Mémoire : Projet de Programmation

Génération automatique de texte

Benoît Barthés
Alexis Boumera
Baptiste Guiomar
Claire Pennarun
Adrien Wintringer

Bordeaux 1
Projet de Programmation

9 avril 2013

Table des matières

Chapitre 1

Présentation du sujet

1.1 Présentation du domaine

Depuis l'apparition des ordinateurs dans les années 50, la question de la reproduction du langage naturel a toujours été une des plus épineuses.

En effet, le langage humain comprend une multitude de paramètres qui ne sont pas évidents à mettre en oeuvre en termes de programmation : syntaxe, grammaire, intonation, structuration de la pensée... Mais à quoi pourrait bien servir un ordinateur possédant le langage, puisqu'il ne possède pas la pensée, l'intention ; à quoi sert-il de savoir *parler* si l'on ne sait rien *dire* ?

La question est donc de savoir quoi faire dire à un logiciel, quel texte lui faire générer, et de quelle manière.

La génération automatique de texte est surtout un domaine où la machine doit savoir faire beaucoup de choix : choix du sujet à traiter, choix du vocabulaire à employer, choix de l'articulation des phrases... Tous ces choix doivent donc être guidés par le linguiste ou l'informaticien responsable du projet.

Les données analysées par le logiciel peuvent être de plusieurs types : tableaux de données financières (comme dans le système EasyText [3]), données statistiques ou météorologiques... Le traitement et l'analyse de données contenues dans une base de données est, par contre, un sujet qui a été très peu, voire pas traité dans le domaine de la génération de texte, sûrement en raison de la difficulté qu'il y a à extirper le sens de telles données.

Quant à la manière utilisée dans la génération de texte elle-même, elle peut également fortement varier : elle peut faire intervenir (ou non) un utilisateur externe, notamment pour l'étape de sélection du contenu pertinent dans les données initiales.

1.2 Présentation du projet

Le but du projet est de réaliser un logiciel permettant de générer du texte de manière automatique à partir de données brutes sélectionnées dans une base de données.

Pour se faire, nous utilisons le logiciel Syntox, qui est un logiciel développé par M. Lionel Clément, et qui génère du texte grammaticalement et syntaxiquement correct à partir d'une entrée d'une forme spécifique (dite *entrée Syntox* dans notre rapport). Syntox contient à l'avance une grammaire et un lexique qui correspondent au thème traité, ces deux composants étant fournis par M.Clément.

Le but de notre logiciel est donc de fournir à Syntox une entrée qui est ensuite transformée en texte.

Ce logiciel peut servir par exemple à analyser automatiquement des données contenues dans une base de données, ou à utiliser ces données intelligemment pour effectuer des tâches répétitives, notamment l'écriture de textes demandant une grande adaptation des termes suivant les situations (on peut penser au mailing automatique pour des commerciaux ou pour une administration, qui doivent adapter leur message à tous les types de destinataires).

La base de données considérée dans ce projet est une base de données relationnelle, contenant des informations intéressantes pour l'utilisateur, mais non exploitables directement ; il s'agit de données chiffrées, de fonctions, de relations, qui doivent être étudiées avant d'être interprétées.

La génération du texte se fait à partir de *scénarios utilisateur* : chaque scénario correspond à un but rhétorique visé par l'utilisateur.

Notre logiciel fait donc intervenir un utilisateur, qui peut construire le scénario voulu à partir de *concepts* : ceux-ci peuvent être une action, un événement, une succession, un objet, une personne... Le concept est complètement indépendant de la manière de le représenter (notamment indépendant du lexique utilisé, et même de la langue).

Les concepts dits "simples" ou atomiques, sont en général les personnes, ou les objets (ce qui correspond souvent aux noms propres ou aux noms communs). Dans l'exemple ci-dessous, les concepts "match" et "équipe" sont des concepts simples.

Par exemple, le concept "Gagner_Match" peut être représenté ainsi :

$$\begin{bmatrix} \text{Gagner\_match} \\ \text{arg1} = \text{équipe} \\ \text{arg2} = \text{match} \end{bmatrix}$$

Les arguments d'un concept peuvent être eux-mêmes des concepts : cela permet de fabriquer des concepts plus complexes. Par exemple, le concept de causalité prend en argument deux concepts, qui sont complexes :

$$\begin{bmatrix} \text{cause} \\ \text{arg1} = \text{antécédent} \\ \text{arg2} = \text{conséquence} \end{bmatrix}$$

Dans notre projet, le logiciel propose à l'utilisateur de choisir entre la création d'un nouveau scénario (et l'utilisateur devra alors choisir les concepts qu'il veut exprimer, ainsi que leurs arguments) et la réutilisation d'un scénario déjà établi et sauvegardé.

Le scénario (c'est-à-dire l'ensemble de concepts plus ou moins complexes que l'utilisateur veut exprimer) est ensuite traité en plusieurs étapes par le logiciel pour générer l'entrée Syntox souhaitée.

Premièrement, le scénario est transformé en un *graphe conceptuel*, c'est-à-dire un graphe acyclique orienté dont les sommets sont des concepts et les arêtes orientées correspondent aux liens entre ces concepts.

Exemple de graphe conceptuel :

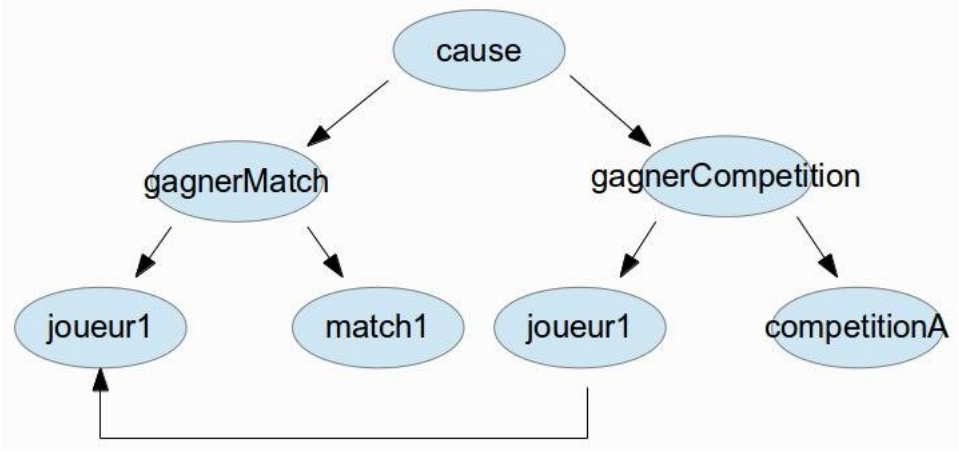


FIGURE 1.1 – La victoire du joueur1 dans le match1 entraîne sa victoire de la compétition A.

Ce graphe est ensuite parcouru pour le retranscrire en entrée Syntox.

Par exemple, le graphe ci-dessus provoque la génération de l'entrée suivante :

```
S Axiom[PRED=cause,
   arg1=[PRED=gagnerMatch,
        tense=present,
        subj=[PRED=joueur1, number=sg, def=+],
        obj=[PRED=match1, number=sg, def=+]],
   arg2 =[PRED=remporterCompetition,
        tense=present,
```

```
subj=[PRED=joueur1, number=sg, def=+],
obj=[PRED=competition, number = sg, def=+]]
```

Chaque entrée permet à Syntox de générer toutes les phrases correspondant aux contraintes qui y sont spécifiées.

Dans notre exemple, Syntox génère alors les phrases :

1. le joueur1 gagne le match1 , par conséquent le joueur1 remporte la compétition .
2. le joueur1 gagne le match1 . le joueur1 remporte la compétition .
3. le joueur1 gagne le match1 , donc le joueur1 remporte la compétition .

On récupère ces phrases, qui correspondent au texte voulu par l'utilisateur. Celui-ci peut ensuite enregistrer le résultat dans un fichier.

Notre logiciel comporte deux types d'utilisateurs : l'utilisateur dit "classique", qui ne se préoccupe pas de la manière dont le texte est généré, et l'utilisateur administrateur, qui peut configurer le système.

Ainsi, l'administrateur peut configurer le système (créer les concepts requis et donner les informations de connexion à la base de données), puis l'utilisateur classique n'a plus qu'à utiliser le logiciel, sans avoir besoin de connaître la base de données utilisée ou les détails des concepts qu'il manipule.

Chapitre 2

Etude de l'existant et bibliographie

2.1 Eléments bibliographiques

L'état de l'art [1] présenté par John Bateman en 1993 (et mis à jour régulièrement depuis) concernant la génération automatique de texte est une base de connaissances très importante pour nous. En effet, l'auteur reprend les avancées des travaux les plus importants dans ce domaine, enrichi d'une grande bibliographie. Il donne également des méthodologies précises pour traiter des problèmes de la génération automatique de texte. Cet état de l'art, assez conséquent, nous permettra de trouver des références rapidement, même si le sujet que nous traitons n'a pas encore fait l'objet d'une étude poussée.

La synthèse du domaine faite par Laurence Danlos ([6]) est d'un grand intêret pour notre projet. En effet il nous présente de façon très concise ce qu'est la génération automatique de texte. Le premier point intéressant concerne les questions qui se posent généralement dans les applications mettant en oeuvre de la génération de texte, "Quoi dire ?" et "Comment le dire ?". Ce seront probablement des questions que nous aurons à nous poser par la suite. Le document met aussi en avant les différents problèmes et difficultés rencontrés par les sytèmes existants et auxquels nous pourrions avoir à faire face, mais aussi les avantages de tels systèmes. Il est par exemple question de leur "spécialisation" à certains domaines (comme la météo, les finances etc) qui leur permet de ne couvrir qu'une partie de la grammaire et du lexique et d'être ainsi relativement performants. Nous sont aussi présentés dans ce document les différents types d'applications déjà existantes, comme TA(O) (première application développée) ou encore PostGraphe (utilisé dans le domaine statistique). Plus intéressant encore, ce document nous décrit ce que peuvent ou pouraient apporter ces systèmes, comme par exemple une sortie orale (si nous couplions un générateur à une synthétiseur vocal). Une présentation précise des mécanismes et tâches mis en oeuvre dans la génération automatique nous est aussi offerte, avec pour chaque étape une présentation des entrées/sorties. Le document nous présente enfin le problème de l'évaluation de tels systèmes, qui s'avère difficile à résoudre

de par leur complexité.

2.1.1 EasyText

Le système de génération de textes EasyText [3] a été développé par Frédéric Meunier, Laurance Danlos, et Vanessa Combet dans le cadre d'une collaboration avec la filliale française Kantar Media de TNS-Sofres. Ce logiciel permet de créer un commentaire à partir d'un tableau de chiffres.

Contrairement à d'autres procédés de génération de textes tel que les textes à trous ou le publipostage, EasyText utilise un formalisme G-TAG. Ce dernier permet "de transformer une représentation conceptuelle d'un ensemble d'informations en un texte" (d'après Laurence Danlos, [4]).

Plusieurs étapes ont lieu afin de réaliser le commentaire. Elles se schématisent par deux questions principales : "Quoi dire ?" et "Comment le dire ?" . Dans une première étape il faut sélectionner les informations que l'on peut dire : ce choix se fait en fonction des éléments intéressants présents dans le tableau. On obtient alors une conjonction de formules logiques, qui est ensuite transformée en un texte, en deux temps.

On cherche en premier lieu à transformer la conjonction en phrases (macro-planification). On s'appuie alors sur des connaissances rhétoriques et linguistiques de la langue pour structurer l'ordre du texte et déterminer ainsi le "schéma" de la phrase. Enfin, on complète la phrase pour la former selon les conventions de la langue (micro-planification). Pour cela, en fonction du "schéma" auquel elle se rapporte, on ajoute des déterminants, des compléments, etc en prêtant attention à la position à laquelle ils se situent. On s'appuie ici sur des connaissances sémantiques et syntaxiques. À la suite de toutes ses étapes on obtient un texte formé selon les critères de la langue qui a pour objet les éléments du tableau.

2.2 Etude de l'existant

2.2.1 Projet GAT-2

Le logiciel GAT-2 (Génération Automatique de Texte 2) [5] est le prédécesseur de notre objectif de travail actuel. Développé l'année dernière par 5 étudiants de master 1 Informatique, il repose principalement sur GAT-1 (développé lui aussi par des étudiants de master 1 Informatique) et sa version améliorée Syntox développée par Lionel Clément : ces deux outils ne se chargent pas du "quoi dire" mais du "comment le dire", prenant en entrée des règles de grammaires énoncées de manière précise et générant un texte fini et syntaxiquement correct.

GAT-2, dans sa version finale, traite simplement une requête SQL énoncée par l'utilisateur (ou plutôt choisie dans une liste prédéfinie par les concepteurs), et y répond

via une phrase rédigée en français. Pour ce faire, il se connecte à la base de données, et traite via des opérations logiques simples la requête afin d'en extraire et d'en déduire plusieurs résultats, qui sont formatés de façon à pouvoir être interprétés par GAT-1 qui se charge d'écrire les données répondant à la requête initiale de l'utilisateur en plein texte. Cependant, il n'arrive pas pleinement à extraire le "sens" (au sens le plus strict du terme) de la base de données, se limitant à la réponse d'une seule question à la fois de manière assez redondante.

2.2.2 Syntox

Syntox [2] est un logiciel développé par Lionel Clément, chercheur au LaBRI dans l'équipe Méthodes Formelles.

Il est défini par son auteur comme un synthétiseur de texte. C'est un type de programme plutôt récent, il en existe donc peu et sont pour la plupart le fait d'universitaires.

Syntox permet de générer du texte à partir de concepts. En d'autres termes il ne vérifie pas qu'une phrase est correcte dans un langage donné, mais il crée une phrase correcte dans une grammaire et un lexique défini à partir d'idées.

Syntox ne se limite donc pas à la génération d'une unique phrase, et peut générer des paragraphes entiers. Sa limite principale se trouve dans les reprises sémantiques comme les anaphores, ou les ellipses. A l'heure actuelle, le projet représente deux ans de travail.

Son interface utilisateur est accessible via un navigateur web pour permettre de porter les efforts sur le programme en lui même et de ne pas perdre du temps à créer des exécutables pour différentes plateformes.

Chapitre 3

Cahier des charges

3.1 Scénarios d'utilisation

3.1.1 Scénarios utilisateur

Scénario 1

1. L'utilisateur se connecte en choisissant le mode "Utilisateur" ne requérant pas de mot de passe.
2. Il sélectionne le projet sur lequel il souhaite travailler, et entre le mot de passe de la base de données correspondante (si il y en a un).
3. Il choisit "Scénario existant" et dans la liste déroulante il choisit celui qui lui convient en fonction de la base de données active.
4. Il clique sur le bouton "Valider" pour lancer l'exécution du logiciel.
5. L'utilisateur peut alors voir le résultat généré par le logiciel.
6. L'utilisateur enregistre le résultat dans un fichier texte en appuyant sur le bouton "Sauvegarder".

