DEPARTEMENT INFORMATIQUE - IUT2 DE GRENOBLE

Année Universitaire 2007/2008
MEMOIRE DE STAGE (REF :JMMC-TRA-2210-0001)

REALISATION DE BRIQUES LOGICIELLES JAVA D'HOMOGENEISATION DE L'APPARENCE ET DE L'UTILISATION DE SUITES LOGICIELLES SOUS LINUX, MAC OS X ET WINDOWS

Laboratoire d'AstrOphysique de Grenoble
(du 07 avril au 27 juin 2008)

Présenté par **Colucci Brice**
PROMOTION DE 2EME ANNEE

IUT : M. PASCAL BERTOLINO
IUT : M. ERIC FONTENAS
JMMC : M. SYLVAIN LAFRASSE

Ce document est accessible sous
http://www.jmmc.fr/doc/approved/JMMC-TRA-2210-0001.pdf

Remerciements

Je tiens à remercier M. Sylvain Lafrasse de m'avoir encadré durant mon stage ainsi que M. Guillaume Mella pour ses compétences techniques qui m'ont été d'un grand secours, sans oublier Mlle Evelyne Altariba pour sa relecture attentive de ce manuscrit.

Merci à M. Pascal Bertolino pour ses conseils concernant ce rapport ainsi qu'à M. Eric Fontenas pour sa présence à ma présentation orale.

Pour finir, je remercie l'ensemble du personnel de l'observatoire qui a été très accueillant.

" Les ordinateurs sont comme les dieux de l'Ancien Testament : avec beaucoup de règles, et sans pitié" - Joseph Campbell

Sommaire

1 Introduction **5**

2 Présentation du stage **6**
2.1 Le LAOG 6
2.2 Le JMMC 7
2.3 Sujet de stage 7
2.3.1 Contexte 7
2.3.2 Définition 8
2.4 Environnement de développement 8
2.5 Méthodologie de travail 9
2.5.1 Outils 9
2.5.2 Organisation 11
2.6 Documentation 11
2.6.1 Développement 11
2.6.2 Utilisateur 12

3 Travail effectué **13**
3.1 Fenêtre d'à-propos/écran de démarrage 13
3.1.1 Analyseur syntaxique XML 16
3.2 Gestion des journaux d'exécution 17
3.2.1 LogGui 19
3.3 Gestion des retours d'erreurs et commentaires 19
3.4 Génération/visualisation de l'aide utilisateur 22
3.4.1 JavaHelp 23
3.4.2 JHelpDev 24
3.5 L'interface "ligne de commande" 25
3.5.1 Analyse des options 26
3.6 Uniformisation des menus/raccourcis claviers 27
3.6.1 Formalisme XML concernant les menus 27
3.7 Mise en oeuvre des fonctionnalités réalisées 28
3.7.1 AppFramework 28
3.7.2 Cycle de vie 29
3.7.3 Application de tests 32

4 Bilan du stage **34**
4.1 Expérience 34
4.2 Compétences acquises 34
4.3 Evolution professionnelle 35

A Organigramme du JMMC **36**

B Code de l'application de tests **37**

Table des figures

1 Patron de conception MVC 10
2 Fenêtre de démarrage 14
3 Fenêtre d'à-propos 15
4 Modèle du fonctionnement de castor 17
5 Capture d'écran LogGui 20
6 Fenêtre de rapports d'erreurs 22
7 Fenêtre d'aide utilisateur 25
8 Schéma du processus de génération de l'aide utilisateur 26
9 Architecture du JSR-296 29
10 Cycle de vie d'une application 31

1 Introduction

Dans le cadre de mes études en IUT Informatique, j'ai effectué en fin de deuxième année, un stage de 3 mois au Laboratoire d'AstrOphysique de Grenoble (LAOG). J'ai évolué au sein du centre de réalisation du Jean-Marie Mariotti Center (JMMC) composé de trois ingénieurs, d'un directeur scientifique ainsi qu'un directeur général. Ce document rapporte le travail effectué au cours de mon stage ainsi que les résultats obtenus.

Mon sujet de stage était le suivant : "Réalisation de briques logicielles JAVA[1] d'homogénéisation de l'apparence et de l'utilisation de suites logicielles sous Linux, Mac OS X et Windows".

Au début, très orienté programmation WEB, j'ai très vite apprécié la programmation Java dans le cadre des projets à l'IUT2. Ce stage était pour moi une occasion de développer mes compétences en Java mais aussi une expérience qui me permettrait de préciser mes préférences quant à ma vie professionnelle.

Dans une première partie, je présenterai rapidement le laboratoire et la cellule informatique, mon sujet de stage ainsi que l'environnement et la méthodologie de travail que j'ai utilisés. Ensuite, dans la deuxième partie, j'exposerai les solutions retenues et les résultats obtenus. Pour terminer, je tirerai les enseignements de ce stage.

[1] Java est à la fois un langage de programmation informatique orienté objet et un environnement d'exécution informatique portable créé par James Gosling et Patrick Naughton employés de Sun Microsystemss avec le soutien de Bill Joy (cofondateur de Sun Microsystemss en 1982), présenté officiellement le 23 mai 1995 au SunWorld. Le langage Java a la particularité principale que les logiciels écrits avec ce dernier sont très facilement portables sur plusieurs systèmes d'exploitation tels que Unix, Microsoft Windows, Mac OS X ou Linux avec peu ou pas de modifications... C'est la plate-forme qui garantit la portabilité des applications développées en Java.

2 Présentation du stage

2.1 Le LAOG

Le LAOG, est une Unité Mixte de Recherche (UMR) de l'Université Joseph Fourier (UJF) et du Centre National pour la Recherche Scientifique (CNRS). Créé en 1979, c'est un pôle de recherche diffusant son savoir. Ses activités sont très variées : Observations astronomiques, modélisations numériques inter/intra stellaires, recherches théoriques et enfin développements technologiques.

Les recherches menées au LAOG sont dédiées à la compréhension de notre univers, elles couvrent un large domaine allant des étoiles proches aux galaxies lointaines. Ces recherches contribuent aux réflexions sur la place de l'homme dans son environnement.

Le LAOG collabore avec des industries et d'autres centres de recherche nationaux et internationaux. Le personnel se compose d'une cinquantaine de chercheurs, d'une trentaine d'ingénieurs, techniciens et administratifs ainsi qu'une vingtaine de post-doctorants, stagiaires et thésards, et parfois quelques visiteurs.

Pour plus d'informations, veuillez vous référer au lien n-1 de la webographie à la fin de l'annexe.

Adresse postale : Laboratoire d'Astrophysique
Observatoire de Grenoble
BP 53
F-38041 GRENOBLE Cédex 9
(France)

Adresse géographique : Laboratoire d'Astrophysique
Observatoire de Grenoble
414, Rue de la Piscine
Domaine Universitaire
38400 Saint-Martin d'Hères

Téléphone : 04.76.51.47.88
+33(0)4.76.51.47.88

Fax : 04.76.44.88.21
+33(0)4.76.44.88.21

Courrier électronique : {Prenom}.{Nom}@obs.ujf-grenoble.fr

2.2 Le JMMC

Le JMMC (cf. organigramme annexe A) est un groupement de laboratoires dont fait partie le LAOG. Il est chargé du développement d'instruments logiciels concernant principalement la préparation d'observations interférométriques[2] ainsi que la réduction de données provenant de ces observations. Il travaille en collaboration avec une dizaine de laboratoires français disposant d'une expertise en interférométrie optique.

Pour plus d'informations, veuillez vous référer au lien n-2 de la webographie à la fin de l'annexe.

2.3 Sujet de stage

2.3.1 Contexte

Dans le cadre du développement des logiciels Java du JMMC (SearchCal et ModelFitting, par exemple) et d'une demande de développement croissante, il est

[2]L'interférométrie est une méthode de mesure qui exploite les interférences intervenant entre plusieurs ondes cohérentes entre elles.

apparu nécessaire de factoriser les parties de développement communes à toute application graphique. Il y avait de plus le besoin d'affirmer la notion de famille d'applications du JMMC. Cette notion de famille est importante car elle amène l'utilisateur à se familiariser à toutes nos applications avec une seule d'entre elle !

C'est dans cette optique qu'un groupement de fonctionnalités a été créé, le module JMCS : Une bibliothèque ayant pour but de simplifier et de généraliser la création d'applications graphiques Java. Un cahier des charges, pointant les différents aspects à uniformiser a été mis à ma disposition.

Certains de ses points ont fait l'objet de quelques ébauches avant mon arrivée. Cela souligne le besoin latent de ce projet. Mais toutes ces ébauches ont finalement été revues durant ce stage.

2.3.2 Définition

Le but du stage est de créer une bibliothèque, facilement réutilisable, qui permette aux développeurs du JMMC d'accélérer et de généraliser la conception d'applications graphiques.

En effet, beaucoup de mécanismes et d'informations sont redondents lors d'un tel développement. Par exemple la fenêtre d'à-propos ou encore, les informations propres à l'application comme son nom, sa version etc. Celles-ci sont utilisées et dupliquées à plusieurs endroits. La bibliothèque doit être capable de centraliser les parties communes à tout développement d'application graphique, et cela de la manière la plus abstraite qui soit. Elle doit aussi factoriser les fonctionnalités récurrentes utilisées.

Un mécanisme peut être extrêmement puissant mais compliqué à mettre en oeuvre. Le but ici est de créer un mécanisme complexe mais extrêmement simple à implémenter afin que les ingénieurs qui ne connaissent pas les applications du JMMC puissent développer une application grâce à JMCS en suivant simplement quelques directives. C'est là que réside, je pense, toute la difficulté et en même temps tout l'intérêt du stage.

2.4 Environnement de développement

La bibliothèque JMCS est écrite en Java compatible version 1.5 et ultérieure. Elle doit être utilisable par des applications Java WebStart, mais aussi par des applications autonomes (archives JARs). Elle est développée dans l'environnement

"Mariotti Common Software" (MCS) qui est une architecture normalisée propre au JMMC. Celle-ci offre un document de conventions de nommage et de codage que les développeurs doivent respecter.

MCS est composé de modules[3]. Ceux-ci sont des ensembles de fonctions traitant une même problématique (par exemple "msg", module de communication à travers un réseau, ou encore "err", module de traitement d'erreurs). Ces modules sont parallèlement archivés sur un serveur et disposent d'une documentation générée par Doxygen[4] (pour plus d'informations, veuillez vous référer au lien n-15 de la webographie à la fin de l'annexe), un outil de création de documentation extraite du code source.

