

Rapport de projet  
Licence 3

## Editeur de sites web



Réalisé par :

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

Sous la direction de :

**Michel Meynard**

# 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
<b>I Analyse</b>	<b>7</b>
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
<b>II Conception</b>	<b>13</b>
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 . . . . .	21
6.5 Classes graphiques . . . . .	22
7 Organisation du code : le modèle MVC	23
<b>III L'oeuvre</b>	<b>24</b>
8 Travail de groupe	25
8.1 Répartition des tâches . . . . .	25
8.2 Conventions techniques . . . . .	25
8.3 Les réunions hebdomadaires . . . . .	26

<b>9</b>	<b>Implémentation</b>	<b>28</b>
9.1	Arborescence du projet . . . . .	28
9.2	Système . . . . .	28
9.3	Fonctionnalités . . . . .	30
9.3.1	Coloration . . . . .	30
9.3.2	Autocomplétion . . . . .	31
9.3.3	Indentation . . . . .	31
9.4	Interface . . . . .	32
<b>10</b>	<b>Résultat</b>	<b>33</b>
<b>11</b>	<b>Discussion</b>	<b>35</b>
11.1	Cahier des charges . . . . .	35
11.2	Considération graphique . . . . .	35
11.3	Aller plus loin . . . . .	35
	<b>Conclusion</b>	<b>37</b>
	<b>Annexe technique</b>	<b>39</b>
11.4	À propos de Qt . . . . .	39
11.5	Le problème du parseur . . . . .	41
11.6	Le problème des fuites mémoires . . . . .	41

# 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	Classes d'indentation . . . . .	20
6.5	Classe CentralEditor . . . . .	21
6.6	Classes de l'interface graphique . . . . .	22
7.1	Les trois catégories du MVC . . . . .	23
8.1	Un extrait du journal de bord . . . . .	27
9.1	Arborescence de projet . . . . .	28
9.2	Les options disponibles au clic droit . . . . .	30
9.3	La coloration en action . . . . .	30
9.4	L'autocomplétion en pratique . . . . .	31
9.5	Un code bien indenté par l'application . . . . .	32
10.1	L'écran à l'ouverture de l'application . . . . .	33

# Liste des tableaux

2.1	Fonctionnalités spéciales dans les éditeurs étudiés . . . . .	10
10.1	Comparatif avec le cahier des charges . . . . .	34

# 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 offrant 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 <sup>1</sup>	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 devons 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.44).

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.