Scénario 2

1. L'utilisateur se connecte en choisissant le mode "Utilisateur" ne requérant pas de mot de passe.
2. Il choisit "Scénario personnalisé" pour pouvoir commencer la création d'un scénario.
3. Dans l'écran de création d'un nouveau scénario, l'utilisateur peut choisir de rajouter ou de modifier les concepts qui composeront son scénario.
4. Il clique sur le bouton "Générer le texte" dans le menu "Fichier" pour lancer l'exécution du logiciel.

5. L'utilisateur trouve que le texte généré ne contient pas assez d'informations, ou des informations non pertinentes, il retourne à l'étape de choix des concepts et les modifie pour affiner le résultat en fonction de ce qu'il recherche.
6. Lorsque le résultat lui convient, l'utilisateur sauvegarde le scénario.
7. Enfin l'utilisateur enregistre le résultat dans un fichier texte.

3.1.2 Scénarios administrateur

Scénario 1

1. L'administrateur choisit de se connecter en mode "Administrateur" et doit rentrer un login et un mot de passe.
2. Il choisit de créer un nouveau projet.
3. Il peut alors entrer le nom du projet, la base de données qui souhaite employer et les identifiants associés.
4. L'administrateur peut alors commencer la création du projet en choisissant les différents concepts qui seront liés à la base de données.
5. L'administrateur enregistre son projet.

Scénario 2

1. L'administrateur choisit de se connecter en mode "Administrateur" et doit rentrer un login et un mot de passe.
2. Il choisit de modifier un projet.
3. Il choisit le nom du projet à modifier.
4. L'administrateur s'attelle à l'association de requêtes SQL avec des concepts nouveaux qu'il crée.
5. L'administrateur enregistre le projet.

3.2 Besoins fonctionnels

3.2.1 Besoins utilisateur

L'utilisateur classique doit pouvoir effectuer plusieurs actions de manière assez intuitive. Nous proposons donc une interface graphique pour accéder aux diverses fonctionnalités.

- Dans un premier temps, l'utilisateur peut sélectionner la base de données sur laquelle il souhaite travailler.
- L'utilisateur doit pouvoir créer un nouveau scénario ou charger un scénario préexistant.

– Lors de la création d'un nouveau scénario, les différents concepts sont sélectionnés et une vue globale du scénario est proposée à l'utilisateur.
– La visualisation du texte généré se fait via l'interface graphique.
– Le résultat peut être enregistré dans un fichier .txt dont le nom est choisi par l'utilisateur. Ce fichier lui permet de réutiliser le texte généré à sa guise.

3.2.2 Besoins administrateur

L'administrateur est un utilisateur avancé qui peut configurer les différentes étapes de la génération du texte. Il possède un mode différent de celui de l'utilisateur classique dans l'interface graphique.

Besoins généraux

L'administrateur doit pouvoir :
– Lier les concepts à la base de données (par des requêtes SQL) via une interface graphique ;
– Choisir des concepts pertinents selon la base de données utilisée. L'administrateur va alors lier à chaque concept une ou plusieurs requêtes SQL nécessaire(s) à son expression ;
– Il a les mêmes besoins que l'utilisateur classique, et doit pouvoir accéder aux mêmes informations ;
– Il doit pouvoir visualiser le texte final, afin de pouvoir le vérifier et éventuellement modifier l'étape de liaison concepts/requêtes en cas de non-satisfaction.

Besoins liés à la base de données

D'un point de vue "administration de la base de données", l'administrateur doit pouvoir effectuer les actions suivantes :
– Ajouter des informations de connexion à une base de données ;
– Modifier les configurations de la base de données, ou changer son emplacement, ainsi que lui associer des profils utilisateur (identifiant/mot de passe) ;
– Vérifier le bon fonctionnement et retour des requêtes SQL utilisées ;
– Tester les requêtes directement sur le logiciel afin de vérifier leur bon fonctionnement.

Besoins liés aux concepts et aux scénarios

– *Créer/modifier/visualiser/supprimer des concepts*
Un concept étant un objet comprenant un nom (le nom du concept lui-même) et des arguments d'un nombre et d'un type précis, l'administrateur doit pouvoir entrer de nouveaux concepts dans la liste des concepts. La visualisation des concepts correspond à la visualisation du graphe conceptuel qui lui est associé.

– *Créer des scénarios*
L'administrateur doit pouvoir créer de nouveaux scénarios qui seront proposés à l'utilisateur s'il choisit de charger un scénario pré-enregistré.

Le client (M. Lionel Clément) se charge de la génération du lexique et de la grammaire associée au logiciel Syntox, il nous fournit également les concepts dont nous aurons besoin.

3.2.3 Besoins système

Indépendamment des besoins fonctionnels de l'utilisateur classique et de l'utilisateur administrateur, nous avons des besoins fonctionnels relevant du système. Parmi ces besoins on trouve :

Interface graphique

Pour répondre aux besoins des utilisateurs classiques et administrateurs, l'interface graphique a pour fonction la création de l'entrée Syntox. Cette création du point de vue de l'utilisateur consiste à choisir, par le biais d'éléments graphiques, les concepts qu'il souhaite employer dans le texte final. Il peut choisir les concepts et leurs arguments, qu'ils soient simples ou complexes.

La fenêtre qui permet la création de l'entrée Syntox est commune à l'utilisateur classique et à l'administrateur. Tous deux ont les mêmes besoins pour cet aspect du programme, il n'est donc pas nécessaire de la dissocier.

L'utilisateur administrateur a accès à une fenêtre spécifique, consacrée à la gestion des concepts et des éléments de la base de données. Son rôle en tant qu'administrateur est de relier les concepts à des requêtes.

Génération de requêtes SQL

On doit prévoir un module capable de communiquer avec la base de données au moyen de requêtes SQL : on doit donc être capable de les envoyer, et de recevoir la réponse de la base de données, ainsi que de traiter ces réponses pour pouvoir les utiliser.

Base de données

Nous avons besoin de pouvoir nous connecter à une base de données en lecture afin de pouvoir lire sa structure en lui soumettant des requêtes SQL. Pour le moment nous ne nous limitons pas à l'utilisation d'une seule base de données, afin de pouvoir étudier l'utilisation de différents concepts sur d'autres données.

Nous envisageons de stocker les concepts dans une base de données indépendante de celle où se trouvent les données initiales, pour pouvoir la transporter sans véhiculer l'ensemble des données.

Création du graphe conceptuel

Pour la création du graphe conceptuel, nous avons besoin d'une librairie de graphes : nous devons pouvoir afficher le graphe, le modifier, le stocker en mémoire et le parcourir.

Ce graphe étant un graphe orienté acyclique, le parcourir pour générer l'entrée Syntox sera assez simple (un parcours en profondeur classique, en marquant les sommets déjà explorés).

Création de la requête pour Syntox

Il s'agit ici de transformer le graphe conceptuel obtenu précédemment en entrée Syntox avec sa syntaxe spécifique. On crée dans ce but une table de correspondance entre chaque concept et l'entrée Syntox correspondante, modifiable par l'administrateur.

Génération de phrases

On utilise Syntox afin de générer les phrases à partir de l'entrée créée par notre programme. On doit alors soit intégrer Syntox dans notre programme afin de lui communiquer directement les entrées, et les récupérer pour les stocker dans un fichier ; soit établir un protocole de communication avec la version en ligne de Syntox fonctionnant dans les deux sens : l'envoi de l'entrée et la récupération des phrases générées par Syntox.

3.3 Besoins non-fonctionnels

Afin d'être conforme aux spécifications fournies par le client, le logiciel doit répondre à un certain nombre de besoins non-fonctionnels.

3.3.1 Génération d'une entrée Syntox

- Le but premier du logiciel est de générer une entrée Syntox. Nous devons donc faire en sorte que cette entrée soit de forme compatible avec le logiciel Syntox. Pour ce faire nous devons avoir en mémoire un ensemble de règles qui relieront chaque concept exprimé au formalisme Syntox correspondant.
- L'entrée Syntox que le logiciel génère doit permettre la production d'une phrase grammaticalement correcte par Syntox, ce qui est assuré par une construction soignée et précise de l'entrée.

3.3.2 Facilité d'utilisation

Le logiciel est conçu selon les besoins de notre client. Nous ne savons pas si d'autres personnes l'utiliseront. Nous partons donc avec l'objectif d'avoir un logiciel utilisable

par une personne ayant des connaissances basiques en informatique. Cependant, il est à noter que pour l'administrateur, des connaissances en base de données sont souhaitées.

3.3.3 Portabilité

Nous développons sous Linux et notre client utilise MacOs X, cela nécessite que notre programme soit le plus portable possible. Pour passer le moins de temps possible à compiler notre logiciel pour différentes plateformes ; l'utilisation d'une machine virtuelle est de ce fait nécessaire. Nous utiliserons l'environnement Java qui permet de combiner portabilité et simplicité. L'unique contrainte est Syntox mais il est utilisé via le réseau, donc il ne doit pas être compilé spécifiquement pour la machine de l'utilisateur final.

3.3.4 Besoins relatifs aux bases de données utilisées

Pour les bases de données sur lesquelles on travaillera, on se limite aux bases relationnelles (SQL) car elles sont les plus répandues et permettent donc un travail sur des thèmes plus diversifiés. Cela permet également de disposer de plus d'outils pour les manipuler de par le nombre de librairies disponibles et d'un langage connu et normé, par rapport à des bases de données non SQL (souvent utilisées pour l'entrepôt de données comme les bases de données MongoDB) dont les outils sont moins accessibles. Dans le cas de bases de données non SQL, chaque base de données développe son propre langage, rendant la lecture par le même logiciel de plusieurs de ces bases impossible, ce qui ne les rend pas intéressantes dans le cadre de notre projet.

En conclusion nous nous limitons pour l'instant à une base de données MySQL, car il s'agit des plus répandues. En parallèle, on s'occupera de rendre notre logiciel adaptable au maximum à des bases de données relationnelles autres que MySQL, afin de rendre possible une extension du programme dans le futur.

Concernant le travail sur la base de données, nous nous contentons tout d'abord de connexions publiques (sans mot de passe) pour l'envoi des requêtes. On veut avant tout ne pas avoir à gérer un stockage de mot de passe car cela poserait des problèmes de sécurité (cryptage etc). On doit donc pouvoir effectuer des requêtes sur la base de données en tant que simple utilisateur, car on aura pas la possibilité d'une connexion en tant qu'administrateur.

En ce qui concerne les besoins internes à notre logiciel, il nous semble que l'utilisation d'une base de données est superflue.

Chapitre 4

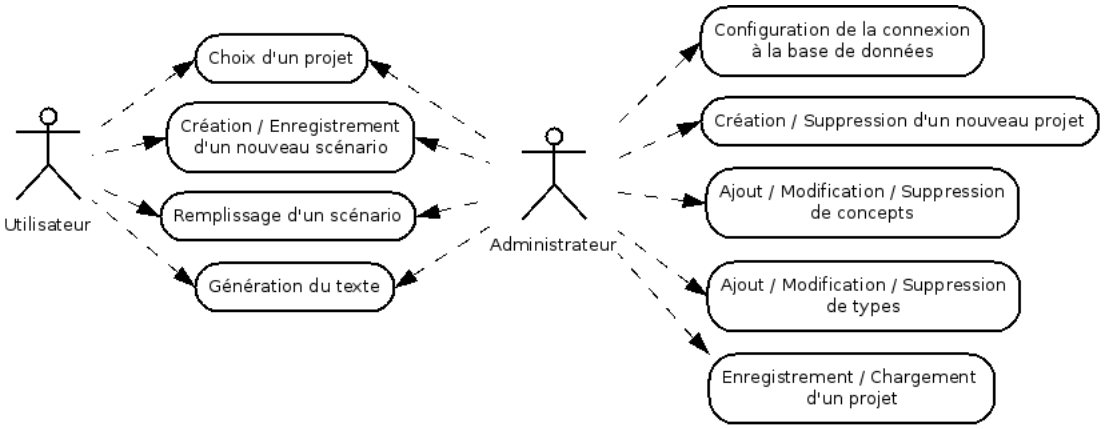
Architecture et implémentation

4.1 Choix du langage de programmation

4.2 Schémas de cas d'utilisation et de fonctionnement

4.2.1 Diagramme de cas d'utilisation

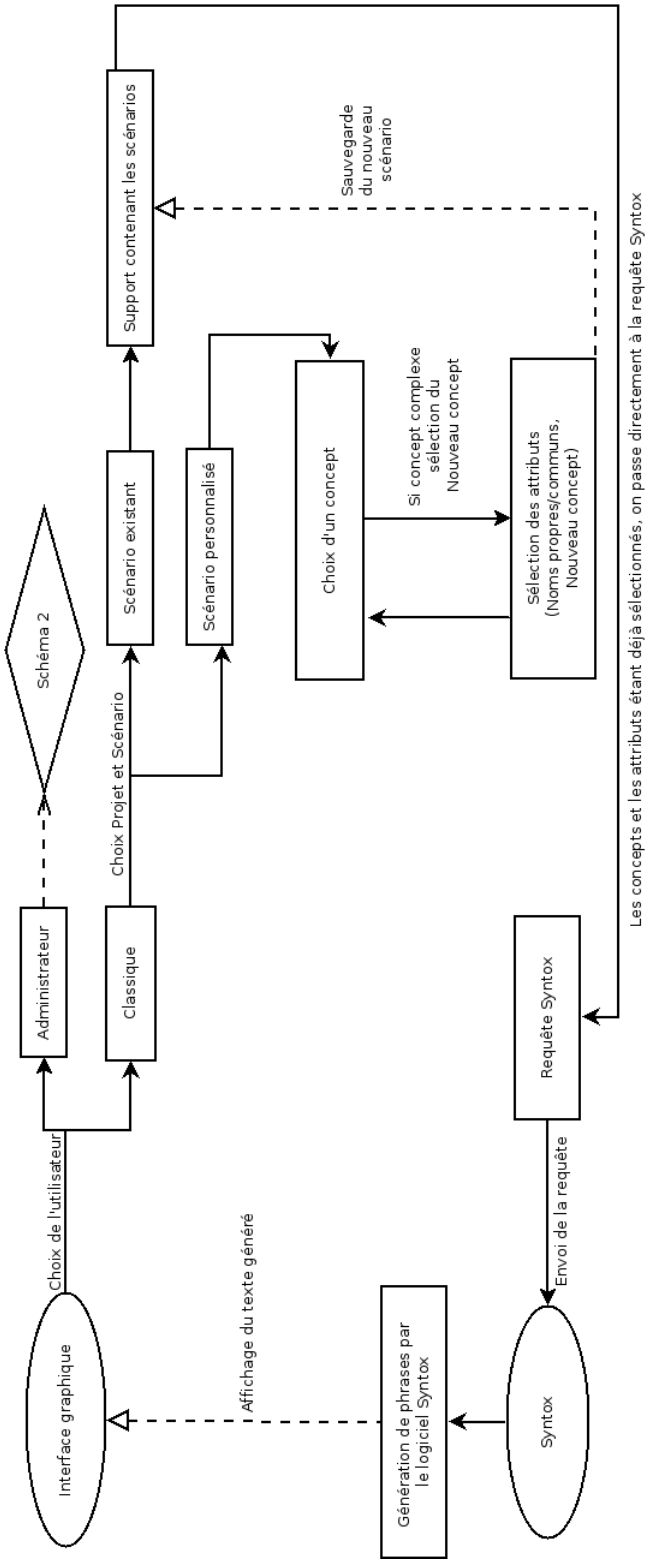
Ce diagramme présente les fonctionnalités accessibles à l'utilisateur classique et à l'administrateur via l'interface graphique.



