UNIVERSITÉ MONTPELLIER 2
SCIENCES ET TECHNIQUES

Rapport de projet
Licence 3

Editeur de sites web

Réalisé par :

Pierre Burc, Olivier Duplouy,
Hamza Erraji, Issame Amal,
Mickaël Berger, Joachim Divet,
Zaydane Sadiki et Abdelhamid Belarbi

Sous la direction de :

Michel Meynard

Année universitaire 2011-2012

Remerciements

Nous tenons à remercier tout particulièrement M. Michel Meynard, notre tuteur de projet qui nous a guidés et épaulés tout au long de ces quelques mois de travail, égrénant ça et là différents conseils utiles à souhait.

Bien entendu nous n'oublions pas de remercier chaleureusement toute l'équipe pédagogique de l'UM2 qui nous a apporté son soutien.

Et il va sans dire que tous les membres du groupe se remercient les uns les autres.

Table des matières

Introduction 6

I Analyse 7

1 Cahier des charges 8

2 Étude de projets existants 9
- 2.1 Espresso . . . 9
- 2.2 Aptana . . . 9
- 2.3 Dreamweaver . . . 10
- 2.4 Bilan comparatif . . . 10

3 Choix des outils 11

4 Organisation 12
- 4.1 Diagramme de Gantt . . . 12

II Conception 13

5 Décomposition en sous systèmes 14
- 5.1 Gestion des projets . . . 14
- 5.2 Gestion des fichiers . . . 16
- 5.3 Gestion du code . . . 17

6 Diagrammes des classes 18
- 6.1 Coloration . . . 18
- 6.2 Système de fichiers . . . 19
- 6.3 Indentation . . . 20
- 6.4 Autocomplétion . . . 20
- 6.5 Classes graphiques . . . 21

7 Organisation du code : le modèle MVC 23

III L'oeuvre 24

8 Travail de groupe 25
- 8.1 Répartition des tâches . . . 25
- 8.2 Conventions techniques . . . 25
- 8.3 Les réunions hebdomadaires . . . 26

9 Implémentation **27**
9.1 Arborescence du projet 27
9.2 Système 27
9.3 Fonctionnalités 28
9.3.1 Coloration 28
9.3.2 Autocomplétion 28
9.4 Interface 29

10 Résultat **30**

11 Discussion **32**
11.1 Cahier des charges 32
11.2 Considération graphique 32
11.3 Aller plus loin 32

Conclusion **33**

Annexe technique **34**
11.4 À propos de Qt 34

Table des figures

2.1 Espresso 9
2.2 Aptana 9
2.3 Dreamweaver 10

4.1 Diagramme de Gantt 12

5.1 Décomposition en sous systèmes 14
5.2 Sous système de gestion des projets 15
5.3 Use-Case modification de projets 15
5.4 Sous système de gestion des fichiers 16
5.5 Sous système de gestion du code 17

6.1 Classe de données pour le langage Html 18
6.2 Classes de coloration 18
6.3 Classes du système de fichiers 19
6.4 Classe CentralEditor 20
6.5 Classes de l'interface graphique 21

7.1 Les trois catégories du MVC 23

9.1 Arborescence de projet 27
9.2 L'autocomplétion en pratique 29

10.1 L'écran à l'ouverture de l'application 30

Liste des tableaux

2.1 Fonctionnalités spéciales dans les éditeurs étudiés 10

10.1 Fonctionnalités spéciales dans les éditeurs étudiés 31

Introduction

C'est par une froide journée d'hiver que nous nous réunîmes pour la première fois à l'université de Montpellier. Huit, tous étudiants préposés au projet numéro vingt-trois nous attendons à notre table l'arrivée de notre tuteur M. Meynard. Ce dernier se présente, nous salue et prononce ce discours mémorable qui restera gravé dans nos mémoires.

> « Dans le cadre de développement de sites Web, on souhaiterait utiliser un éditeur multi-fichiers permettant de réaliser différentes actions sur des fichiers relatifs à un site : édition de fichiers Html, Php, JavaScript et Css.
>
> Il serait également appréciable que cet éditeur proposât un mode de visualisation du site dans un navigateur et un mode arborescence.
>
> Et tant que nous y sommes, mettons de l'autocomplétion, de la coloration syntaxique, un accès aux manuels des langages cités précédemment et une validation Html. ».

Après ce laïus prononcé d'une traite M. Meynard disparut soudain, nous laissant là, le regard vide.

Néanmoins, nous nous remîmes assez vite de notre ahurissement et un sage parmi nous s'écria soudain :

> « Nous commencerons par étudier quelques éditeurs existants sur le marché pour nous faire une idée. Ensuite nous concevrons le programme avec le langage UML en nous organisant pour la réalisation. Enfin, nous implémenterons l'application insolents et sûrs de nous. Qu'en dites-vous ? ».

Nous n'en dîmes que du bien. En effet, l'idée loin d'être incroyablement novatrice avait l'avantage d'être logique et cohérente.

C'est ainsi que démarra le projet *Éditeur web* décrit dans ce document, entrons donc sans plus attendre dans le vif du sujet.

Première partie

Analyse

Chapitre 1

Cahier des charges

Voici une liste exhaustive des fonctionnalités prescrites par M. Meynard :
- Édition des fichiers JavaScript, Php, Css et Html ;
- Un système multi-onglets permettant de naviguer rapidement entre plusieurs fichiers d'un même projet ;
- Possibilité de visualiser le site sous forme d'arborescence ;
- Mode autocomplétion et coloration syntaxique ;
- Mode auto-indentation ;
- Accès en un clic au manuel Php/Html/Css/JavaScript ;
- Validation Html ;
- Visualisation du site dans un navigateur ;
- Squelette de site préexistant.

La liste d'exigences ci dessus se résume en une besoins graphiques, il nous fallait donc « deviner » ce qui se déroule en interne dans le programme. Malgré les différents avis données par M. Meynard, il était difficile pour la majorité des membres de comprendre comment fonctionnait un tel programme.

Après plusieurs colloques, discussions, palabres et réunions, il fut décidé que le meilleur moyen de se rendre compte de ce que représentait un tel cahier des charges en terme de produit fini était d'étudier les différents éditeurs existants offrants ce genre de fonctionnalités.

Chapitre 2

Étude de projets existants

2.1 Espresso

Figure 2.1 – Espresso

Le premier programme d'édition de site web que nous avons étudié propose les mêmes fonctionnalités que le logiciel que nous nous proposons de développer, à savoir coloration syntaxique, indentation automatique, arborescence des codes, etc. Il sera à priori une bonne source d'inspiration pour nous d'autant plus que l'interface est très soignée et agréable d'utilisation.

2.2 Aptana

Figure 2.2 – Aptana

Cet éditeur propose un ensemble de fonctionnalités nombreuses et variées. Beaucoup plus fourni que Espresso, il propose des fonctionnalités complexes dans divers langages : déboguage, déploiement automatique, gestionnaire de version inclus, terminal intégré et moult autres outils. Pour nous c'est un bon modèle à suivre en évitant tout de même de tomber dans le piège du sur-nombre d'options qui importuneraient l'utilisateur.

2.3 Dreamweaver

Figure 2.3 – Dreamweaver

Dreamweaver est une référence en la matière. Il réunit les atouts des deux éditeurs sus-cités en joignant l'utile à l'agréable. Il possède en outre un mode dit WYSIWYG permettant de dessiner l'interface d'un site web.

2.4 Bilan comparatif

Le tableau 2.1, donne la distributions de quelques fonctionnalités parmi les logiciels cités précédemment. Il est inutile de mettre dans ce tableau des fonctionnalités telles que la coloration ou l'édition de fichier, car il est évident que des lesdits logiciels possèdent ce genre de spécificités.

Table 2.1 – Fonctionnalités spéciales dans les éditeurs étudiés

Éditeur	onglets	autocomplétion	auto-indentation[1]	validation Html	documentations
Espresso	oui	oui	non	non	non
Aptana	oui	oui	non	oui	oui
Dreamweaver	oui	oui	oui	oui	non

Ainsi, il ressort du tableau 2.1 qu'aucun des outils étudiés ne possède toutes les spécificités de notre programme. Il faudra donc, si nous voulons de l'inspiration, naviguer d'un éditeur à l'autre selon que nous voulons implémenter telle ou telle fonctionnalité.

Maintenant que l'objectif est discernable et que le groupe a une idée générale du logiciel à produire, il est temps de penser à choisir les outils à utiliser.

1. À noter que l'auto-indentation dans le tableau 2.1 concerne la capacité de formater tout un fichier et non pas d'ajouter une tabulation après saut de ligne.