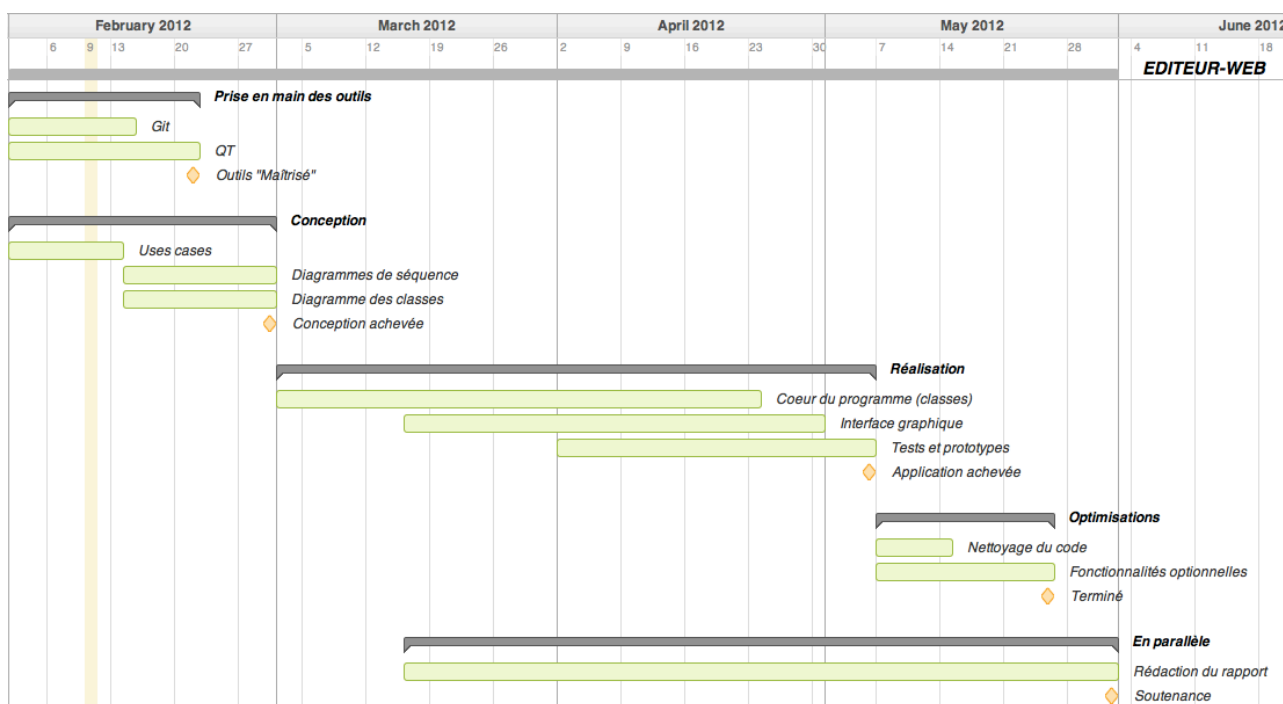


Figure 4.1 – Diagramme de Gantt

## **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.

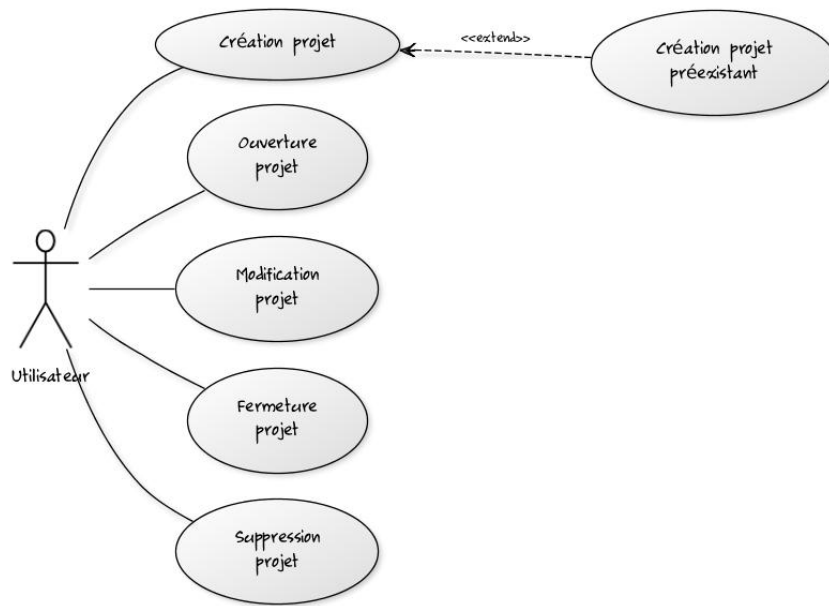


Figure 5.2 – Sous système de gestion des projets

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

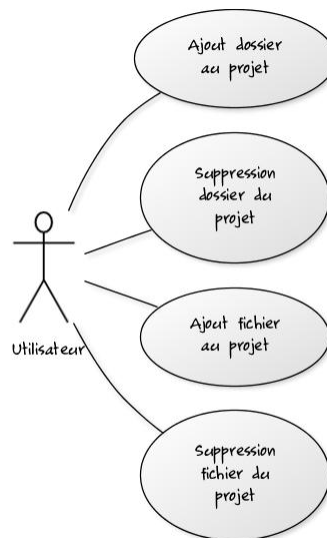


Figure 5.3 – Use-Case modification de projets

Ainsi comme exposé dans ce diagramme, l'utilisateur pourra créer autant de fichiers et de dossiers qu'il le désire dans son 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.

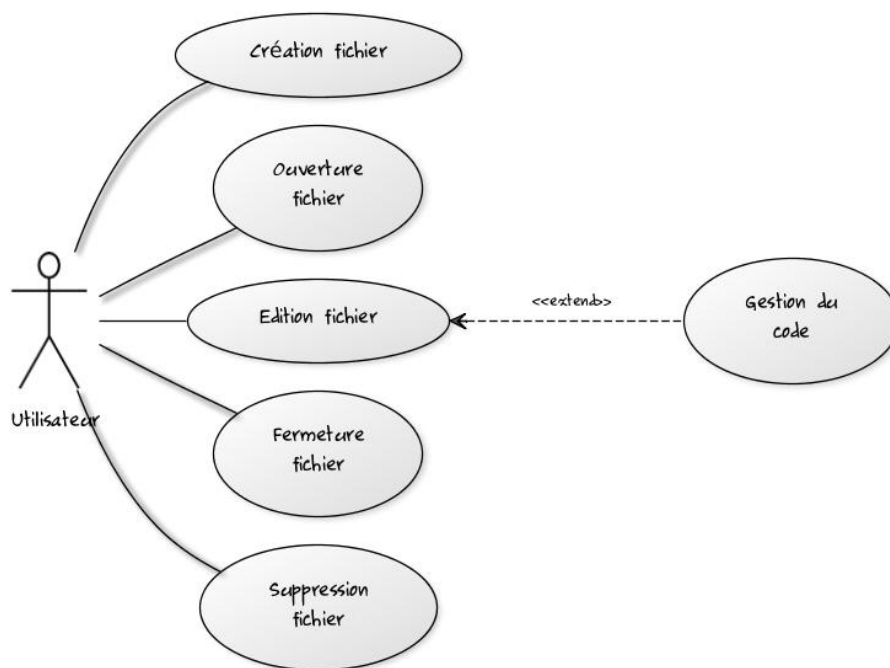


Figure 5.4 – Sous système de gestion des fichiers

## 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.

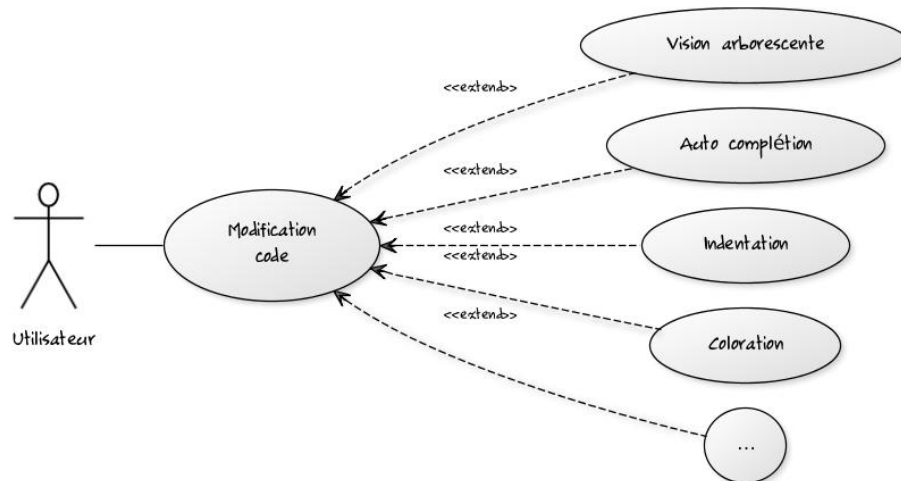


Figure 5.5 – Sous système de gestion du code

# 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.

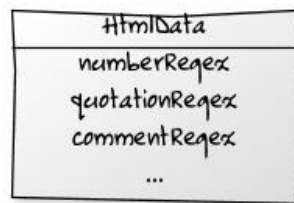


Figure 6.1 – Classe de données pour le langage Html

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*.

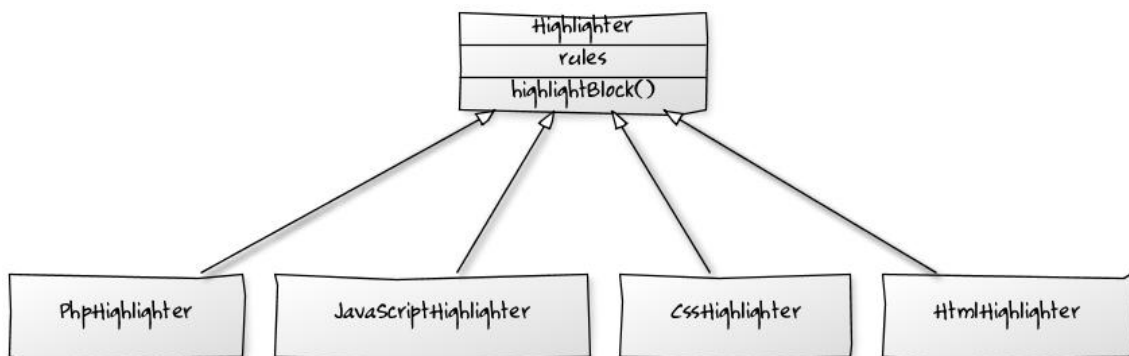


Figure 6.2 – Classes de coloration

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 39.

