IDEC

Projet de fin de 1ère année

Générateur de templates LaTeX

Thierry Raeber

25 juin 2018

Table des matières

Première partie

Manuel utilisateur

1 Fenêtre de gestion principale

Au lancement du programme, celui-ci se présente comme illustré par la figure 1.

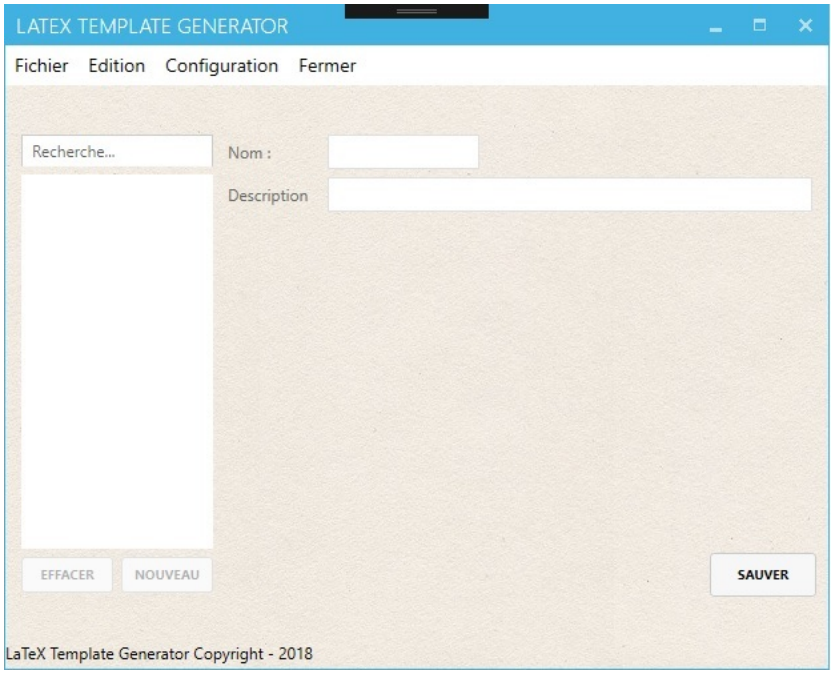

FIGURE 1 – Fenêtre principale (vide)

Cette fenêtre présente la structure commune à toutes les fenêtres de gestion, que ce soit la gestion des templates, des macros, des environnements ou des meta-packages. Comme rien n'est sélectionné, tout est grisé.

1.1 Gestion des templates

Le menu "Edition" permet de choisir le type d'élément à gérer. Ainsi, la fenêtre de gestion des templates est illustré par la figure 2.

On constate d'une part que la liste de gauche a été populée avec les templates présents dans la base de donnée. De plus, en plus des champs "Nom" et "Description" déjà présents sur l'interface, un certain nombre d'éléments ont été rajoutés. Un template doit en effet également avoir un "type" (article, book, letter, etc). On lui attribue également une langue qui permet de définir l'hyphénation, certains caractères spéciaux, les règles typographiques spécifiques à

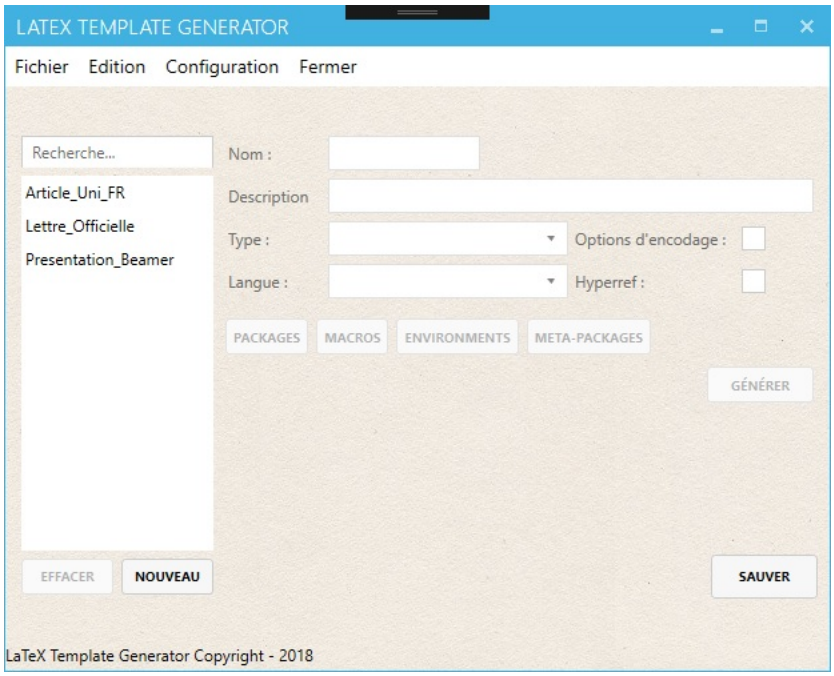

FIGURE 2 – Fenêtre de gestion des templates (vide)

une langue donnée, comme la forme des guillemets, etc. On a ensuite le choix d'ajouter les options d'encodage pour formatter son document en UTF-8 (cette option est activée par défaut), et finalement, d'ajouter ou non le package *hyperref*. Ce dernier sert à gérer les liens hypertexte. Or il a la particularité de devoir être toujours placé en dernier, après tous les autres packages. Cette gestion indépendante des autres packages rendait son intégration plus aisée.

On observe également qu'un ligne de boutons "Packages", "Macros", "Environements", "Meta-packages" est apparue. Cette ligne reviendra dans chacune des fenêtre de gestion pour les autres entités, et permet la gestion des *dépendances*. Nous y revenons au chapitre 2.

