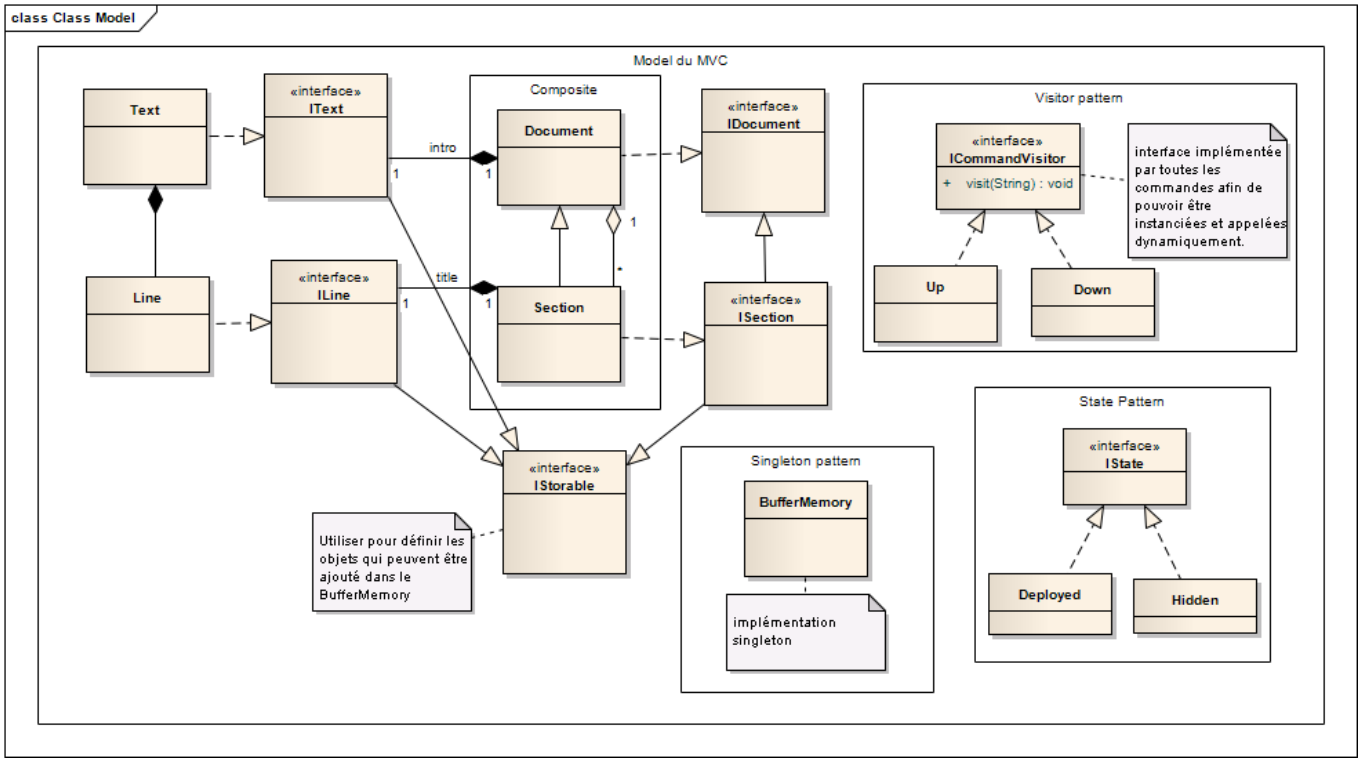
Rapport d'Architecture Logicielle :

Développement d'un éditeur de texte prenant en charge les fichiers arborescents de type Org.

I - Architecture du projet

L'objectif de ce projet est d'essayer d'établir l'architecture optimale pour un logiciel d'édition de fichier Org (structure arborescente) puis d'essayer de l'implémenter. La contrainte de ce projet était de respecter les contraintes normales de développement à savoir : garantir la maintenabilité et la lisibilité du code. Pour cela, il est nécessaire de passer par une étape de modélisation importante afin d'essayer de parvenir à la meilleure structure. L'utilisation de patrons de conception standardisés est aussi un point clé car cela permet à quelqu'un connaissant le patron de voir plus facilement les aboutissants de la classe et de détecter plus facilement les erreurs de code.



Notre projet emploie divers patrons de conception qui sont indiquées par des carrés sur le diagramme ci-dessus. Ces patrons de conception seront détaillés dans la partie suivante.

Concernant la principale structure de données qui est la représentation de l'arbre, nous avons décidé d'utiliser le patron composite qui nous semblait le plus adapté. Celui-ci implique divers structure que nous allons détaillés ci-dessous :

- IDocument & Document : représente la structure de base c'est-à-dire qu'un document est composé d'un texte introductif et d'une suite de section. Cette structure : introduction + sous-sections est commune avec la structure d'une section.
- ISection & Section : elle représente une sous-partie du document et est composé d'une introduction, de sous-sections et d'un titre. Cette structure étant très proche de celle de document, il y a une relation d'héritage car en terme de structure : "une section est un document avec un titre".