4.2.2 Schémas de fonctionnement

Ils schématisent les interactions entre le programme, l'utilisateur et l'administrateur, avec la base de données et Syntox.

Schéma de fonctionnement - Utilisateur



Le schéma ci-dessus décrit le fonctionnement du logiciel du point de vue d'un utilisateur classique.

Après avoir choisi le mode d'utilisation "Classique", l'utilisateur choisit le projet sur lequel il souhaite travailler. Il choisit ensuite de travailler sur un scénario existant ou de créer un nouveau scénario.

- S'il choisit "Scénario existant", il a accès à la liste des scénarios existants dans le projet sélectionné et peut générer une entrée Syntox à partir du scénario qu'il aura choisi.
- En choisissant "Scénario personnalisé", il a la possibilité de manipuler des concepts et de former ainsi un scénario adapté à ses besoins. Une fois que son travail est terminé, il peut l'enregistrer. Son nouveau scénario devient disponible dans la liste des scénarios existants.

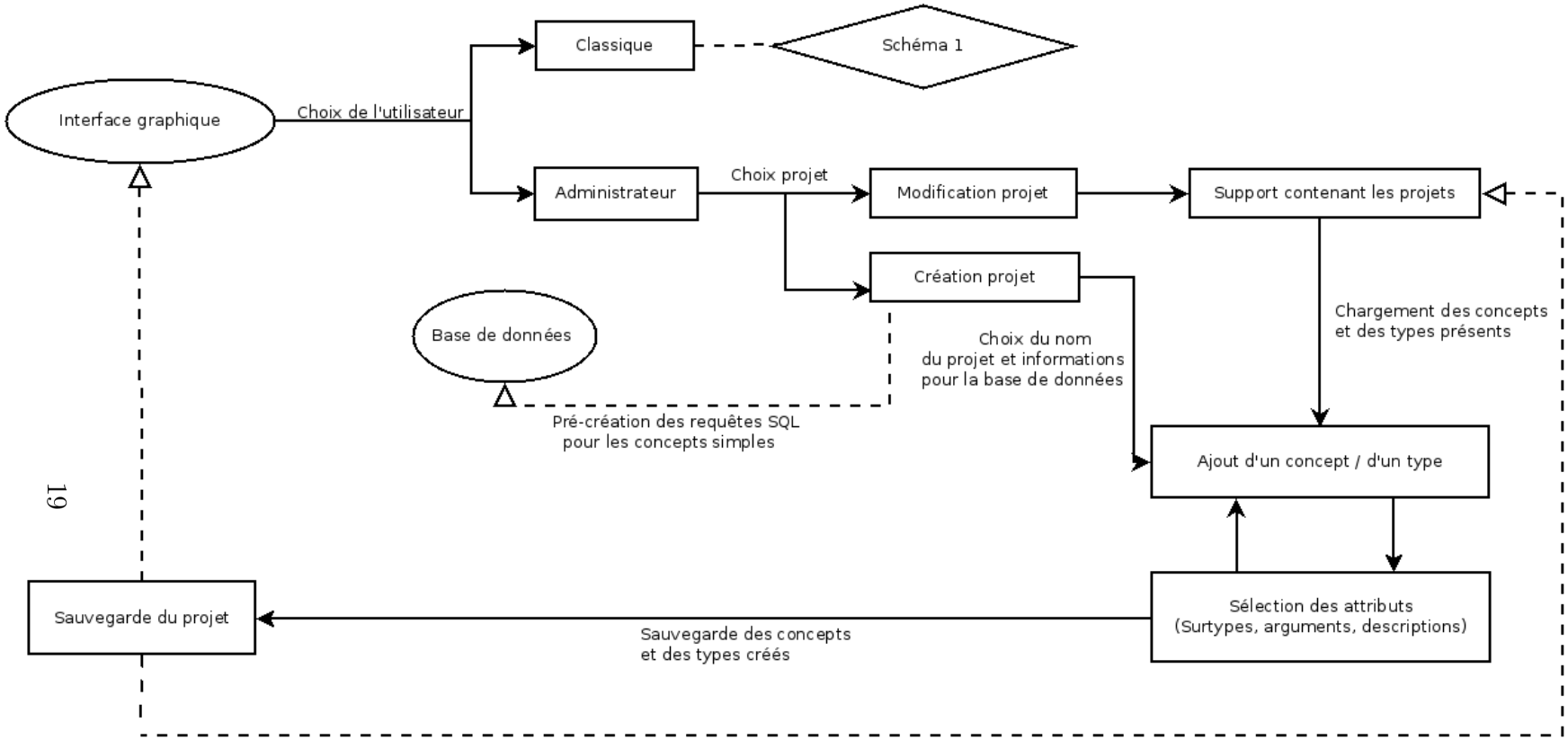
Schéma de fonctionnement - Administrateur

Le schéma ci-dessous décrit le fonctionnement du logiciel du point de vue d'un administrateur.

Après avoir choisi le mode "Administrateur", l'utilisateur administrateur choisit entre la création d'un nouveau projet et la modification d'un projet existant.

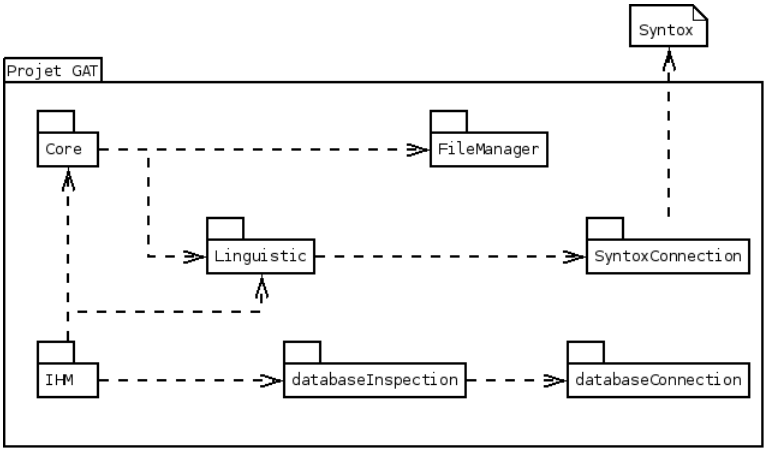
- Dans le cas où il choisit de créer un nouveau projet, il commence par entrer les informations demandées, à savoir : le nom du projet et les informations nécessaires à la connexion à la base de données. Ces informations sont ensuite utilisées pour créer de manière automatique les types de base contenus dans la base de données. L'administrateur peut alors commencer à créer du contenu pour son projet (nouveaux concepts et nouveaux types).
- L'administrateur peut aussi choisir de modifier un des projets existants ; il charge alors le projet voulu, et le complète avec de nouveaux concepts et types.

Quand l'administrateur a fini de modifier / créer son projet, il l'enregistre et pourra y accéder plus tard via la liste des projets existants.


Classique
Schéma 1
Interface graphique
Choix de l'utilisateur
Administrateur
Choix projet
Modification projet
Support contenant les projets
Création projet
Chargement des concepts
et des types présents
Base de données
Choix du nom
du projet et informations
pour la base de données
Pré-création des requêtes SQL
pour les concepts simples
Ajout d'un concept / d'un type
Sauvegarde du projet
Sélection des attributs
(Surtypes, arguments, descriptions)
Sauvegarde des concepts
et des types créés

4.3 Diagramme de packages

Le diagramme suivant présente la structure générale du logiciel.



Notre programme se compose de sept packages : `core`, `databaseInspection` et `databaseConnection`, `linguistic`, `syntoxConnection`, `fileManager` et `IHM`, correspondant aux différentes fonctionnalités du programme.

Le package `core` correspond au coeur du programme et permet le lancement de l'application et l'instanciation initiale de l'interface graphique et des fonctionnalités du package `linguistic`.

Les packages `databaseInspection` et `databaseConnection` s'occupent de la connexion à la base de données et de la récupération des données qui seront utilisées par le logiciel.

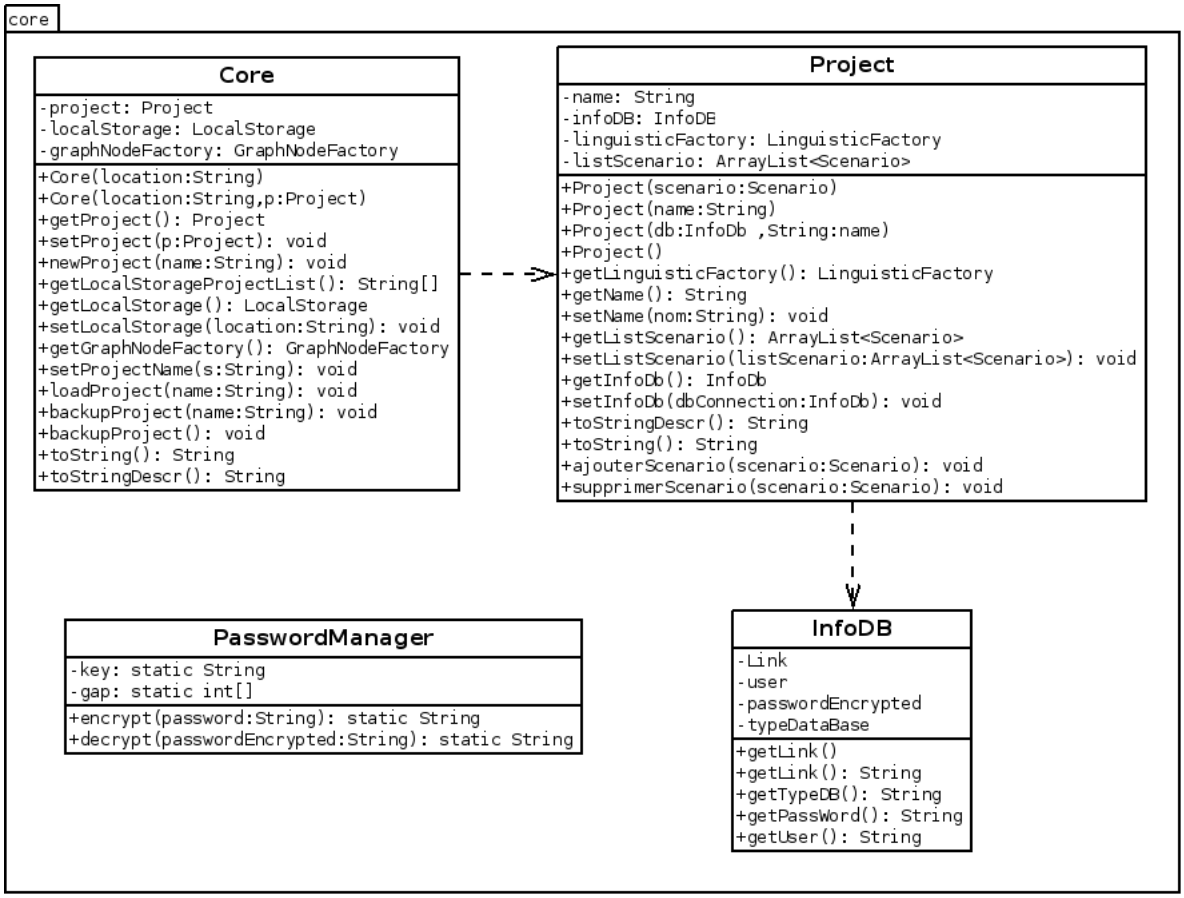
Le package `linguistic` s'occupe de la gestion des concepts et des types, ainsi que du fonctionnement des graphes conceptuels.

Le package `syntoxConnection` gère la connexion au logiciel Syntox.

Le package `fileManager` gère le chargement et l'enregistrement des scénarios et des projets dans des fichiers.

Le package `IHM` est dédié à l'interface graphique.

4.4 Core



Le package `core` est utilisé par notre programme afin de centraliser toutes les données nécessaires à son exécution : gestion de la connexion à la base de données, chargement et enregistrement des projets et lancement du programme.

4.4.1 Implémentation

Le package `core` est composé des cinq classes suivantes :

- Classe `Main` : se charge d'initialiser l'interface graphique et de créer l'objet central "Core" - Classe `Project` : il s'agit de l'objet central que l'on manie dans le programme : il regroupe tous les scénarios, concepts, types, ainsi que les informations de connexion liées à la base de données associée. - Classe `InfoDB` : cette classe implémente un objet gérant la connexion à la base de données (identifiant et mot de passe, ainsi que son URL) - Classe `PasswordManager` : classe qui implémente un mécanisme basique de cryptage / décryptage des mots de passe associés à la base de données, dans l'unique but de ne pas les stocker en clair sur le disque dur. - Classe `Core` : contient les constructeurs et les accesseurs de l'objet Core, qui sera un objet unique lors de l'exécution du programme.

Cet objet contient une instance de `Project`, une instance de `LocalStorage` qui indique le répertoire de travail ou sauvegarde les projets ainsi qu'une référence à une instance de `GraphNodeFactory`.

4.4.2 Accès aux données, sauvegarde et chargement

Lors de l'exécution du programme, un unique objet `Core` est créé au lancement. Cet objet permet de faciliter les communications entre l'implémentation et l'interface graphique, en regroupant toutes les fonctions d'accès et de modifications de l'objet `Project`. Il possède aussi les méthodes de sauvegarde et de chargement des fichiers XML des projets. De plus, il permet de se détacher de l'implémentation des objets Project et donc d'etre moins dépendant de son implémentation. De plus, il permet de ne pas encombre l'objet `Project` d'une `GraphNodeFactory`, le limitant à une structure de données concrète contenant uniquement l'information utilisée par l'utilisateur.