Finalement, un bouton "Générer" est également présent, qui servira à présenter le code final, que nous décrivons au chapitre 3

A noter que tant qu'aucun template n'est sélectionné, presque tout est grisé. Seul le bouton "Nouveau" (et le bouton "Sauver" mais ça doit être corrigé) est disponible, car il ne nécessite pas qu'un élément soit sélectionné. Une fois qu'on a sélectionné un template, tout se déverrouille, comme le montre la figure 3.

1.2 Gestion des macros

La fenêtre des macros est très similaire avec la figure 4.

Elle ne possède comme champ propre que le champ "Code", qui permet de définir le contenu de la macro dans le document final.

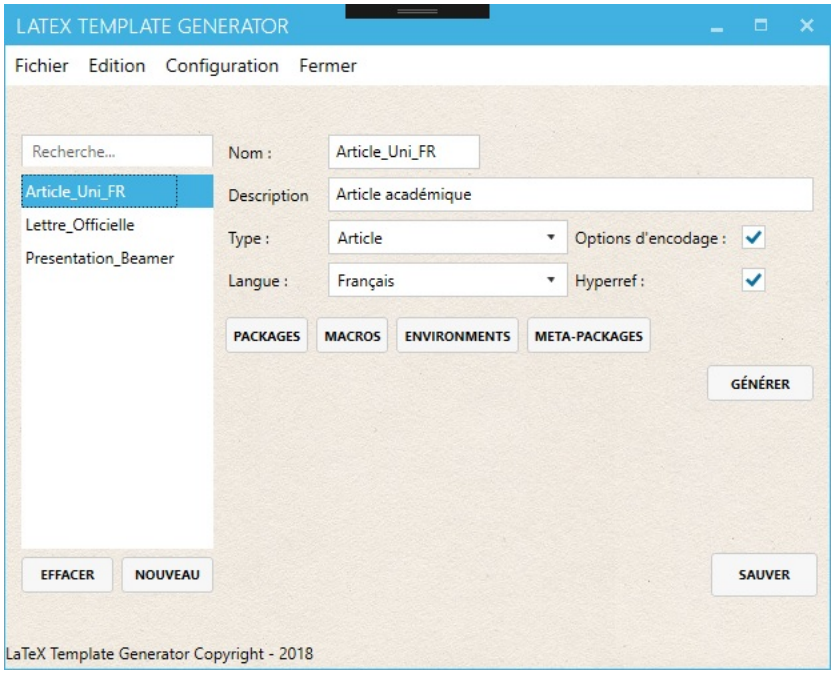

FIGURE 3 – Fenêtre de gestion des templates

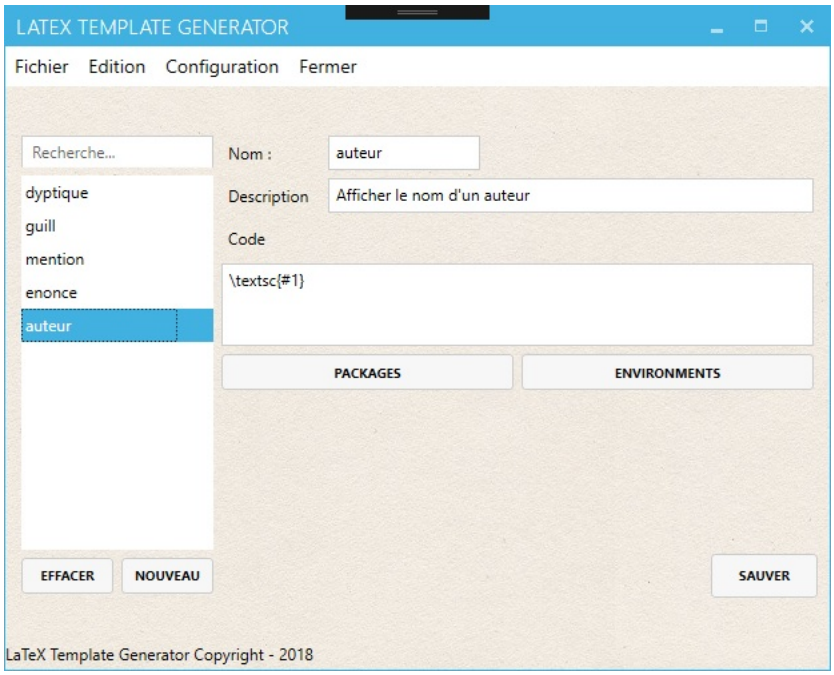

FIGURE 4 – Fenêtre de gestion des macros

Elle n'intègre par ailleurs que les boutons "Package" et "Environement", les autres n'étant pas nécessaires. En effet, selon ce design, une macro ne peut avoir comme dépendance une autre macro, ou un méta-package.

A noter que le bouton "Générer" a également disparu, car il ne peut être associé qu'avec un template.

1.3 Gestion des environements

Très similaire à la fenêtre de gestion des macros, celle des environnements est illustrée par la figure 5.

