interface sous forme d'Applet**

Cette version a pour but d'ajouter une nouvelle interface à l'application sous forme d'Applet afin de permettre une utilisation via Internet du logiciel.

Cette adaptation a rendu le logiciel de moins bonne qualité. De nombreuses fonctionnalités se sont avérées détériorées voir inutilisables : ainsi les recherches ne sont plus disponibles et de plus important bogues ont été détectés.

Une étude comparative de ces deux versions a eu pour conclusion que la version 1.1 rend le logiciel inexploitable et pourrait retarder l'avancé des travaux à cause d'une longue étape de réparation des problèmes. D'un commun accord, l'équipe et les clients ont décidé d'abandonner cette version et de mettre de coté temporairement l'aspect Applet, n'étant plus sur la liste des priorités. La décision prise mutuellement a déterminée la version 1.0 comme base des travaux.

5.3. JavInspector version 1.2

La version 1.2 n'a pour objectif qu'une radicale restructuration du code source. Les paquetages java ont été tous repris :

- modification des noms pour respecter la convention de l'espace de nommage mais aussi pour avoir des noms plus cohérents et plus appropriés que les précédents
- modification de paquetage pour de nombreux fichiers afin d'améliorer la hiérarchie des classes

L'objectif de cette version est de commencer à résoudre les bogues, mais aussi d'améliorer les sources ainsi que de les commenter abondamment. Ces trois tâches pourront être étalées durant la durée du projet.

De même, l'ajout de fonctionnalités facilitant l'ergonomie du logiciel et le rendant plus convivial est laissé au soin de l'équipe. Ainsi le rajout d'une astuce du jour est présent dans cette version permettant d'éclairer l'utilisateur sur quelques principes de fonctionnement.

5.4. JavInspector version 1.3

- **Actions provoquées par les différentes méthodes**

Cette fonctionnalité va permettre de compléter l'analyse de classe en indiquant les modifications entraînées par l'appel d'une méthode. Cela devrait permettre de détecter les variables modifiées, de savoir quelles fonctions ont été appelées à partir d'une autre, etc.

5.5. JavInspector version finale 2.0

- **exportation en xml**

Une analyse de classe pourra être adaptée en fichier xml afin de modéliser les informations dans un format spécifique d'affichage ou pour être utilisé par un programme externe.

- **menu d'aide**

Le logiciel final devra être muni d'une barre de menu assez conséquente composée surtout d'un menu d'aide.

- **Internationalisation**

L'équipe a proposée cette fonctionnalité et le client apprécierait que l'on puisse sélectionner la langue pour certains modules du logiciel dans la mesure du possible.

6. Contraintes non fonctionnelles

Après une étude comparative des précédentes versions de JavInspector, les clients et l'équipe ont décidé d'utiliser la version 1.0 comme base de travail.

Plusieurs versions de JavInspector orientées sur des qualités ou fonctionnalités vraiment différentes étaient disponibles avant le début de ce projet. Pour déterminer une base de travail pour la réalisation des nouvelles versions, l'équipe a réalisé une étude comparative. Après une longue série de tests, celle-ci ainsi que ses clients ont convenu de la version de départ, la 1.0. Ce n'est pas la dernière qui était disponible, mais sensiblement, celle dont les ajouts n'étaient pas au dépit des fonctionnalités précédentes. La version 1.1 ajoutait un module Web intéressant, mais se voyait inutilisable et très mal implémentée.

Pour éviter de réitérer les erreurs passées, une contrainte importante sera de respecter le séquencement des phases imposées afin de respecter les priorités du projet, et de n'aboutir à des versions supérieures que si celles-ci améliorent la version précédente. C'est bien entendu une entreprise assez logique mais assez mal respectée.

Les numérotations des versions seront significatives, 1.1, 1.2 et 1.3 correspondront à des révisions, corrections ou petites améliorations alors que 1.1, 2.1 et 3.1 correspondent à de réels ajouts tel qu'un module ou une fonction d'amélioration.
Il n'est pas possible de déterminer le numéro de la version finale, celui-ci sera choisi à l'expiration des délais de conception en fonction de l'avancement des travaux.

Du fait que le travail demandé est une continuité d'un projet, le langage de programmation est imposé. L'équipe utilisera ainsi la version j2sdk1.4.2 de java.

Lors de l'ajout de fonctionnalités ou de la représentation d'un arbre de classe ou d'héritage, la contrainte majeure qu'il faudra respecter est bien entendu la complexité. Le client attend une application simple et rapide, l'exploration ne devra prendre que quelques secondes avec sa représentation complète.

En enfin, nous ne pouvons oublier la fiabilité du produit, celui-ci devra fournir de bons résultats, et ceci dans tous les cas, et surtout dans les plus complexes. Il va de soit que l'utilisation d'un explorateur de classe se fera dans la plupart des cas sur des classes compliquées et volumineuses, le client attend d'excellents résultats.

III. GESTION DU TRAVAIL

1. Les imprévus

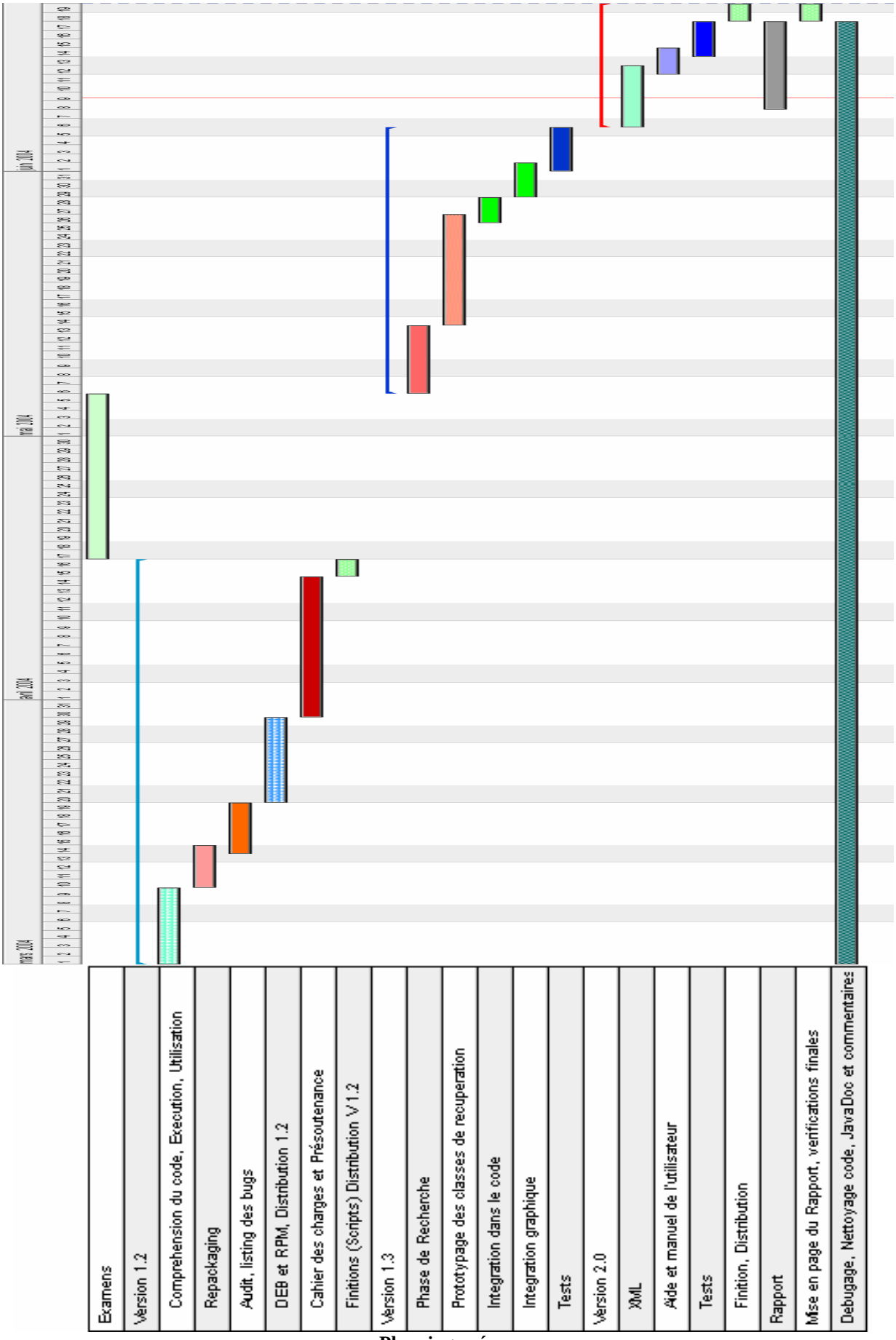
Le premier imprévu fut la réalisation de paquetages DEB et RPM. En effet, la création de paquetage DEB fut une étape très laborieuse, la plupart des outils disponibles sur des distributions non Debian nécessitant d'être administrateur nous empêchait de les utiliser.
Nous avons dû d'abord procéder à l'apprentissage de la création de paquetages Debian mais surtout à une recherche d'outils disponibles pour les distributions Mandrake ou RedHat mises à notre disposition. Cette recherche étant infructueuse ou non concluante, nous nous sommes finalement penchés vers nos encadrants, qui nous ont offert un espace sous une distribution Debian ou des outils adéquates comme DPKG ou ALIEN furent mis à notre disposition.
De plus pour la version 1.2, nous n'avions pas prévu l'élaboration d'autant de scripts de compilation ou exécution, offrant à l'utilisateur plus de dix façons de procéder. Ceci ne fut pas imposé mais apprécié par les encadrants et nous sommes certains qu'il en sera de même pour les utilisateurs.

Ensuite lors de l'audit des bogues de la version précédente de JavInspector, nous avons constaté des failles importantes dans le logiciel. En effet l'utilisateur ne pouvait pas inspecter une classe dont les classes de dépendances n'étaient pas trouvées par l'application. Il a donc fallu permettre l'inspection dans ce genre de situation en reprenant les chargeurs de classe de l'application et en demandant à l'utilisateur le chemin ou le jar contenant les classes concernées.
Cette résolution des bogues déjà présents dans l'application fut une étape longue qui s'est étendu tout au long du développement et qui suscita quelques problèmes bien entendu imprévisibles.