## 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 avons 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 eu l'idée de traiter dossiers, projets et espaces de travaux comme des instances de classes, pour finalement donner naissance (après quelques essais) au diagramme 6.3.

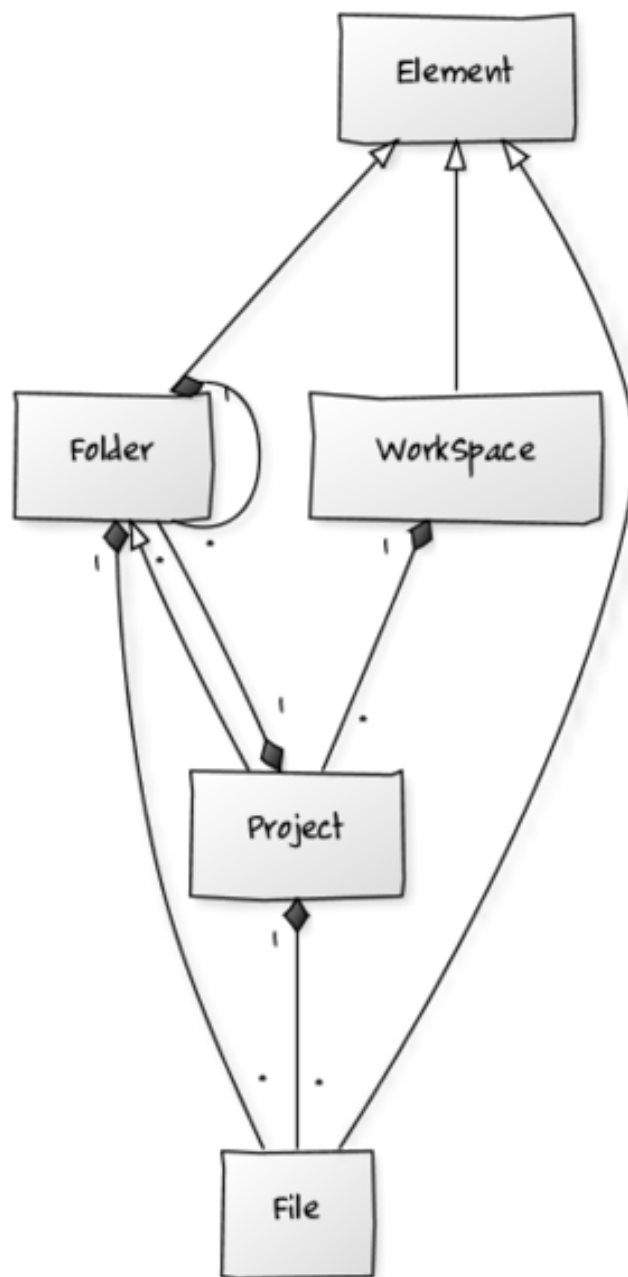


Figure 6.3 – Classes du système de fichiers

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 *Element* 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* quant à lui ne sera pas traité comme un dossier mais plutôt comme un élément particulier ne contenant que des projets.

Pour le reste, les classes *Folder* et *File* (resp. *Dossier* et *Fichier*) parlent d'elles-mêmes.

## 6.3 Indentation

Nous avons définis avec M. Meynard l'indentation automatique comme étant la capacité qu'a le programme à ajouter (ou retrancher) un caractère de tabulation avant une ligne. Nous résolûmes de la mettre en place comme sur le diagramme 6.4.

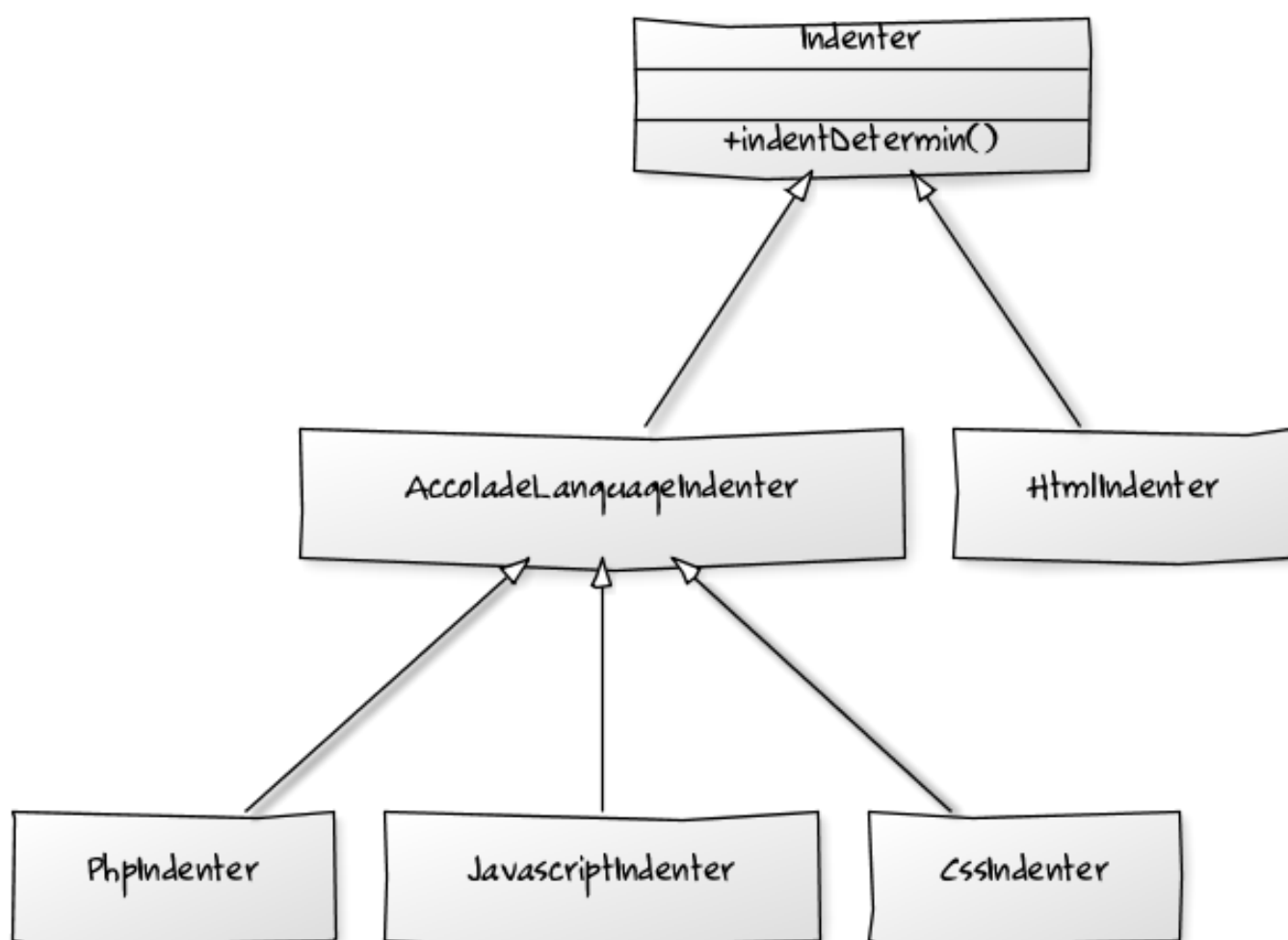


Figure 6.4 – Classes d'indentation

La classe mère *Indenter* est une classe abstraite qui regroupera les méthodes communes à tout indenteur qui se respecte.

