Conception d'un éditeur d'annotations de documents textuels en bio-informatique

Hassiba Zidelkheir

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

CENTRE REGIONAL RHÔNE-ALPES

CENTRE D'ENSEIGNEMENT DE GRENOBLE

MEMOIRE

présenté par Hassiba Zidelkheir

en vue d'obtenir

LE DIPLÔME D'INGENIEUR C.N.A.M.

en INFORMATIQUE

Conception d'un éditeur d'annotations de documents textuels en bio-informatique

Soutenu le 30/06/2010

JURY

Président : M. Eric Gressier-Soudan

Membres : M. Jean-Pierre Giraudin
M. André Plisson
M. Mathias Voisin-Fradin
M. Gilles Bisson
M. Eric Gaussier

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

CENTRE REGIONAL RHÔNE-ALPES

CENTRE D'ENSEIGNEMENT DE GRENOBLE

MEMOIRE

présenté par Hassiba Zidelkheir

en vue d'obtenir

LE DIPLÔME D'INGENIEUR C.N.A.M.

en INFORMATIQUE

Conception d'un éditeur d'annotations de documents textuels en bio-informatique

Soutenu le 30/06/2010

Les travaux relatifs à ce mémoire ont été effectués au laboratoire LIEBNIZ sous la direction de M. Gilles Bisson

MEMOIRE D'INGENIEUR C.N.A.M. en INFORMATIQUE

Conception d'un éditeur d'annotations de documents textuels

Hassiba Zidelkheir **Grenoble, le 30/06/2010**

Résumé

Il s'agit de réaliser un éditeur permettant d'annoter interactivement tout ou partie d'un texte à l'aide de balises XML décrites dans une DTD. Contrairement à ce qui est couramment proposé dans ce type d'éditeur, l'utilisateur n'a pas à structurer directement le texte sous la forme d'un arbre XML. Il peut introduire les balises une à une dans l'ordre où il le souhaite en restant dans une optique "traitement de texte". Cet éditeur est principalement conçu pour être utilisé dans un contexte de bioinformatique, notamment par les annotateurs du SIB (Swiss Institute of Bioinformatics). Le logiciel est écrit en JAVA, il est portable sur Linux, MacOs X et Windows.

Le développement est réalisé sur la plate-forme Eclipse. Il utilise le parseur Xerces d'Apache pour la lecture des documents XML, la bibliothèque graphique Swing ainsi que les feuilles de styles de type CSS. Une démarche de conception en spirale a été adoptée et les spécifications ont été réalisées en UML.

Mots-clés : Annotation, DTD, XML, Editeur

Abstract

The objective is to realize an editor which enables to annotate with interactivity all or a part of a text document using XML tags described in a DTD. Unlike what is usually available in this type of editor, the user does not have to structure the text as a XML tree. The user can introduce tags one by one, the way he wants like in a standard word processing application. This editor will be mainly designed for a bio-informatics context, especially for the SIB (Swiss Institute of Bioinformatics) annotators. The software will be written in JAVA and can be used in the following platforms: Linux, MacOs X and Windows.

The design is made on Eclipse platform. It uses the parser Xerces from Apache to read the XML documents, the graphic library swing and CSS style sheets. The spiral conception has been chosen and the specifications have been designed in UML.

Keywords: Annotation, DTD, XML, Editor

A Djilali,
Lyna et Yanis

Remerciements

Je tiens tout d'abord à remercier toutes les personnes qui me font l'honneur de participer au jury de ce mémoire : merci donc à Monsieur Eric Gressier-Soudan, professeur au CNAM Paris ; à Monsieur Jean-Pierre Giraudin, professeur à l'Université Pierre Mendès France ainsi qu'à Monsieur André Plisson, Directeur du centre d'enseignement CNAM de Grenoble.

J'adresse principalement mes remerciements à Gilles Bisson, Chargé de recherche au CNRS, pour m'avoir proposé ce sujet de stage et de l'avoir dirigé. Ces précieux conseils, sa grande disponibilité et son aptitude à éclaircir des situations souvent compliquées m'ont permis de mener à bien ce travail. Je dois reconnaître qu'il m'a souvent poussée sur des sujets que je maîtrisais mal. J'avoue que par moment cela m'a déstabilisée, mais cette attitude a été très bénéfique dans la mesure où j'ai dû pousser mes raisonnements encore plus loin.

Je remercie Mme Mirta Gordon, responsable du service Liebniz, pour m'avoir accueillie dans son équipe et permis de réaliser ce mémoire d'ingénieur.

Le travail réalisé durant cette année s'est déroulé en étroite collaboration avec le SIB (Swiss Institute of Bioinformatics) et l'INRA (Institut National de la Recherche Agronomique) de Jouy en Josas avec qui j'ai eu plaisir à travailler.

Enfin, je n'oublie pas toutes les personnes qui m'ont soutenu et ont participé à ce travail. Frédéric, avec qui j'ai passé de nombreuses années sur les bancs du CUEFA, François pour ses bons conseils. Sans oublier, bien sûr tout le personnel du CUEFA de Grenoble, en passant par l'ensemble des professeurs qui m'ont fait découvrir un enseignement de qualité ainsi que l'ensemble du personnel administratif qui se sont toujours rendus disponibles.

J'adresse un très grand merci à tous mes proches qui n'ont jamais cessé de croire en moi. Ils ont su faire preuve de patience, de compréhension et d'écoute tout au long de ce parcours. Merci à mes parents, ma belle famille et à tous mes amis.

Et pour finir, je remercie ceux qui ont supporté mes absences répétées alors que je rédigeais ce mémoire : Dji, pour m'avoir encouragé et soutenu tout au long de ces années, mes deux petits amours, Lyna et Yanis, à qui je dédicace ce mémoire. Je vous aime.

Tables des matières

Index des figures

Conventions Typographiques

Le présent rapport applique certaines conventions typographiques, détaillées ci-dessous :

Convention	Signification
[ ]	Les références bibliographiques sont notées entre crochets et renvoient à la bibliographie en fin de rapport.
( )	Les acronymes sont indiqués entre parenthèses après la définition complète du terme qu'ils désignent.
Italique	L'Italique désigne des termes très importants ou nouveaux ainsi que les variables utilisées dans les programmes et les équations.
Note[1]	Les notes dans le texte sont repérées à l'aide d'un numéro figurant en exposant à la suite du terme qu'elles décrivent et sont affichées en bas de page.
`Pseudo-code`	Les algorithmes et fonctions en pseudo-code utilisent cette police de caractère.
Section (notée §)	Les sections forment une rubrique. Les numéros des sections démarrent à 1 et à chaque nouveau chapitre et sont précédés du numéro de la rubrique.

Abréviations

AJAX : Asynchronous JavaScript and XML

ANR : Agence Nationale de la Recherche

CADERIGE : Catégorisation Automatique de Documents pour l'Extraction de Réseaux d'Interactions GEniques

CADIXE : Caderige Interactif Xml Editor

CSS : Cascading Style Sheets

CVS : Concurrent Version System

DOM : Document Object Model

DTD : Document Type Definition

EI : Extraction d'Information

EN : Entité Nommée

HTML : Hyper Text Markup Language

IHM : Interface Homme-Machine

JDK : Java Development KIT

JVM : Java Virtual Machine

INRA : Institut National de la Recherche Agronomique

OOSE : Object Oriented Software Engineering

OMG : Object Management Group

MIG : Mathématiques, Informatique et Génome

PHP : Hypertext Preprocessor

RAD : Rapid Application Development

RI : Recherche d'Information

RTF : Rich Text Format

SAX : Simple API for XML

SGML : Standard Generalized Markup Language

SIB : Swiss Institute of Bioinformatics

UML : Unified Modeling Language

W3C : World Wide Web Consortium

XP : eXtreme Programming (XP)

XML : Extensible Markup Language

XPCOM : Cross-Platform Component Object Model

XSL : eXtensible Stylesheet Language

XUL : XML User Interface Language

Chapitre 1

INTRODUCTION

1 Contexte

Plusieurs communautés travaillent depuis de nombreuses années sur l'information textuelle, linguistes, statisticiens, informaticiens, avec des outils et des objectifs souvent très différents. Ce n'est que plus récemment que la communauté d'apprentissage artificiel (ou « Machine Learning »), dont le but est de concevoir des systèmes capables de s'adapter à leur contexte d'utilisation par induction de connaissances, a tenté d'appliquer ses outils à l'information textuelle. L'arrivée d'Internet et l'émergence du gigantesque ensemble de données qui lui est associé a constitué le coup de pouce qui a incité les chercheurs en apprentissage à investir ce domaine. Les chercheurs en apprentissage sont très souvent venus au texte en se penchant sur les problèmes soulevés par l'accès à l'information sur le web ou dans les entrepôts de données. Ils ont ainsi intégré leurs travaux dans les outils développés en recherche d'information (RI) et en extraction d'information (EI).

La recherche d'information (RI) s'intéresse au document d'un point de vue global : on cherche à retrouver, à partir d'une requête, ceux qui sont les plus pertinents au milieu d'une collection de documents. L'extraction d'information (EI) vise à collecter des informations spécifiques au sein de chaque document d'une collection afin de constituer, par exemple, une base de connaissances (carnet d'adresses, liste d'entreprise ou de personnes, etc). Pour accomplir une telle tâche, il faut être capable d'analyser le contenu des documents afin de reconnaître les éléments spécifiques présents dans le texte tels les entités nommées (EN) et de détecter les structures complexes comme la négation ou les anaphores. On doit souligner que la reconnaissance des entités nommées est une sous-tâche fondamentale de l'activité d'extraction d'information dans des corpus documentaires. Elle permet de modéliser correctement un texte en identifiant les objets textuels remarquables du domaine concerné (un mot, ou un groupe de mots) classés dans des catégories telles que : noms de personnes, noms d'organisations ou d'entreprises, noms de lieux, quantités, etc.

Deux grandes familles de méthodes sont utilisées pour reconnaître des EN :

- Une première méthode repose sur des « règles d'extraction », de type expression régulière, souvent associées à des modèles statistiques. Ces règles sont rédigées manuellement, elles obtiennent de bons résultats. L'inconvénient est que ce type de méthode requiert parfois des mois de travail de rédaction et que ces règles sont généralement assez spécifiques d'une collection de documents donnés.
- Une seconde méthode, dans laquelle il n'est plus nécessaire de rédiger de nombreuses règles à la main, vise à les faire apprendre automatiquement à l'aide d'un logiciel d'apprentissage. Il est toutefois nécessaire d'annoter préalablement un corpus qui sert de base d'entraînement pour le logiciel. Cette méthode, bien que plus générale que la précédente reste cependant elle aussi coûteuse en temps humain. Pour résoudre ce problème, des travaux de recherches ont vu le jour pour développer des outils d'aide à l'annotation. Le projet présenté ici qui consiste à développer un éditeur permettant l'annotation de document textuel en bio-informatique s'inscrit dans cette dynamique.

2 Articulation du document

Suite à ce chapitre introductif, nous structurons ce mémoire en 6 autres chapitres :

- Une présentation du projet Caderige (Catégorisation Automatique de Documents pour l'Extraction de Réseaux d'Interactions GEniques) et une introduction des concepts de base du langage XML dans le chapitre 2 ;
- Le chapitre 3 propose un état de l'art de plusieurs éditeurs XML et éditeurs d'annotations présents actuellement sur le marché. Notre étude a porté essentiellement sur les outils gratuits ayant la possibilité de rajouter des nouvelles fonctionnalités ;
- Le chapitre 4 est entièrement consacré à la méthodologie de développement logiciel et son langage associé. Le choix d'une méthode de développement est primordial pour garantir un logiciel de qualité ;
- L'étude et le recueil des besoins des biologistes sont détaillés dans la première partie du chapitre 5 « Besoin fonctionnel et analyse ». La seconde partie de ce chapitre est consacrée à l'analyse du processus d'annotation présenté sous forme de diagramme ;
- Nous détaillons la construction de notre éditeur dans le chapitre 6 intitulé « Conception et implémentation de l'éditeur ». Dans un premier temps nous verrons, l'organisation logicielle mise en place, puis nous verrons les différents composants de base de l'éditeur (l'interface, l'édition, etc.) Enfin nous aborderons l'implémentation de l'éditeur en présentant les modules les plus importants ;
- Le dernier chapitre « Conclusion et perspectives » résume l'apport essentiel de ce travail et présente les perspectives d'évolution possibles pour l'éditeur. Pour cela, nous verrons que depuis 2005 de nouveaux travaux d'annotations ont vu le jour. Nous verrons les nouvelles fonctionnalités à implémenter afin d'offrir aux utilisateurs une application plus complète. Nous terminerons ce chapitre et ce mémoire, en présentant brièvement les nouvelles possibilités de mise en œuvre en présentant des nouvelles bibliothèques de fonctions utilisées aujourd'hui par Apple, Mozilla ou Nokia.

Chapitre 2

PRESENTATION DE L'EXISTANT

Comme nous l'avons vu dans l'introduction, l'une des difficultés rencontrées dans la détection d'entités nommées est la constitution de corpus annoté. Ainsi, dans ce chapitre, nous verrons dans un premier temps comment le projet Caderige répond à cette problématique. Puis, dans un second temps, au vu de la richesse des contenus des corpus et l'importance de leurs structures, nous verrons comment le choix d'un langage d'annotation, basé sur le format XML (Extensible Markup Language) et ses technologies associées s'est d'emblée imposé.

1 Projet Caderige

1.1 Présentation du projet

Dans ce cadre, le projet CADERIGE vise un double objectif scientifique :

- Sur le plan informatique, il s'agit de développer de nouvelles techniques d'extraction de connaissance dans les bases documentaires écrites en langage naturel ;
- Sur le plan biologique, il s'agit d'appliquer ces techniques dans le domaine de la génomique fonctionnelle et plus spécifiquement sur celui de la modélisation des interactions géniques.

Aujourd'hui on constate que la majeure partie de la connaissance biologique est décrite sous la forme d'articles scientifiques. Pour en extraire des connaissances pertinentes, les approches de type RI sont insuffisantes dès lors qu'une compréhension profonde du texte est nécessaire. Il faut aller au delà et mettre en œuvre des méthodes d'EI plus complexes s'appuyant sur des ressources lexicales, syntaxiques et sémantiques spécifiques au domaine étudié. Ces ressources étant généralement difficiles et longues à acquérir, l'objectif de CADERIGE - et son aspect novateur - résident dans la définition et dans l'implémentation de techniques informatiques originales permettant une acquisition automatique ou semi-automatique de telles ressources à partir de corpus. Ainsi, CADERIGE vise à mettre en place une collaboration étroite entre des biologistes et des informaticiens respectivement spécialistes du traitement automatique de la langue, de l'EI et de l'apprentissage automatique.

L'extraction d'information mise en œuvre dans ce projet n'est pas basée, comme c'est souvent le cas, sur l'utilisation de scripts (PERL ou autres) écrits manuellement mais il s'agit plutôt d'apprendre (automatiquement ou semi-automatiquement) des règles d'extraction à partir d'ensembles d'exemples et de contre-exemples. Dans le cas de la génomique les exemples à traiter sont des phrases, extraites de bases bibliographiques, parlant d'interactions entre gènes et les contre-exemples des phrases n'en parlant pas.

Les règles d'extraction sont exprimées sous la forme de clauses PROLOG, voici un exemple d'une des règles apprises :

```
interaction (Y, Z) :-
protein(Y), gene(Z), interaction(V), subject(Y,V), Obj(U,V), NprepN(of)(Z,U).
```

Ce qui s'interprète comme :

> «Il y a une interaction entre les termes Y et Z si Y est une protéine et Z un gène et qu'il y a un terme V qui dénote une interaction avec Y étant le sujet de V et U son complément d'objet direct, Z et U étant lié par la préposition "of" »

Comme on le voit dans cette règle, l'apprentissage a pris en compte simultanément des informations de type lexicales, syntaxiques et sémantiques. Si les deux premiers types d'informations peuvent être engendrés automatiquement par des analyseurs lexicaux et syntaxiques ce n'est pas le cas des connaissances sémantiques, qui sont spécifiques au domaine de la génomique, et qui doivent donc être introduites manuellement par un expert du domaine.

A terme, l'objectif du projet CADERIGE est de développer et de valider deux catégories d'outils qui permettront aux biologistes d'interroger la base Medline[1] en langage naturel et aux bio-informaticiens d'acquérir, en amont, l'ensemble des connaissances nécessaires à cette consultation. La validation des résultats s'effectue principalement sur l'analyse de notices bibliographiques portant sur la transcription des gènes chez la bactérie modèle Bacillus subtilis[2]. Le projet s'est officiellement déroulé entre les années 2000 et 2003, mais les équipes qui ont participé à ce projet poursuivent le travail et c'est donc naturellement dans la continuation de ce projet que se déroule ce stage.

1 MEDLINE est une base de données bibliographique produite par la National Library of Medicine (US). Ces références proviennent d'environ 4500 revues publiées aux Etats-Unis et dans plus de 70 autres pays. Disponible en ligne (online) depuis 1971, MEDLINE inclut les références des articles indexés de 1966 à nos jours.

2 Bactérie du sol, non pathogène, Bacillus subtilis est un organisme de choix pour l'étude de la sécrétion protéique et un des outils génétiques les plus aisément manipulables avec la levure.

1.2 Description technique

1.2.1 Architecture générale

De manière plus technique, l'architecture du projet CADERIGE s'articule autour de deux grands modules (Figure 1) :

- D'une part, le module d'extraction d'information qui extrait les informations des fragments de textes pertinents et les formalise ;
- D'autre part, le module d'acquisition des ressources linguistiques qui permet lui d'apprendre ou de recueillir de manière semi-automatique l'ensemble des connaissances nécessaires à cette extraction.

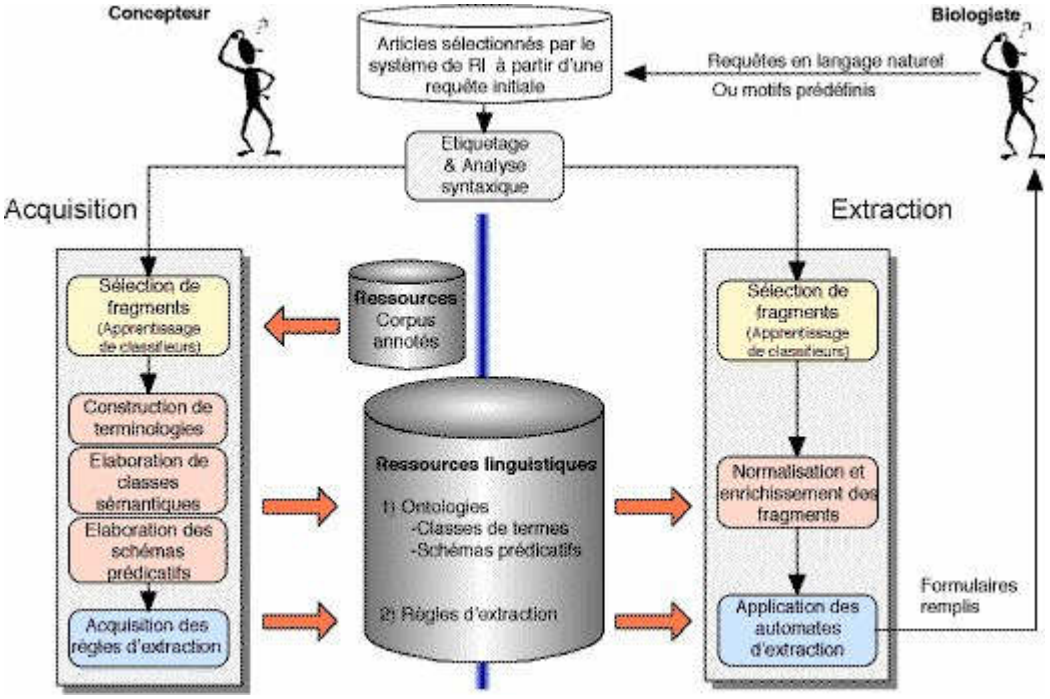


Figure 1 : Architecture des modules d'acquisition et d'extraction

1.2.2 Annotation des corpus

Dans le projet, l'acquisition des ressources linguistiques repose en partie sur des techniques d'apprentissage supervisées. Dans ce contexte, il est nécessaire qu'une partie des documents de travail soient préalablement annotés manuellement par les biologistes de manière à mettre en évidence les éléments (et relations) qui sont à apprendre.

1.2.2.1) Qu'entend-t-on par annotation ?

La lecture sur document papier n'est pas une activité passive : le lecteur travaille activement avec le texte afin de rendre explicite l'interprétation qu'il en fait. Par exemple, en surlignant et annotant des passages importants. Prendre des notes aide le lecteur à se construire une représentation interne du document.

En général, le lecteur surligne les passages importants devant être relus et mémorisés, et rature les parties inintéressantes. Ces annotations permettent de créer une version personnalisée du texte. Une autre fonction des annotations concerne l'interprétation du texte, une autre encore la restructuration du document, pour par exemple suggérer des modifications ou réordonner des parties. Sur les documents papier, les annotations sont généralement créées et relues par une seule personne. Elles peuvent parfois être créées pour d'autres lecteurs, par exemple pour recommander la lecture d'un document ou exprimer son point de vue dans des tâches de correction.

Les formes des annotations sont très diverses. La plupart font référence à une partie du document, par exemple un paragraphe, une phrase ou même un mot. La première caractéristique d'une annotation est donc de marquer une partie d'un document. Plusieurs techniques de marquage sont possibles, mais surligner et souligner restent les plus utilisées. Après avoir marqué une partie du texte, le lecteur ajoute éventuellement un commentaire sous forme d'icônes ou de textes. La fonction d'une annotation est parfois donnée par la forme de la marque. Les couleurs sont ainsi utilisées pour interpréter les parties marquées comme étant intéressantes, fausses, à revoir, etc.

Une annotation est composée de deux parties :

- une **ancre :** qui permet d'attacher l'annotation à une partie précise du document ;
- des **attributs :** facultatifs comme un commentaire.

Cette tâche d'annotation peut-être effectuée en introduisant des balises XML (Chapitre 2, §2.1) dans le corps du document. Par exemple, dans le domaine de la génomique, la phrase :

« GerE stimulates cotD transcription and cotA transcription […], and, unexpectedly, inhibits […] transcription of the gene (sigK) encoding sigmaK » pourra être annotée de la manière suivante

```
<sentence>
        The
        <agent type=protein>GerE </agent>
        …
        <interaction type=negative>inhibits </interaction>
        <target type=expression>transcription of
                the
                <source type=gene> gene (sigK) </source> encoding
                <product>sigmaK</product>
        </target>
</sentence>
```

Il est facile de constater que cette tâche d'annotation est relativement complexe. D'une part, le nombre de termes à introduire peut-être élevé comme on le voit ici. D'autre part, le langage d'annotation peut-être complexe, ainsi dans le cas du projet Caderige le langage développé comprenait une cinquantaine de balises certaines contenant jusqu'à 6 attributs. Il n'est donc pas simple de se souvenir et de manipuler l'ensemble des annotations possibles et l'utilisation d'un éditeur de texte classique n'est plus adaptée. Enfin, pour constituer l'ensemble des exemples et des contre-exemples, l'expert est amené à annoter plusieurs centaines de phrases ce qui représente une tâche assez lourde.

1.2.2.2) Naissance du projet Cadixe (Caderige Interactif XML Editor)

Dès lors la nécessité de disposer d'un éditeur spécialisé dans la gestion des annotations s'est imposée aux membres du projet. Or, s'il existe de nombreuses implémentations d'éditeurs XML, la plupart d'entre eux sont orientés vers la création, ou la modification de documents arborescents dont le format est constamment valide vis-à-vis de la DTD (Document Type Définition) (Chapitre 2, §2.3). Comme nous allons le voir cette contrainte est peu naturelle dans une tâche d'annotation. De manière synthétique, l'éditeur doit répondre aux critères suivants :

- L'insertion des balises doit pouvoir être effectuée dans n'importe quel ordre ;
- Le document annoté doit-être valide par rapport à la grammaire XML utilisée ;
- Le document doit apparaître sous la forme d'un texte et non pas d'une arborescence ;
- Il faut faciliter l'annotation collaborative ;
- La courbe d'apprentissage du logiciel doit être très rapide.

Le premier point est tout à fait central car lors de l'annotation d'un document, l'utilisateur interprète en « temps réel » sa signification. Par exemple dans le cas des interactions entre gènes, il repère dans un ordre quelconque les entités (gène, protéines), le type de l'interaction, les conditions expérimentales… Il est donc très difficile de lui imposer de rentrer les balises dans l'ordre où elles sont structurées dans la DTD, ainsi, le premier élément identifié dans un texte n'est jamais la racine de la DTD. Cette possibilité ne doit évidemment pas se faire au détriment de la validité du code XML produit : même si les balises sont introduites dans le désordre le document annoté doit rester conforme à la grammaire choisie.

Par ailleurs l'annotation étant faite par des experts qui ne sont pas, a priori, des informaticiens, le texte doit apparaître sous une forme classique, non arborescente, et l'utilisation de l'éditeur doit rester intuitive en reprenant la logique d'un outil classique comme le traitement de texte.

Enfin, dans certains cas le processus d'annotation est collaboratif (c'est le cas au SIB), il est donc nécessaire que chaque annotateur puisse laisser des notes expliquant à ses collègues les raisons qui l'on conduit à annoter de telle ou telle façon et à valider/infirmer une balise.

2 Le langage de balisage XML

XML (eXtensible Markup Language), version simplifiée de SGML (Standard Generalized Markup Language), est un langage de description et d'échange de documents structurés. Il permet de décrire la structure logique d'un document à l'aide d'un système de balises. Ce langage offre la possibilité de marquer les éléments qui composent la structure, ainsi que les relations entre ces éléments [MIC00].

En XML, à la différence de HTML (Hyper Text Markup Language), les balises et la sémantique qui leur est associée ne sont pas prédéfinies. Ainsi, l'auteur d'un document XML peut inventer librement les balises qui lui paraissent utiles pour indiquer les éléments de son document.

« Ce langage établit une passerelle entre, d'une part la richesse mais aussi la complexité de SGML et, d'autre part la pauvreté sémantique du langage HTML par exemple ». [BON00]

Enfin, XML distingue deux classes de documents :

- les documents "bien formés" qui obéissent aux règles du langage XML ;
- les documents "valides" qui d'une part sont "bien formés" et d'autre part obéissent à une structure type définie dans une DTD.

2.1 Structure d'un document XML

Tout document XML se compose :

- d'un prologue contenant différentes déclarations ;
- d'un arbre d'éléments ;
- de commentaires et d'instructions de traitements qui peuvent apparaître aussi bien dans le prologue que dans l'arbre des éléments.

2.1.1 Le prologue

La présence du prologue dans un document XML est facultative mais conseillée. Ce premier se compose de :

- la déclaration XML qui peut posséder trois attributs. L'attribut version (présence obligatoire) indique la version du langage XML, l'attribut encoding (présence facultative) indique le codage de caractères utilisé dans le document, et finalement l'attribut standalone (présence facultative) qui fixé à "no" indique que le document XML peut dépendre d'autres documents, fixé à "yes" indique que le document est indépendant. Voici un exemple d'une telle déclaration :

  ```
  <?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
  ```

- la déclaration de type de document. Si cette dernière apparaît le document auquel elle est associée doit respecter une structure type particulière définie dans une DTD. Ainsi, cette déclaration est indispensable dans les documents "valides" étant donné que ceux-ci font toujours référence à une DTD. Cette déclaration commence par : *< !DOCTYPE* suivi d'un nom qui peut être celui du document et qui doit être le nom de l'élément racine du document. Ensuite vient *SYSTEM* suivi du nom de la DTD qui doit être mis entre guillemets. Voici un exemple d'une telle déclaration :

  ```
  <!DOCTYPE text SYSTEM "basic.dtd">
  ```

2.1.2 L'arbre des éléments

Chaque élément d'un document XML se compose d'une balise ouvrante, d'un contenu et d'une balise fermante - <*balise*>contenu de la balise</*balise*>, à l'exception des éléments vides constitués d'une balise spécifique dont la syntaxe est la suivante <*balise/*>.

Le nom d'une balise est soumis à un certain nombre de règles :

- Il peut être constitué par des caractères alphanumériques ainsi que par les caractères le tiret-souligné, le signe moins et le point. Il ne doit pas contenir de caractères d'espacement et ne peuvent commencer par les lettres xml (ou XML ou Xml...). D'une manière générale, les noms d'éléments reflètent leur contenu et non la façon dont ceux-ci devraient être affichés ;

- Le langage XML étant sensible à la casse, les majuscules et les minuscules dans un nom d'élément ne sont pas équivalentes. Ainsi, les balises <auteur>, <Auteur> et <AUTEUR> sont différentes ;
- Chaque document XML est constitué d'une hiérarchie d'éléments qui forment une structure arborescente. Grâce à cette dernière, une relation du type "parents-enfant(s)" peut s'établir entre ces éléments. Un élément peut contenir d'autres éléments (les éléments fils), une chaîne de caractères (des données), un mélange des deux, ou encore être vide. Le premier élément d'un document XML, appelé l'élément racine, ne possède pas de parents. Il doit être unique et englober tous les autres éléments. En revanche, tout élément fils doit être complètement inclus dans son élément père, étant donné que XML n'admet pas de chevauchements d'éléments.
- Un élément peut posséder des attributs qui sont principalement utilisés pour préciser ou simplement décrire certaines propriétés de ce premier. Le nom de l'attribut, précédé d'un caractère d'espacement, suit le nom de l'élément. Chaque attribut est composé de son nom et de sa valeur (nom = "valeur"). Le premier est soumis aux mêmes règles que le nom de l'élément. En ce qui concerne la valeur de l'attribut, elle est constituée d'une chaîne de caractères toujours encadrée par des guillemets ou par des apostrophes simples. Voici l'exemple d'un attribut dont le nom est *type*, la valeur *place* et le nom de l'élément auquel celui-ci est attaché *name* :

```
<name type="place">Paris</name>
```

2.1.3 Les commentaires et les instructions de traitement

Des commentaires et des instructions de traitement peuvent être inclus dans un document XML. Ces premiers ont l'allure suivante :

```
<!-- Nom de fichier : mémoire.xml -->
```

Leur début est marqué par la chaîne <!— et leur fin par la chaîne -->. Le contenu d'un commentaire peut être constitué de n'importe quelle chaîne de caractères, à l'exception de la chaîne double tiret --. Les commentaires ne sont pas lus par les processeurs XML.

En revanche et à titre d'exemple, les instructions de traitement permettent d'attacher à un document XML des feuilles de styles CSS (Cascading Style Sheets) ou XSL (eXtensible Stylesheet Language) qui sont responsables de la réalisation physique d'un document XML.

Les feuilles de styles contiennent des règles de "mise en page" pour chaque type d'éléments apparaissant dans un document. Voici un exemple d'instruction de traitement :

```
<?xml-stylesheet type="text/css" href="FeuilledeStyle.css"?>
```

2.2 Les principaux atouts du langage XML

Les principaux atouts du langage XML sont :

- La lisibilité : aucune connaissance ne doit théoriquement être nécessaire pour comprendre le contenu d'un document XML ;
- Une structure arborescente : permettant de modéliser la majorité des problèmes informatiques ;
- L'universalité : les différents jeux de caractères sont pris en compte ;

- Le déploiement : il peut être facilement distribué par n'importe quel protocole à même de transporter du texte, comme http ;
- L'intégralité : un document XML est utilisable par toute application pourvue d'un parseur (c'est-à-dire d'un logiciel permettant d'analyser un code XML) ;
- La portabilité : un document XML doit pouvoir être utilisable dans tous les domaines d'applications.

2.3 Intérêt des modèles de document

La DTD est une structure type à laquelle obéit chaque document XML valide. Attachée à un document XML une DTD permet d'éviter de réinventer des structures complexes déjà disponibles. La DTD est donc réutilisable et peut s'appliquer à un type de document particulier. Ainsi, La DTD définit :

- le nom des éléments qui peuvent être utilisés ;
- le contenu de chaque élément ;
- combien de fois et dans quel ordre les éléments peuvent apparaître ;
- les attributs éventuels et leur valeur par défaut ;
- le nom des entités qui peuvent être utilisées.