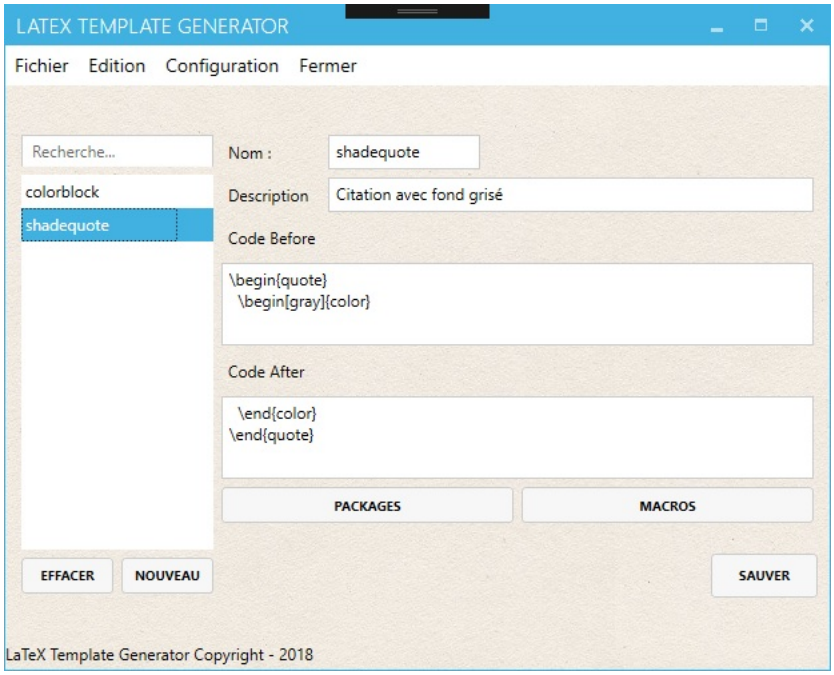

FIGURE 5 – Fenêtre de gestion des environements.

On retrouve la plupart des champs précédents. Toutefois, un environnement est défini par le code initial, et le code final. Le reste est très similaire à ce qu'on a vu jusque là.

1.4 Gestion des méta-packages

La gestion des méta-packages est encore plus simple, comme le montre la figure 6.

Puisqu'un méta-package ne possède aucun code propre, mais n'est qu'un regroupement d'éléments sous une même fonctionnalité, il n'implémente qu'une gestion des dépendances pour tous les types d'objets : packages, macros, environnements et méta-packages (un méta-package pour appeler un autre méta-package).

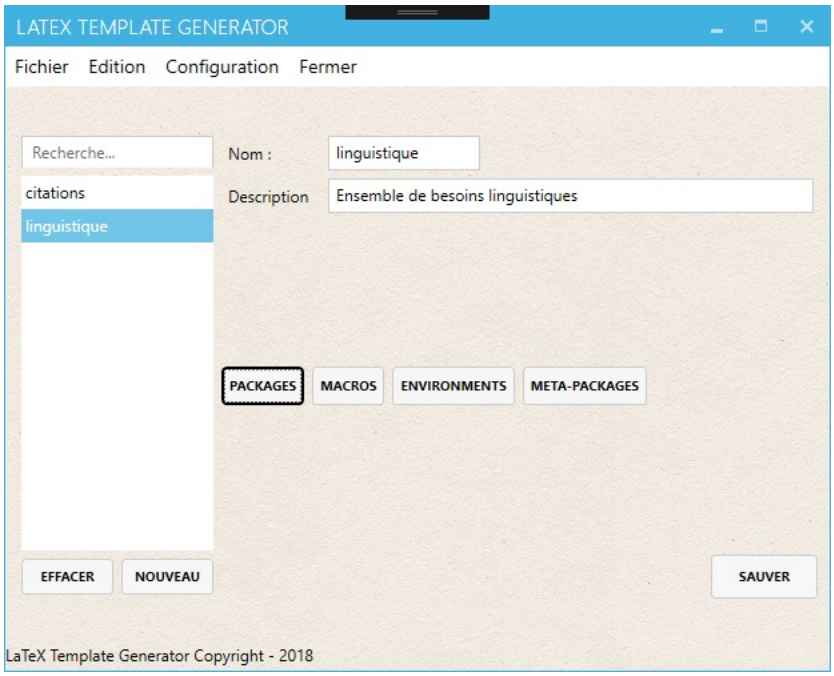

FIGURE 6 – Fenêtre de génération du code

2 Fenêtre de gestion des dépendances

Nous allons expliquer ici ce qui apparaît lorsqu'on clique sur les boutons de gestion des dépendances. En effet, une entité (un template, une macro, etc) peut nécessité l'intégration d'autres éléments pour être fonctionnelle. Chaque fenêtre de gestion offre donc les boutons permettant d'ouvrir la boite de gestion des dépendances. Celle-ci fonctionne de manière identique pour tous les type d'objets, nous n'en décriront donc qu'une seule. Sa forme est illustrée par la figure 7.

La liste de gauche présente l'ensemble des éléments disponibles (ici, les ∼10'000 packages), dont on peut réduire le nombre grâce au champ de recherche. La liste de droite présente les éléments inclus dans la relation de dépendance avec l'item préalablement sélectionné dans la fenêtre parente.

Sélectionner un item dans la colonne de gauche et cliquer "Ajouter" permet de le placer dans la colonne de droite. Sélectionner un item dans la colonne de droite et cliquer "Retirer" permet de l'enlever. Pour éviter les problèmes, sélectionner un item dans une colonne dé-sélectionne automatiquement l'item de l'autre colonne.

3 Fenêtre de code généré

La fenêtre de génération du code LaTeX final est représenté par la figure 8.