4.5 Base de données

Package `databaseInspection`

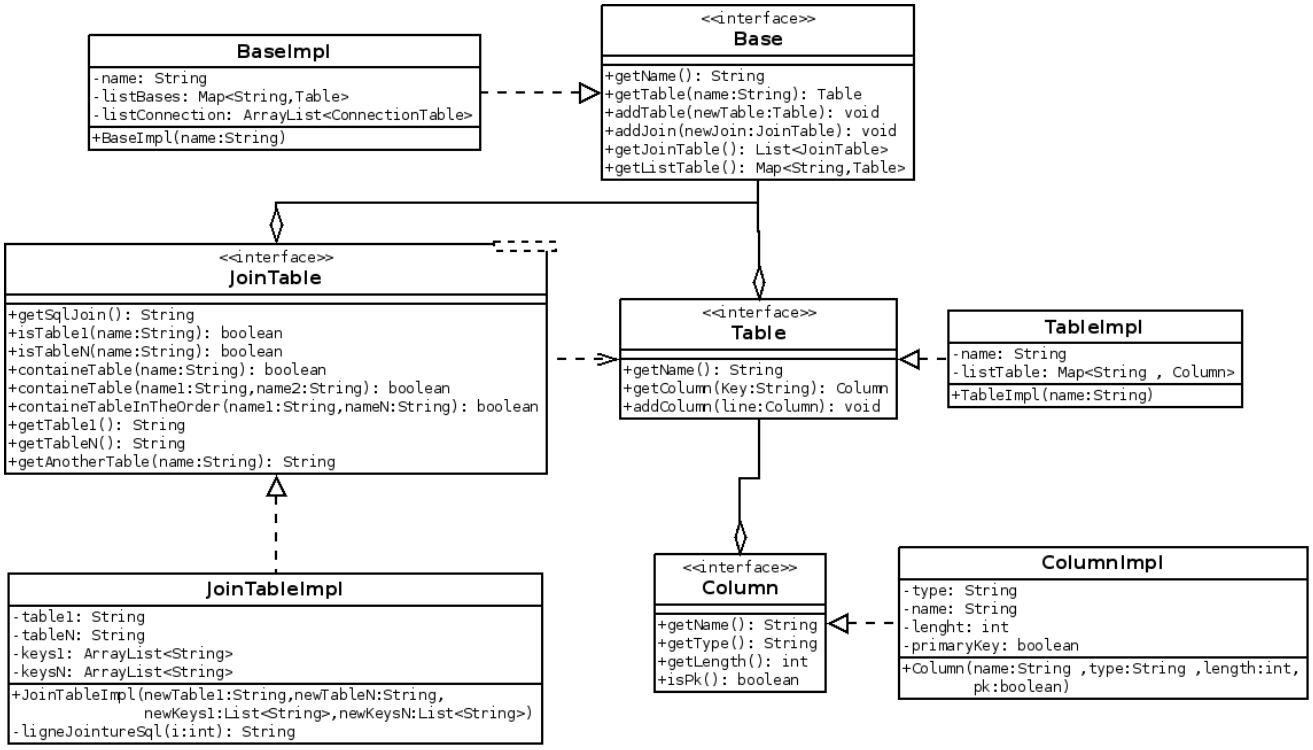
Ce paquet contient les classes nécessaires à la création dynamique de requetes SQL sur une base de données, dans le cadre de la récupération des informations nécessaires à l'expression des concepts.

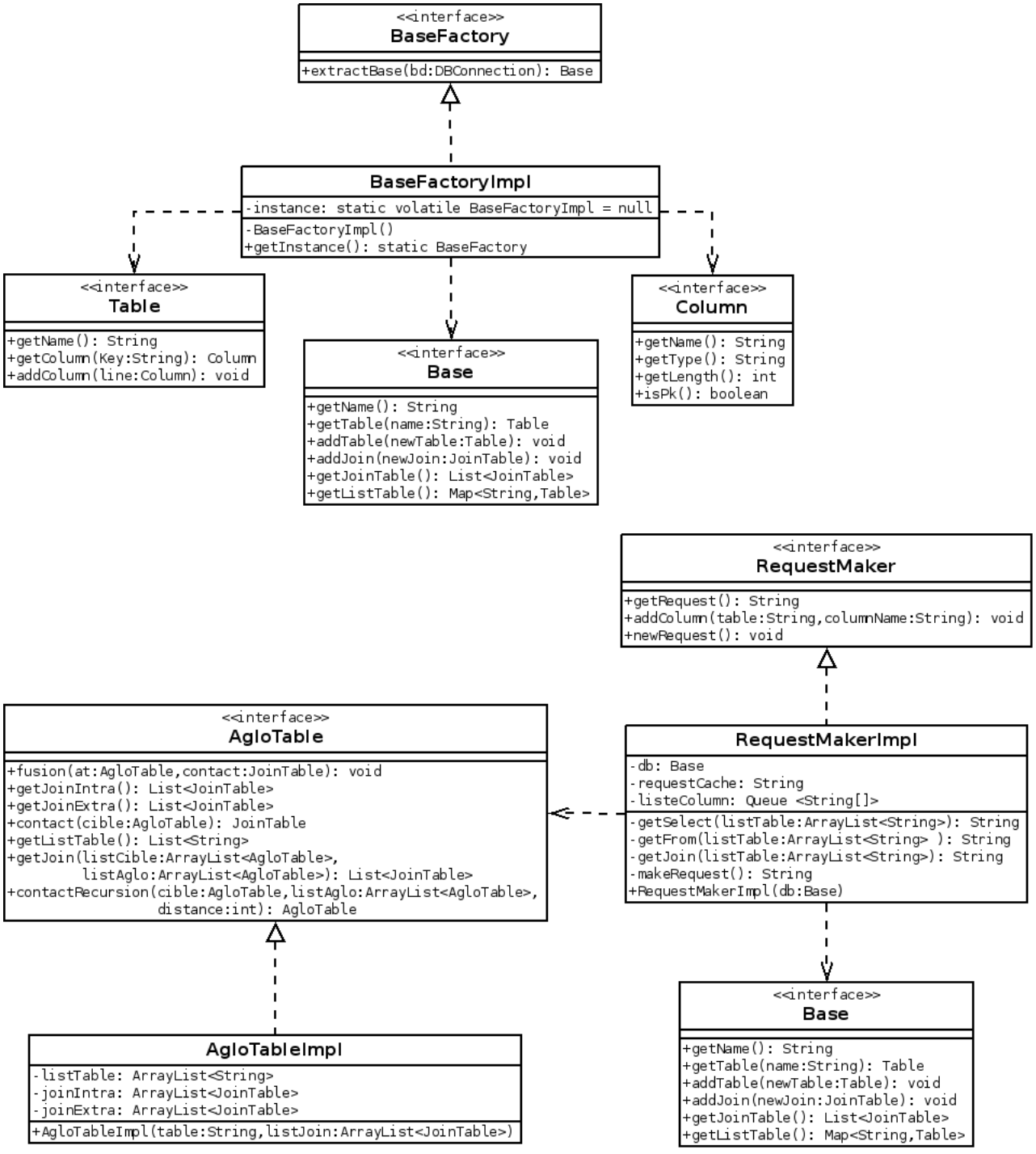
- Interface `Base` : Cette interface est implémentée par `BaseImpl`. Elle contient les tables et leurs relations (jointure).
- Interface `Table` : Cette interface est implémentée par `TableImpl`. Elle représente une table et contient les listes de ses colonnes.
- Interface `Column` : Cette interface est implémentée par `ColumnImpl`. Elle représente une colonne d'une table.
- Interface `JoinTable` : Cette interface est implémentée par `JoinTableImpl`. Elle représente la liaison entre deux tables.
- Interface `BaseFactory` : Cette interface est implémentée par `BaseFactoryImpl`. elle construit un objet `Base` à partir d'une base de données SQL.
- Interface `RequestMaker` : Cette interface est implémentée par `RequestMakerImpl`. Elle construit une requête SQL.
- Interface `AgloTable` : Cette interface est implémentée par `AgloTableImpl`. Elle calcule la jointure minimale entre plusieurs tables.

De manière générale, les gestionnaires de bases de données relationnelles (SGBDR) enregistrant l'information des clés étrangères dans des tables séparées de la base, on a préféré créer un objet JoinTableImpl qui s'occupe de faire les liens entre deux tables via les clés étrangères. La classe `BaseFactory`, à partir de la connexion à la base de données effectuée, se charge de récupérer le schéma de la base de données afin dans l'enregistrer

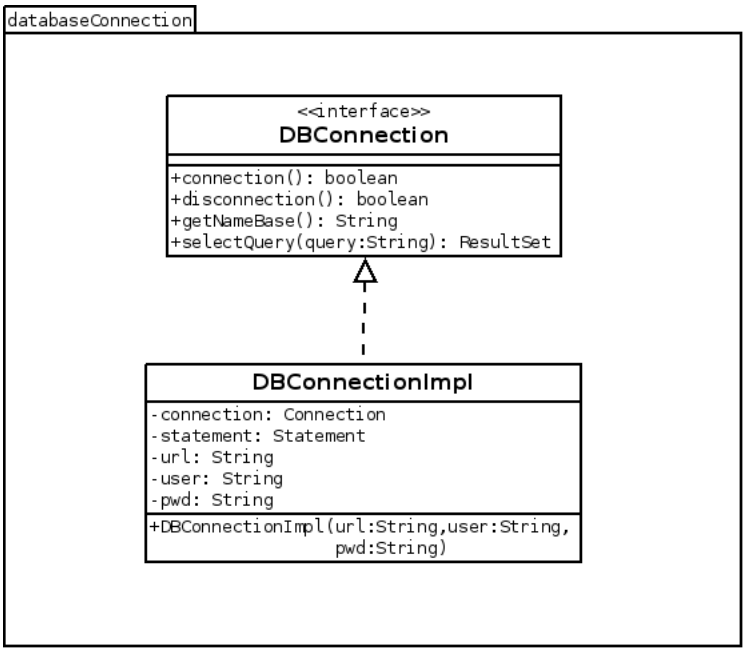
en tant qu'objet utilisable par le programme. L'objet `RequestMaker` permet quant à lui de créer les requêtes SQL nécessaires à l'interrogation de la base de données : à partir des fonctions qu'il implémente, il permet de générer automatiquement des requêtes SQL simples afin de récupérer une certaine colonne d'une table choisie. Il se base sur l'interface `AlgoTable` pour pouvoir générer une jointure minimale.

Perspectives `AlgoTable` ne fonctionne pas comme elle le devrait : il reste une anomalie liée à une copie d'objet par référence qui empêche son exécution.




<<interface>>
BaseFactory
+extractBase(bd:DBConnection): Base
BaseFactoryImpl
-instance: static volatile BaseFactoryImpl = null
-BaseFactoryImpl()
+getInstance(): static BaseFactory
<<interface>>
Table
+getName(): String
+getColumn(Key:String): Column
+addColumn(line:Column): void
<<interface>>
Base
+getName(): String
+getTable(name:String): Table
+addTable(newTable:Table): void
+addJoin(newJoin:JoinTable): void
+getJoinTable(): List<JoinTable>
+getListTable(): Map<String,Table>
<<interface>>
Column
+getName(): String
+getType(): String
+getLength(): int
+isPk(): boolean
<<interface>>
RequestMaker
+getRequest(): String
+addColumn(table:String,columnName:String): void
+newRequest(): void
<<interface>>
AgloTable
+fusion(at:AgloTable,contact:JoinTable): void
+getJoinIntra(): List<JoinTable>
+getJoinExtra(): List<JoinTable>
+contact(cible:AgloTable): JoinTable
+getListTable(): List<String>
+getJoin(listCible:ArrayList<AgloTable>,
listAglo:ArrayList<AgloTable>): List<JoinTable>
+contactRecursion(cible:AgloTable,listAglo:ArrayList<AgloTable>,
distance:int): AgloTable
RequestMakerImpl
-db: Base
-requestCache: String
-listeColumn: Queue <String[]>
-getSelect(listTable:ArrayList<String>): String
-getFrom(listTable:ArrayList<String>): String
-getJoin(listTable:ArrayList<String>): String
-makeRequest(): String
+RequestMakerImpl(db:Base)
AgloTableImpl
-listTable: ArrayList<String>
-joinIntra: ArrayList<JoinTable>
-joinExtra: ArrayList<JoinTable>
+AgloTableImpl(table:String,listJoin:ArrayList<JoinTable>)
<<interface>>
Base
+getName(): String
+getTable(name:String): Table
+addTable(newTable:Table): void
+addJoin(newJoin:JoinTable): void
+getJoinTable(): List<JoinTable>
+getListTable(): Map<String,Table>

Package `databaseConnection`



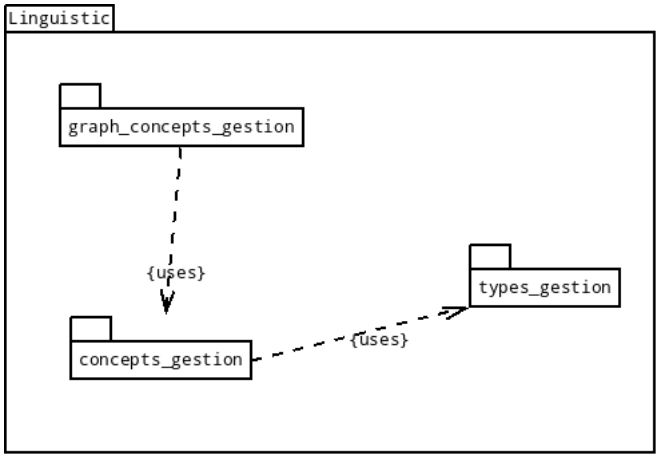
L'interface `DBConnection` est implémentée par la classe `DBConnexionImpl`. Elle contient les outils nécessaires à la connexion aux bases de données.

Il s'agit d'un objet simple permettant de factoriser les codes autour de JDBC (Java DataBase Connectivity). Il permet également de limiter les accès à la base de données à des SELECT (pas de modification ni de la structure ni des informations de la base). Il possède les fonctions de connexion, déconnexion, accès au nom de la base de données utilisée, et exécution de requêtes SQL de type SELECT.

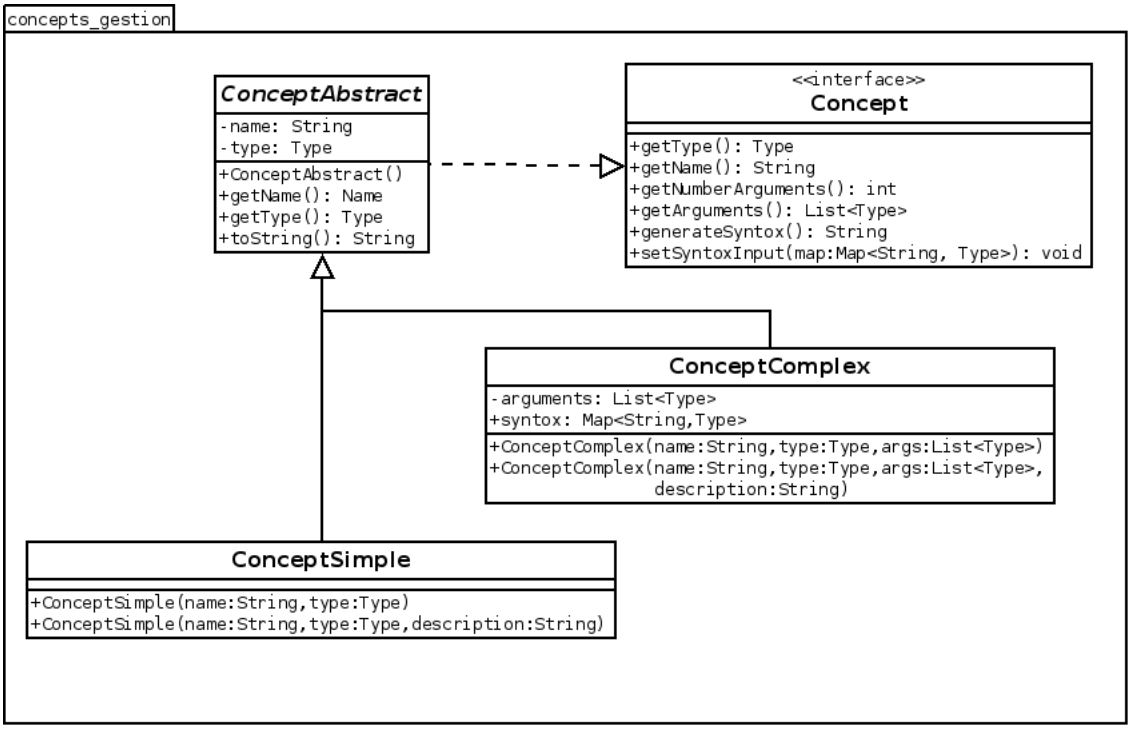
4.6 Package `linguistic`

Ce package comprend toutes les classes nécessaires à la gestion des concepts, la mise en place d'un système de typage et la génération du graphe conceptuel correspondant au scénario généré par l'utilisateur.