2.3.1 Déclaration d'éléments

Tout élément d'un document XML doit être déclaré. La syntaxe de sa déclaration est la suivante :

```
< !ELEMENT nom-de-l'élément contenu-de-l'élément>
```

Cette déclaration définit un type d'éléments et associe un modèle de contenu à ce dernier. Le modèle de contenu peut autoriser à créer des éléments qui contiennent :

- un ou plusieurs éléments fils ;
- des données (chaîne de caractères) ;
- un mélange de données et d'éléments fils.

Finalement, ce modèle autorise également l'existence d'un élément au contenu libre et l'existence d'un élément vide. Nous allons à présent considérer chacun des cas.

2.3.1.1) Elément fils

Les éléments fils peuvent apparaître dans un ordre imposé ou libre. Lorsque l'ordre libre est utilisé les noms des éléments autorisés seront séparés par des barres verticales et non par des virgules.

```
<!ELEMENT TEXT (TITLE, SUB-TITLE, PARAGRAPH)>
```

L'exemple qui précède nous informe que tout élément TEXT devrait contenir un élément TITLE, un élément SUB-TITLE et un élément PARAGRAPH. Les éléments doivent apparaître dans l'ordre indiqué.

En revanche, l'exemple qui suit indique que les mêmes éléments peuvent apparaître dans n'importe quel ordre. A noter que cette structure n'est pas logique par rapport à la précédente.

```
<!ELEMENT TEXT (TITLE | SUB-TITLE| PARAGRAPH)>
```

Il est possible de faire suivre les noms des éléments fils par des indicateurs d'occurrence.

Symbole	Signification
P?	L'élément p peut apparaître zéro ou une fois
P*	L'élément p peut apparaître zéro ou plusieurs fois
P+	L'élément p peut apparaître une ou plusieurs fois
P	L'élément p peut apparaître uniquement une seule fois

2.3.1.2) Données

Un élément peut contenir des données (chaîne de caractères) qui seront lues par le processeur XML. Celles-ci sont indiquées par le mot #PCDATA, abréviation de Parsed Character Data.

```
<!ELEMENT TITRE (#PCDATA)>
```

2.3.1.3) Modèle mixte

Ce modèle autorise le mélange de données et d'éléments.

```
< ! ELEMENT p (#PCDATA | em |exposant | indice | renvoi)*>
```

Dans la déclaration d'un élément au contenu mixte, la suite de caractères #PCDATA doit apparaître en première position. Le caractère de répétition qui suit la déclaration doit être obligatoirement l'étoile "*". Cela signifie que chaque composant de l'élément n'est pas obligatoire mais qu'il peut apparaître n fois. Le texte et les autres composants peuvent apparaître dans n'importe quel ordre où ne pas apparaître du tout. Ce modèle est utilisé fréquemment étant donné qu'il offre beaucoup de combinaisons possibles d'un élément.

2.3.1.4) Contenu libre

Un élément peut également posséder un contenu libre, un contenu quelconque, que ce soit d'autres éléments ou des données. La déclaration d'un tel élément utilise le mot ANY et a l'allure suivante :

```
< !ELEMENT rapport ANY>
```

Comme le souligne [MIC00] le modèle de contenu ANY peut être utile lors de la création et de la mise au point d'une DTD complexe. Il permet de valider celle-ci avant de définir des structures un peu plus contraignantes.

Un élément peut être déclaré comme étant vide, ce qui sera indiqué par le mot EMPTY.

```
<!ELEMENT language EMPTY>
```

2.3.2 Déclaration d'attributs

La déclaration d'attributs permet de spécifier quels sont les attributs qui pourront ou devront être associés à un élément et d'indiquer éventuellement quelle sera la valeur par défaut d'un attribut. La syntaxe de cette déclaration est la suivante :

```
< !ATTLIST nom-de-l'élément nom-de-l'attribut type-d'attribut attribut-par-défaut>.
```

Les trois genres de types d'attributs sont :

- type chaîne ;
- type atomique ;
- type énuméré.

2.3.2.1) Type chaîne

A ce type d'attributs appartient CDATA qui signifie que la valeur d'attribut sera une chaîne de caractères prise littéralement.

```
<!ATTLIST annotated-document reference CDATA #REQUIRED>
```

Dans cet exemple, l'attribut *référence*, attaché à l'élément *annotated-document*, est déclaré comme une suite de caractères.

2.3.2.2) Type atomique

A ce type d'attributs appartiennent :

- **ID** qui indique que la valeur de l'attribut est un symbole commençant par une lettre et ne contenant que des lettres, des chiffres, ou les caractères « - _ : . ». *ID* indique également que la valeur de l'attribut doit être unique sur l'ensemble du document XML ce qui permet de définir des renvois à l'intérieur de celui-ci ;
- **IDREF ou IDREFS** qui permettent également de définir des renvois à l'intérieur du document XML ;
- **ENTITY ou ENTITIES** qui indiquent que la valeur de l'attribut peut prendre le nom d'une entité externe non XML (par exemple une image) ;
- **NMTOKEN ou NMTOKENS** (name token) qui indiquent que la valeur de l'attribut est un symbole commençant par une lettre et ne contenant que des lettres, des chiffres, ou les caractères « - _ : . ». Il indique également que la valeur est le plus souvent un seul mot et elle ne doit pas contenir d'espace blanc.

2.3.2.3) Type énuméré

Ce type d'attributs permet de spécifier la liste des valeurs d'attributs qu'il peut prendre dans un document. La déclaration de ce type d'attributs permet également de spécifier sa valeur par défaut.

```
<!ATTLIST pause type (long | medium | short) #IMPLIED>
```

Pour pouvoir indiquer une valeur par défaut il faut insérer celle-ci à la place de *#IMPLIED* entre les guillemets, ce qui donnerait :

- la **valeur** par défaut de l'attribut :

  ```
  <!ATTLIST pause type (long | medium | short) "short">
  ```

- **#REQUIRED** indiquant que l'attribut est obligatoire :

  ```
  <!ATTLIST annotated-document reference CDATA #REQUIRED>
  ```

- **#IMPLIED** indiquant que l'attribut est optionnel :

  ```
  <!ATTLIST annotated-document reference CDATA #IMPLIED>
  ```

- **#FIXED 'valeur'** indiquant que l'attribut prend toujours la même valeur :

  ```
  <!ATTLIST formulaire methode CDATA #FIXED 'ENVOI'>
  ```

2.3.3 Déclaration d'entités internes

Comme nous l'avons mentionné dans la section §2.3, les entités générales permettent de définir des éléments pouvant être substitués dans le corps du document XML (bien qu'ils soient définis au sein de la DTD et non du document XML lui-même). La syntaxe d'une entité générale est la suivante :

```
<!ENTITY nom_de_l_entite "Contenu de l'entite">
```

Il est par exemple possible de déclarer l'entité générale suivante dans la DTD :

```
<!ENTITY site "CUEFA">
```

Les entités définies dans la DTD peuvent ainsi être utilisées dans le code XML en les appelant avec la syntaxe suivante : `&nom_de_l_entite;`

2.4 Résumé sur les DTD

Les modèles de documents servent donc à définir la cohérence d'un ensemble de documents, lesquels peuvent être utilisés par n'importe quelle application informatique en ne se définissant que par rapport au modèle sous-tendu. Ceci permet évidemment de gagner beaucoup de temps, d'argent et de fiabilité. Pour résumer :

- Une DTD décrit la structure d'un ensemble de documents XML valides ;
- Les parseurs XML permettent de valider un document XML par rapport à une DTD;
- Une DTD n'est pas un document XML;
- Il existe d'autres langages pour la description d'un document XML : XML Schema (Chapitre 7, §4.1), Relax NG.

Nous avons choisi d'utiliser les DTD car elles sont aujourd'hui plus répandues et elles ont l'avantage d'être relativement simples à utiliser même si elles sont aussi un peu limitées. Les schémas permettent de décrire de façon plus précise encore la structure d'un document. Ils sont plus sophistiqués mais plus difficiles à manipuler. Un exemple de DTD et un extrait d'un fichier XML utilisant cette même structure sont fournis en annexe I.

Comme nous avons pu le voir lors de ce tour d'horizon, XML offre d'énormes possibilités quant à la création de documents structurés (ouvrages techniques, emails, code de programmes,…) sans inclure obligatoirement une déclaration figée de cette structure. L'indépendance des données d'une feuille XML de toute plate-forme et de format de mise en page est un grand facteur de pérennité. Etant donné que les données sont indépendantes de la mise en forme, plusieurs API [3](*Application Programmable Interface*) XML sont aujourd'hui disponibles et offrent aux programmeurs des méthodes pour la création, la visualisation et les données XML (vers des bases de données par exemple). XML est de plus en plus utilisé pour le développement d'applications dans divers domaines (commerce, service sur le web, finance, santé, …) et plusieurs grandes entreprises de l'informatique ont participées à sa standardisation (Adobe, HP, IBM, Microsoft, Netscape, Oracle, Sun, …). C'est pour toutes les raisons évoquées ci-dessus que l'éditeur que nous envisageons doit supporter le format XML.

3 Une API (Application Programmable Interface, traduisez « interface de programmation » ou « interface pour l'accès programmé aux applications) est un ensemble de fonctions permettant d'accéder aux services d'une application, par l'intermédiaire d'un langage de programmation.

Chapitre 3

ETAT DE L'ART

Il est tout à fait possible d'éditer un document XML avec un éditeur de texte standard. Cependant, il est beaucoup plus confortable, voire nécessaire, de disposer d'un outil permettant de valider, de connaître les éléments autorisés, de sélectionner du texte, d'annoter une partie du document, etc. Les éditeurs XML actuels permettent d'éditer tout document XML respectant une DTD. L'éditeur que nous souhaitons développer doit aller plus loin, et répondre à certaines caractéristiques que nous regroupons de la façon suivante :

- **Garder une logique d'utilisation proche de celle d'un traitement de texte**. Nos utilisateurs n'ayant aucune compétence en informatique mais une forte connaissance des éditeurs classiques, l'idée étant de leur offrir un éditeur simple à utiliser reprenant l'ensemble des fonctionnalités d'un éditeur de texte classique ;
- **Il doit s'appuyer sur des bases qui doivent être évolutives et durables.** Il doit être possible, sans toucher aux fonctionnalités internes de l'éditeur, de rajouter des fonctionnalités comme l'appel à des scripts externes par exemple ;
- **Il doit être capable de lire des documents déjà annotés.** Il doit être possible de récupérer d'anciens documents XML annotés. Nous ne pouvons pas demander à nos clients utilisateurs de ré-annoter l'ensemble de leurs fichiers.
- **Laisser l'utilisateur libre d'annoter le texte dans l'ordre où il souhaite le faire.** L'éditeur ne doit imposer aucune contrainte quant à l'annotation du texte. L'idée étant de rendre l'éditeur paramétrable selon les besoins des utilisateurs. Certains veulent être guidés dans l'application des annotations alors que d'autres veulent être complètement autonomes ;
- **Pouvoir automatiser les tâches répétitives**. Lorsque l'on annote un document, certaines tâches peuvent s'avérer répétitives, l'éditeur doit offrir des fonctions permettant de les automatiser. Par exemple, un script qui repère des formes syntaxiques particulières (nom de gène par exemple) et qui les annote automatiquement.
- **Générer des documents XML facilement exploitables**. L'éditeur recherché doit être capable de lire et de générer des documents XML ;
- **Etre adaptable aux besoins spécifiques de la bio-informatique**. Le besoin d'annotation est né dans un contexte biologique, c'est pour cette raison que l'éditeur doit répondre à leurs besoins. Cependant, la population des utilisateurs visés ne se résume pas qu'à des biologistes, il est donc important que l'application soit générique, utilisable dans différents domaines ;
- **Il est préférable que l'application soit portable sur différentes plates-formes.** Un pré-requis important pour ce type d'outils est de pouvoir l'utiliser sous un système Windows, Unix ou MacOs.

L'éditeur doit implémenter des fonctionnalités d'édition classique pour travailler sur le document, qui sont : des styles pour modifier la structure visuelle du document dans l'éditeur, le copier/coller, la recherche et le remplacement, la navigation à travers le document, etc.

Maintenant que nous savons qu'elles sont les fonctionnalités attendues par notre éditeur, nous avons orienté nos recherches vers les éditeurs XML et également vers des éditeurs d'annotations. Il existe un grand nombre d'éditeurs XML, mais seuls les éditeurs gratuits ont retenu notre attention.

1 Les éditeurs XML

Des éditeurs XML pouvant créer ou modifier un fichier XML, tout en respectant une DTD existent. Ces éditeurs sont divisés en deux familles :

- Les éditeurs XML, les plus simples, ne travaillent que sur l'arborescence et n'acceptent pas de style sur les balises. Ces éditeurs sont en général des *freewares* et servent à valider le document avec une DTD ;
- Les éditeurs XML qui acceptent les styles CSS sur les balises. L'avantage est que l'édition devient plus facile en se rapprochant des éditeurs WYSIWYG. CSS ajoute des styles sur les balises, des listes, une numérotation, l'affichage des images, le rendu de tableaux.

1.1 XMLMind XMLEditor – XXE

De nombreux éditeurs existent pour les documents XML, mais la plupart sont orientés structure. C'est-à-dire qu'ils présentent le fichier XML sous forme d'arbre, auquel on peut ajouter des éléments, définir les attributs, etc. Le texte est présenté et utilisé comme un élément, ce qui rend peu pratique l'utilisation de tels outils pour éditer un document contenant essentiellement du texte. XXE à l'avantage de fonctionner de la même manière qu'un traitement de texte. Il permet d'éditer des fichiers dépendant d'une DTD et plus particulièrement les fichiers DocBook.

DocBook est une DTD développée à l'origine par Hal Computer System et par l'éditeur de livres O'Reilly & Associates vers 1991. Le consortium Oasis en assure à présent le développement. Basée originellement sur le SGML, la DTD DocBook est à présent développée conjointement pour les langages SGML et XML.

L'avantage de la DTD DocBook est qu'elle est particulièrement adaptée à la rédaction de documentation technique et du fait que ce soit fondamentalement en XML, la pérennité des données est complètement assurée. En revanche ce qui est moins bien, c'est que DocBook n'est qu'une DTD. Il faut donc trouver les outils permettant de l'exploiter (éditeur, feuilles de styles, …). Cet éditeur XXE est donc adapté à la rédaction de documents DocBook.

Développée par des français, cette application java est utilisable sur Windows, Linux et Mac OS X. La Figure 2 représente l'interface de XXE.

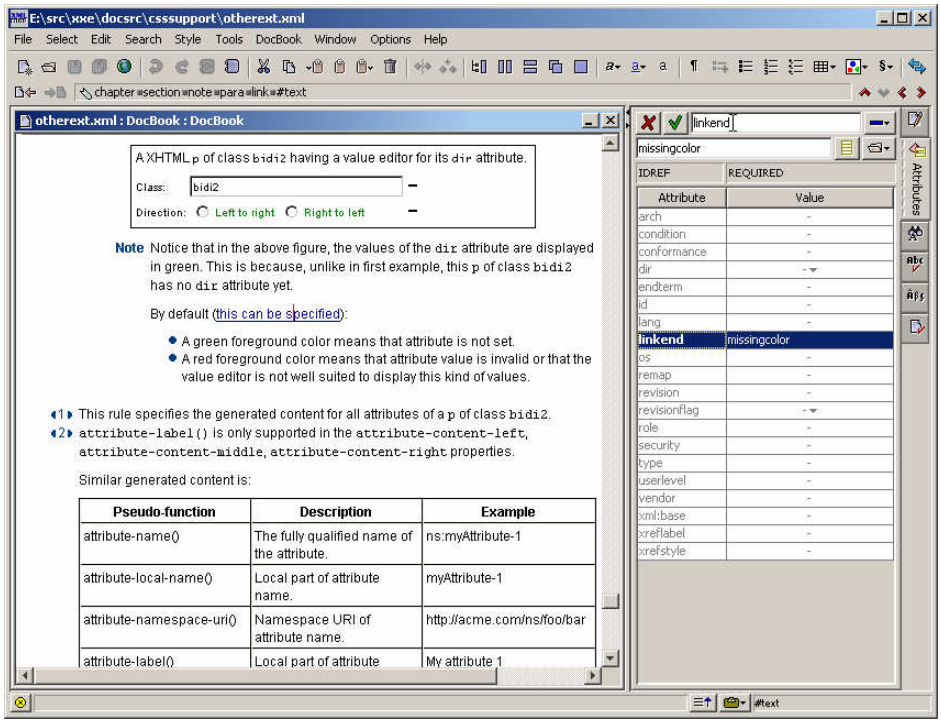


Figure 2 : Interface XXE

Cet éditeur propose quelques fonctionnalités particulièrement intéressantes. Comme par exemple, lorsqu'on insère une balise à un endroit, on se voit proposer celles qui sont valides dans le contexte. Cela est bien entendu réalisé grâce à la DTD. XXE offre également la possibilité de vérifier l'orthographe de ce qui est tapé.

On peut également à tout moment s'assurer que le document que l'on édite est toujours valide. Grâce à l'éditeur, il est difficilement possible de rendre un document invalide.

Les points ci-dessous reprennent les principales fonctionnalités de l'outil :

- Gestion des feuilles de styles (CSS2) ;
- Différentes vues synchronisées peuvent être utilisées pour éditer le même document ;
- Support des différents types d'images (GIF, JPEG, PNG, SVG, …) ;
- Paramétrable selon les besoins du client (menu, barre d'outil, raccourcis clavier, etc.) en utilisant des fichiers de configuration ;
- Possibilité d'appeler des scripts ou des commandes externes.

La solution XXE est très intéressante, on retrouve beaucoup de similitudes entre nos besoins et les fonctionnalités proposées. Cependant, malgré tous ces avantages, elle ne répond pas réellement aux besoins d'annotations mais donne la possibilité d'afficher des documents de manière lisible. Il existe une version gratuite de ce logiciel, mais cette dernière est incomplète.

1.2 Morphon

L'éditeur Morphon se compose de deux parties intégrales : un éditeur de code XML et l'éditeur CSS. Morphon permet aux utilisateurs de créer des documents XML valides et bien formés. Il aide donc l'utilisateur à écrire des documents selon le type de données choisies, rendant impossible l'écriture incorrecte du code XML.

Le CSS-Editor de Morphon permet aux utilisateurs de créer un modèle visuel pour XML en utilisant une interface graphique facile. CSS ajoute des styles sur les balises, les listes, la numérotation, l'affichage des images, le rendu des tableaux. Il offre de nombreux avantages :

- Un éditeur de texte presque WYSIWYG et une vue hiérarchique du document ;
- Il est possible d'éditer des tableaux et d'afficher des images JPEG et GIF lors de l'édition ;
- Il est entièrement écrit en Java 2 et est disponible sur beaucoup de plates-formes : Windows, Unix et Mac OS ;
- Il possède des avantages comme l'internationalisation et la transformation du document en utilisant XSL ;
- Il possède un correcteur orthographique.

2 Les éditeurs d'annotations

2.1 Palinka (Perspicuous and Adjustable Links Annotator)

Palinka reprend et généralise certains principes déjà utilisés dans CLinKa [PALI]. Alors que ce dernier était irréductiblement lié à la question de la coréférence, Palinka autorise la définition, par l'utilisateur, d'un schéma d'annotation. L'idée fondamentale de ce nouvel outil est d'offrir la possibilité de décomposer une tâche d'annotation en se basant sur trois types d'opérations :

- Insertion d'information non explicitement marquée dans le texte (par exemple points de suspension) ;
- Annotation d'éléments dans un texte (par exemple groupes nominaux, expressions, phrases) ;
- Indication des liens entre les éléments.

Palinka donne la possibilité d'insérer un nombre illimité de tags XML dans le document texte. Il est possible d'associer à chaque tag un nom ainsi que des valeurs d'attributs. Toutefois, pour chacun des tags, il est nécessaire de définir le type d'opérations qui leur est attaché. Par exemple, pour une information manquante l'outil insérera un marqueur dans le texte, alors que pour un lien, il demandera à l'annotateur de préciser l'élément référé. L'ensemble des tags qui peuvent être utilisés pour l'annotation est chargé à partir d'un fichier de préférences.

La figure suivante reprend une partie du fichier de préférences employé pour annoter le corpus. Il peut sembler compliqué pour un utilisateur de base, mais sa syntaxe se fonde sur un nombre limité de règles qui sont décrites dans la documentation.

```
[MARKER]
<EXP ID="#" COMMENT="">...</EXP>
NAME:EXP
BGCOLOR:23,255,254
FGCOLOR:123,111,10
ATTR:ID=# ;unique id
ATTR:COMMENT=!
INSERT_BEFORE:[
INSERT_AFTER:]
```

Figure 3 : Partie du fichier de préférence utilisé dans l'annotation du corpus de co-référence

Le fichier de préférences indique le genre d'annotation qui peut être appliqué à un texte. Il contient différents types d'informations. Par exemple, les informations relatives aux balises XML et leurs attributs peuvent être distinguées par :

- L'étiquette MARKER : définit une balise ;
- L'étiquette RELATION : définit une relation ;
- L'étiquette EMPTY : définit un élément manquant ;
- L'option NAME : définit le nom de la balise ;
- L'option ATTR et ATTROPT : définit la liste des attributs pour une balise.

Il contient également d'autres informations sur la façon dont les annotations seront affichées, par exemple FGCOLOR et BGCOLOR qui indique la couleur utilisée en arrière et premier plan pour une balise ou les options INSERT_BEFORE et INSERT_AFTER qui indique le caractère à utiliser avant et après une balise.

La Figure 4 représente l'écran principal de l'application Palinka. Comme on peut le voir, aucun tag XML n'apparaît dans le document, ainsi le texte peut être facilement lu. Afin d'identifier facilement les tags dans le texte, l'utilisateur peut indiquer des couleurs pour montrer les portions de textes annotés ou peut choisir des délimiteurs (dans l'exemple, le choix c'est porté sur des crochets).

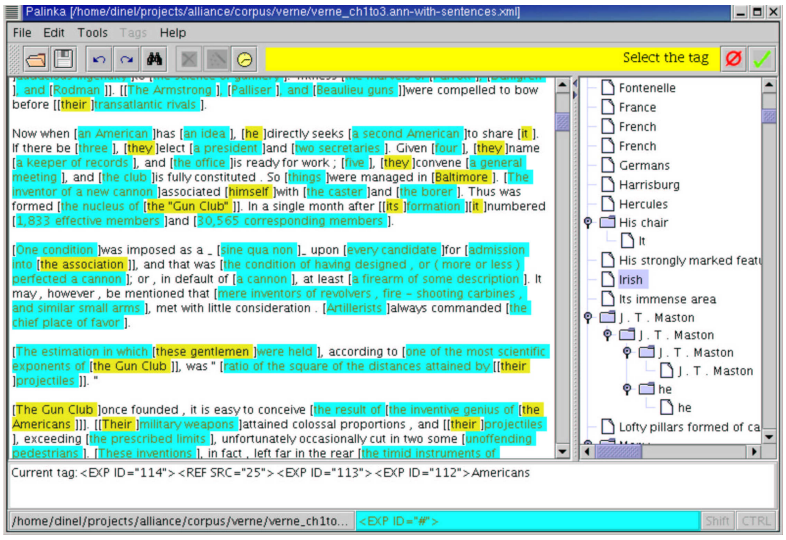


Figure 4 : Ecran principal de PalinKa

Palinka peut être utilisé pour annoter des fichiers contenant d'autres annotations. Cependant, si les annotations proviennent d'un autre fichier de préférences, ces dernières n'apparaîtront pas à l'écran. Le processus d'annotation a été défini de la façon la plus simple possible. De plus, l'outil reprend les fonctionnalités de base d'un outil d'annotations : faire/défaire, insertion, suppression et modifications de tags. Les fichiers générés par l'application sont au format XML, l'application s'assurant de l'intégrité du format et du respect des règles de construction du format XML. Palinka est implémenté en Java. Le programme peut être exécuté sur n'importe quelle plate-forme qui a une machine virtuelle Java installée.

Avantages :

- Interface facile d'utilisation ;
- Intègre les fonctionnalités d'un éditeur de texte classique ;
- Utilise le format XML en entrée et génère également un format XML ;

Inconvénients :

- Le principal inconvénient est que les tags sont inclus dans un fichier de préférences respectant une syntaxe interne à l'outil ;
- Limitation au niveau de l'annotation ;
- Difficilement adaptable au domaine de la bio-informatique ;
- Impossible d'ajouter des appels à des scripts externes ;
- Peu de documentation.

2.2 WordFreak

WordFreak est un outil d'annotations qui a été conçu entièrement en java. Ce qui le rend portable sur les plates-formes Linux, Windows, Mac OS X ainsi que sur Solaris. Son architecture modulaire facilite l'ajout de nouveaux composants. Ainsi, il est possible de l'adapter à des nouveaux besoins [WORD]. Aujourd'hui, WordFreak peut être utilisé pour annoter des textes en anglais, chinois et également en arabe. L'absence d'évolution récente de cet outil pose cependant la question de sa pérennité.

2.3 System Clark

System CLaRK est un logiciel développé par le BulTreeBank project et soutenu par le Tubingen-Soa International Graduate Programme en Computational Linguistics and Represented Knowledge (ClaRK). Ce logiciel bulgare est téléchargeable gratuitement pour la Recherche, il est écrit en java. Par contre il est propriétaire et on n'a donc pas accès aux sources. CLaRK System est un logiciel XML de traitement de corpus implémenté en Java. Ce système permet notamment le balisage de corpus en XML, l'interrogation du corpus à l'aide de Xpath[4] ou d'expressions régulières mais peut également servir de concordancier [CLAR]. Ce logiciel offre de nombreuses possibilités, cependant il est difficile d'annoter facilement les documents selon nos besoins.

4 XPath est un langage de requêtes non XML utilisé pour adresser des items dans un document XML.

2.4 XMLJava

XMLJava est un prototype qui a été développé dans le cadre du projet Caderige. Dans la mesure où les documents doivent être analysés manuellement par des biologistes pour constituer des données d'apprentissage de la sélection de fragments, et de l'apprentissage de règles d'extraction, il est nécessaire d'outiller ce travail d'annotation manuelle. C'est le rôle de l'éditeur XMLJava.

Le biologiste doit charger dans l'éditeur XMLJava les fragments analysés qu'il souhaite annoter, choisir une DTD spécifiant la grammaire des annotations et le guide d'annotations associé qui précise les consignes d'annotations pour la DTD. Il ajoute aux fragments analysés une couche d'annotation biologique.

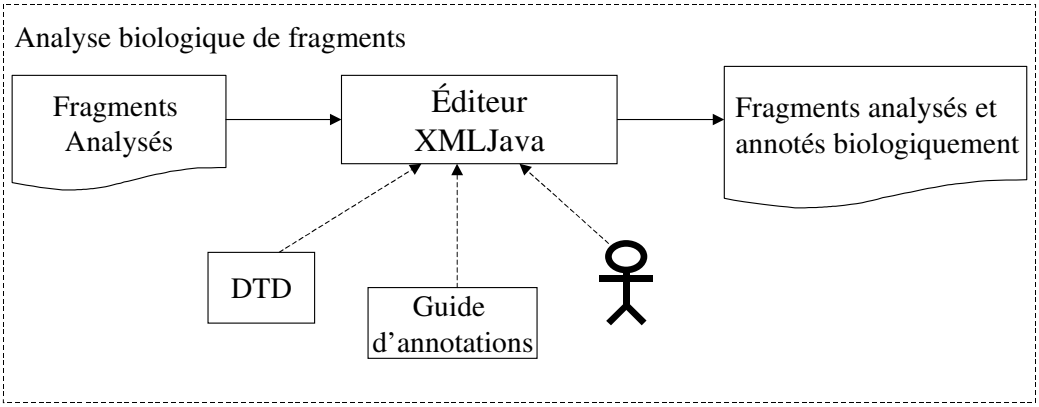


Figure 5 : Module d'annotation biologique

XMLJava est donc un éditeur XML (version très limitée) qui permet d'annoter de manière interactive un texte en génomique à l'aide de balises XML décrites dans une DTD. Les zones de textes balisées sont visualisées à l'aide de feuilles de styles librement éditables.

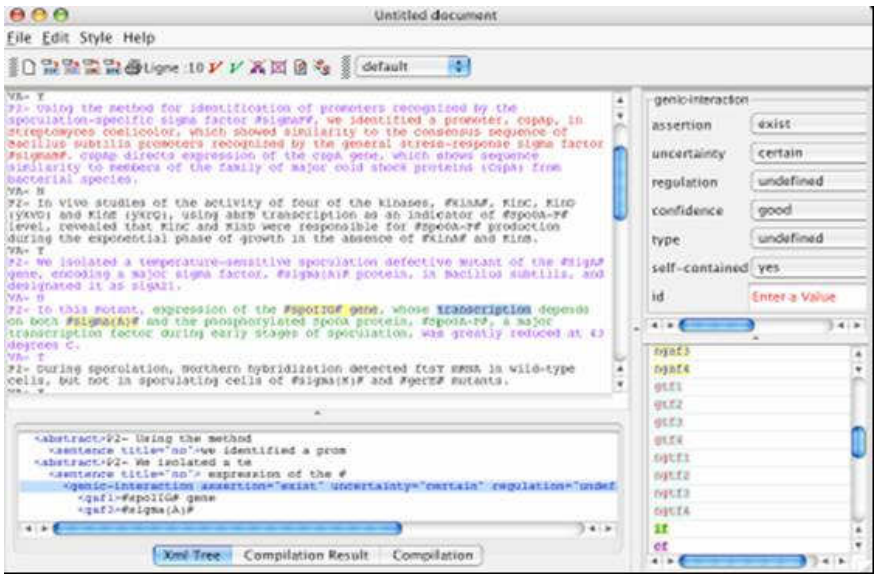


Figure 6 : Interface principale de l'éditeur XMLjava

Cet éditeur est aujourd'hui utilisé par le SIB à Genève et l'INRA de Jouy en Josas - Unité MIG (Mathématique, Informatique et Génome). Cependant, il est très difficile voire impossible de le maintenir ou de le faire évoluer.

En effet, il a été développé sous la contrainte temps en partant d'un cahier des charges en perpétuelle évolution. Il fait partie des développements que l'on peut classer dans la catégorie "Quick and Dirty" :

- Code très complexe et mal structuré ;
- Utilisation de structures de données mal adaptées au problème ;
- Nombreux bugs récurrents occasionnés par des « effets de bords » dans le code ;
- Peu évolutif et fonctionnalités manquantes (balisage limité, pas de collaboration, …).

3 Etat de l'art : Bilan

La confrontation entre ces différents travaux et outils et les besoins présentés auparavant, il en ressort distinctement que ces dernières ne sont pas pleinement satisfaites. Les éditeurs XML n'offrent pas la possibilité d'annoter le texte aisément et nous avons noté les inconvénients suivants :

- le plus souvent ces éditeurs sont des produits commerciaux. Le code source n'étant pas disponible, le seul moyen d'ajouter de nouvelles fonctionnalités est les plug-ins[5] ;
- tous ces éditeurs ne travaillent qu'avec des documents bien formés et valides. À chaque fois que l'on veut insérer une nouvelle balise, l'éditeur propose toutes les balises possibles. Par exemple, dans la DTD développée pour les besoins de la biologie, lorsque l'utilisateur est en train d'annoter une partie (fragment) de phrase, l'éditeur ne doit proposer que les balises portant sur l'annotation des gènes et des annotations, les autres balises portant sur les annotations de paragraphes doivent être masquées.

Notre choix s'est donc tourné vers la construction entière d'un éditeur. Il existe plusieurs façons de construire un éditeur, nous verrons dans les chapitres suivants les outils mis en œuvre pour la construction de notre éditeur. D'un point de vue fonctionnel, le futur éditeur reposera sur les fonctionnalités d'XMLJava qui constitue notre cahier des charges initial.

[5] Les plug-ins sont des modules qui complètent un logiciel hôte pour lui apporter de nouvelles fonctionnalités.