- IText & Text : c'est l'implémentation dans le modèle du texte, il est basé sur des ILine et non sur des String. Il emploie des ArrayList<ILine> afin de pouvoir représenter le fait qu'un texte est composé de plusieurs lignes.
- ILine & Line : c'est une classe d'encapsulation du StringBuilder qui permet d'effectuer certains comportement spécifiques avant d'appeler les méthodes standards. Un patron proxy aurait aussi pu être utilisé mais la classe d'encapsulation nous paraissait plus adéquate.
- BufferMemory : c'est l'implémentation de la pile de sauvegarde des données qui est appelée lors d'un copier/couper - coller. Cette partie est implémentée en singleton car a pour vocation à être utilisée lors de l'édition de plusieurs documents afin de permettre de faire des copier-coller d'un document à l'autre.
- IState, Deployed & Hidden : c'est la représentation des différents états pour une section, suivant l'état le comportement ne va pas être le même (patron état).
- ICommandVisitor & commandes : c'est l'implémentation basée sur un visiteur qui permet d'ajouter et de renommer une commande de manière dynamique et simple.

Cela ne présente que la partie modèle et pourrait être complétée par les parties dédiées à la vue et aux contrôleurs. Les contrôleurs servent uniquement de liaison entre la couche haute et la couche basse et sera détaillée dans la partie II.

II - Principes et patrons utilisés

L'objectif de ce projet est d'obtenir un code évolutif et facile à maintenir. Afin de pouvoir remplir cette mission, divers patrons de conceptions ont été utilisé. Ce paragraphe va décrire les différents principes utilisés.

1 – Le patron MVC : Une méthode de conception garantissant l'évolutivité

C'est une méthode de conception dont l'objectif est d'imposer un découplage entre le modèle, la vue et les contrôleurs. Cette méthode de programmation part d'un constat simple : alors qu'un modèle est le plus souvent unique car développé une fois pour toute au début d'un projet ; la vue peut nécessiter d'être remaniée afin de pouvoir être améliorée par des ergonomes et des designers ou simplement pour pouvoir s'adapter aux nouvelles technologies (mobilité). Par conséquent, il est devenu nécessaire de pouvoir découpler le modèle et la vue afin de pouvoir avoir un code évolutif. Cette méthode se compose de trois différentes parties :

- Le modèle : c'est l'implémentation de bas niveau et le « cœur » du programme, il est en charge du traitement des données et de la gestion de l'intégrité de celles-ci.
- La vue : C'est la couche de présentation du logiciel, l'utilisateur intéragit avec cette couche, elle est nécessaire pour la représentation (le plus souvent graphique) des résultats et des données transmises par le modèle.
- Les contrôleurs : C'est la couche de liaison, elle transmet les modifications du modèle à la vue et les interactions de l'utilisateur au modèle.

Ce patron permet de respecter le principe de modularité imposé par le sujet et décrit comme suit : « l'ajout d'un second mode d'interaction ».

2 – Le patron observateur

Définition : « *Le pattern Observateur définit une relation entre objets de type un-à-plusieurs, de façon que, lorsque un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.*» (Designs Pattern – La tête la première).
Ce patron présente le principal intérêt de permettre à une application d'être dynamique. En effet, un changement dans le modèle sera notifié directement à la vue qui va se mettre à jour instantanément. Cela permet d'éviter tous les problèmes d'attentes actives et permet ainsi de minimiser les ressources CPU surtout sur les projets de grandes envergures.

3 – Le patron composite

Définition : « *Le pattern Composite compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet aux clients de traiter de la même façon les objets individuels et les combinaisons de ceux-ci.*» (Designs Pattern – La tête la première).
Ce patron est essentiel dans le cas de la représentation de structure arborescente, le standard de fichier Org étant une structure arborescente, ce patron semblait le plus adapté pour la réalisation de ce projet. Il a été utilisé pour la structure des ISection et des Section. Ce patron présente souvent la contrainte d'imposer une programmation récursive qui peut, parfois, poser des problèmes de critères d'arrêts.

4 – Le patron fabrique

Ce patron est utilisé afin de faciliter l'utilisation d'un logiciel ou d'une API par l'utilisateur/le développeur. En effet, lorsqu'une fonctionnalité est ajoutée à un logiciel, il y a le plus souvent plusieurs implémentations pour la même interface ; à l'utilisation, les utilisateurs ne savent pas quelle est la classe d'instanciation idéale. L'intérêt de ce patron est donc de masquer l'instanciation par l'utilisation de méthodes spécifiques. Cela a été utilisé afin de permettre d'avoir **un code ouvert aux extensions mais fermé aux modifications.**

5 – Le patron Singleton

Définition : « *Le Pattern Singleton garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.* » (Designs Pattern – La tête la première).
Ce patron de conception est utile afin de garantir l'unicité de certains objets, par exemple le curseur. Il présente l'intérêt principal de donner accès à un objet depuis n'importe quel endroit du code, ce qui permet de faciliter certains développements.
Cependant, une limite importante existe avec ce patron. En effet, il introduit une notion de mode "super-utilisateur", car ce patron donne accès à certaines ressources du système depuis n'importe quelle partie du code ce qui peut créer des problèmes lorsqu'un code doit être fermé aux modifications.