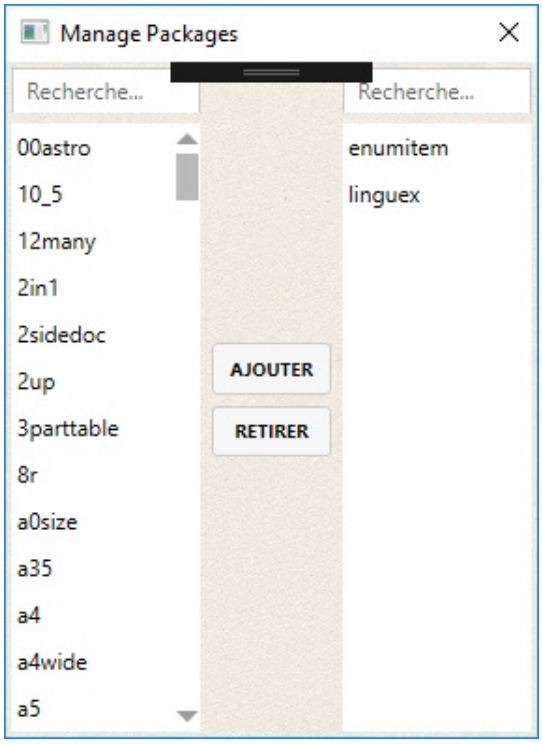

FIGURE 7 – Fenêtre de gestion des dépendances

FIGURE 8 – Fenêtre de génération du code

Cette fenêtre contient une textbox non éditable contenant l'ensemble du code généré. Un bouton "Copier" permet de copier le contenu du code dans le presse-papiers. Un bouton "Sauver" permet de sauver le code directement dans un fichier .tex, et un bouton "Fermer" permet de fermer la fenêtre.

4 Fenêtre de gestion des packages

La fenêtre de gestion des packages se présente comme illustré par la figure 9.

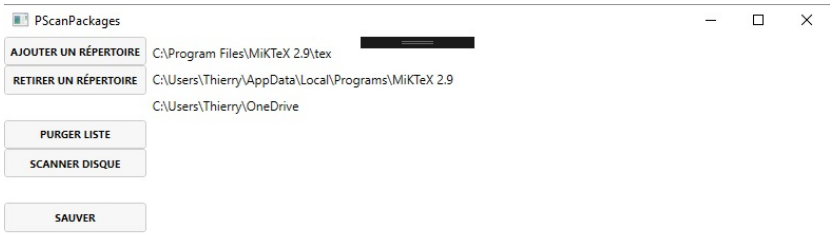

FIGURE 9 – Fenêtre de gestion des packages

Cette fenêtre offre la possibilité d'ajouter / retirer des dossiers à scanner. Elle offre aussi la possibilité de scanner les dossiers intégrés à la recherche de nouveaux fichiers .sty. La fonction "Purge" n'est pas encore implémentée.

Deuxième partie

Cahier des charges

1 Introduction

LaTeX est à la fois un langage et un système de composition de documents. En substance, LaTeX est à un document écrit ce qu'un ensemble de plans est à un bâtiment. L'utilisateur écrit le code source dans un fichier .tex, et ce dernier est lu par un compilateur qui transforme ce code source en fichier PDF. Le présent cahier des charges est d'ailleurs rédigé en LaTeX. Le but de ce projet est de réaliser une application permettant de générer et gérer des templates de documents LaTeX.

2 Contexte

Tout document LaTeX est structuré en deux parties distinctes : le préambule, et le corps. Le corps forme le *contenu* du document, et est délimité par les balises `\begin{document}` et `\end{document}`. Le préambule est l'ensemble des instructions qui définissent ce à quoi devra ressembler et se comporter le code présent dans le code. Si le contenu d'un document change à chaque fois, le préambule est souvent le même d'un type de document à l'autre, et sa gestion peut être fastidieuse si elle doit être faite manuellement. Ce logiciel a pour objectif de faciliter la gestion du préambule d'un document à l'autre.

Plus précisément, un préambule LaTeX est constitué, en simplifié, de 3 composants différents :

1. les packages
2. les macros personnalisées
3. les environnements personnalisés

Un package est un ajout aux fonctionnalités de bases offertes par LaTeX, un peu comme un add-on pour Firefox. Il existe des packages quasi systématiquement ajoutés, comme le package *hyperref* qui gère les liens hypertextes, ou le package *babel* qui gère les spécificités liées à la langue dans laquelle le document est rédigé (caractères spéciaux, hyphénation, forme des guillemets, etc). Il existe également des packages beaucoup plus spécialisés, comme le package *linguex* qui permet de gérer la numérotation des exemples dans les textes de linguistique.

Les macros et les environnements sont à voir comme des fonctions en programmation. Chaque macro ou fonction est invoquée par son nom, et exécute le code qui lui est associé. La différence est que la macro a la forme `\<nom-de-macro>` , alors que l'environnement apparaît comme une balise qui s'exécute sur le texte inclus :

```
\begin{<nom_de_l'environement>}
Texte sur lequel s'exécutera le code de l'environement.
\end{<nom_de_l'environement>}
```

Le but de ce logiciel est de faciliter la gestion de ces inclusions.

3 Objectif

Le but de ce programme est de simplifier et automatiser la réalisation de documents LaTeX. En particulier, l'objectif est d'aider à générer un préambule en indiquant quels packages et macros sélectionner.

4 Fonctionnalités métiers

4.1 Gestion des templates

Un template est un modèle de document type. Ça peut être un article scientifique, une lettre officielle, un diaporama, etc. Or comme nous l'avons vu, le préambule d'un document LaTeX est constitué d'un nombre potentiellement important de packages, de macros et d'environnements. La fonctionalité principale de ce logiciel sera d'assister l'intégration de ces différents éléments au sein du préambule. Voici donc la liste des fonctionalités liées à la gestion des templates, par ordre de priorité d'implémentation :

4.1.1 Définir un type de document

Un document LaTeX est en priorité défini par son type (article, book, letter, etc). Ceci impactera fortement sur le rendu de mise en page. Il est donc nécessaire qu'une option définisse ce paramètre. Celui-ci prendra la forme d'une combobox, car les possibilités de type sont limitées, et fixées d'avance dans le programme.