Chapitre 4

METHODOLOGIE DE DEVELOPPEMENT LOGICIEL

La recherche d'un éditeur d'annotations XML adapté à nos besoins s'est révélée peu fructueuse. Parmi les outils existants sur le marché, seul XMLJava est susceptible de nous intéresser, les autres éditeurs se sont avérés inadaptés aux besoins. En effet, l'éditeur recherché devait répondre à un certain nombre de caractéristiques techniques et fonctionnelles. Il fallait avant tout une interface de travail ergonomique, permettant une prise en main rapide sans un lourd apprentissage. Donc, notre choix s'est porté sur une partie des fonctionnalités d'XMLJava, qui devient le point d'entrée (cahier des charges initial) de ce mémoire. L'arrivée de nouveaux utilisateurs issus d'un domaine différent à celui de la biologie, a introduit de nouveaux besoins qui viennent élargir notre offre. Pour garantir la qualité de notre éditeur nommé Cadixe et les possibilités d'évolutions ultérieures, nous avons choisi de suivre une méthode de développement logiciel.

En génie logiciel, il existe de très nombreuses méthodes de développement d'un projet informatique, nous allons rapidement présenter les principales approches [DLOG]. Nous passerons en revue ces différents processus. Nous expliquerons également notre choix et verrons pourquoi la méthode choisie répond fortement à notre besoin puis le langage UML sur lequel elle se base. Dans le dernier paragraphe de ce chapitre, nous établissons un parallèle entre la méthode choisie au laboratoire Liebniz et la logique de construction des chapitres de ce mémoire.

1 Les méthodes de développement logiciel

Un *processus de développement logiciel* définit une séquence d'activités ordonnées qui concourent à l'obtention d'un système logiciel ou à l'évolution d'un système existant. L'objectif d'un tel processus est de produire des logiciels de qualité qui répondent aux besoins de leurs utilisateurs dans des temps et des coûts prévisibles.

Un *processus unifié* est un processus de développement logiciel construit sur le langage UML [JAC00], qui regroupe les activités à mener pour transformer les besoins d'un utilisateur en un système logiciel comme le montre la Figure 7.

Figure 7 : Processus de développement logiciel

1.1 Le modèle en cascade

Le principe du modèle en cascade est de découper le projet en phases distinctes sur le principe du non-retour. Lorsqu'une phase est achevée, son résultat sert de point d'entrée à la phase suivante. Ce modèle, développé dans les années 1970 par W. ROYCE a servi pendant des années de modèle de référence.

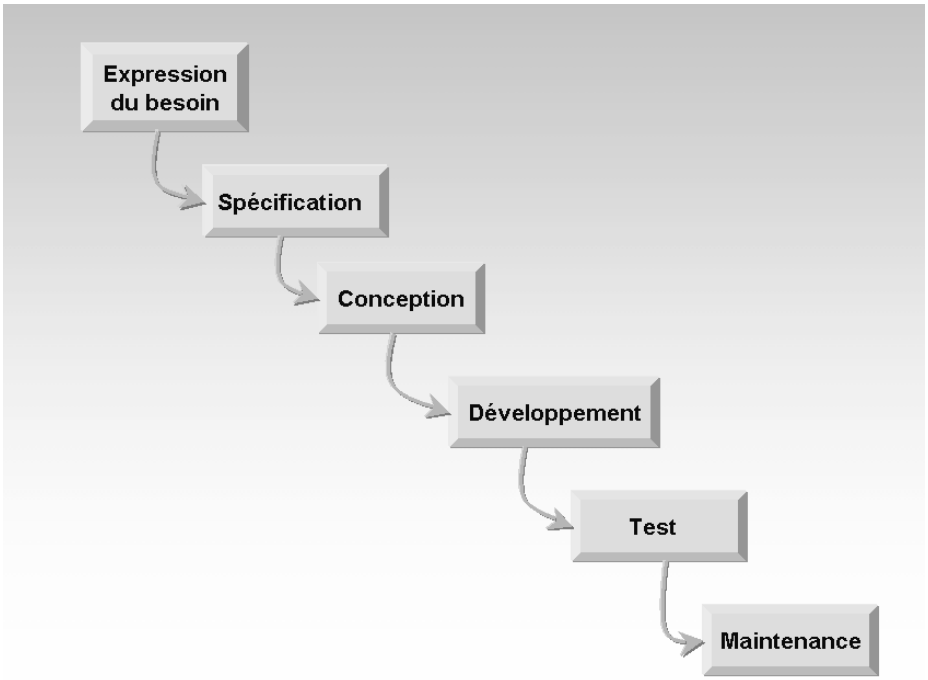


Figure 8 : Le modèle de développement en cascade

L'avantage de ce modèle est de proposer au fur et à mesure une démarche de réduction des risques, en minimisant au fur et à mesure l'impact des incertitudes. L'impact d'une incertitude dans la phase de développement étant plus faible que l'impact d'une incertitude dans les phases de Conception ou de Spécification, plus le projet avance, plus les risques diminuent.

Néanmoins, cette démarche, basée sur un processus de contrôle qualité en fin de chaque phase, a l'inconvénient d'exclure l'utilisateur dès la phase de conception car trop technique. Le contrôle qualité significatif survient alors en fin de projet, et, à ce moment, si l'utilisateur s'aperçoit que le système ne répond pas correctement aux besoins exprimés, il peut être trop tard.

1.2 Le modèle en V

L'assurance qualité est le processus qui permet de vérifier en continu la qualité du produit au fur et à mesure de sa fabrication. Le modèle en V met l'accent sur ce processus. Il confronte les différents niveaux de test avec les phases de projet de même niveau.

Ceci permet à chaque étape de définir non seulement les fonctions, mais également les critères de validation. La cohérence entre les deux éléments permet de vérifier en continu que le projet progresse vers un produit répondant aux besoins initiaux.

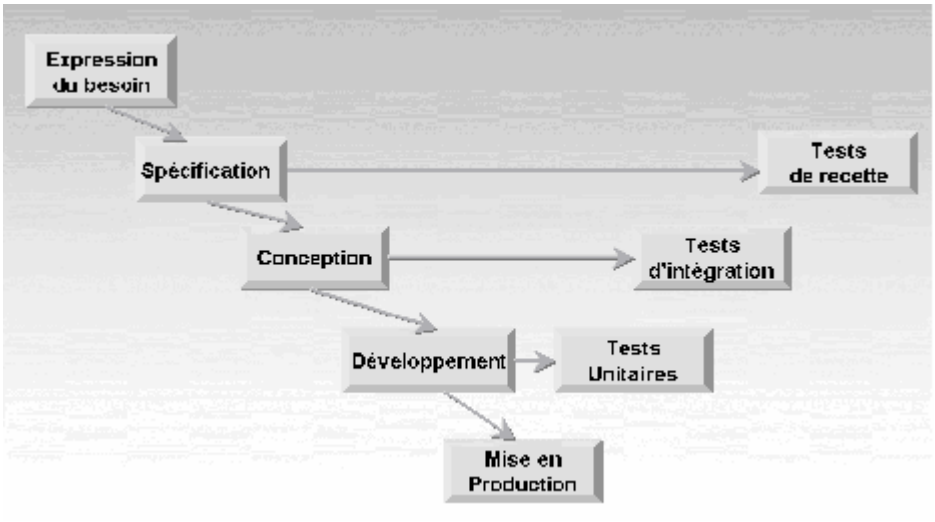


Figure 9 : Le modèle de développement en V

Ce modèle est adapté aux projets de taille et de complexité moyenne. C'est une amélioration du modèle en cascade traditionnel. Il permet d'identifier et d'anticiper très tôt les éventuelles évolutions des besoins. C'est aussi un moyen de vérifier la maturité des utilisateurs, car s'il en était autrement, ils se trouveraient dans l'incapacité de fournir des tests de recettes dès la phase de spécification.

1.3 Le modèle par incrément

Dans les modèles précédents un logiciel est décomposé en composants développés séparément et intégrés à la fin du processus.

Dans les modèles par incrément un seul ensemble de composants est développé à la fois : des incréments viennent s'intégrer à un *noyau de logiciel* développé au préalable. Chaque incrément est développé selon l'un des modèles précédents.

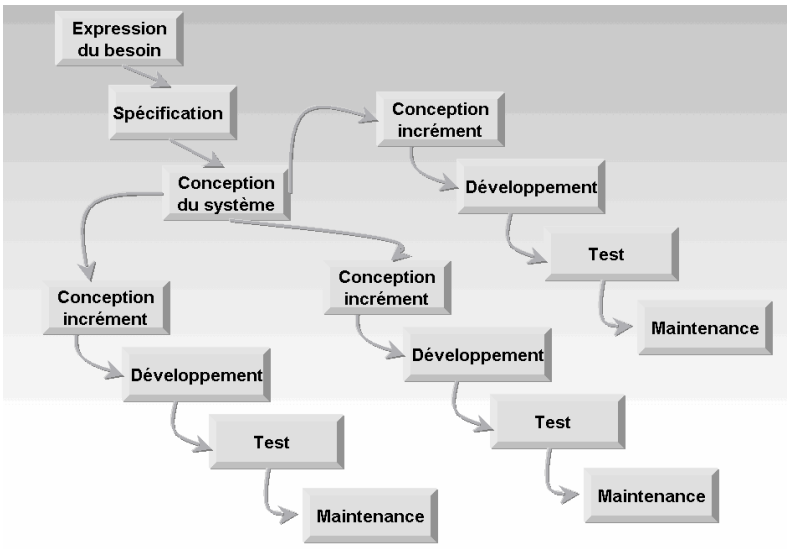


Figure 10 : Le modèle de développement par incrément

C'est un modèle adapté aux grands projets. Néanmoins, l'architecture du système doit permettre de définir des domaines suffisamment découplés. Dans le cas contraire, certains incréments doivent attendre que les incréments avec lesquels ils sont liés, soient suffisamment développés.

1.4 Le modèle en spirale

Le cycle de vie en spirale est un modèle générique de cycle de vie évolutif qui a été proposé par Barry W. Boehm en 1984. Ce modèle, axé sur la maîtrise et la réduction des risques, est davantage un cadre de travail guidant la construction d'une démarche spécifique de projet, plutôt qu'une démarche formalisée.

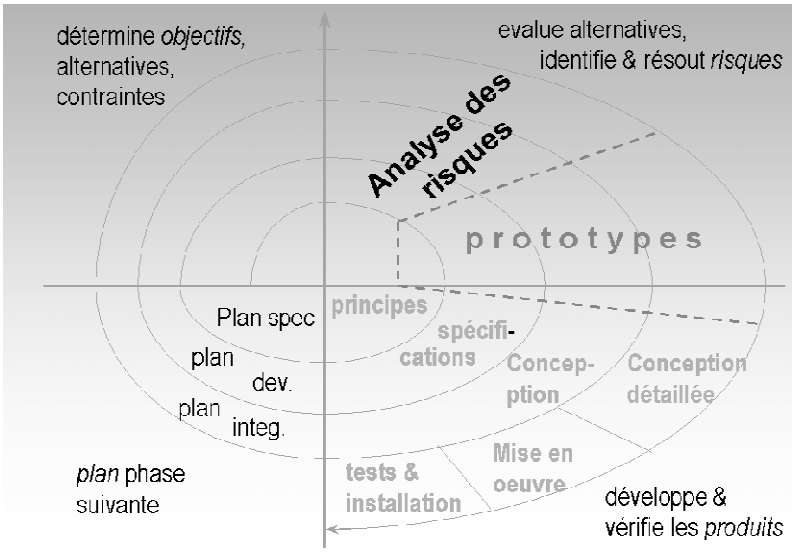


Figure 11 : Le modèle de développement en spirale

Le modèle en spirale :

- Est un modèle itératif, chaque cycle produit une version opérationnelle du logiciel;
- Combine les meilleurs aspects du modèle classique et du modèle par prototypage;
- Introduit la notion d'analyse de risques;
- Nécessite peu de gens au début et introduit plus de gens lorsque le projet évolue.

Il se décompose en quatre phases principales :

- Une phase d'identification: qui permet d'identifier les besoins, déterminer les objectifs, déterminer des alternatives pour atteindre les objectifs, déterminer les contraintes;
- Une phase d'évaluation : qui permet d'analyser les éventuels risques, évaluer des alternatives, identifier et résoudre ces risques;
- Une phase de réalisation : qui permet le développer et de vérifier la solution retenue;
- Une phase de vérification : qui permet de vérifier, valider le produit élaboré dans la phase de réalisation et planifier la prochaine phase.

1.5 Les méthodes agiles

Les méthodes de développement dites « **méthodes agiles** » (en anglais *Agile Modeling*, noté *AG*) visent à réduire le cycle de vie du logiciel (donc accélérer son développement) en développant une version minimale, puis en intégrant les fonctionnalités par un processus itératif basé sur une écoute client et des tests tout au long du cycle de développement.

Les quatre grandes valeurs du Développement Agile sont :

- Méthodes centrées sur les hommes : Personnes et interactions plutôt que processus et outils ;
- Méthodes centrées sur une application qui fonctionne : Logiciel fonctionnel plutôt que documentation complète ;
- Méthodes centrées sur la collaboration avec le client : Collaboration avec le client plutôt que négociation de contrat ;
- Méthodes centrées sur l'acceptation du changement : Réagir au changement plutôt que suivre un plan.

Il existe de nombreuses méthodes Agiles, on ne citera que eXtreme Programming (XP), RAD (*Rapid Application Development).*

L'inconvénient le plus connu vient de leur avantage principal : la réactivité et l'absence de prédéfinition (surtout dans XP). Si le client ne sait pas vraiment ce qu'il veut, ou ne sait pas le définir, on se retrouve avec beaucoup d'allers-retours. Cet inconvénient se ressent également dans la documentation. Si l'on souhaite garder la réactivité (autre avantage de ces méthodes), à quoi bon documenter ce qui sera obsolète dans quelques jours.

1.6 Notre choix de méthode

La méthode mise en place est proche de la méthode en spirale. Le terme "proche" est volontairement utilisé car les risques étant minimes, la phase d'évaluation a été occultée. Cette méthode permet de prendre en compte les attentes des utilisateurs. Contrairement aux autres méthodes, l'analyse des besoins n'est pas figée lors de la première étape de développement, mais progressivement affinée au fur et à mesure des cycles de développement. Les utilisateurs sont confrontés le plus tôt possible aux prototypes développés. Ils collaborent tout au long du développement du logiciel, y compris, tout au moins idéalement, dans la rédaction des notices d'emploi. Ainsi, développer une application en restant au plus près d'une utilisation réelle de par les expérimentations successives permet de mieux tenir compte des usages de cette application.

2 Le language UML

Le langage UML (Unified Modeling Language que l'on peut traduire par "*langage de modélisation unifié*) est un langage de spécification, de représentation graphique et de modélisation des systèmes logiciels. Il est né de la fusion de trois méthodes orientées objet Booch, OMT (Object Modeling Technique) et OOSE (Object Oriented Software Engineering), conçues respectivement par Grady Booch, James Rumbaugh et Ivar Jacobson. UML a été normalisé en 1997 par l'organisme de normalisation industrielle OMG[6] (Object Management Group) [OMG] et a été depuis sujet à plusieurs améliorations.

C'est aujourd'hui un outil de communication standardisé qui propose un nombre important de diagrammes. Il est important de souligner qu'UML est une notation et n'est pas une méthode ni un processus de développement. A ce titre, il ne suffit pas de connaître et comprendre la syntaxe et la sémantique des diagrammes pour être à même de les utiliser correctement.

Les diagrammes UML utilisés pour mettre en place l'application Cadixe sont les diagrammes de cas d'utilisation, les diagrammes d'activités et les diagrammes de classes. Ils sont introduits dans les sections suivantes.

2.1 Le diagramme de cas d'utilisation

Le cas d'utilisation décrit un système en privilégiant le point de vue de l'utilisateur. Composé d'un ensemble d'actions, il est déclenché par un *acteur* externe et produit un résultat identifiable. Il définit les limites du système et ses relations vis-à-vis de son environnement. Les cas d'utilisation permettent aux utilisateurs de structurer et d'articuler leurs besoins ; ils les obligent à définir la manière dont ils souhaitent interagir avec le système. Les cas d'utilisation sont organisés dans des diagrammes de cas d'utilisation, qui montrent notamment les relations qui existent entre les cas d'utilisation et les acteurs comme l'illustre la figure ci-dessous où les limites du système Cadixe sont repérées au moyen d'un rectangle. L'acteur "Utilisateur" interagit avec le cas d'utilisation "Annoter le document".

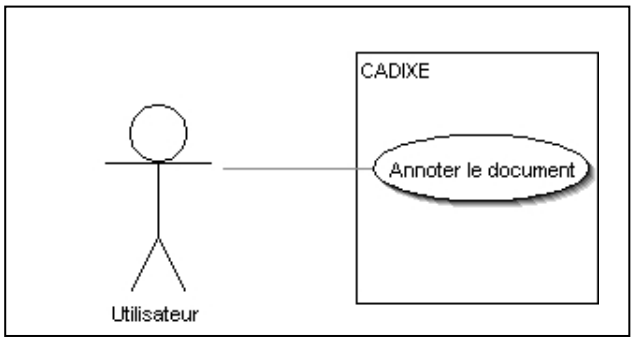


Figure 12 : Exemple de diagramme de cas d'utilisation

Dans le travail rapporté ici, les diagrammes de cas d'utilisation ont été utilisés pour capturer des besoins fonctionnels.

6 OMG : Object Management Group est une association américaine à but non-lucratif créée en 1989 à l'initiative de grande société ((HP, Sun, Unisys, American Airlines, Philips...) dont l'objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes.

2.2 Le diagramme de classes

Un diagramme de classes représente la structure statique d'un système et contient un ensemble de classes avec leurs attributs et leurs méthodes. En pratique, son intérêt majeur est qu'il modélise les entités du système d'information. Ainsi, toutes les informations mémorisées, manipulées, transformées et analysées pour accomplir les finalités du domaine figurent quelque part dans le diagramme de classes. Pour décrire une classe, la nécessité de faire référence à un attribut d'une autre classe conduit à introduire la notion d'*association*, qui est une relation particulière entre les deux classes.

Cette relation porte un nom qui traduit généralement sa signification comme l'illustre la Figure 13 suivante.

Figure 13 : Exemple d'association entre classes. Le diagramme d'activités

Les classes Tag et Style sont reliées au moyen de l'association contient ; on dit alors qu'un Tag contient un style et l'on écrit contient (Tag, Style)

Le diagramme d'activités schématise la dynamique du système d'information et est attaché à une classe, à un cas d'utilisation. C'est un graphe orienté qui décrit un enchaînement de traitements. Un exemple est présenté dans la Figure 14.

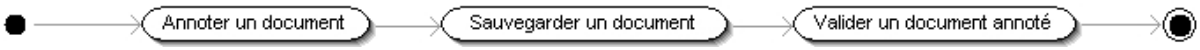


Figure 14 : Exemple de diagramme d'activités.

Les activités "Annoter un document" et "Valider un document annoté" sont reliées par une transition ; le point initial, représenté par un disque noir, précède l'activité " Annoter un document " et le point final, représenté par un disque noir entouré d'un cercle, succède à " Valider un document annoté ".

L'apport du diagramme d'activités est essentiel pour modéliser les processus car il permet de représenter aussi bien les traitements à effectuer que les acteurs impliqués et l'utilisation des informations.

2.3 Mise en œuvre au laboratoire LIEBNIZ

L'adoption du processus de développement en spirale et celle du langage UML constituent des éléments de réussite de ce projet. Le suivi de cette procédure de conduite de projet a été productif à plusieurs points de vue.

En effet, cette méthode nous a permis d'instaurer un réel dialogue avec les biologistes du SIB, ce qui a contribué à une meilleure compréhension des besoins des utilisateurs.

Pour réaliser ce travail, nous avons adapté le processus de développement en spirale en insistant sur les aspects cahier des charges et analyse, ce qui a permis d'être disponible et très réactif pour modifier le modèle métier du projet et les fonctionnalités à intégrer à Cadixe.

La Figure 15 présente les activités que nous avons suivies et situe les deux chapitres suivants.

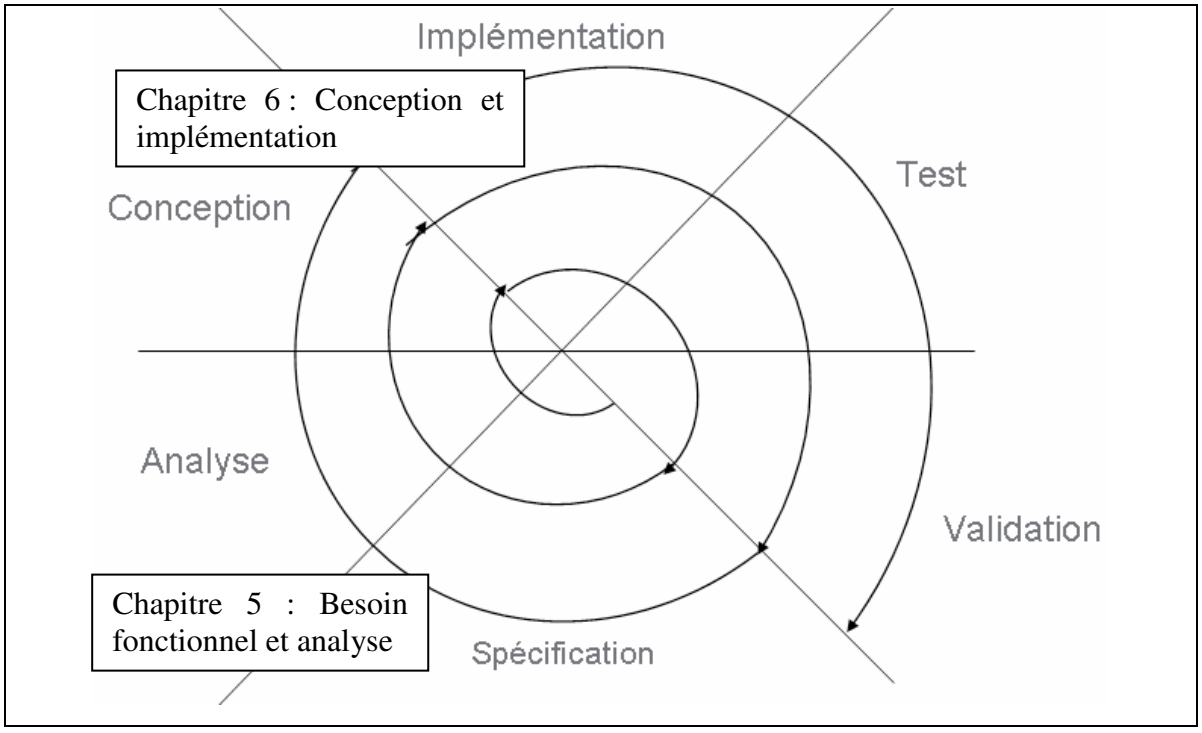


Figure 15 : Cycle de développement en spirale adapté au contexte du projet pour la mise en place de l'application CADIXE

Les phases de test et de validation ne font pas l'objet d'un chapitre à part entière, mais il va de soit que l'application CADIXE a fait l'objet de nombreux tests avant d'être remise au client. Le Chapitre 6, §3.4 est dédié à cette dernière phase.

Chapitre 5

BESOINS FONCTIONNELS ET ANALYSE

L'état de l'art des différents éditeurs XML et outils d'annotations ainsi que le choix d'une méthode de développement nous permettent de rentrer dans le vif du sujet et d'aborder dans ce chapitre les fonctionnalités de l'application Cadixe. L'étude et le recueil des besoins des biologistes sont détaillés dans la première partie de ce chapitre. Puis dans un second temps, nous présentons une analyse du processus d'annotation sous forme de cas d'utilisation et de diagramme de classes.

1 Définition du besoin fonctionnel

Dans le chapitre précédent, nous avons étudié les différents outils d'annotations, le seul qui répond complètement à notre besoin est celui qui a été développé dans le cadre du projet Caderige : le logiciel XMLJava. Le cahier des charges de Cadixe reprend l'ensemble des fonctionnalités de XMLJava. Toutefois, l'aspect ergonomique de l'application a été révisé de manière à offrir une interface permettant une prise en main rapide sans un lourd apprentissage. De nombreuses réunions avec les biologistes, utilisateurs de XMLJava, ont permis de recueillir de nouveaux besoins fonctionnels. Ces réunions ont porté sur la description et la compréhension de leurs travaux d'annotations ainsi que sur leurs méthodes de travail en s'appuyant sur une application concrète : accompagnement des biologistes dans leurs tâches d'annotation de corpus.

Dans le schéma ci-dessous, l'éditeur accepte en entrée un document texte ou un document XML référençant obligatoirement une DTD et une (des) feuille(s) de styles. Ce document subit une transformation au moyen de règles de transformation transparentes pour l'utilisateur. Les styles, pour l'édition, sont utilisés pour modifier la structure du document, et formatent le texte à l'écran. Les règles et les styles pour l'édition sont construits indépendamment du document lu, mais dépendent de sa DTD : il peut y avoir différentes feuilles de styles, donnant différentes visualisations pour un même document.

Une fois que le document est importé dans l'éditeur, l'utilisateur dispose des fonctionnalités habituelles pour l'édition : insertion et effacement de caractères, copier / coller (texte simple ou balisé), rechercher et remplacer, ajout de balise, de commentaires, valider le document XML, etc.

Le schéma suivant nous donne l'architecture générale de l'éditeur.

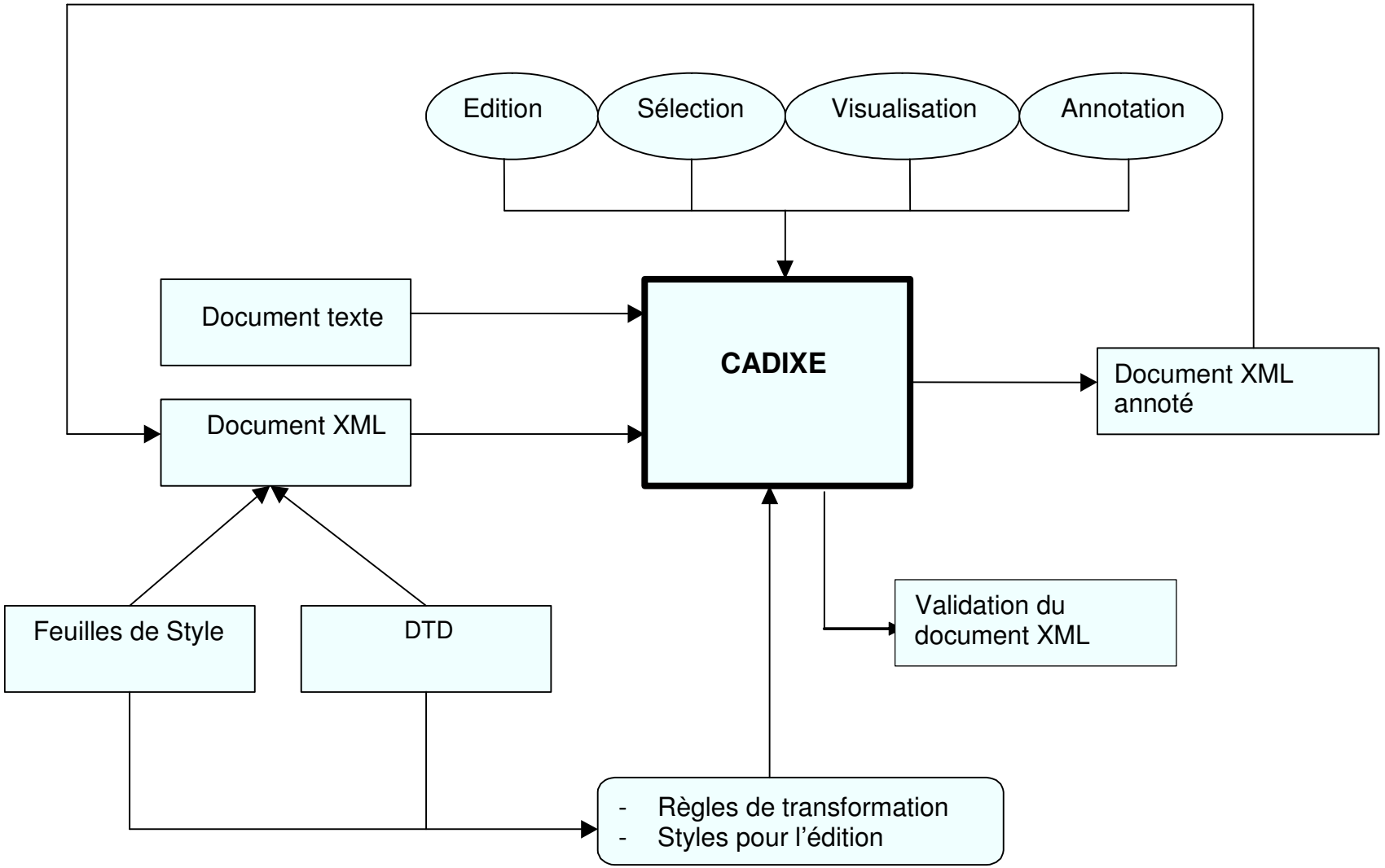


Figure 16 : Architecture générale de l'éditeur d'annotation

Les nouvelles fonctionnalités que l'on souhaite implémenter ont été regroupées ci-dessous par thème.

1.1 Description des données manipulées

L'éditeur doit permettre de manipuler plusieurs types d'informations. Respectivement, on trouvera les «préférences» qui spécifient la configuration générale de l'éditeur, les «documents» qui correspondent aux fichiers annotés et les «modèles» qui caractérisent des documents de référence.

- Les *préférences* ne sont plus gérées de manière globale comme dans la version actuelle. Elles définissent les formats d'affichage, le comportement de l'éditeur et permettent de sélectionner le niveau d'utilisation de l'interface (expert, …) ;
- Un *document annoté* avec une extension «.xml», est défini par les éléments ci-dessous, toutes les informations qui ne sont pas prises en compte par la DTD du domaine sont écrites sous la forme de commentaires dans le fichier :
    - Référence à une DTD (fichier de type «.dtd ») ;
    - Référence à une ou plusieurs feuilles de styles (fichier «.css») ;
    - Des données spécifiques au document (position curseur, …) ;
    - Le texte du document en cours d'annotation.
- Un *modèle* «.model» est identique dans sa structure à un document sinon que les parties liées au texte sont vides. L'éditeur les utilise lorsqu'il doit créer un nouveau document vide dans l'éditeur.

1.2 Organisation de l'interface

L'organisation de l'interface a été revue de manière à offrir à l'utilisateur toutes les fonctions nécessaires pour sa tâche d'annotation. Voici les modifications à effectuer :

- Création d'une interface personnalisable, en offrant la possibilité à l'utilisateur de gérer l'organisation de l'interface graphique via des préférences (Chapitre 6, §2.2.2) ;
- Possibilité de charger plusieurs documents en même temps dans l'éditeur, chaque document pouvant correspondre à des DTD différentes. Le passage d'un document à l'autre s'effectuant à l'aide de la barre d'outils ou de la barre de menu ;
- Possibilité de créer plusieurs vues (a priori 2 suffisent) sur le même document (division horizontale de la fenêtre de texte) de manière à faciliter la comparaison entre les différentes parties du document et entre les annotations ;
- Intégration d'un zoom au niveau de la visualisation du document ;
- Possibilité de fixer un interlignage de hauteur fixe dans le document, ce qui facilite la lecture lorsque les styles induisent des hauteurs de lignes variables ;
- Adaptation pour rendre les menus de l'éditeur personnalisables selon les différents niveaux des utilisateurs. Par exemple, la validation d'un document sera à disposition des utilisateurs avec un niveau d'expertise élevé.