Chapitre 3

Choix des outils

Il est des choix qui influent sur l'ensemble d'un projet et le choix des outils est de ceux là. Nous nous devions de faire un choix judicieux compte tenu de différents critères, à savoir la taille de l'équipe (huit personnes), l'ampleur du projet, le temps imparti et autres menus détails.

La modélisation formelle des spécifications du programme nécessite un langage adapté à ce genre de besoin. C'est donc presque sans discussion que nous prîmes l'évidente décision d'utiliser le langage UML pour ce faire. Les diagrammes seront dessinés grâce à une application web nommée yUML (cf. sitographie, p.39).

Concernant le langage de programmation principal, nous choisîmes C++. Ce choix est motivé par deux raisons, d'une part nous apprenons actuellement ce langage en cours, d'autre part, il s'agit d'un langage stable, documenté et qu'il fait bon d'avoir dans sa besace.

Après cela M. Meynard nous proposa le framework Qt. Un peu austère de prime abord, presque effrayant, il s'avéra finalement très sympathique grâce notamment à une syntaxe lisible et à une documentation bien feuillue.

Pour générer une documentation de notre code source, nous fîmes le choix de Doxygen, un outils spécialisé dans ce genre de tâches.

Pour travailler en équipe sur un projet de ce genre il est utile voire indispensable de disposer d'un outil de synchronisation et de partage. Notre choix s'est porté sur le gestionnaire de version Git. Ceci sans raison particulière car ses semblables offrent des fonctionnalités similaires.

Bien que les huit membres de notre groupe furent géographiquement proches à vol d'oiseau, aucun d'entre nous ne possédait l'étonnante capacité de se déplacer dans les airs. Par conséquent, hormis les outils sus-cités il fallut quelques outils auxiliaires de communication.
Nous utilisâmes donc des outils web tels que :

MicroMobs.com : Pour les discussions professionnelles ;

Facebook.com : Pour les annonces importantes, les messages, les dates de réunions, etc.

Citons aussi TeamGantt.com, une application qui permet de dessiner un (très joli) diagramme de Gantt en ligne.

Chapitre 4

Organisation

4.1 Diagramme de Gantt

Le projet se déroulant sur quatre mois, il a fallu faire un planning qui s'étale sur ledit laps de temps. Voici toutes les étapes du projet inscrites sur un diagramme de Gantt.

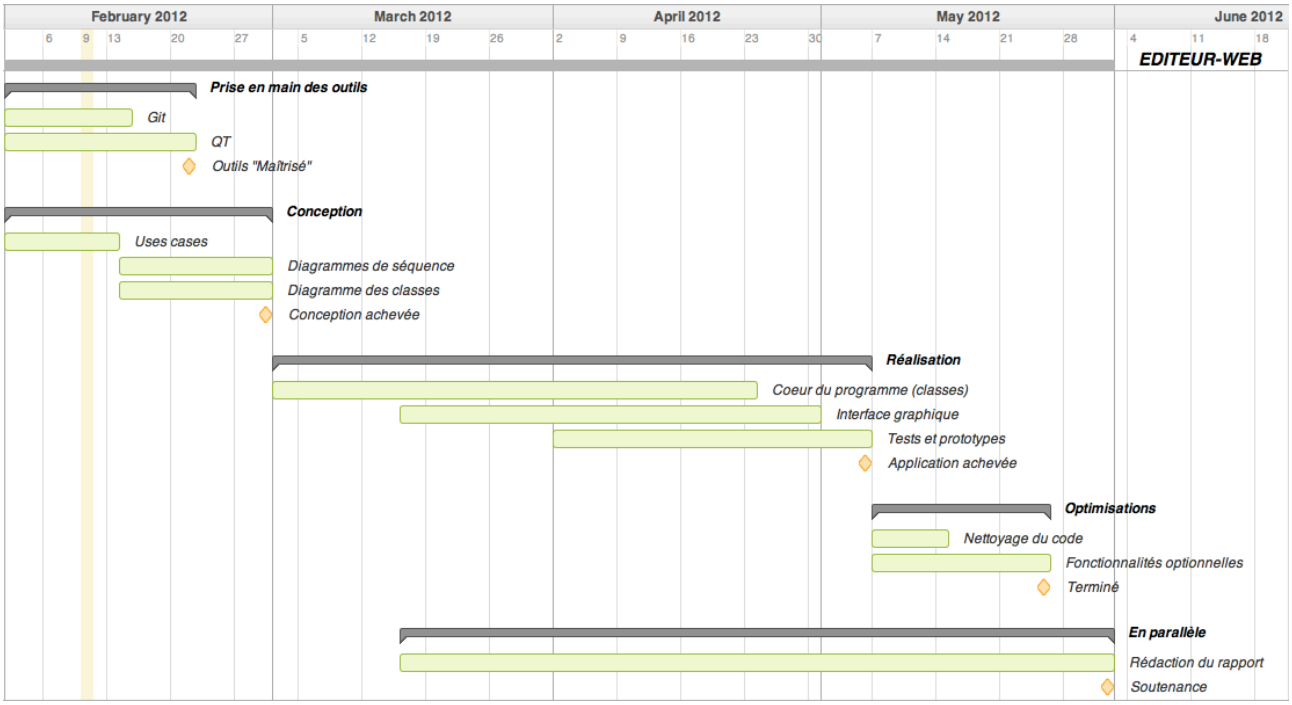

Figure 4.1 – Diagramme de Gantt

Deuxième partie

Conception

Chapitre 5

Décomposition en sous systèmes

Le programme se découpe grossièrement en trois sous-systèmes complémentaires comme sur le diagramme 5.1.

Figure 5.1 – Décomposition en sous systèmes

Le système de gestion des fichiers est un système très important de l'application et c'est pourquoi il est requis par le système de gestion des projets (flèche *include*).
Le système de gestion du code est vue comme une extension aux fichiers. En effet, il serait possible d'éditer un site web sans gérer la coloration, l'indentation et toutes les fonctionnalités qui rendent un éditeur de code si agréable à utiliser.

5.1 Gestion des projets

Le sous-système de gestion des projets est un élément important de l'application, en ce sens qu'il permet à l'utilisateur de bien s'organiser de manière simple et efficace.

À la manière de beaucoup de logiciels connus, il sera nécessaire à l'utilisateur de sélectionner un espace de travail, regroupant un ensemble de projets. Cette petite obligation n'est que très peu contraignante et favorise l'organisation, sans pour autant l'imposer réellement à l'utilisateur puisque l'emplacement de cet espace de travail est donné par l'utilisateur.

Un projet est concrètement représenté par un dossier sur la machine de l'utilisateur. Partant de cette idée, notre système de gestion de projet sera implémenté en utilisant les normes et références communément admises en termes de création, modification et destruction de dossiers.

L'utilisateur doit pouvoir effectuer les actions classiques relatives à un projet, à savoir création, modification, suppression, ouverture et fermeture, il peut également comme stipulé dans le cahier des charges, créer un nouveau projet basé sur un squelette de site pré-existant.
Ce dernier cas de figure, non obligatoire, permet à un utilisateur d'avoir un aperçu de ce que peut être un projet de site internet en développement sous notre application.
L'unique caractéristique d'un projet est que l'on pourra trouver à la racine un fichier spécial résumant les paramètres utilisateur, les caractéristiques du projet, et autres informations diverses.

En résumé, concernant les projets, l'application fournit les fonctionnalités décrites dans le diagramme 5.2.