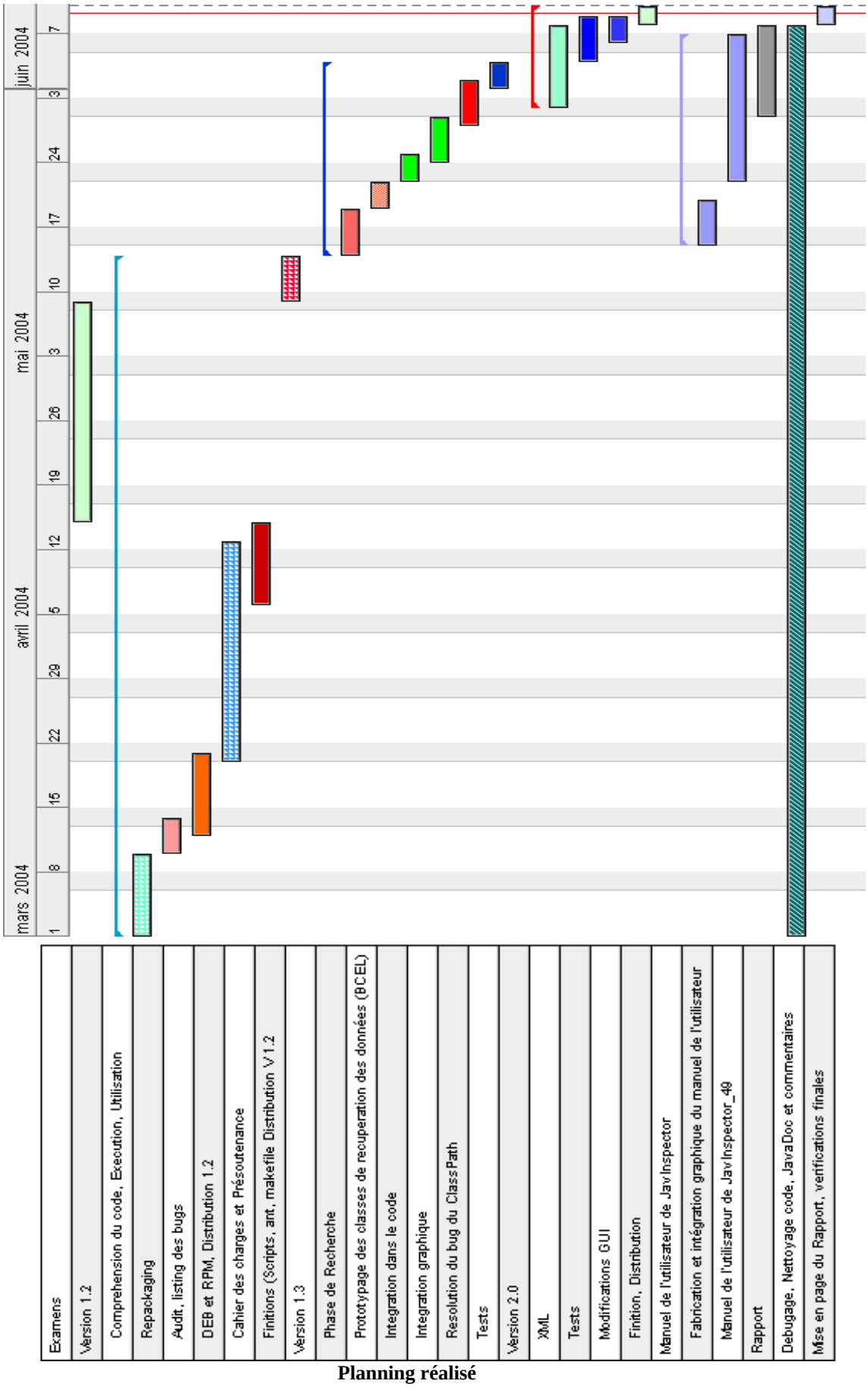
Par ailleurs, de nouveaux bogues dus à l'exploration du corps des méthodes nous ont contraint à adapter le chargeur de classe de notre application pour certains cas. Le chargement de classes ne se trouvent pas dans le chemin (classPath) posait un problème est à nécessité plus de 3 jours de débogage intensifs.

2. Planning prévu vs planning réalisé

Devant fournir trois versions du produit pendant la conception de ce projet, nous étions dans l'obligation de respecter très précisément le planning prévu.
Pour pouvoir fournir les différentes versions en respectant les délais au cours du développement, nous devions respecter les phases décrites dans le cahier des charges. En s'apercevant de nos mauvais jugements quant aux temps nécessaires à la réalisation de certaines étapes, plutôt que de prendre du retard et risquer de ne plus atteindre nos buts, nous avons préféré travailler beaucoup plus que ce qui était prévu. Les échelles de mesures étant en jour et non en heure, nous avons réussi à tenir les délais mais nous avons eu à fournir à peu près un tiers de travail (en heure par personne) en plus que ce qui était prévu.
Voici les deux schémas représentant le planning prévu et le planning réalisé :



Planning prévu



Planning réalisé

IV. Travail réalisé

Nous allons vous présenter dans cette partie les différentes phases de réalisation de ce projet.
Nous avons choisi de les diviser en quatre parties principales:

Pour la première, la réorganisation des fichiers, qui est une phase indispensable à la création d'une application professionnelle correctement structurée.
Nous présenterons ensuite, les nouvelles fonctionnalités avancées ajoutées au logiciel, puis la robustesse, caractéristique importante pour tout logiciel amené à être diffusé.
Enfin, nous regrouperons celles destinées au confort de l'utilisateur.

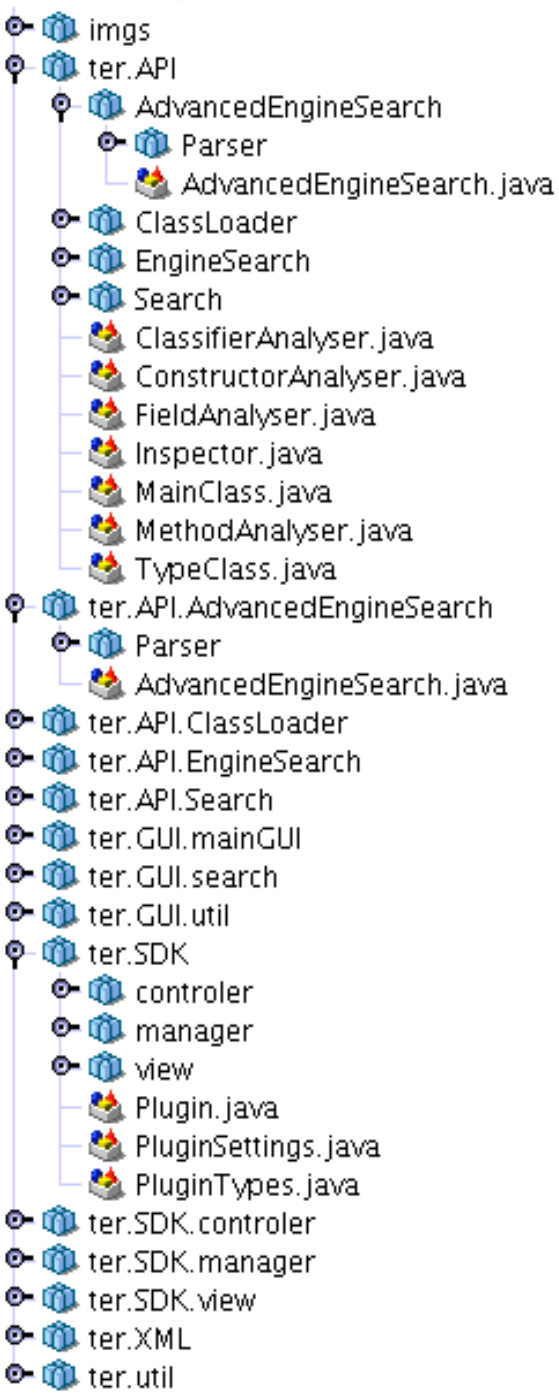
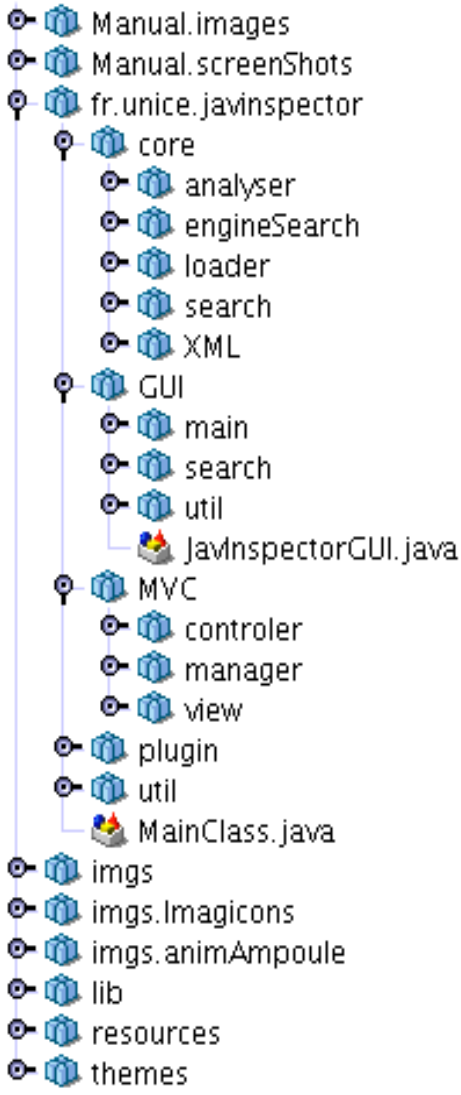
1. Réorganisation des fichiers (le « repackaging »)

JavInspector étant une application déjà existante au début de nos travaux, la première étape imposée par les encadrants fut la réorganisation de ses sources. En effet, le principal objectif de ce projet était de fournir une application professionnelle, propre et diffusable. La suite du déroulement de ce projet devant s'effectuer sur des bases solides, la finalité de cette étape fut la réalisation de la version 1.2 correctement organisée et empaquetée.

1.1. Réorganisation des sources :

Afin de faciliter la diffusion du logiciel, il a fallu réorganiser les sources et les binaires. Pour cela nous avons modifié l'ensemble des paquetages structurant les sources en utilisant des noms respectant les conventions de nommage. Nous avons également séparé les classes métiers des classes graphiques en améliorant le modèle MVC.

Voici la nouvelle organisation des fichiers que nous avons adopté :

Avant	Après
	

imgs
ter.API
AdvancedEngineSearch
Parser
AdvancedEngineSearch.java
ClassLoader
EngineSearch
Search
ClassifierAnalyser.java
ConstructorAnalyser.java
FieldAnalyser.java
Inspector.java
MainClass.java
MethodAnalyser.java
TypeClass.java
ter.API.AdvancedEngineSearch
Parser
AdvancedEngineSearch.java
ter.API.ClassLoader
ter.API.EngineSearch
ter.API.Search
ter.GUI.mainGUI
ter.GUI.search
ter.GUI.util
ter.SDK
controler
manager
view
Plugin.java
PluginSettings.java
PluginTypes.java
ter.SDK.controler
ter.SDK.manager
ter.SDK.view
ter.XML
ter.util

Manual.images
Manual.screenShots
fr.unice.javinspector
core
analyser
engineSearch
loader
search
XML
GUI
main
search
util
JavInspectorGUI.java
MVC
controler
manager
view
plugin
util
MainClass.java
imgs
imgs.Imagicons
imgs.animAmpoule
lib
resources
themes

1.2. Outils d'utilisation :

Dans l'optique de faciliter la diffusion du logiciel, nous avons défini de nombreux moyens permettant l'utilisation ou la modification de celui-ci.
Dans un premier temps, nous avons écrit un Makefile permettant de compiler, d'exécuter ou de générer la javadoc du logiciel. Deux cibles ont été créées permettant de générer le fichier jar ou d'exécuter l'application en utilisant ce fichier. Ayant remarqué que l'inspection d'énormes classes nécessitait quelquefois un surplus de mémoire dû à l'affichage graphique (OutOfMemory), deux cibles permettant d'exécuter l'application (à partir des binaires ou du fichier jar) en augmentant la quantité maximale de mémoire disponible pour la JVM à 100 Mo au lieu de 64Mo.
Finalement, un script ant ainsi que des scripts sh et bat ont été fournis pour permettre là encore la compilation et l'exécution possible sur de nombreux systèmes d'exploitation.