Le package `linguistic` a été subdivisé en trois sous-packages : `concepts_gestion`, `types_gestion` et `graph_concepts_gestion`.



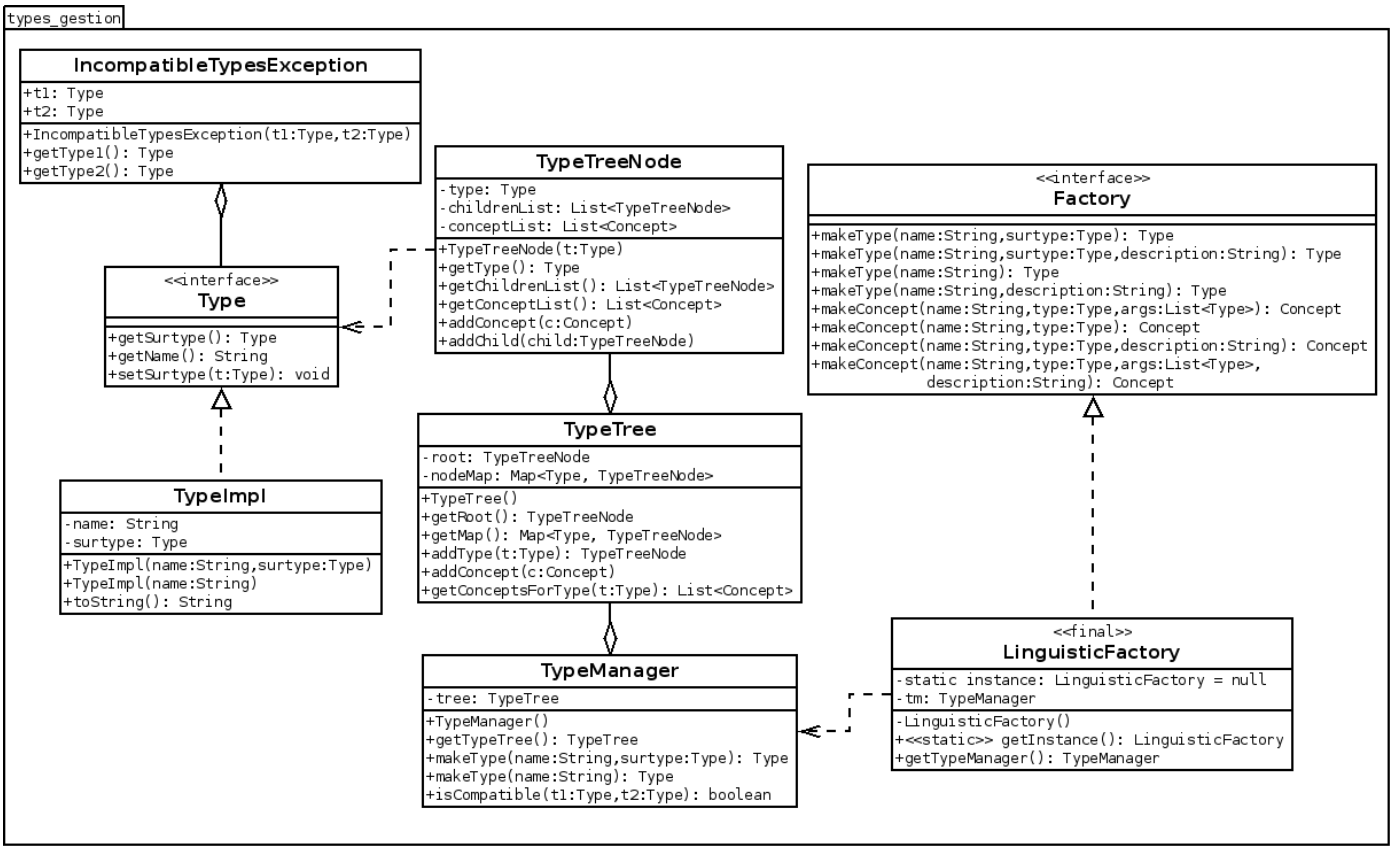
4.6.1 Package `concepts_gestion`



Ce package contient la structure représentant un concept. Chaque `Concept` est défini par (au moins) un nom et un `Type`. Le `Concept` peut être simple ou complexe ; s'il est complexe, il possède également une liste d'arguments (qui sont des types, ce qui permet une indépendance des différents concepts).

La classe `ConceptAbstract` permet de factoriser du code, notamment les accesseurs. La méthode *generateSyntox()* sera analysée dans la partie "Package Syntox".

4.6.2 Package `types_gestion`



Le package `types_gestion` contient les classes gérant le système de typage des `Concept`. Le coeur de ce package est donc bien sûr l'interface `Type` et la classe `TypeImpl` qui l'implémente. Chaque `Type` est défini par son nom et peut être relié à d'autres types par une relation surtype/sous-type.

Les `Type` sont stockés dans un arbre de types, implémenté par la classe `TypeTree`. Cet arbre de types contient une référence à la racine de l'arbre (qui correspond au Type racine), et une `Map` qui permet de faire la correspondance entre un `Type` stocké dans l'arbre et le noeud de l'arbre associé à ce `Type`. Nous avons choisi d'implémenter cette correspondance avec une `Map` car c'est une structure qui permet une indexation par les `Type`, et un accès rapide en lecture.

L'arbre de types `TypeTree` est composé de plusieurs `TypeTreeNode`, chacun correspondant à un `Type` précis et une liste de `Concept` ayant ce `Type`. Chaque `TypeTreeNode` comprend aussi une liste des `TypeTreeNode` qui sont "en-dessous" de lui dans l'arbre de types (et qui sont donc associés à ses "sous-types").

Cette connaissance par chaque `TypeTreeNode` de ses fils permet au `TypeTree` de parcourir les `TypeTreeNode` à partir de la racine, ce qui sert notamment pour l'implémenta-

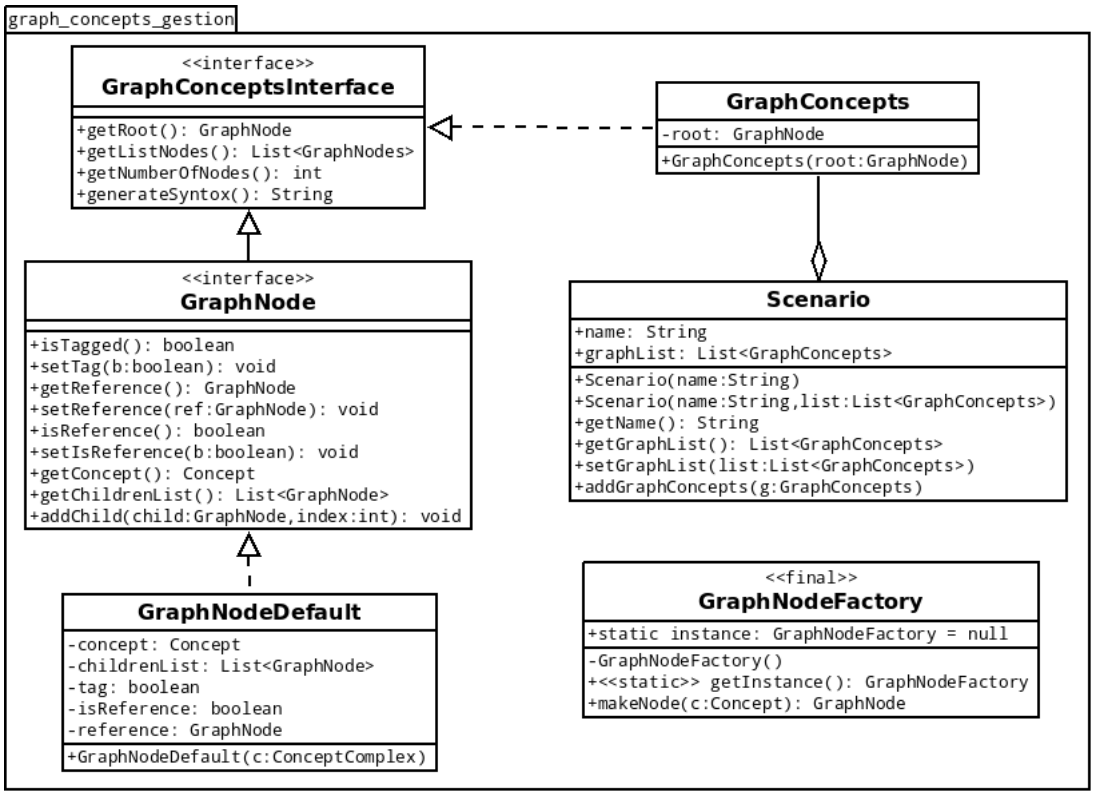
tion de la méthode *getConceptsForType(Type t)*, qui retourne une liste des instances de `Concept` ayant le `Type` t ou un de ses sous-types.

La classe `TypeManager` est une classe qui permet de gérer la classe `TypeTree` et d'instancier la classe `TypeImpl`, elle se base sur le modèle du pattern Factory. En effet, l'instanciation de `TypeImpl` est faite par le `TypeManager`, qui crée et modifie dynamiquement l'arbre de types correspondant. Il est aussi responsable de la compatibilité de deux `Type` *t1* et *t2*, c'est-à-dire la vérification que le `Type` *t1* est un sous-type (strict ou non) du `Type` *t2*.

L'arbre de types et donc la classe `TypeManager` ne doivent avoir qu'une seule instance par `Projet`, nous avons donc mis en place la classe `LinguisticFactory`, qui se base sur le pattern Singleton, pour gérer la création de nouveaux `Type` et de nouveaux `Concept`. C'est cette classe qui sera utilisée lors de l'instanciation des `Type` et des `Concept`, et qui "cachera" à l'utilisateur le système de gestion et de stockage des `Type` et des `Concept`.

Le package `types_gestion` comprend également une classe d'exception `IncompatibleTypesException`, qui permet de gérer l'incompatibilité de deux types.

4.6.3 Package `graph_concepts_gestion`



A la fin de la création et du remplissage d'un nouveau scénario, les `Concept` manipulés par l'utilisateur sont stockés dans un graphe, implémenté à l'aide des classes `GraphNodeDefault` et `GraphConcepts`, et des interfaces `GraphConceptsInterface` et `GraphNode`.

Chaque `GraphConcepts` est constitué d'un `GraphNode` racine. Un `GraphNode` est identifié par le `Concept` qu'il représente et la liste des `GraphNode` dont il est le "parent".

Par construction, le graphe créé à partir des `Concept` manipulés par l'utilisateur est un graphe orienté acyclique (l'orientation se fait d'un `GraphNode` vers ses "fils", dont il possède la liste). Cependant, ce graphe doit être enregistré dans un fichier XML et il était plus simple pour cette utilisation d'avoir une structure purement arborescente.

Nous avons donc ajouté trois champs qui nous permettent d'avoir une structure d'arbre tout en conservant la possibilité d'avoir des références multiples à un même concept ; ces champs sont principalement utilisés dans la méthode *addChild(GraphNode child, int index)* de la classe `GraphNodeDefault`.

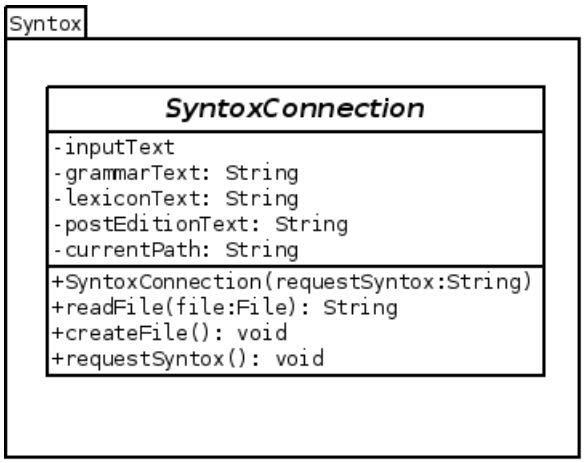
- *tag* (`boolean`) : ce champ est modifié lors du parcours de la liste de `GraphNode` *childrenList*, il permet de savoir si le `GraphNode` que l'on veut ajouter a déjà été rencontré parmi les enfants du `GraphNode` considéré. Si cela est le cas, cela signifie que l'on va avoir une référence multiple au même `GraphNode`, et donc au même `Concept`.
- *isReference* (`boolean`) : ce champ a pour valeur *true* si d'autres `GraphNode` font référence au `GraphNode` considéré.
- *reference* (`GraphNode`) : ce champ contient le `GraphNode` auquel on fait référence, le cas échéant.

On a donc un arbre de `GraphNode` au lieu d'un graphe acyclique ; si deux `GraphNode` ayant trait au même `Concept` sont présents dans cet arbre, alors c'est que l'un fait référence à l'autre.

Les scénarios (qui seront plus tard enregistrés dans un fichier .XML) sont représentés par un nom et une liste de `GraphConcepts`.

Le pattern Singleton est ici réutilisé comme modèle pour la classe `GraphNodeFactory`, qui permet d'instancier la classe `GraphNodeDefault` en cachant l'implémentation du graphe de `Concept`.

4.7 Package `syntox`



Ce package contient les éléments permettant de communiquer avec le module `syntox` accessible depuis Internet. Le package `syntox` comporte les éléments suivants : une classe `SyntoxConnection`, et des fichiers de configuration grammar.txt, lexicon.txt, postEdition.txt, headerPage.html et endingPage.html.

La classe `SyntoxConnection` permet de coordonner les éléments présents dans le package afin de concevoir une page HTML, *pdp.html*, permettant d'établir la communication avec le logiciel Syntox. Elle reçoit en entrée la requête Syntox au préalable formatée par la méthode *generateSyntox()* des classes `GraphConcepts` et `GraphNodeDefault` du package `linguistic`.

Les fichiers *.txt* comportent les autres éléments que le logiciel Syntox à besoin de posséder pour la génération du texte mais dont nous ne devons pas nous charger de générer. Parmi ces éléments nous avons la grammaire qui est stockée dans *grammar.txt*, le lexique dans le fichier *lexicon.txt*, et enfin les outils syntaxiques supplémentaires dans *postEdition.txt*.