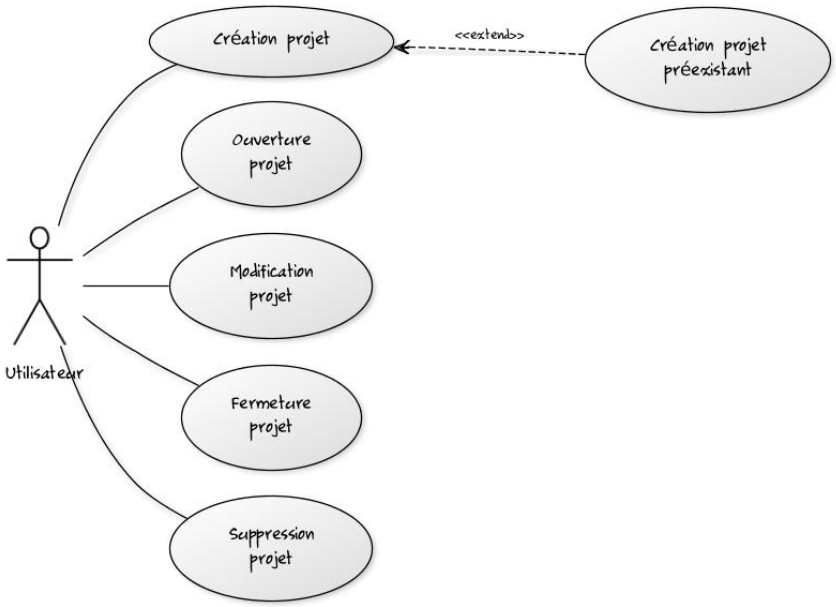

Figure 5.2 – Sous système de gestion des projets

À noter que cette méthode d'utilisation de l'application nous a principalement été inspirée par des logiciels que nous utilisons tous en cours et chez nous, comme Eclipse par exemple.
À ces actions d'ordre « général » s'ajoutent quelques actions propres à la modélisation physique des projets dans notre application, qui sont exposées dans le diagramme 5.3

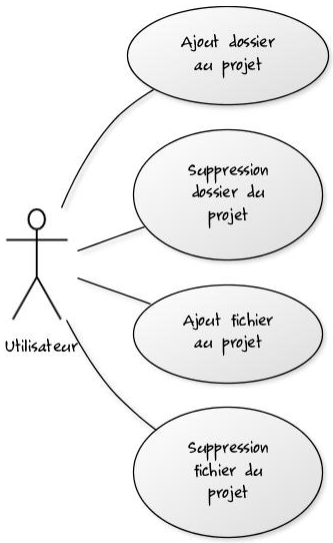

Figure 5.3 – Use-Case modification de projets

Ainsi comme exposé dans ce diagramme, l'utilisateur pourra créer autant de fichiers et de dossiers qu'il le désire dans sont projet, lui permettant ainsi de pouvoir organiser son travail comme il le souhaite, plutôt que devoir se plier à un mode d'organisation imposé par l'application.

Toutefois, ces actions d'apparences basiques sont signes d'un travail important à venir puisqu'elles constituent des libertés supplémentaires accordées à l'utilisateur.

5.2 Gestion des fichiers

Le diagramme 5.4 détaille le contenu du système de gestion des fichiers. Notez l'incroyable similitude entre ce dernier et le système de gestion des projets décrits précédemment.
En effet, comme pour les projets, l'application doit permettre d'effectuer des actions basiques en terme de manipulation de fichiers : création, ouverture, etc.

L'application jouera cependant un rôle particulier au niveau de l'édition d'un fichier puisque c'est à ce niveau qu'interviendront les fonctions de coloration, d'indentation et d'auto-complétion du code, représentées par le système de « gestion du code » dans le diagramme 5.5.

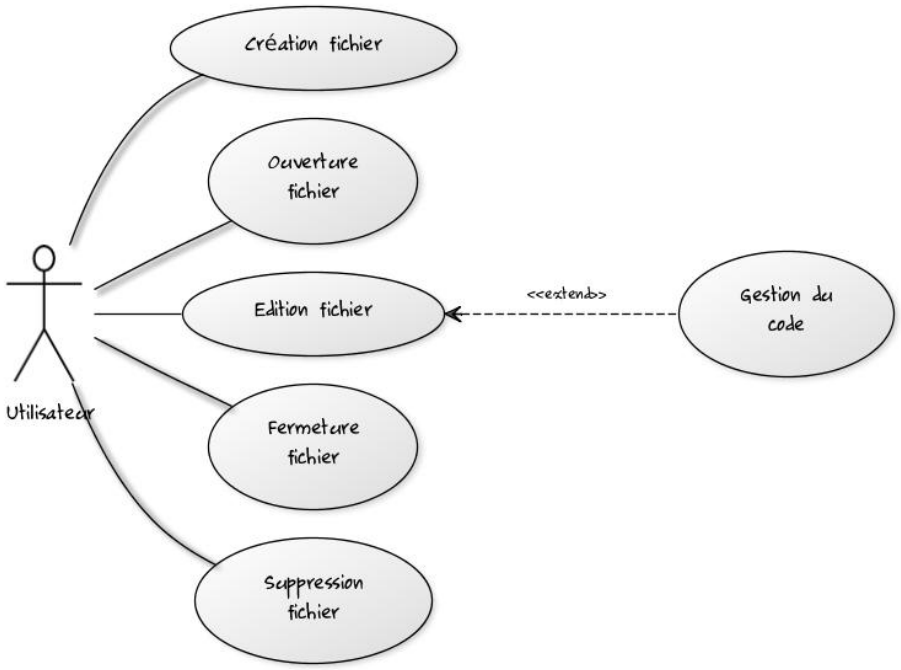

Figure 5.4 – Sous système de gestion des fichiers

5.3 Gestion du code

Derrière ce nom quelque peu singulier se cache un concept fort simple, la gestion du code consiste à indenter, colorer, visualiser et sublimer le code source présent dans un fichier tel que demandé dans le cahier des charges. Le diagramme 5.5 expose les différentes propriétés que doit posséder le système de gestion du code.
L'action étiquetée *Modification du code* correspond à l'édition graphique du texte à l'écran. L'interface graphique n'est pas un sous-système en soi du fait notamment qu'elle est incluse dans sous-système de gestion de code.

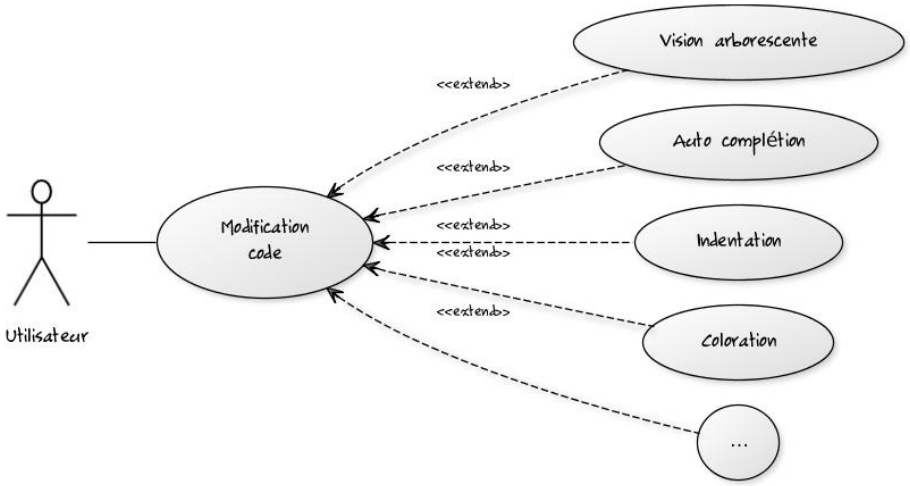

Figure 5.5 – Sous système de gestion du code

Chapitre 6

Diagrammes des classes

6.1 Coloration

La conception de la partie « coloration syntaxique » de notre application a requis l'introduction de plusieurs classes différentes. Tout d'abord des classes dites de données, suffixées du mot anglais *Data*, qui contiennent les différents mots clés relatifs à chaque langage sous forme d'expressions régulières. Dans le diagramme 6.1, nous donnons en exemple une seule classe *HtmlData*, car les autres sont similaires, aux particularités du langage près.

Figure 6.1 – Classe de données pour le langage Html

De la même manière, on trouve les classes *CssData, PhpData et JavaScriptData*.

Une autre part de la coloration est l'utilisation concrète des classes décrites précédemment. Sur le diagramme 6.2 sont représentées toutes les classes qui permettent la coloration à l'écran. Elles héritent d'une superclasse *Highlighter* qui factorise la méthode *highlightBlock()*, celle-ci, comme son nom l'indique, colore le texte bloc par bloc en fonction des règles données dans les classes *Data*.

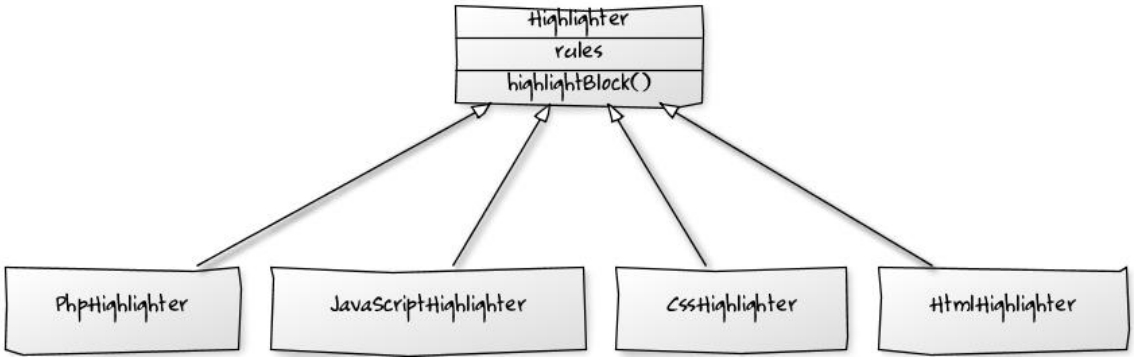

Figure 6.2 – Classes de coloration

Le fonctionnement du module de coloration syntaxique est simple. Une classe est associée à un fichier (voir décomposition de l'application, diagramme 5.1) selon le code utilisé dans ce fichier. Le texte est

analysé grâce aux expressions régulières des classes de données et les mots reconnus sont formatés dans l'observance des règles édictées dans les classes du diagramme 6.2.
Une présentation plus détaillée se trouve dans l'annexe technique à la page 34.