Elle est spécialisée par la classe *AccoladeLanguageIndenter* qui, comme son nom l'indique s'occupera de l'indentation des langages dont les blocs sont délimités par des accolades, en l'occurrence le Php, le JavaScript et le Css.

Ces trois langages posséderont chacun une classe d'indentation spécialisée.

Quant à la classe associée au Html, comme celui-ci est langage un peu particulier car il fonctionne par balises, elle spécialisera directement la classe *Indenter*.

## 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 répéter 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 6.5 représente la classe *CentralEditor* limitée aux attributs et méthodes concernant l'autocomplétion :

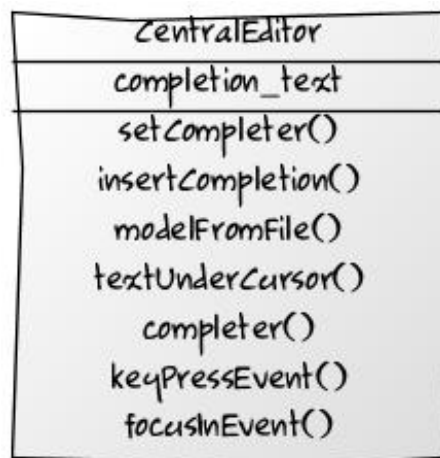


Figure 6.5 – Classe *CentralEditor*

Nous décrivons 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 sélectionner 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 6.6.

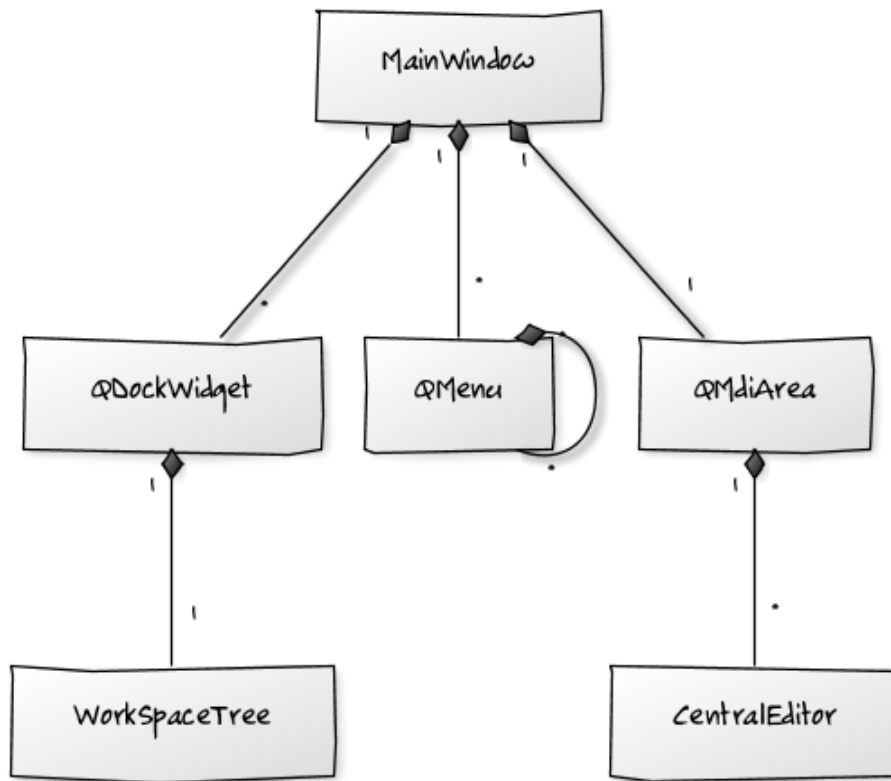


Figure 6.6 – Classes de l'interface graphique

Le principe est d'avoir un grand conteneur qui sera la classe *MainWindow*. Ce qui a le double avantage de centraliser tout ce qui concerne l'interface au même endroit et de respecter les conventions standard Qt.

Toutes les relations présentes dans le diagramme sont des compositions, on observe donc que le *MainWindow* (fenêtre principale) est composée de *QDockWidget*, de *QMenu* et d'un *QMdiArea* respectivement des barres latérales, des menus et des MDI (espaces multidocuments).

On retrouve dans notre barre latérale un *WorkspaceTree* associé au *Workspace* décrit dans la partie 6.2 et qui contiendra une arborescence du ou des projets ouvert(s).

Le *QMenu* sera dédié à des menus classiques avec sous-menus.

Quant au *QMdiArea*, il accueillira un ou plusieurs *CentralEditor*, l'éditeur de texte dans lequel seront appliquées toutes les règles de formatages prévues.

Il est bon de préciser que dans le diagramme 6.6, les classes dont le nom est préfixé par la lettre *Q* sont des classes de Qt, pour les autres, nous en serons les auteurs.

# 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 allons 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.

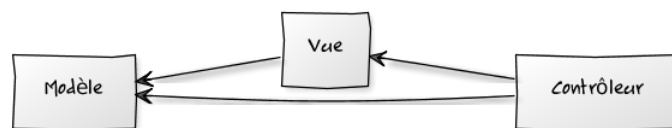


Figure 7.1 – Les trois catégories du MVC

**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 interagir 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 défini et d'éviter des erreurs de planification. 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 furent 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 étudiants 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<sup>2</sup>, 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, sous la forme d'un site créé par les soins d'un membre du groupe. Après chaque colloque, notre scribe désigné se chargeait de résumer la session afin de garder une trace datée et titrée qui servirait de journal de bord.