De la même façon, les fichiers *headerPage.html* et *endingPage.html* contiennent les éléments nécessaires à la page HTML. C'est à dire l'entête de la page, qui comporte entre autres le descriptif de la page, l'ouverture du formulaire ainsi que la requête Syntox, et le bas de page, qui quant à lui contient les balises de fermeture du formulaire et du corps de la page.

L'utilisation de la méthode *requestSyntox()* de la classe `SyntoxConnection` permet la construction de la page HTML ainsi que le lancement automatique de cette dernière dans le navigateur par défaut de l'utilisateur. La méthode *requestSyntox()* fait appel

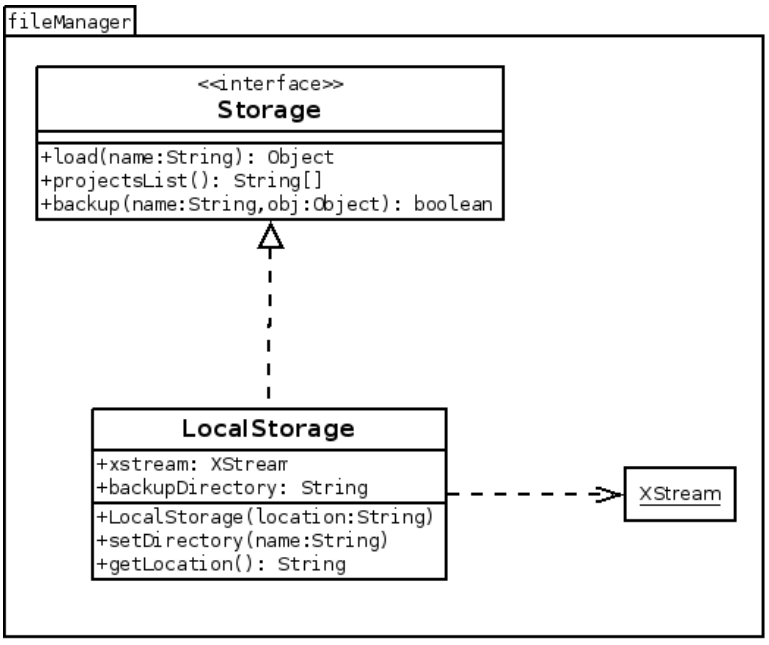
à *createFile()*, une méthode qui va assembler chaque élément nécessaire dans le fichier *pdp.html*. Nous avons dans l'ordre :

– L'écriture de l'entête de la page HTML, *headerPage.html*.
– La requête Syntox, contenu dans la variable *inputText*.
– Respectivement l'ajout du contenu des fichiers *grammar.txt*, *lexicon.txt* et *postEdition.txt* afin de constituer le corps de la page.
– Le contenu du pied de page, *endingPage.html* pour clôturer la création de la page.

Une fois la page HTML créée, cette dernière est ouverte dans le navigateur par défaut de l'utilisateur. La requête peut être modifiée avant de lancer la compilation du texte via le bouton. Notons que la requête appairait également dans notre application pour permettre, en cas d'échec d'ouverture du navigateur, de faire un copier/coller directement sur le site.

Nous avons choisi d'implémenter ce package de cette manière suite à un problème rencontré. En effet, nous ne réussissions pas à communiquer avec le formulaire présent sur la page internet. Les données devaient être mal envoyées et le formulaire ne se déclenchait pas. Toutefois, cette implémentation apporte certains avantages. En effet, la création de la page se fait indépendamment de la version en ligne, par conséquent nous n'avons pas besoin de posséder une connexion internet. De cette manière on peut concevoir la page *pdp.html*, la sauvegarder en dehors du répertoire et la lancer plus tard. En contrepartie, les fichiers *grammar.txt*, *lexicon.txt* et *postEdition.txt* doivent être manuellement modifiés si ces derniers ont besoin d'être changés.

4.8 Gestion des fichiers



4.8.1 Présentation globale

Dans le cadre de notre programme, il est indispensable de pouvoir à tout moment enregistrer le travail en cours sur le disque dur et de pouvoir y accéder pour le modifier : en effet, les projets, scénarios et concepts doivent être modulables et surtout, du fait de leur structure complexe, nécessitent un travail conséquent de la part de l'utilisateur, qui appréciera de ne pas avoir besoin de recommencer son travail à chaque démarrage du programme.

La classe `Projet` regroupe tous les champs nécessaires au travail sur une base de donnée précise : scénarios, concepts et `LinguisticFactory`. Il a donc été décidé que chaque `Projet` serait stocké sur le disque sous forme d'un fichier regroupant tous ses champs.

Afin de faciliter la sérialisation des données, nous avons choisi d'utiliser la bibliothèque `XStream`. Cette bibliothèque permet d'exporter facilement les objets sur le disque dur sous forme de fichier XML, et de les charger de manière également intuitive.

Pour cela nous avons ajouté la bibliothèque simplifiée `xstream-1.4.4.jar`, qui regroupe les fonctions principales dont nous aurons besoin afin de stocker et récupérer les objets de type `Projet`.

4.8.2 Implémentation

Les fonctions nécessaires pour le chargement et enregistrement des données sont regroupées dans le package `fileManager`. Ce paquet contient l'interface `Storage` et son implémentation `LocalStorage`.

Nous avons choisi d'implémenter cette partie de la manière suivante :

- Une fonction de chargement : *public Object load (String name)*
- Une fonction d'enregistrement : *public boolean backup(String name, Object obj)*
- Une fonction d'affichage des fichiers XML du répertoire courant : *public String[] projectsList()*

Par défaut, on considère que le répertoire contenant les projets existants et stockant les nouvelles sauvegarde sera (en chemin relatif par rapport au répertoire du programme) : ../Projets

Enregistrement La bibliothèque `XStream` permet donc l'enregistrement d'Objects sur le disque. Cette fonction se contente de créer un `FileWriter` du nom du paramètre *name* à l'emplacement ../Projets et d'appeler la fonction *xstream.toXML* sur l'argument de type `Object`(on doit donc effectuer un cast sur le `Project`) : on obtient un fichier .XML dans le répertoire contenant toutes les informations nécessaire au travail sur le projet : les concepts existants, les scénarios déjà créés, et tous les types.

Chargement La fonction *load* appelle simplement la fonction *xstream.fromXML* sur un fichier dont le nom est passé en paramètre de la fonction, et renvoie un `Object`, que

l'on cast en `Project` et qui peut être ensuite utilisé et modifié dans le programme.

Affichage du répertoire courant Cette fonction renvoie un tableau de `String` dont les éléments sont les noms de fichiers .XML contenus dans le répertoire ../Projets, sans l'extension de fichier. Cette fonction est utilisée dans la partie utilisateur du programme, au moment de choisir un projet existant.

Utilisation des fonctions de chargement et d'enregistrement De part la manière de sauvegarder choisie, chaque modification (que ce soit un concept, un scénario ou un projet intégral) nécessite de sauvegarder l'intégralité du projet : toutes les fonctions de sauvegarde du programme appellent la fonction *backup* de la classe `LocalStorage` après avoir modifié l'objet chargé dans le programme.

On garantit ainsi l'intégrité du travail en cours par l'utilisateur au détriment éventuel du temps d'exécution : cependant, cette solution a été préférée pour sa simplicité d'utilisation (une seule fonction de sauvegarde), pour sa propreté (un seul et unique fichier par projet, et non par scénario ou concept).

De plus, les structures de données utilisées, même si non limitées en taille par l'implémentation, ne seront de manière intuitive jamais gigantesques : on ne peut pas imaginer une base de données associée à plusieurs millions de concepts ou de scénarios. Les temps d'éxécution des fonctions d'enregistrement et de chargement sont donc considérées comme triviaux.

La fonction d'enregistrement est donc appelée à chaque clic sur un bouton de type "Enregistrer" (panel Scénario, Projet et Concept, administrateur ou utilisateur). La fonction de chargement est utilisée lors du choix du projet existant par l'administrateur ou l'utilisateur (au lancement du programme dans la plupart des cas, si l'utilisateur ne quitte pas le projet en cours pour en changer). La fonction d'affichage du répertoire, quand à elle, est appelée au choix du projet, juste avant le chargement.

Chapitre 5

Présentation du logiciel fourni

Dans cette partie, nous allons présenter les fonctionnalités du logiciel que nous avons développé.

Au lancement du logiciel, le menu principal s'affiche, offrant la possibilité à l'utilisateur soit de passer en mode "Utilisateur classique", soit en mode "Administrateur".

Le mode "Administrateur" permet de créer ou de modifier des projets. Pour rappel, un projet regroupe des informations de connexion à une base de données, un ensemble de concepts et de types, ainsi que des scénarios, conçus à partir de ces mêmes objets.

Pour créer un nouveau projet, le logiciel demande à l'administrateur de fournir un certain nombre d'informations :

- le nom du projet
- l'url d'une base de données MySQL valide
- les informations de connexion correspondant à cette base de données (identifiant et mot de passe)

L'administrateur arrive sur l'écran d'édition du projet. Cet écran possède deux onglets : un destiné à la création et à l'édition des concepts, l'autre à celles des types. Un certain nombre de types ont déjà été ajoutés automatiquement par le logiciel en inspectant la base de données qui a été fournie.

Pour aider l'administrateur, plusieurs outils d'édition sont disponibles. Il peut ainsi créer des types du nom de son choix facilement et leur attribuer un surtype, ce surtype étant modifiable et supprimable à tout moment. Il en va de même pour l'ajout des concepts, qui nécessite que l'administrateur fournisse son nom et son type. Il peut aussi leur ajouter des arguments, qui seront des types.

La suppression, le renommage et l'édition de la description de ces types et concepts sont disponibles via un clic droit.

A la fin de l'édition du projet, sa sauvegarde est possible via la commande "Enregistrer" du menu "Fichier". Les concepts et les types sont alors stockés dans un fichier .XML, qui correspond au projet. Ce fichier a pour nom le nom du projet.

Si l'administrateur veut modifier un projet déjà créé, il peut le faire grâce au bouton "Modifier Projet", puis sélectionner le projet à modifier dans la liste des projets existants.

Il se retrouve ensuite sur l'écran d'édition, décrit plus haut.

Dans le cas où le bouton "Utilisateur" est sélectionné dans le menu principal, l'utilisateur peut accéder à un menu lui permettant soit d'utiliser un scénario déjà existant dans le projet qu'il a sélectionné préalablement (dans une liste présentant tous les projets existants), soit d'en créer un lui-même.

Grâce au menu "Scénarios existants", l'utilisateur a donc accès à la liste des scénarios déjà présents dans le projet. Il peut, s'il le souhaite, supprimer un scénario de la liste en faisant un clic droit ou tout simplement en choisir un et cliquer sur valider.

L'entrée Syntox correspondant au scénario qu'il a choisit s'affiche alors, et une page permettant de visualiser le texte généré par cette entrée Syntox est ouverte sur le navigateur par défaut de l'utilisateur.

Le menu "Scénario personnalisé" permet quant à lui à l'utilisateur de créer un scénario à partir des concepts présents dans le projet. Une première colonne affiche les racines de chacun des arbres conceptuels du scénario, ces arbres correspondent en fait à chaque phrase que contiendra ce même scénario. L'utilisateur peut donc former les phrases qu'il désire, en précisant au fur et à mesure leurs attributs.

Une fois le scénario terminé, l'utilisateur peut enregistrer son travail via le menu "Fichier" présent dans la barre de menu.