6 – Le patron État

Définition : « *Le pattern État permet à un objet de modifier son comportement quand son état interne change. Tout se passera comme si l'objet changeait de classe.* »
Ce patron permet de représenter facilement les machines à états. Il permet donc de spécifier un comportement suivant un état de l'objet. Par exemple, pour les sections, on peut distinguer deux états : dépliés et pliés. L'intérêt du patron état est d'éviter la multiplication des instructions conditionnels (imbrication de if) dont les actions seront, en réalité, délégué aux états.
Cela permet de déléguer l'implémentation du ISection.toString aux deux états qui vont se charger de retourner soit "Titre...", si la section est pliée; soit la représentation complète si la section est dépliée.

7 – Le patron Visiteur

Le patron visiteur a pour premier objectif de permettre un ajout ou une modification de fonctionnalités à un ensemble composite d'objets. Cet ajout de fonctionnalités ne modfiie à aucun moment la structure de l'objet. De plus, les opérations disponibles des différents objets sont centralisés et donc plus facile à maintenir
Dans le cadre de notre projet, nos objets visités sont les lignes, textes, sections, documents et cursor. Nos visiteurs sont l'ensemble des classes implémentant les commandes (copy, paste, up, down …) héritant de l'interface ICommandVisitor.
Ainsi, si une nouvelle commande veut être ajouté à notre Editeur, il suffira de créer une nouvelle classe héritant de IComandVisitor et d'implémenter les méthodes visites souhaitées (selon le comportement et l'objet voulu).
On peut se demander comment, selon la commande tapée par l'utilisateur, les visiteurs sont appelés. Pour rendre transparent l'appel des visiteurs, le choix de l'introspection a été fait et vous est décrit dans le paragraphe suivant.

8 – Utilisation de l'introspection

Afin de pouvoir permettre l'ajout d'une nouvelle commande ainsi que le re-nommage des commandes existantes, nous avons décidé d'utiliser deux choses :

1. Le fichier de configuration : Commands.xml
   Ce fichier permet de spécifier tous les raccourcis ainsi que les classes d'implémentations correspondantes. Le format est le suivant :
   ```
   <CommandsConfiguration>
      <config key="\n" value="model.classes.commands.Up" />
      <config key="\s" value="model.classes.commands.Down" />
   </CommandsConfiguration>
   ```
   Ce fichier est composé de balises "config" qui sont celles chargées par le parser XML. Cette balise "config" a un attribut "key" qui correspond au raccourci correspondant et l'attribut "value" qui contient le chemin complet vers la classe correspondante. Par exemple, si la "value" est "model.classes.commands.Up" alors la class Up sera instanciée.
2. L'utilisation de l'introspection (reflection)
   Après le chargement du fichier, le logiciel utilise l'introspection pour charger dynamiquement les classes spécifiées. Cela permet donc à l'utilisateur de rajouter

des classes externes qui implémenteraient aussi l'interface ICommandVisitor (en les rajoutant dans le path) mais aussi de renommer toutes les commandes afin d'avoir des raccourcis plus adéquates.

9 – Utilisation des hiérarchies d'interfaces

L'utilisation d'interfaces et de hiérarchies d'interfaces permet de structurer aisément un code et de répartir les développements entre plusieurs développeurs ce qui est utile dans un contexte de travail distribué.

10 – SRP & loi de démeter

Dans ce projet, nous avons tenté d'appliquer le principe de responsabilité unique qui permet de faciliter les développements. En effet, si une classe n'a qu'une seule raison de change , son débuggage n'est que plus aisé. C'est pourquoi, nous avons essayé, au moins au début, de respecter ce principe.
D'autre part, nous avons essayé de mettre en application la loi/le principe de démeter qui est une procédure de développement visant à simplifier le débuggage (comme toujours). L'objectif est de forcer le développeur à limiter les appels du genre : a.getX().getY().getZ().size(); et d'appeler plutôt dans chaque méthode une méthode qui délègue pour obtenir plutôt un appel ressemblant à: a.getSizeOfZ();.

III - Commandes

Diverses commandes ont été réservées (par la création de fichier .java associés), cependant seules les commandes implémentées et fonctionnelles vont être présentées ci-dessous :

- “\n” : déplace le curseur au nord (vers le haut)
- “\s” : déplace le curseur au sud (vers le bas)
- “\o” : déplace le curseur vers l’ouest (vers la gauche)
- “\e” : déplace le curseur vers l’est (vers la droite)
- “\p” : pour plier une section créée
- “*” : pour créer des sections de rang 1

IV - Utilisation

L’utilisation du logiciel est simple car seul la barre de saisie des commandes permet d'interagir avec le logiciel. Dans cette ligne de saisie, l’utilisateur peut saisir soit une commande interne au logiciel soit du texte qui sera dans ce cas là inséré. Afin d’obtenir une interface graphique avec un curseur, nous avons décidé d’utiliser un JLabel car le JLabel présente l’intérêt majeur d’interpréter le HTML. Par conséquent, le curseur peut être représenté par un arrière plan de couleur derrière la lettre courante. Cela permet donc d’obtenir l’interface présentée ci-dessous :