Le projet ELIZA-GPT 2023

- Arthur Desbiaux, P2006393
- Valentin Cuzin-Rambaud, P2003442

Contents

I. Présentation du projet

Ce projet se déroule dans le cadre de l'UE Gestion de projet. On y aborde la thématique de l'IA en recodant l'une des premières IA conversationnel eliza-gpt. Le projet est de réaliser un chatbot qui permet d'avoir des discussions avec réponses multiples, situationnelles. Tout cela en respectant une norme de structure de code et une rigueur jamais vue auparavant, notamment en utilisant l'architecture MVC, en appliquant les principes de GRASPS et en utilisant des Design patterns...

II. Principe GRASP

Parmi tous les principes GRASP[1] nous essayons au mieux de respecter le faible couplage, notamment avec la classe MessagePattern qui permet de rajouter facilement des messages prédéfinis dans la logique. De plus avec le package SelectAnswer, on respecte le polymorphisme, avec le type générique T. Les principes GRASPS permettent de généraliser son code, pour faciliter le développement du code, l'ajout de nouvelle fonctionnalités, la réutilisation, surtout dans un cycle de projet SCRUM.

III. Design Patterns

0. MVC

Le projet est structuré avec l'architecture **??** afin de séparer la logique métier de l'affichage, qui réduit notre couplage (principe de GRASP). Nous avons fait une implémentation pull-based, c'est-à-dire que la vue tire de l'information de la donnée généré par le modèle. La vue observera les changements dans le modèle (observé), c'est l'utilisation du design Patterns Observer[3] Cela permet par exemple de synchroniser nos vues.

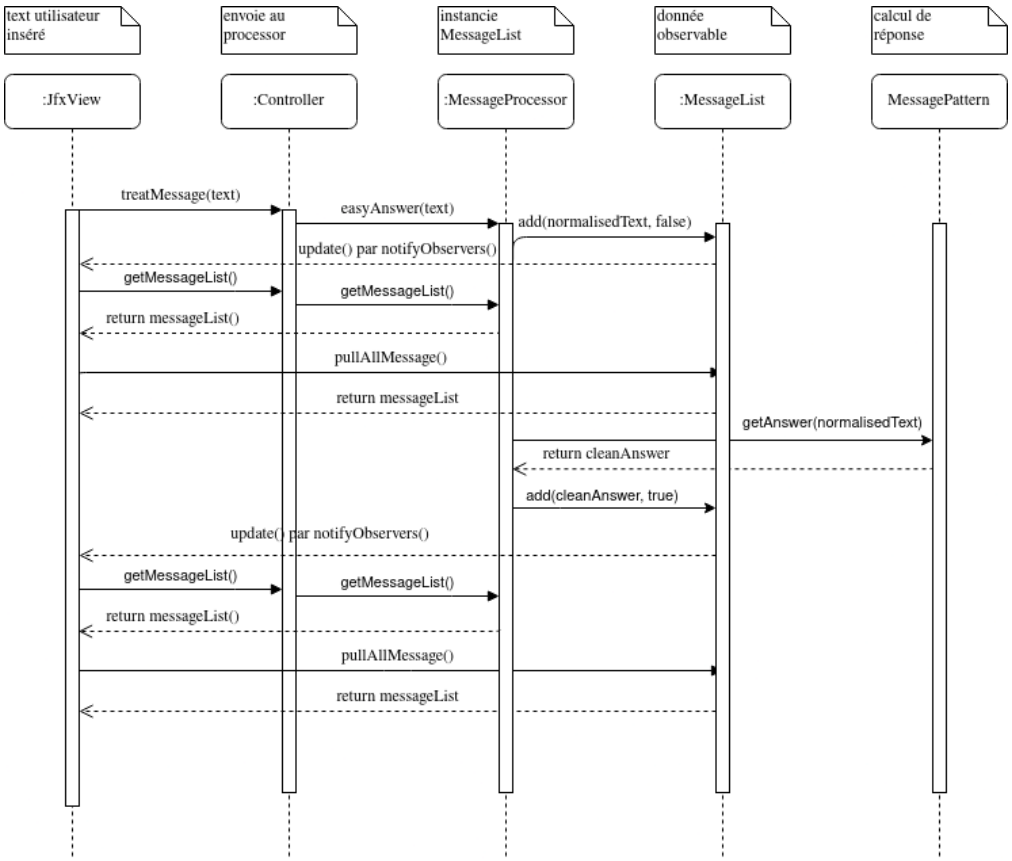

Figure 1: Diagramme MVC

1. Stratégie

Nous avons décidé d'utiliser Stratégie[4] pour la réalisation de nos filtres. Nous voulions manipuler un objet Filtre qui s'instancie avec le bon filtre à utiliser (regex, complete word, sub-string). Stratégie est parfait dans ce cas d'utilisation, car il permet le changement dynamique de comportement de l'objet.