Les modules sont compilées grâce à des "makefiles" qui leurs sont propres. Il existe même un script bash[5] qui génère une arborescence de module MCS, des fichiers de classes[6] C, C++, Java etc.

Comme outil de travail, on a mis à ma disposition un poste Linux (distribution Mandriva) et j'ai, tout au long du stage, développé JMCS sur un serveur distant Linux proposant l'architecture MCS, via le protocol SSH (protocol de communication sécurisé) et grâce à un éditeur de texte commun nommé Kate.

2.5 Méthodologie de travail

2.5.1 Outils

Dans l'optique d'une programmation modulaire et facilement maintenable, nous avons développé chaque brique logicielle selon le patron de conception Modèle-Vue-Contrôleur (MVC[7], cf. figure "Patron de conception MVC" page 10). C'est un patron de conception qui permet de séparer efficacement les données des traitements, afin

[3]Ensemble de fichiers sources partagant un même thème

[4]Doxygen est un logiciel informatique libre permettant de créer de la documentation à partir du code source d'un programme. Pour cela, il tient compte de la grammaire du langage dans lequel est écrit le code source, ainsi que des commentaires s'ils sont écrits dans un format particulier. Le code de Doxygen a été écrit en grande partie par Dimitri van Heesch.

[5]Bash est un logiciel libre publié sous GNU GPL. Il est l'interprète par défaut sur de nombreux Unix libres, notamment sur les systèmes GNU/Linux. C'est aussi le shell par défaut de Mac OS X et il a été porté sous Windows par le projet Cygwin.

[6]En informatique, la classe est un des concepts de base de la programmation orientée objet. On appelle classe un ensemble d'objets partageant certaines propriétés (les méthodes et les attributs).

[7]Le Modèle-Vue-Contrôleur (en abrégé MVC, de l'anglais Model-View-Controller) est une architecture et une méthode de conception qui organise l'interface Homme-machine d'une application logicielle. Il divise l'ihm en un modèle (modèle de données), une vue (présentation, interface utilisateur) et un contrôleur (logique de contrôle, gestion des évènements, synchronisation), chacun ayant un rôle précis dans l'interface.

de garantir une indépendance entre chaque élément. Ce qui offre ainsi plus de souplesse en vue de futures solutions.

Pour plus d'informations, veuillez vous référer au lien n-3 de la webographie à la fin de l'annexe.

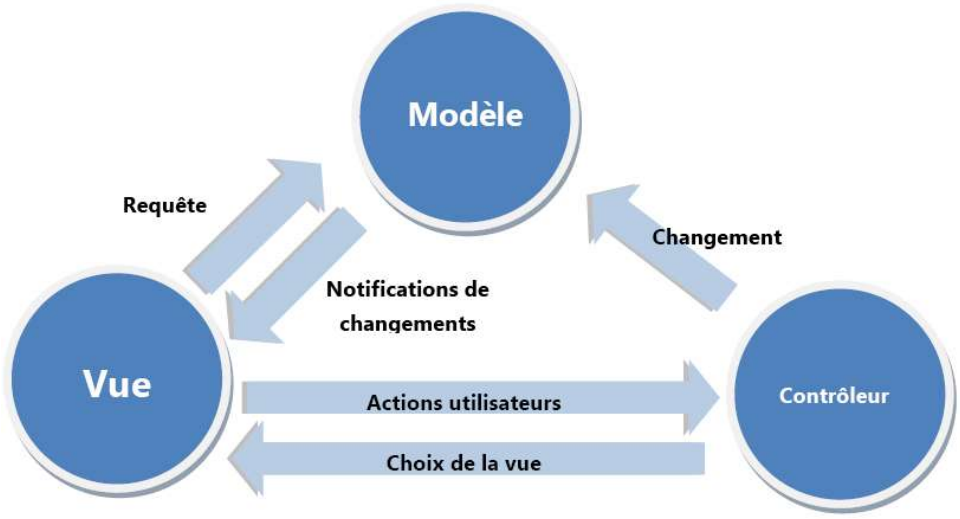


FIG. 1 – Patron de conception MVC

Afin de respecter le patron MVC, nous devons créer trois entités :

- Le modèle : Il représente les données que l'on veut manipuler. Par exemple, cela peut être une classe Java représentant une personne. Quand on met à jour le modèle, celui-ci en avertit la vue.
- La vue : Elle représente l'affichage du modèle : phrases, fenêtres...
- Le contrôleur : Il crée le modèle et lui affecte une vue. C'est l'interface entre l'utilisateur et le système.

Grâce à la modularité offerte par ce patron de conception, si un jour nous changeons de méthode de stockage pour nos données, il suffira de changer la manière d'accéder aux données dans le modèle, le reste n'aura pas besoin d'être modifié ! De même pour l'affichage.

En ce qui concerne maintenant la sauvegarde de mon travail, j'au utilisé CVS[8], un système de gestion de fichiers en configuration. MCS offre de plus une structure

[8]CVS, acronyme de Concurrent Versions System, est un système de gestion de versions libre, successeur de SCCS, originellement écrit par Dick Grune en 1986, puis complété par Brian Berliner (avec le programme cvs lui-même) en 1989, et par la suite amélioré par de très nombreux contributeurs. Puisqu'il aide les sources à converger vers la même destination, on dira que CVS fait la gestion concurrente de versions ou de la gestion de versions concurrentes.

adaptée à CVS. Cela nous permet de revenir à une version précédente en cas d'erreur, de développer en parallèle ou de distribuer un version stable.

Et pour finir, nous utilisons UML[9] afin de schématiser nos réflexions (pour plus d'informations, veuillez vous référer au lien n-5 de la webographie à la fin de l'annexe).

2.5.2 Organisation

Quant à l'organisation et répartition des tâches au sein de l'équipe, chaque lundi matin, une réunion est organisée afin de discuter, chacun à son tour, de l'état d'avancement des différents projets et des problèmes rencontrés.

Pour garantir enfin un niveau de qualité élevé du code, nous effectuons des lectures de codes croisées, c'est-à-dire que notre code est relu par d'autre programmeurs afin qu'ils puissent nous faire des remarques constructives ou nous fassent des suggestions de nouvelles idées ou d'améliorations.

Par ailleurs, les discussions sont fréquentes entre les membres de l'équipe et, ensemble, elles permettent d'avancer vite et surtout d'avancer mieux en accord avec tout le monde.

2.6 Documentation

Une bonne documentation est indispensable à tout développement de qualité. Dans le cadre de mon stage, il en faut deux distinctes. Une de développement, et une à destination des utilisateurs.

2.6.1 Développement

La documentation pour les développeurs du JMMC sert à garantir la pérénité de mon travail. Elle est constituée de commentaires dans le code, et générée ensuite automatiquement à la compilation par un outil nommé Doxygen. De plus, ce rapport sera disponible afin de faciliter de futures améliorations ou maintenances de JMCS.

[9] UML (en anglais Unified Modeling Language, « langage de modélisation unifié ») est un langage graphique de modélisation des données et des traitements. C'est une formalisation très aboutie et non-propriétaire de la modélisation objet utilisée en génie logiciel.

2.6.2 Utilisateur

La documentation destinée aux utilisateurs a pour but d'expliquer aux scientifiques l'utilisation de nos applications, notament les fonctionnalités de celles-ci. Nous devons donc créer une documentation claire et accompagnée de captures d'écran.

3 Travail effectué

Mon travail a donc constitué en l'élaboration de JMCS, une librairie de développement d'applications graphiques Java adaptée aux besoins du JMMC.

JMCS est destiné à deux classes d'utilisateurs. En premier lieu, aux développeurs du JMMC qui l'utilisent pour la conception de leurs applications graphiques. En effet, une fois la bibliothèque créée, ceux-ci l'utiliseront à chaque nouveau développement dans le but de ne pas chaque fois réécrire le même code (c'est ce que l'on appele de la mutualisation). Enfin, n'oublions pas que JMCS est à destination des scientifiques qui sont par conséquent nos utilisateurs finaux.

De plus, le fait de définir une méthodologie de création, une structure d'application commune, permet à n'importe quel développeur ne connaissant pas JMCS, de créer facilement une application compatible avec l'environnement du JMMC. Le fait de comprendre le fonctionnement d'une application utilisant JMCS permet de comprendre comment toutes les autres applications JMCS fonctionnent, facilitant ainsi les opérations de maintenance.

Enfin, dans le but de préserver les réflexes attenant à l'environnement de travail favori de chacun de nos utilisateurs et pour leur éviter un temps d'adaptation ou d'apprentissage superflu, JMCS doit s'adapter parfaitement aux différents systèmes d'exploitation. On peut, par exemple, citer ici les raccourcis claviers qui changent selon les plateformes d'exécution, et qu'il faudra donc gérer.

Concernant l'organisation de mon travail, j'ai commencé par développer les différentes fonctionnalités de la librairie comme l'affichage de la fenêtre d'à-propos ou l'affichage de la fenêtre d'aide utilisateur. Puis j'ai implémenté celles-ci au fur et à mesure dans une classe application afin de les généraliser. En parallèle à tout cela, j'ai testé les fonctionnalités en créant un module de tests.

Dans le point suivant, je vais vous expliquer comment j'ai procédé à la création de la fenêtre d'à-propos ainsi qu'à celle de la fenêtre de démarrage.

3.1 Fenêtre d'à-propos/écran de démarrage

Dans l'optique d'affirmer une certaine appartenance commune aux logiciels du JMMC, il fallait proposer un mécanisme standard d'affichage des informations propres à chaque application. Bien sûr, on voulait faire en sorte que le mécanisme soit ex-

trêmement simple à mettre en oeuvre, voire complètement automatique !

Les informations à afficher étaient les suivantes :

- Logo du JMMC ainsi qu'un lien pointant vers son site principal.
- Lien pointant vers la page web de l'application.
- Nom et version de l'application.
- Liste des dépendances et modules.
- Licence et copyright.
- Texte libre pour d'éventuelles informations complémentaires.
- Date de compilation et version du compilateur.

Ces informations doivent être affichées dans la fenêtre d'à-propos (AboutBox, cf. figure "Fenêtre d'à-propos" page 15), et un sous-ensemble (logo, nom et version de l'application et compilateur), dans la fenêtre de démarrage de l'application (SplashScreen, cf. figure "Fenêtre de démarrage" page 14). Il faut donc proposer un système unique d'accès à ces données. Cela nous permettra d'ailleurs par la suite de les utiliser dans d'autres circonstances.