## 4eme jour

PAR MICKAEL LE 2012-02-20

Notre première réunion avec Mr Meynard sur lavancement de notre projet sest plutot bien passée, mais la quantité de travail demandée commence à être importante. Nous avons donc décidé, dun commun accord avec notre professeur, de commencer à préparer notre programmation avec lanalyse que nous avons à ce moment là, afin de ne pas passer trop de temps à parfaire notre analyse, sous peine de ne plus avoir suffisamment de temps pour terminer notre application.

## 3eme jour

PAR MICKAEL LE 2012-02-14

Au vu dune première réunion avec M Meynard pour voir où nous en sommes, nous avons déjà divisé notre groupe de 8 personnes en 3 groupes à effectif réduit, et chacun des groupes à commencé à réaliser des schémas UML détaillant les parties qui lui sont affectés

## 2eme jour

PAR MICKAEL LE 2012-02-13

Une semaine après, nous n avons pas réellement avancé la rédaction mais nous nous sommes mis daccord notamment pour le choix des outils (comme git pour le SVN, QT Creator pour le développement, etc...) et nous avons choisis également des outils de partage et de collaboration (micromobs, facebook, etc...), Nous decidons egalemt dorganiser des reunions de groupe toutes les semaines, pour des raisons dorganisation.

## 1er Jour

PAR MICKAEL LE 2012-02-07

Tous convoqués par monsieur Meynard, nous découvrons ensemble les attentes de notre professeur quand au sujet que nous avons choisi. On nous demande dores et déjà de choisir un langage de programmation, ainsi qu une librairie à utiliser pour développer notre application (en loccurrence Qt).

Figure 8.1 – Un extrait du journal de bord

# 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.

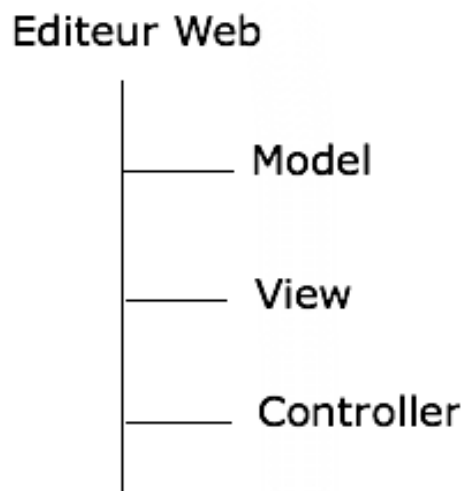


Figure 9.1 – Arborescence de projet

### 9.2 Système

Arrivés au stade de l'implémentation système, nous avons une idée relativement fixe de l'organisation des fichiers servant au développement de l'aspect système de l'application. Nous savions également 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. 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 nous documenter sur ce que ce framework proposait pour répondre à nos attentes.

La réponse ne tarda point puisque le *QDirModel* apparaissait comme correspondant parfaitement, étant donné qu'il s'agit d'un modèle de données spécialement conçu pour être peuplé par une arborescence de fichiers.

Malheureusement, cette solution s'est rapidement montrée insuffisante pour ce que nous prévoyions, 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.

Le *QDirModel* ne permet pas de filtrer la population du modèle selon un quelconque paramètre donc a fortiori le paramètre d'existence d'un éventuel *.pro*.

Nous arrê tâmes de créer nous même des fonctions remplissant notre modèle de données selon le type de l'élément rencontré (*WorkSpace*, *Project* ou *Folder*). Ce qui permettait non seulement de filtrer le remplissage des modèles mais également d'effectuer certaines actions directement lors du parcours (récursif) de l'arborescence de fichiers. Par exemple attribuer des noms aux éléments trouvés, instancier des objets correspondant aux éléments (*File*, *Folder*, *Project*) et bien d'autres choses.

Ces fonctions (nommées *scan* dans les classes), consistent en des parcours récursifs de l'arborescence *WorkSpace*. La difficulté réside dans la cohabitation de nos classes avec celles du C++.

Outre quelques initialisations de structures (*Dirent*, *Dir*, *stat*, ...) et variables de parcours, il faut des conversions des chemins de fichiers de type *string* vers le type *char\** pour correspondre aux signatures de certaines fonction de C++.

Si l'ouverture ne pose pas de problèmes, la ligne

```
folderPath = (char*)malloc(folderPathSize * sizeof(char));
```

alloue la mémoire nécessaire pour le parcours.

Cette ligne d'apparence ordinaire mérite d'être soulignée puisqu'elle fait partie de celles qui ont engendré des problèmes de fuites mémoire, dont la résolution fut longue et pénible.

L'intérêt de créer notre propre fonction de scan commence d'ailleurs par cette ligne, puisque chaque élément de l'arborescence qui sera ajouté au modèle se voit instancié et affecté d'un attribut *path*, une chaîne de caractère permettant de stocker le chemin (relatif à la racine de l'arborescence) du fichier qu'il représente. S'ensuit un léger jeu de concaténations pour éventuellement créer le chemin d'un fichier de configuration *.pro*, qui s'il est trouvé augmentera la liste des projets gérés.

Une fois le workspace choisi, une vue interactive sur l'arborescence des projets est disponible. Le double clic entraîne l'ouverture du fichier ou l'ouverture/fermeture d'un dossier/projet. Le clic droit quand à lui ouvre un menu déroulant, dans lequel on retrouve les options classiques : déplacement, copie, renommage, ouverture, suppression, ajout de fichier, de dossier, et caetera.

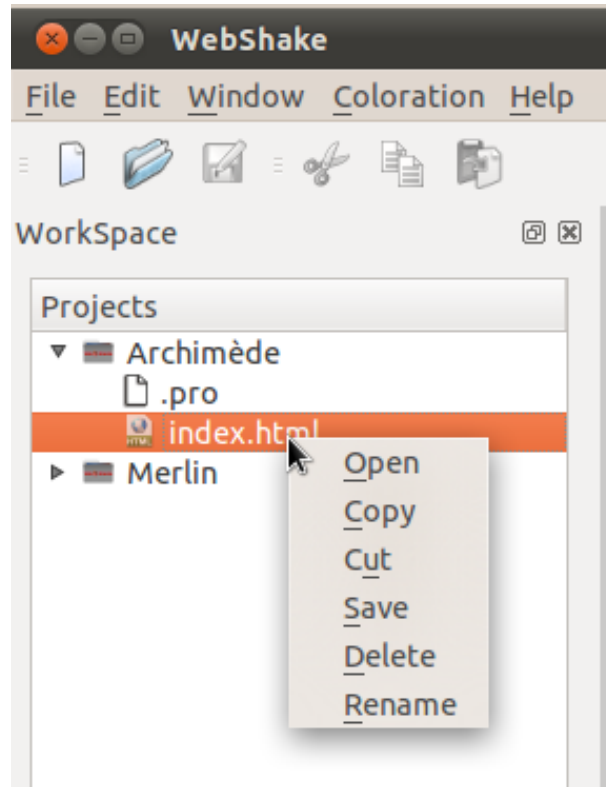


Figure 9.2 – Les options disponibles au clic droit

## 9.3 Fonctionnalités

### 9.3.1 Coloration

La coloration est un principe simple et rapide en mettre en oeuvre, ce fut la première fonctionnalité à être implémentée.