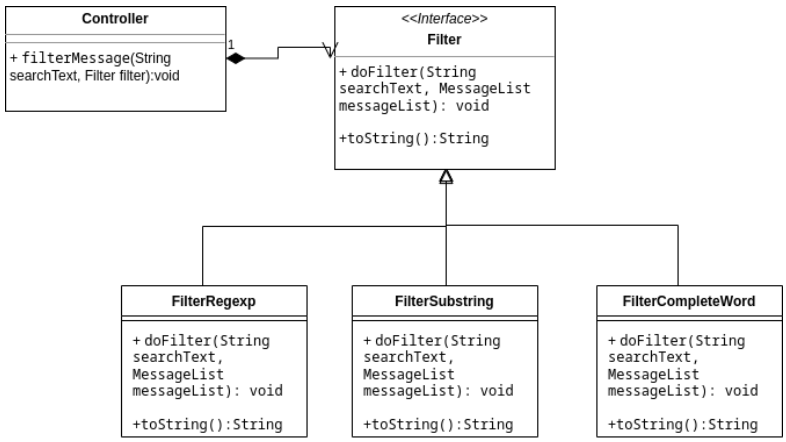

Figure 2: Diagramme UML Stratégie

Voilà comment nos filtres sont appelé, depuis la vue on récupère le filtre que l'utilisateur utilise. Lorsqu'on veut appliquer la recherche, on utilise donc Filter comme objet qui va ensuite utiliser la bonne méthode de filtre. Chaque Filtre est un extends de l'interface Filter qui possède deux méthodes, une pour afficher correctement le nom du filtre et l'autre qui va trier la liste de messages (et donc la modifier).

2. DAO

L'utilisation de DAO[5] pour la conjugaison, des verbes permettent d'implémenter l'accès à un dictionnaire en synchronisant avec un fichier de configuration. Ainsi, le fichier de configuration peut être modifié à partir du code, pour rajouter, modifier, supprimer des verbes. On peut aussi charger un autre fichier de configuration par exemple si on veut conjuguer des verbes en anglais. Le Dao implémenté nous permet ainsi une configuration persistante et complémente le modèle MVC en séparant encore plus la donnée du modèle.