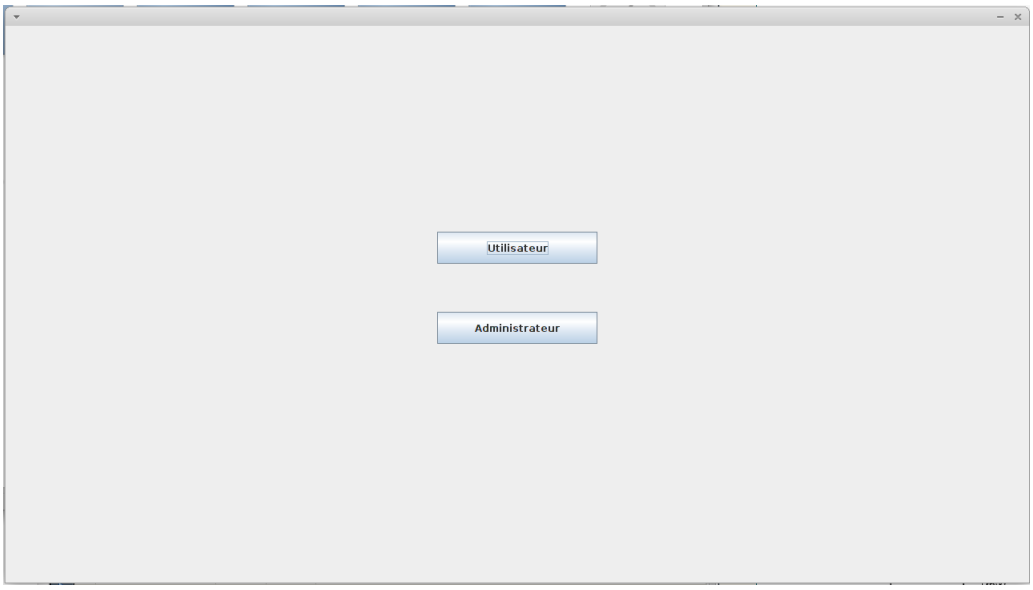


FIGURE 5.1 – Ecran d'accueil

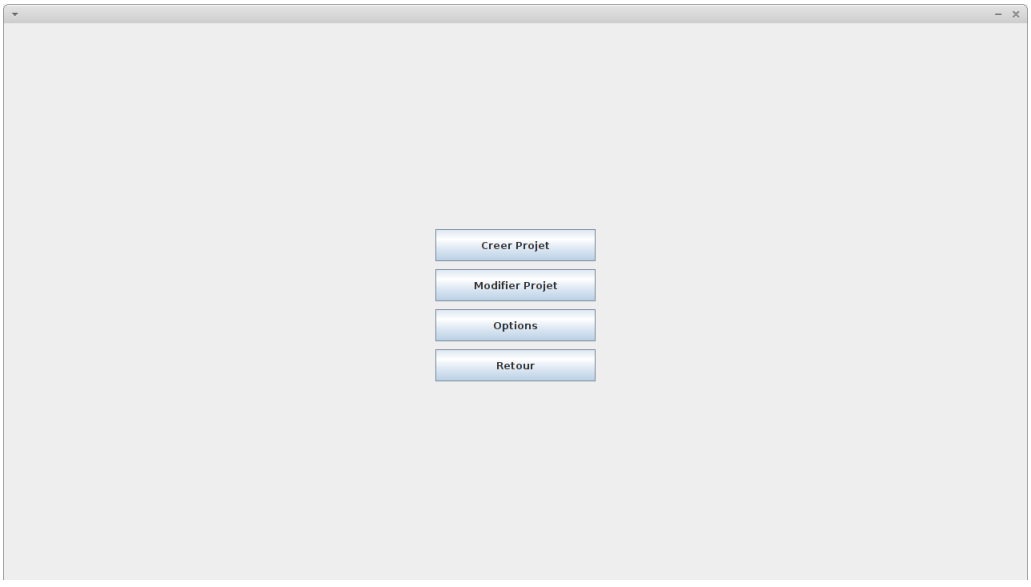


FIGURE 5.2 – Ecran du menu administrateur

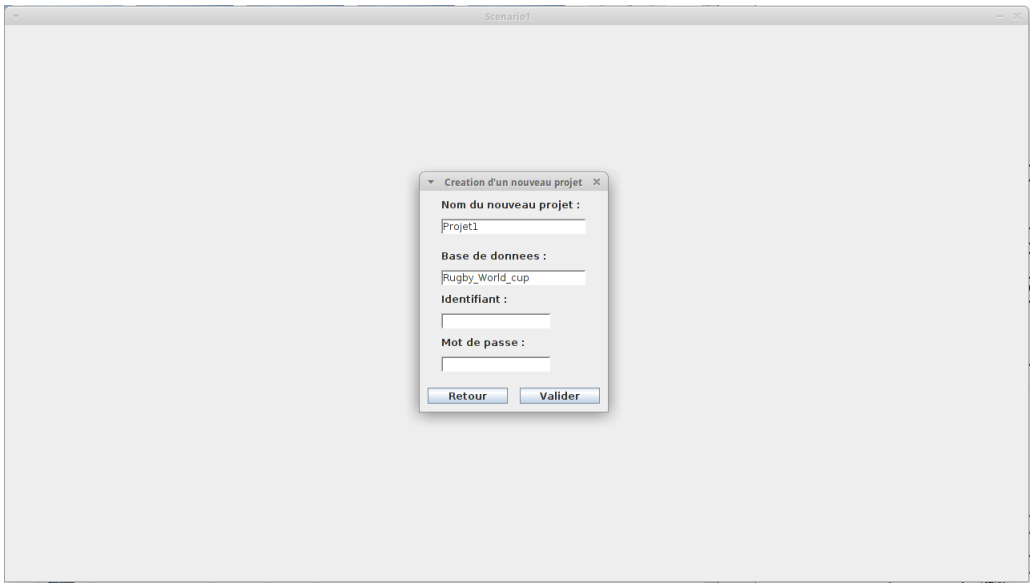


FIGURE 5.3 – Ecran de création d'un nouveau projet

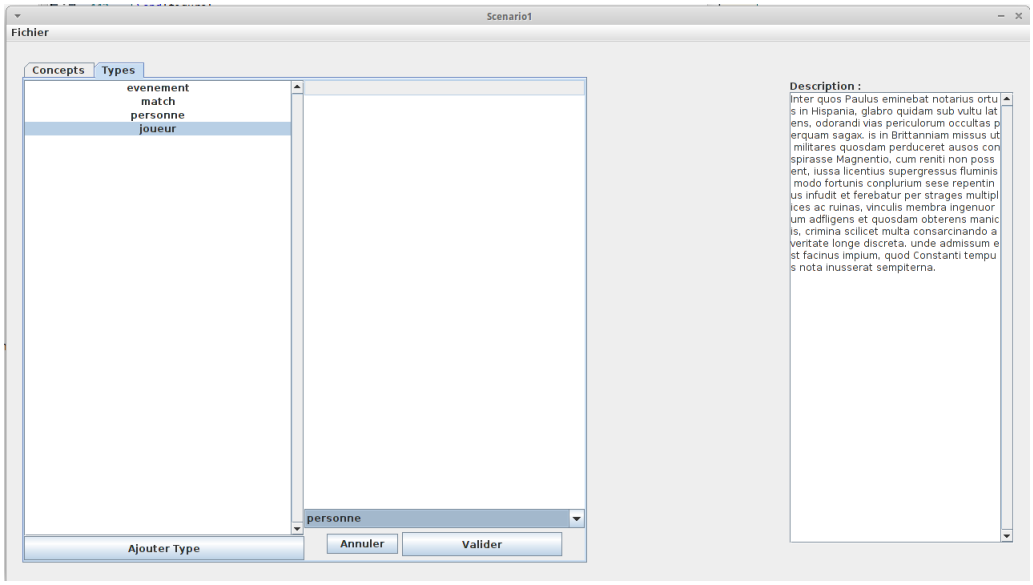


FIGURE 5.4 – Ecran de création des types

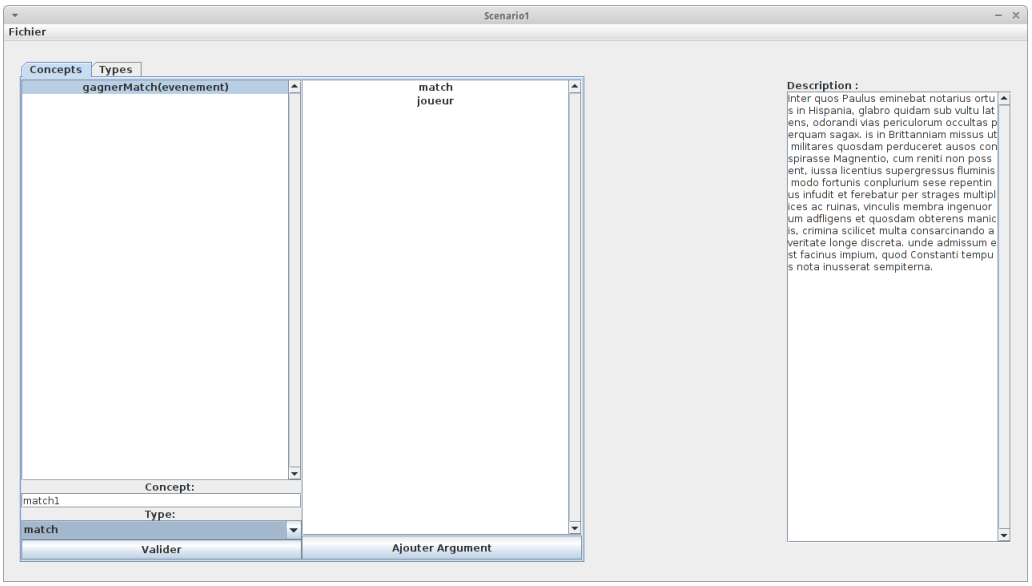


FIGURE 5.5 – Ecran de création des concepts

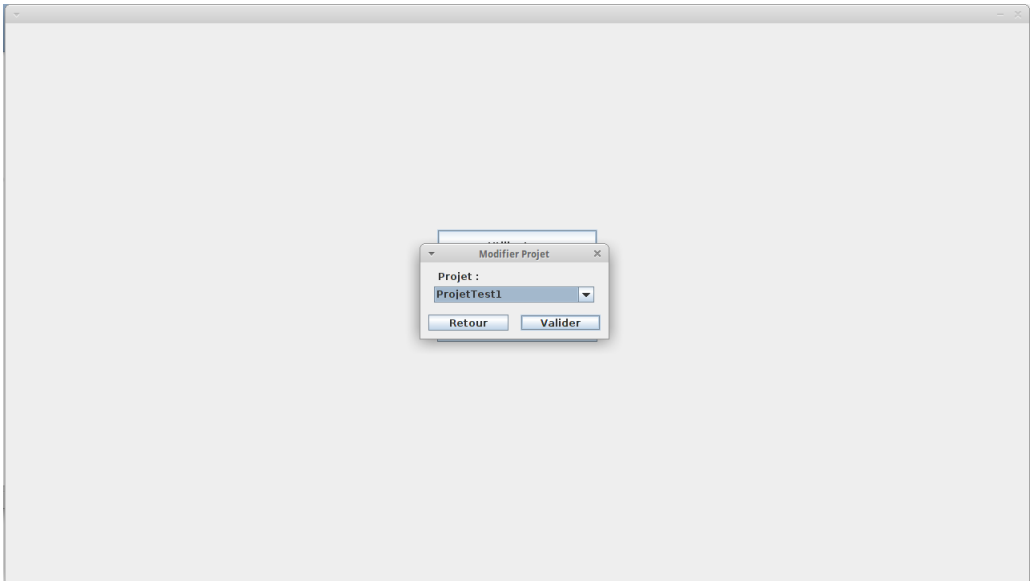


FIGURE 5.6 – Ecran d'ouverture d'un projet

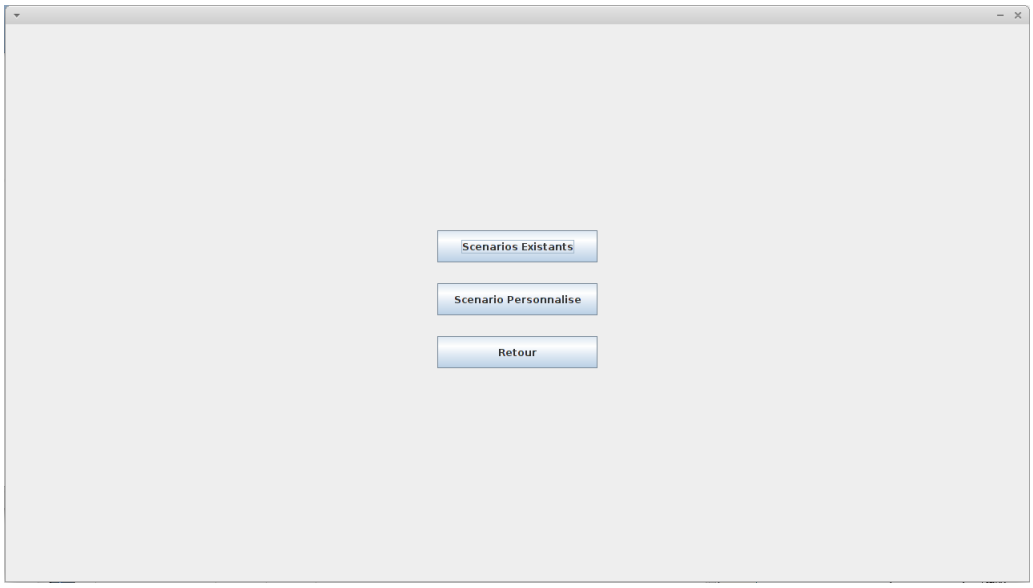


FIGURE 5.7 – Ecran de sélection de scénario existant/personnalisé

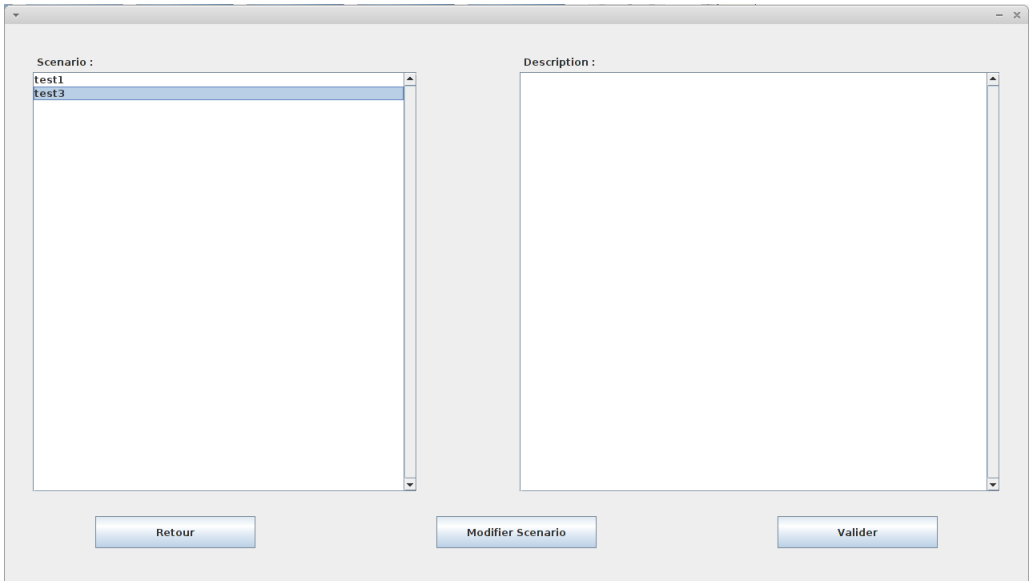


FIGURE 5.8 – Ecran de sélection de scénario existant

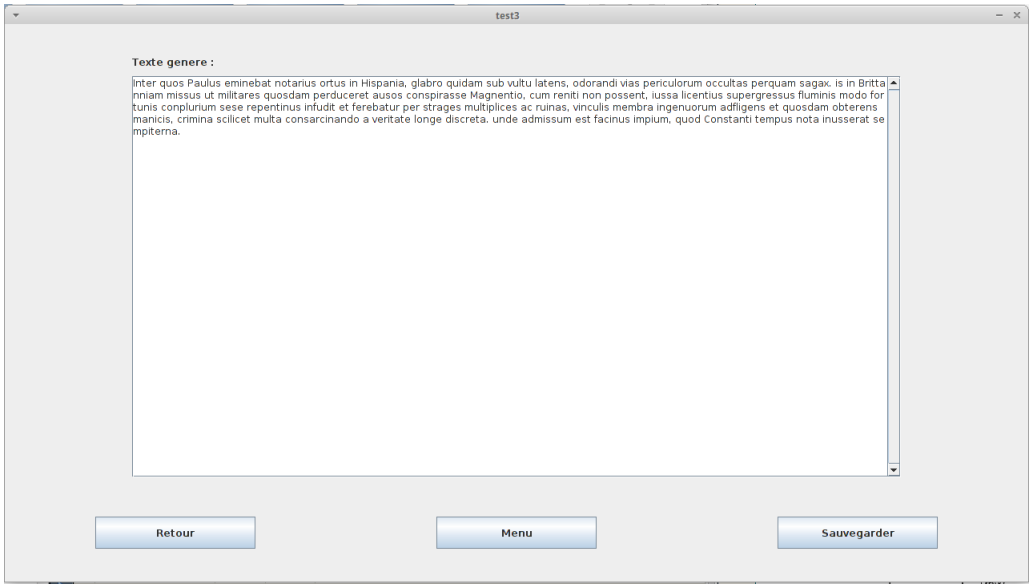


FIGURE 5.9 – Ecran affichant le texte généré

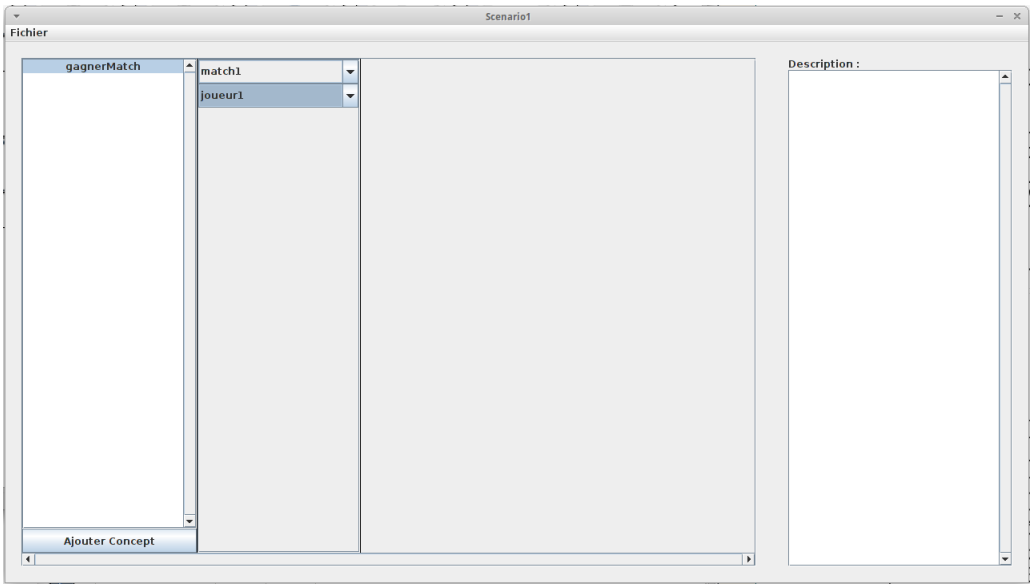


FIGURE 5.10 – Ecran de création d'un nouveau scénario

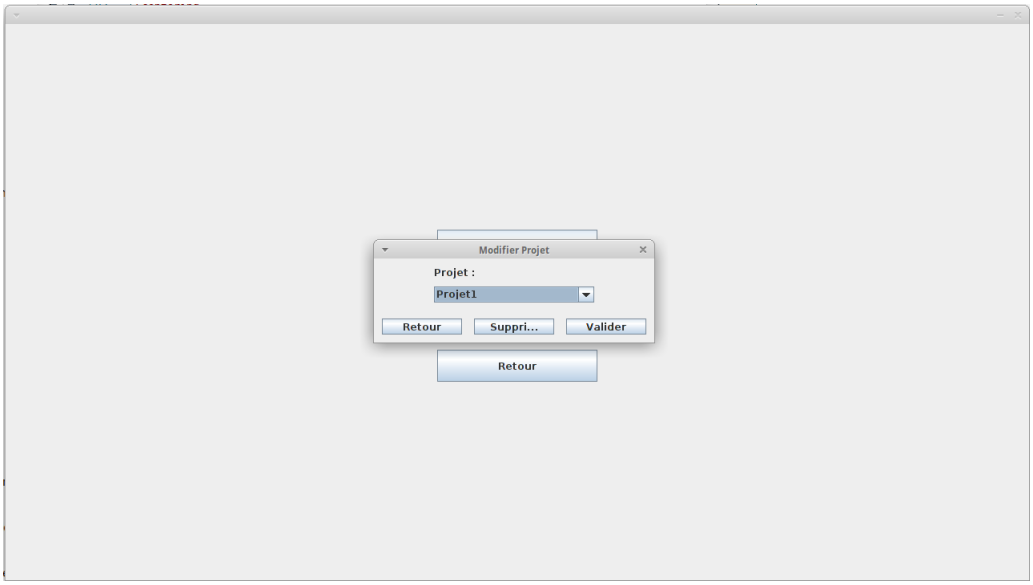


FIGURE 5.11 – Ecran de choix du projet à modifier

Chapitre 6

Tests et résultats

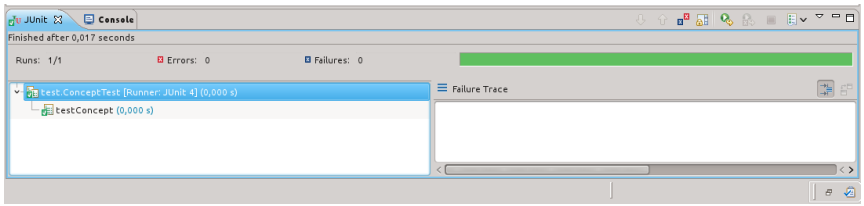
6.1 Tests effectués

Le but de ces tests est d'assurer le bon fonctionnement de notre application. Nous pouvons les regrouper selon l'ensemble de fonctionnalités auquels ils se rattachent. Afin de ne pas mélanger le code source et les essais, nous avons créé un package différent regroupant l'ensemble des classes et méthodes de test.

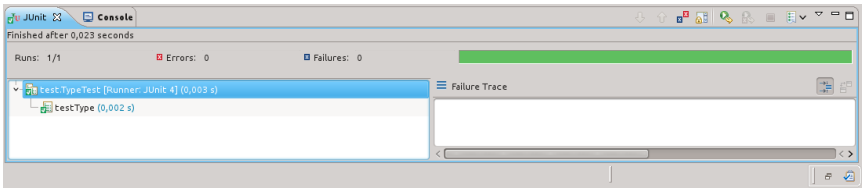
6.1.1 Package Linguistique

Ce package regroupe les principaux objets utilisés par l'interface graphique tels que les concepts, graphes, noeuds et système de typage des concepts. Son bon fonctionnement était donc primordial pour ne pas avoir d'erreurs lors de l'utilisation de ces derniers par l'interface graphique. Nous avons donc élaboré des tests unitaires, au moyen de JUnit, sur chacune des classes et méthodes.

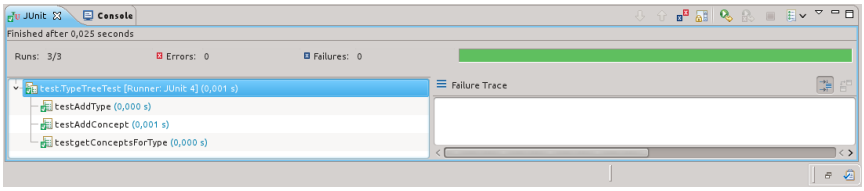
- `ConceptTest` : Dans les classes `Concept(Abstract/Complex/Simple)` nous rappelons qu'il y a majoritairement des accesseurs mais également des méthodes utilisées pour la génération de l'entrée Syntox. Notons que les accesseurs ne sont normalement pas des éléments soulevant des problèmes. Toutefois nous nous sommes assurés que les éléments retournés sont bien ceux attendus, surtout dans le cas de la liste et du nombre d'éléments qu'elle contient. Concernant la méthode *generateSyntox()* nous avons choisi de la tester lors de l'utilisation d'une instance de `GraphConcepts`.

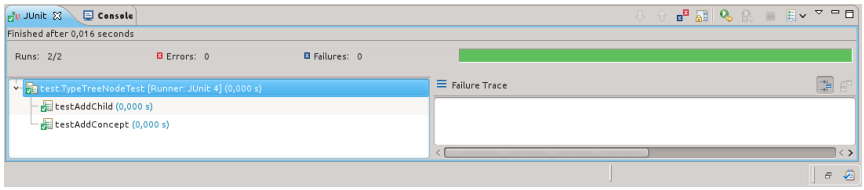


– `TypeTest` : Nous avons ici testé de manière similaire à `ConceptTest` les accesseurs présents.

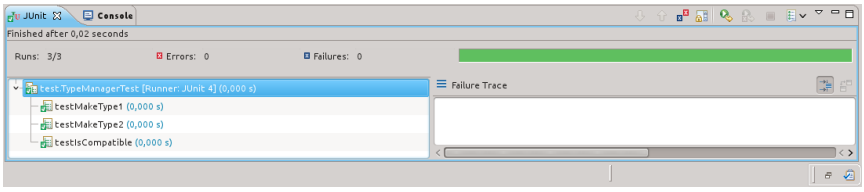


– `TypeTreeTest` et `TypeTreeNodeTest` : Nous avons fait le choix de tester l'intégralité des méthodes présentes dans ces classes, à savoir :
  – L'ajout d'un nouveau concept à la liste du noeud en question.
  – L'ajout d'un fils à la liste concernée.
  – L'ajout d'un type dans l'objet `TypeTree` (Plus précisément dans la `Map` de l'objet) en respectant les liaisons avec un type prédécesseur (surType). Et de façon similaire, l'ajout d'un concept à ce même objet.
  – La méthode permettant de retourner l'ensemble des sous-types que contient un type.

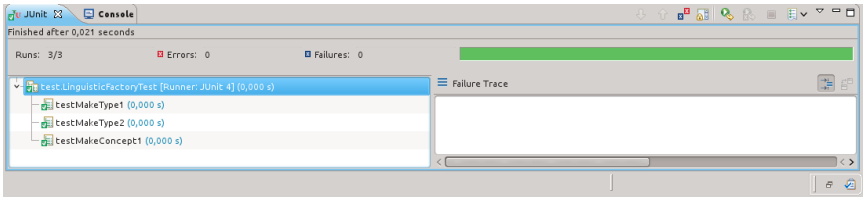




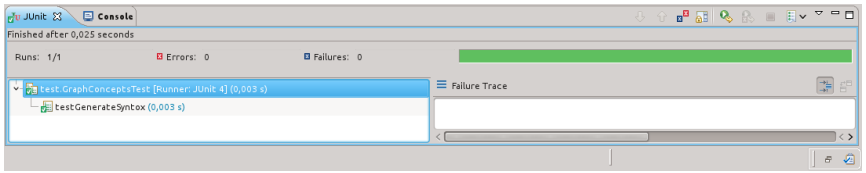
– `TypeManagerTest` : Nous rappelons que cette classe permet de gérer la création des types ainsi que leur insertion dans l'arbre. Nous avons testé la méthode permettant l'ajout d'un type dans l'arbre (*makeType()* avec un et deux arguments) ainsi que la méthode retournant si deux types sont compatibles (*isCompatible()*).