6.2 Système de fichiers

La réflexion autour de l'organisation des fichiers gérés par l'application fût longue, bien que quelque peu avancée par les diagrammes de cas d'utilisation que nous avions créés.
En effet, ces derniers nous donnaient un premier aperçu de l'approche à adopter pour traiter le sujet mais nous laissaient quand même beaucoup de choix quant au schéma de fichiers à adopter. Notre application suivant bien sûr un développement orienté objet, nous avons desuite eu l'idée de traiter dossiers, projets et espaces de travails comme des instances de classes, pour finalement donner naissance (après quelques essais) au diagramme 6.3.

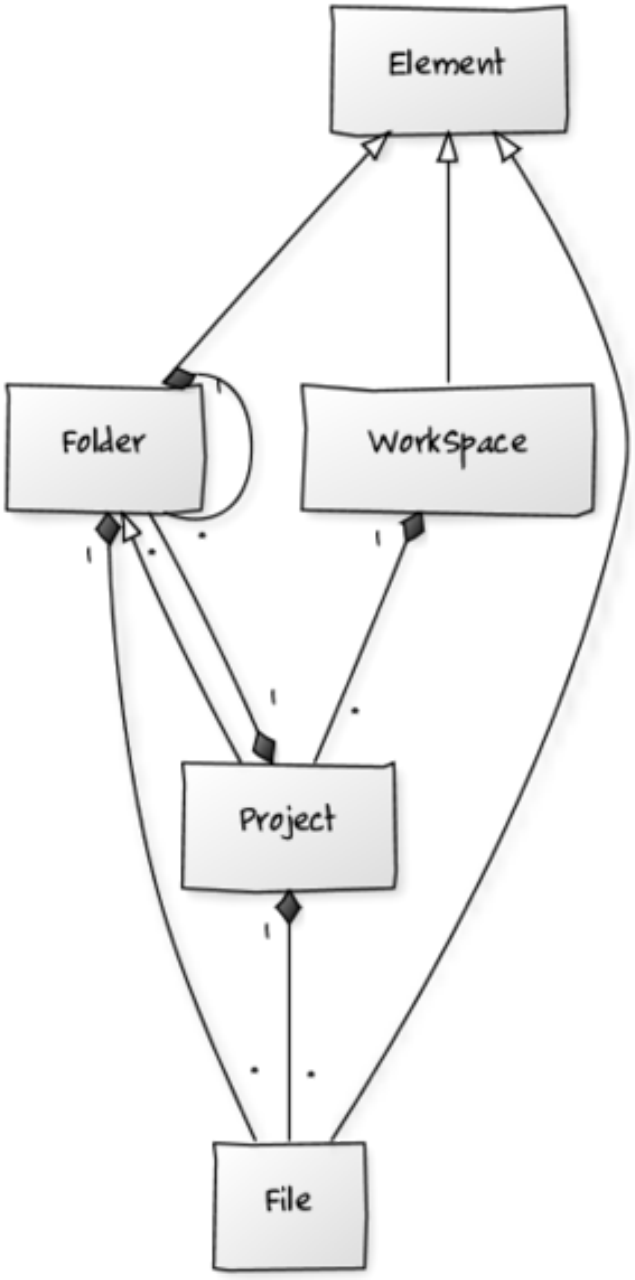

Figure 6.3 – Classes du système de fichiers

Avant d'éditer ce diagramme, nous nous étions mis d'accord, pour des raisons évidentes, sur une factorisation importante de notre code. Ainsi, la classe Élément regroupe tous les attributs, constantes et méthodes communs à tous les types d'items traités par l'application.
À la manière d'un dossier, un Projet pourra être composé de fichiers et de dossiers, mais aura des méthodes qui lui sont propres, telles qu'une suppression de son contenu. Le WorkSpace lui ne sera pas traîté comme un dossier mais plutôt comme un élément particulier ne contenant que des projets.
Pour le reste, les classes Folder et File parlent d'elles-mêmes.

6.3 Indentation

" " " " " " " "

6.4 Autocomplétion

L'autocomplétion, est une fonctionnalité informatique permettant à l'utilisateur de limiter la quantité d'informations qu'il saisit avec son clavier, en se voyant proposer un complément qui pourrait convenir à la chaîne de caractères qu'il a commencé à taper.
Dans le domaine de la programmation où il est souvent besoin de répeter les mêmes instructions et d'écrire les mêmes mots clés, cette fonctionnalité est presque vitale.

Il n'existe pas à proprement parler de classe dédiée à l'autocomplétion dans notre application. En effet, nous intégrâmes directement cette fonctionnalité dans la classe *CentralEditor* par soucis d'optimisation et de simplicité.

Différents facteurs entrent en ligne de compte lorsque l'on souhaite parler d'autocomplétion :
- l'emplacement du curseur de rédaction ;
- le contexte du fichier, c'est-à-dire le(s) langage(s) utilisé(s) dans ce fichier ;
- le côté graphique, la visualisation ;
- le choix de l'utilisateur d'activer ou de désactiver tel ou tel langage.

Le diagramme 6.4 représente la classe *CentralEditor* limitée aux attributs et méthodes concernant l'autocomplétion :

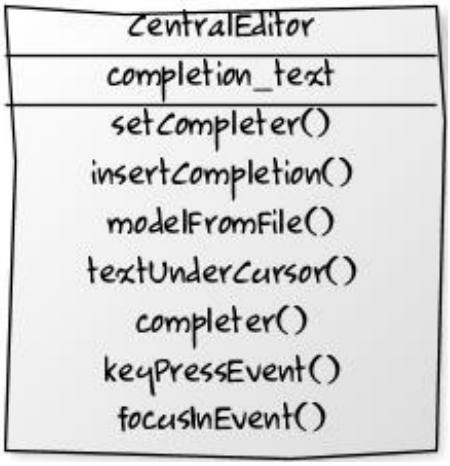

Figure 6.4 – Classe CentralEditor

Nous décrirons ici le comportement général des ces méthodes sans entrer dans le détail de chacune d'entre elle, ce qui sera fait au chapitre 9.

L'utilisateur commence à taper un mot *m*, l'application ouvre un dictionnaire de donnée correspondant au langage actuel et charge une liste des mots préfixés par *m*. Une fois cette liste construite, elle est affichée à l'utilisateur qui choisit de selectionner un mot proposé ou alors de continuer à taper son texte.

Quant aux dictionnaires sus-cités, ils sont au nombre de quatre et contiennent les mots clés de chacun des langages traités (à savoir Html, Css, JavaScript, Php).

6.5 Classes graphiques

La partie graphique fût pour nous une nouveauté puisqu'aucun d'entre nous n'avait jusqu'alors participé à un projet d'une si grande ampleur avec une interface graphique. Autant dire que les débats furent houleux, chacun ayant sa propre vision des choses.
Et une fois de plus, M. Meynard trancha. L'interface sera organisée comme sur le diagramme 6.5.