1.3 Edition du document et palette d'outils

Dans l'éditeur actuel, certaines fonctionnalités permettant l'édition du document ont été implémentées, cependant les plus importantes restent à développer. La nouvelle version de l'application devra donc offrir les possibilités suivantes :

- Implémentation des fonctions Copier/Coller, Couper/Coller de blocs de textes annotés avec conservation des balises. Et ce en respectant, la structure du format XML. Il ne sera pas possible de réaliser cette action si le texte sélectionné chevauche plusieurs balises ;
- Ajout de fonctions permettant de modifier la «portée» d'une balise (ajout/retrait de mots/lettres) :
    - EG : Extension de la balise courante vers la gauche
    - ED : Extension de la balise courante vers la droite
    - RG : Réduction de la balise courante vers la gauche
    - RD : Réduction de la balise courante vers la droite
- La recherche d'information peut s'effectuer soit à l'aide d'une fenêtre de dialogue comme dans les éditeurs classiques, où par l'intermédiaire d'une zone de recherche présente dans l'éditeur. L'éditeur proposera également la possibilité de rechercher des balises selon son nom, sa validité, des commentaires, etc. Ce type de recherche permet de retrouver facilement une annotation spécifique ;
- Implémentation de la fonction «remplacer» qui permet de remplacer une occurrence dans le texte annoté. Le remplacement doit pouvoir se faire seulement si l'ensemble de la chaîne retrouvée appartient à une seule et même balise.

1.4 Gestion des commentaires et erreurs

Dans cette nouvelle version, on introduit la possibilité de mettre des annotations sur n'importe quelle balise du document. Ces annotations sont à stocker dans l'en-tête du fichier XML sous la forme de commentaires (balise <-- … -->) qui seront interprétés par l'éditeur au moment de la lecture. L'annotation des documents peux se faire en plusieurs passes, pour cela il est intéressant, pour faciliter le remplissage des attributs, d'attribuer un commentaire booléen: « validated » / « unvalidated » à chaque balise.

On introduit également un mécanisme permettant de récupérer les erreurs lors de la validation du document XML. L'analyse des erreurs doit être simple pour l'utilisateur, pour cela il suffit de sélectionner l'erreur, et l'éditeur se positionne automatiquement sur la balise incriminée dans le texte

1.5 Budget et planning prévisionnel

Un planning prévisionnel a été élaboré, nous l'avons illustré dans la Figure 17.

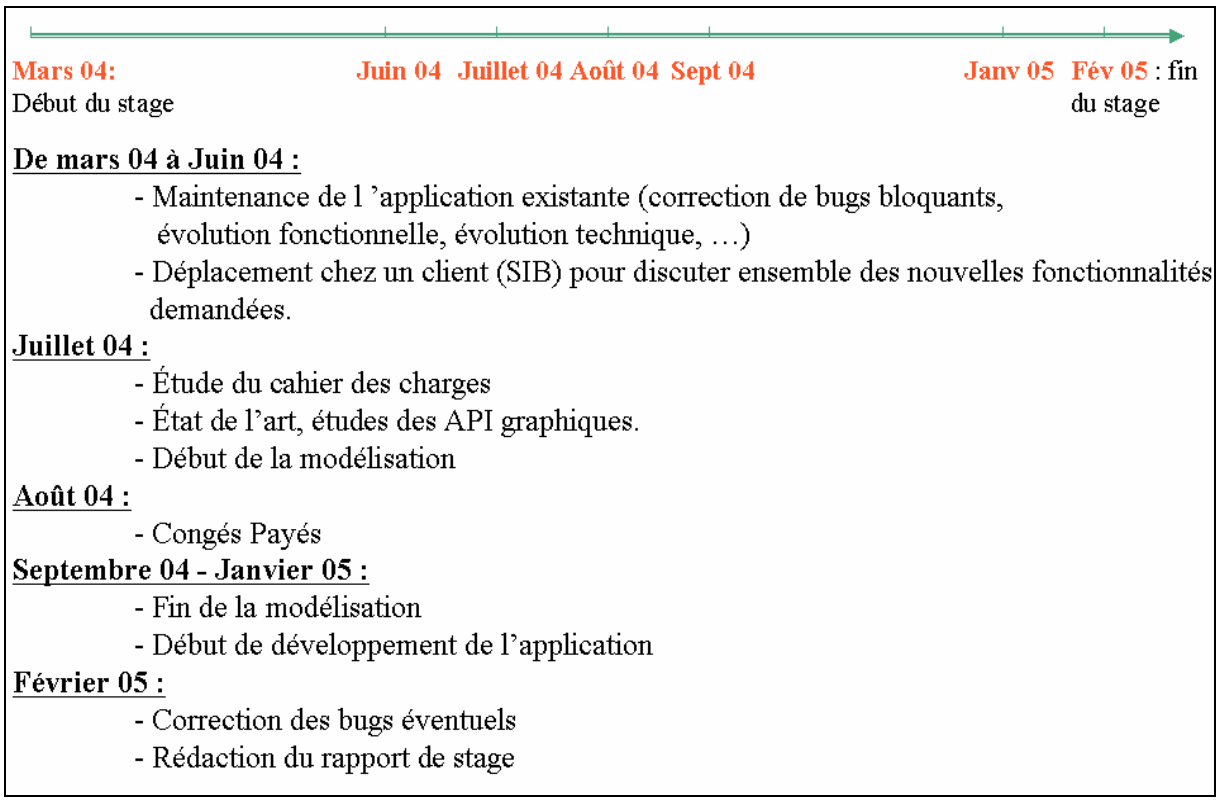


Figure 17 : Planning prévisionnel

Dans le planning ci-dessus, nous avons seulement fait apparaître les jalons les plus importants.

Notons que ce projet s'insère dans le cadre d'un projet CNAM financé par le FONGECIF pendant 9 mois. Le laboratoire de recherche Liebniz reçoit des subventions de leurs clients afin de rémunérer certains travaux. Ce dernier se déroulant sur une période d'un an, le laboratoire a financé les trois derniers mois à hauteur de 500 euros/mois.

2 *Analyse du besoin*

Nous présentons en premier lieu les acteurs (au sens UML du terme) qui interagissent avec le système [MUL00, ROQ02]. Ensuite, nous décrivons les *processus métier* (cas d'utilisation de haut niveau) de l'application CADIXE avant de décrire avec plus de précision les cas d'utilisation détaillés que nous avons développés dans ce travail.

2.1 Les acteurs du système

La lecture du cahier des charges a permis d'identifier les acteurs suivants :

- L'utilisateur (annotateur), est une personne physique qui utilise l'application dans le but d'annoter du texte ;
- L'interface, qui peut être représentée par un menu, une alerte, une boîte de dialogue ou un évènement géré par le logiciel système.

Les cas d'utilisation qui relèvent du processus métier sont décrits ci-dessous.

2.2 Processus métier

La réalisation de l'application Cadixe met en évidence un seul processus métier qui concerne l'annotation d'un document textuel. La lecture du cahier des charges, la méthode de travail des annotateurs et la complexité de ce processus, nous ont permis de le décliner en 2 sous-processus

- Annoter un document ;
- Valider et sauvegarder document.

En termes UML, ce sont des cas d'utilisation de haut niveau qui décrivent les activités principales de l'application Cadixe.

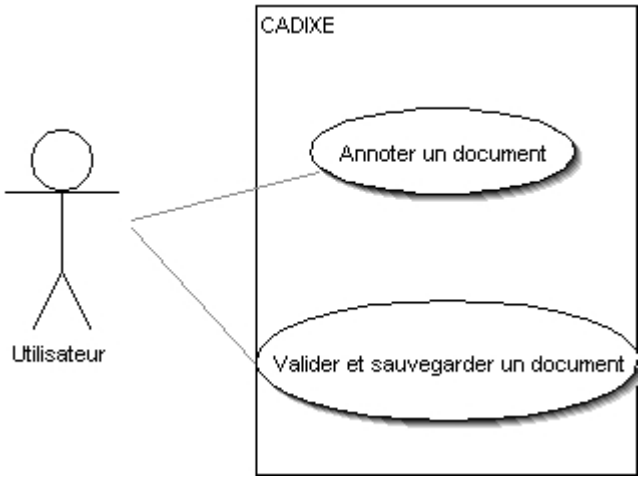


Figure 18 : les processus métier de Cadixe

Remarque :

Nous décrirons ci-dessous les cas d'utilisation liés à ces différentes étapes, mais comme nous l'avons précisé précédemment, l'éditeur offre l'ensemble des fonctionnalités d'un éditeur de texte classique.

2.3 Annoter un document

L'annotation est une opération manuelle et se fait à partir de balises contenues dans une DTD. Pour que les annotations soient facilement compréhensibles à l'écran, un style graphique est associé à chacune des balises (Chapitre 6, §2.2.1.

Ce sous-processus métier " Annoter un document" est composé des cas d'utilisation suivants :

- Ouvrir un document (avec ou sans annotation) ;
- Appliquer la balise ;
- Créer un arbre d'annotation ;
- Saisir des attributs et commentaires.

2.3.1 Ouvrir un document (avec ou sans annotation)

Objectif	L'annotation doit être effectuée sur un document texte. Il peut s'agir d'un document contenant des annotations existantes ou d'un document texte.
Acteurs	L'annotateur

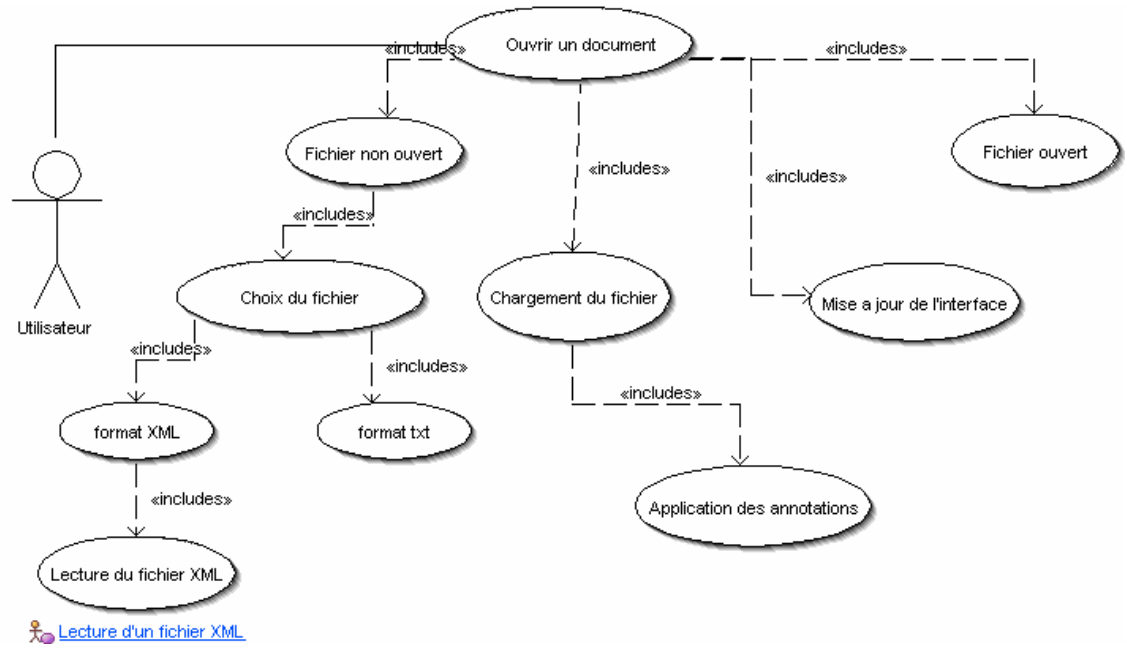


Figure 19 : Cas d'utilisation – Ouvrir un document

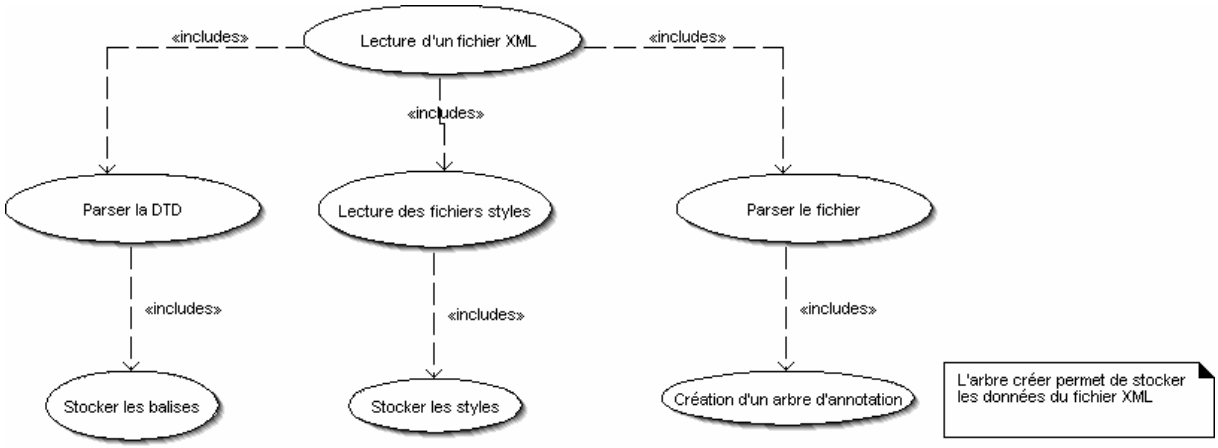


Figure 20 : Cas d'utilisation - Lecture d'un fichier XML

L'annotateur charge en mémoire un document qui peut être annoté ou non. S'il s'agit d'un document annoté, l'éditeur va automatiquement chercher dans le dossier la DTD (celle qui avait été utilisée dans le document annoté) ainsi que la feuille de styles associée. La lecture de la DTD nécessite l'utilisation d'un parseur (Chapitre 6, §1.3)

2.3.2 Appliquer la balise

Objectif	A l'aide de la souris ou du clavier, l'annotateur sélectionne la partie du document (mot, paragraphe) qu'il souhaite annoter puis il sélectionne la balise à ajouter présente dans la liste des balises
Acteurs	L'annotateur

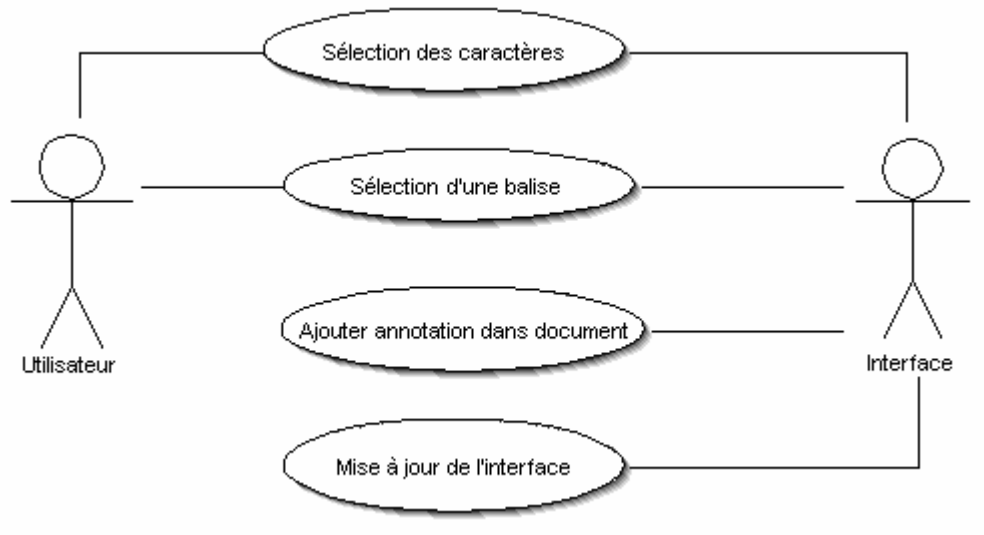


Figure 21 : Cas d'utilisation - appliquer une balise dans l'éditeur

L'annotateur clique sur l'étiquette choisie dans la liste des balises visibles à l'écran. La portion de texte sélectionnée apparaît alors dans le format d'affichage appelé style qui est associé à la balise. Suite à cette action, une nouvelle balise est insérée dans la l'arbre XML.

2.3.3 Créer un arbre d'annotation

Objectif	Toute annotation ajoutée à un texte est également ajoutée dans un arbre d'annotation
Acteurs	L'interface

Toutes les annotations appliquées dans le document sont également ajoutées dans un arbre d'annotation au format XML. Ce cas d'utilisation est détaillé dans le chapitre suivant.

2.3.4 Saisir des attributs et commentaires

Objectif	Saisir des attributs et des éventuels commentaires relatifs à une balise
Acteurs	L'annotateur

Selon la DTD utilisée, chaque balise peut avoir un certain nombre d'attributs qui servent à préciser sa signification et qui apparaissent à l'écran dans l'éditeur. Leurs valeurs peuvent être modifiées par l'utilisateur : lorsqu'elles appartiennent à un type énuméré (à une liste), les valeurs possibles sont sélectionnables à partir d'un menu, dans le cas contraire une simple zone d'édition permet de saisir le texte associé à l'attribut. L'attribut ainsi que sa valeur sont ajoutés automatiquement dans l'arbre XML. Les commentaires permettent à l'utilisateur d'associer à toutes les balises du document des informations en texte libre expliquant ses choix d'annotations.

2.4 Valider et sauvegarder un document

Le processus métier " Valider et sauvegarder un document" se décline en 3 cas d'utilisation :

- Valider le document ;
- Générer erreur si nécessaire ;
- Enregistrer le fichier sous format XML.

Une fois le travail d'annotation terminé, il est nécessaire de valider le document. La validation automatique de balises n'est pas proposée par Cadixe. Cette action permet de vérifier si le document XML généré est bien valide, que les attributs obligatoires sont bien renseignés. Cette validation s'appuie sur l'utilisation du parseur (Chapitre 6, §1.3). L'arbre des annotations abordées dans le paragraphe précédent va permettre la sauvegarde du fichier au format XML.

Le schéma de la Figure 22 représente bien l'association existante entre les différentes classes que nous venons de présenter :

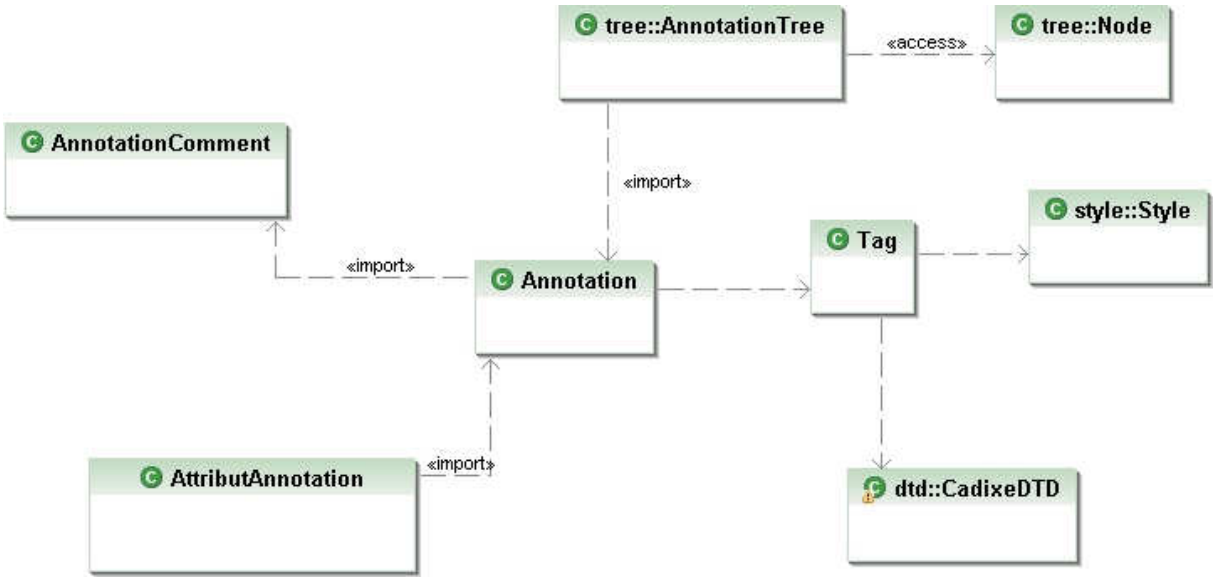


Figure 22: Associations entre certaines classes de l'application Cadixe

.

Chapitre 6

CONCEPTION ET IMPLEMENTATION DE L'EDITEUR

La conception et l'implémentation de l'éditeur correspondent aux dernières activités du cycle de développement tel que nous l'avons mis en œuvre dans ce travail. Dans ce chapitre, nous abordons les choix techniques et les choix d'implémentation qui ont été réalisés, contrairement à l'activité d'analyse dans laquelle nous sommes restés très éloignés des choix techniques. Plutôt qu'écrire le détail des étapes de conception et d'implémentation de Cadixe, nous préférons exposer les points qui nous paraissent les plus intéressants et les plus représentatifs du travail réalisé.

La première rubrique présente l'organisation logicielle mise en place pour la mise en œuvre de Cadixe. Dans un second temps, nous verrons la démarche générale de conception que nous avons suivie. Nous clôturerons ce chapitre en décrivant quelques points d'implémentation particulièrement intéressants à souligner.

1 Organisation logicielle

1.1 Choix du langage de développement

Depuis le début, Java se distingue des autres langages de programmation par son indépendance vis-à-vis de toute plate-forme [JAVA] et la richesse de ses librairies. Les objets créés en Java sont exploitables dans tout environnement supportant Java. Si l'on considère XML comme indépendant de toute plate-forme, on s'aperçoit très vite de l'intérêt de le coupler avec la technologie Java. Les principaux atouts de java sont :

- Java est fortement typé, ce qui signifie, que beaucoup d'erreurs sont détectées automatiquement à la compilation ;
- Java incorpore des nombreux traits de programmation de haut niveau : orienté objet, exceptions, polymorphisme, gestion de la mémoire, transparence des pointeurs ;
- Java possède une sémantique précise ;
- Les programmes Java sont portables. Leur exécution est indépendante de la plate-forme d'installation (type de machine). Il suffit de disposer d'un environnement d'exécution Java pour l'exécuter ;
- Java est robuste et sécurisé.

1.2 Choix d'un environnement de développement

Eclipse est un environnement de développement intégré (Integrated Development Environment) [JDOU] développé et maintenu par IBM dont le but est de fournir une plate-forme modulaire pour permettre de réaliser des développements informatiques.

I.B.M. est à l'origine du développement d'Eclipse qui est d'ailleurs toujours le cœur de son outil Websphere Studio Workbench (WSW). Eclipse utilise énormément le concept de modules nommés "plug-ins" dans son architecture.

D'ailleurs, hormis le noyau de la plate-forme nommé "Runtime", tout le reste de la plate-forme est développé sous la forme de plug-ins. Ce concept permet de fournir un mécanisme pour l'extension de la plate-forme et ainsi fournir la possibilité à des tiers de développer des fonctionnalités qui ne sont pas fournies en standard par Eclipse.

Les principaux modules fournis en standard avec Eclipse concernent Java mais des modules sont en cours de développement pour d'autres langages notamment C++, Cobol, mais aussi pour d'autres aspects du développement (base de données, conception avec UML, ...). Ils sont tous développés en Java soit par le projet Eclipse soit par des tiers commerciaux ou en open source.

Eclipse possède de nombreux points forts qui sont à l'origine de son énorme succès dont les principaux sont :

- Une plate-forme ouverte pour le développement d'applications et extensible grâce à un mécanisme de plug-ins ;
- Support de plusieurs plates-formes d'exécution : Windows, Linux, Mac OS X, … ;
- Malgré son écriture en Java, Eclipse est très rapide à l'exécution grâce à l'utilisation de la bibliothèque SWT ;
- Une ergonomie entièrement configurable qui propose selon les activités à réaliser différentes « perspectives » ;
- Une exécution des applications dans une JVM (Java Virtual Machine) dédiée sélectionnable avec possibilité d'utiliser un débogueur complet (points d'arrêts conditionnels, visualisation et modification des variables, évaluation d'expressions dans le contexte d'exécution, changement du code à chaud avec l'utilisation d'une JVM 1.4, ...) ;
- Possibilité d'utiliser des outils open source : CVS (Concurrent Version System), Ant, Junit.

1.3 Choix d'un analyseur syntaxique XML

Un analyseur syntaxique XML (ou parseur) permet de récupérer dans une structure XML, des balises**,** leurs contenus, leurs attributs et de les rendre accessibles [HUN01, RAY02, PARS]. Le parseur est l'élément incontournable d'une application XML, faire le bon choix est primordial. XML est uniquement un langage de structuration et de représentation de données. Il ne comporte pas d'instructions de contrôle et ne permet donc pas d'exploiter directement les données. Il existe deux types de parseurs :

- les analyseurs **non-validants**, qui contrôlent simplement que le document XML est bien formé ;
- Les analyseurs **validants**, qui vérifient que le document XML est valide. Cette notion a été abordée dans le Chapitre 2, §2.

Le parseur sert à "découper" le fichier de données en un ensemble de "mots" (tokens) et à les mettre à disposition d'applications. Il est associé à un module de traitement : le **processeur XML**, qui a un rôle plus important. Il en existe deux types :

- Les **processeurs SAX** (Simple API for XML), orientés événement ;
- Les **processeurs DOM** (Document Object Model) orientés hiérarchie.

1.3.1 Les processeurs SAX

SAX fournit une "*interface événementielle*" pour parcourir un document XML. En effet, cette API renvoie à l'application qui manipule le document XML des "événements" (ouverture de balise, fermeture de balise, contenu textuel, etc.). SAX est bien adapté pour les traitements qui nécessitent une seule passe sur le document, ou dans le cas de gros volumes de données, s'il n'est pas nécessaire d'avoir une représentation complète des données en mémoire.

L'interface SAX est composée de deux packages de classes **:** org.xml.sax et org.xml.sax.helpers. Parmi ces classes, org.xml.sax.ContentHandler contient les méthodes qui permettent la gestion des événements (startDocument(), endDocument(), startElement(..), endElement(..) ...).

Les processeurs SAX présentent une interface de plus bas niveau que les processeurs DOM mais ils sont plus performants (les processeurs DOM utilisent souvent un processeur SAX en interne).

1.3.2 Les processeurs DOM

Le modèle DOM est une recommandation du W3C (World Wide Web Consortium) depuis octobre 1998. Les processeurs DOM, à la différence des processeurs SAX, utilisent une "*approche hiérarchique*". Ce sont ceux que l'on rencontre le plus souvent. Ils permettent une navigation aisée dans un document mais nécessitent le chargement complet en mémoire de sa structure arborescente.

Un parseur utilisant DOM prend en entrée un document XML et construit, à partir de celui-ci, un arbre formé d'objets : chaque objet appartient à une sous-classe de "*Node*" et des opérations sur ces objets permettent de créer de nouveaux nœuds, ou de naviguer dans le document.

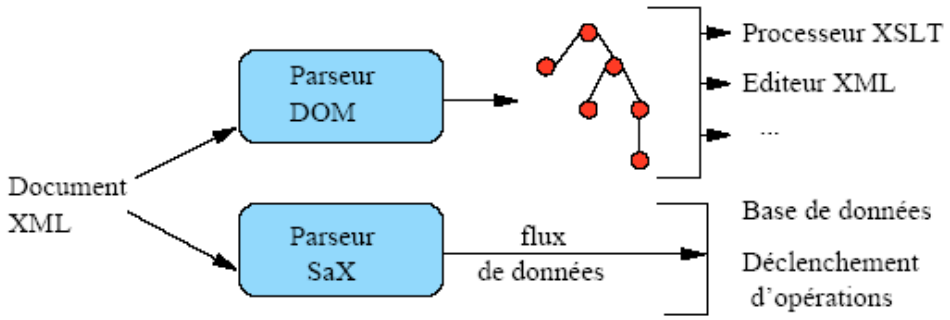


Figure 23 : Les processeurs SAX et DOM

SAX et DOM ne fournissent que des définitions ; ils ne fournissent pas d'implémentation utilisable. L'implémentation est laissée aux différents éditeurs qui fournissent un parseur compatible avec SAX et/ou DOM. L'un des parseurs les plus connus et utilisés à l'heure actuelle : "**Xerces d'Apache**". Il supporte à la fois DOM et SAX.

Dans le tableau ci-dessous, vous trouverez les avantages et inconvénients des parseurs SAX et DOM, en effet le choix d'utiliser l'un ou l'autre des deux parseurs doit tenir compte de leurs points forts et de leurs faiblesses.

	Avantages	Inconvénients
DOM	- Parcours libre de l'arbre ; - Possibilité de modifier la structure et le contenu de l'arbre ;	- Gourmand en mémoire ; - Doit traiter tout le document avant d'exploiter les résultats.
SAX	- Peu gourmand en ressources mémoire ; - Rapide ; - Principe facile à mettre en œuvre ; - Permet de ne traiter que les données utiles.	- Traite les données séquentiellement ; - Un peu difficile à programmer, il est souvent nécessaire de sauvegarder des informations pour les traiter.

Figure 24 : Avantages et inconvénients des API SAX et DOM

En regard des informations figurant dans le tableau ci-dessus, nous avons décidé d'utiliser Xerces et SAX dans l'implémentation de notre éditeur.

1.4 Choix de l'API graphique

Il existe à l'heure actuelle deux solutions majeures d'API graphiques en Java : "**Swing**" et "**SWT**" :

Swing est une bibliothèque graphique multi-plates-formes décrivant des composants graphiques complexes, complètement écrit en langage Java, et ne faisant pas appel aux composants de la plate-forme, ce qui le rend totalement indépendant du système d'exploitation.

Par opposition, la stratégie de SWT repose sur l'utilisation des composants natifs de l'OS. SWT est malgré tout écrit à 100% en Java : les appels natifs servent uniquement à invoquer les composants de l'OS. En conséquence, l'implémentation de l'API SWT n'est pas multi-plates-formes mais l'application qui l'utilise l'est.

Le comparatif ci-dessous nous a permis de prendre une décision quant au choix de l'API graphique. Ce comparatif a pour objectif de donner des éléments de comparaison en faveur ou en défaveur de chacune des solutions pour nous aider à choisir l'une d'entre elles. Les différents aspects abordés sont : les performances, la portabilité, l'apparence et l'utilisation.

	SWT	SWING
Performances	- les données à afficher sont systématiquement copiées dans les composants natifs - meilleure rapidité de chargement des composants - meilleure réactivité aux sollicitations de l'utilisateur	- les données sont utilisées directement, sans recopie - le chargement des composants peut être lent en fonction du nombre d'éléments à afficher
Portabilité	- utilisation de bibliothèques natives spécifiques à l'OS - API portable mais nécessite l'installation des bibliothèques et les jars nécessaire sur le système du client maître	- aucune contrainte particulière - API portable, ne nécessite aucune action particulière
Apparence	- le mécanisme de "Look and Feel"[7] n'est pas possible en SWT	- permet le changement et la redéfinition totale de son style par le biais des "Look and Feel"
Utilisation	- utilise les ressources de l'OS, le développeur a la charge de les libérer - documentation légère	- l'allocation et la libération des ressources sont gérées automatiquement avec le ramasse miettes[8] - API bien documentée

Figure 25 : Comparatif entre SWT et SWING

1.4.1 Conclusion

Le choix d'une de ces deux API reste assez difficile car chacune d'elle présente des avantages importants par rapport à l'autre. Il doit être orienté en fonction des besoins de l'application à développer.

SWT reste globalement plus performant que Swing. Sa rapidité d'exécution rend les composants graphiques plus réactifs qu'en Swing et procure donc un meilleur confort d'utilisation.

Esthétiquement, Swing est plus personnalisable que SWT. Il présente l'avantage de ne pas dérouter l'utilisateur en utilisant les composants natifs du système d'exploitation. La nécessité d'utilisation de bibliothèques natives en SWT est incontestablement moins pratique qu'une application Swing sous forme d'un unique jar, il se déploie facilement et sur n'importe quelle plate-forme.

[7] Look and Feel : il s'agit d'un mécanisme permettant de changer l'apparence des composants, soit globalement par exemple en adoptant le style du système, soit composant par composant.

[8] Le ramasse miettes (ou Garbage Collector en anglais) est un système qui est chargé de libérer les espaces mémoire qui sont devenus inutilisables.