1.3. Les fournitures :

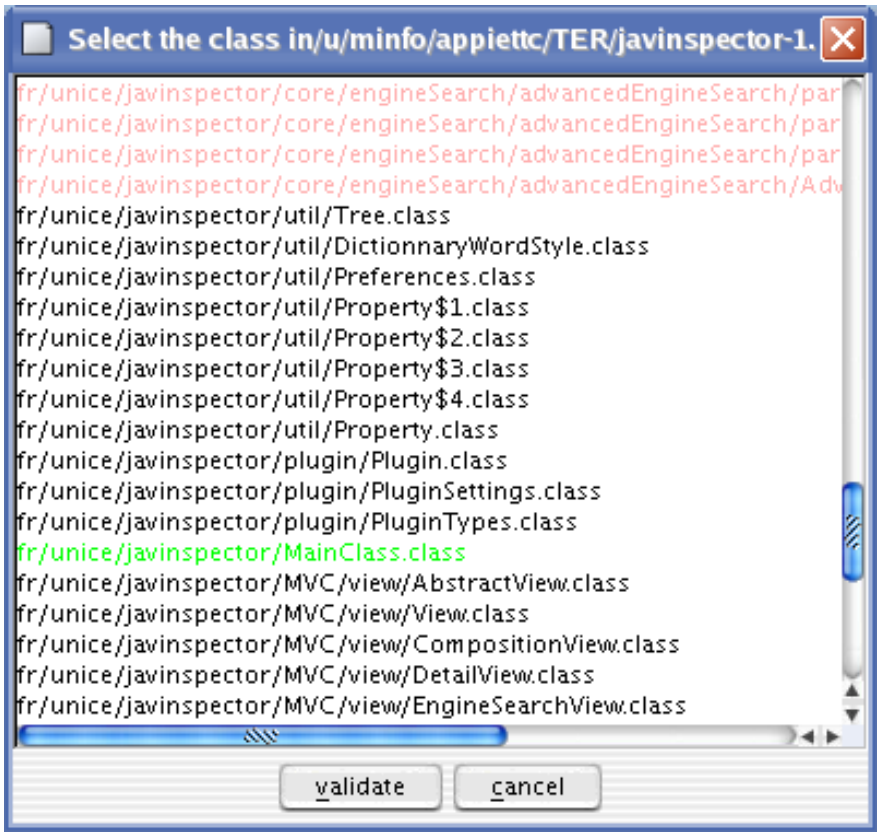
Dans le but de satisfaire l'ensemble des utilisateurs de JavInspector, travaillant sous différents systèmes d'exploitation ou même sous différentes distributions linux, nous avons décidé de distribuer notre application sous différents systèmes de paquetage ou différentes archives.
Vous pouvez vous référer à la section Fourniture de la partie « Travail Réalisé » pour obtenir plus d'informations.

2. Fonctionnalités avancées

2.1. La sélection de fichiers:

Lors de l'utilisation de l'application, la première tâche à effectuer par l'utilisateur est la sélection d'un fichier en vue d'une exploration. Celle-ci étant assez fastidieuse dans les précédentes versions, pour obtenir une meilleure ergonomie, nous avons choisi de l'améliorer.
Dans les précédents sélecteurs, il n'était pas prévu de visualiser les classes contenues dans un jar, nous étions donc obligés de connaître le nom complet de la classe ainsi que sa localisation dans la hiérarchie des paquetages pour ouvrir celle-ci.
Ainsi, nous avons réalisé un JFileChooser disposant de filtres de fichiers class et jar pour une visualisation plus claire, ainsi qu'une gestion des erreurs en cas de non sélection (ou celle-ci est redemandée).
Lors du choix d'une classe, celle-ci s'ouvre directement dans l'explorateur en créant un nouvel inspecteur de classe du nom de la classe principale sélectionnée. Notons que dans notre version, celle-ci affiche directement la vue détaillée de la classe choisie sans que l'on est besoin de l'indiquer.

Lors de la sélection d'un jar, nous avons choisi de développer un moyen convivial et pratique pour permettre la sélection de la classe à ouvrir. Le JarFileChooser prend en paramètre le nom d'un jar et permet de représenter dans une nouvelle fenêtre son contenu sous forme d'une JList. Nous avons choisi de faire étendre JDialog par notre nouveau composant afin d'avoir une fenêtre modale. On peut désirer annuler la sélection en fermant la fenêtre ou en pressant le bouton "Cancel" (Ctrl-c). Par contre, pour valider une sélection, un double-clique, une pression sur le bouton "Validate" (Ctrl-v) ou sur la touche entrée ouvrira l'inspecteur.
Afin de donner à l'utilisateur une vision encore plus agréable du jar, les classes sont représentées de différentes couleurs en fonction de leur nom et de leur paquetage. Nous avons essayé de répertorier les noms les plus utilisés dans des paquetages et classes : en vert toutes les classes dont le paquetage ou la classe contient une sous-chaîne de "main", en rose pour une sous-chaîne de "core", en bleu pour "test" et en gris pour "gui". Une classe sélectionnée s'affiche en rouge avec un fond de couleur cyan.
Une fois la sélection validée, la classe s'affiche comme expliqué précédemment dans l'inspecteur.
Le JFileChooser dispose d'une gestion d'erreur complète qui ouvrira en fonction des anomalies détaillées dans la section « **test »,** différentes boites de dialogues pour avertir l'utilisateur. Certaines fonctionnalités comme l'ouverture d'un fichier zip renommé en jar sont implémentées et sont possibles également.



2.2. Exploration du corps des méthodes

L'exploration du corps des méthodes dépasse les capacités offertes par la réflexivité de Java. Pour réaliser celle-ci, nous avons donc dû analyser la structure des classes java et en extraire les données qui nous intéressaient. Plusieurs possibilités se sont offertes à nous pour réaliser cette exploration et nous avons choisi d'utiliser une bibliothèque d'Apache BCEL. Kopi était également disponible et non moins puissante, mais étant de taille plus importante, nous avons préféré utiliser BCEL pour éviter une application « trop lourde ».
L'exploration s'est effectuée en plusieurs étapes :

- Récupération du code complet de la classe
- Prétraitement pour ne récupérer que les parties propres aux méthodes
- Analyse de celui-ci
- Traitement en fonction du type d'information récupéré

Nous allons vous présenter les différents traitements en détaillant les principales classes et leur façon de procéder, nous présenterons d'abord leur architecture puis le rôle de chacune.

2.2.1. Architecture des classes

Les classes **ConstructorAnalyser** et **MethodAnalyser** permettent comme leur nom l'indique d'analyser les constructeurs et les méthodes.
La classe ConstructorAnalyser possède les attributs de classe suivants:

- ***constructor*** de type *java.lang.reflect*
- ***listTypeClass*** qui est un *tableau de TypeClass* et qui répertorie l'ensemble des types des paramètres du constructeur
- ***modifier*** de type int qui est le modifier du constructeur c'est-à-dire sa portée
- ***classifierAnalyser*** de type *ClassifierAnalyser* qui répertorie toutes les informations sur la classe déclarant le constructeur

La classe MethodAnalyser possède les attributs de classe suivants:

- ***method*** de type *java.lang.reflect*
- ***modifier*** de type *int* qui est le modifier du constructeur c'est-à-dire sa portée
- ***listTypeClass*** qui est un *tableau de TypeClass* et qui répertorie l'ensemble des types des paramètres du constructeur
- ***returnType*** de type *TypeClass* qui est le type de retour de la méthode
- ***classesWhereThisMethodIsDefined*** qui est une *liste de toutes les classes* où cette méthode est déclarée
- ***classifierAnalyser*** de type *ClassifierAnalyser* qui répertorie toutes les informations sur la classe déclarant le constructeur

Afin de récupérer des informations concernant le corps d'une méthode ou d'un constructeur, nous avons écrit une classe ***BodyAnalyser*** et ses deux classes filles ***BodyMethodAnalyser*** et ***BodyConstructorAnalyser***. Les classes ConstructorAnalyser et MethodAnalyser admettront alors un nouvel attribut privé qui sont respectivement bodyConstructorAnalyser de type *BodyAnalyser* et bodyMethodAnalyser de type *BodyAnalyser*.

La classe **BodyAnalyser** permet de récolter des informations concernant le corps d'un constructeur ou d'une méthode.

Les attributs de cette classe sont les suivants:

- ***method*** qui est l'objet *Method* du paquetage org.apache.*bcel*.classfile de la librairie BCEL représentant un constructeur ou une méthode.
- ***modifiedFields*** qui est la *liste des attributs d'instance modifiés* dans le corps de la méthode ou du constructeur
- ***obtainedFields*** qui est la *liste des attributs d'instance accédés* dans le corps de la méthode ou du constructeur
- ***modifiedStaticFields*** qui est *la liste des variables de classe statiques modifiées* dans le corps de la méthode ou du constructeur
- ***obtainedStaticFields*** qui est la *liste des variables de classe statiques accédées* dans le corps de la méthode ou du constructeur
- ***caughtExceptions*** qui est la *liste des exceptions attrapées* dans le corps de la méthode ou du constructeur
- ***localVariables*** qui est la *liste des variables locales* du corps de la méthode ou du constructeur
- ***virtualMethods*** qui est la *liste des méthodes d'instance invoquées* dans le corps de la méthode ou du constructeur
- ***staticMethods*** qui est la *liste des méthodes statiques invoquées* dans le corps de la méthode ou du constructeur
- ***interfaceMethods*** qui est la *liste des méthodes d'interface invoquées* dans le corps de la méthode ou du constructeur
- ***specialMethods*** qui est *la liste des constructeurs invoqués* dans le corps de la méthode ou du constructeur

Voici quelques informations techniques concernant les différentes initialisations des variables et attributs.
Les informations concernant les attributs de classes sont initialisées par la procédure *initialiseFields(org.apache.generic.Instruction[], org.apache.generic.ConstantPoolGen).*
Les attributs privés concernant les invocations de méthodes et les exceptions attrapées sont initialisés respectivement par les procédures *initialiseMethodsCall (Instruction [], ConstantPoolGen)* et *initialiseCaughtExceptions (ConstantPoolGen)*.
La méthode *initialiseLocalVariables (String, ConstantPoolGen)* initialise la liste des variables locales au corps de la méthode ou du constructeur. Il faudra faire attention à cette liste qui sera **vide** si le fichier .class n'a pas été compilé avec l'option du debogueur de javac.

La classe **BodyConstructorAnalyser** permet d'analyser le corps d'un constructeur à partir d'une instance de ConstructorAnalyser passée en paramètre. Le constructeur de cette classe initialisera le champ privé *method,* hérité de le classe mère BodyAnalyser.
On peut là encore faire une précision, la librairie BCEL utilise la même classe (org.apache.bcel.classfile.Method) pour représenter une méthode ou un constructeur. Pour établir la correspondance entre une instance de la bibliothèque java (java.lang.reflect.Constructor) et une instance de la librairie BCEL (org.apache.bcel.classfile.Method) nous avons du écrire une méthode statique *getMethod(org.apache.bcel.classfile.JavaClass, java.lang.reflect.Constructor)* qui retourne une instance de la classe org.apache.bcel.classfile.Method correspondant au constructeur passé en paramètre. Cette méthode utilise la méthode d'instance

getSignature(java.lang.reflect.Constructor) qui retourne une chaîne de caractères qui représente la signature du constructeur et que l'on compare avec la signature de chaque méthode de la classe fournit par la méthode d'instance getSignature() de la classe org.apache.bcel.classfile.Method.