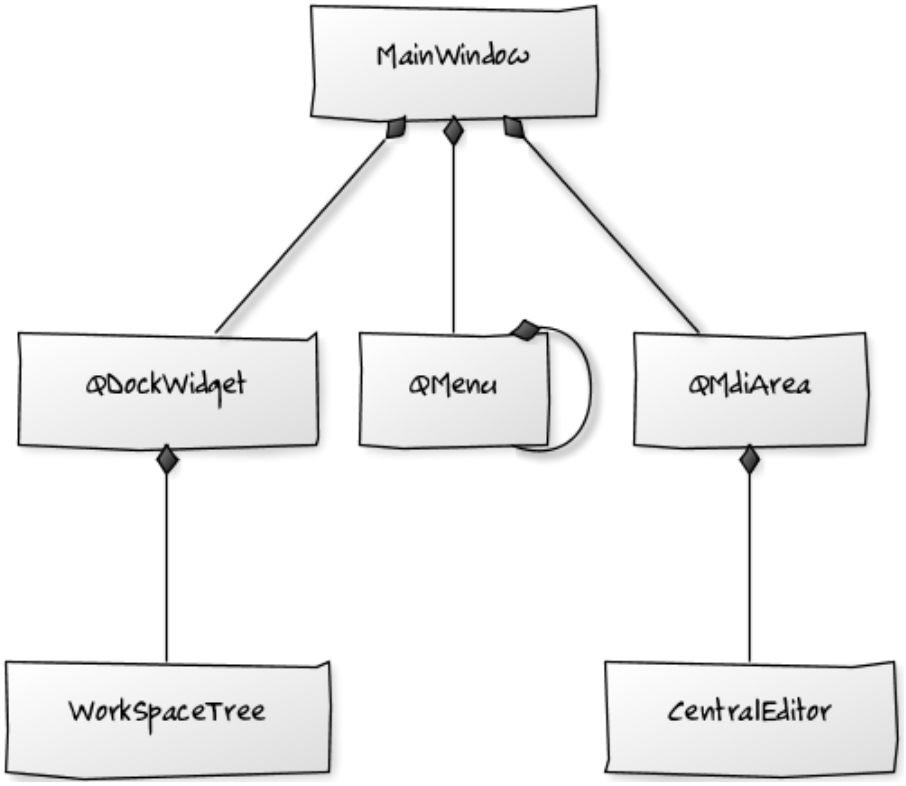

Figure 6.5 – Classes de l'interface graphique

Présentation des classes utilisées pour l'interface.

CentralEditor : Permet d'obtenir un onglet au centre de l'interface. Cet onglet possède les fonctionnalités d'auto- Complétion et de coloration syntaxique en fonction du langage choisi. Elle permet aussi la création d'un onglet, le Lancement de la complétion, la création d'un nouveau Fichier avec un nom par défaut, charger un fichier ouvert et de sauvegarder le fichier courant. WorkSpaceTree : Permet la sélection d'un espace de travail et de le parcourir, ensuite afficher dans la partie gauche Tous les fichiers et les dossiers qu'il contient. Aussi on y trouve la possibilité de renommer, Supprimer, copier et de coller un fichier ainsi qu'un dossier et d'ajouter un nouveau fichier ou projet à partir du TreeView. EditorTab : C'est la partie

droite de l'interface qui contient TreeWidget qui permet d'afficher l'arborescence du fichier ouvert, ainsi que WebView qui permet à son tour de visualiser la page HTML au fur et à mesure que la rédaction du code. MainWindow : Permet d'obtenir ou définir la fenêtre principale de l'application avec son menu, sa zone d'édition des fichiers PHP, CSS, HTML, et JavaScript, son workspace et son WebView.

Chapitre 7

Organisation du code : le modèle MVC

Avant de réellement commencer la programmation (partie suivante), il nous a fallu parler de la façon dont nous allions organiser notre code. Qt est conçu pour gérer des projets organisés selon le modèle MVC, il était donc logique de se pencher sur la question. D'autant plus que ce patron de conception est actuellement très prisé dans le monde de la programmation.

Le MVC est donc un modèle de programmation qui repose sur la séparation du code produit en trois grandes catégories représentées sur le schéma 7.1.

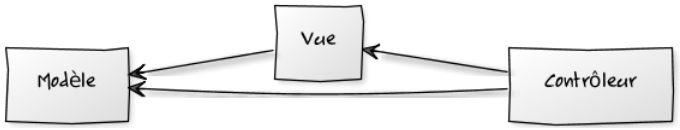

Figure 7.1 – Les trois catégories du MVC

Le modèle : représente tous les aspects du comportement de l'application, c'est-à-dire le traitement des données et toutes les interactions avec une éventuelle base de données, c'est lui-même qui décrit toutes les données et qui définit comment l'application va accéder et intéragir avec ces dernières.

La vue : l'interface sur laquelle va agir l'utilisateur, c'est elle qui se charge de présenter les données renvoyées par le modèle, de les afficher, et éventuellement de donner la possibilité à l'utilisateur d'interagir sur ces dernières, mais la vue n'effectue aucun traitement, son travail ne consiste qu'en la présentation des données, en aucun cas elle ne doit effectuer de traitement direct sur des données. Dans certains environnements, elle est écrite grâce à des langages de présentation uniquement (comme Html et Css), ce qui permet de la limiter à sa fonction première.

Le contrôleur : Il est l'agent de réponse à un utilisateur. Généralement, lui non plus n'effectue pas de traitement, mais se charge d'analyser chaque requête de l'utilisateur, chacune de ses interactions avec la vue, et il fera appel au modèle et à la vue qui conviennent à chacune de ces requêtes. Le contrôleur est un peu l'interprète de l'utilisateur, c'est lui qui va comprendre qui concerne la requête souhaitée, appeler le modèle correspondant, et renvoyer une vue liée à la demande.

Troisième partie

L'oeuvre

Chapitre 8

Travail de groupe

8.1 Répartition des tâches

C'est une fois la conception terminée que se fit sentir le besoin de distinguer différentes parties dans le projet. Ceci afin de marcher fermement dans un sens bien définit au lieu de nous aventurer au hasard. Nous sollicitâmes donc quelques conseils auprès de M. Meynard qui nous aida donc à déterminer les grandes parties qui pouvaient être séparées. C'est ainsi que virent le jour trois orientations :

Interface : tout ce qui se voit à l'écran tels fenêtres, onglets, menus et autres.

Système : fonctionnalités qui touchent au système de fichier comme la gestion des fichiers ou d'un espace de travail.

Fonctions : fonctionnalités de gestion du code comme le sont la coloration, l'indentation ou l'autocomplétion

Il est judicieux lorsqu'un projet est fractionné comme l'est le nôtre, de constituer autant de groupes que de parts de travail.
En vertu de ce principe, nous décidâmes de constituer trois groupes constitués comme suit :

- Groupe *interface* : Hamza, Issame et Zaydane ;
- Groupe *système* : Mickael et Joachim ;
- Groupe *fonctionnalités* : Pierre, Olivier et Abdelhamid.

Il est arrivé bien des fois qu'un groupe rencontre un problème ardu ne pouvant être surmonté sans aide extérieure. Heureusement pour lui, les deux autres étaient là pour lui prêter main forte. Ainsi fûrent maintenues tout au long de notre travail une grande cohérence et une grande adéquation relativement à l'évolution des différentes parties.

Néanmoins, les membres du groupe étant d'horizons estudiantins différents (comprenez par là qu'ils ne besognent pas pareillement), il s'avéra nécessaire d'édicter différentes normes quant aux méthodes de travail à adopter.

8.2 Conventions techniques

Voici une liste des conventions prises pour la programmation, les commentaires du code source et le nommage des fichiers.

2. Le nom de fonction désigne aussi les méthodes.

- Les noms de classes, structures, fonctions[2], variables et constantes seront donnés en anglais.
- Les noms de fonctions et variables commencent par une minuscule (une minuscule en anglais bien sûr).
- Les noms de classes et structures commencent par une majuscule.
- Les noms de constantes s'écrivent avec uniquement des majuscules et des tirets bas (le fameux trait du 8).
- Les noms de fichiers qui déclarent et définissent une classe (.h et .cpp) portent le même nom que la classe, avec la majuscule.
- Les noms de fichiers normaux commencent par une minuscule (p.ex. main.cpp).
- L'indentation est laissée libre (accolade en fin de ligne ou en ligne suivante) pour n'embarrasser personne et ne pas créer de controverse.
- Toute classe, fonction ou structure sera bien commentée en utilisant la notation Doxygen.
- Dans une moindre mesure, il sera possible de commenter les variables et constantes importantes avec la même notation.

Bien que les règles citées ci-dessus soient des conventions communément admises (dans le monde C++ du moins), il valait mieux bien les préciser de manière à éviter toute mauvaise surprise.

8.3 Les réunions hebdomadaires

Le grand chamane de l'UM2 a prévu dans notre emploi du temps un créneau d'une heure trente le lundi et le mardi intitulé « Projet ». Étant par nature opportunistes, nous tînmes nos conseils de projet précisément durant ces créneaux.

Une réunion de projet se déroule toujours de la même manière. Chacun des trois groupes résume le travail effectué durant la semaine, fait part de ses problèmes éventuels et donne la liste des choses prévues pour la semaine d'après. S'ensuit une séance de remue-méninges pour discuter, critiquer, approuver ou réprouver ce qui vient d'être présenté. Parfois, des discussions moins techniques venaient nourrir le débat, relatives par exemple à l'organisation de ce document.