Le principal problème est de gérer une coloration sur plusieurs lignes étant donné que l'analyse du texte se fait ligne par ligne.

```
div.bloc_inscription a, #logbox div.bloc_connexion a { color: #D3E9FF;
div.bloc_inscription { float: left; margin: 0 }
div.bloc_connexion { margin: 0 0 0 51% }
a strong.liens_invites { font-size: 1.5em; color: #174A86; }
a: hover strong.liens_invites { color: #C87B02 }
edbarre

lin: 0;
ling: 95px 0 0 26px;
tyle-type: none;
tt: 27px;
ground: url("../images/designs/2/speedbarre/bg_left_speed.png") no-

chap /* L'ensemble icône + titre chapitre */
t: 71px;
left;
ground-color: #FFFFFF;
lin-right: 0 20px 0 0;
size: 12px;
ion: relative; top: -8px;
```

Figure 9.3 – La coloration en action

### 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.

Le principal problème fut de gérer une autocomplétion contextuelle de bloc<sup>3</sup> 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 9.4 représente le module d'autocomplétion en pleine action dans le contexte du langage Php.

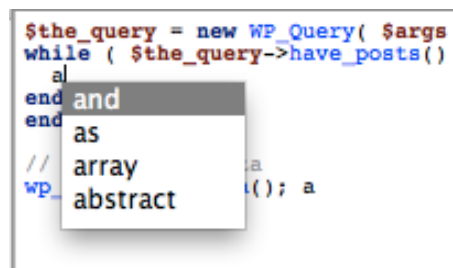


Figure 9.4 – L'autocomplétion en pratique

### 9.3.3 Indentation

L'implémentation de ce module ne fut pas une mince affaire, car qui veut indenter un morceau de code passe de préférence par une arborescence représentative dudit code. Ce qui nécessite la mise en place d'un parseur.

Après nous être renseignés, notamment auprès de M. Meynard, nous prîmes la décision de créer un analyseur basique. Celui-ci se chargera de lire le code, d'en extraire des mots importants (avec des expressions régulières) et de décider en fonction de deux paramètres le niveau d'indentation de la ligne courante.

Ces paramètres sont :

- La profondeur d'indentation de la ligne précédente ;
- La présence ou non de caractère spéciaux dans la ligne précédente (par exemple l'accolade ouvrante) ;

Dès lors la fonction d'indentation ne fut plus qu'une formalité si l'on néglige les quelques heures à traquer les erreurs telles que balise ouvrante et fermante sur la même ligne par exemple.

---

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



```

function __construct()
{
    $this->connexion=new Connexion_Base();
    $this->tab_num_musique=array();//sera en hidden
    $this->tab_titre=array();
    $this->tab_album=array();
    $this->tab_duree=array();
    $this->tab_groupe=array();
}

```

Figure 9.5 – Un code bien indenté par l'application

## 9.4 Interface

Le principal inconvénient concernant la mise en oeuvre de l'interface fut l'apprentissage de la documentation.

Il fallait non seulement comprendre les concepts de programmation d'interface graphique mais en même temps mettre en application lesdits concepts.

Nous rencontrâmes donc plusieurs problèmes de synchronisation car les différents membres du groupe n'étaient pas toujours d'accord sur des détails techniques.

Finalement, les trois personnes chargées de l'interface conçurent un modèle simple basé sur la stratégie de l'éparpillement.

Chaque élément de l'interface est séparé des autres dans une classe spécialisée avant d'être regroupés dans la *MainWindow*.

Il est possible d'admirer le résultat du travail du groupe *Interface* dans le chapitre suivant.

# 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.

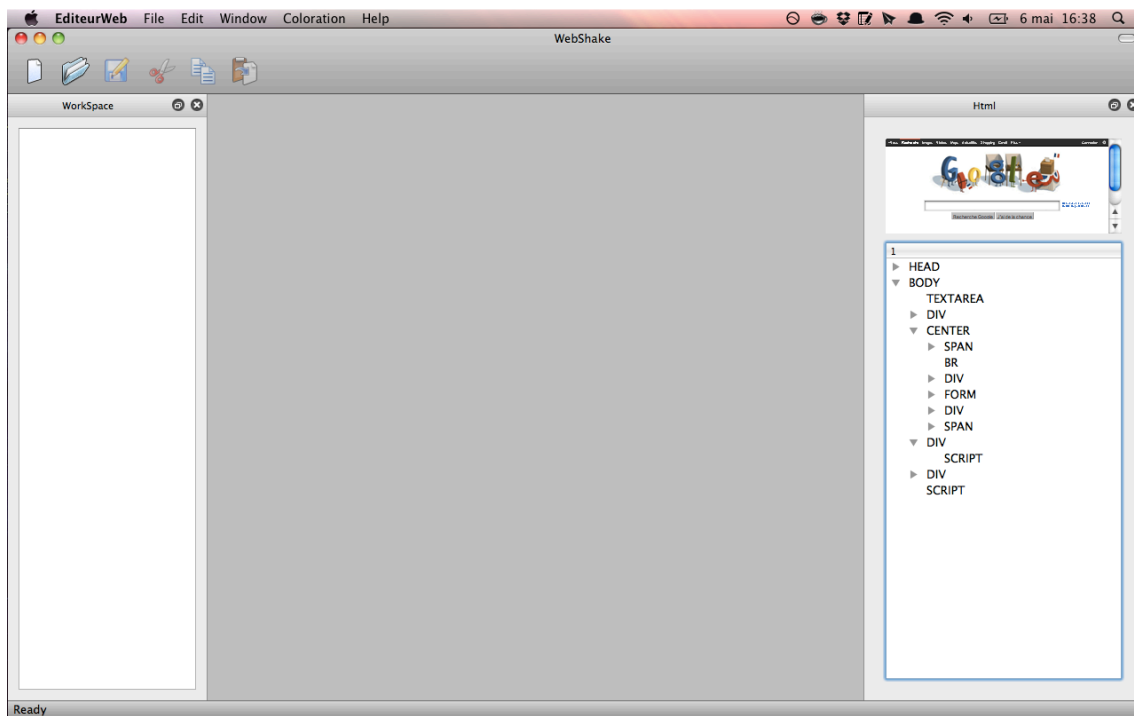


Figure 10.1 – L'écran à l'ouverture de l'application

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 – Comparatif avec le cahier des charges

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 30, 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 pragmatique, 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 31).