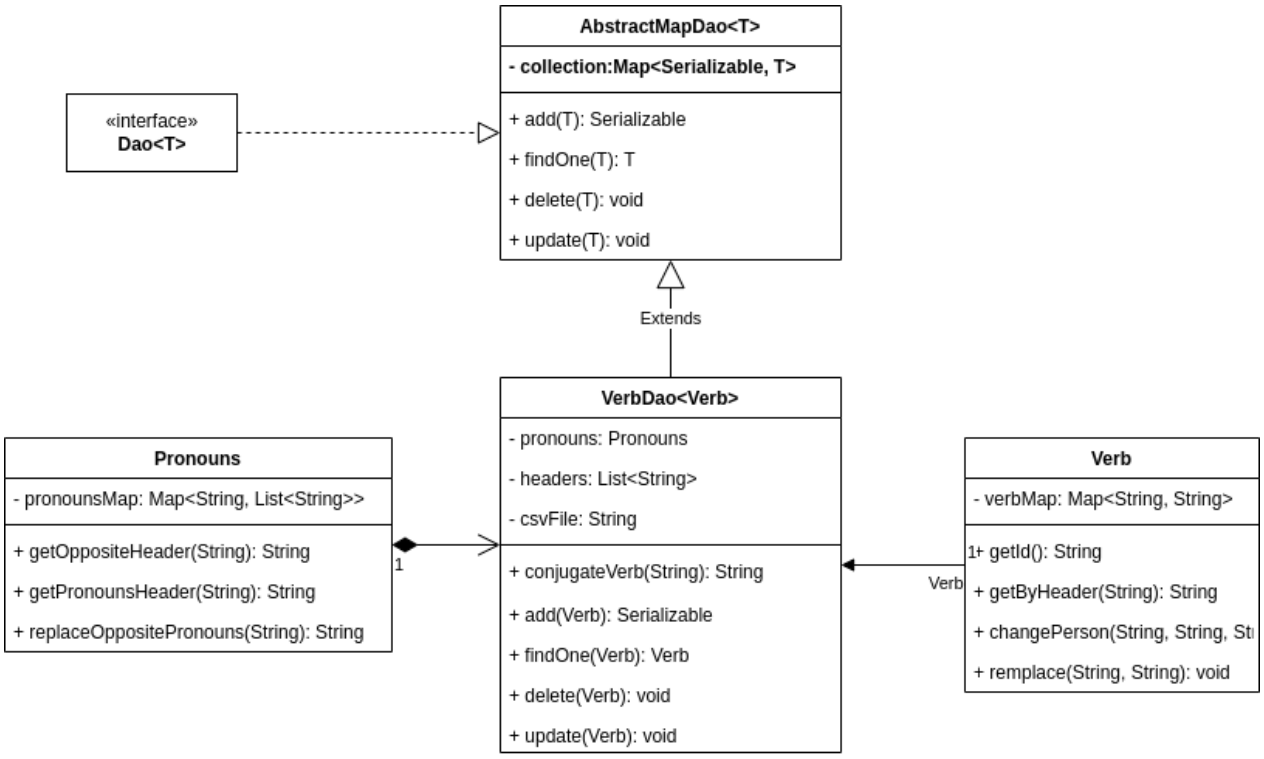

Figure 3: Diagramme UML DAO

Nous avons repris le code de Lionel Médini afin de partir sur un Dao propre. On extends la classe VerbDao pour réécrire le CRUD (Creator, Read, Update, Delete) afin de gérer à la fois le fichier de configuration et la map. Pour le csv nous utilisons la bibliothèque *commons-csv de apache*. La fonction ConjugateVerb cherche dans la phrase les verbes conjugués dans le temps 1 puis les remplace par le temps 2 en accédant à la collection de Verb.

3. Adaptateur

L'Adaptateur[6] a été utilisé pour implémenter la météo. Lorsque nous voulons faire des requêtes sur l'api de météo, il faut pouvoir récupérer les données et les transformer pour les utiliser.

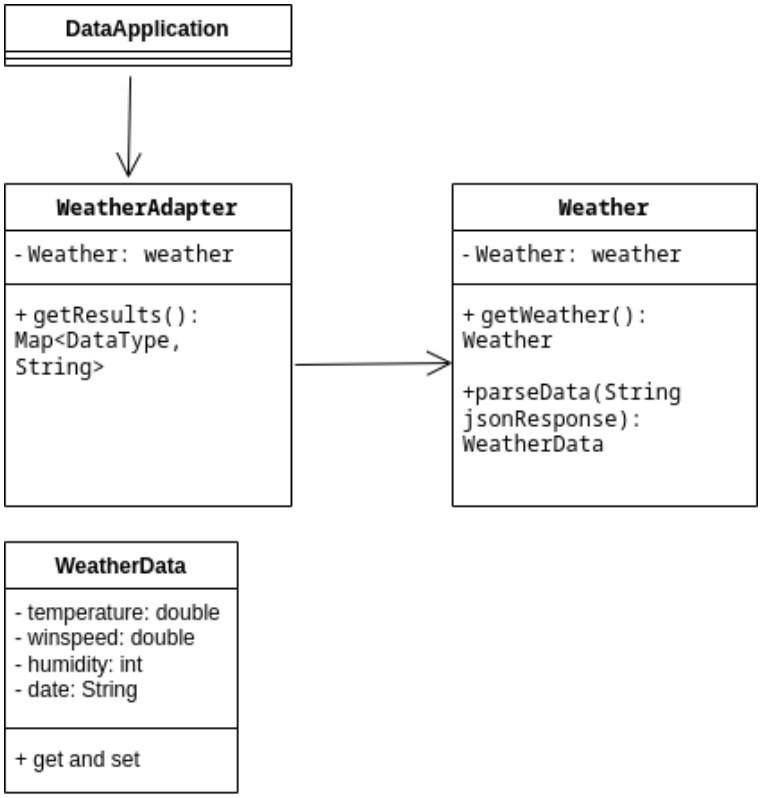

Figure 4: Diagramme UML Adaptateur

Pour l'implémenter, nous avons tout d'abord fait une classe qui possédera toutes nos data sur la météo. Ensuite, nous avons une classe Weather qui a une fonction pour faire la requête sur l'api et une autre fonction pour parser correctement nos données. La classe WeatherAdapter quand t'as elle se contente de récupérer la réponse de la requête (une instance de WeatherData) et de la transformer en Map. En procédant ainsi, on peut stocker nos données dans notre type DataType et ainsi les récupérés et réutilisés quand on veut dans l'application. Cela permet également de faire une seule requête API par moment où l'application est utilisée.

IV. Ethique dans l'IA

1. Enjeux , risques et bénéfices

Un enjeu majeur est la collecte de données, en effet, elle est nécessaire pour améliorer en continu le chatbot, mais récupère donc énormément de donnée privée. Les utilisateurs n'étant pas renseignés sur le sujet, ils ne savent pas les conséquences de la collecte d'informations et la façon dont ces données sont utilisées. Un autre problème est la possibilité de pouvoir orienter la façon d'interagir du chatbot, notamment en le nourrissant de mauvaises informations à caractères racistes (voir journaldugeek[7]).

