Projet MIF01 : ELIZA-GPT

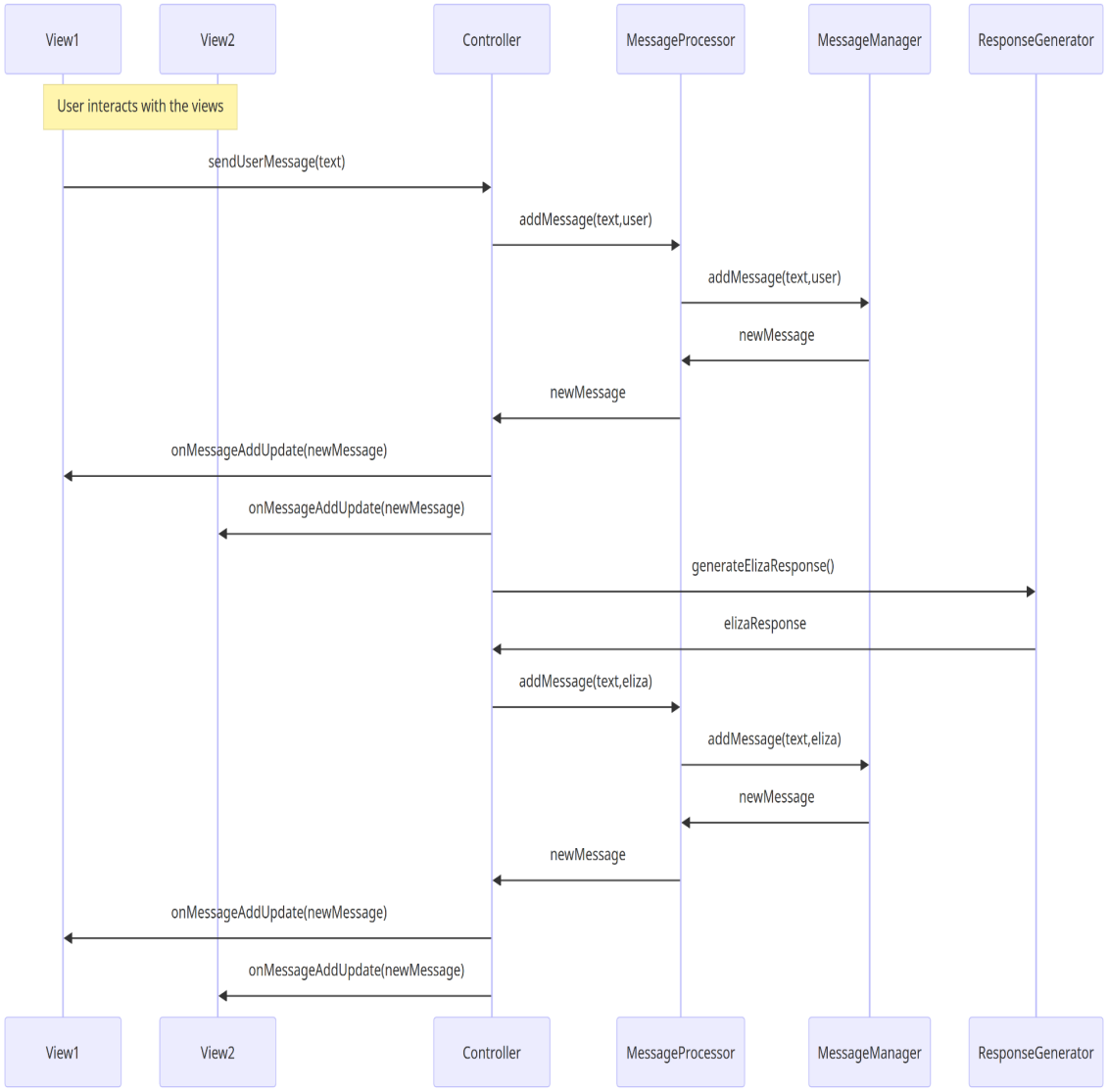
MAYOUF Lotfi - SLIMANI Abdennour

Ce projet a été réalisé en binôme pour l'UE de M1 Informatique "Gestion de projet et génie logiciel". L'objectif était de restructurer et d'améliorer une application de chatbot, Eliza GPT, à partir d'une base de code initiale en Java. Notre mission consistait à appliquer des patrons de conception (design patterns) pour renforcer la maintenabilité et l'efficacité du code, ainsi que d'implémenter des tests automatiques pour garantir sa fiabilité.