FIG. 2 – Fenêtre de démarrage

Afin de n'avoir qu'un seul fichier répertoriant les informations propres à l'application, nous avons choisi d'utiliser l'"eXtensible Markup Language" (XML[10]). Un langage très souple qui se couple très bien à la programmation Java. Ce n'est pas nécessairement la solution habituelle en Java, que sont les ressources (fichiers "properties"), mais elle répond parfaitement aux besoins ennoncés et nous permet, de plus, de facilement utiliser le fichier XML à travers des scripts bash ou bien de le transformer pour un affichage sur Internet par exemple grâce aux scripts de transformation XSL.

[10]XML (eXtensible Markup Language (en)[1], « langage de balisage extensible ») est un langage informatique de balisage générique. Le World Wide Web Consortium (W3C), promoteur de standards favorisant l'échange d'informations sur Internet, recommande la syntaxe XML pour exprimer des langages de balisages spécifiques.

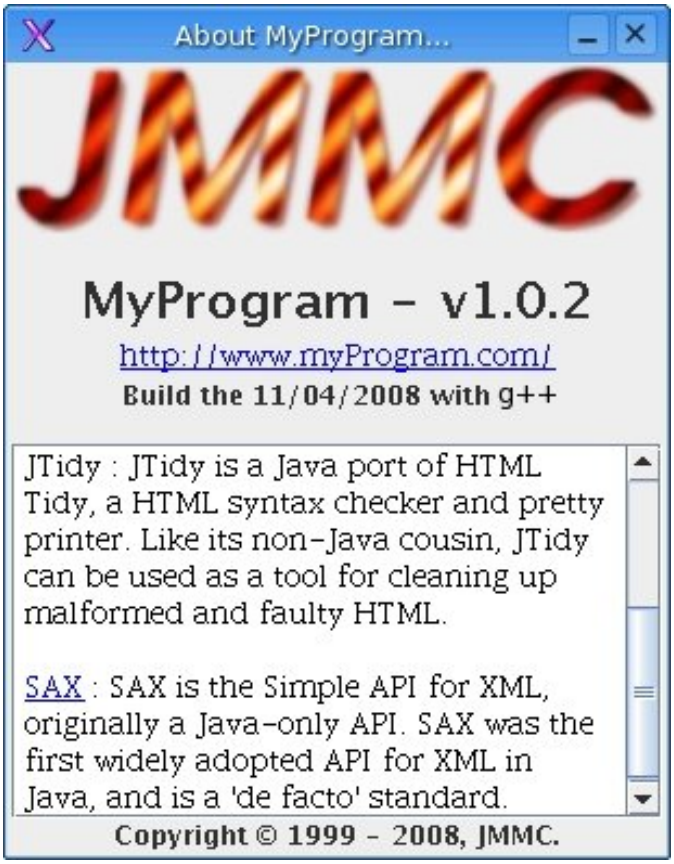


FIG. 3 – Fenêtre d'à-propos

Lors du développement de ces accès aux données, un des problèmes rencontrés était lié à l'environnement de l'application et plus particulièrement à l'emplacement des ressources. En effet, une application au sein de son module dispose localement de ses ressources (fichiers quelconques). Mais le code permettant l'accès à ces ressources est quant à lui dans le module JMCS. J'ai donc dû faire particulièrement attention au mécanisme de liaison entre modules afin de récupèrer les bonnes informations et surtout, j'ai dû chaque fois penser à l'éventualité de l'abscence d'une ressource afin de garantir la robustesse de mon travail.

Fallait-il mettre systématiquement un fichier par défaut dans le module de la classe application ou user de valeurs par défaut, comme par exemple mettre "MyApplication" si le nom de l'application est introuvable ou vide ? Le fait de directement mettre une valeur par défaut à l'avantage d'être plus rapide à programmer, cependant, un fichier par défaut offre d'autres avantages comme le fait d'être plus souple en cas de modification de ses valeurs (pas de recompilation nécessaire).

Ci-dessous, la structure retenue pour le fichier XML :

```
<ApplicationData link="">
        <program name="" version=""/>
        <compilation date="" compilator=""/>
        <text></text>
        <copyright></copyright>
        <dependences>
                <package name="" description="" link=""/>
        </dependences>
</ApplicationData>
```

Afin d'utiliser un fichier XML pour stocker les informations d'une application, j'ai dû rechercher le moyen d'utiliser celui-ci avec Java.

3.1.1 Analyseur syntaxique XML

J'ai étudié plusieurs solutions pour accéder aux données encapsulées dans les fichiers XML. Seulement, nous ne voulions pas développer nous même un analyeur syntaxique[11] XML. Ni utiliser, par exemple, JDOM (un analyseur syntaxique XML justement) car cela impliquait plus de code d'un entretien coûteux en temps en cas de modification du schéma XSD[12] décrivant la structure de notre fichier XML.

Au cours de recherches sur Internet, nous avons retenu Castor (pour plus d'informations, veuillez vous référer au lien n-6 de la webographie à la fin de l'annexe), une bibliothèque Java opensource offrant la génération automatique de classes Java à partir d'un schéma XSD. Les classes ainsi générées possèdent des accesseurs (fonctions permettant l'accès à des données d'une classe) permettant de manipuler aisément le contenu du fichier XML qui répond au schéma XSD.

De cette façon, si nous souhaitons ajouter une information concernant l'application, il nous suffit de compléter le schéma et d'ajouter la liaison avec cette donnée dans le modèle. Cette bibliothèque répondait parfaitement à nos besoins et je l'ai donc mise en oeuvre.

[11] L'analyse syntaxique consiste à exhiber la structure d'un texte, généralement un programme informatique ou du texte écrit dans une langue naturelle. Un analyseur syntaxique (parser, en anglais) est un programme informatique qui réalise cette tâche. Cette opération suppose une formalisation du texte, qui est vu le plus souvent comme un élément d'un langage formel, défini par un ensemble de règles de syntaxe formant une grammaire formelle.

[12] XML Schema est un langage de description de format de document XML permettant de définir la structure d'un document XML. La connaissance de la structure d'un document XML permet notamment de vérifier la validité de ce document. Un fichier de description de structure (XML Schema Description en anglais, ou fichier XSD) est donc lui-même un document XML.

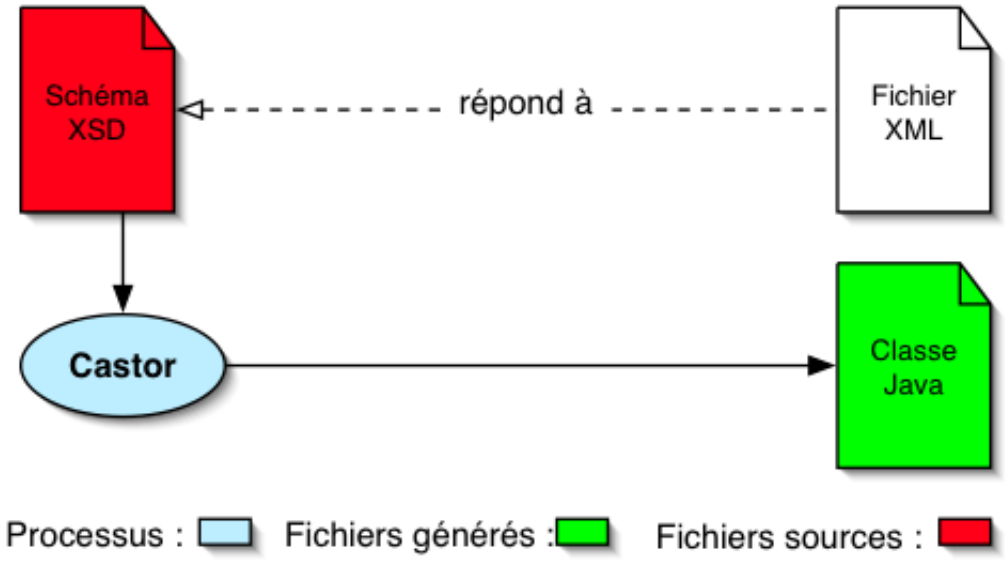


FIG. 4 – Modèle du fonctionnement de castor

Nous nous sommes donc fixé un schéma XSD à partir duquel nous regénérons les classes Java à chaque compilation par le biais d'un script bash que nous avons créé et qui utilise Castor (cf. figure "Modèle du fonctionnement de castor" page 17)

Enfin, nous créons un fichier XML dans le module JMCS, que nous appelons "fichier de données par défaut" et que nous utiliserons au cas où l'application ne disposerait pas encore du fichier dans son module. Cette vérification est effectuée au lancement de l'application, avant le SplashScreen. De cette manière, on fournira à l'utilisateur des valeurs par défaut garantissant l'exécution de l'application ! Encore une fois, un mécanisme fiable d'accès aux ressources entre les modules a dû être mis au point par mes soins.

Une fois le système d'affichage de ces deux fenêtres ainsi que le mécanisme d'accès aux données de l'application terminé, j'ai décidé de commencer la généralisation de la gestion des journaux d'exécution.

3.2 Gestion des journaux d'exécution

Afin de faciliter le travail des développeurs lors des phases de test, il est nécessaire d'avoir une trace du déroulement de l'exécution de l'application.

Le but ici est donc de généraliser la gestion du journal d'exécution d'une application. En effet, durant le développement d'une application, les programmeurs insèrent dans le code des signaux commentés, ayant une importance variable de

"FINEST" à "SEVERE" en comptant 5 autres paliers appelée niveaux de verbosité qu'ils utilisent pendant le débogage.

On utilise les niveaux les plus hauts comme "FINEST" pour des informations de débogage très précises comme l'entrée dans une méthode. A l'inverse, les signaux "SEVERE" indiquent que l'application va se terminer car une opération, nécessaire au bon déroulement de l'exécution, ne s'est pas effectuée correctement.

Ci-dessous, la liste des niveaux de verbosité :

- 1 SEVERE : L'application va se terminer. Une opération cruciale ne s'est pas bien terminée.
- 2 WARNING : Une opération ne s'est pas correctement déroulée. Cependant, il a étépossible de contourner le problème et l'application va continuer.
- 3 INFO : Information importante à connaître (connexion d'un client sur un serveur par exemple) et à destination de l'utilisateur.
- 4 CONFIG : Information de configuration utile.
- 5 FINE : Information assez précise (résultat d'une condition).
- 6 FINER : Information précise (création d'un objet).
- 7 FINEST : Information très précise (affectation d'une variable, parcours d'un vecteur etc.).

Il est important de préciser ici que dans la suite du développement, nous avons regroupé les trois derniers niveaux en un seul : FINE.