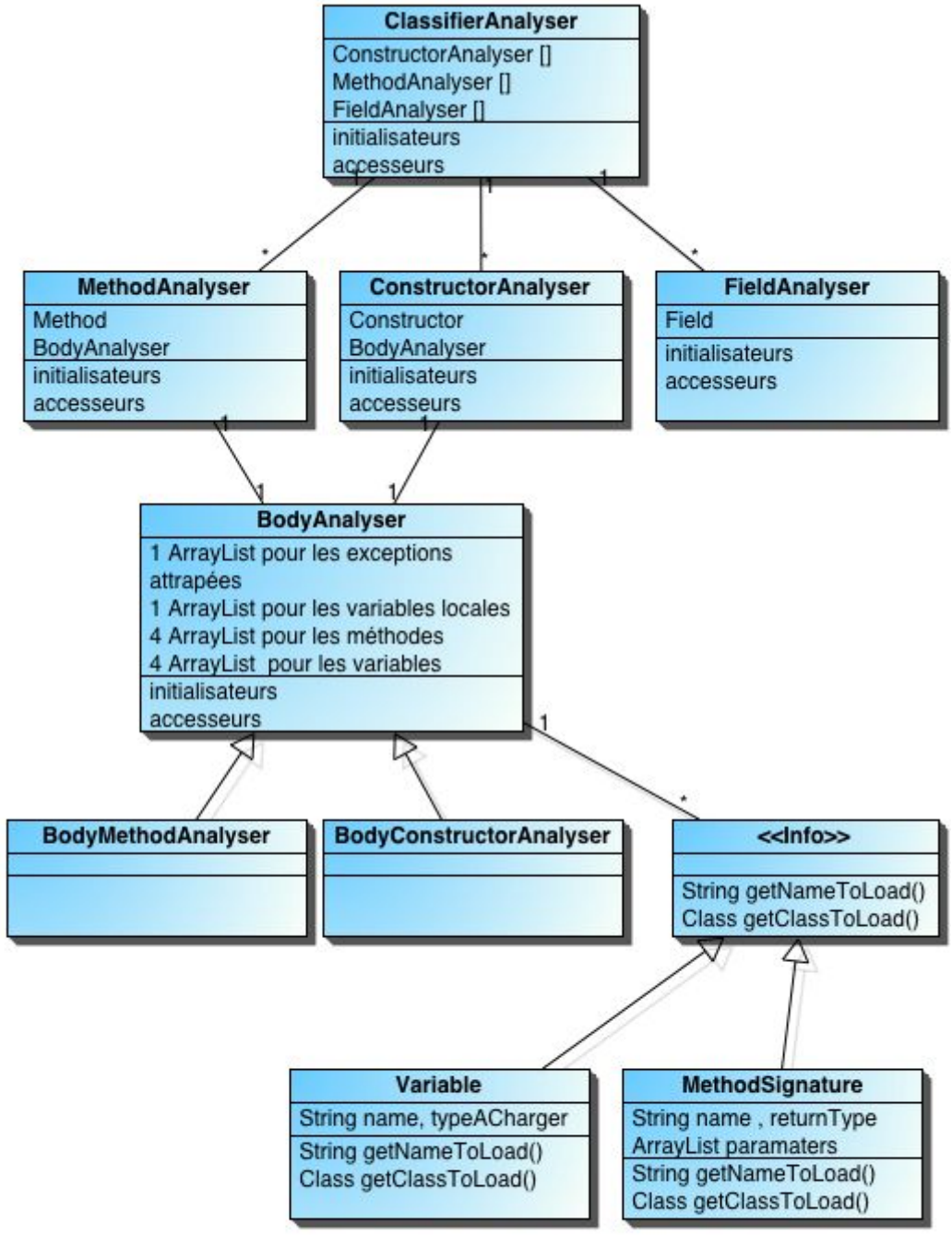
La classe **BodyMethodAnalyser** utilise le même schéma que la classe BodyConstructorAnalyser détaillée précédemment et permet d'analyser le corps d'une méthode. Le constructeur de cette classe prend en paramètre une instance de la classe java.lang.reflect.Method et initialise le champ privé *method* de type org.apache.bcel.classfile.Method. Cette classe possède également une méthode statique *getMethod (org.apache.bcel.classfile.JavaClass, java.lang.reflect.Method)* qui retourne une instance de la classe org.apache.bcel.classfile.Method correspondant à la méthode passée en paramètre. Celle-ci fait appel à la méthode d'instance getSignature(java.lang.reflect.Constructor) qui retourne une chaîne de caractères qui représente la signature de la méthode à analyser et que l'on compare avec la signature de chaque méthode de la classe fournit par la méthode d'instance getSignature() de la classe org.apache.bcel.classfile.Method.

Pour la représentation des différentes informations récoltées, nous avons choisi d'utiliser le système d'interface de Java permettant une plus grande souplesse d'utilisation des classes précédentes. Il s'agit de l'interface ***Info*** *et de ses deux classes l'implémentant* ***Variable*** *et* ***MethodSignature****.*

L'interface **Info** va permettre de représenter les informations concernant toutes les classes implémentant celle-ci. Dans notre cas, ce sera donc l'ensemble des variables, exceptions ou méthodes invoquées récoltées dans le corps d'une méthode ou d'un constructeur qui seront représentées.
Elle possède deux méthodes d'interface à savoir getClassToLoad() qui retourne la classe à charger pour cette information et getNameToLoad() qui retourne le nom de la classe à charger (util afin de ne pas avoir à charger la classe quand celle-ci est inconnu du chargeurs de classe).

La classe **Variable** permet de représenter une variable locale ou un attribut de classe privé, publique ou statique, ou enfin une exception. Elle implémente getClassToLoad() et getNameToLoad().

La classe **MethodSignature** permet de représenter les informations concernant les invocations de méthodes effectuées à l'intérieur d'un corps d'un constructeur ou d'une méthode. Elle implémente getClassToLoad() et getNameToLoad().



Les informations importantes quant à l'analyse du corps des méthodes et constructeurs

2.2.2. Intégration graphique de BCEL

L'intégration des classes finales pour la récupération des informations dans le corps des méthodes s'est faite en plusieurs étapes. La première phase fut le respect d'une contrainte que nous nous sommes imposés, une intégration en respectant le modèle MVC. Les dites classes de récupération effectuaient leurs tâches mais les résultats n'étaient toujours pas exploités.
La seconde phase constitua l'intégration dans les différentes vues en se souciant principalement de la facilité de lecture des résultats par l'utilisateur.

- **Graph detail view**

Nous avons beaucoup réfléchi à la façon de représenter les données extraites de la nouvelle exploration du corps des méthodes, et nous avons essayé de représenter celles-ci de la manière la plus visible possible sans pour autant surcharger la fenêtre d'affichage (ce qui aurait très vite gêné les utilisateurs).
Nous avons donc opté pour les JTree comme composant swing car ils permettent de façon assez simple et naturelle d'afficher ou de masquer l'ensemble des données représentées.
L'utilisateur dispose donc d'un arbre représentant les données dont la visualisation et l'utilisation est des plus simples car il lui suffit de cliquer sur un des éléments qui l'intéresse et le contenu apparaît immédiatement en dessous de celui-ci. Il pourra ouvrir dans le panneau d'héritage une analyse de la classe double-cliquée. Notons que les types primitifs sont affichés en rouge et en italic et que l'on ne peut ouvrir d'analyse de ceux-ci.

- **Composition view**

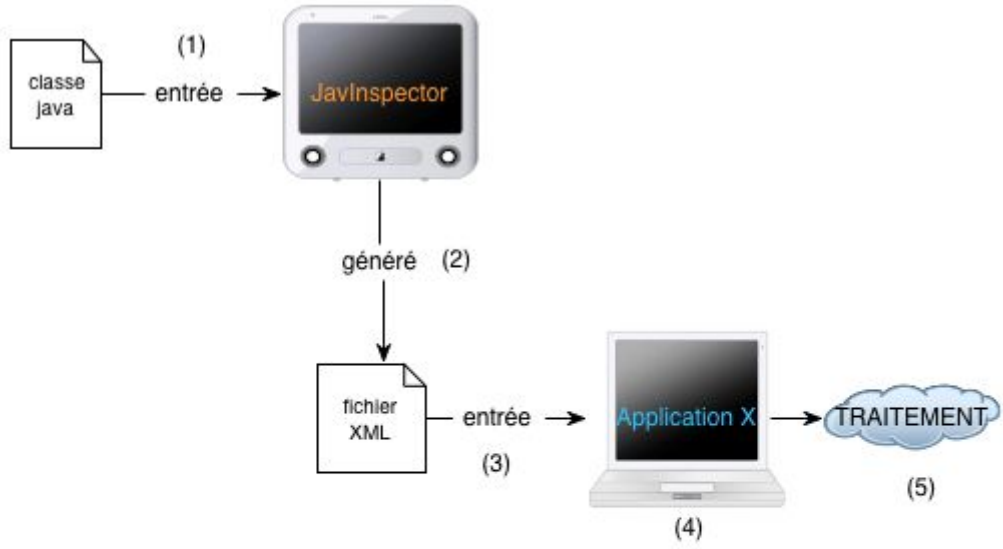
La vue de composition apporte une vision globale de toutes les dépendances de la classe analysée. Elle va permettre de trier les informations en fonction des trois axes : constructeurs, champs et méthodes. Par axe, les informations sont également triées en fonction de nombreux autres critères (exceptions, type de retour, ..). Nous offrons la possibilité de cacher ou d'afficher le contenu de chaque critère par le biais d'un bouton en forme de flèche. L'utilisateur peut lancer dans le panneau d'héritage une analyse de classe en double-cliquant sur le bouton correspondant.

Les boutons bleus représentent les types primitifs et donc impossible à ouvrir, les rouges représentent des classes pas encore chargées donc potentiellement inconnues. Il faut s'attendre à devoir spécifier un chemin ou un jar pour pouvoir ouvrir l'analyse. Toutes les autres couleurs peuvent être ouvert sans difficulté.

2.3. XML

2.3.1. L'exportation XML

Un des objectifs de JavInspector est de pouvoir enregistrer les informations relatives à une exploration dans le but de réutiliser celles-ci dans une autre application par exemple. Pour permettre une réutilisation automatique de ces données, nous avons choisi la technologie XML, ainsi, JavInspector permet la sauvegarde des informations récoltées lors de son inspection sous la forme d'un fichier XML. Ce fichier devant être traité par la suite, nous avons été dans l'obligation de fournir un Schema XML permettant de décrire le fichier que nous avons bien entendu tout deux validés par le validateur officiel du W3C.
Après avoir concerté nos encadrants, nous avons décidé de n'utiliser dans le document que des balises sans attributs, d'une part pour faciliter la génération du fichier XML (même si cela n'aurait pas été un problème) et d'autre part pour faciliter la récupération et la manipulation des données par l'utilisateur. Dans la même optique, nous avons décidé de ne pas insérer de balises vides dans le document en exécutant des tests préalables lors de la génération.



(1) JavInspector prend en entrée une classe java
(2) Javinspector génér le fichier XML associé à la classe
(3) Application X prend en entrée le fichier XML
(4) Application X connaît alors le contenu de la classe
(5) Application X peut alors effectuer son traitement

Intérêt d'exporter une analyse de classe en fichier XML

2.3.2. Principe de programmation :

Toutes les informations collectées doivent être sauvegardées dans le fichier XML. Ces informations sont accessibles à partir de l'instance ClassifierAnalyser propre à chaque classe inspectée. Nous avons donc récupéré toutes ces données polymorphes (représentées sous forme de liste, d'entier, de chaînes de caractères) et nous les avons transformées à l'aide de l'API (ou Application Programming Interface) DOM (ou Document Object Model). Nous avons utilisé cette API car tout d'abord elle est une norme du W3C donc un gage de fiabilité et ensuite nous avions besoin de stocker toutes les données et la programmation est plus aisée que SAX par exemple.

La classe fr.unice.javinspector.core.XML.ExportXML permet de réaliser ce travail. Nos choix ont été de morceler le travail en réalisant de petites fonctions retournant pour chaque information récoltée un ensemble de balises la représentant.