Design Patterns:

MVC

Notre architecture MVC, intégrée avec le patron **Observateur**, est une structure sophistiquée pour l'application eliza-gpt. Dans cette architecture, le Modèle gère la logique métier et les données. Il comprend divers sous-systèmes comme le traitement des messages, la génération des réponses, et des fonctionnalités de recherche, chacun encapsulé dans des modules distincts pour une meilleure organisation et maintenabilité. La Vue, développée en JavaFX, est responsable de l'interface utilisateur, affichant les données et capturant les interactions des utilisateurs. Le Contrôleur agit comme un médiateur entre le modèle et la vue, traitant les entrées utilisateur et mettant à jour le modèle ou la vue en conséquence. Crucialement, dans notre mise en œuvre du patron Observateur, le contrôleur sert de sujet, notifiant les vues, qui sont les observateurs, des changements d'état. Cette intégration permet une mise à jour dynamique et réactive de l'interface utilisateur en fonction des changements dans le modèle. Cette combinaison de MVC et du patron Observateur assure que notre application est non seulement bien structurée, mais aussi efficace et réactive aux actions des utilisateurs.
Le diagramme de séquence suivant représente un scénario d'envoie d'un message dans notre architecture MVC.



Observer Pattern

Dans notre mise en œuvre, le patron Observateur est utilisé pour permettre une communication efficace et dynamique entre les composants de l'application, en particulier entre le Contrôleur (agissant comme Sujet) et les Vues (agissant comme Observateurs).

Subject(Classe Abstraite) : Le Sujet est une classe abstraite qui maintient une liste de ses dépendants, les observateurs. Elle fournit des méthodes pour attacher et détacher des observateurs (attach, detach) au sujet. Chaque fois qu'il y a un changement dans l'état du sujet, il notifie tous les observateurs attachés via la méthode notifyObservers.

Observateur (Interface) : Dans notre architecture, l'interface Observer a été enrichie avec des méthodes spécifiques pour chaque type de mise à jour, au lieu de se limiter à une simple méthode update. Cette conception permet d'éviter de charger tous les messages à chaque fois qu'une modification se produit. En définissant des méthodes distinctes telles que “onMessageAddUpdate”, “onDeleteUpdate”, “onSearchUpdate”, et “onUndoSearchUpdat”e, nous assurons que les observateurs réagissent uniquement aux types de mises à jour qui les concernent. Cette approche ciblée améliore