4.1.2 Un bouton "Exporter"

Ce bouton appellera le constructeur du rendu final. Une page s'ouvrira, dans laquelle apparaît le préambule complet, compilable tel quel (en théorie). Le texte sera sélectionnable, mais en lecture seule. L'option "Enregistrer" sauvera le texte dans un fichier au format .tex. Un bouton "Copier dans le presse-papiers" sera peut-être également présent.

4.1.3 Un bouton "Sauver"

Ce bouton enregistrera les modifications apportées à template dans la base de données.

4.1.4 Choisir les packages, macros et environnements

Le programme donnera la possibilité de déterminer les différents packages, macros et environnements désirés. Pour ce faire, un bouton par type sera présent sur l'interface. Chacun d'eux ouvrira une nouvelle fenêtre avec à gauche, la liste des éléments disponibles, et à droite, la liste des éléments prévus pour intégration.

4.1.5 Choisir la langue du document

Une option pourra être ajoutée pour définir la langue d'édition du document.

4.1.6 Intégrer le package *hyperref*

Une option spécialement dédiée au package hyperref est présente, car ce package a la particularité de devoir toujours être ajouté en dernier.

4.2 Gestion des macros

La fenêtre d'édition des macros offrira une liste recherchable contenant toutes les macros déjà existantes dans la base de données. Une fois sélectionnée, la fenêtre affichera le nom de la macro, sa description, ainsi que son code interne.

la fenêtre permettra également d'effacer une macro existante, ou d'en créer de nouvelles. Dans un tel cas, les champs se vident, et l'utilisateur pour les remplir à sa guide.

Les macros peuvent nécessiter la présence de packages préalablement inclus, ou d'autres macros/environnements préalablement définis. Pour cette raison, il sera possible de gérer leurs dépendances de la même manière que pour les templates.

4.3 Gestion des environnement

Étant donné que l'environnement fonctionne de la même manière qu'une macro (seule la syntaxe de son appel change), il reçoit les mêmes fonctionnalités que la macro.

4.4 Gestion des meta-package

Un meta-package est un ensemble de packages, de macros et d'environnements qui sont réunis pour servir un objectif unifié. On peut imaginer un objectif très minimal, comme "écrire en français", qui demandera le package *babel* pour l'hyphénation et les guillemets à chevrons, et le package *fontenc* pour les accents. Rien de plus. Mais on peut imaginer le bien plus imposant meta-package "article scientifique en français", qui, en plus du meta-package pour le français, demandera le meta-package qui gère la bibliographie aux normes APA, un meta-package gérant les indexes, un meta-package pour les entêtes et pieds de page,

un meta-package pour la table des matières, etc. Un meta-package peut être vu comme un UserControl en programmation WPF. C'est une brique formée de briques plus petites, parfois intégré à une brique de plus haut niveau, et qui finit par être intégrée dans un template final.

Un meta-package possède un nom, une description, mais pas de code propre. L'interface de gestion permet de lui attribuer les packages, macros, et environnements désirés, comme vu précédemment.

4.5 Gestion de packages

La gestion des packages est plus complexe. Les packages disponibles sont définis en fonction de l'installation TeXLive faite sur la machine. Il est donc nécessaire de scanner le disque à la recherche des fichiers .sty présents. Une interface a donc été crée pour permettre de

1. Gérer les dossiers à scanner à la recherche de fichiers .sty
2. Une fonction "Scan" qui opère un scan récursif des dossiers concernés, et qui ajoute les éventuels packages qui ne sont pas déjà présent dans la base de donnée
3. Dans le futur, une option "Purge" pourra être implémentée, qui retire tous les packages non sollicités par les templates, et qui permet de remettre la base à zéro pour nettoyer les éventuels packages qui auraient disparus.

5 Données

5.1 Tables dans la DB

Tables principales :
- Package
- Template
- Macro
- Environment
- meta-package
- Langue
- TypeDoc
- ScanDir

A l'exception de la table ScanDir, toutes les autres tables principales possèdent un champ "Nom" qui permet l'identification de l'entité par l'utilisateur. Les tables *Template*, *Package*, *Macro*, *Environment* et *Meta* possèdent toutes un champ "Description" qui permet à l'utilisateur d'insérer une remarque liée à la fonction de l'entité. Certaines tables, comme la table *Macro* et la table *Environment* possèdent des champs permettant de définir le code LaTeX qui constitue leur définition.