Enfin, on peut souligner qu'à l'heure actuelle, il est plus difficile de trouver des développeurs SWT que des développeurs Swing. De ce fait, le choix de SWT nécessite souvent une formation, même si l'API est assez simple à utiliser. N'ayant eu aucune formation sur SWT, l'API étant peu documentée, nous nous sommes orientés vers l'API Swing d'autant plus que celle-ci nous offre la possibilité de porter l'application sur différents OS. Contrainte importante que nous avons évoquée dans le Chapitre 3 Etat de l'art.

1.5 Composants logiciels utilisés pour l'édition de texte

Le but de ce paragraphe est de faire une introduction au package ″javax.swing.text″ qui propose un ensemble de classes pour construire aussi bien un simple éditeur sans style qu'un très riche traitement de texte.

Nous avons utilisé les composants de ce package car :

- Swing respecte les contraintes du projet ;
- Ils sont inclus avec le JDK (Java Development KIT) ou tout autre machine virtuelle compatible Java ;
- Leur pérennité est mieux assurée contrairement aux packages indépendants ;
- La correction éventuelle est gérée par SUN ;

Une description exacte des classes est donnée par la JavaDoc de Sun à l'url suivante : http://java.sun.com/j2se/1.3/docs/api/javax/swing/text/package-summary.html.

1.5.1 Classe javax.swing.text.JTextComponent

La super-classe pour éditer du texte dans les composants Swing est la classe racine `JTextComponent` dérivée de la classe `JComponent`. `JTextComponent` propose un ensemble de caractéristiques pouvant être personnalisé par tous ses descendants :

- un modèle appelé ″*Document*″ pour contrôler le contenu du composant ;
- une vue qui s'occupe d'afficher le composant à l'écran ;
- un kit (prêt-à-monter) appelé *editor kit* qui implémente un ensemble de fonctions telles que lire et écrire dans un flux, des actions qui peuvent être invoquées sur le document et un support pour contrôler le rendu (la classe `View`) ;
- un support de commandes au clavier, comme les raccourcis claviers ;
- une fonction annulation de la dernière commande effectuée et rappel de la dernière commande ;
- la possibilité de changer le curseur par défaut et d'utiliser un écouteur (`Listener`) pour tous les changements.

1.5.1.1) Interface Document

L'interface `Document` modélise un conteneur de textes qui sert de modèle pour les composants textes de Swing. La structure est définie par un ensemble d'éléments (interface `Element`). Une implémentation de base de cette interface est faite avec la classe abstraite `AbstractDocument` utilisée par les modèles de composants textes.

1.5.1.2) Interface StyledDocument et la classe defaultStyledDocument

Avec cette interface, on ajoute la notion attribut de paragraphes et de caractères. L'interface `StyledDocument` étend l'interface `Document` pour travailler avec des styles. Un style est une collection d'attributs à appliquer à une partie du document. Tous les styles sont créés et maintenus par la classe `StyleContext`. Son principal rôle est d'être un conteneur de styles qui peut être utilisé par un ou plusieurs documents, l'accès à un style se fait par un nom.

La classe `DefaultStyledDocument` est la classe par défaut pour travailler sur des documents avec des styles, cette classe étend la classe `AbstractDocument` et implémente l'interface `StyledDocument`.

1.5.1.3) Structure d'un document avec styles

La structure du document peut être vue comme un arbre. Pour les textes simples sans style, il y a un niveau, c'est-à-dire que chaque ligne représente un noeud de la racine. Pour les textes enrichis de styles, il y a un niveau supplémentaire.

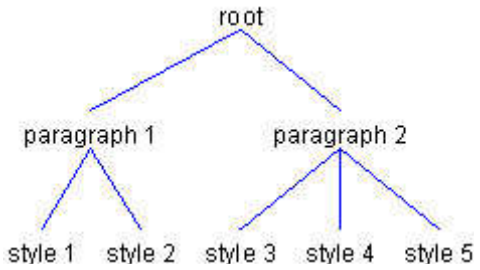


Figure 26 : La structure interne d'un document

L'arbre ci-contre représente la structure interne à l'intérieur du document. Les noeuds de l'arbre sont représentés par l'interface `Element`, chaque noeud peut recevoir des attributs de la forme (nom, valeur) (définition de l'interface `AttributeSet`).

Cet arbre représente une racine (root) et deux paragraphes ; le premier paragraphe possède deux styles et le deuxième paragraphe possède trois styles. Les feuilles représentent une suite de caractères.

1.5.1.4) Classes JEditorPane et JTextPane

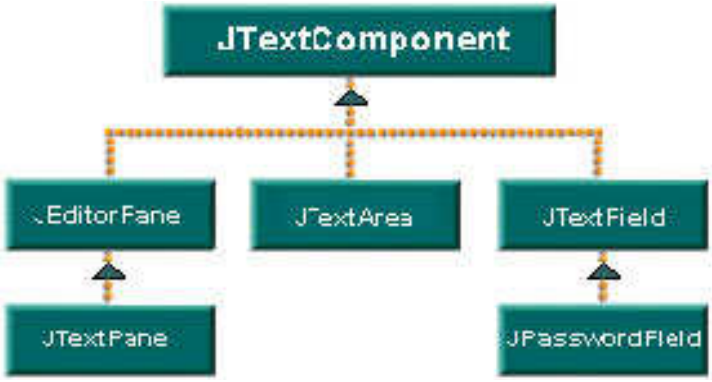


Figure 27 : Hiérarchie des classes Swing de textes Java

`JEditorPane` est un composant dérivé de `JTextComponent` pour éditer du texte sans style, du texte HTML ou du texte RTF (Rich Text Format). Cette classe offre la possibilité d'afficher du texte HTML à partir d'une URL, de lire et d'écrire directement du HTML avec des images. `JtextPane` est un composant dérivé de `JEditorPane` qui permet d'éditer du texte formaté en plusieurs polices, styles ou couleurs. Il accepte du texte, des images et des composants. `JtextPane` prend comme modèle la classe `DefaultStyledDocument` qui implémente l'interface `StyledDocument` pour travailler avec du texte enrichi.

1.5.1.4.1) Evénements

Quand un changement intervient sur le contenu du composant, la où les vues associées doivent être informées du changement. C'est via l'interface `DocumentEvent` que Swing décrit les changements. `JTextComponent` est donc responsable de tout changement à l'intérieur du composant. Plus exactement, il modifie son contenu puis informe les vues de toute modification.

Figure 28 : Propagation des événements.

Le schéma (cf Figure 28) montre l'interface `Document` avec deux `JTextComponent`. À chaque modification dans le document, un événement `DocumentEvent` est propagé vers les deux composants et vers un historique (`UndoManager`).

2 Construction de l'éditeur

L'éditeur Cadixe est une application interactive qui exploite des documents de type XML ou texte. Pour les documents XML, il base son traitement sur la structure logique du document. Il assure la lecture de ces derniers à l'aide d'un analyseur (SAX) qui offre un ensemble de primitives permettant d'accéder aux éléments, aux attributs et au contenu du document.

Un mécanisme de notification permet à l'application d'associer des traitements à des événements déclenchés par les actions de l'utilisateur sur le document. Ces actions incluent l'insertion, la modification, la suppression de caractères ou d'attributs dans le document.

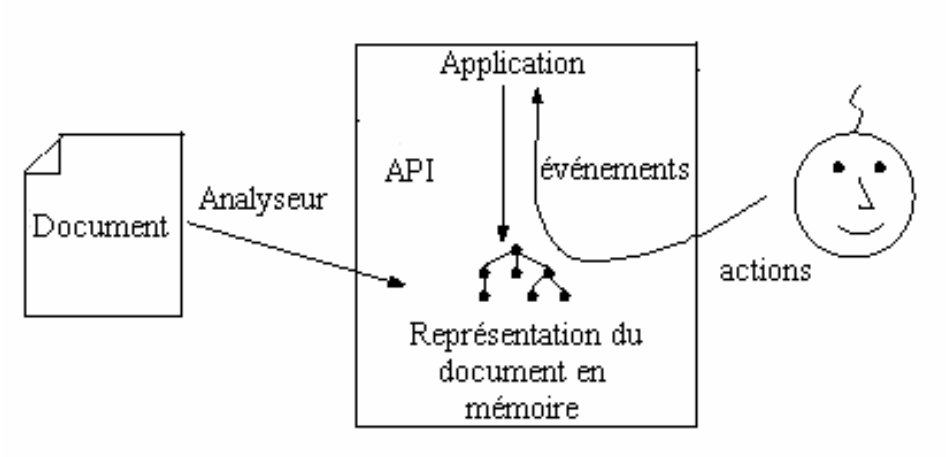


Figure 29 : Application interactive pour éditer et annoter des documents XML

2.1 Eléments constituants l'éditeur

Cette partie décrit les principaux éléments utilisés pour construire l'éditeur. Les API Swing de Java permettent de construire les fonctionnalités de base d'un éditeur, que sont : la lecture, l'édition et la sauvegarde d'un document. Nous verrons la composition de l'interface utilisateur, son principe de base ainsi que les éléments spécifiques qui ont été rajoutés.

2.1.1 Interface utilisateur

L'interface utilisateur de Cadixe a été organisée de la même manière que les éditeurs de texte que l'on retrouve sur le marché. Outre la barre de menu et la ligne d'outils, la fenêtre principale de Cadixe est organisée en plusieurs zones qui peuvent être redimensionnées (ou masquées) en faisant glisser les barres de séparation. Dans le détail, l'interface comprend :

- Le menu de l'éditeur : toutes les actions et les options de l'éditeur sont représentées ;
- La barre d'outils : les principales fonctions y sont regroupées. La barre d'outils est complètement paramétrable. Pour cela, il suffit d'aller dans le menu Edit/Préférences et choisir l'option "Toolbar". Les feuilles de styles sont incluses dans une zone de liste modifiable (*Combo Box*) ;
- La zone d'édition : zone où est représenté le document stylé à l'utilisateur ;

- La liste des balises : cette fenêtre représente les différentes balises contenues dans la DTD liées au document en cours de visualisation. L'utilisateur a la possibilité de choisir son mode d'affichage dans les préférences de l'éditeur ;
- Les attributs : cette fenêtre permet d'afficher et de saisir les valeurs d'attributs des balises. Elle est mise à jour automatiquement à chaque fois qu'on change de balise. Il existe différents types d'attributs : les énumérés et les attributs simples. L'éditeur offre également la possibilité d'activer ou de désactiver des attributs. Cela est possible en précédant l'attribut par une « CheckBox » ;
- Les commentaires : L'utilisateur a la possibilité d'associer à chacune des annotations un commentaire. On lui donne également le moyen de naviguer d'un commentaire à un autre ;
- La vue rechercher/remplacer : Cette vue permet de rechercher/remplacer une chaîne de caractères, une balise, un commentaire ;
- La vue XML : Cette vue est le résultat de la transformation du document XML. Elle est composée de plusieurs colonnes :
    - la colonne "XMLTree", cette zone contient trois informations importantes :
        - le nom de la balise ;
        - le nom des attributs et leurs valeurs ;
        - le texte annoté;
    - la colonne "comment" : lorsqu'un commentaire est associé à une annotation ;
    - la colonne "start" : cette colonne correspond au début de la sélection ;
    - la colonne "end" : elle correspond à la fin de la sélection ;
    - la colonne "error" : à la validation du document si des erreurs existent une pastille rouge s'affiche sur la ligne correspondante.

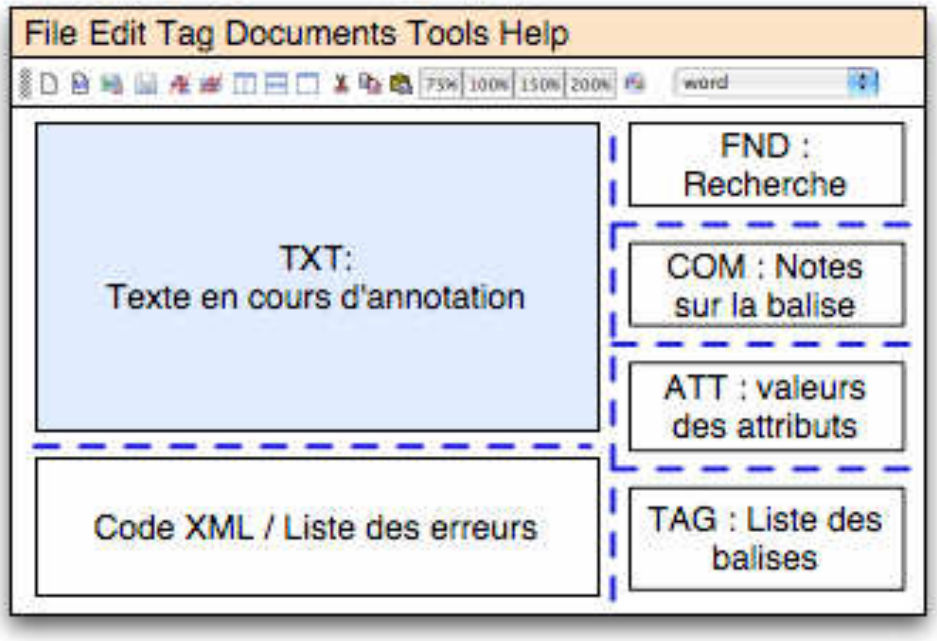


Figure 30 : Organisation de l'interface de l'éditeur CADIXE

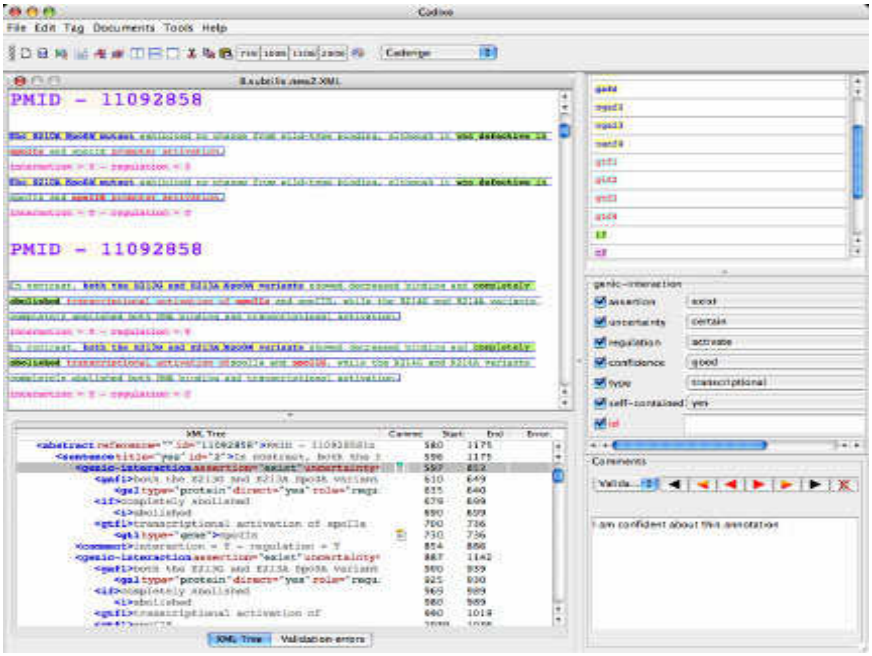


Figure 31 : Fenêtre principale de l'application CADIXE

2.2 Eléments ajoutés dans l'éditeur

2.2.1 Styles

Pour construire l'image du document, le processus de formatage parcourt la structure logique du document en appliquant des règles de présentation sur les éléments rencontrés. Ces règles sont des styles, un style étant composé de taille de caractères, fonte, couleur, etc. Afin d'enrichir la palette visuelle, nous avons rajouté dans Cadixe la possibilité de dessiner des rectangles autour des annotations et des styles paramétriques. Ils permettent d'associer plusieurs feuilles de styles à une même balise, le choix du style à afficher étant fonction de la balise et de la valeur d'un ou plusieurs attributs. Les feuilles de styles utilisent le format CSS. Notre choix s'est porté sur ce format car il présente les avantages suivants :

- La structure et la présentation sont gérées séparément ;
- La conception est indépendante de la présentation ;
- Le code HTML est réduit en taille et en complexité.

Il existe des éditeurs CSS dans le domaine public que nos utilisateurs pourront utiliser pour la création de leurs feuilles de styles. En annexe II, vous trouverez un exemple de feuille de styles.

2.2.2 Préférences

La boîte de préférences permet de configurer le comportement général de l'éditeur CADIXE. A travers cette boîte de dialogue, il est possible d'effectuer différents points de paramétrage, des copies d'écrans seront fournies en annexe III :

- *Default paths* : indique à l'éditeur où il peut trouver les différentes ressources telles que: le nom du modèle par défaut ainsi que les chemins d'accès pour les modèles, les DTD, les feuilles de styles, les scripts et les documents annotés. Lorsque le chemin est relatif, la position peut-être donnée soit par rapport à l'application (c'est le cas standard), soit par rapport au chemin d'accès qui a été passé dans le paramètre *–userhome* de la ligne de

commande. La sélection de l'une ou l'autre de ces possibilités se fait à l'aide du « radio-bouton ». On peut ainsi configurer l'éditeur pour que les ressources « partagées » (DTD, modèles, styles, …) soient gérées globalement au niveau de l'application alors que les ressources « utilisateurs » (les préférences, les documents) sont rangées au niveau du dossier de travail.

- *Editor* : Permet de définir l'ordre dans lequel les outils (palettes) sont rangés dans la partie droite de l'interface. Cette option permet également le réglage des paramètres généraux de l'éditeur comme : l'ouverture automatique d'un nouveau document initialisé avec le modèle par défaut lors du lancement de l'application, le nombre maximum de documents visibles simultanément lorsque l'on est dans le mode multi-documents et enfin le nombre de fichiers qui sont mémorisés dans la liste de la commande "Load Recent" du menu "File".
- *Annotation* : option d'annotation permettant de dire si la sélection d'une zone de texte s'effectue au niveau du caractère ou directement au niveau des mots (défaut).
- *XML zone* : permet la configuration de l'affichage de la zone XML. On peut ainsi choisir la police et la taille des caractères utilisés dans cette fenêtre. Il est également possible de sélectionner les couleurs et styles associés aux balises, aux attributs, aux valeurs ainsi qu'à l'extrait de texte placé à droite de la balise.
- *Attributes* : paramétrage de l'éditeur pour renseigner la valeur des attributs lors de l'insertion d'une balise et contrôler comment sont générés les attributs dans le code XML du document annoté.
- *Tags* : option d'affichage de la liste des balises. Trois types d'affichages sont proposés par l'éditeur :
    - Affichage de la liste complète des balises. Ce type permet d'appliquer les balises dans n'importe quel ordre ;
    - Affichage des balises plus spécifiques que la balise courante. Ce mode "strict" laisse moins de liberté dans l'ordre dans lequel les annotations peuvent être insérées, mais il est très utile pour guider les utilisateurs qui ne maîtrisent pas très bien une DTD ;
    - Affichage de la liste complète des balises en activant seulement les balises applicables (les balises non applicables apparaissent grisées). Cette possibilité est intéressante si on est en présence d'une petite DTD comportant peu de balises.
- *Toolbar* : configuration des commandes à mettre dans la barre d'outils. L'affichage se compose de 2 colonnes : à gauche la liste des commandes à faire apparaître dans la barre d'outils et à droite la liste de toutes les commandes qui sont disponibles dans l'éditeur. L'utilisateur peut changer l'ordre d'apparition des commandes par « drag & drop » dans la liste de gauche.

2.2.3 Documents

Les commandes de chargement de fichiers se trouvent dans le menu "File" et sont également accessibles sous la forme d'icônes dans la barre d'outils. Comme dans tout éditeur de texte, on retrouve les commandes d'ouverture de fichier, de fermeture et de création de nouveau document. La seule différence notable est que le chargement d'un nouveau fichier prend en compte la notion de "modèle". En effet, l'éditeur construit un nouveau document en prenant comme modèle le fichier "*default.model*" qui se trouve dans le dossier *"/formats/models/*". Un modèle n'est rien d'autre qu'un fichier qui contient les noms des DTD et des feuilles de styles à charger dans le nouveau document.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE annotated-document SYSTEM "Caderige(1.61).dtd">
<!--StyleSheet "Caderige(1.61).css"-->
```

L'éditeur offre également la possibilité d'ouvrir un nouveau fichier directement en laissant à l'utilisateur le choix du modèle parmi ceux existants. L'ouverture d'un fichier annoté ne nécessite pas de modèle, les informations concernant la DTD et le fichier style sont inclues dans ce dernier.

En effet, lors de la sauvegarde du fichier annoté, l'ensemble des informations concernant le contexte de travail (la DTD, le(s) fichier(s) styles, la position du curseur dans le document, le zoom, l'espacement, …) sont sauvegardées dans le fichier XML. De plus, l'éditeur permet à travers une boîte de dialogue d'afficher et d'ajouter des "méta-informations » sur le document afin de l'identifier complètement. Ces informations sont stockées sous la forme de commentaires dans l'en-tête du fichier XML. On peut indiquer notamment :

- le titre du document ;
- le nom des annotateurs ;
- les références du document original ;
- des commentaires sur l'historique de l'annotation.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE transcription SYSTEM "word.dtd">
<!--StyleSheet "word.css"  -->
<!-- Document settings "CursorPosition=126"-->
<!-- Document settings "Zoom=100"-->
<!-- Document settings "Spacing=10"-->
<!-- Document Information "Modified=26/1/2005 11:9:21:"-->
<!-- Document Information "Title=Test drive"-->
<!-- Document Information "References="-->
<!-- Document Information "Annotators=Hassiba Zidelkheir"-->
<!-- Document Information "Comments= Petit test "-->
<transcription>
<turn who="u" id="turn_50">  <utt id="utt_52">    <word id="word_260"><!-- CadixeComments
"Validated;Comment about this tag"-->use<emptyWord> </emptyWord></word> <emptyWord>
</emptyWord>    <word id="word_261">the    </word>    <word
id="word_262">helicopter<emptyWord2 id=""> </emptyWord2></word>    <word std="miss"
id="word_263">to</word>   <word id="word_264">get</word>   <word
```

Figure 32 : Exemple de fichier XML contenant l'ensemble des informations relatives à un document

2.2.4 Balises contenues dans la DTD

Les balises sont les éléments de base de notre application. Elles sont contenues dans la DTD et chargées en mémoire au lancement de l'application (selon le modèle par défaut) ou chargées avec un fichier annoté. On les retrouve dans l'écran TAG de l'interface utilisateur de la manière suivante :

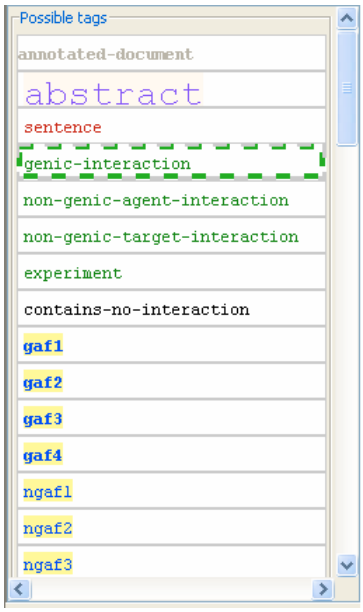


Figure 33 : Liste des balises présentent dans la DTD Caderige

Dans la zone d'édition, on les différencie par rapport à leur style.

Figure 34 : Application des balises dans la zone d'édition

Comme dans tout éditeur, l'utilisateur est libre d'insérer, copier/coller, supprimer du texte dans le document. Si le texte modifié appartient à une balise, ses positions de début et fin sont automatiquement mises à jour, toutefois si l'ensemble des caractères appartenant à cette zone sont supprimés, la balise est automatiquement supprimée de l'arbre XML.

Il existe deux types de suppression :

- Suppression de la balise courante : on détermine la balise courante par rapport à la position du curseur dans le document annoté. Cette action ne modifie pas les balises qui seraient incluses dans celle-ci. Dans l'exemple ci-dessous, si l'on supprime la balise ayant la feuille de styles rouge :
  - Avant : `contrast, both the E213G and E213A Spo0A variants showed`
  - Après : `contrast, both the E213G and E213A Spo0A variants showed`
- Suppression d'un bloc de balise : cette action entraîne la suppression des balises inclues dans la balise courante. Dans le cas ci-dessous, on aurait :
  - Avant : `contrast, both the E213G and E213A Spo0A variants showed`
  - Après : `contrast, both the E213G and E213A Spo0A variants showed`

2.2.5 Attributs

Selon la DTD utilisée, chaque balise peut avoir un certain nombre d'attributs qui servent à préciser sa signification et qui apparaissent dans la partie ATT de l'éditeur (Figure 30). Leurs valeurs peuvent être modifiées par l'utilisateur : lorsqu'elles appartiennent à un type énuméré (liste), les valeurs possibles sont sélectionnables à partir d'un menu, dans le cas contraire une simple zone d'édition permet de saisir le texte associé à l'attribut.

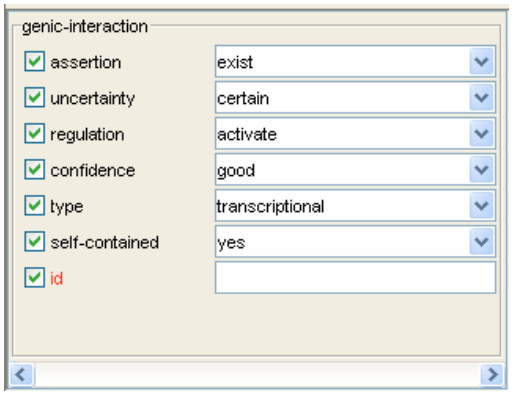


Figure 35 : Fenêtre représentant les attributs de la balise "Genic-Interaction"

Dans les préférences de l'éditeur, il existe une option qui permet l'activation/désactivation des attributs. Lorsque cette option est active, une "check-box" apparaît à côté de chaque attribut. Un attribut désactivé ne sera pas écrit dans le code XML à moins que celui-ci soit obligatoire (Required). Cette option de paramétrage évite d'inclure dans le code XML des attributs qui ne sont pas renseignés.

2.2.6 Arbre XML

L'arbre XML est représenté dans la zone inférieure gauche de l'éditeur. Cette zone est composée de deux onglets, un premier traité par ce paragraphe et un second qui concerne les erreurs (§2.2.7. L'arbre XML permet d'afficher les balises XML insérées dans le texte ainsi que leurs positions de début et de fin (exprimées en nombre de caractères). Cet onglet permet de contrôler si l'annotation

en cours est satisfaisante. En effet, les balises sont insérées dans l'ordre dans lequel elles apparaissent dans la DTD. La navigation entre la fenêtre d'édition (TXT) et l'arbre XML est synchronisée, lorsqu'on déplace le curseur dans l'une de ces fenêtres, l'autre est automatiquement rafraîchie de manière à visualiser la même information.

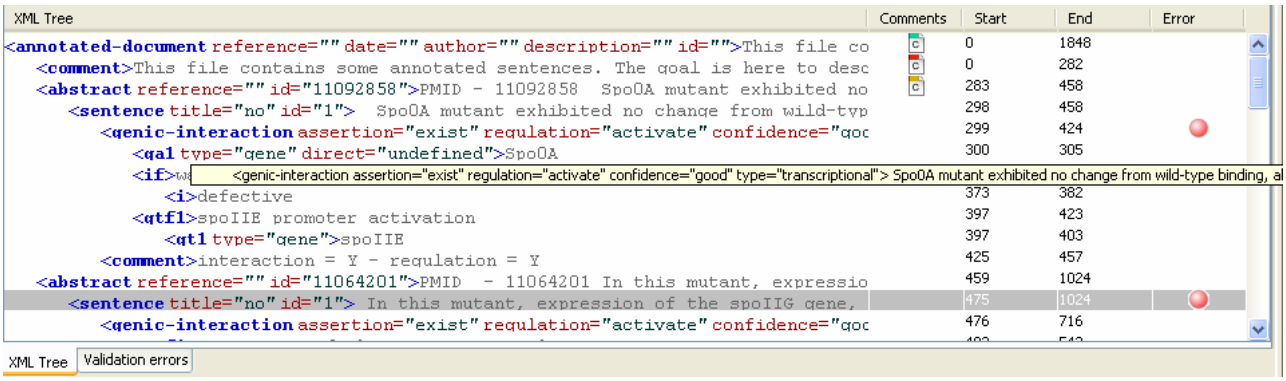


Figure 36 : Extrait d'un arbre XML

2.2.7 Validation du document et gestion des erreurs

Comme tous les éditeurs XML, Cadixe permet de valider un document annoté. Le résultat de la validation est généré dans le second onglet de la partie inférieure de l'éditeur.

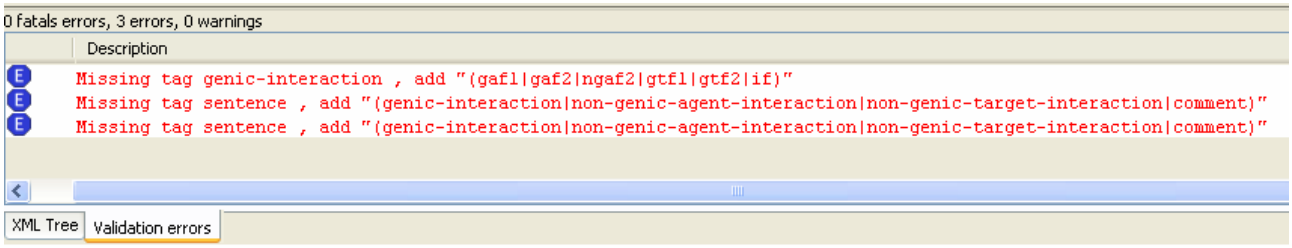


Figure 37 : Exemple d'erreurs générées suite à une validation de document

Les erreurs survenues lors de la compilation sont toute remontées dans Cadixe. Lorsque l'utilisateur clique sur une erreur de la liste, l'éditeur positionne automatiquement le curseur sur la position incriminée dans le document. Par contre, si l'on double-clique sur une erreur, Cadixe repasse dans le mode de visualisation des balises XML en se positionnant au niveau de la balise en erreur. Pour repérer plus facilement les balises en erreur dans l'arbre XML, les balises fautives sont marquées d'une pastille rouge (Figure 36).

Les erreurs ont été classifiées en trois types :

- "warning" : elles signalent généralement un problème concernant les attributs des balises (valeurs manquantes, valeurs erronées, …) ;
- "error" : ce type d'erreur est le plus fréquent. Il correspond typiquement à l'absence d'une balise dans le code XML ;
- "fatal error" : a priori ce dernier type d'erreur ne devrait pas se produire. Lorsqu'il est détecté, il stoppe la validation du document. Il se produit par exemple, si une balise fermante dans le code est manquante.

2.2.8 Commentaires

Pour chacune des balises qui sont insérées dans le texte, l'utilisateur peut associer un commentaire lui permettant d'expliquer l'annotation qu'il a effectuée et éventuellement ses doutes ou interrogations sur la validité de cette annotation. Cette possibilité est à la fois intéressante dans le cadre d'un travail qui peut se dérouler sur plusieurs mois et/ou qui est effectué conjointement par différents experts. Les commentaires sont organisés en deux parties :

- Une partie en texte libre ;
- Un « état » qui prend sa valeur parmi : {validated, unvalidated, erroneous}. La signification de l'état est purement informelle et permet juste d'indiquer le degré de certitude que l'on possède sur l'annotation.

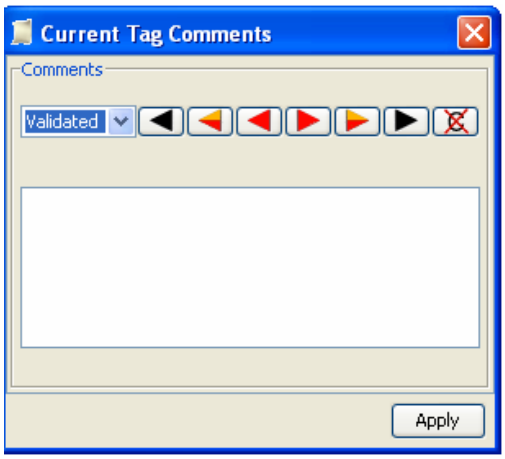


Figure 38 : Boîte de dialogue permettant la saisie d'un commentaire.