A l'exécution, on fixe le niveau de verbosité le plus haut souhaité. L'application affiche alors ou non ces signaux selon leur niveau. C'est ce qu'on appelle des traces. Ces traces forment le journal d'exécution, notamment utilisé pour comprendre l'origine d'un problème en cas d'un retour d'erreurs.

Ci-dessous, un exemple de trace :

```
29 mai 2008 14:04:09 fr.jmmc.mcs.gui.App interpretArguments
INFO: Set logger level to 5
29 mai 2008 14:04:09 fr.jmmc.mcs.gui.App <init>
FINE: Application arguments interpreted
29 mai 2008 14:04:10 fr.jmmc.mcs.gui.App showSplashScreen
INFO: Show SplashScreen
29 mai 2008 14:04:10 fr.jmmc.mcs.gui.ApplicationDataModel getLogoURL
INFO: Logo URL = logo.png
```

Tous ces messages peuvent être stockés ou affichés de différentes manières (fi-

chier, variable, affichage console etc.). Dans notre cas, on en a utilisé deux : affichage console et variable. Ainsi, nous pouvons suivre le déroulement de l'application sur la console tout en le stockant dans une variable en vue d'un éventuel envoi par d'autres canaux de communication.

Dans le cadre d'une grosse application, il est très difficile de distinguer les traces intéressantes au cours d'un bug. C'est pourquoi, il nous fallait un utilitaire permettant de trier les classes pendant que l'application s'exécute.

3.2.1 LogGui

A la recherche d'un utilitaire de filtrage en temps réel de traces (pour ne garder par exemple que celles appartenant à une classe), j'ai trouvé sur Internet un utilitaire nommé LogGui (cf. figure "Capture d'écran LogGui" page 20, pour plus d'informations, veuillez vous référer au lien n-7 de la webographie à la fin de l'annexe).

LogGui est un utilitaire graphique de gestion des traces d'exécution. En effet, durant leur affichage pendant l'exécution, on ne peut pas changer "à la volée" le niveau de verbosité des traces. LogGui offre cette fonctionnalité. Il permet aussi de changer le format d'affichage des traces ou encore de cacher les traces d'une classe qui ne nous intéresse pas etc.

J'ai donc intégré cet outil à JMCS pour le généraliser à tous les journaux d'exécution de toutes les applications du JMMC.

Une fois le journal d'exécution de l'application disponible, je me suis penché sur le système de retours d'erreurs et commentaires.

3.3 Gestion des retours d'erreurs et commentaires

Dans le but de corriger le plus d'erreurs possibles, il est souhaitable d'implémenter dans chaque application un système de retours d'erreurs à destination des utilisateurs.

Mais ce n'est pas l'unique raison d'un tel mécanisme. Les retours de commentaires, avis ou remarques des utilisateurs sont très intéressants et permettent d'améliorer sensiblement la qualité d'une application, et la perception des utilisateurs quant à la prise en compte de leurs besoins.

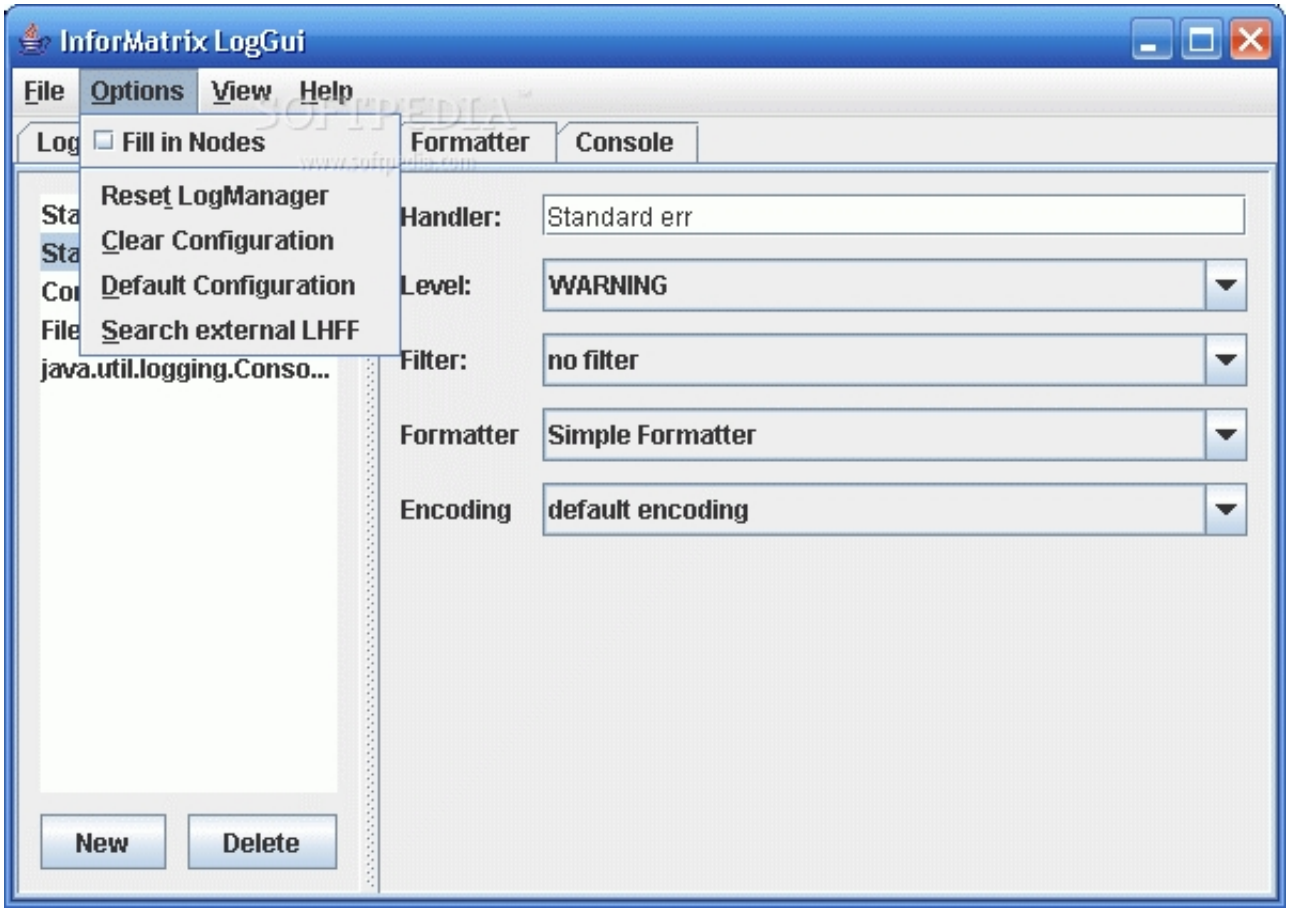


FIG. 5 – Capture d'écran LogGui

Lors de la programmation ou du développement en général, on est plongé dans une logique de conception. On perd donc un certain recul et on ne dispose plus d'une vue en tant qu'utilisateur de l'application. Ainsi, des détails ergonomiques importants peuvent être complétement négligés. Les retours d'erreurs ne servent donc pas uniquement à signaler les bugs mais aussi à améliorer les logiciels.

Il s'agit d'une aide offerte à nos développements, qui s'ajoute aux nombreuses revues de l'application. Il est rare qu'une application ne contienne aucun bug, c'est pourquoi le fait que les utilisateurs puissent nous les faire parvenir nous est d'une grande aide. De plus, cela nous permet de donner une bonne image du service de développement !

Le but est donc de généraliser la gestion des retours d'erreurs ainsi que les commentaires de l'utilisateur. Ceci par le biais d'une fenêtre et de l'envoi d'un rapport à l'équipe de support du JMMC.

Ce rapport devra contenir les informations suivantes :

- Nom et version de l'application concernée.
- Journal d'exécution.
- Propriétés du système hôte (SE, JVM etc.)
- Préférences de l'application.
- Données supplémentaires sur l'application.

Cette fenêtre (cf. figure "Fenêtre de rapports d'erreurs" page 22) doit permettre à l'utilisateur de préciser le type d'erreur rencontrée (bug, évolution, documentation etc.), optionnellement son adresse de messagerie Internet afin de le contacter ultérieurement, et son commentaire.

Comme le JMMC disposait déjà d'un formulaire PHP[13] assurant l'envoi de mails de commentaires, nous avons décidé d'effectuer la transmission du rapport par la soumission des données via une requête HTTP (POST) depuis Java vers ce formulaire, grâce à une librairie opensource nommée "commons-httpclient" (pour plus d'informations, veuillez vous référer au lien n-8 de la webographie à la fin de l'annexe). Les données sont ensuite traitées par le script PHP. Il en résulte soit un message d'erreur si le serveur est inaccessible ou s'il manque un champ, soit une message de confirmation si le mail a bien été envoyé.

Suivant le modèle MVC, nous utiliserons ici deux objets : FeedbackReport (la vue et le contrôleur à la fois), FeedbackReportModel (le modèle). La transmission du rapport sera effectué par le modèle, détaché de l'application (dans un processus s'exécutant en parallèle). Ainsi, si l'envoie échoue, la fenêtre reste utilisable.

Lors d'un clic sur le bouton "submit", celui-ci se désactive afin d'empecher un reclic. La vue se met à jour en affichant une barre de progression non-déterminée. Une fois le rapport envoyé le modèle notifie à la vue qu'il a essayé d'envoyer le rapport (cela a pû echouer...). Celle-ci se met ensuite à jour. Soit elle affiche un remerciement et se ferme seule au bout de quelques secondes, soit elle affiche une fenêtre d'erreur et l'utilisateur a la possiblité de réessayer un autre envoi.

Le point suivant traite de la génération et de la visualisation automatique de l'aide utilisateur. Une des fonctionnalités les plus complexes à mettre en oeuvre.

[13]PHP (acronyme récursif pour PHP : Hypertext Preprocessor[1]), est un langage de scripts libre[2] principalement utilisé pour produire des pages web dynamiques via un serveur HTTP[1], mais pouvant également fonctionner comme n'importe quel langage interprété de façon locale, en exécutant les programmes en ligne de commande.