2.3.3. Le fichier XML et son Schema:

Nous allons maintenant décrire les balises les plus pertinentes que nous avons utilisées dans le fichier XML ainsi que leurs contraintes exprimées dans le Schema XML:

La balise <class>

- Description :

La balise <class> permet tout d'abord de représenter la classe inspectée c'est à dire la classe dont les informations sont synthétisées dans le fichier XML. C'est donc la balise racine de notre document.
Cette balise est formée des balises obligatoires <name> qui est le nom de la classe et <modifier> qui est le modifier de la classe. Nous avons décidé que le contenu de la balise <modifier> serait un entier pour faciliter l'utilisation des informations synthétisées dans le document XML(en JAVA le modifier d'une classe est un int). La balise <class> peut toutefois être composée des balises facultatives si les informations respectives existent :
<packageName>le nom du paquetage si la classe inspectée appartient à un paquetage
<superClasses>, les classes mères de la classes inspectée
<superInterfaces>, les interfaces qu'implémentent la classe inspectée
<internalClasses>, les classes internes
<fields>, les attributs de la classe inspectée
<constructors>, les constructeurs de la classe inspectée
<methods>, les méthodes de la classe inspectée

La balise <class> sert également à représenter une classe dans la liste des classes mères (balise <superClasses>), une interface dans la liste des interfaces qu'implémentent la classe (balise <superInterfaces>).

- Contraintes :

Cette balise est composée des balises obligatoires <name> et <modifier>. Les balises facultatives sont les balises <packageName>, <superClasses>, <superInterfaces> , <internalClasses> , <fields> , <constructors> et <methods> .

- Représentation :

```
<class>
    <name>...</name>
   <modifier>...</modifier>
   <packageName>...</packageName>
   <superClasses>...</superClasses>
   <superInterfaces>...</superInterfaces>
   <internalClasses>...</ internalClasses>
   <fields>...</fields>
   <constructors>...</constructors>
   <methods>...</methods>
</class>
```

Les balises <fields> et <field>

- Description :

La balise <fields> représente l'ensemble des attributs de la classe inspectée. Elle est formée d'un ensemble de balises <field> représentant un attribut de classe. Une balise <field> est composée des balises obligatoires :
<name>, le nom de l'attribut de classe
<modifier>, le modifier de l'attribut
<type>, le type de l'attribut
La balise <value> est facultative et représente la valeur de l'attribut s'il a été initialisé de façon statique.

- Contraintes :

La balise <fields> est composée obligatoirement d'au moins une balise <field>

- Représentation :

```
<fields>
   <field>
     <name>...</name>
       <modifier>...</modifier>
       <type>...</type>
       <value>...</value>
  </field>
   <field>
      <name>...</name>
      <modifier>…</modifier>
     <type>…</type>
  </field>
   …
</fields>
```

Les balises <constructors> et <constructor>

- Description :

La balise <constructors> référence les informations concernant l'ensemble des constructeurs de la classe. Elle est formée d'au moins une balise <constructor> représentant un constructeur. La balise <constructor> est constituée des balises obligatoires :

<modifier>, le modifier du constructeur
<params>, la liste des paramètres du constructeur si la méthode en possède
<thrownExceptions>, la liste des exceptions jetées par la méthode si elle en jette toutefois.
<body>, les informations récupérées sur le corps du constructeur si elles existent

- Contraintes :

La balise <constructors> possède au moins une balise <constructor>.

- Représentation :

```
<constructors>
        <constructor>
                <modidifer>…</modifier>
                <params>…</params>
                <body>…</body>
        <constructor>
        ...
<constructors>
```

Les balises <methods> et <method> :

- Description :

La balise <methods> représentent l'ensemble des méthodes de la classe inspectée. Elle est formée de balise <method> représentant une méthode.La balise <method> est composée des balises :
<name>, le nom de la méthode
<returnType>, le type de retour de la méthode
<modifier>, le modifier de la méthode
<params>, la liste des paramètres du constructeur si la méthode en possède
<thrownExceptions>, la liste des exceptions jetées par la méthode si elle en jette toutefois.
<body>, les informations récupérées sur le corps du constructeur si elles existent

- Contraintes :

La balise <methods> contient au moins une balise <method>.
La balise <method> possède les balises obligatoires <name>, <returnType> et <modifier> et les balises facultatives <params>, <thrownExceptions> et <body> si ces informations existent.

- Représentation

```
<methods>
    <method>
        <name>...</name>
        <returnType>...</returnType>
        <modifier>...</modifier>
```

```
    <params>...</params>
    <thrownExceptions>...</ thrownExceptions >
    <body>...</ body>
  <method>
  …
</methods>
```

La balise <body>

- Description

La balise <body> représente les informations récoltées sur le corps d'une méthode ou d'un constructeur.
Ces informations sont représentées par les balises suivantes :

Nom de la balise	Description
<obtainedFields>	la liste des attributs de classes accédés dans le corps de la méthode ou du constructeur
<modifiedFields>	la liste des attributs de classes modifiés dans le corps de la méthode ou du constructeur
<obtainedStaticFields>	la liste des attributs de classes statiques accédés dans le corps de la méthode ou du constructeur
<modifiedStaticFields>	la liste des attributs de classes modifiés statique dans le corps de la méthode ou du constructeur
<caughtExceptions>	la liste des exceptions attrapées dans le corps de la méthode ou du constructeur
<virtualMethods>	la liste des appels aux méthodes d'instance réalisés dans le corps de la méthode ou du constructeur
<specialMethods>	la liste des appels aux constructeurs réalisés dans le corps de la méthode ou du constructeur
<interfaceMethods>	la liste des appels aux méthodes d'interface réalisés dans le corps de la méthode ou du constructeur
<staticMethods>	la liste des appels aux méthodes statiques réalisés dans le corps de la méthode ou du constructeur
<localVariables>	la liste des variables locales utilisées dans le corps de la méthode

- Contraintes

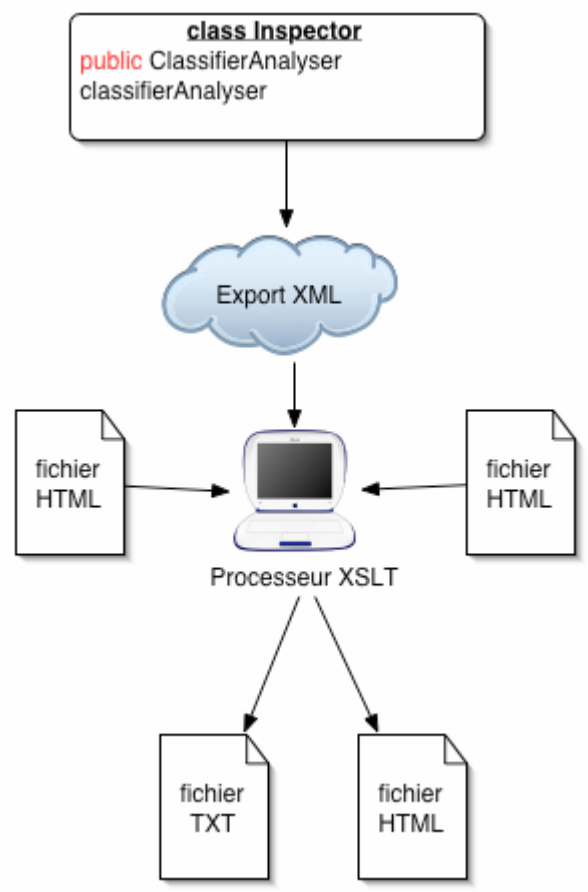
La balise <body> possède au moins une balise décrivant une information sur le corps de la méthode ou du constructeur.

- Représentation

```
<body>
    <obtainedFields>...</obtainedFields>
    <modifiedFields>...</ modifiedFields >
    <obtainedStaticFields>...</obtainedStaticFields>
    <modifiedStaticFields>...</ modifiedStaticFields >
    <caughtExceptions>...</ caughtExceptions >
    <virtualMethods>...</virtualMethods>
    <specialMethods>...</specialMethods>
    <interfaceMethods>...</interfaceMethods>
    <staticMethods>...</staticMethods>
    <localVariables>...</localVariables>
<body>
```

2.3.4. Les transformations TXT et HTML :

Ayant terminé l'exportation XML, en plus de celle-ci nous avons pris l'initiative d'intégrer à l'application les transformations des informations en fichier TXT et en fichier HTML.
Cette fonctionnalité utilise en fait l'exportation pour générer les données XML qu'elle ne sauvegarde pas dans un fichier mais sur lesquelles elle applique une feuille de style spécifique à la transformation HTML ou TXT désirée.



3. La robustesse

3.1. Débogage

La partie de débogage a été la plus longue et la plus difficile que nous ayons eue à réaliser. Ayant un nombre conséquent de classe à implémenter, lors de l'établissement des précédentes versions, les choix des structures de données ont privilégié la qualité du code. En effet, certaines classes étaient mal implémentées, de nombreux cas d'erreur n'étaient pas traités et les exceptions attrapées étaient souvent la plus générale et non la plus précise. Nous avons du reprendre l'ensemble des classes en y ajoutant des commentaires et en établissant la Javadoc que nous avons jugé indispensable à la réutilisation du code de JavInspector.
Cette tâche fut plus longue que ce que nous avions prévu.

Nous allons faire deux distinctions de bogues dans cette partie : les bogues des versions précédentes et les bogues que nous avons rajouté essentiellement dû aux librairies utilisées.

Parmi les bogues des versions précédentes, ceux qui méritent qu'on en parle concernent l'incapacité à ouvrir des classes sans en connaître la raison. Nous avons localisé cette anomalie comme étant la plus difficile à résoudre et nous avons du ainsi reprendre point par point le chargement de classe. Dans un premier temps, nous avons étudié de nombreux cas afin de répertorier les raisons de cette incapacité à ouvrir des analyses. Nous avons abouti à la conclusion les classes de chargement devaient être toutes revues voir entièrement reprogrammées et qu'il fallait modifier aussi le système de classpath pour permettre à l'utilisateur ajouter un chemin ou un jar.

Bien évidemment l'ajout de nombreuses fonctionnalités a engendré des anomalies.

La principale a été lors de l'intégration de l'obtention du corps des méthodes et constructeurs avec la librairie bcel: en effet l'erreur que nous avions commis a été pendant cette phase d'intégration de n'effectuer des tests qu'avec des classes situées dans le classpath. Ainsi, une fois intégré, il a fallu revoir le chargement de classe car nous nous étions rendu compte des problèmes liés à l'ouverture de classe en dehors du classpath avec bcel.

La première étape nous permettant de résoudre ce problème a été de rechercher et d'étudier le principe de fonctionnement du classpath pour la librairie bcel car c'était fondamentalement différent des chargeurs de classe de java. La difficulté de cette étape se résumait au simple fait que la javadoc de bcel était

vraiment mal conçue et parfois même la documentation concernant certaines fonction intéressantes étaient absentes.

De même, l'aide de l'Internet a été inutile car nous n'avons réussi à consulter aucune page traitant de ce problème. Il a fallu essayer de deviner le fonctionnement de certaines fonctions pour découvrir enfin comment résoudre notre problème.

La seconde étape fut l'intégration dans les chargeurs de classe de JavInspector. Il est clair que ce fut la tâche la plus laborieuse et fastidieuse de tout le projet. C'est à la suite d'étude de plusieurs centaines de fichiers de trace que l'on a pu progressivement résoudre les problèmes pour toutefois découvrir que l'on rajoutait à nouveau des problèmes.Le chargement de classe système ou l'affichage de l'analyse de la classe java.awt.Component étant tellement volumineuse, il n'y avait pas assez de mémoire java.

Nous avons donc du mettre à jour tous les scripts, (répertoire scripts, build.xml et Makefile) pour permettre à l'utilisateur d'ajouter un peu plus de mémoire java. Il est clair que ceci ne faisait que reporter le problème. Il a fallu donc trouver une solution pour permettre absolument l'ouverture de classe malgré l'incapacité à l'afficher entièrement.

Pour cela, nous avons eu l'ingénieuse idée de tout simplement couper l'analyse des méthodes de la classe trop volumineuse puisque c'est celle-ci qui provoque le dépassement de mémoire. Mais d'autres erreurs nous ont obligé à passer de nombreuses heures de déboguage intensif pour tenter de comprendre certains problèmes, mais cette-fois ci au niveau de la machine virtuelle de java (VerrifyError, …).