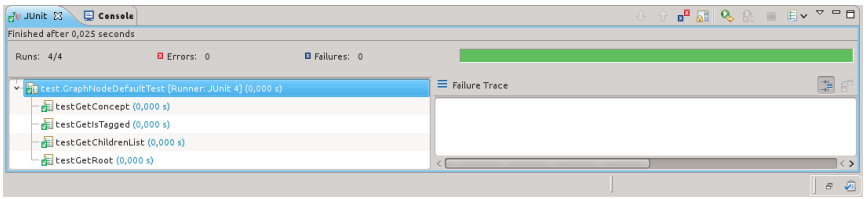


– `LinguisticFactoryTest` : Nous avons choisi de tester les méthodes qui apportent les fonctionnalités suivantes :
  – L'ajout d'un type au `TypeManager` au moyen de la méthode *makeType*.
  – L'ajout de concept, simple ou complexe à l'arbre qu'utilise le `TypeManager`.



– `GraphConceptsTest` et `GraphNodeDefault` : Les classes `GraphConcepts` et `GraphNodeDefault`, comme nous le rappelons, permettent la représentation sous forme de graphe des concepts utilisés et sélectionnés par l'utilisateur au moyen de l'interface graphique. Nous avons testé pour ces classes :
  – Certains accesseurs qui nous semblaient importants dans le sens où ces classes utilisent de nombreux objets d'autres classes, il est donc nécessaire de pouvoir garantir l'accès à ces objets et à leur référence.
  – La méthode permettant la création de la requête qui sera envoyée à Syntox (*generateSyntox()*). Cette méthode est appelée depuis `GraphConcepts` et fait appel à son homologue présente dans `GraphNodeDefault`.





6.1.2 Core

Dans ce package se trouvent toutes les classes permettant le lancement de notre application et les outils nécessaires tel que les éléments permettant la gestion de la sauvegarde des projets, un utilitaire regroupant les informations nécessaires pour la base

de données et un système de cryptage/décryptage de mot de passe pour sécuriser l'accès à la base de données. Nous avons choisi de tester particulièrement les classes présentes dans les fichiers `PasswordManager`, `Project` et `InfoDB`. Plus précisément nous avons validé les fonctionnalités suivantes :

- L'encryptage ainsi que le décryptage d'un mot de passe. En effet, comme indiqué dans le cahier des besoins, nous ne souhaitons pas laisser le mot de passe utilisé pour la connexion à la base de données en clair. Il nous a donc fallu crypter une fois la connexion faite mais également décrypter si on doit réutiliser le mot de passe.
- La conservation des informations saisies pour la base de données au moyen de la méthode *initBase*.
- L'ajout et la suppression de scénarios relatifs à un `Projet`.

6.1.3 Database[Connection/Inspection]

6.1.4 Syntox

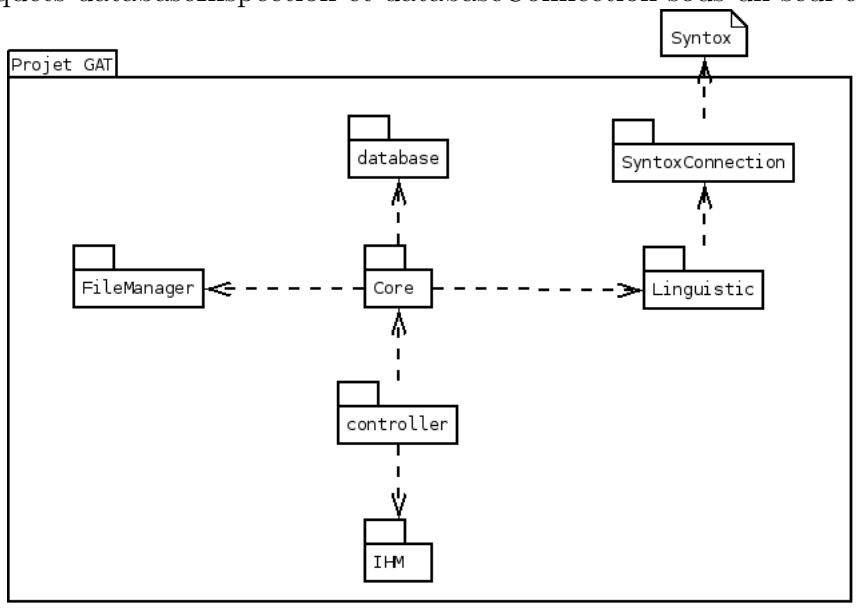
6.1.5 Mémoire et temps de calcul

6.2 Résultats

Chapitre 7

Conclusion et perspectives

On peut noter de manière positive que le diagramme ne possède pas de cycles. Il serait possible de supprimer le DAG en créant une nouvelle classe séparant le controleur de l'IHM. Ainsi, on briserait les dépendances entre l'interface graphique et les paquets LinguisticFactory et databaseInspection. De plus, il faudrait envisager regrouper les paquets databaseInspection et databaseConnection sous un seul et meme paquet.



Bibliographie

[1] John Bateman. Natural Language Generation : an introduction and open-ended review of the state of the art. `http://www.fb10.uni-bremen.de/anglistik/langpro/webspace/jb/info-pages/nlg/ATG01/`, 1993 (dernier accès : 16 janvier 2013).

[2] Lionel Clément. Synthèse de texte avec le logiciel Syntox. In AFCP ATALA, editor, *Actes de la conférence conjointe JEP-TALN-RECITAL*, 2012.

[3] Meunier F. Combet V., Danlos L. EasyText : an operational NLG System. In *European Workshop on Natural Laguage Generation*, 2011.

[4] Laurence Danlos. Ecriture automatique. In *Les cahiers de l'INRIA*, 2010.

[5] Mourhri et al. Générateur automatique de texte. Mémoire de Projet de Programmation, Université Bordeaux 1, 2012.

[6] Danlos L. et Roussarie L. *La génération automatique de textes*. 2000.