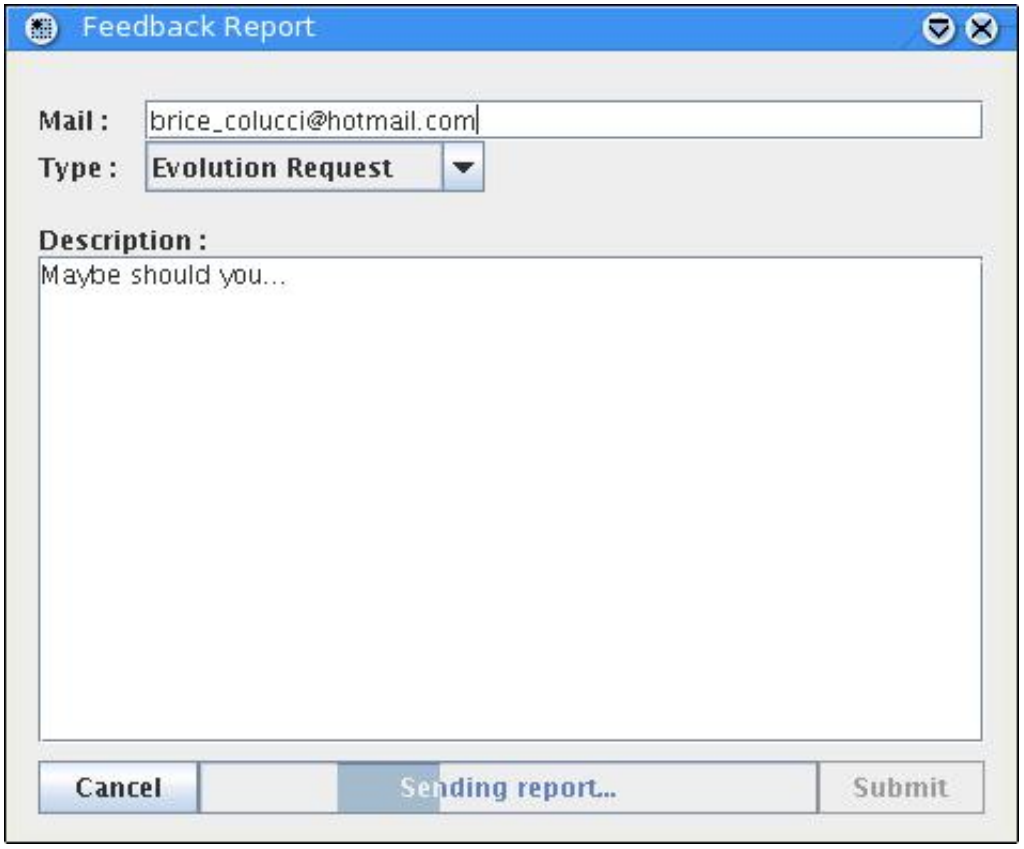


FIG. 6 – Fenêtre de rapports d'erreurs

3.4 Génération/visualisation de l'aide utilisateur

Dans la même optique de généralisation de l'interface graphique des applications du JMMC, nous souhaitions offrir aux utilisateurs un affichage d'aide standardisé.

Le but est donc d'uniformiser la génération de l'aide utilisateur ainsi que le mécanisme d'affichage de celle-ci dans les applications. Au sein du JMMC, les manuels utilisateurs sont rédigés avec LaTeX[14] par les scientifiques participant à l'élaboration des logiciels.

Nous avions envisagé plusieurs solutions pour la consultation de l'aide comme la mise à disposition d'un manuel utilisateur PDF stocké sur le serveur et pointé par un lien dans l'application, ou la création d'un composant de navigation spécialisé.

Nous avons trouvé au cours de recherches sur Internet un système nommé JavaHelp, utilisé par beaucoup d'applications, qui permet justement l'affichage de pages

[14] LATEX est un système logiciel de composition de documents créé par Leslie Lamport, ou plus exactement : une collection de macro-commandes destinées à faciliter l'utilisation du « processeur de texte » TEX.

HTML.

3.4.1 JavaHelp

JavaHelp (pour plus d'informations, veuillez vous référer au lien n-10 de la webographie à la fin de l'annexe) est un système crée par Sun Microsystems. Il permet d'afficher une fenêtre d'aide à partir de fichiers HTML (cf. figure "Fenêtre d'aide utilisateur" page 25). Mais avant d'y parvenir, il nous faut créer d'autres fichiers intermédiaires.

Voici les étapes de création d'une fenêtre d'aide utilisateur au format JavaHelp :

- Créer une archive Jar avec :
  - Un fichier "Table Of Content" (TOC) représentant la table des matières de l'aide utilisateur.
  - Un fichier "Java Help Map" (JHM) définissant les différents liens entre les pages et les titres.
  - Un fichier "HelpSet" (HS) représentant le fichier de configuration de la fenêtre et pointant le fichier TOC ainsi que le fichier JHM.
  - Les fichiers HTML de notre documentation.
- Instancier une classe qui affiche la fenêtre d'aide générée.

Les manuels d'aide utilisateur du JMMC étant créés avec LaTeX (fichiers TEX), nous devons utiliser la commande "latex2html" afin de générer des fichiers HTML 3.2. J'utilise de plus la librairie JTidy (pour plus d'informations, veuillez vous référer au lien n-13 de la webographie à la fin de l'annexe) afin d'analyser syntaxiquement les fichiers HTML 3.2 générés pour ainsi les corriger et les valider selon les normes de codage XHTML actuellement recommandées par W3C[15].

Chaque application dispose d'une documentation qui lui est propre. Cependant, elle peut utiliser d'autres modules ayant leur propre documentation. Il faut donc penser à combiner la documentation de ces modules avec celle de l'application. Dans notre cas, nous souhaitons automatiser la création de ces fichiers par un script bash. Ce script récupère la documentation de chaque module, les convertit en fichiers HTML, crée l'archive Jar et la place dans la bibliothèque de l'application.

Cependant, afin d'automatiser la création des fichiers propres à JavaHelp, nous

[15]Le World Wide Web Consortium, abrégé par le sigle W3C, est un organisme de normalisation fondé en octobre 1994 pour promouvoir la compatibilité des technologies du World Wide Web telles que HTML, XHTML, XML, RDF, CSS, PNG, SVG et SOAP. Le W3C n'émet pas des normes au sens européen, mais des recommandations à valeur de standards industriels.

devions trouver un utilitaire qui permettrait de les générer automatiquement à partir de fichiers HTML. C'est durant cette recherche que nous avonc trouvé JHelpDev.

3.4.2 JHelpDev

JHelpDev (pour plus d'informations, veuillez vous référer au lien n-14 de la webographie à la fin de l'annexe) est une application Java opensource graphique permettant de générer manuellement à partir de fichiers HTML, les fichiers indispensables à JavaHelp (TOC, JHM et HS). Il suffit alors de compresser le tout grâce à la commande "jar -cf". Cependant, il est indispensable afin d'automatiser le processus, d'utiliser de manière systématique les méthodes adéquates fournies par cette application.

Après en avoir obtenu l'autorisation du créateur, j'ai donc créé une application Java utilisant les méthodes nécessaires à JHelpDev de manière automatique. Ensuite, j'ai crée un script bash utilisant cette procédure Java pour simplifier autant que possible le processus de génération de la documentation au format JavaHelp.

Afin de résumer la création de la fenêtre d'aide utilisateur, voici les grandes étapes du processus (cf. figure "Schéma du processus de génération de l'aide utilisateur" page 26) de génération :

- Je récupère chaque module de l'application et je les compile.
- J'utilise latex2html pour créer les fichiers HTML correspondant que je passe ensuite dans JTidy qui les valide et les corrige.
- J'utilise des méthodes de l'application graphique JHelpDev à travers une application Java nommée "jmcsGenerateHelpsetFromHtml" afin de générer automatiquement les fichiers nécessaires à la création d'une archive d'aide utilisateur au format JavaHelp.
- J'archive les fichiers générés je place l'archive dans la librairie de l'application pour préparer son utilisation par JavaHelp.

Toutes ces opérations sont lancées à partir d'un script bash que nous avons appelé "jmcsHTML2HelpSet" et que j'ai créé. Il suffit de lancer celui-ci sur le serveur de développement à partir du dossier source de l'application.

Dans le but enfin de généraliser la définition d'options pour une application, j'ai commencé le développement, détaillé au point suivant, de l'interface "ligne de commande".

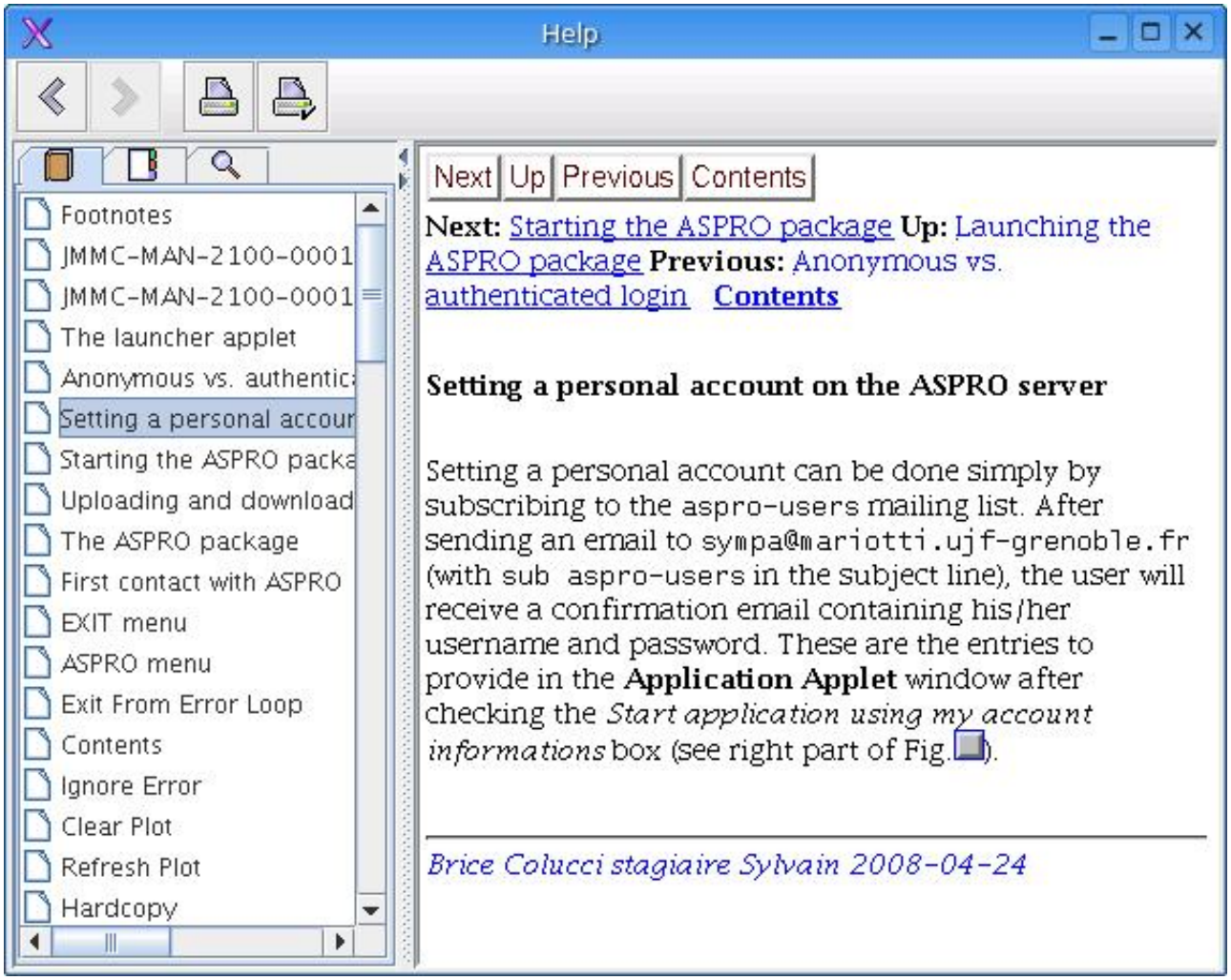


FIG. 7 – Fenêtre d'aide utilisateur

3.5 L'interface "ligne de commande"

Il est possible depuis une console de lancer une application avec des arguments et des options.