### 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

L'application est perfectible à bien des égards, les fonctionnalités principales sont présentes mais nous pourrions tout aussi bien ajouter :

- Une fonctionnalité de glisser déposer pour gérer les fichiers intuitivement dans le panneau gauche.
- Différentes palettes de couleurs pour la coloration syntaxique.
- Un *WorkSpace* par défaut.
- Un mode de formatage d'un fichier mal indenté.

De même, il est possible d'ajouter des éléments graphiques pertinents, comme un panneau proposant la documentation du langage actuellement utilisé.

# Conclusion

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

## Pierre

Pour ma part, ces 3 mois de travail ont été une grande source d'apprentissage. J'ai découvert une nouvelle façon de travailler, grâce à l'utilisation d'un gestionnaire de version tel que GitHub. Réaliser un projet regroupant 8 personnes était également une grande première. Ceci a pu nous donner un aperçu des inconvénients et des avantages de réaliser un logiciel en grand nombre. Difficultés de communication, de synchronisation, de partage des tâches. . . Cependant, cette expérience a surtout été bénéfique. Apprendre à dialoguer, s'entraider, échanger ses idées et débattre autour des stratégies à adopter a été très enrichissant. Enfin, la découverte du framework QtCreator m'a énormément plu. Relativement intuitif et assez rapide à prendre en main, il offre un confort non négligeable dans la programmation, avec des bibliothèques très nombreuses et très utiles.

En somme, la tâche qui nous a été confiée m'a appris énormément de choses, que ce soit dans le domaine de la programmation pure ou dans les relations entre tous les membres d'un groupe.

## Olivier

Tout au long de ces trois mois, il n'y avait pas une seule semaine qui apportait sa nouveauté. L'initiation à un système de gestion de version tel que GitHub, l'apparition d'un nouvel environnement de développement avec QtCreator (et ses nombreuses bibliothèques, sa documentation touffue), la direction d'un groupe de huit personnes, toutes ces choses ont été nouvelles pour moi.

Concernant la gestion du groupe, il m'a fallu être démocratique, laisser fleurir les idées en vrac, canaliser chaque esprit vers notre objectif. Il a fallu parfois prendre des décisions difficiles rapidement, tout en respectant les opinions de chacun. Occasionnellement, il a fallu également forcer le destin et s'imposer dans les moments difficiles.

Pour ce qui est de la programmation le MVC m'a permis de ne pas me disperser, de respecter une standardisation connue. L'utilisation d'un nouveau framework, et l'utilisation de sa documentation a été forcément pour moi une nouveauté. Ce dernier m'a réellement plu pour son utilisation simpliste, épurer ... bref tous les avantages d'un IDE.

Ce projet m'a également ouvert les yeux sur l'ampleur de la prise en main d'un tel projet. Proche d'un logiciel commercial, ce dernier m'a encouragé à persévérer dans mes convictions et dans la concrétisation sous mes yeux d'un projet d'importance.

Ce projet m'a beaucoup plu. Il m'a apporté toutes les expériences proches de celles des entreprises. Tant au point de vue collaboration concernant le groupe, et tout autant au point de vue apprentissage et prise en main de nouveaux outils très pratique pour la situation à laquelle nous étions confrontés.

**Hamza**

**Issame**

**Mickaël**

**Joachim**

Ma conclusion : Pour ma part le projet fût une bonne expérience, premièrement il m'a permit de développer mes compétences dans le langage c++, apprendre à utiliser Qt et tout un tas d'outils jamais utilisé jusqu'alors : Valgrind, Git, yUml, etc. Deuxièmement, ayant déjà réalisé un projet à l'IUT, il ne m'était jamais arrivé de travailler avec autant de personne. Celà fût fort agréable, car en ayant réussi à diviser notre projet en sous parties, on voyait vraiment que chacun des groupes apportait sa pierre à l'édifice tout en restant cohérent sur le résultat final. Pour finir je dirais que le projet nous permet à chacun de nous de toucher du doigt notre futur travail, ce qui est vraiment motivant. J'ai lu le rapport hier, j'ai souligné quelques fautes par ci par là, tu voudras que je te les montre ?

**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 `JavaScriptHighlighter` 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.



```

1  #include "JavaScriptHighlighter.h"
2
3  JavaScriptHighlighter::JavaScriptHighlighter(QTextDocument *parent)
4  : Highlighter(parent)
5  {
6      // Les nombres.
7      numberFormat.setFontWeight(QFont::Bold);
8      numberFormat.setForeground(Qt::darkBlue);
9      addRule(JavascriptData::numberRegex, numberFormat);
10
11     // Les méthodes et fonctions.
12     functionFormat.setFontWeight(QFont::Bold);
13     addRule(JavascriptData::functionRegex, functionFormat);
14
15     // Les mots clé.
16     keywordFormat.setForeground(Qt::darkBlue);
17     keywordFormat.setFontWeight(QFont::Bold);
18     addRule(JavascriptData::keywordRegex, keywordFormat);
19
20     // Les mots clé de déclaration.
21     keywordDeclarationFormat.setFontWeight(QFont::Bold);
22     keywordDeclarationFormat.setForeground(Qt::darkMagenta);
23     addRule(JavascriptData::keywordDeclarationRegex, keywordDeclarationFormat);
24
25     // Les mots réservés.
26     keywordReservedFormat.setFontWeight(QFont::Bold);
27     keywordReservedFormat.setForeground(Qt::darkYellow);
28     addRule(JavascriptData::keywordReservedRegex, keywordReservedFormat);
29
30     // Les fonctions internes de JavaScript.
31     builtInFormat.setForeground(Qt::darkYellow);
32     addRule(JavascriptData::builtInRegex, builtInFormat);
33
34     // Commentaire sur une seule ligne.
35     singleLineCommentFormat.setForeground(Qt::red);
36     addRule(JavascriptData::singleLineCommentRegex, singleLineCommentFormat);
37
38     // Les simples et doubles quotes.
39     quotationFormat.setForeground(Qt::darkGreen);
40     addRule(JavascriptData::quotationRegex, quotationFormat);
41
42     // Commentaires multilignes.
43     multilineCommentFormat.setForeground(Qt::darkRed);
44     addMultilineRule(JavascriptData::multilineCommentStartRegex,
45                     JavascriptData::multilineCommentEndRegex,
46                     multilineCommentFormat,
47                     IN_COMMENT_STATE);
48 }

```

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.