Cette phase de débogage s'achève bien évidemment sur un succès car nous avons réussi à résoudre ou à documenter tous les bogues que nous avions répertoriés au cours du développement de ce projet.

3.2. Tests

La qualité et la robustesse sont deux caractéristiques obligatoires pour un logiciel destiné à une diffusion Internet. Nous avons décidé d'établir de nombreuses séances de tests intensifs afin de pouvoir mettre à l'épreuve le logiciel. Pour cela, nous avons donc conçu et testé de nombreuses classes dans différentes configurations.

Propriétés	Résultat (Ok, Boites de dialogues (BdD))
Fichier .class vide (0 ko)	BdD : La classe est corrompue
Classe vide	RAS
Fichier autre que .class	BdD : Le fichier sélectionné n'est pas un fichier .class ou un jar.
Fichier autre que .jar	BdD : Le fichier sélectionné n'est pas un fichier .class ou un jar.
Fichier autre que .class renommé en .class	BdD : Le fichier jar ou le .class sélectionné est corrompu
Fichier autre que .jar renommé en .jar	BdD : Le fichier jar est corrompu
Fichier .class corrompu	BdD: Le fichier sélectionné semble corrompu
Archive jar corrompu	BdD : Le fichier sélectionné ne ressemble pas à un jar valide.
Archive jar vide	BdD : Le fichier sélectionné ne ressemble pas à un jar valide.
Archive zip	BdD : Le fichier sélectionné n'est pas un fichier .class ou un jar.
Archive jar : class dans le classpath	RAS
Archive jar : class pas dans le classpath	RAS
Archive jar : class bcel spéciale*	BdD : Problème de vérification de la classe par la JVM
Class bcel spéciale	RAS
Class normale dans le classpath	RAS
Class normale hors du classpath	RAS
Class graphique avec option mémoire JVM	RAS
Class graphique avec pas assez de mémoire	BdD : Vous ne disposez pas d'assez de mémoire pour explorer la classe complètement. Exploration partielle.
Archive Zip renommée en jar	RAS (ouverture comme s'il s'agissait d'un jar.

* Ces classes sont extrêmement rares

4. Convivialité, facilité, critères de qualité

4.1 L'Astuce du jour

Comme tout bon logiciel qui se respecte, la fenêtre d'astuce du jour s'imposait d'elle-même. Nous avons donc composé une classe dédiée que nous avons créer dans le but de la faire la plus générique possible.
L'astuce du jour ou "tip of the day" est composée de plusieurs panneaux. L'un de contrôle permettant de lire l'astuce précédente ou suivante ou de fermer la fenêtre (avec autant de raccourcis clavier). L'un de convivialité avec une animation d'ampoule qui clignote. A ce propos, nous avons également créé ce composant d'animation. Il suffit d'indiquer le chemin de l'image, le nombre d'image, quelques paramètres de taille et hauteur, ainsi qu'un booléen pour savoir si l'animation est un cycle pour avoir un JPanel contenant une animation. Le panneau central est le plus important, c'est celui qui affiche l'astuce courante.
Le fonctionnement de ce composant dans notre application est très simple. On obtient l'arbre DOM en analysant lexicalement un fichier xml de ressources "tips.xml" que nous avons créé. Ce fichier est très simple il contient autant de balises <tip> que d'astuces fournies. Une fois l'arbre DOM acquis, on récupère la liste de nœud de <tip>, puis on doit appliquer une fonction de formatage du texte afin d'éliminer les espaces inutiles ainsi que pour rajouter des sauts à la ligne quand on détecte que la ligne est remplie.
Le panneau du bas permet à l'utilisateur de spécifier s'il souhaite avoir l'astuce à chaque démarrage, il peut aussi l'indiquer dans le menu "Préférences".

Ce système fabriqué pour JavInspector est des plus réutilisables, en quelques minutes on peut l'intégrer à n'importe quelle application java et nous le réutiliserons probablement dans les prochains logiciels que nous serons menés à développer.

4.2. Préférences

Pour améliorer le confort et l'utilisation de l'application, nous avons décidé d'y ajouter une fonctionnalité permettant de mémoriser les préférences de l'utilisateur. Celui-ci peut choisir ou non l'affichage d'une « Astuce du jour au démarrage de l'application », celui-ci peut paramétrer les couleurs de l'affichage de l'exploration par exemple. Cette fonctionnalité utilise la classe Property de Java qui permet de stocker et de récupérer des données dans un fichier. La classe Préférence est assez simple :à chaque démarrage de l'application elle analyse un fichier pref.jvi situé à la racine de celle-ci et stocke l'ensemble des informations récoltées dans des variables propres. Toute information non trouvée ou fichier de configuration absent provoque le chargement des préférences par défaut. L'application ne s'occupera plus par la suite de ce fichier sauf en cas de modification des préférences.

4.3. Manuel de l'utilisateur

Pour un logiciel de l'envergure de JavInspector, un guide complet pour aiguiller l'utilisateur s'est avérer nécessaire.
Nous avons décidé de réaliser encore une fois un système vraiment réutilisable et interne au logiciel. L'idée était de faire le manuel en html afin de le rendre le plus convivial possible. Nous avons donc créé un visualisateur html capable d'afficher des pages utilisant des cadres et gérant les événements des liens des cadres. Pour le rendre le plus ergonomique possible, nous avons mis en place un système d'historique. Ainsi on peut noter la présence de boutons précédent et suivant rendus désactivés en fonction des sélections effectués par l'utilisateur durant l'exploration du manuel. De même un menu d'historique permet d'aller directement à l'une des dernières pages visitées. La taille de l'historique est paramétrable et nous enregistrons les quinze dernières pages.
L'historique est géré à partir de la classe Historique et la fenêtre du manuel est composée dans DocHelp.

Un point méritant une certaine attention est l'internationalisation de ce manuel. Pour la version rendue nous avons fais le guide en français mais aussi en anglais. Par un système astucieux, le menu langage permet de changer en temps réel la langue de l'affichage du manuel.
Nous avons mis en place un système assez puissant pour placer les pages html que nous avons créé tout au long du projet. Nous plaçons les pages dans un répertoire "Manual". 4 répertoires y sont disposés :

- "screenShots" : toutes les captures d'écran utilisées dans le guide.
- "images" : toutes les images utilisées dans le manuel.
- "fr": les pages en français.
- "en" : les pages en anglais.

On peut imaginer dans des futures versions la traduction des pages en italien par exemple : elle serait dans un répertoire "it". Notre système est vraiment puissant car il suffit de rajouter dans le tableau de DocHelp des langues parlées, les noms des langues parlées, et le visualisateur ira automatiquement chercher les pages dans le répertoire du nom des deux premières lettres de la langue.

Les sections détaillées dans le manuel concernent certains points méritant une aide. Ainsi nous expliquons comment ouvrir une analyse de classe en passant aussi bien par la sélection directe de ".class" ou indirecte de ".jar". Des parties sont consacrées exclusivement aux panneaux de détails, d'héritage et de composition. De même, nous expliquons comment utilisé le menu de recherche et de recherche avancée. Puis nous détaillons l'export xml puis la transformation du xml.
Pour finir cette partie sur le manuel de l'utilisateur, on peut préciser que la réutilisabilité a été mis en avant et qu'il est très facile d'inclure notre système dans n'importe quel logiciel.

V. RESULTAT

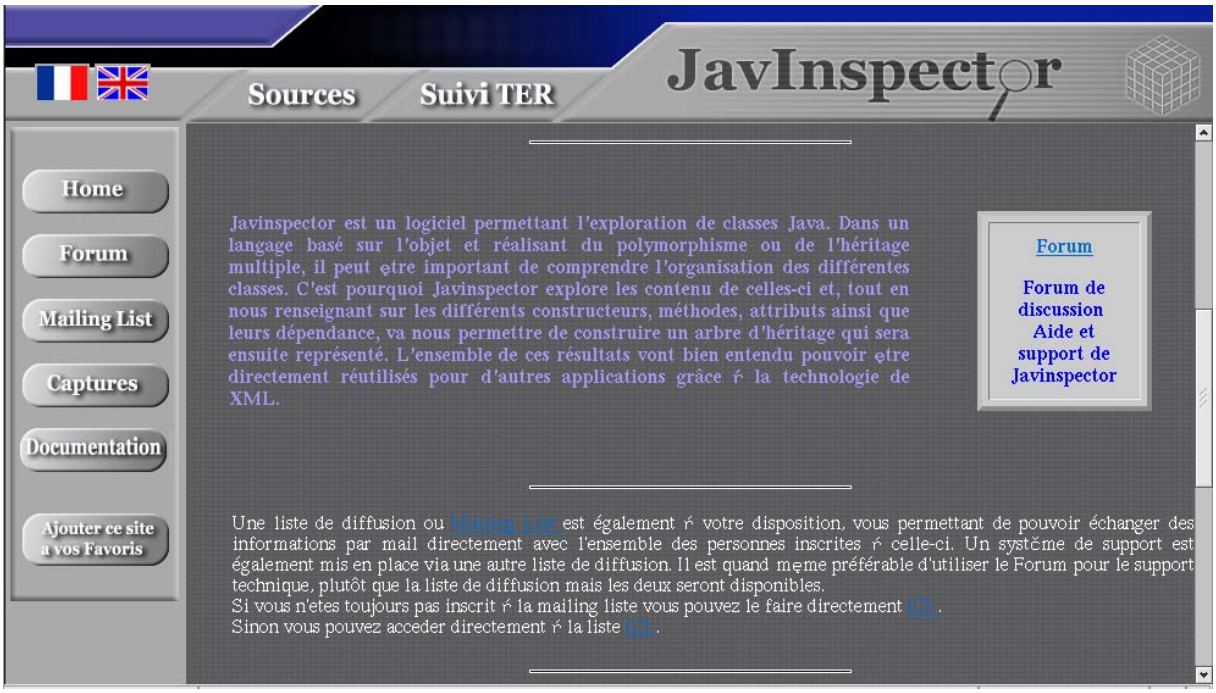
1. Fournitures

1.1. Site Internet (javinspector.sourceforge.net)

Dans le but de simplifier le travail des enseignants, de faciliter la documentation ou la demande d'aide des utilisateurs et par soucis de professionnalisme, nous avons choisi de développer un site Internet dédié à JavInspector. Ce site a été développé en php pour permettre une certaine dynamicité et surtout faciliter les liens avec les pages automatiquement créées par notre hébergeur «**sourceForge** » (pages de sources lors des dépôts)**.**

Celui-ci contient donc l'ensemble des sources, binaires et exécutables de toutes les versions que nous avons réalisé ainsi qu'un forum de discussion permettant d'obtenir de l'aide des utilisateurs et concepteurs inscrits ou enfin de soumettre des rapports de bogues qui permettrons l'amélioration future de l'application. Ce site à également été créé dans le but de centraliser les informations relatives au produit à savoir le manuel de l'utilisateur ou des captures d'écran, mais aussi celles relatives à la conceptions, comme l'état d'avancement des travaux ou un suivi quotidien de notre développement (lien vers le twiki de l'université de Nice-Sophia Antipolis).

L'hébergeur a été choisi pour de multiples raisons, les principales sont, la centralisation des sources d'une application sous licence LGPL sur un serveur unique, les services offerts comme un « shell » permettant la mise à jour des sources via ssh ou CVS outil puissant de conception permettant un meilleur rendement des programmeurs de l'application et enfin un espace disponible grandissant en fonction des mises à jour distribuées (donc considéré comme illimité).