Par exemple, dans le cadre d'une utilisation courante, le fait d'afficher les logs ralentit l'exécution de l'application. A contrario, le fait de ne pas les afficher nuit au débogage. Nous aimerions donc, fixer le niveau de verbosité du journal d'exécution sans avoir à recompiler le logiciel.

Tous ces aspects doivent donc à êtres centralisés à travers un gestionnaire d'interprétation d'arguments commun.

Nous souhaitons supporter les arguments suivants pour toute application :

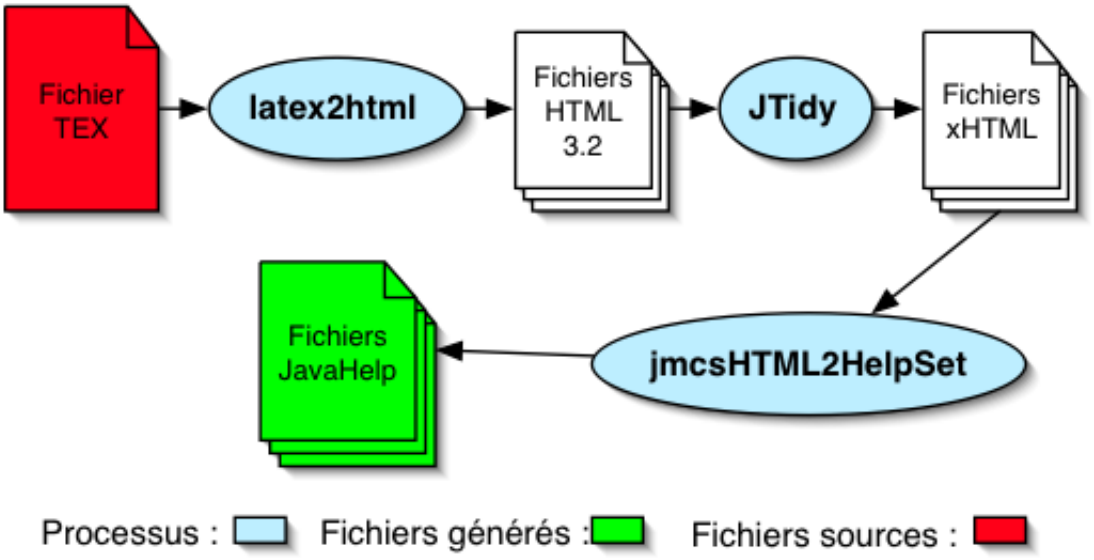


FIG. 8 – Schéma du processus de génération de l'aide utilisateur

- -h : affiche l'aide à propos des commandes en ligne.
- -version : affiche le nom et la version de l'application.
- -v [0|1|2|3|4|5] : définit le niveau de verbosité du journal d'exécution (0 correspond à aucun, 1 à SEVERE et 5 à FINE).

3.5.1 Analyse des options

Afin d'éviter d'implémenter un système de parsage d'une ligne d'arguments déjà existant, nous avons choisi d'utiliser Getopt (pour plus d'informations, veuillez vous référer au lien n-11 de la webographie à la fin de l'annexe), une bibliothèque open-source de GNU, très simple d'utilisation, et qui permet d'analyser syntaxiquement les arguments passés à une application. On peut définir un formalisme comme "h :v" qui signifie qu'on attend une clef "-h" avec un argument et une autre "-v" sans argument. Cette bibliothèque gère également les "arguments longs" comme par exemple "-version".

Getopt est de plus généralisé et a déjà fait ses preuves, notament dans divers langages comme le C ou le python.

Enfin, nous souhaitons pouvoir ajouter d'autres options, spécifiques à l'application, en utilisant par exemple une surchage de méthode.

3.6 Uniformisation des menus/raccourcis claviers

Toujours dans le but de conserver un affichage homogène de la famille d'applications du JMMC, nous devions généraliser les menus standards à toute application comme "File", "Edit" etc., ainsi que les raccourcis claviers qui varient selon le système d'exploitation.

Pour ce faire, j'ai implémenté des fonctions introspectives qui gèrent automatiquement les menus standards et ajoutent à la barre des menus, dans un ordre définissable, des fonctions existantes de l'application pouvant êtres dans n'importe quelle classe.

Ce mécanisme, pourtant puissant, ne répondait toutefois pas à certains besoins concernant ce point du cahier des charges, notamment l'injection de separateurs entre les menus ainsi que d'autres fonctionnalités comme les menus à case a cocher "checkbox".

Nous avons donc envisagé une nouvelle solution basée sur l'utilisation d'un fichier XML décrivant les menus. Dans un premier temps, le fait d'ajouter directement dans le fichier XML existant un formalisme lié aux menus nous semblait simple et adapté. Cependant, nous avons tout de même cherché s'il existait d'autres solutions par ailleurs.

3.6.1 Formalisme XML concernant les menus

Le fait d'utiliser le fichier XML existant, afin d'y implémenter un formalisme pour les menus, nous semblait simple, certes. Cependant, peut être qu'un mécanisme semblable existait déjà.

Au cours de recherches sur Internet, nous avons étudié JXUL (pour plus d'informations, veuillez vous référer à la webographie à la fin de l'annexe), une librairie de création de menu pour les applications Java, dont le noyau XUL est notament utilisé dans la construction d'applications WEB telle Firefox.

Basé sur un formalisme XML, la création d'un menu devient triviale et, dans notre cas, celui-ci nous laisse libre pour diverses manipulations. Cette librairie ayant déjà fait ses preuves, nous pouvons sans inquiètudes l'utiliser. De plus, nous pouvons parfaitement coupler ce mécanisme avec celui déjà implémenté concernant les fonctionnalités d'accès aux informations propres à l'application.

Mon stage, durant deux semaines de plus que prévu, ne me permet pas ici de vous exposer la suite des étapes concernant ce point du cahier des charges car il est actuellement en développement !

3.7 Mise en oeuvre des fonctionnalités réalisées

En général, une application utilise des fonctions dans le but de traiter des données. Ces données peuvent êtres de différentes natures et se localisent à divers endroits. Cela oblige, à chaque nouveau développement, de recréer les mécanismes d'accès et de traitements à celles-ci.

Pourtant, ces procédures ont des points communs et plusieurs applications utilisent de temps en temps les mêmes données. Si on ne les généralise pas, celles-ci sont dupliquées. Ainsi, quand on doit changer une donnée, il faut chercher où celle-ci est utilisée ailleurs et mettre à jour, le même code qui permet d'y accèder, à des emplacements différents.

C'est afin de remédier à ces situations, dans le cas des informations concernant une application, qu'il faut créer un mécanisme standard de création d'application. Il faut réunir, en suivant un modèle commun, les informations communes ainsi que les fonctionnalités redondantes.

3.7.1 AppFramework

La logique nous a tout de suite conduit à chercher si un tel mécanisme existait déjà. Nous avons trouvé AppFramework (pour plus d'informations, veuillez vous référer au lien n-12 de la webographie à la fin de l'annexe), un espace de travail (en anglais "framework"[16]) correspondant à nos besoins, développé par Sun Microsystems dans le cadre du "Java Specification Requests"(JSR) numéro 296 (cf. figure "Architecture du JSR-296" page 29). Une demande des utilisateurs concernant un framework d'application graphique Java.

AppFramework offre de nombreuses et très puissantes fonctionnalités concernant les applications graphiques. Il a même été directement utilisé dans certains environnements de développement intégré (IDE en anglais) Java et notamment dans NetBeans (conçu par Sun). Il gère le cycle de vie d'une application, c'est-à-dire tous les

[16]En informatique, un framework est un espace de travail modulaire. C'est un ensemble de bibliothèques, d'outils et de conventions permettant le développement d'applications. Il fournit suffisamment de briques logicielles et impose suffisamment de rigueur pour pouvoir produire une application aboutie et facile à maintenir. Ces composants sont organisés pour être utilisés en interaction les uns avec les autres.

évènements depuis son lancement jusqu'aux opérations à effectuer, pour finir par sa fermeture ainsi que la sauvegarde de son état.

Mais il permet aussi de gérer très simplement les ressources de l'application, sous forme classique en Java de fichiers "properties", ainsi que beaucoup de fonctionnalités concernant les interfaces graphiques.