Une icône spécifique vient s'ajouter dans la colonne "Comment" de la zone XML pour les balises commentées. La couleur de l'icône est représentative de l'état du commentaire (Figure 36). Il faut noter que ces commentaires sont stockés dans le document annoté (XML) dans des champs commentaires « <-- ... --> » (au sens XML du terme) qui sont « interprétés » par CADIXE. Ces ajouts ne changent donc pas la conformité du document avec la DTD.

2.2.9 Palette d'outils intégrés

2.2.9.1) Outils de recherche

En plus des fonctions de recherche/remplace d'une chaîne de caractères dans le document annoté, Cadixe permet de rechercher des éléments propres à l'annotation. En effet, il est possible de rechercher des balises, des attributs ainsi que des commentaires. Il est même possible de combiner ces trois options. La recherche de texte dans le document peut être effectuée à l'aide d'une expression régulière lorsque la « checkbox » correspondante est activée.

Voici quelques directives classiques utilisées dans les expressions régulières :

.	: le point indique un caractère quelconque dans la recherche.
^ et &	: correspond respectivement au début et à la fin d'une ligne.
\d, \w, \s	: désignent respectivement un chiffre, un caractère de mot, un espace.
[ ]	: indique une liste de caractères (ex : [0-9] ; [a-z] ; [abcdef0123] …).
?	: séquence de caractères répétée 0 ou 1 fois.

+	: séquence de caractères répétée 0 ou plusieurs fois.
*	: séquence de caractères répétée 1 ou plusieurs fois.
|	: alternative (si l'on cherche les mots « le » ou « la » : « le|la »).
{*num*}	: séquence de caractères répétée *num* fois exactement.
{*min,max*}	: séquence de caractères répétée entre *min* et *max* fois.
( )	: pour délimiter des séquences (ex : « [0-9\s(protein|gene)+ »).

2.2.9.2) Masquage des écrans

Cette option de l'éditeur permet d'afficher ou à l'inverse de masquer les palettes d'outils affichables dans la partie droite de l'éditeur. L'état des palettes est automatiquement stocké dans le fichier des préférences. A chaque lancement de l'éditeur, l'utilisateur retrouve la configuration qu'il avait lors de la session précédente. L'ordre d'affichage est modifiable via les préférences (§2.2.2. Ainsi, en fonction de la tâche que l'on est en train d'effectuer, on peut utiliser une ou plusieurs palettes. Par exemple, durant l'annotation, on utilise principalement les palettes "attributs" et "balises" alors que si on est en train de commenter une annotation déjà réalisée on utilisera plutôt les palettes de "recherche" et "commentaires".

En annexe IV, vous trouverez un récapitulatif sur la configuration, l'installation et l'exécution de l'application Cadixe.

3 Implémentation technique des principales fonctionnalités

Après avoir détaillé les principaux écrans de Cadixe, nous allons aborder le dernier sujet de ce chapitre qui traite de l'implémentation des fonctionnalités clés du système. L'objectif n'est pas de mettre à disposition un grand nombre de lignes de codes mais plutôt d'expliquer la logique technique mise en œuvre en s'appuyant sur du pseudo-code.

3.1 Lecture du document XML

Nous avons utilisé SAX pour lire le document XML. Le schéma en Figure 39 : Architecture d'une application utilisant SAXreprésente l'architecture d'une application utilisant SAX

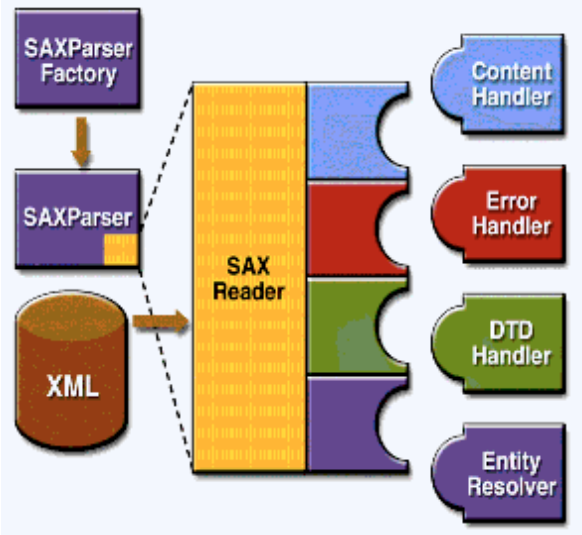


Figure 39 : Architecture d'une application utilisant SAX

Pour commencer, l'application récupère un **parseur** (*javax.xml.parsers.SAXParser*) à partir d'une **fabrique de parseurs** (*javax.xml.parsers.SAXParserFactory*). Ce parseur parcourt le **document XML** grâce à un **lecteur** (*org.xml.sax.XMLReader*). Ce dernier contient plusieurs **gestionnaires** (ou *handlers*). Ce sont ces différents gestionnaires qui sont chargés du traitement des "événements" lors du parsing. Voici les quatre principaux types de handlers (interfaces du package *org.xml.sax*) :

- **Gestionnaire de Contenu** : Le *ContentHandler* est chargé des événements comme le début ou la fin du document, l'ouverture ou la fermeture de balises ou encore la lecture de caractères ;
- **Gestionnaire d'Erreurs** : Le *ErrorHandler* va traiter les trois types d'erreurs possibles lors du parsing : les erreurs simples, les erreurs fatales et les warnings ;
- **Gestionnaire de DTD** : Le *DTDHandler* gère les événements relatifs aux DTD ;
- **Gestionnaire d'entités externes :** L'*EntityResolver* est chargé de gérer les entités externes, en fournissant une InputSource adéquate.

Il n'est pas nécessaire d'implémenter les quatre types de handler. Nous avons utilisé une classe qui étend la classe `DefaultHandler` et implémente l'interface `LexicalHandler` du package `"org.xml.sax"` ainsi qu'un parseur de type `SAXParser` instancié de la manière suivante :

```
SAXParserFactory fabrique = SAXParserFactory.newInstance();
SAXParser parseur = fabrique.newSAXParser();
File fichier = new File("./FichierALire.xml");
DefaultHandler gestionnaire = new GestionFile();
parseur.parse(fichier, gestionnaire);
```

Ce parseur lit alors le fichier XML que nous lui fournissons et appelle les méthodes du `DéfaultHandler` et `LexicalHandler` à chaque fois qu'il rencontre des balises ou des chaînes de caractères et les reconnaît. Les méthodes que ces interfaces nous offrent et que l'on doit ré-écrire sont celles concernant l'entrée dans un document ou une balise, la sortie, le contenu, les commentaires, la récupération du nom de la DTD contenu dans le fichier, …

Les méthodes utilisées sont :

startElement()	cette méthode est appelée lors de la détection d'un tag de début
endElement()	cette méthode est appelée lors de la détection d'un tag de fin
characters()	cette méthode est appelée lors de la détection de données entre deux tags
endDocument()	cette méthode est appelée lors de la fin du traitement du document XML

Nous ne sommes pas rentrés dans le détail, mais toutes ces méthodes sont dotées d'attributs.

Par exemple, la méthode `startElement` détecte l'entrée dans une balise, il faut alors lui dicter le traitement à réaliser. Pour se faire, nous bénéficions d'attributs "name" et "attrs", ils correspondent au nom et aux attributs de la balise dans laquelle on se trouve. C'est ainsi que nous récupérons les balises constituants notre document XML. La méthode `characters` nous permet quant à elle de récupérer les données comprises entre deux balises. C'est grâce à cette méthode qu'on arrive à retrouver facilement la position de début et fin de balise dans le document.

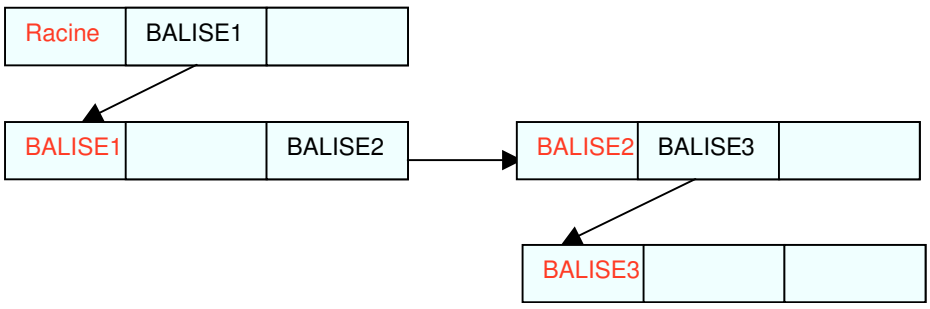
L'interface `LexicalHandler` propose différentes méthodes mais nous avons seulement utilisé la méthode `comment()`. Cette dernière nous permet de récupérer l'ensemble des commentaires associés aux annotations.

3.2 Transformation du document XML en arbre binaire

Un document XML importé dans l'éditeur possède une structure logique différente de la structure utilisée en mémoire. En effet, à la lecture, le document subit des transformations afin de générer en sortie une structure facile à exploiter : un arbre binaire.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ANNOTATION SYSTEM "GS.dtd">
<!-- version 9 6_1_2003 20_11_2004_9_3_47_132 -->

<RACINE> Balise représentant la racine du document
<BALISE1> Début de la balise " BALISE1 "
</BALISE1> Fin de la balise " BALISE1 "
<BALISE2> Début de la balise " BALISE2 "
<BALISE3> Début de la balise " BALISE3 "
</BALISE3> Fin de la balise " BALISE3 "
</BALISE2>. Fin de la balise " BALISE2 "
</RACINE>Fin de la balise RACINE. Représente la fin de document
```



3.2.1 Notion d'arbre binaire

L'arbre binaire de recherche est une arborescence binaire représentée sous forme chaînée : chaque nœud de l'arbre contient :

- un enregistrement (informations) muni de sa clé,
- deux champs droit et gauche, pointeurs vers les deux fils du nœud,
- un champ père, pointeur vers le père du nœud.

Lorsque les sous-arbres droit ou gauche sont vides, les champs correspondants valent null. Le nœud racine est le seul pour lequel le champ père est null.

3.2.1.1) Spécificité de Cadixe

Nous avons utilisé la notion d'arbre binaire en modifiant légèrement la structure d'un nœud. Un nœud Cadixe est composé de trois champs :

- un enregistrement clé (informations) qui pointe sur un objet de type "Tag" (annotation) ;
- un champ qui pointe vers le nœud frère ;
- un champ qui pointe vers le nœud Fils.

Figure 40 : Structure d'un nœud

Voici un exemple de description d'un nœud Cadixe en Java :

```
class Node
{
      Object      info;
      Node        Fils;
      Node        Frere;

/** Constructeur.
  * On met juste l'information dans le nœud,
  * Le reste sera garni lors de l'insertion. */
      public Node(Object o){
            info = o;
      }
}
```

3.2.2 Principales API utilisées pour la gestion de l'arbre

Ces API ont été mises en place pour déterminer la relation qui existe entre deux nœuds (annotations). Elles nous sont utiles pour l'insertion et la suppression de nœuds dans l'arbre.

3.2.2.1) Egalité entre deux nœuds

Cette fonction a été mise en place pour vérifier l'égalité entre deux nœuds. On dit que deux nœuds sont égaux si :

Soit,

```
Node  n1 ;
Node  n2 ;

Annotation key1 = n1.getKey() ;
Annotation key2 = n2.getKey() ;

Si    (key1.getTag().getName() = key2.getTag().getName()) et
      (key1.getStart() = key2. getStart ()) et
      (key1.getEnd () = key2. getEnd ()) Alors
      n1 = n2
FinSi
```

3.2.2.2) Fraternité entre deux nœuds

Cette fonction renvoie vrai si les deux nœuds passés en paramètres sont frères et faux sinon. On dit que deux nœuds sont frères si :

```
public boolean isBrother(Node n1, Node n2)
{
Annotation annot1 = n1.getKey() ;
      Annotation annot2 = n2.getKey() ;

            int startAnnot1 = annot1.getStart();
            int startAnnot2 = annot2.getStart();

            int endAnnot1 = annot1.getEnd();
            int endAnnot2 = annot2.getEnd();

return (endAnnot2<=startAnnot1||startAnnot2>=endAnnot1);}
```

3.2.2.3) Recherche d'un nœud

Voici l'algorithme de recherche du nœud n dans le sous-arbre de racine x :

```
public Node findNode(Node n)
{
      Node x = racine;

      while ( (x!=null) && (!isEqualsWithName(x,n)) )
      {
            if (isBrother(x, n))
                  x = x.brother;
            else if (isChildren(x,n))
                  x = x.children;
      }
      return x;
}
```

Cet algorithme est itératif. Depuis la racine x on descend un chemin dans l'arbre, en direction d'un fils ou d'un frère, selon le résultat de comparaison entre le nœud cherché et le nœud visité. La descente s'arrête

- soit sur un pointeur null, ce qui signifie que la clé n'est pas dans l'arbre.
- soit sur le nœud cherché : il est renvoyé.

La fonction `isEqualsWithName(x,n)` vérifie que le nom de l'objet annotation contenu dans la clé du nœud passé en paramètre est égal à celui du nœud visité.

3.2.2.4) Insertion d'un nœud dans un arbre

Contrairement aux opérations de recherche, l'insertion et la suppression modifient la structure de l'arbre. Cette modification doit se faire en conservant la propriété fondamentale de l'arbre.

Voici l'algorithme général d'insertion d'un nouveau nœud "`newNode`". Seules les informations (champ info) du nœud ont été initialisées à la création du nœud. Cet algorithme explique l'insertion d'un nœud dans l'arbre. Dans la réalité, il est un peu plus complexe dans la mesure où il prend en compte tous les cas possibles d'insertion.

Soit,

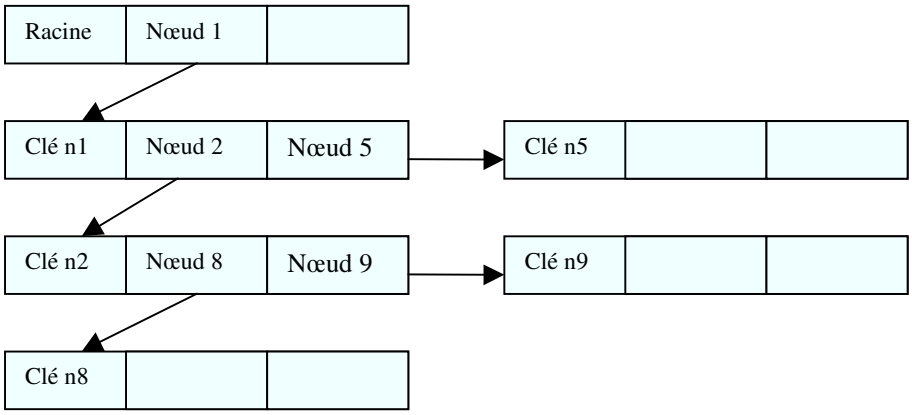
```
Node nodeCourant = racine;
Node nodePrecedent = null;

while( _nodeCourant != null)
{
      Si newNode est père de nodeCourant Alors
      {
Insertion de newNode en tant que père de NodeCourant. L'insertion doit se faire entre le nodePrécédent et le nodeCourant. L'insertion est verticale. Cette insertion entraîne la mise à jour des pointeurs.
Stop le traitement.
      }
      Sinon si newNode est fils du nodeCourant Alors
      {
            nodePrecedent = nodeCourant ;
            nodeCourant = nodeCourant.getFils() ;
            Continue le traitement ;
}
Sinon si (newNode est frère de nodeCourant) et (newNode est inférieur à NodeCourant) Alors
{
Insertion de newNode en tant que frère de NodeCourant. L'insertion doit se faire entre le nodePrecedent et le nodeCourant. Il s'agit ici d'une insertion horizontale. Cette insertion entraîne la mise à jour des pointeurs.
Stop le traitement.
}
Sinon si (newNode est frère de nodeCourant) et (newNode est Supérieur à NodeCourant) Alors
{
nodePrecedent = nodeCourant ;
nodeCourant = nodeCourant.getFrere() ;
Continue le traitement ;}}
```

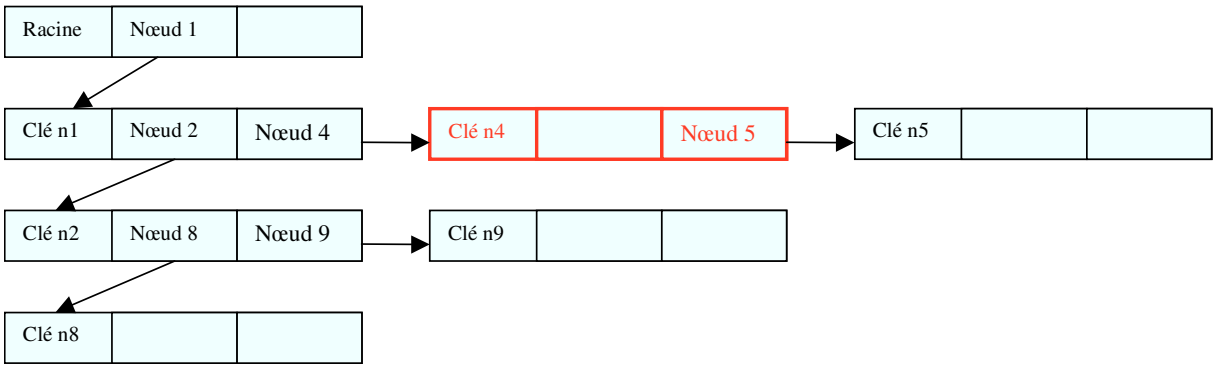
L'idée est simple : comme pour une recherche, on descend dans l'arbre en suivant les frères ou les fils selon les résultats des comparaisons entre la clé du nœud z et les clés rencontrées sur le chemin.

Exemple : cet exemple reprend la logique d'insertion d'un nœud. Nous insérons le nœud 4. Le chiffre étant la clé de l'objet.

Arbre avant insertion :



Arbre après insertion :



3.2.2.5) Suppression d'un nœud

C'est l'opération la plus délicate, car le nœud à supprimer peut se trouver à différents niveaux dans l'arbre, et il faut conserver l'intégrité de l'arbre. Il y a trois cas, selon que le nœud "n" à supprimer :

- (a) est une feuille (ne contient aucun pointeur fils, ni frère),
- (b) n'a qu'un seul fils,
- (c) possède un fils et un frère.

Si "n" n'a pas de fils (cas (a)), on modifie son père pour remplacer le champ fils par null.

Si "n" n'a qu'un fils (cas (b)), on modifie le père de "n" pour que son champ fils pointe directement sur le fils de n.

Le dernier cas, lorsque le nœud "n" a deux fils (cas (c)), est le plus délicat. On va détacher "y", le successeur de "n", et le mettre à la place de "n".

3.3 Aspect graphique des annotations dans le document

Cette partie technique a fait l'objet de nombreuses recherches et sa mise en œuvre n'a pas été simple. Plus précisément, nous avons rencontré des difficultés à dessiner des contours (rectangle, ovale, ..) autour des annotations. La classe `JTextPane` permet de gérer des documents stylés mais dès lors qu'on décide de rajouter d'autres composants graphiques, on est dans l'obligation de compléter voir d'implémenter de nouvelles méthodes. A savoir, qu'il existe très peu de documentation concernant le package javax.swing.text.

3.3.1 Les styles

Les données graphiques sont référencées dans les feuilles de styles dont la syntaxe est inspirée des CSS ce qui facilite leur création et modification par l'utilisateur. Pour chacune des balises définies dans la DTD, un style graphique lui est associé. Au lancement de l'application, les feuilles de styles associées à la DTD sont chargées et stockées en mémoire.

La syntaxe emprunte les méthodes statiques de la classe Java `StyleConstants.` Par exemple, `StyleConstants.setBold=true` affiche les caractères en gras.

Pour la gestion des styles, Cadixe utilise trois types d'objets :

- La classe "StyleCollection": les feuilles de styles associées à une DTD et chargées en mémoire sont stockées dans cet objet ;
- La classe "StyleSheet": cette classe représente directement une feuille de styles. Elle contient deux attributs :

  - StyleSheetName : nom de la feuille de styles ;

  - DicoTagNameStyle : table de hashage qui permet de faire la correspondance entre les noms de balise et les styles.

```
public StyleSheet()
{
String     _styleName = new String("default");
      Hashtable  _dicoTagNameStyle = new Hashtable();
}
```

La classe "Style": fait référence aux attributs graphiques des balises (police, fontes, taille des caractères, …). Cette classe utilise la classe `SimpleAttributeSet` pour l'ajout de nouveaux attributs :

```
SimpleAttributeSet attr=new SimpleAttributeSet();
attr.addAttribute("cadixe-border-shape",new CadixeRectangle());
```

3.3.2 Les annotations

Comme nous l'avons dit dans le chapitre 2, une annotation est représentée par, une balise contenue dans une DTD, une ancre de début ainsi qu'une ancre de fin. Dans Cadixe, une annotation est représentée par :

```
public Annotation(Tag tag,int debut,int fin)
{
      Tag _tag   = tag;      /** correspond a la balise **/
      int _debut = debut;    /** début de la sélection  **/
      int _fin   = fin;      /** fin de la sélection    **/
}
```

Un Tag est identifié par son nom, un style, une liste d'attributs, une DTD ainsi qu'un booléen permettant d'identifier si la balise est de type vide:

```
public Tag(String Name, boolean empty, CadixeDTD dtd){
      if (Name!=null)
            _name=new String(Name);
      _empty = empty;
      _styleTag = new Style();
      _attr = new Vector(10,10);
      _dtd = dtd;      }
```

3.4 Validation du document XML

Au niveau du **document XML**, il existe deux types d'erreurs possibles.

- Le document peut être **non valide**. Dans ce cas, le document n'obéit pas à la DTD ;
- Le document peut être **mal formé**. Dans ce cas, il y a simplement une erreur par rapport au standard XML lui même. Par exemple, des balises mal fermées ou se chevauchant ;

Par contre, au niveau du **parseur SAX**, il existe trois niveaux d'erreurs. Ces trois niveaux d'erreurs sont représentés au niveau du **Gestionnaire d'Erreurs** (*ErrorHandler*) :

- **Erreur fatale** (*fatalError(SAXParseException exception) throws SAXException*) : Cette erreur ne peut être récupérée. C'est le cas par exemple pour un document mal formé. Après ce type d'erreur, le parseur SAX arrête son travail ;
- **Erreur** (*error(SAXParseException exception) throws SAXException*) : Cette erreur peut être récupérée, c'est-à-dire que le parseur SAX peut continuer à traiter le reste du document XML. Ce genre d'erreur peut se produire lors d'une violation d'une contrainte imposée par la DTD ou le schéma ;
- **Warning** (*warning(SAXParseException exception) throws SAXException*) : C'est un simple avertissement. Après cela, le parseur SAX continue le parsing du document.

Ci-dessous l'implémentation faite de la classe *ErrorHandler* dans Cadixe

```
public void warning(SAXParseException exception) throws SAXException{
      _typeError = 2;
      _nbWarning++;
      _message = getMessage(exception);
}
public void error(SAXParseException exception) throws SAXException{
      _typeError = 1;
      _nbError++;
      _message = getMessage(exception);
}
public void fatalError(SAXParseException exception) throws SAXException{
      _typeError = 0;
      _nbFatalError++;
      _message = getMessage(exception);
}
public Vector getMessageError()
{
      return _vMessage;
}
```

4 Phase de Recette et validation

Les phases de tests et de validation ont été réalisées tout au long du cycle de développement de Cadixe. Parmi les tests effectués, on retrouve :

- les tests unitaires réalisés par moi-même afin de m'assurer que les composants sont développés sans erreur de programmation ;
- les tests fonctionnels réalisés conjointement avec Gilles Bisson pour s'assurer que les exigences fonctionnelles sont implantées convenablement ;
- le test du produit afin de s'assurer que le logiciel s'exécute correctement ;
- le test du système afin de vérifier si le logiciel fonctionne sur la machine cible (avec les logiciels et matériel du client) ;
- les tests de performance pour s'assurer que Cadixe est capable de traiter des gros volumes de données ;
- les tests de validation qui sont réalisés par le client pour vérifier le niveau de qualité de l'application.

Chapitre 7

CONCLUSION ET PERSPECTIVES

Dans cette conclusion, nous proposons un bilan qui fait état des actions menées et des réalisations effectuées au cours de cette année de préparation du mémoire d'ingénieur du CNAM. Nous dressons ensuite un panorama des sujets abordés et des perspectives soulevées par ce travail en commençant par faire une analyse critique des choix effectués en 2005.

1 Conclusion et apport personnel

Ce mémoire a été réalisé dans le cadre du cycle C du CNAM. Il a été effectué sous la responsabilité de mon tuteur Gilles Bisson. L'objectif étant de développer un éditeur permettant l'annotation de documents textuels en bio-informatique. En effet, ce mémoire s'insère dans le projet Caderige (Chapitre 2, §1) dont la finalité vise un double objectifs. Sur le plan biologique, il s'agit d'appliquer ces techniques dans le domaine de la génomique fonctionnelle et plus spécifiquement sur celui de la modélisation des interactions géniques. Et sur un plan informatique, il s'agit de développer de nouvelles techniques d'extraction de connaissances dans les bases documentaires écrites en langage naturel. Dans ce contexte, pour faciliter le travail des biologistes, la nécessité d'outiller ce travail d'annotation manuelle, est devenue primordiale. C'est dans cette logique, que le prototype XMLJava et aujourd'hui le logiciel CADIXE ont vu le jour. En terme de spécification fonctionnelle, mon objectif se traduit par la nécessité de proposer un éditeur permettant :

- De lire et de sauvegarder des documents au format XML ;
- De charger en mémoire la DTD associée au document XML que l'on souhaite éditer ;
- D'associer une ou plusieurs feuilles de styles à un même document ;
- D'appliquer des annotations sur le texte, de saisir des attributs et d'éventuels commentaires ;
- Fournir toutes les fonctionnalités de base d'un éditeur de texte classique.

Ce mémoire a été effectué selon une démarche comportant 4 phases :

Un état de l'art

Pour réaliser nos objectifs, nous avons commencé par faire une étude sur les éditeurs XML ou éditeurs d'annotations existants sur le marché. Nous nous sommes intéressés aux éditeurs gratuits et offrant la possibilité d'ajouter de nouvelles fonctionnalités. Ce tour d'horizon nous a permis de montrer que seul XMLJava re-baptisé Cadixe répondait à nos exigences. Ce qui nous a conduit à, dans un premier temps corriger les bugs existants dans XMLJava puis de travailler sur une nouvelle version de l'éditeur.

Le choix d'une méthode de développement

Le choix d'une méthode de développement s'est avéré nécessaire pour mettre en œuvre notre application. Nous avons donc choisi d'utiliser le modèle en spirale. En effet, ce modèle était semblable à nos méthodes de travail. Avec le recul, nous aurions également pu utiliser les nouvelles méthodes telles que XP ou RAD.

Le recueil des besoins et analyse

Nous avions dans ce cadre rencontré des biologistes du SIB à Genève, utilisateurs de XMLJava, pour mieux comprendre leurs besoins et leurs attentes. En effet, à cette occasion, nous avons pris connaissance de leurs méthodes de travail, en les accompagnant dans leurs tâches d'annotations de documents. Le retour d'expérience sur l'utilisation d'XMLjava s'est avéré essentiel dans la mesure où les biologistes ont pu exprimer leurs besoins de façon efficace et stable. L'ensemble des fonctionnalités de XMLjava a été maintenu. Cependant, l'interface graphique a été révisée pour offrir aux utilisateurs une meilleure utilisation et de nouvelles fonctionnalités ont vu le jour, notamment le chargement de plusieurs fichiers en mémoire, le copier/coller d'annotations, la gestion des erreurs… (Chapitre 5)

La conception et l'implémentation de l'éditeur

Les phases de conception et d'implémentation de l'éditeur ne se sont pas déroulées sans difficultés, du fait d'un manque de connaissance préalable des outils de développement et du langage de programmation Java. En effet, l'application est entièrement écrite en Java, de ce fait elle est exécutable sur plusieurs plates-formes : Linux, MacOs X et Windows. Pour construire notre éditeur en un temps raisonnable, nous avons utilisé les API Java qui contiennent des éléments pour construire un éditeur de texte sur mesure, ainsi les composants Swing nous ont permis de modéliser rapidement notre application. Les supports disponibles que sont SAX et DOM nous ont facilité le travail. Dans notre cas, nous avons utilisé SAX qui est capable de lire, de construire un document XML et de le valider en un temps acceptable pendant l'édition. L'utilisation des feuilles de styles de type CSS nous a permis de gérer l'aspect graphique des annotations.

En dépit des problèmes rencontrés, l'application a été mise à disposition des utilisateurs dans le délai imparti avec une prise en compte des besoins identifiés dans le cahier des charges.

Ce stage m'a permis d'être confrontée à un panel de technologies, recommandations et problématiques très variées. Notamment dans l'apprentissage du langage Java, l'utilisation des parseurs, etc. Un challenge qui fut également une difficulté a été de travailler seule sur les différents sujets et sur des technologies que j'avais peu approfondies. J'en profite donc pour remercier tous les internautes des forums qui ont pris le temps de répondre à des problématiques rencontrées souvent résumées en quelques phrases. Outre l'aspect technique, d'un point de vue personnel, travailler au sein d'une équipe de recherche a été très enrichissant et m'a permis d'appréhender les problèmes différemment en me permettant de développer une plus grande ouverture d'esprit. En effet, le domaine de la recherche exige une remise en cause fréquente des méthodes et des solutions adoptées, ce qui est rarement le cas dans le milieu industriel où on est souvent challengé par les composantes : planning et budget.

Ce stage s'est déroulé en 2005 et se termine en 2010. Me replonger dans la rédaction de mon mémoire après ces années de recul m'a permis d'avoir un œil critique sur mes choix de l'époque. Pour cela je tiens à remercier à nouveau Gilles Bisson pour tout le temps qu'il m'a accordé pendant et après le stage. Merci.

2 *Perspectives en 2010*

Bien que le développement du logiciel ait été achevé en 2005, il est toujours régulièrement utilisé par l'unité MIG de l'INRA Jouy en Josas pour annoter des documents biologiques avec différentes DTD. En outre, comme le logiciel est documenté dans un site WEB libre d'accès : http://caderige.imag.fr/Cadixe/index.html, de nombreuses personnes demandent régulièrement de pouvoir télécharger l'application. Cela montre amplement que ce développement répond non seulement à un besoin mais que celui-ci est toujours présent, même si comme nous le verrons dans le §4 de ce chapitre de nouveaux outils sont apparus entre temps.

Toutefois même si la version actuelle de Cadixe continue à être d'actualité, la technologie a beaucoup évolué depuis 2005 et il est clair que si le développement du logiciel devait être repris aujourd'hui[9] d'autres choix d'implémentation seraient sans doute plus pertinents. Par ailleurs au cours de l'utilisation nos partenaires ont mis en lumière de nouveaux besoins ou ont proposé diverses améliorations. Nous allons maintenant examiner ces deux aspects.

3 *Nouvelles visions pour le projet*

3.1 Choix technologique

L'une des principales difficultés du projet, en terme de coût de développement, a été d'implémenter et de tester le composant chargé de l'affichage des documents annotés et de l'interaction (ajout, sélection, suppression) avec les balises. Or, les fonctionnalités nécessaires à ces tâches correspondent assez exactement à ce qui est réalisé dans un navigateur WEB lorsqu'il affiche des pages HTML. Aujourd'hui des moteurs d'affichage « open-source », ayant des API stables, sont disponibles ce qui n'était pas vraiment le cas en 2005. Les deux plus connus sont d'une part, « WebKit » (§4.3.1, à l'initiative de la société Apple et « Gecko » (§4.3.2 de la fondation Mozilla. Ces deux moteurs constituent le cœur de l'ensemble les navigateurs Web modernes si l'on fait exception d'Internet Explorer de la société Microsoft. L'utilisation de ces librairies comme base de développement de Cadixe présenterait de très nombreux avantages :

- Simplification très importante du code de Cadixe et possibilité de consacrer d'avantage de temps aux fonctionnalités spécifiques correspondant à la tâche d'annotation ;
- Prise en compte de l'intégralité des possibilités d'affichage du CSS 3.0 et non plus juste un sous-ensemble comme ce qui est fait dans le système actuel. Ainsi l'utilisateur de Cadixe aurait des possibilités bien plus étendues pour représenter les balises ;
- Performances et possibilités du système s'accroissant automatiquement lorsque de nouvelles versions plus puissantes des moteurs d'affichage sont disponibles ;
- Possibilité de transformer Cadixe en « application Web » ce qui éviterait aux utilisateurs d'avoir à installer le logiciel et d'avoir à gérer les fichiers sur leur poste. Cet aspect est particulièrement pertinent dans le cas où l'annotation se fait de manière collaborative au sein d'un laboratoire comme cela se passe, par exemple, au SIB à Genève.

[9] Notons à ce sujet que l'équipe à laquelle appartient Gilles Bisson a proposé cette année dans le cadre d'un projet ANR (Agence Nationale de la Recherche) une tâche pour développer un nouvel outil d'annotation basé sur Cadixe et qui doit intégrer dans une large partie les propositions qui sont faites dans ce chapitre.

Nous présentons les différents aspects de cette nouvelle architecture dans la section 4 de cette partie en examinant les avantages comparés des librairies « WebKit » et « Gecko » et en examinant comment implémenter le protocole de communication entre l'application et un serveur WEB.

3.2 Nouvelles fonctions

En utilisant Cadixe les utilisateurs ont très naturellement imaginé de nouveaux usages pour le logiciel et notamment cherché à l'utiliser pour réaliser des tâches d'annotations plus complexes que l'on trouve assez souvent dans les documents en biologie.

3.2.1 Annotation de zones complexes

La version actuelle de Cadixe permet de faire de l'annotation selon une structure hiérarchique qui est naturellement celle d'un document XML : les balises découpent (partitions) de manière récursive le texte en unités disjointes. Cependant lorsque l'on annote un document sa structure sémantique réelle ne suit pas toujours une logique aussi stricte. Deux types de problèmes peuvent se rencontrer soit séparément soit de manière conjointe :

- *Balisage discontinu*. Lorsqu'on introduit une balise il peut arriver que la zone qui doit être couverte par la balise ne soit pas continue mais segmentée dans le paragraphe. Si l'on prend l'exemple suivant qui est fictif mais représentatif du type de phrases annotées :

  « Les protéines Green et Plc, qui sont impliquées dans la synthèse de la chlorophylle, lorsqu'elles sont méthylées ***inhibent*** les gènes TYPE et PLC »

  Dans cette phrase si l'on annote les relations d'activation (ou d'inhibition) d'un gène par un autre avec les trois balises <AGENTS>, <RELATION> et <CIBLES> on voit que la balise <AGENTS> doit couvrir les termes « Les protéines Green et Pic … méthylées» mais pas la relative qui ne fait que mentionner le rôle du gène sur la chlorophylle. La solution actuelle consiste à introduire plusieurs fois la balise <AGENTS> avec la même valeur pour attribut d'identification ID, toutefois cela complique la tâche de l'utilisateur.

- *Balisage avec recouvrement*. Parfois certaines parties du texte peuvent appartenir à plusieurs balises simultanément qui se chevauchent. Ainsi, dans l'exemple précédant l'inhibition porte sur les gènes TYPE et PLC mais comme ce dernier, en s'exprimant, produit la protéine de même nom (Plc), l'annotateur pourrait considérer qu'elle devrait apparaître à la fois dans la balise <AGENTS> et <CIBLES> (on a une boucle de rétroactions). Cela conduit au code XML suivant : <AGENTS> … <CIBLES> … </AGENTS> … </CIBLES>. Or ce type d'imbrication n'est évidemment pas autorisé dans le langage XML. Pour résoudre efficacement ces deux cas de figure, il faut modifier la manière dont le code XML est géré dans Cadixe et mettre en place ce qui est classiquement appelé du « XML standoff » (ou XML déporté). L'idée est la suivante : on met explicitement dans les balises XML les positions de début et fin de caractères des différentes parties balisées. Voici un exemple pour le cas précédent :