1.2. Distribution

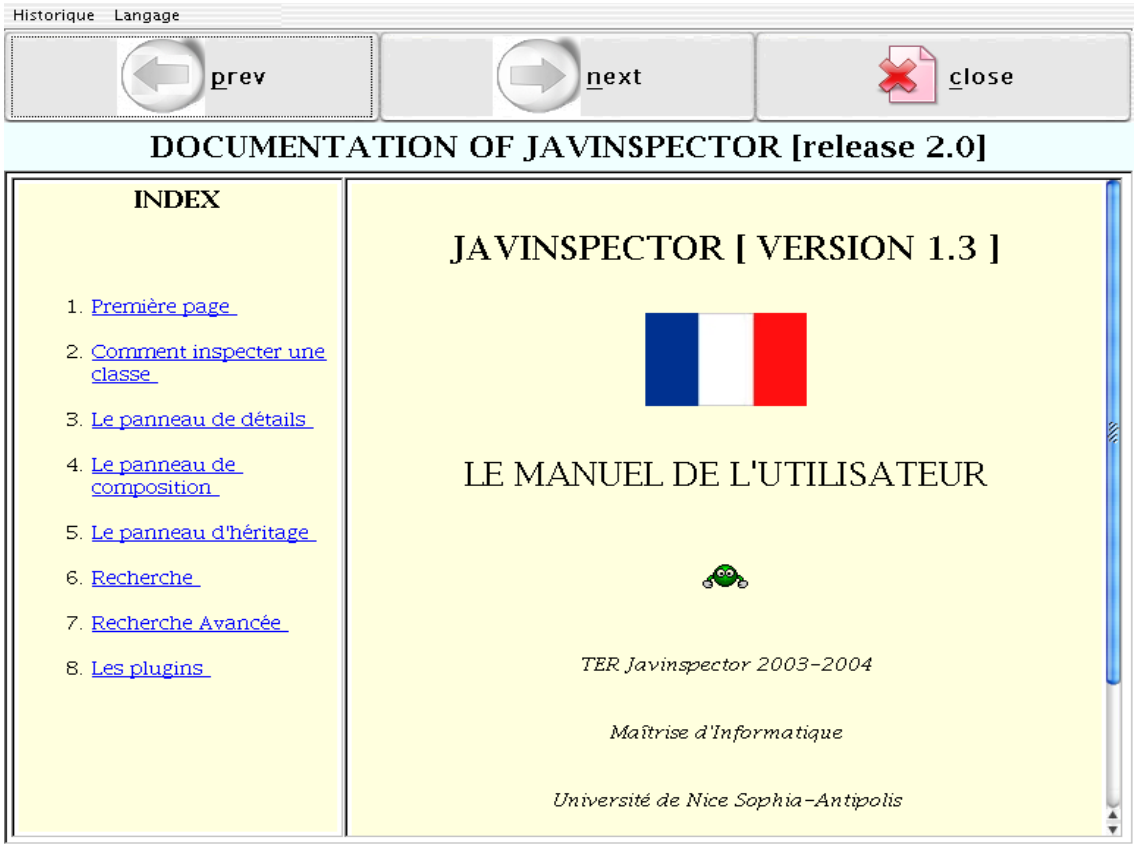
Etant un des objectifs de notre travail, nous avons porté une extrême attention à la façon dont nous allions distribuer l'application.
Pour faciliter la récupération et la décompression des archives de JavInspector et également pour contenter tous les utilisateurs, de nombreux formats d'empaquetage sont disponibles sur la page des sources et des binaires. En voici une énumération :

- tgz : Archive tar compressée avec l'outil gzip.
- tbz : Archive tar compressée avec l'outil bzip2
- zip : Archive et compression à l'aide de l'outil zip
- RPM : Paquetage RPM (RedHat, Mandrake, Suse …)
- DEB : Paquetage DEB (Debian …)

De plus, pour simplifier l'exécution ou la compilation des utilisateurs, un ensemble de scripts est disponible et offrent plus de dix méthodes d 'exécution ou de compilation. On distinguera un Makefile, des scripts sh et bat et un script ant.
Pour finir, d'après notre expérience, la simplicité aide souvent à la notoriété d'un produit. Pour cela, nous avons décidé d'ajouter à toutes les fournitures précédentes des exécutables disponibles pour les systèmes les plus courants.

2. Manuel de l'utilisateur

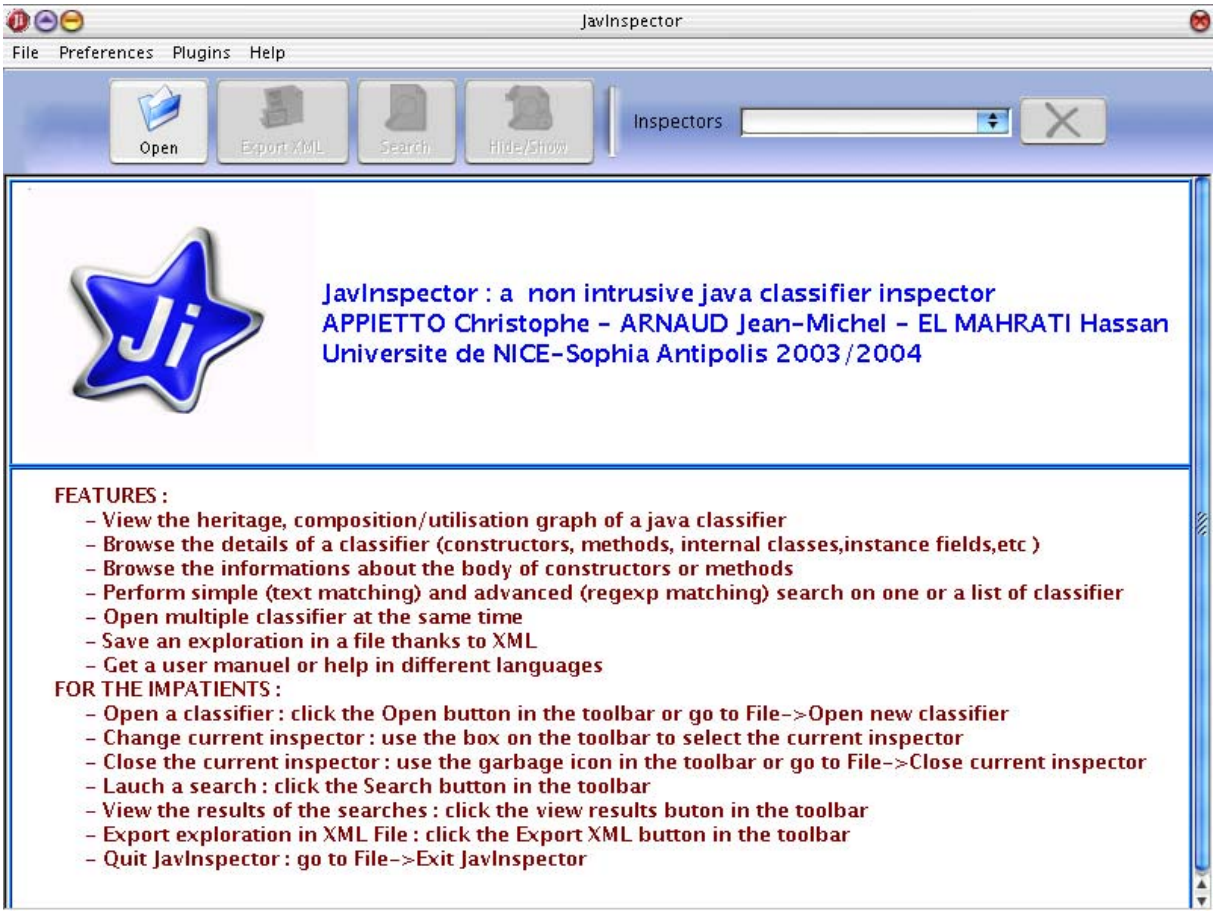
Afin de permettre d'obtenir de l'aide à propos des fonctionnalités disponibles de l'application, un manuel de l'utilisateur est également fourni. Nous avons choisi de réaliser celui-ci en HTML, ce qui permet à tout utilisateur de naviguer sur les pages de façon assez naturelle. Pour une meilleure convivialité, nous avons intégré celui-ci directement dans l'application en implémentant un navigateur interne. Celui-ci dispose des fonctionnalités de base requise par une application professionnelle, comme un système d'historique des pages visitées ou un système de boutons **précédent** et **suivant** permettant une navigation assez facile.
Enfin, tout comme « les astuces du jour » disponibles dans l'application, ce manuel à été rédigé dans différentes langues pour permettre une internationalisation du produit.