Il existe un rapport de chacune de ces réunions.

Chapitre 9

Implémentation

9.1 Arborescence du projet

Le projet est organisé comme sur l'arborescence 9.1. On dispose à la racine du projet de trois sous-dossiers correspondants aux trois éléments du MVC. Chaque fichier source créé trouve sa place de manière évidente dans cette arborescence.
Par exemple, pour qui cherche le fichier *WorkspaceTree.cpp*, il est manifeste que celui-ci se situe dans le dossier *View*.
De même que le fichier *CSSData* qui décrit des données, sera dans le dossier *Model*.
Cette distribution nous mis à l'abri de beaucoup de problèmes qui adviennent dans un travail de groupe, comme la suppression accidentelle ou le sabotage du travail d'autrui.

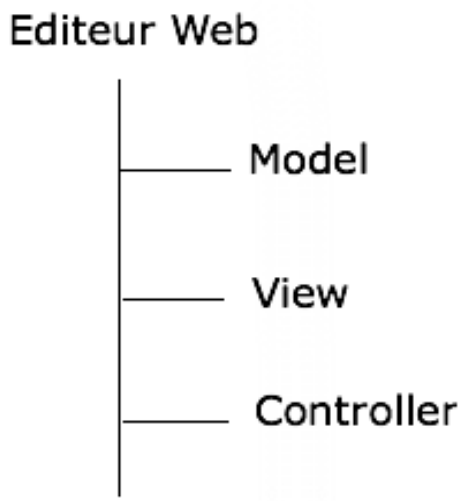

Figure 9.1 – Arborescence de projet

9.2 Système

À ce stade là nous avions donc une idée relativement fixe non seulement de l'organisation des fichiers servant au développement de l'aspect *système* de l'application, mais également de ce qu'allait donner

visuellement notre travail, à savoir une arborescence de fichiers sur la gauche de la fenêtre principale, à la manière d'un Eclipse ou d'un QT Creator, et l'envie de commencer à programmer se faisait plus que jamais ressentir, mais quelques petits détails restaient encore à régler.

En effet, l'utilisation de Qt ainsi que de l'architecture MVC nous contraint à d'abord nous documenter sur ce que ce framework proposait pour répondre à nos attentes. Et la réponse ne se fit pas attendre longtemps puisque le *QDirModel* apparaissait comme correspondant parfaitemement, étant donné qu'il s'agit d'un modèle de données spécialement conçu pour être peuplé par une arborescence de fichiers, à partir du seul chemin vers la racine de cette arborescence.

Mais cette solution s'est rapidement montrée insuffisante pour ce que nous avons prévu, notamment à cause du fichier de configuration de nos projets. Notre idée étant de remplir l'arborescence d'espace de travail de l'utilisateur uniquement par des projets, qui, comme expliqué plus haut, sont des dossiers contenant un fichier de configuration nommé *.pro* à leur racine (dont nous exposerons l'utilité plus tard), et ledit *QDirModel* ne permet pas de filtrer la population du modèle selon un quelconque paramètre.

Nous avons donc pris la décision de créer nous mêmes une fonction remplissant notre modèle de données (selon le type de l'élément à parcourir, i-e WorkSpace, Project ou Folder), nous permettant donc, non seulement de filtrer le remplissage des modèles, mais également d'effectuer certaines actions directement lors du parcours (récursif) de l'arborescence d'espace de travail, comme l'attribution de noms aux éléments trouvés, l'instanciation d'objets correspondant aux éléments (File, Folder, Project), etc...

Ces fonctions (nommées scan dans les classes), consistent en des parcours récursifs de l'arborescence WorkSpace. Outre quelques initialisations de structures (Dirent, Dir, stat, ...) et variables nécessaires au parcours, ainsi que des conversions des chemins de fichiers de type string vers le type char* (pour correspondrent aux signatures de certaines fonction de C++), elles commencent par tester que l'ouverture du dossier choisi comme racine de l'espace de travail s'est bien déroulée, affichant une erreur si ce n'est pas le cas.
Si l'ouverture ne pose pas de problèmes, la ligne folderPath = (char*)malloc(folderPathSize * sizeof(char)) ; alloue la mémoirenécessaire au nom du dossier à parcourir. Cette ligne d'apparence ordinaire, vue maintes et maintes fois dans les applications, mérite d'être soulignée puisqu'elle fait partie de celles qui ont engendré des problèmes de fuites mémoires, longs et pénibles à résoudre, dont nous parlerons plus tard.
L'intérêt de créer notre propre fonction de scan commence d'ailleurs par cette instruction, puisque chaque élément de l'arborescence qui sera ajouté au modèle se voit instancié et affecté d'un attribut "path", une chaine de caractère permettant de stocker le chemin

(relatif à la racine de l'arborescence) du fichier qu'il représente.S'ensuit un léger jeux de concaténations, pour, dans le cas du scan de l'espace de travail, créer le chemin d'un éventuel fichier de configuration ".pro", qui s'il est trouvé engendrera l'ajout à un vecteur de projets du dossier directement parent dudit fichier. Dans ce cas de figure, le scan du projet ajouté est alors lancé, et celui-ci va alors ajouter tous les fichiers et dossiers qu'il contient à ses vecteurs respectifs *file* et *folders*.
Ces fonctions,

9.3 Fonctionnalités

9.3.1 Coloration

9.3.2 Autocomplétion

La mise en place du système d'autocomplétion n'a pas posé de problèmes principalement parce qu'il est inspiré d'un exemple de la documentation.

3. le bloc désigne un morceau de code inclus dans un autre morceau de code.

Le principal problème fut de gérer une autocomplétion contextuelle de bloc[3] et force est de constater que la tâche n'est pas tout à fait achevée.
Pour réaliser une telle prestation, il est nécessaire que notre application puisse faire l'analyse grammaticale du code source. Or aucun analyseur syntaxique (parseur) n'est inclus dans le programme ce qui empêche un fonctionnement optimal de cette fonctionnalité.

Néanmoins nous avons pu ruser et essayer de deviner le bloc courant en fonction des lignes qui précèdent celle où se trouve le curseur. Par exemple, une ligne qui se trouve après une autre ligne qui contient l'expression ‹*script*› dans un code Html, sera autocomplétée avec le dictionnaire du JavaScript.
Cette stratégie trouve rapidement ses limites, par exemple si une balise fermante ‹*/script*› se trouve en début de ligne.

La figure 9.2 représente le module d'autocomplétion en pleine action dans le contexte du langage Php.

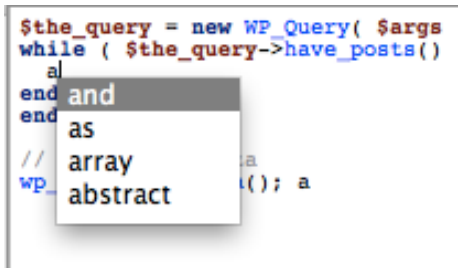

Figure 9.2 – L'autocomplétion en pratique

9.4 Interface

Problèmes rencontrés : Par Hamza. • On a rencontré des difficultés à rassembler le travail qu'a fait Chaque membre du groupe mais qui ont étaient réglées après plusieurs réunions et avec l'aide du groupe chargé de la réalisation du système (Joakim et Mickael) qui nous a aidé à progresser Dans notre travail et aboutir aux contraintes de notre mission. (Exemple : Utilisation de nom de variable différent…)

• On a rencontré des problèmes de lecture et navigation dans la documentation qui n'est disponible qu'en anglais mais qui ont étaient résolues a l'aide des petites recherches sur les forums et sites et avec beaucoup d'efforts a lire l'anglais même si nous y sommes allergiques.

Par zaydane et issam.

1/On n'a pas pu implémenter l'ensemble des éléments utilisés dans l'interface au sein d'une seule classe (MainWindow), par exemple : la TreeView qui se trouve à gauche est qui s'occupe du WorkSpace ainsi que EditorTab qui contient WebView. 2/Lors de l'execution du programme au début, les éléments de l'interface c'est à dire la zonne de texte l'emplacement du TreeView et le menu ont été éparpeillé dans l'ecran et chacune d'eux comme une propre fenetre. 3/ Le travail sur la zonne de texte pour générer plusieurs zonne de textes succéssive en meme temps.