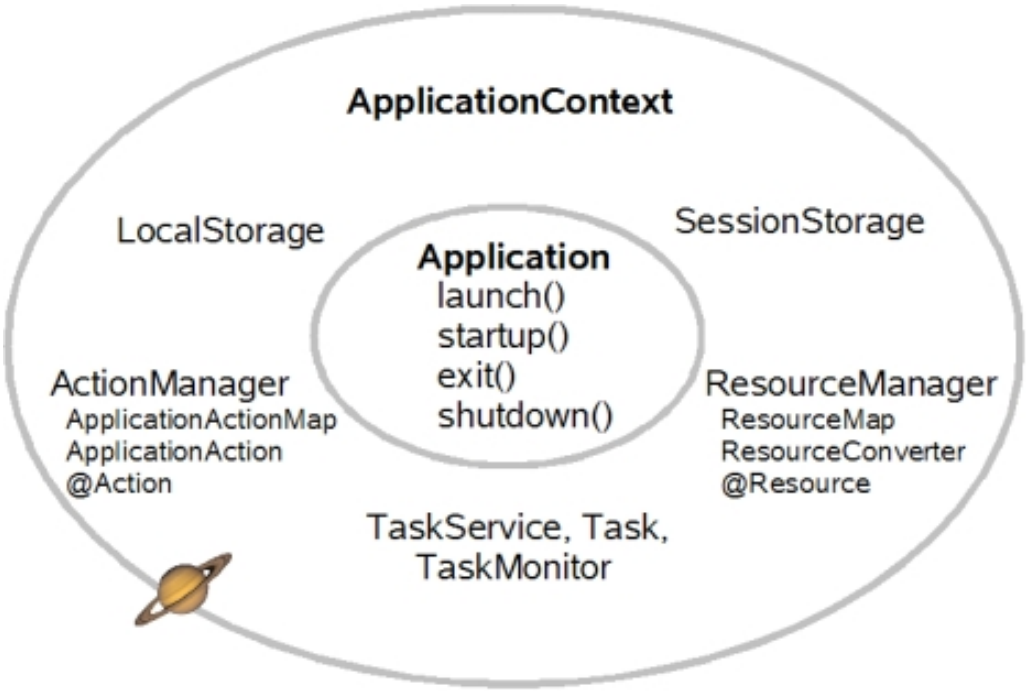


FIG. 9 – Architecture du JSR-296

Le problème est qu'il ne gère tout cela que si l'on respecte une structure très précise. On ne peut pas, non plus, paramétrer ou compléter certains points de son fonctionnement du fait de sa grande rigidité (qui par ailleurs fait sa force). Ainsi, certains de nos besoins ne pouvaient être solutionnés via ce framework.

Nous avons donc décidé non pas de l'utiliser, mais de s'inspirer de son mode de fonctionnement en adaptant celui-ci à nos besoins. C'est ainsi que toutes les réflexions à propos d'une classe application et d'un cycle de vie commun commencèrent.

3.7.2 Cycle de vie

Dans le but d'uniformiser, non plus cette fois l'aspect mais le fonctionnement général de toute application, nous devions réfléchir à un rythme, un ordre de lancement des tâches communes. Un cadre fixe que toutes nos applications doivent respecter

afin de garantir leur généralisation.

Pour ce faire, nous avons utilisé la notion d'abstraction des langages de programmation. En quelque sorte, nous avons imposé des fonctions obligatoires à implémenter et nous ordonnons leur lancement à partir d'une classe mère de toute application, nommé ici "classe application".

Le cycle de vie que nous avons retenu (cf. figure "Cycle de vie d'une application" page 31) est très simple. Le SplashScreen est affiché tant que l'application s'initialise en arrière-plan avec un temps minimum d'affichage de 2 secondes pour éviter un effet de scintillement que l'utilisateur ne comprendrait pas.

Ensuite, le SplashScreen est caché et l'application s'exécute normalement.

Une fois l'exécution terminée, une procédure facultative est appelée à la fermeture au cas où il y aurait des demandes de sauvegardes, des fichiers à écrire ou toutes autres possiblités.

Afin de cadencer systématiquement la vie d'une application, j'ai donc créé les procédures abstraites "init", "execute" et "exit' devant êtres implémentées dans chaque application, et appelées depuis la classe mère ! Ainsi, je peux contrôler leur déclenchement temporel.

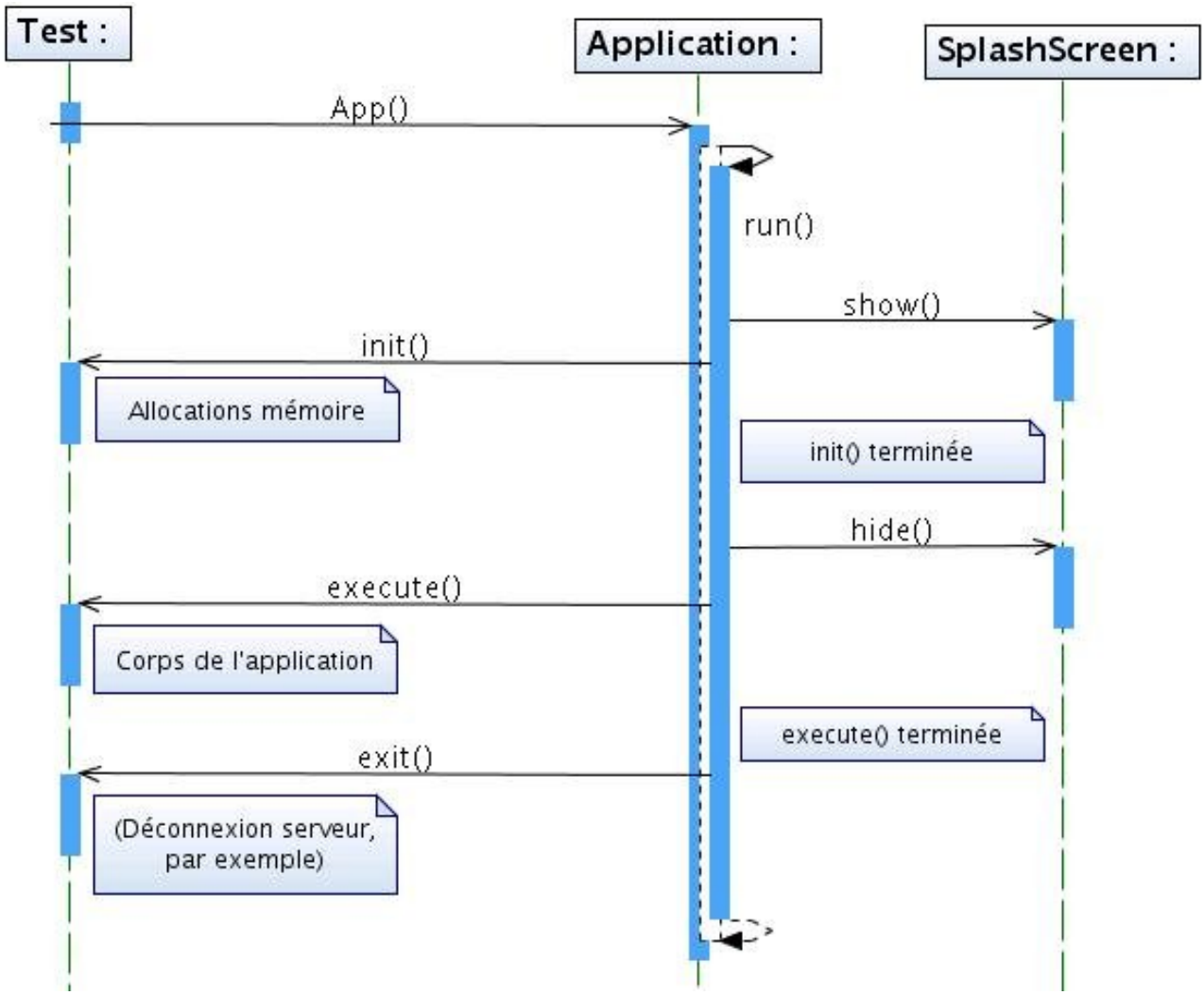


FIG. 10 – Cycle de vie d'une application

Ci-dessous, un exemple général d'utilisation :

```
// Dans JMCS
Classe Application
{
        methode abstraite initialiser();
        methode abstraite executer();
        methode abstraite quitter();

        constructeur()
        {
                recuperer_donnees();
                cycle_de_vie();
        }

        methode cycle_de_vie()
```

```
        {
                montrer_splashscreen(); // En parallele

                initialiser();

                cacher_splashscreen();

                executer();
                quitter();
        }
}

// Dans un module quelconque
Classe UneApplication derivee_de Application
{
        constructeur() {} // Appel du constructeur de Application

        initialiser() {...} // Allocations memoire

        executer() {...} // Corps principal de l'application

        quitter() {...} // Liberation de la memoire, sauvegarde des
           documents etc.

        main(arguments)
        {
                nouveau UneApplication();
        }
}
```

3.7.3 Application de tests

La première étape du développement fut de factoriser les choses simples qui pouvaient l'être dans une classe application et de construire, pour les futurs tests, un module distinct jouant le rôle d'une application quelconque (cf. annexe B). Ce module hériterait des fonctionnalités de la classe mère "Application".

Ainsi, j'ai pu factoriser dans un premier temps l'affichage de l'AboutBox et du SplashScreen. J'ai commencé ensuite à développer, point par point, les fonctionnalités du cahier des charges tout en les testant dans le module distant. Certains points ou fonctionnalités dépendant d'autre(s) point(s) m'ont obligé quelques fois à entamer leur création.

Plus tard, les problèmes liés aux localisations des modules ont fait surface et

ont demandé quelques jours de travail. Dans l'ensemble, et une fois ces problèmes surmontés, le développement des briques logicielles était constant. En parallèle, mon maître de stage, ainsi que son coéquipier, ont commencé à utiliser mon travail sur les différentes autres plateformes de déploiement (Mac OS X, Windows) tout en me rapportant les bugs constatés.

Pour finir, divers changements comme la transformation des fonctions d'affichage en actions eurent lieu. L'intérêt réel de la classe application était indéniable. Le gain en simplicité est de mise, et la mise en oeuvre est intuitive. Tous les points traités du cahier des charges ont été entièrement testés ainsi. C'est en quelque sorte un framework d'application graphique Java correspondant aux besoins du JMMC qui a été développé.

4 Bilan du stage

Je tiens à préciser que mon stage s'étend sur 3 mois au total. Et donc que certains points du cahier des charges, comme l'uniformisation des menus et raccourcis claviers ou encore la gestion des préférences, sont encore en cours de développement !

4.1 Expérience

La plus grande partie de l'expérience que j'ai pu tirer de ce stage provient sans nul doute du travail en équipe. Le fait de participer à des réunions et d'exposer ses idées et ses problèmes, ainsi que celui de réfléchir à plusieurs sur une idée donnent plus d'expérience que de programmer seul.

En effet, j'ai appris à envisager plus de possibilités dans mes réflexions et ainsi j'ai élargi ma façon de voir les choses et de penser. J'ai appris à écouter quelqu'un même si je pensais avoir raison, et j'ai même certaines fois admis avoir tort ! Et grâce à cela j'ai pu avancer un peu plus.