3. L'Application

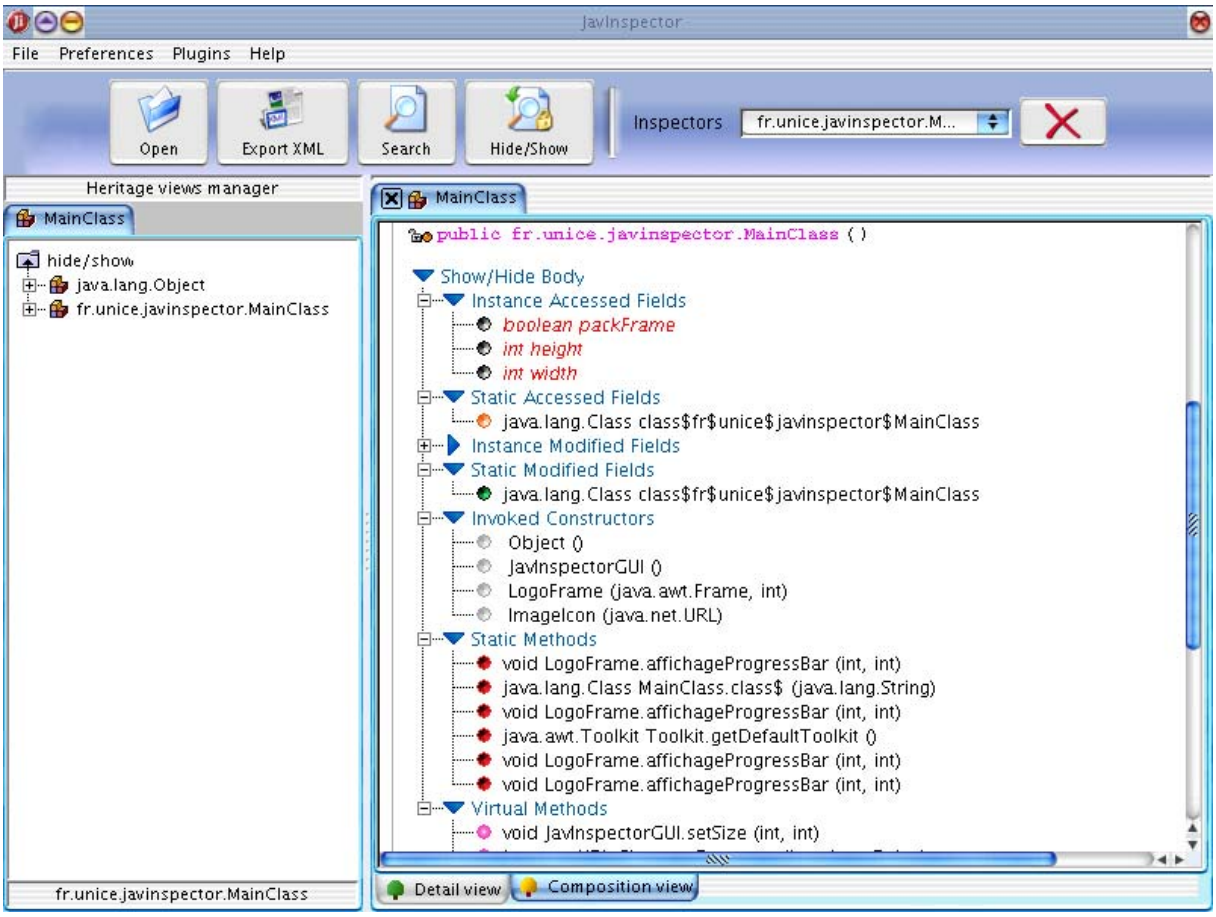
3.1. Interface

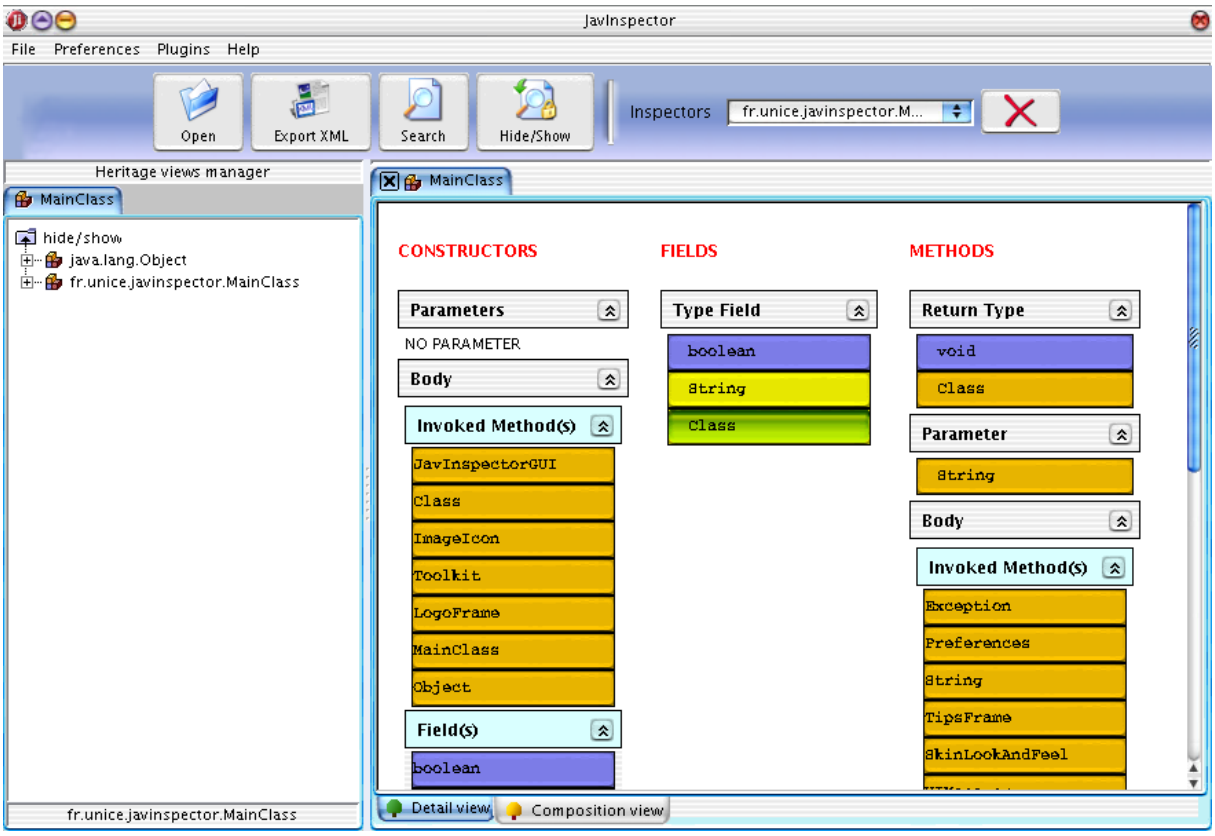
Ayant atteint les buts que nous nous étions fixés, nous avons décidé lors de la réalisation de la version 2.0 de s'attarder d'avantage sur l'aspect graphique de l'application. Il était prévu dès le début de notre projet de reprendre quelques parties en vue d'une simplification de la visualisation des données, mais nous ne pensions pas avoir le temps de modifier le côté convivial, du moins pas d'une telle façon. Les thèmes, couleurs et images de l'application (logo de chargement par exemple) ont été complètement réalisés avec des logiciels de dessin. Voici une petite présentation en image.



3.2. L'Exploration

Comme ceci a pu être énoncé précédemment, les améliorations les plus importantes furent celles relatives à la représentation de l'information. Nous avons voulu que celle-ci soit la plus agréable tout en étant la plus complète possible.


JavInspector
File Preferences Plugins Help
Open
Export XML
Search
Hide/Show
Inspectors
fr.unice.javinspector.M...
Heritage views manager
MainClass
hide/show
java.lang.Object
fr.unice.javinspector.MainClass
public fr.unice.javinspector.MainClass ( )
Show/Hide Body
Instance Accessed Fields
boolean packFrame
int height
int width
Static Accessed Fields
java.lang.Class classfrunice$javinspector$MainClass
Instance Modified Fields
Static Modified Fields
java.lang.Class classfrunice$javinspector$MainClass
Invoked Constructors
Object ()
JavInspectorGUI ()
LogoFrame (java.awt.Frame, int)
ImageIcon (java.net.URL)
Static Methods
void LogoFrame.affichageProgressBar (int, int)
java.lang.Class MainClass.class$ (java.lang.String)
void LogoFrame.affichageProgressBar (int, int)
java.awt.Toolkit Toolkit.getDefaultToolkit ()
void LogoFrame.affichageProgressBar (int, int)
void LogoFrame.affichageProgressBar (int, int)
Virtual Methods
void JavInspectorGUI.setSize (int, int)
fr.unice.javinspector.MainClass
Detail view
Composition view


JavInspector
File Preferences Plugins Help
Open
Export XML
Search
Hide/Show
Inspectors
fr.unice.javinspector.M...
Heritage views manager
MainClass
hide/show
java.lang.Object
fr.unice.javinspector.MainClass
CONSTRUCTORS
FIELDS
METHODS
Parameters
NO PARAMETER
Body
Invoked Method(s)
JavInspectorGUI
Class
ImageIcon
Toolkit
LogoFrame
MainClass
Object
Field(s)
boolean
Type Field
boolean
String
Class
Return Type
void
Class
Parameter
String
Body
Invoked Method(s)
Exception
Preferences
String
TipsFrame
SkinLookAndFeel
fr.unice.javinspector.MainClass
Detail view
Composition view

VI. BILAN

S'étant fixé des objectifs forts ambitieux, avoir atteint tous nos buts à la fin de la réalisation de ce projet fut pour nous une grande satisfaction.

La réalisation de ce projet a été très bénéfique pour de multiples raisons, qui peuvent être considérées comme l'aboutissement de notre cursus universitaire.
La principale est sans aucun doute le travail en équipe, où la planification ainsi que l'organisation, ont été les bases de notre aboutissement. Nous avons véritablement pris conscience de l'importance de la première phase, la conception du cahier des charges, qui détermine en général le déroulement de tout le projet.

Ensuite, l'utilisation de java et XML nous a permis d'acquérir une certaine expérience, dans deux domaines indispensables de l'informatique qui nous passionnent tous les trois depuis déjà quelques années.
De plus, pour notre tâche constituant à reprendre une application existante, la compréhension du code ainsi que la structure des classes fut une phase non négligeable que nous avons jugé intéressante, se rapprochant d'un travail professionnel d'entreprise.

Enfin, la partie où nous étions les moins initiés et que nous appréhendions le plus, la recherche de moyens permettant la réalisation de notre travail, fut en fait celle que nous avons le plus apprécié par la façon dont celle-ci s'est déroulée et par ce que nous en avons appris.

Ce travail fut pour nous une grande contribution à notre formation, l'intérêt que nous lui avons porté et la rigueur que nous avons tenté de conserver tout au long, nous à permis d'atteindre nos objectifs et d'en être satisfaits.

BIBLIOGRAPHIE

1. Livres

- *Java in a Nutshell 3ème edition* David Flanagan ed.O'REILLY 775 pages 2000
- *PhP & MySQL,* Luke Weilling, Laura Thomson ed.CampusPress 924 pages 2003
- *The XML HandBook End ed*, Charles F.Goldfarb ed PaperBack 639 pages 1999
- *Comprendre XSLT,* Bernard Amann et Philippe Rigaux ed O'REILLY 517 pages 2002

2. Sites Internet

- *http://java.sun.com*
- *http://www.sourceforge.net*
- *http://javinspector.sourceforge.net*
- *http://www.crescenzo.nom.fr*
- *http://jakarta.apache.org/bcel*
- *http://www.developpez.com*
- *http://www-106.ibm.com/developerworks/java/*

3. Cours

- Cours de Java de Richard Grin à l'université de Nice-Sophia Antopolis, Licence Informatique.
- Cours de Java de Michel Buffa à l'université de Nice-Sophia Antipolis, Licence et Maîtrise Informatique.
- Cours d'Administration système de Olivier Dalle à l'université de Nice-Sophia Antipolis, Maîtrise Informatique.
- Cours de programmation XML de Philippe Poulard à l'université de Nice-Sophia Antipolis, Maitrise Informatique.
- Cours de Compilation (Kopi) de thomas Graff à l'université de Nice-Sophia Antipolis, Maîtrise d'Informatique.

LIBRAIRIES UTILISEES

1. Présentations

1.1. La librairie BCEL

La librairie BCEL offre la possibilité d'analyser, de créer et de modifier des fichiers **.class** java. Elle permet de récupérer de précieuses informations au sujet d'une classe donnée : les méthodes, les champs d'instance, les constructeurs, les instructions en particulier. BCEL est utilisé avec succès dans de nombreux projets tels les compilateurs, les optimisateurs, les générateurs de code et les outils d'analyse. Ant, le célèbre outil pour la construction d'application java utilise aussi la librairie BCEL.
Dans le cadre de JavInspector, BCEL nous permet de récupérer les instructions concernant le corps d'une méthode ou d'un constructeur et de récupérer des informations comme l'accès ou la modification des champs d'instance, les appels de méthodes les exceptions attrapées ou les variables locales.

1.2. La librairie SkinLF

SkinLF est un Look And Feel pour Java qui supporte les thèmes GTK et KDE. Il permet de modifier le Look And Feel d'une application et dispose de nombreux thèmes offrant un aspect graphique agréable à l'utilisateur de l'application.
La librairie que nous utilisons est une version exclusivement modifiée pour JavInspector.

2. Licences

Ces deux librairies sont sous licence LGPL, il s'agit de la licence la plus libre disponible. En effet celle-ci permet d'utiliser ces librairies, de les modifier, de les redistribuer même à des fin commerciales.
Vous pouvez consulter chacune d'elle dans les pages d'annexes.

REMERCIEMENTS

Nous tenons à remercier un certain nombre de personnes qui nous ont été d'une aide précieuse lors du développement de ce projet sans qui nous n'aurions pu atteindre nos objectifs.

Nous remercions :

Nos encadrants **Pierre Crescenzo** et **Philippe Lahire** pour leur disponibilité, leurs conseils et leur aide en nous fournissant un environnement de travail sous la distribution Debian, des outils de développement , …

Les auteurs des librairies utilisées à savoir :

- **Markus Dahm, Jason van Zyl et Enver Haase** : pour bcel version 5.1
- **Frédéric Lavigne** : pour skinlf (version exclusivement modifiée pour JavInspector après notre rencontre à la réunion JAOO au Majestic Barrière)

Nos prédécesseurs, pour leur travail.

La communauté Java pour leurs précieuses réponses apportées à nos questions sur de nombreux forums.

Nous tenons également à remercier **l'ensemble des membres du département informatique de l'université de Nice-Sophia Antipolis**, pour les moyens et locaux qui furent mis à notre disposition ainsi que pour la formation que nous avons suivi.

ANNEXE

LICENCES

BCEL:

SKINLF :