Solutions : 1/ En faisant appel dans la classe MainWindow aux autres classes comme TabEditor et CentralEditor on a pu sortir du conflit de tout faire dans MainWindow sans avoir une interface complète (qui contient tout les éléments). 2/Mettre toutes les fonctions principales d'affichage comme (createMenu() ,createToolbars …) dans une fonction init() était la solution pour rassembler tous les éléments de l'interface dans une seule fenetre. 3/L'appel à la fonction QmdiArea nous a permit de travailler sur plusieurs zonnes de textes en meme temps au lieu d'une seule générée par la Qtextedit comme avant.

Chapitre 10

Résultat

Le meilleur moyen de faire le bilan de ces trois mois de travail est de comparer les résultats obtenus avec le cahier des charges initial.
Mais avant cela, jetons un regard sur le côté graphique de l'application.

Dès l'ouverture de l'application, l'utilisateur peut voir sur son écran là même chose que sur la figure 10.1.

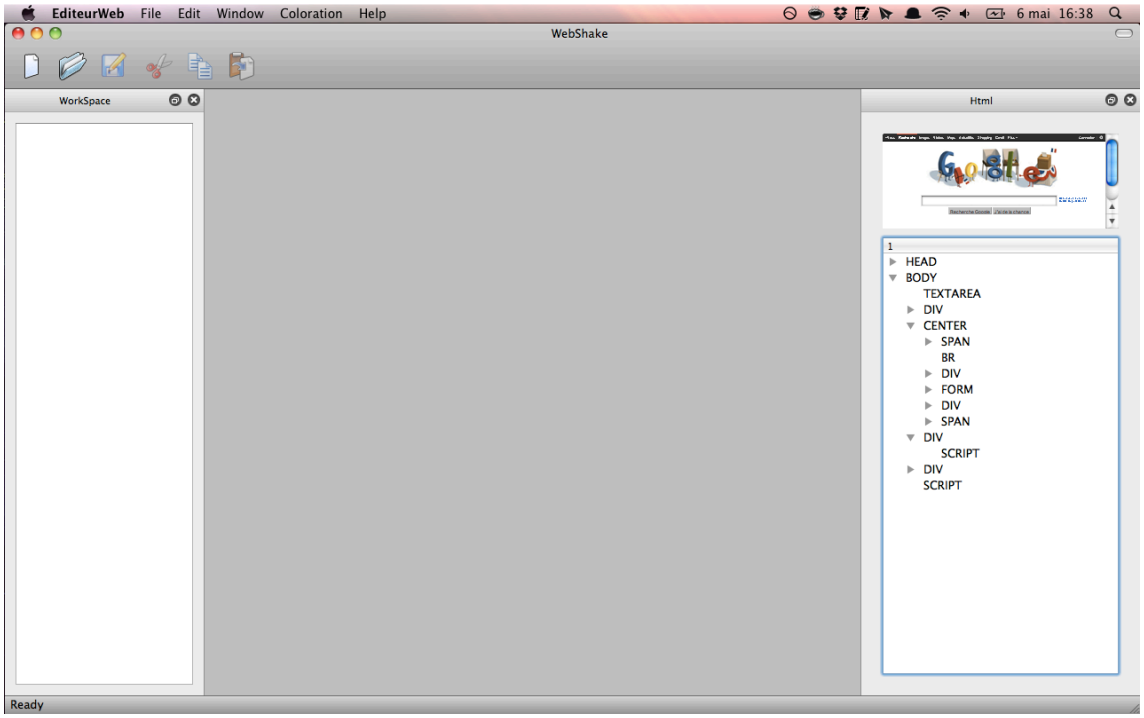

Figure 10.1 – L'écran à l'ouverture de l'application

On aperçoit différents éléments classiques comme la barre de menus par exemple et les trois inévitables boutons des coins de fenêtre qui sont fermer, réduire ou agrandir. Le bandeau d'actions donne à l'utilisateur le pouvoir d'ouvrir ou d'enregistrer, de copier ou de coller à sa guise.

À gauche, se trouve un espace vide pour l'instant mais qui va bientôt accueillir l'arborescence du projet que l'utilisateur s'apprête à créer ou à ouvrir, cet espace incarne le fameux *workspace*, oeuvre du groupe *système*.

La grande région grisâtre au centre de l'écran va recevoir un (ou plusieurs) éditeur de texte spécialisé dans l'un des langages Html, Css, Php ou JavaScript. C'est ici que prennent forme les heures de dur labeur du groupe *fonctionnalités*.

Enfin, la partie de droite est une idée du groupe qui n'était pas dans le cahier des charges. Cet espace semblait vide, inerte et désolé. Alors nous eûmes l'idée d'en faire un emplacement de visualisation. Entendez par là un emplacement dédié au programmeur, pour qu'il voie en direct les effets de ce qu'il est en train d'écrire.
La partie haute est un aperçu sur navigateur du site en cours d'édition et la partie basse une arborescence de l'inclusion des différents blocs de code.
Malheureusement cette fonctionnalité n'a pas abouti du fait notamment qu'il faut parseur pour générer une arborescence.

Dans le tableau 10.1, sont rassemblées toutes les spécifications du cahier des charges (colonne de droite) et il est précisé si notre application possède ou non ces propriétés.

Table 10.1 – Fonctionnalités spéciales dans les éditeurs étudiés

Fonctionnalité	présente ?
Édition de fichiers dans quatre langages	oui
Système multi-onglets	oui
Arborescence du site	oui
Autocomplétion	oui
Coloration syntaxique	oui
Indentation automatique	oui
Accès aux manuels	non
Validation Html	non
Visualisation du site	oui
Squelette de site préexistant	non

Malgré le fait que le cahier des charges soit presque complété, il subsiste dans les parties achevés quelques bogues épars dont nous discuterons dans le chapitre suivant.

Chapitre 11

Discussion

11.1 Cahier des charges

Comme on peut le constater sur le tableau précédent (10.1), le cahier des charges est presque complété. Il reste néanmoins l'accès aux manuels, la validation Html et des sites préfabriqués.
À notre décharge, nous pouvons arguer que ces fonctionnalités ne nous ont pas posé de problème technique mais plutôt organisationnel. En effet, si l'on regarde le diagramme de Gantt de la page 12, on remarque que nous nous apprêtions à travailler jusqu'à la fin du mois de mai ce qui nous aurait laissé le temps de terminer ces modules très simples à mettre en place.

Si l'on juge du point de vue quelque peu pragrmatique, on pourra constater quelques manques à l'application. Citons une fois de plus le parseur qui aurait dû être la base de l'application mais dont la réalisation demandait un temps trop considérable au vu de celui qui nous était imparti.
Fut donc réalisé, avec astuce et stratagème, un « pseudo-parseur » réalisant l'analyse syntaxique à la volée. Cette analyse ne génère pas d'arborescence comme il se devrait et voilà le problème, c'est que celle-ci est indispensable au bon fonctionnement de certains module comme l'autocomplétion contextuelle (cf. page 28).

11.2 Considération graphique

L'application utilise les éléments graphiques du système dans lequel elle est utilisée. Elle n'est donc pas réellement personnalisée et ne possède pas d'identité graphique propre (mis à part les icônes).
Ce manque de charisme, cette sobriété diraient certains, n'influent que très peu sur l'impact réel provoqué sur l'utilisateur. Celui-ci étant un programmeur il est certainement habitué à d'horribles interfaces de programmation voire de ligne de commandes.

Aucune instruction ne nous ayant été donnée quant à la réalisation de cette interface, elle est entièrement de notre cru et nous sommes satisfait du résultat.

11.3 Aller plus loin

Conclusion

Pour faire le bilan de ce projet, rien de tel que donner à chaque membre du groupe quelques lignes pour s'exprimer.

Pierre

Olivier

Hamza

Issame

Mickaël

Joachim

Zaydane

Abdelhamid

Ainsi s'achève ce document (en fait non, il y a encore une annexe après).

Annexe technique

11.4 À propos de Qt

Travailler avec Qt, c'est travailler avec un immense ensemble de classes diverses et variées. Il y a des classes graphiques, des classes pour le XML, des classes pour le son, des classes pour tout ce qu'une application classique peut nécessiter pour fonctionner.

Cet état des choses peut sembler idéal de prime abord, amenant rapidement à des énoncés tels que « Facile, il suffit de prendre telle classe et de l'utiliser. » ou bien « Ah beh je vais employer cette classe elle fait ça très bien. ». Néanmoins, prendre l'habitude de ce genre de discours conduit à la paresse du programmeur qui fournira une application avec des fonctionnalités réduites, limitées par la classe de Qt.

De plus, construire soit même une ensemble de classe en dehors de Qt est très difficile. Au vu du grand nombre de classe qui accomplissent telle ou telle tâche, on se sent obligé de les utiliser. D'autant plus que la documentation nous y encourage en donnant des exemples de codes orientés Qt, transformant de ce fait Qt en élément du langage.