```
<agents ...>
        <fragment begin=450 end=472>Les protéines Green et Plc</fragment >
        <fragment begin=512 end=520>méthylées</fragment >
</agents>
<cibles ...>
        <fragment begin=453 end=462>protéines</fragment >
        <fragment begin=469 end=472>Plc</fragment >
        <fragment begin=532 end=547>gènes TYPE et PLC</fragment >
</cibles>
```

Dans ce codage les zones de texte non-contiguë (délimitées par la balise <FRAGMENT) des balises <AGENTS> et <CIBLES> restent enchevêtrées (les indices des positions se chevauchent) mais la structure du code XML reste par contre totalement hiérarchique donc valide. Évidemment, l'affichage des annotations est plus complexe à réaliser dans l'éditeur puisque il faut maintenant passer par une phase « d'interprétation » des positions pour savoir comment gérer les styles. De même les opérations d'ajout, ou de suppression, de caractères dans le document sont un peu plus délicates puisqu'elles nécessitent une mise à jour des positions. Cependant, avec cette méthode, on bénéficie d'une très grande liberté dans le type d'annotation qui peut être mis en place.

Du point de vue de l'implémentation de l'éditeur, il devient sans doute plus intéressant d'utiliser les « XML Schema » décrit dans la section - plutôt que les seules DTD pour décrire les grammaires d'annotation ; même si le support des DTD reste nécessaire pour assurer la compatibilité ascendante. Concernant l'interface, il faudra aussi imaginer une manière élégante de montrer le lien qui existe entre les différents fragments d'une balise. La solution qui consiste à dessiner à l'écran des traits entre ces fragments, comme cela est fait dans le logiciel GLOZZ (cf § 3.3.1), n'est en effet pas satisfaisante lorsque le nombre de fragments devient trop important.

3.2.2 Annotation de documents complexes

Dans la version actuelle de Cadixe, l'éditeur ne gère en entrée que du texte « brut », c'est-à-dire sans enrichissement typographique (gras, italique, etc). Il n'est pas capable nom plus de gérer des contenus complexes comme des tableaux et des images. Or très souvent les biologistes voudraient annoter directement des articles. L'utilisation d'un moteur d'affichage standard ouvrira cette possibilité puisqu'on aura accès à toute la richesse de représentation du HTML et des CSS. La seule limitation reste que le codage des documents à annoter doit être effectué en HTML.

3.2.3 Aide à l'annotation collaborative

Dans sa version actuelle, le logiciel offre une aide à l'annotation collaborative par l'intermédiaire des commentaires (Chapitre 6, §2.2.8). Les utilisateurs aimeraient cependant avoir plus de possibilités et notamment pouvoir partager la détection des « entités nommées ». En effet, lorsqu'on annote un texte biologique, il y a une forte variabilité dans certains noms de gènes selon les auteurs. Par exemple le gène « Sigma K » peut s'écrire aussi : « SigmaK », « sigK », « facteur K », etc. Toutes ces notations ont le même sens et devraient donc être annotées de façon identique.

Une solution serait la suivante : il faudrait gérer une base de données au niveau d'un laboratoire dans laquelle on garderait l'ensemble des annotations qui ont été effectuées sous la forme de couples (caractères annotés, balise). Lorsqu'un annotateur sélectionnerait dans l'éditeur une zone de document pour l'annoter, l'éditeur interrogerait alors cette base afin de savoir si une chaîne de caractères identique (ou similaire) n'a pas déjà été annotée par une autre personne. Si c'est le cas, il proposerait alors à l'utilisateur d'utiliser la même annotation. Un tel système permettrait d'accélérer énormément la vitesse d'annotation et de diminuer le nombre d'erreurs de saisie. En outre, on constituerait automatiquement une base recensant toutes les variations de la terminologie.

3.3 Actualisation de l'état de l'art

L'état de l'art réalisé en 2005 peut être actualisé en évaluant quelques nouveaux produits.

3.3.1 Projet GLOZZ

La plate-forme d'annotation manuelle Glozz [GLOZ], développée au Greyc dans le cadre du projet ANR Annodis, permet de travailler à différents niveaux de grain sur un même document. Cette plate-forme est suffisamment générique pour s'adapter aux modèles d'annotation utilisés dans les différentes campagnes d'Annodis, et peut probablement être utilisée à d'autres fins. Ce logiciel présente des aspects qu'il serait intéressant d'intégrer dans une nouvelle version de Cadixe. La figure suivante représente la fenêtre principale de l'application.

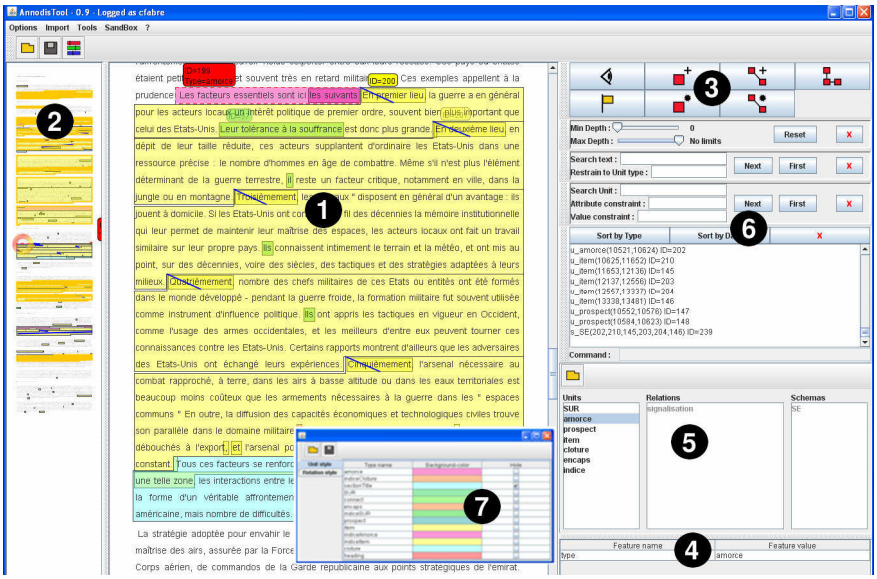


Figure 41 : La plate-forme d'annotation Glozz

Les quatre points suivants permettent d'en comprendre les principes :

- L'interface dispose simultanément de deux « vues » sur le texte de travail. La vue principale [1] est la fenêtre de travail, dans laquelle les annotations sont saisies à la souris, tandis que la vue [2], appelée « ruban », présente le texte 'vu de haut'. Dans ces deux vues, les indices prédéfinis sont associés à des styles. De tels styles sont également appliqués aux annotations effectuées. Ainsi, la vue "ruban" permet une appréhension globale du marquage, et la navigation rapide au sein des zones de concentration d'indices. Les vues locales et macroscopiques peuvent être utilisées conjointement. Si l'on souhaite par exemple établir une relation entre deux unités distantes, on pourra repérer les deux unités en zone (2), et les sélectionner précisément en zone (1) ;
- Une boîte à outils [3] permet de choisir parmi les trois types d'objets d'annotation. Les «unités» sont des blocs (apparaissant sous forme de cadres colorés dans [1] et [2]). Les «relations» permettent de relier deux unités (apparaissant sous forme de lignes transverses dans [1] et [2]), et les «schémas» permettent d'agglomérer unités et relations en une même entité. On a donc une grande richesse dans la manière dont on peut introduire les annotations et surtout une meilleure structure que dans Cadixe ;

- Pour une campagne donnée, ou pour l'une de ses sous-tâches, un modèle d'annotation peut être chargé afin de définir les catégories disponibles pour chacun des trois types d'objet (par exemple des unités de catégorie «Amorce», des relations de catégorie «Elaboration», des schémas de catégorie «Structure énumérative», etc.). Ces catégories apparaissent dans les trois colonnes de [5], qui permettent d'assigner la catégorie souhaitée à chaque annotation. Le modèle d'annotation prévoit aussi d'associer un jeu d'attributs/valeurs à chaque catégorie, lequel peut être vu et modifié dans la fenêtre [4]. Des styles (notamment des couleurs) sont associés aux différents types, et peuvent être modifiés grâce à l'interface [7] ;
- Enfin, différents outils de recherche et de navigation [6] peuvent être ajoutés ou retirés à la demande.

Une des particularités de Glozz est que l'ensemble des annotations décrites au format XML sont stockées dans un fichier à part. On parle dans ce cas de description des données "standoff", c'est-à-dire que le fichier XML ne comporte pas lui-même les données textuelles sur lesquelles il positionne des annotations, mais fait référence à des positions dans le texte en question. Par exemple, une annotation nommée "unité1" ira du caractère 3045 au caractère 3072.

3.3.2 Projet MMAX2

MMAX2 a été conçu pour annoter manuellement les corpus au niveau référentiel, et plus précisément les relations de coréférence, d'anaphore associative et les corpus multimodaux. Dans une fenêtre de l'application, on peut lire le texte à annoter. Par un simple système de sélection de caractères à l'aide de la souris, on insère des balises XML qui permettent d'identifier le type de relation anaphorique. Ci-dessous une copie de l'écran principal :

Figure 42 : Fenêtre principale de l'application MMAX2

Contrairement à l'outil GLOZZ, il semble que MMAX2 ne soit plus dans une phase de développement actif. Par ailleurs son interface, plus élémentaire, rend l'annotation plus délicate pour les utilisateurs novices. A ce propos, on peut souligner que l'interface de CADIXE est généralement très bien perçue offrant un compromis ergonomie/possibilités jugé intéressant.

3.3.3 Projet WebAnnot

Ce projet s'est déroulé dans le cadre d'un stage de fin d'étude (2008 – 2009) en vue d'obtenir un diplôme d'ingénieur de l'Ecole nationale Supérieur d'Informatique (ESI) en Algérie [WEBA]. Brièvement, le projet consiste à proposer un outil d'annotation dédié à l'enseignant. Les enseignants annotent souvent des documents pédagogiques afin de mémoriser des idées directement sur le document. Vu que les enseignants utilisent de plus en plus des documents numériques, il est devenu important de leur mettre à disposition un outil leur offrant cette possibilité d'annotation et surtout la possibilité d'échanger, de partager avec d'autres enseignants. C'est dans ce contexte que s'insère ce stage. Il vise donc à réaliser un outil d'annotation pédagogique personnel qui permet d'annoter directement les documents numériques. L'outil offre à l'annotateur différentes formes d'annotations. Chaque annotation a une sémantique explicite et un contexte de création. Il permet à l'enseignant d'annoter de deux manières :

- Manuellement, où le processus de création est complètement à la charge de l'enseignant qui sélectionne l'ancre où poser l'annotation, ensuite il sélectionne la forme d'annotation et enfin il remplit les informations de la facette sémantique et décide de la créer ;
- Semi-automatique, qui signifie qu'un des trois processus de la création d'annotation est exécuté par l'outil d'annotation en utilisant des patrons.

L'outil d'annotation est réalisé sous forme d'une extension FireFox qui ajoute à ce navigateur les fonctionnalités d'annotation. Les composants de FireFox sont détaillés un peu plus loin dans le §4.3.2.

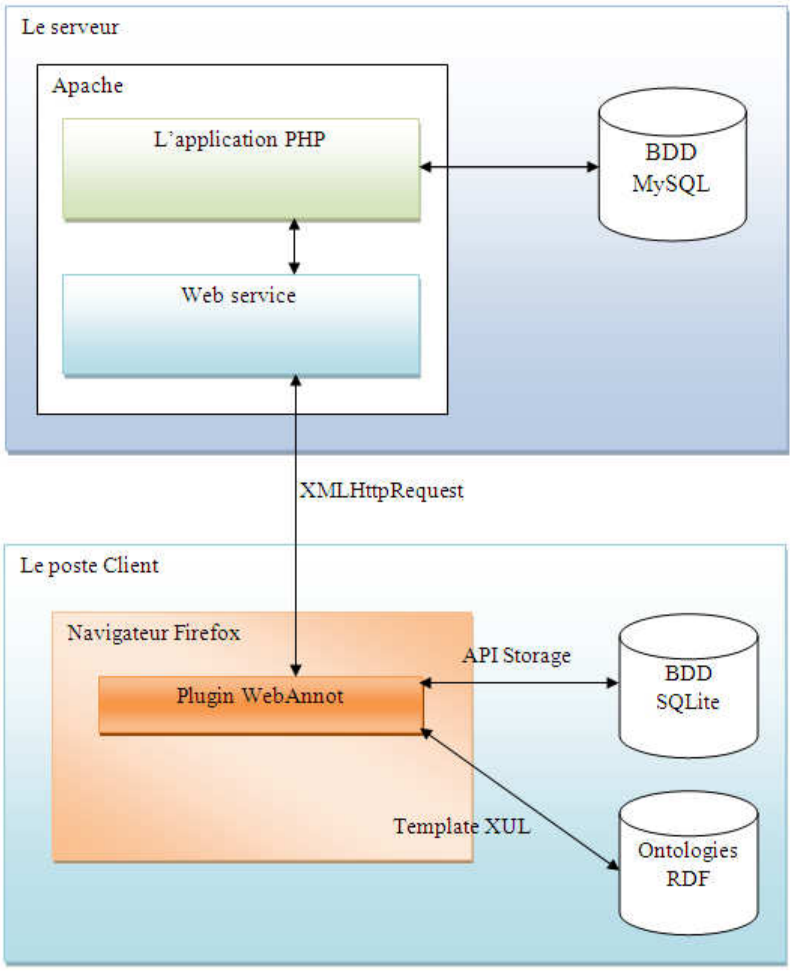


Figure 43 : Architecture de l'application WebAnnot

4 Cadixe nouvelle version – architecture cible

Dans l'optique d'une nouvelle version, comme nous l'avons dit au début de ce chapitre, il serait intéressant de se décharger de l'aspect graphique. L'objectif serait donc d'implémenter les fonctions de bases de Cadixe et les nouvelles fonctionnalités, en nous appuyant sur des nouvelles technologies utilisées pour le développement d'applications web. L'architecture du projet WebAnnot reflète bien la solution que nous pourrions mettre en œuvre. Nous donnons ci-dessous une proposition d'architecture logicielle à étudier dans le cas où la réponse à l'appel d'offre ANR s'avérerait positive :

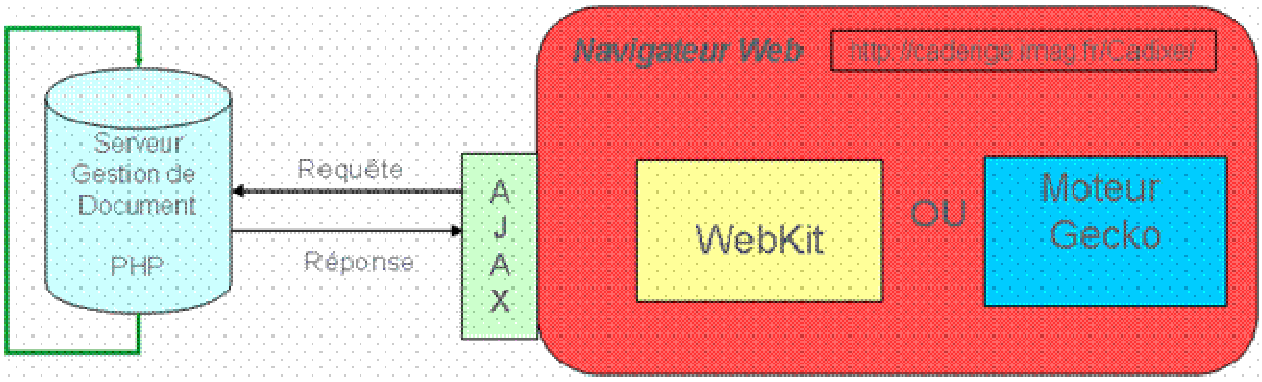


Figure 44 : Architecture logicielle cible pour une version future de Cadixe utilisant Webkit ou Gecko

- Structure des fichiers : XML, XML-Schema et éventuellement XHTML
- Serveur web : Apache2 et comme langage de programmation PHP (Pre Hypertext Processor) et PHP-XML pour parser les fichiers
- Interface Graphique : Javascript, WebKit ou Gecko et enfin AJAX pour communiquer avec le serveur et gérer les requêtes et les interactions.

Chacun des composants de l'architecture logicielle proposée ci-dessus est détaillé dans les paragraphes qui suivent.

4.1 Structure des fichiers

Concernant le format des documents manipulés, XML reste clairement le plus adapté, nous ne reviendrons pas sur ce choix. Pour plus d'informations, se référer au chapitre 2. Cependant, pour créer des modèles de documents, deux recommandations existent ; Les DTD et les schémas XML :

- La première sur les DTD est historique et est devenue reconnue dans tous les domaines applicatifs, c'est d'ailleurs pour cette raison que nous l'avons utilisée pour la version actuelle de Cadixe ;
- La deuxième XML Schema qui vient palier les déficiences de la première approche en ce qui concerne notamment la gestion du typage de données, du langage utilisé et du support des espaces de noms.

En effet, XML Schema qui est un nouveau langage proposé par le W3C, propose, en plus des fonctionnalités fournies par les DTD, plusieurs nouveautés à savoir :

- un grand nombre de types de données comme les booléens, les entiers, les intervalles de temps, etc. ce qui permet de mieux contrôler les annotations effectuées en proposant à l'utilisateur des boîtes de saisie (ou « widget ») adaptées au type de données ;
- des types de données utilisateurs qui nous permettent de créer notre propre type de données nommées. Ces nouveaux types peuvent être également créés par simple ajout de contraintes sur un type déjà existant ;
- la notion d'héritage : Les éléments peuvent hériter du contenu et des attributs d'un autre élément. C'est sans aucun doute l'innovation la plus intéressante de XML Schema ;
- le support des espaces de nommage ;
- les indicateurs d'occurrences des éléments peuvent être tout nombre non négatif ;
- une grande facilité de conception modulaire de schémas ce qui est particulièrement intéressant dans le cas de grammaires d'annotations complexes.

Pour une version future de Cadixe, il serait donc plus intéressant de partir sur les XML Schema. Le problème de la compatibilité ascendante avec l'outil actuel peut-être géré aisément car il existe actuellement des outils sur le net qui permettent de convertir les DTD existantes en XML Schema.

4.2 Serveur web - PHP

PHP (Hypertext Preprocessor) est un langage interprété (un langage de script) exécuté du côté serveur et non du côté client. La syntaxe du langage provient de celles du langage C, du Perl et de Java. Ses principaux atouts sont :

- Une grande communauté de développeurs partageant des centaines de milliers d'exemples de script PHP ;
- La gratuité et la disponibilité du code source ;
- La simplicité d'écriture de scripts ;
- La possibilité d'inclure le script PHP au sein d'une page;
- La simplicité d'interfaçage avec des bases de données (le plus utilisé avec ce langage est MySQL, un SGBD gratuit disponible sur de nombreuses plates-formes) ;
- L'intégration au sein de nombreux serveurs web (Apache, Microsoft IIS, etc.).

Un script PHP est un simple fichier texte contenant des instructions écrites dans un code HTML à l'aide de balises spéciales et stocké sur le serveur. Ce fichier doit avoir l'extension « .php » pour pouvoir être interprété par le serveur. Ainsi, lorsqu'un navigateur (le client) désire accéder à une page dynamique réalisée en PHP :

- le serveur reconnaît l'extension d'un fichier PHP et le transmet à l'interpréteur PHP ;
- Dès que l'interpréteur rencontre une balise indiquant que les lignes suivantes sont du code PHP, il ne lit plus les instructions: il les exécute ;
- L'interpréteur exécute l'instruction puis envoie les sorties éventuelles au serveur ;
- A la fin du script, le serveur transmet le résultat au client (le navigateur).

Le schéma ci-dessous illustre son fonctionnement.

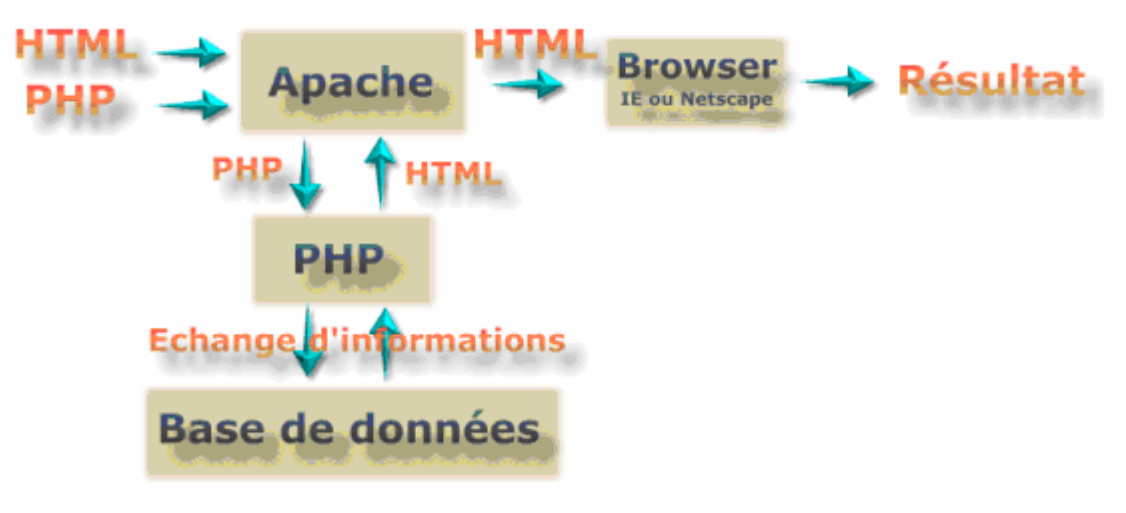


Figure 45 : Echanges de donnée en utilisant des pages HTML ou des scripts PHP

Le format HTML n'est pas toujours le plus approprié pour répondre à notre besoin d'annotation, cependant tout l'affichage doit passer par lui si l'on utilise un moteur de rendu « standard ». Il est dont nécessaire de pouvoir adapter, côté client, les données envoyées au format XML par le serveur. Coté serveur, nous devons être en mesure d'envoyer et de lire un format XML (Chapitre 2, §2). Contrairement au langage HTML le langage XML permet ainsi de définir nos propres balises, ce qui permet de séparer la présentation du document de son contenu.

Cette séparation entre le contenu et la présentation se fait à l'aide d'un analyseur syntaxique (parseur), c'est-à-dire un programme capable de vérifier la cohérence de la syntaxe du document et de l'interpréter afin de mettre en page son contenu. PHP propose une extension permettant de mettre au point facilement des analyseurs XML. Cette extension utilise la librairie *expat* disponible à http://www.jclark.com/xml/. Pour activer le support de cette librairie lors de l'installation de PHP il suffit de lancer la configuration de PHP avec l'option *--with-xml*. Les parseurs possibles sont DOM et SAX largement abordés dans le Chapitre 6, §1.3. Nous avions retenu SAX pour le développement de Cadixe, et il semble que cette solution soit toujours la plus adaptée dans une version ultérieure. Nous ne rentrerons pas dans le détail, mais pour que cela fonctionne il suffit de créer une instance de parseur grâce à la fonction *xml_create_parser()*.

L'installation de PHP sous Windows peut se faire via un package (appelé *EasyPHP*) contenant 3 produits incontournables :

- Le serveur Web Apache ;
- Le moteur de scripts PHP4 ;
- La base de données MySQL ;
- Un outil de gestion de base de données graphiques, Phpmyadmin.

EasyPHP est ainsi un paquetage fonctionnant sous Windows permettant d'installer facilement les éléments nécessaires au fonctionnement d'un site web dynamique développé en PHP

EasyPHP est disponible sur les sites suivants : www.manucorp.com et www.easyphp.org.

4.3 Gestion de l'interface Graphique

Pour la gestion de l'interface graphique, nous proposons d'utiliser les logiciels suivants :

4.3.1 WebKit (Safari)

Sur la plate-forme MacOs, Internet Explorer jouissait d'un quasi-monopole en 2003. L'abandon progressif du support de ce logiciel par Microsoft a incité la société Apple à écrire son propre navigateur, nommé Safari. Grâce à ce nouveau navigateur, Apple a gagné son indépendance pour contrôler les évolutions de sa vision Internet, non seulement sur le Mac, mais encore où bon lui semble [VAU07]. En 2007, en même temps que l'iPhone est commercialisé dans la version 3 de Safari, ce même navigateur est proposé sur PC.

Lors de la conception de Safari, Apple n'a pas cherché à créer de toute pièce un navigateur de plus. Les ingénieurs ont évalué les produits Open Source existants et leur choix s'est arrêté sur deux solutions complémentaires réalisées pour l'environnement KDE (http://fr.kde.org/) car leur API était relativement simple comparativement à celle du navigateur Firefox développé par la fondation Mozilla (cf §4.3.2):

- KHTML l'interpréteur HTML, nommé WebCore dans Mac OS X ;
- KJS interpréteur JavaScript, nommé JavaScriptCore dans Mac OS X.

Les modifications apportées par Apple à ces solutions sont, à leur tour, disponibles en Open Source. Elles sont principalement de deux ordres :

- Des optimisations pour rendre ces interpréteurs les plus efficaces possible;
- L'ajout d'un maximum des particularités d'Internet Explorer Windows et surtout la prise en compte des différentes évolutions des standards du W3C. La version 3 de Safari supporte :
  - HTML 4.01 ;
  - XHTML 1.0 ;
  - CSS 2.1 et une très large partie de CSS3.xx ;
  - JavaScript 1.4 avec support du DOM ;
  - Les technologies AJAX et XMLHTTPRequest.

En plus de ces modifications, Apple a inséré WebCore et JavaScriptCore dans un *FrameWork* Objective-C. Ce *FrameWork* [10] s'appelle "**WebKit**" et il permet la parfaite intégration à Mac OS X, aussi bien pour l'utilisateur final que pour le développeur. Actuellement cette librairie a été portée sur d'autres environnements (QT, etc) et est utilisée notamment par la société Nokia pour ses navigateurs pour téléphones mobiles ainsi que par Google pour son navigateur Chrome.

La principale classe de WebKit est l'objet `WebView` qui implémente à la fois une vue graphique de l'interprétation HTML et un environnement d'exécution JavaScript. Safari se présente alors simplement comme une application dont les fonctionnalités sont essentiellement une interface utilisateur autour d'une `WebView`.

[10] Un FrameWork désigne un ensemble de classes, au sens de la programmation objet, pour former une structure.

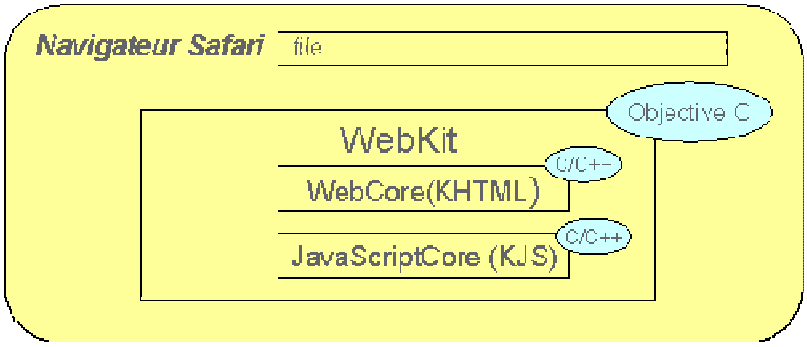


Figure 46 : Safari et le webkit

Les versions avancées du WebKit

Des versions avancées, ou Nightly builds, sont le résultat des toutes dernières évolutions du WebKit par le développeur s'intéressant à ce projet Open source. Ces versions permettent de percevoir les futures directions de cette technologie et les nouvelles fonctionnalités ainsi proposées sont ensuite incorporées, après validation dans la version publique du WebKit.

Ainsi des fonctions avancées peuvent être disponibles dans les Nightly builds mais pas encore dans la version publique. Un système de chargement dynamique du dernier framework permet d'accéder simultanément à la version publique et à la version avancée. On peut donc installer ces versions sans compromettre son système. Le site de téléchargement des Nightly builds est : http://nightly.webkit.org/

4.3.2 Gecko (Firefox)

Le développement de Gecko a commencé en 1997 et a évolué vers un moteur de rendu léger, rapide et robuste, caractérisé par une architecture ouverte, portable, extensible et personnalisable. Le moteur de rendu Gecko, intégrable et multi plates-formes, est le coeur de Mozilla. Il n'a aucune interface utilisateur; il décode simplement le contenu Web et l'affiche. Il est aussi utilisé par d'autres navigateurs web comme Galéon, Epiphany, etc.

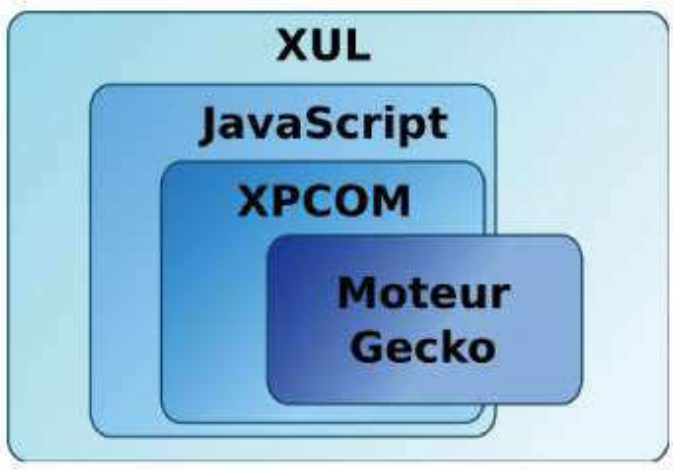


Figure 47 : Les différents composants de firefox

Le but principal de Gecko est d'afficher un fichier HTML ou XML selon les styles indiqués dans un fichier CSS. Gecko propose également :

- une API DOM ;
- un système de plugin (pour afficher des documents comme du flash, de la vidéo, ...) ;
- un interpréteur javascript ;
- une bibliothèque d'accès au réseau, Necko (utile pour charger les pages web).

XPCOM (Cross-Platform Component Object Model) est le modèle de composants de Mozilla. Ce modèle permet de développer des composants dans différents langages (C++, C, JavaScript, Python et d'autres) et de les faire communiquer. La plupart des fonctionnalités du moteur Gecko sont visibles sous forme de composants XPCOM.

Mozilla Firefox embarque le moteur SpiderMonkey capable d'interpréter et d'exécuter des instructions JavaScript. Elles utilisent la technologie XPConnect (Cross Platform Connect) pour manipuler les composants XPCOM de Firefox et accéder aux fonctionnalités du navigateur. Le code JavaScript fait donc le lien entre l'interface graphique décrite en XUL (XML User Interface Language) et les composants XPCOM de Gecko.

XUL, qui se prononce "zoul", est un langage de description d'interfaces graphiques, basé sur le standard XML. Il s'utilise conjointement avec CSS, DOM et JavaScript. Il permet de décrire et de positionner en XML des composants graphiques ou widgets (fenêtres, boutons, listes, menus, zones d'édition, etc.), de leur associer des évènements, des styles et des sources de données, ainsi que de les modifier via des scripts.

4.3.3 Pour Cadixe : Gecko ou WebKit ?

Si l'on examine les intérêts réciproques de Gecko et du WebKit pour le développement du moteur d'affichage de Cadixe, on peut faire les remarques suivantes :

Dans les deux cas, il s'agit de librairies stables et faisant l'objet de développements importants. En 2003, Apple avait choisi KHTML (le futur WebKit) car son API et son code était plus simple, mais il faudrait voir si cet avantage est toujours présent en 2010, des études supplémentaires seraient nécessaires sur ce point.

Du point de vue de la « notoriété » des deux environnements chacun à des point forts et faibles. Clairement en termes de pourcentage dans le marché des navigateurs c'est Gecko (via Firefox) qui est clairement la librairie la plus répandue, cependant l'adoption du Webkit par un grand nombre d'acteurs principaux du marché (Apple, Google, Nokia, …) est un point important. De plus c'est cette dernière librairie qui obtient les meilleurs scores dans les tests de vitesse de JavaScript , ce qui est utile dans le cas d'une application interactive comme Cadixe.

Pour l'instant et sous réserve d'études plus complètes, nous aurions tendance à privilégier un développement basé sur le WebKit.

4.4 JQuery et Ajax

Pour optimiser les échanges entre le client et le serveur nous utiliserons Ajax (Asynchronous JavaScript and XML), qui permet de d'envoyer des requêtes au serveur et de récupérer une réponse sans rafraîchir complètement la page mais seulement les parties qui nous intéressent [DAR08].

Ajax est apparu il y a déjà quelques années et est un ensemble de technologies libres couramment utilisé dans les applications web comme :

- HTML ;
- CSS pour la présentation de la page ;
- JavaScript pour les traitements locaux, DOM ou SAX qui accède aux éléments de la page ou du formulaire ou aux éléments d'un fichier xml pris sur le serveur en s'appuyant sur l'objet XMLHttpRequest ;
- PHP ou un autre langage de scripts peut être utilisé côté serveur.

L''utilisation d'Ajax donne à l'utilisateur une impression d'interactivité beaucoup plus grande. En effet, une bonne répartition des ressources doit solliciter les postes clients, plutôt que le serveur et le réseau. Ajax permet d'effectuer des traitements sur le poste client (avec JavaScript) à partir d'informations prises sur le serveur. Il permet de modifier partiellement la page affichée par le navigateur pour la mettre à jour sans avoir à recharger la page entière. Par exemple le contenu d'un champ de formulaire peut être changé, sans avoir à recharger la page avec le titre, les images, le menu, etc.

Ajax couplé à Jquery permet de créer en quelques lignes des requêtes AJAX [CHA09, DEF10,]. jQuery est une bibliothèque JavaScript libre très pratique, ayant une syntaxe courte et logique, compatible avec tous les navigateurs courants. Il est également possible, à l'aide de paramètres simples, de personnaliser le formatage des requêtes AJAX. De plus, il facilite le changement ou l'ajout d'une classe CSS, la création des animations, modification des attributs, etc.

5 Epilogue

Dans le cadre d'un projet ANR, l'équipe à laquelle appartient Gilles Bisson a proposé une offre pour développer un nouvel outil d'annotations basée sur l'application Cadixe, qui prend en compte les nouvelles fonctions demandées par les clients ainsi que la possibilité d'annoter des images ou des tableaux. Cette offre doit intégrer en grande partie les propositions qui sont faites dans ce chapitre. Pour cela, nous avons présenté l'architecture cible qui pourrait être mise en œuvre ainsi que les nouvelles fonctions à développer. Différentes solutions technologiques sont suggérées, avec une préférence pour Webkit comme moteur d'affichage. Si l'appel d'offre aboutit, un nouveau stage pourrait voir le jour et ce chapitre pourrait en être la première pierre…

ANNEXE I.
Exemple de DTD et fichier XML

Exemple de DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD Basic -->
<!-- du 29/06/05 -->
<!ELEMENT text (title | sub-title| paragraph)*>
<!ELEMENT title (#PCDATA)>
<!ELEMENT sub-title (#PCDATA)>
<!ELEMENT paragraph (#PCDATA | important | comment | reference)*>
<!ATTLIST paragraph color (white|yellow|blue)            "white">
<!ELEMENT important (#PCDATA)>
<!ELEMENT comment (#PCDATA)>
<!ELEMENT reference (#PCDATA)>
```