## 11.5 Le problème du parseur

Comme il a été dit plusieurs fois le long de ce document, un bon parseur eut été utile pour simplifier le travail de programmation. Nous ne demandions pourtant rien de plus qu'un petit parseur qui prendrait en compte la grammaire la plus grossière de chaque langage et en générerait une arborescence. Au lieu de cela, nous trouvâmes sur le web des outils mal adaptés, trop complexes, non documentés voire non fonctionnels.

Pour remédier à celà, il eut fallu effectuer un travail plus poussé (et bien plus long) sur l'analyse du texte. Par exemple, créer un parseur basique à l'aide des outils Flex/Bison. Ce fut un moment notre idée, mais vite abandonnée par manque de temps.

## 11.6 Le problème des fuites mémoires

Dans la section 9.2, sont brièvement évoqués des problèmes de fuites mémoire. Ce problème fut découvert à la suite d'un test effectué comme suit. En supposant que des utilisateurs viendraient à utiliser notre application pour gérer un nombre important de sites web, nous avons importé environ 16000 fichiers dans un *workspace*.

Pour une telle base, le temps pour ouvrir l'espace de travail était de l'ordre de 6 à 7 secondes, sur un PC équipé d'un processeur i3.

Mais le plus inquiétant n'était pas vraiment cette lenteur.

En effet en étudiant l'occupation mémoire de l'application, nous aperçûmes grâce à la commande *top*, qu'à chaque réouverture du même *workspace*, l'occupation en mémoire augmentait de 2 à 3% sur une mémoire de 1Go.

Il fallut donc utiliser un outil conçu pour corriger ces fuites : Valgrind.

L'analyse de notre logiciel avec cet outil nous permis de trouver plusieurs mauvaises utilisations d'allocations et libérations mémoires. En effet nous constatâmes que des allocations mémoires se faisaient sur un même objet en boucle, remplissant rapidement l'espace disponible.

Valgrind nous fit remarquer que sur certains objets ou pointeurs, des confusions entre les libérateurs *free*, *delete* et *delete[]* étaient présentes, conduisant à une mauvaise libération de la mémoire.

Finalement les fuites se sont fortement réduites passant à 0.1%.

Malheureusement, certaines ne sont pas inhérente à notre code, Valgrind notant des fuites dans des appels de Qt (fonctions que nous ne pouvons pas modifier), nous ne sommes pas parvenus à atteindre le 0%.

En ce qui concerne le temps de chargement, nous nous sommes aperçus qu'à chaque création d'un élément (fichier, dossier, projet etc...) une icône était créée pour ce dernier.

Nous avons donc choisi d'insérer dans la classe *Tools* qui contient un certain nombre d'éléments statiques, de charger toutes les icônes du projet une fois statiquement, et de lier ces icônes aux éléments.

Finalement le temps d'ouverture de ce workspace est tombé à environ une seconde, ce qui est nettement mieux.

# 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. 5, 7, 20, 24, 29, 30, 34

**Bison** GNU Bison est l'implémentation de l'analyseur syntaxique yacc par le projet GNU. 40

**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. 33

**C++** Le C++ est un langage de programmation permettant la programmation sous de multiples paradigmes comme la programmation procédurale, la programmation orientée objet et la programmation générique. C++ est actuellement le 4e langage le plus utilisé au monde. 10, 25, 28

**Css** (Cascading Style Sheets : feuilles de style en cascade) est un langage informatique qui sert à décrire la présentation des documents Html et XML. 5, 7, 19, 20, 22, 32

**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. 10, 11

**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. 10, 25

**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. 17, 30

**Flex** Flex est un outil pour générer des analyseurs, programmes qui reconnaissent des motifs lexicaux dans du texte. 40

**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. 10, 41

**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. 5, 7, 9, 19, 20, 22, 30, 32--34

**JavaScript** JavaScript est un langage de programmation de scripts principalement utilisé dans les pages web interactives. 5, 7, 19, 20, 30, 32

**MDI** En Informatique, Multiple Document Interface ou MDI désigne l'organisation de l'interface graphique d'une application où des fenêtres « parentes » contiennent en leur sein des fenêtres enfants. Le cas typique d'application consiste en la fenêtre principale de l'application, avec un menu et des barres d'outils, contenant une (sous-)fenêtre par fichier ou projet ouvert. 21

- 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). 22, 27, 28
- 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. 30, 33, 34, 40
- 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. 5, 7, 19, 20, 30, 32
- 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. 10, 21, 22, 28, 38--40
- UML** Unified Modeling Language, langage de modélisation graphique à base de pictogrammes. 41
- Valgrind** Valgrind (prononcé [væl r nd], et non [væl ra nd]<sup>1</sup>) est un outil de programmation libre pour déboguer, effectuer du profilage de code et mettre en évidence des fuites mémoires. 40
- 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. 9

# Sitographie

- [1] Github : [www.github.com](http://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](http://www.wikipedia.org)  
L'encyclopédie en ligne de laquelle j'ai tiré certaines définitions présentes dans le glossaire.
- [3] yUML : [www.yuml.me](http://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](http://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](http://www.teamgantt.com)  
Création en ligne de diagrammes de Gantt beaux et lisibles.
- [6] Facebook : [www.facebook.com](http://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 événements.
- [7] Prezi : [www.prezi.com](http://www.prezi.com)  
Un générateur de présentation en ligne nouvelle génération.
- [8] Qt Reference Documentation : [www.doc.qt.nokia.com](http://www.doc.qt.nokia.com)  
La documentation Qt, là où nous avons tout appris.
- [9] Stackoverflow : [www.stackoverflow.com](http://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.
- [10] Le site du zéro : [www.siteduzero.com](http://www.siteduzero.com)  
Un ensemble bien organisé de cours dans différentes branches de l'informatique et un forum très actif font du site du zéro un bon choix pour les débutants.
- [11] Developpez.com : [www.developpez.com](http://www.developpez.com)  
Ce site est une source inépuisable de données, documentations, questions et réponses concernant la programmation dans tout les langages.

[12] Sourceforge : [www.sourceforge.net](http://www.sourceforge.net)

Le site de référence en matière de projets open source, c'est ici que nous avons trouvés des exemples de parseurs.

[13] QtFr.org : [qtfr.org](http://qtfr.org)

Forum de la communauté francophone de Qt, pour les anglophobes.