5.2 Schéma

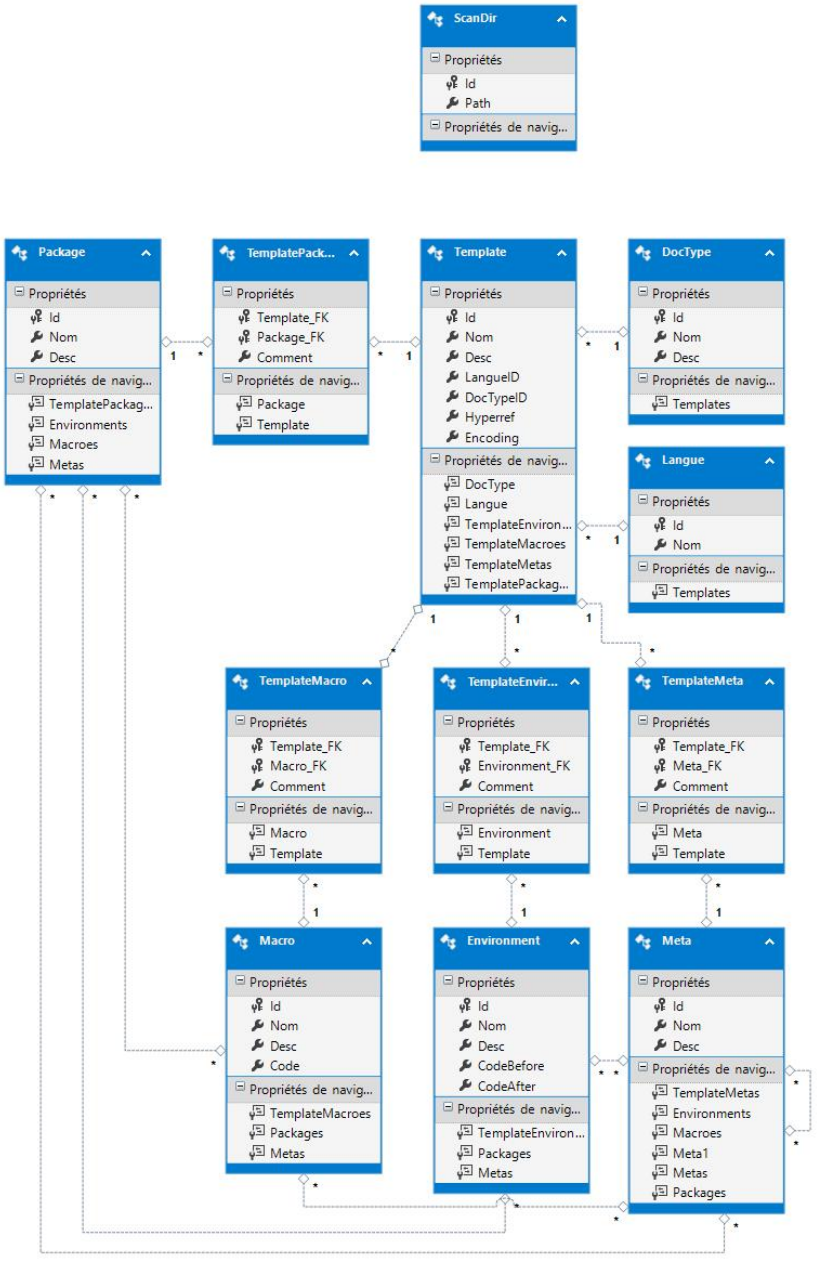

FIGURE 10 – Modèle edmx de la base de données

Troisième partie

Documentation technique

1 Généralités

Le présent logiciel a été réalisé en C#, depuis la plateforme de développement .Net, à l'aide de l'IDE Visual Studio 2017. Il intègre une base de données réalisée avec SQLServer. L'interface graphique est réalisée en WPF.

1.1 Dépendances

Ce logiciel requiert l'installation des plugins NuGets suivants :

- *EntityFramework*, pour la gestion de la base de données
- *MahApps.Metro*, pour l'esthétique des fenêtres WPF.
- *MvvmLight*, qui facilite la communication entre boutons et méthodes du code-behind.
  * Architecture en couches

La couche *Données* est implémentée dans le projet "$\text{LTG}_{\text{Entity}}$". La base de données est externe, mais le plan edmx et l'ensemble des entités complétées se trouve dans ce projet. En particulier, les objects de la base de données qui ont nécessité une classe partielle complétée se trouvent dans le dossier *Entity*. Quelques requêtes SQL utiles ont été sauvées dans le dossier *Queries*.

Les couches *Métier* et *Interface* sont toutes deux implémentées dans le projet "WpfMainView". Toutefois, elles ont été réparties dans des dossiers séparés. Les éléments d'interface sont répartis en deux dossier. Le dossier "Controls" contient tous les *UserControls*, et le dossier "Views" contient toutes les pages et les fenêtres. Concernant la couche *Métier*, les différents ViewModels sont placés dans le dossier du même nom, et les différentes classes responsables de tâches particulières aidant au bon déroulement du programme sont placées dans le dossier "Helpers".

2 Base de données

Le schéma edmx de la base de données se présente comme suit :

Les entités *Template*, *Macro*, *Meta*, *Environment* et *Package* entretiennent entre elles des relations n-n. En effet, un template par exemple peut être associé avec autant de packages qu'on le souhaite, y compris 0. Ceci explique la présence des tables intermédiaires (comme TemplatePackage par exemple). A noter que le schéma edmx ne montre pas les tables intermédiaires lorsque ces tables de contiennent que les clés étrangères (par économie de place). Pour cette raison, la table *MetaPackage* par exemple n'apparaissent pas. Toutefois, les tables liées à l'entité *Template* apparaissent car elles possèdent en propre un champ *commentaire*, qui servira dans une version ultérieure de ce programme.