Il y a également un problème d'humanisation du chatbot, étant donné qu'il se nourrit de donnée et de texte écrit par l'humain, le modèle générera des réponses qui ressemblent a une réponse potentiellement humaine. Ce qui peut entraîner des drames auprès des personnes instables (voir lefigaro[8]).

Un problème est également la manière dont ont nourri le chatbot, en effet pour lui faire discerner le bien du mal, il faut lui montrer des choses bienveillantes mais aussi malveillantes.

Mais il y aussi des points importants tels que la création de nouveaux emplois très demandé et accessible pas seulement aux informaticiens (voir presse-citron[9]).

2. Mesures prises

En termes de mesure légale, l'Italie a pris une grosse décision qui est la fermeture à l'accès à ChatGPT sur l'ensemble du territoire. Elle permet de montrer aux acteurs de l'intelligence artificielles que les questions de protection de donnée sont importantes, notamment avec la RGPD (voir wiki-RGPD[10]).

Également plusieurs experts pense qu'il faudrait freiner la course à l'IA. Ils la qualifient comment un "danger pour l'humanité" mais également comme imprévisible et impossible a contrôlé. Au final, aucune mesure n'a été vraiment prise de manière officielle, mais seulement des mises en garde (voir lemonde[11]).

Des mesures prises par OpenIA, est l'interdiction des propos caractères sexuel, illégaux ou en lien avec se faire du mal. Mais cela en détriment de la qualité de vie de ceux qui ont du évaluer ces choses (voir journaldugeek[12]).

3. L'éthique dans notre projet

Notre chatbot, n'utilise pas de réseaux de neurones, il répond seulement avec des réponses prédéfinies et se nourrit exclusivement de donnée que l'on cherche à récupérer telle que le nom de l'utilisateur ou des données météo. Ainsi, les dangers de notre chatbot sont moindres, cependant, il serait possible de créer des réponses automatiques "malveillantes" assez facilement. Comme des blagues à caractères raciste, sexuel ou encore misogyne.

Également nous ne gérons actuellement pas tous les cas des accords de genre et cela peut provoquer des problèmes d'inclusivité.

4. Test de l'éthique de ChatGPT et Bard

Nous avons essayé de tester les chatbots sur quelques points.

Pour ChatGPT, nous avons essayé de lui faire dire des choses à caractère misogynes :

https://chat.openai.com/share/ed8021d0-6ce5-433a-afb0-c644aff9b703

Dans cette conversation, nous nous présentons comme une humoriste professionnelle qui cherche à faire un spectacle misogyne.

Pour Bard :

https://g.co/bard/share/7af90e03de79

Dans cette conversation, on demande au chatbot ce qu'il pense des problèmes d'éthique dans l'IA conversationnelle.

V. Tests

Nous avons fait des tests par cas d'utilisations pour vérifier le bon fonctionnement de l'application. Par exemple avec nos deux vues, on a pu vérifier que notre MVC pull-based fonctionne bien, car les vues sont synchronisées. Lorsque l'on ajoute, supprime un message, l'affichage se met à jour sur les 2 vues. Cela fonctionne également de la même manière lors de l'application du filtre. Pour l'interface utilisateur, on s'est assuré que le changement de mode dark/light fonctionne correctement. Et que la structure de la fenêtre nous conviennent.

Pour la création de nos tests automatique, nous avons fait en sorte de découvrir le plus de code possible.(en se rapprochant le plus de 100% des méthodes utilisé)

Element	Class, %	Method, %	Line, %
fr.univ_lyon1.info.m1.elizagpt	84% (22/26)	66% (89/133)	55% (350/634)
controller	100% (1/1)	85% (6/7)	90% (10/11)
Controller	100% (1/1)	85% (6/7)	90% (10/11)
model	91% (21/23)	79% (83/104)	69% (340/487)
Adapter	33% (1/3)	7% (1/14)	2% (2/89)
Answer	100% (10/10)	96% (27/28)	87% (130/149)
Dao	100% (3/3)	92% (25/27)	80% (101/125)
Filter	100% (3/3)	100% (6/6)	100% (37/37)
Message	100% (3/3)	80% (17/21)	78% (40/51)
MessageProcessor	100% (1/1)	87% (7/8)	83% (30/36)
view	0% (0/1)	0% (0/20)	0% (0/130)
JfxView	0% (0/1)	0% (0/20)	0% (0/130)
Observer	100% (0/0)	100% (0/0)	100% (0/0)
App	0% (0/1)	0% (0/2)	0% (0/6)

Figure 5: testAuto

VI. bibliographie

GRASP[1]

Observer[3]

Stratégie[4]

DAO[5]

Adaptateur[6]

journaldugeek[7]

lefigaro[8]

presse-citron[9]

wiki-RGPD[10]

lemonde[11]

journaldugeek[12]

liberation

radioFrance