J'ai beaucoup apprécier l'atmosphère de travail en laboratoire et contrairement aux idées reçues on y travaille dur et pour de bonnes causes ! Le fait aussi de participer à des séminaires et de se cultiver dans un domaine très interessant n'était pas non plus superflus.

Pour finir, on m'a souvent dit de prendre des notes et j'en ai saisi l'intérêt. Je pense que j'ai beaucoup évolué, encore plus qu'avec mes anciennes expériences ou en tout cas sur d'autres aspects. Je pense aussi aujourd'hui être mieux préparé à m'intégrer au monde du travail même si je souhaite continuer mes études.

4.2 Compétences acquises

Je suis très attiré par le langage Java. Je le trouve très complet. Cependant, le plus important de nos jours n'est pas de savoir programmer. C'est de savoir assembler ce qui existe déjà. L'exemple même est mon stage où je n'ai cessé de chercher des bibliothèques, de les tester et enfin, de les relier avec l'existant.

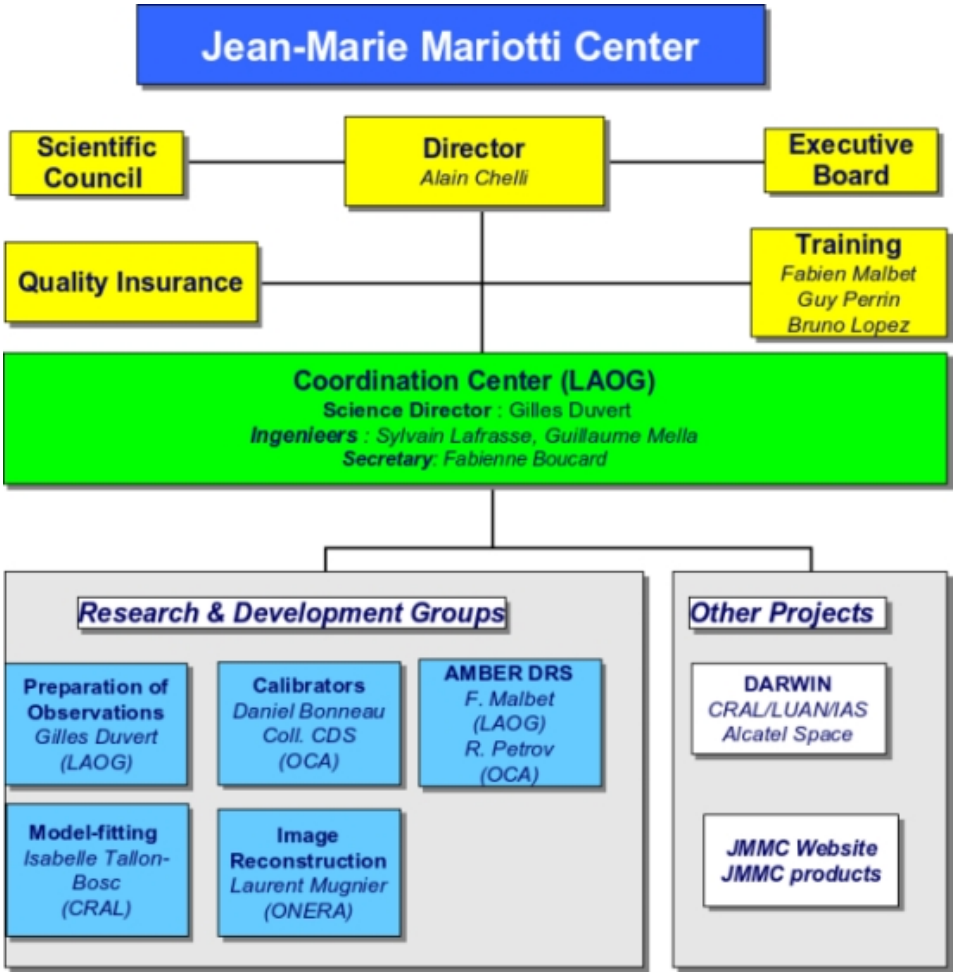
Je l'avoue, l'aspect d'assemblage me plait un peu moins. Cependant, je pense avoir acquis ce qui me manquait le plus. De la patience, la capacité à chercher quelque chose qui réponde à des besoins précis, l'adaptation et une vue plus "globale" d'une application.

4.3 Evolution professionnelle

Le Java et la programmation WEB m'intéressent tous deux autant l'un que l'autre. Cependant, dans les temps actuels, les possibilités qu'offre Internet me passionnent ! Je ressens un lien plus fort des gens sur Internet qu'à travers l'utilisation d'une application.

C'est pourquoi, je pense continuer dans une voie plus orientée WEB pour les années à venir s'il m'en est donné l'occasion.

A Organigramme du JMMC



B Code de l'application de tests

```
package fr.jmmc.test;

import fr.jmmc.mcs.gui.*;
import java.awt.BorderLayout;
import java.net.URL;
import java.util.logging.*;
import javax.swing.JButton;
import javax.swing.JFrame;

public class Main extends App
{
    /** Logger */
    private static final Logger _logger = Logger.getLogger(Main.class.
        getName());

    /** Frame */
    private JFrame _frame = null;

    /** Button to launch about box window */
    private JButton _aboutBoxButton = null;

    /** Button to launch feedback report window */
    private JButton _feedbackReportButton = null;

    /** Button to launch helpview window */
    private JButton _helpViewButton = null;

    /** Button to launch exit method */
    private JButton _exitButton = null;

    /** Constructor */
    public Main(String[] args)
    {
        super(args);
    }

    /** Initialize application objects */
    @Override
    protected void init()
    {
        _logger.warning("Initialize application objects");

        // Instantiate JFrame
        _frame                      = new JFrame("- App Test -");
```

```
        // .. buttons
        _aboutBoxButton             = new JButton(aboutBoxAction());
        _feedbackReportButton       = new JButton(feedbackReportAction())
            ;
        _helpViewButton             = new JButton(helpViewAction());
        _exitButton                 = new JButton(exitAction());

        // Set borderlayout
        _frame.getContentPane().setLayout(new BorderLayout());

        // Add buttons to panel
        _frame.getContentPane().add(_aboutBoxButton, BorderLayout.NORTH
            );
        _frame.getContentPane().add(_feedbackReportButton, BorderLayout
            .CENTER);
        _frame.getContentPane().add(_helpViewButton, BorderLayout.WEST)
            ;
        _frame.getContentPane().add(_exitButton, BorderLayout.SOUTH);
    }

    /** Execute application body */
    @Override
    protected void execute()
    {
        _logger.warning("Execute application body");

        // Set the frame properties
        _frame.pack();
        _frame.setLocationRelativeTo(null);
        _frame.setVisible(true);

        // we do something...
        try
        {
            Thread.sleep(5000);

            /* Our program is running and
               we wait the exitAction to execute
               last operations before closing application */
        }
        catch (Exception ex)
        {
            _logger.severe("Cannot wait 5000ms");
            ex.printStackTrace();
        }
```

```
    }

    /** Execute operations before closing application */
    @Override
    protected void exit()
    {
        _logger.warning("Execute operations before closing application"
            );

        System.exit(0);
    }

    /**
     * Main
     *
     * @param args command line arguments
     */
    public static void main(String[] args)
    {
        new Main(args);
    }
}
```

Webographie

1. Site du Laboratoire d'AstrOphysique :
http://www-laog.obs.ujf-grenoble.fr

2. Site du Centre Jean-Marie Mariotti :
http://www.jmmc.fr

3. Wiki sur le patron de conception Modèle-Vue-Contôleur :
http://fr.wikipedia.org/wiki/Mod\%C3\%A8le-Vue-Contr\%C3\%B4leur

4. Site de SUN Microsystems (on y trouve aussi la documentation API Java) :
http://www.sun.com

5. Wiki sur l'UML :
http://fr.wikipedia.org/wiki/Unified_ModÃÍleing_Language

6. Site de la librairie Castor (analyseur syntaxique) :
http://www.castor.org

7. Site de la librairie LogGui (utilitaire de logging Java) :
http://www.informatrix.ch/loggui

8. Site de la librairie HTTPClient (pour envoyer des mails depuis Java) :
http://hc.apache.org/httpclient-3.x/

9. Site de la librairie JXUL (pour créer des menus SWING en XML) :
http://jxul.sourceforge.net/

10. Système JavaHelp (système de génération et d'affichage de l'aide utilisateur) :
http://java.sun.com/javase/technologies/desktop/javahelp/

11. Librairie Getopt (interpréteur d'arguments) :
http://freshmeat.net/projects/getopt/

12. Framework AppFramework (JSR-296 sur les applications graphiques Java) :
https://appframework.dev.java.net/

13. JTidy (analyseur syntaxique) :
http://jtidy.sourceforge.net/

14. JHelpDev (utilitaire de création de fichiers TOC, JHM et HS) :
http://jhelpdev.sourceforge.net/

15. Doxygen (générateur de documentation) :
http://www.stack.nl/~dimitri/doxygen/

Résumé

Ce rapport de stage, de fin d'étude d'IUT Informatique, relate la création ainsi que l'intégration d'une bibliothèque facilitant la conception d'applications graphiques Java adaptées au Centre Jean-Marie Mariotti (Laboratoire d'AstrOphysique de Grenoble). Les principales étapes du développement ont été la réalisation du système de centralisation et d'accès aux données à travers l'utilisation du XML, la génération automatique du manuel d'aide utilisateur ainsi que des menus, l'uniformisation des processus comme le rapport d'erreur, le fait de sauvegarder du journal d'exécution et enfin l'affichage standardisé des fenêtres couramment utilisées telles que la fenêtre d'à-propos ou de démarrage. Cette Bibliothèque est notamment composée de diverses librairies sous licence GPL, et offre de puissantes fonctionnalités faciles à mettre en oeuvre.

Mots clefs : bibliothèque, application graphique, java, centralisation, manuel d'aide utilisateur, menus, uniformisation, processus, journal d'exécution, standardisation, fonctionnalité.

Abstract

This training report explains the creation and the use of a library designed to ease the developpement of Java graphical applications adapted to the Jean-Marie Mariotti Center (Laboratoire d'AstrOphysique de Grenoble). The main steps of the development were the realisation of the centralization system and the data access using XML, the automatical generation of the user manual and of menus, the process uniformization like feedback report, the fact to save the execution log and finaly the standardizated view of windows often used like aboutbox window or splashscreen. This library is composed of other libraries under the GPL licence, and provide powerful fonctionnalities while staying easy to use.

Keywords : library, graphical application, java, centralization, user manuel, menus, uniformization, process, execution log, standardization, fonctionnality.