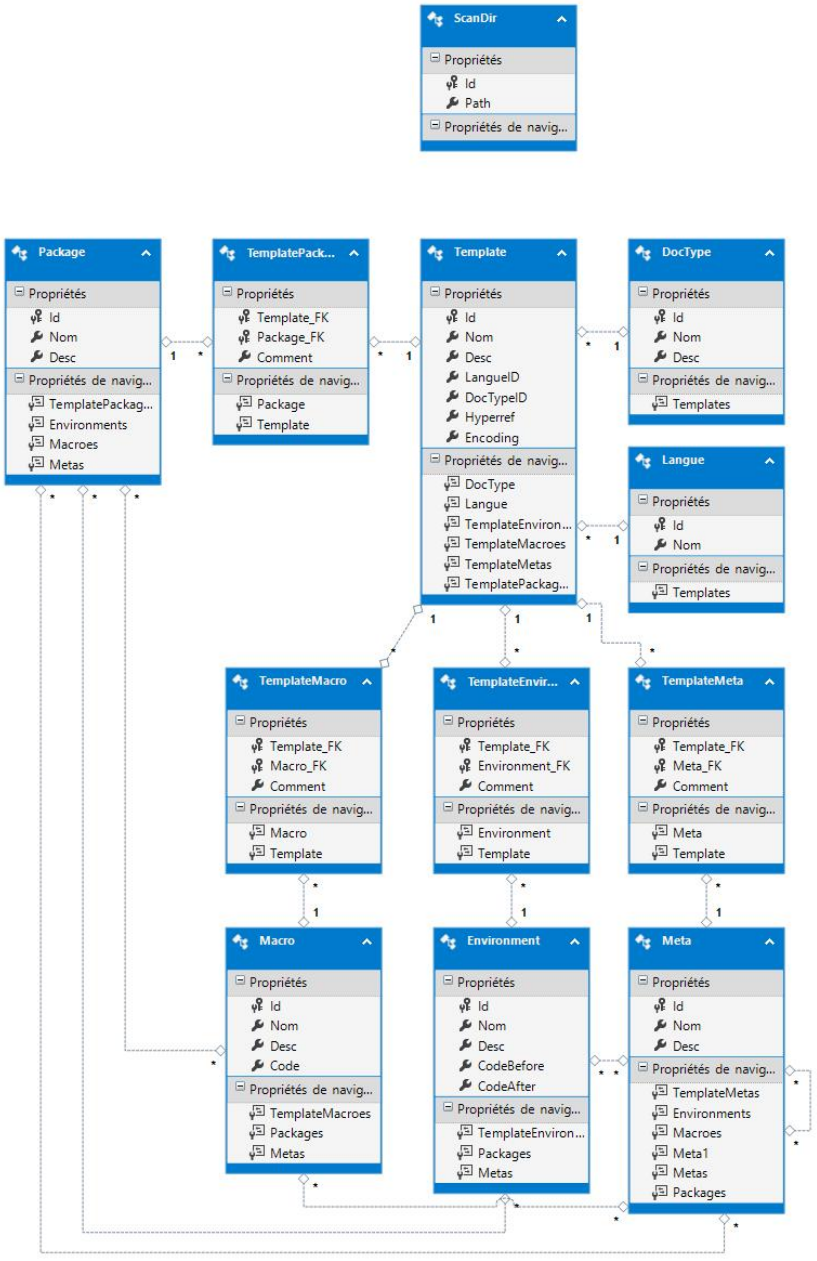

FIGURE 11 – Schéma edmx de la base de données

L'entité *Template* entretient également une relation 1-n avec les entités *Langue* et *DocType* car un template possède obligatoirement 1 et un seul type de document. Mais un type de document peut être associé à plusieurs templates. Pareil pour la langue.

Pour finir, l'entité ScanDir n'est reliée à rien, car elle ne sert qu'à enregistrer les dossier à scanner pour trouver les dossiers .sty.

2.1 Classes liées aux données

Afin de permettre au programme de proprement caster chacune des listes de données à afficher, une classe mère *DBItem* est crée. Celle-ci possède une propriété *Name* qui servira à afficher le nom de l'entité dans la liste.

Chaque entité principale de la base possède un complément de classe partiel permettant d'assurer sa bonne gestion. En particulier, chacune possède une méthode pour créer une nouvelle entité. Cette méthode contrôle la définition des différents champs. Ceci permet entre autre de définir à -1 l'ID d'une nouvelle entité, ce qui permet de repérer chaque entité nouvellement crée, et de l'intégrer proprement à la base de données.

3 Classes utilitaires

Les classes suivantes, localisées dans le dossier "Helpers" du projet WpfMainView, ont été créées afin de réaliser certaines fonctions pratiques et récurrentes. Leur fonctionnalité est détaillée ici.

3.1 La classe Linq()

La classe Linq sert à regrouper sous une même enseigne des méthodes liées à l'accès aux données via des commandes Linq. En particulier, ces méthodes permettent d'obtenir facilement les macros associées à un template particulier par exemple.

Les méthodes *overloaded* de structure `DependenciesList(DBItem, DBItemType)` retournent la liste de dépendances de type *DBItemType* de l'objet *DBItem* passé en paramètre.

Les méthodes de structure `List*(DBItem)` (ListPackages par exemple) retournent également une liste de dépendances de l'objet passé en paramètre, et dont le type est défini par le nom. Contrairement aux méthodes de type DependenciesList, celles-ci retournent une liste de type correspondant au nom de la méthode, et non une liste de DBItems

Les méthodes *overloaded* de structure `RemoveJoin(DBItem, DBItem)` servent à retirer le lien potentiellement existant dans la base de donnée entre les deux entités. Ces méthodes sont en particulier utilisées lors de la destruction d'un objet, pour s'assurer que toutes les connexions intermédiaires sont également détruites.

Étant donné que ces méthodes ne sont là que pour offrir de manière plus aisée accès à certaines procédures Linq, elles sont toutes statiques.