Par exemple, au lieu d'écrire ce code :

```
1 std::string avertissement = "Holà ! cavalier, vous voyagez sans selle !";
2 std::vector<int> v;
```

Il est conseillé d'écrire :

```
1 QString avertissement = "Holà ! cavalier, vous voyagez sans selle !";
2 QVector<int> v;
```

Il est donc presque impossible, si l'on suit cette philosophie, de créer une classe que l'on va utiliser dans un projet Qt et ensuite récupérer dans un autre projet sans faire de modification.

Au contraire, à qui sait bien l'utiliser Qt est une véritable merveille.
Ci-après, le code de la classe JavaScriptHighligher qui regroupe les différents formats associés à des mots du langage. Grâce à l'utilisation pertinente des classes *QSyntaxHighlighter* (dont hérite la classe Highlighter) ou *QRegex*, on peut réduire un colorateur syntaxique complet à ces quelques lignes de code.

La méthode *addRule* ajoute une nouvelle règle de coloration au colorateur tandis que le attributs dont le nom termine par *Format* sont de type *QTextCharFormat*, une classe qui définit des règles de formatage de texte.

```cpp
#include "JavaScriptHighlighter.h"

JavaScriptHighlighter::JavaScriptHighlighter(QTextDocument *parent)
: Highlighter(parent)
{
    // Les nombres.
    numberFormat.setFontWeight(QFont::Bold);
    numberFormat.setForeground(Qt::darkBlue);
    addRule(JavaScriptData::numberRegex, numberFormat);

    // Les méthodes et fonctions.
    functionFormat.setFontWeight(QFont::Bold);
    addRule(JavaScriptData::functionRegex, functionFormat);

    // Les mots clé.
    keywordFormat.setForeground(Qt::darkBlue);
    keywordFormat.setFontWeight(QFont::Bold);
    addRule(JavaScriptData::keywordRegex, keywordFormat);

    // Les mots clé de déclaration.
    keywordDeclarationFormat.setFontWeight(QFont::Bold);
    keywordDeclarationFormat.setForeground(Qt::darkMagenta);
    addRule(JavaScriptData::keywordDeclarationRegex, keywordDeclarationFormat);

    // Les mots réservés.
    keywordReservedFormat.setFontWeight(QFont::Bold);
    keywordReservedFormat.setForeground(Qt::darkYellow);
    addRule(JavaScriptData::keywordReservedRegex, keywordReservedFormat);

    // Les fonctions internes de JavaScript.
    builtInFormat.setForeground(Qt::darkYellow);
    addRule(JavaScriptData::builtInRegex, builtInFormat);

    // Commentaire sur une seule ligne.
    singleLineCommentFormat.setForeground(Qt::red);
    addRule(JavaScriptData::singleLineCommentRegex, singleLineCommentFormat);

    // Les simples et doubles quotes.
    quotationFormat.setForeground(Qt::darkGreen);
    addRule(JavaScriptData::quotationRegex, quotationFormat);

    // Commentaires multilignes.
    multilineCommentFormat.setForeground(Qt::darkRed);
    addMultilineRule(JavaScriptData::multilineCommentStartRegex,
                     JavaScriptData::multilineCommentEndRegex,
                     multilineCommentFormat,
                     IN_COMMENT_STATE);
}
```

Une autre particularité de Qt est que le modèle-vue-contrôleur qu'il utilise n'est pas standard, en ce sens qu'il n'est composé en réalité que du modèle et de la vue, le contrôleur étant fusionné à celle-ci.

Il advint donc souvent que le groupe soit partagé à propos d'un fichier, est-ce une vue ou un contrôleur ? Citons l'exemple des classes de coloration syntaxiques (suffixées par *Highlighter*) qui furent finalement classées parmi les contrôleurs.

Glossaire

Autocomplétion l'autocomplétion, est une fonctionnalité informatique permettant à l'utilisateur de limiter la quantité d'informations qu'il saisit avec son clavier, en se voyant proposer un complément qui pourrait convenir à la chaîne de caractères qu'il a commencé à taper. 2, 3, 6, 8, 20, 25, 28, 29, 32

Bogue En informatique, un bug (de l'anglais bug, « insecte ») ou bogue est un défaut de conception d'un programme informatique à l'origine d'un dysfonctionnement. 31

Css (Cascading Style Sheets : feuilles de style en cascade) est un langage informatique qui sert à décrire la présentation des documents Html et XML. 6, 8, 21, 23, 30

Diagramme de Gantt Le diagramme de Gantt est un outil utilisé en ordonnancement et gestion de projet et permettant de visualiser dans le temps les diverses tâches liées composant un projet. Il permet de représenter graphiquement l'avancement du projet. 12

Doxygen Doxygen est un logiciel informatique libre permettant de créer de la documentation à partir du code source d'un programme. Pour cela, il tient compte de la grammaire du langage dans lequel est écrit le code source, ainsi que des commentaires s'ils sont écrits dans un format particulier. 11, 26

Expression régulière Une expression rationnelle ou expression régulière est en informatique une chaîne de caractères que l'on appelle parfois un motif et qui décrit un ensemble de chaînes de caractères possibles selon une syntaxe précise. Les expressions rationnelles sont issues des théories mathématiques des langages formels. 18

Git Un logiciel de gestion de versions décentralisé. C'est un logiciel libre créé par Linus Torvalds, le créateur du noyau Linux, et distribué selon les termes de la licence publique générale GNU version 2. 11, 39

Html L'Hypertext Markup Language, généralement abrégé Html, est le format de données conçu pour représenter les pages Web. C'est un langage de balisage qui permet d'écrire de l'hypertexte, d'où son nom. 6, 8, 10, 21, 23, 29--32

JavaScript JavaScript est un langage de programmation de scripts principalement utilisé dans les pages web interactives. 6, 8, 21, 29, 30

MVC Le modèle-vue-contrôleur (en abrégé MVC, de l'anglais Model-View-Controller) est un patron d'architecture et une méthode de conception qui organise l'interface homme-machine (IHM) d'une application logicielle (voir chap. 7). 23, 27

Parseur L'analyse syntaxique consiste à mettre en évidence la structure d'un texte, généralement un programme informatique ou du texte écrit dans une langue naturelle. Un analyseur syntaxique (parser, en anglais) est un programme informatique qui réalise cette tâche. 28, 31, 32

Php Le Php : Hypertext Preprocessor, plus connu sous son sigle Php, est un langage de scripts libre principalement utilisé pour produire des pages Web dynamiques. 6, 8, 21, 29, 30

Qt C'est un framework orienté objet et développé en C++. Il offre des composants d'interface graphique (widgets), d'accès aux données, de connexions réseaux, de gestion des fils d'exécution, d'analyse XML, etc. 11, 23, 28, 34, 35

UML Unified Modeling Language, langage de modélisation graphique à base de pictogrammes. 39

WYSIWYG Un WYSIWYG est une interface utilisateur qui permet de composer visuellement le résultat voulu, typiquement pour un logiciel de mise en page, un traitement de texte ou d'image. C'est une interface « intuitive » : l'utilisateur voit directement à l'écran à quoi ressemblera le résultat final. 10

Sitographie

[1] Github : www.github.com
Le site spécialisé dans la gestion de dépôts Git. Possède une interface très pratique pour gérer les groupes, les projets, les fichiers et plus encore.

[2] Wikipédia : www.wikipedia.org
L'encyclopédie en ligne de laquelle j'ai tiré certaines définitions présentes dans le glossaire.

[3] yUML : www.yuml.me
Ce site permet de générer à la volée des diagrammes UML, extrêmement utile lorsqu'il s'agit de travail de groupe.

[4] Micromobs : www.micromobs.com
Le site de discussion de groupe au principe intéressant : il vaut mieux une interface dédiée aux discussions plutôt qu'une boîte e-mail encombrée et mal organisée.

[5] TeamGantt : www.teamgantt.com
Création en ligne de diagrammes de Gantt beaux et lisibles.

[6] Facebook : www.facebook.com
Le réseau social que l'on ne présente plus, il nous a permis d'échanger des messages moins techniques que sur Micromobs et de nous organiser grâce à différents évènements.

[7] Prezi : www.prezi.com
Un générateur de présentation en ligne nouvelle génération.

[8] Qt Reference Documentation : www.doc.qt.nokia.com
La documentation Qt, là où nous avons tout appris.

[9] Stackoverflow : www.stackoverflow.com
Un forum où sont posés et résolus de nombreux problèmes de programmation. Les personnes qui répondent le font de manière très professionnelle ce qui nous a beaucoup aidé.