Exemple de fichier XML utilisant cette DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE text SYSTEM "Basic.dtd">
<?xml-stylesheet type="text/css" href="Basic.css"?>
<!--StyleSheet "Basic.css"  -->
<!-- Document settings "CursorPosition=281"-->
<!-- Document settings "Zoom=100"-->
<!-- Document settings "Spacing=10"-->
<!-- Document Information "Modified=7/8/2005 12:4:51:"-->
<!-- Document Information "Title=Welcome document"-->
<!-- Document Information "References=Leibniz-IMAG"-->
<!-- Document Information "Annotators=G. Bisson"-->
<!-- Document Information "Comments=V 1.0"-->
<text><title>Welcome to Cadixe</title>
<paragraph color="yellow">Cadixe is a specific XML editors well suited to the <important>annotation task</important>. In this process, the user wants to add progressively some XML tags to express the semantic of an existing text according to a given DTD (a grammar) which is specific to the application domain.</paragraph>
<sub-title>First step ...</sub-title>
<paragraph color="white">You will find some examples of annotated files in the Document folder. To load an example, select the command <important>"Load document"</important> in the <important>File</important> menu. There are two files :
 - <reference>Article.xml</reference> which is an example of annotated scientific paper.
 - <reference>Genomic.xml</reference> wontaining some annotated sentences in genomic</paragraph>.
<sub-title>Help menu</sub-title>
<paragraph color="white">The <important>Help</important> menu contains a complete documentation about Cadixe. It explains the structure of the interface, the annotation commands and the way to customize the editor.</paragraph></text>
```

ANNEXE II.
Exemple de feuille de styles – Basic.css

```
title {
	font-weight: bold;
	font-size: 20;
	cadixe-border-shape : Rectangle;
	border-color : #5e5fff;
	border-style : solid;
	border-width : 2;
	color: #7b0404;
	background-color: #ececec;
	display : block;
}
sub-title {
	background-color: #ffd2ad;
	font-size: 14;
	cadixe-border-shape : Rectangle;
	border-style : dotted;
	border-width : 1;
	display : block;
}
paragraph {
	background-color : #ffffff;
	cadixe-index-attribute : color;
}
paragraph_white {
	background-color : #ffffff;
	display : block;
}
paragraph_yellow {
	background-color : #fdffc7;
	cadixe-border-shape : Rectangle;
	display : block;
}
paragraph_blue {
	background-color : #e0e5ff;
	display : block;
}
important {
	background-color: #ffecf2;
	font-weight: bold;
}
comment {
	background-color: #dfffe3;
	font-style: italic;
}
reference {
	cadixe-border-shape : Rectangle;
	border-style : solid;
	border-width : 1;
	border-style : dashed;
	background-color: #e9e9e9;}}
```

ANNEXE III.
Paramétrage des préférences

Default path : indique les chemins par défauts à utiliser pour récupérés les fichiers utiles au démarrage de l'application et au chargement des documents.

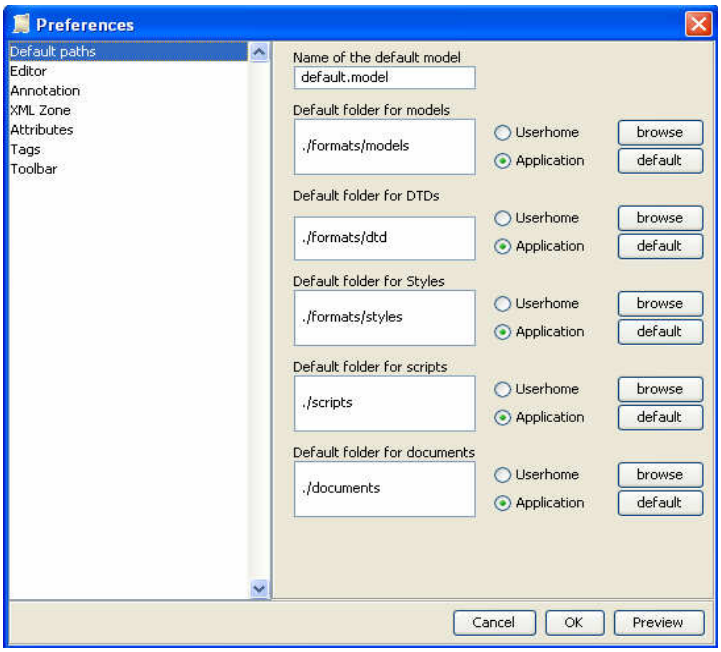


Figure 48 : Paramétrage des préférences « default Paths »

Editor : offre à l'utilisateur de gérer des paramètres concernant l'interface de l'application (positionnement des palettes d'outils, ouverture d'un document par défaut, etc).

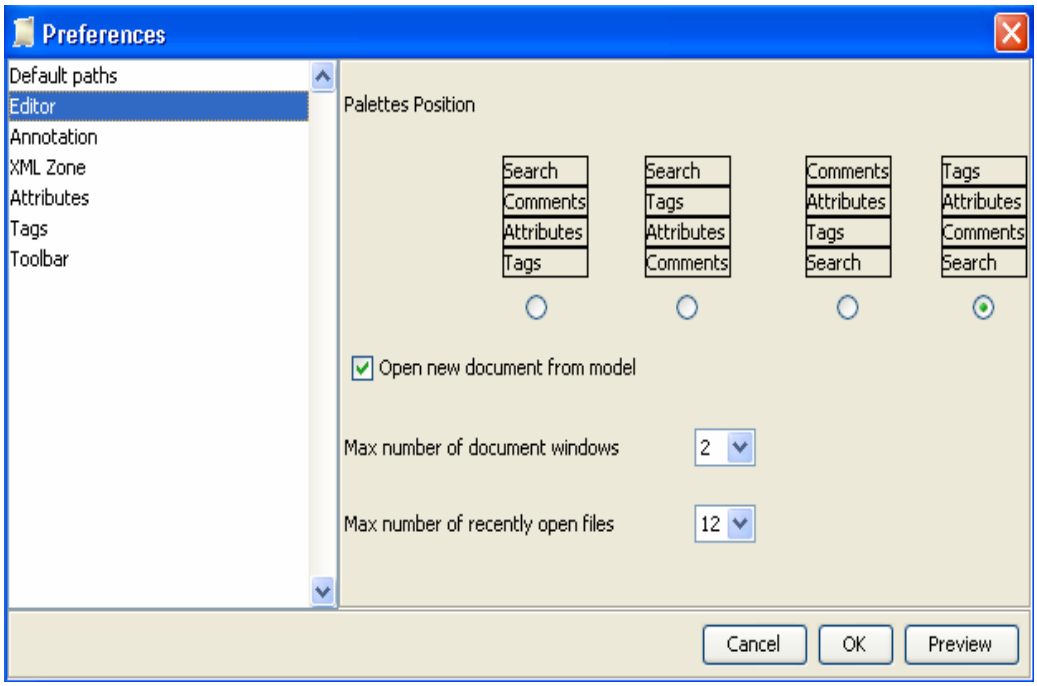


Figure 49 : Paramétrage des préférences « Editor »

Annotation : Indique si la sélection se fait au niveau du mot ou au niveau des caractères.

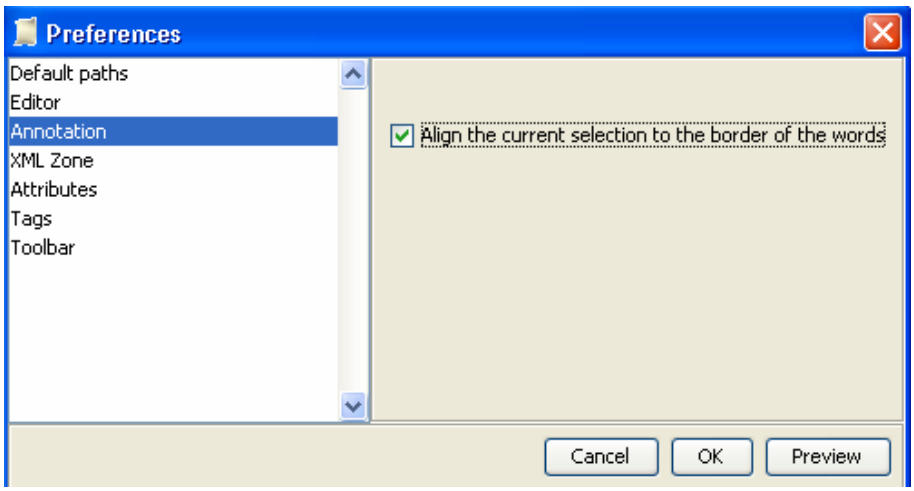


Figure 50 : Paramétrage des préférences « Annotation »

XML Zone : permet la gestion graphique de l'arbre XML.

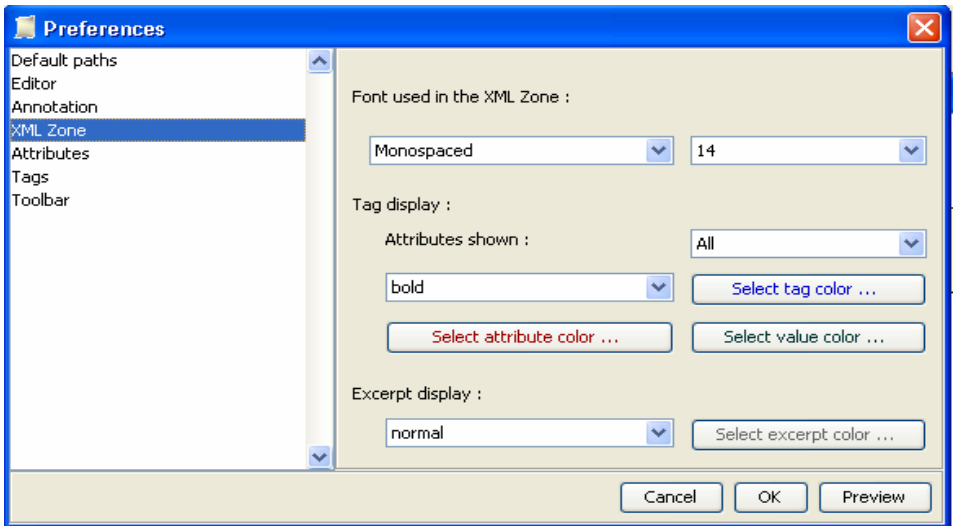


Figure 51 : Paramétrage des préférences « XML Zone ».

Attributes : permet la gestion des attributs à l'ajout d'une annotation dans le texte.

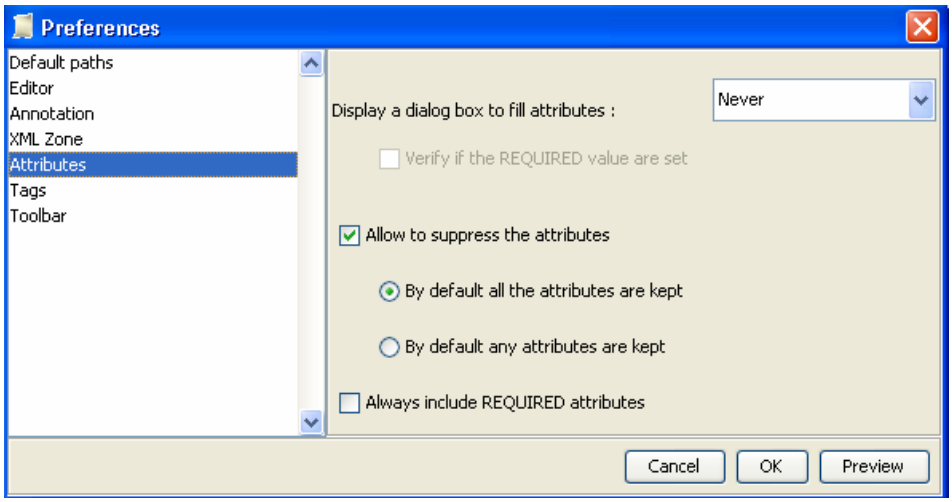


Figure 52 : Paramétrage des préférences « Attributes »

Tags : propose des options d'affichage pour la liste des balises.

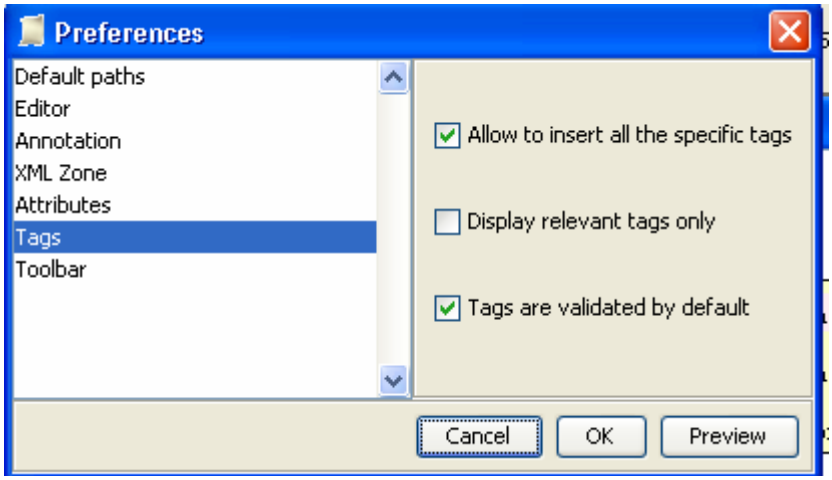


Figure 53 : Paramétrage des préférences « Tags »

Toolbar : permet la configuration des commandes à mettre dans la barre d'outils.

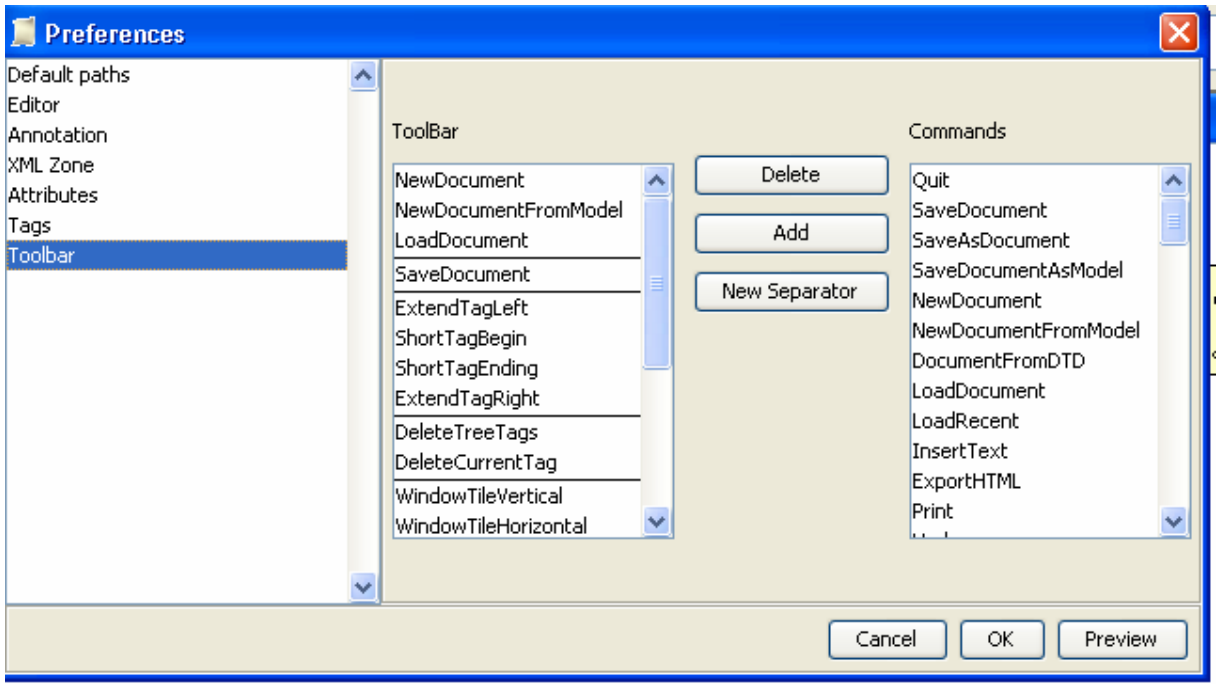


Figure 54 : Paramétrage des préférences « Toolbar »

ANNEXE IV.
Configuration et installation de l'application

Le logiciel étant écrit en Java, il est portable et exécutable sur différentes plates-formes : Linux, Windows et MacOs X. La machine virtuelle doit être récente : version ≥ 1.4.2 et l'on doit disposer d'une quantité de mémoire raisonnable (≥256).

L'application distribuée est nommée CADIXE.jar. Lors de l'exécution, le logiciel construit automatiquement la hiérarchie des fichiers dont il a besoin dans le dossier courant, comme dans le tableau ci-dessous :

CADIXE.jar	l'exécutable en Java
/resources	les ressources (icônes, fichiers des messages d'erreurs, …) utilisées par l'application
/cadixe-preferences - cadixe.properties	les configurations de l'éditeur fichier des préférences par défaut
• /formats • - /models • - /dtd - /styles	modèles de documents DTD et règles de complétion feuilles de styles au format CSS
• /documents	le dossier contenant les documents utilisateurs
• /scripts	le dossier des scripts (non actif dans cette version)

Les emplacements et les noms de dossiers notés en vert dans le tableau ci-dessus, sont modifiables via le paramétrage de l'application ou via la ligne de commande. Ainsi, il est possible de configurer précisément l'éditeur de manière à ce que, par exemple, les DTD et styles soient globaux à tous les utilisateurs et que les documents et les préférences restent locales. Pour récupérer les informations issues d'une version précédente, il suffit de les remettre dans les dossiers correspondants (dtd, styles, documents, …).

Le lancement de l'éditeur sous UNIX (ou Linux) s'effectue à partir d'un terminal (Xterm). Il faut se placer au niveau de l'application puis exécuter la ligne de commande suivante :

java -jar CADIXE.jar

Si l'application n'est pas au niveau du dossier courant, on peut aussi la lancer en indiquant simplement le chemin d'accès :

java -jar /MemoireCNAM/editeur/version 2/CADIXE.jar

Sous d'autres systèmes (OS X notamment, Windows, …), le lancement de l'application s'effectue simplement en double-cliquant sur l'exécutable *CADIXE.jar*. Après quelques instants, la fenêtre principale de l'éditeur apparaît, elle s'adapte en fonction du système hôte et prend en compte les paramètres de lancement qui ont été utilisés.

Lors du lancement de l'éditeur, on peut spécifier dans la ligne de commande les options suivantes permettant de modifier le comportement par défaut de l'éditeur :

- *userhome path* : par défaut l'éditeur stocke ses dossiers de travail à l'endroit ou se trouve l'application, cet argument permet d'indiquer un autre chemin d'accès.

- *prefs name* : indique le fichier de configuration (qui contient les préférences) qui est à charger dans le dossier « Cadixe-preferences » du dossier de travail courant. Par défaut le fichier qui est chargé est « cadixe.properties ».

- *load path* : charge le fichier qui est indiqué automatiquement lors du lancement de l'éditeur. Le fichier peut-être un fichier texte ou un fichier annoté (au format XML). Lorsqu'il s'agit d'un fichier texte, il est chargé en utilisant la DTD et les styles qui sont définis dans le modèle par défaut.

- *name string* : Ouvre un nouveau document (qui est construit à partir du modèle par défaut) avec comme nom celui qui est passé en paramètre.

- *model path* : Indique le modèle qui doit être utilisé, en remplacement du fichier modèle utilisé par défaut.

Bibliographie

[CHA09] Chaffer J., Swedberg J., 2009, jQuery - *Simplifiez et enrichissez vos développements Javascript*, Editions Eyrolles, ISBN 2-7440-2381-7

[BON00] Bonhomme P., 2000, Codage et normalisation des ressources textuelles. Ingénieries des langues, Hermes : 174 – 191.

[DAR08] Darie C., Bucica M., Chereches-Tosa F., Brinzarea B., 2007, *AJAX et PHP*, Editions Dunod, ISBN 2-100-50684-6

[DEF10] Defrance J., 2010, *Premières applications Web avec Ajax, jQuery et PHP*, Editions Eyrolles, ISBN 2-212-12672-7

[JAC00] Jacobson I., Booch G., Rumbaugh J., 2000, *Le processus unifié de développement logiciel*, Collection Technologies objet, Editions Eyrolles, ISBN 2-212-09142-7

[HUN01] Hunter D., 2001, *Initiation à XML*, Edition Eyrolles, ISBN 2-212-09248-2

[MIC00] Michard A., 2000, *XML: langage et applications*, Editions Eyrolles, ISBN 2-212-09206-7

[MUL00] Muller P.-A., Gaertner N., 2000, *Modélisation objet avec UML*, Eyrolles, ISBN 2-212-09122-2

[RAY02] Ray E., *Introduction à XML*, 2002, Editions O'Reilly.

[ROQ02] Roques P., Vallée F., 2002, *UML en action, De l'analyse des besoins à la conception en Java*, Collection Technologies objet, Editions Eyrolles, ISBN : 2-212-11213-0

[VAU07] Vautherin E., 2007, *Mac OS X Programmation*, Version 10.5, Edition Dunod.

[CADE] http://caderige.imag.fr/ : Site du projet Caderige

[CLOG] http://capirossi.org/info/prj_mgt/cycle_1.htm : Présentation des différentes approches utilisées dans le développement logiciel.

[CLAR] http://www.bultreebank.org/clark/index.html : La page d'accueil de CLaRK System, pour télécharger le programme, consulter le manuel en ligne, avoir une démo ou des références bibliographiques

[JAVA] http://java.sun.com : Site officiel de Sun.

[JDOU] http://jmdoudoux.developpez.com/cours/developpons/eclipse/ : Développons en Java avec Eclipse de Jean-Michel Doudoux.

[GLOZ] http://www.glozz.org : La plate-forme Glozz, environnement d'annotation et exploration de corpus.

[OMG] http://www.uml.org/ : UML sur le site de l'Object Management Group.

[PALI] http://clg.wlv.ac.uk/projects/PALinkA : Site dédié à l'application Palinka par C.Orasan.

[PARS] http://www.commentcamarche.net/contents/xml/xmldomsax.php3 : Introduction au analyseur syntaxique SAX et DOM.

[WEBA] http://share.esi.dz/index.php : Ingénieur 2009/ Système d'information/ Outil d'annotation web dédié aux enseignants.

[WORD] http://wordfreak.sourceforge.net : A cette adresse, on trouvera le code source, des copies d'écrans ainsi qu'une version de l'application téléchargeable.

Glossaire

Annotation : une annotation est un commentaire, une note, une explication ou tout autre remarque externe qui peut être attachée à un document.

Apprentissage automatique (Wikipedia) : l'apprentissage automatique fait référence au développement, à l'analyse et à l'implémentation de méthodes qui permettent à une machine (au sens large) d'évoluer grâce à un processus d'apprentissage, et ainsi de remplir des tâches qu'il est difficile ou impossible de remplir par des moyens algorithmiques plus classiques.

Arbre binaire : il s'agit d'une structure de données qui peut se représenter sous la forme d'une hiérarchie dont chaque élément est appelé nœud, le nœud initial étant appelé *racine*

Balise : elle sert à délimiter des ensembles de données contenues dans un document afin de permettre la structuration de ce document à l'aide d'un langage spécialisé (ici le langage XML).

DTD : il s'agit d'un document SGML qui accompagne un modèle de document XML. La DTD donne les règles de cette structuration.

Entité nommée : les informaticiens, qui travaillent dans le domaine de l'extraction d'information, ont défini la notion d'**entités nommées** pour regrouper tous les éléments du langage définis par référence : les noms propres au sens classique, les noms propres dans un sens élargi mais aussi les expressions de temps et de quantité.

Extraction d'information (EI) : consiste à remplir automatiquement des formulaires ou une banque de données à partir de textes écrits en langue naturelle.

Feuille de styles : elles sont connues sous le nom de Cascading Style Sheets (CSS) Il s'agit d'un document numérique qui va pouvoir spécifier toutes les caractéristiques de mises en formes du document lié à la balise à laquelle elle s'applique.

Parseur : il s'agit d'un analyseur syntaxique destiné à récupérer les informations contenues dans les balises d'un document XML. Cet outil distinguera les informations en fonction de leur contenu et de leur situation dans le document : balise de début, balise de fin, etc. Plus généralement, un parseur peut être assimilé à un outil d'analyse syntaxique. C'est d'ailleurs le sens premier du terme anglais parser.

Préférences : les préférences permettent de gérer de manière globale le comportement de l'éditeur.

Recherche d'information (RI) : vise à retrouver dans une base de documents un ensemble de documents pertinents au regard d'une question.

XML : est un langage de description des documents qui utilise des balises, qui permet l'utilisation de balises personnalisées et l'échange des données

XML SCHEMA : procédé qui vise à remplacer les DTD qui décrit les documents XML. XML Schema est lui même un document XML, alors que les DTD sont des documents SGML. Il fourni des types de données plus aboutis et va procéder à la validation du code XML grâce à l'utilisation des espaces de nommage.