3.2 La classe PackageScan()

Cette classe sert à gérer le scan du disque à la recherches de nouveaux .sty. Pour l'instant, elle ne contient que la méthode `StyFromDir(string Dir)` qui retourne une liste de strings correspondant au nom de tous les fichiers .sty trouvés dans le dossier passé en paramètre. Dans le futur, elle pourrait accueillir d'autres méthodes, comme le retrait de packages non utilisés par exemple, ou la gestion des dossiers eux-mêmes, qui pour l'instant sont gérés ailleurs.

3.3 La classe TemplateContent()

Cette classe stocke et gère toutes les informations nécessaires au formattage d'un template en code LaTeX. Elle n'implémente qu'un seul constructeur, qui reçoit un template en paramètre. Le constructeur fixe certaines informations et appelle ensuite la fonction `ComputeContent()` qui s'occupe de récupérer l'ensemble des packages, macros, environnements et méta-packages passés en dépendance préalablement.

Les méthodes *overloaded* de structure `Add(DBItem)` gèrent l'intégration des informations liées à l'item passé en paramètre. Par exemple, si une macro en paramètre, la fonction `Add(Macro m)` s'assure d'une part que la macro est bien ajoutée à la liste, mais que tous les packages dont elle dépend sont également ajoutés. Pareille pour un environnement. La méthode `Add(Meta m)` s'assure que tous les packages, tous les environnements et toutes les macros associées sont ajoutés, mais également tous les métas-packages qui pourraient y être associés également. Pour éviter tout problème de circularité infinie, un item est tout d'abord ajouté à la liste finale, avant de vérifier ses dépendances. Ceci permet que dans un cas où le méta-package A inclut le méta-package B qui inclut le méta-package C qui lui-même inclut le méta-package A, chacun des méta-packages n'est contrôlé qu'une seule fois.

La méthode `ComputeContent()` appelle donc simplement la méthode `Add()` sur toutes les dépendances directes du template. Ce sont les méthodes `Add()` elles-mêmes qui s'occuperont de retrouver les dépendances récursives.

3.4 La classe TemplateFormatter()

La classe `TemplateFormatter()` sert à générer le code LaTeX final. L'unique constructeur de cette classe prend un objet de type TemplateContent en paramètre. Celui-ci contient la liste exhaustive de tous les éléments associés au template. Il suffit donc de les prendre dans l'ordre, et de les transformer en texte proprement formatté.

Les méthodes de type `ToLatex()` prennent soit un package, soit une macro, soit un environnement, et retournent un string correspondant au code LaTeX de leur

définition / inclusion. Leur contenu est peu intéressant, et consiste simplement à respecter les règles de syntaxe du langage TEX.

La méthode `generate()` appelle les unes après les autres les méthodes nécessaires au bon formattage du document final. A noter simplement que les entêtes de type "Liste des macros" n'apparaissent que si le nombre de macros est non nul. La méthode retourne un string correspondant au code complet du template.

3.5 La classe VMHelper

Cette petite classe contient une énumération qui a été crée pour permettre de déterminer plus facilement de quel type de donnée l'utilisateur est en train de gérer. Elle implémente également une méthode retournant la liste d'items de la base de donnée en fonction du type passé en paramètre.

4 Interface graphique

Les fenêtres se trouvent toutes dans le dossier "Views". Les UserControls se trouvent dans le dossier "Controls".

Une grande partie des UserControls qui ont été développés ne sont finalement pas utilisés. En effet, je ne suis pas parvenu à correctement connecter les données entre les ViewModels et les UserControls pour parvenir à mes fins. Je ne maîtrise pas encore assez la gestion des DataContexts. J'ai donc du me résigner à répéter le code en question, ce qui est totalement non-optimal. Mais ça marche. Désolé.

4.1 Les fenêtres

4.1.1 Fenêtre principale

La fenêtre principale, sur laquelle on arrive au lancement du programme, est implémentée dans le fichier "MainWindow.xaml". Cette fenêtre implémente simplement un menu en haut, une barre d'informations en bas, et un espace pour contenir le UserControl implémenté dans le fichier "UCManager.xaml". Il serait idéalement préférable de fondre le UserControl *UCManager* directement dans la fenêtre *MainWindow*. Toutefois, faire cela requiert de modifier les ViewModels associés, ce qui prendrait trop de temps. J'ai donc décidé de laisser les choses ainsi pour l'instant.

4.1.2 Fenêtre de gestion des dépendances

La fenêtre de gestion des dépendances est implémentée dans le fichier "PManageDependencies.xaml". Elle est constituée de deux listes à gauche et à droite, et de deux boutons au centre.

4.1.3 Fenêtre de gestion des packages

La fenêtre de gestion des packages est implémentée dans le ficher "PScanPackages.xaml". J'avoue qu'elle est assez moche, vu qu'elle a été implémentée tout à la fin.

4.2 Les UserControls

Le UserControl le plus important est *UCManager*, dans le fichier du même nom. Ce UC définit le contrôle principal de gestion des entités de la base de données. Il contient une colonne à gauche pour afficher l'ensemble des entités de la liste sélectionnée, et les champs communs à toutes les entités. Il laisse une grande zone libre en bas à droite pour afficher les particularités de chaque entité. Celles-ci sont représentées par les UC *UCTemplate*, *UCMacro*, *UCEnvironment* et *UCMeta*. En soit, *UCManager* n'est pas bien complexe. Son importance réside dans le fait qu'il accueille le ViewModel principal, *VMmanages*, et qu'il accueille la plupart des autres entités en son sein.

La méthode la plus importante du Code Behind de *UCManager* est `SetManager()`. Cette méthode reçoit un type de DBItem en paramètre, et sur cette base, change le DataContext afin qu'il affiche les bonnes données, et il remplace le UC modulable pour afficher celui du type en cours.