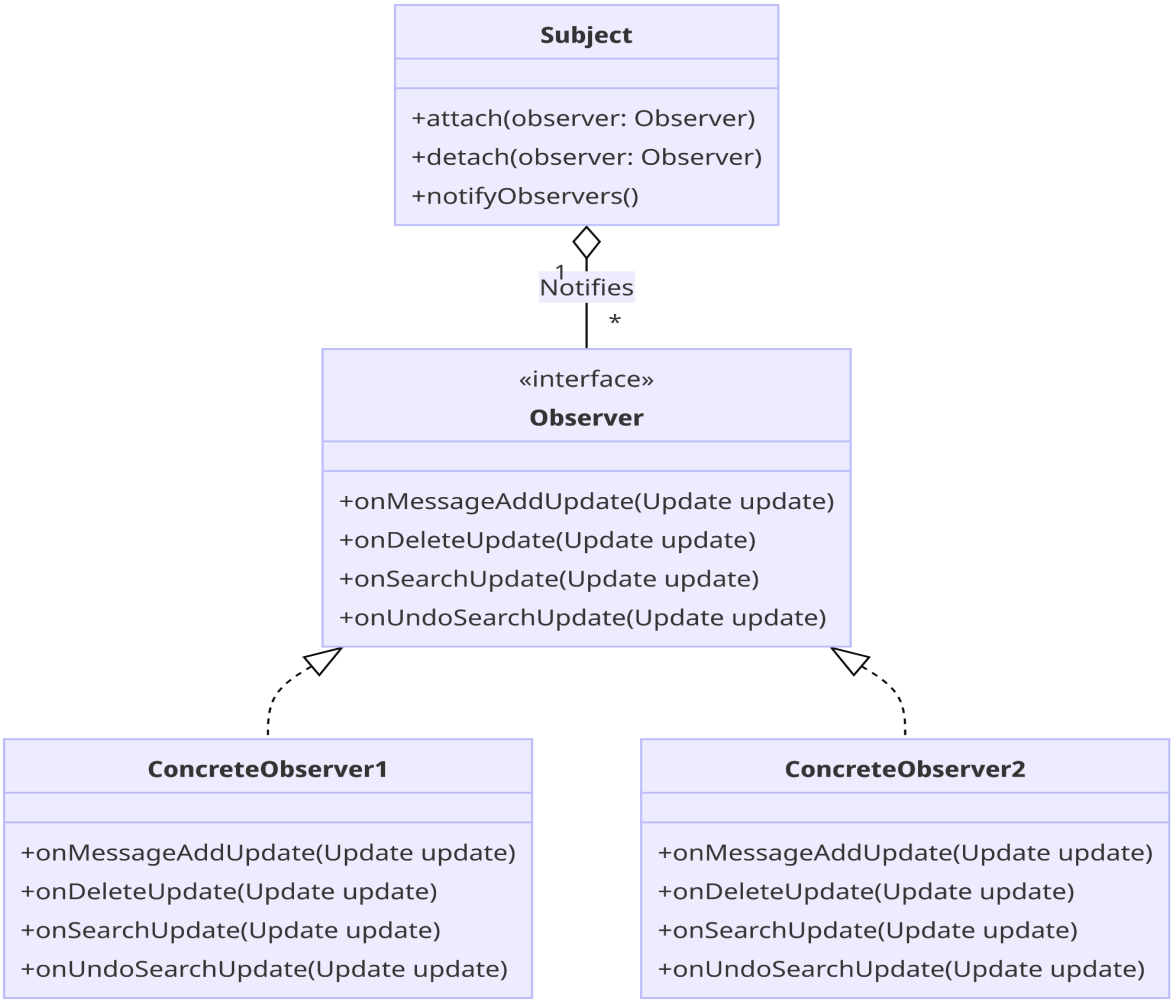
considérablement les performances et l'efficacité de l'application, car elle minimise le traitement inutile et optimise la réponse de l'interface utilisateur face aux changements d'état spécifiques.

Contrôleur en tant que Sujet : Dans notre architecture, le Contrôleur hérite de la classe Subject, assumant un rôle central dans le patron Observateur. En plus de gérer les entrées utilisateur et de mettre à jour le Modèle/Vue, le Contrôleur agit comme le principal point de notification pour les Vues. Cette décision de faire du Contrôleur, et non du Modèle, le Sujet est stratégique. Elle permet de maintenir une séparation claire entre la logique métier (Modèle) et la gestion des notifications (Contrôleur). Ainsi, le Modèle reste concentré sur la gestion des données et des opérations métier, tandis que le Contrôleur, en tant que Sujet, se charge de déterminer quand et comment les Vues doivent être informées des changements. Cela offre une meilleure modularité et flexibilité, car le Contrôleur peut choisir de notifier les Vues en fonction des contextes spécifiques, évitant ainsi une dépendance directe et une charge excessive sur le Modèle. En résumé, cette approche renforce la cohésion du Modèle tout en offrant une gestion efficace et centralisée des mises à jour de la Vue, ce qui améliore l'expérience utilisateur en garantissant que l'interface réagit de manière appropriée aux changements d'état.

Vue en tant qu'Observateur : Chaque Vue implémente l'interface Observer. Cette configuration permet à chaque Vue de mettre à jour son affichage ou son état en réponse aux notifications reçues du Contrôleur.

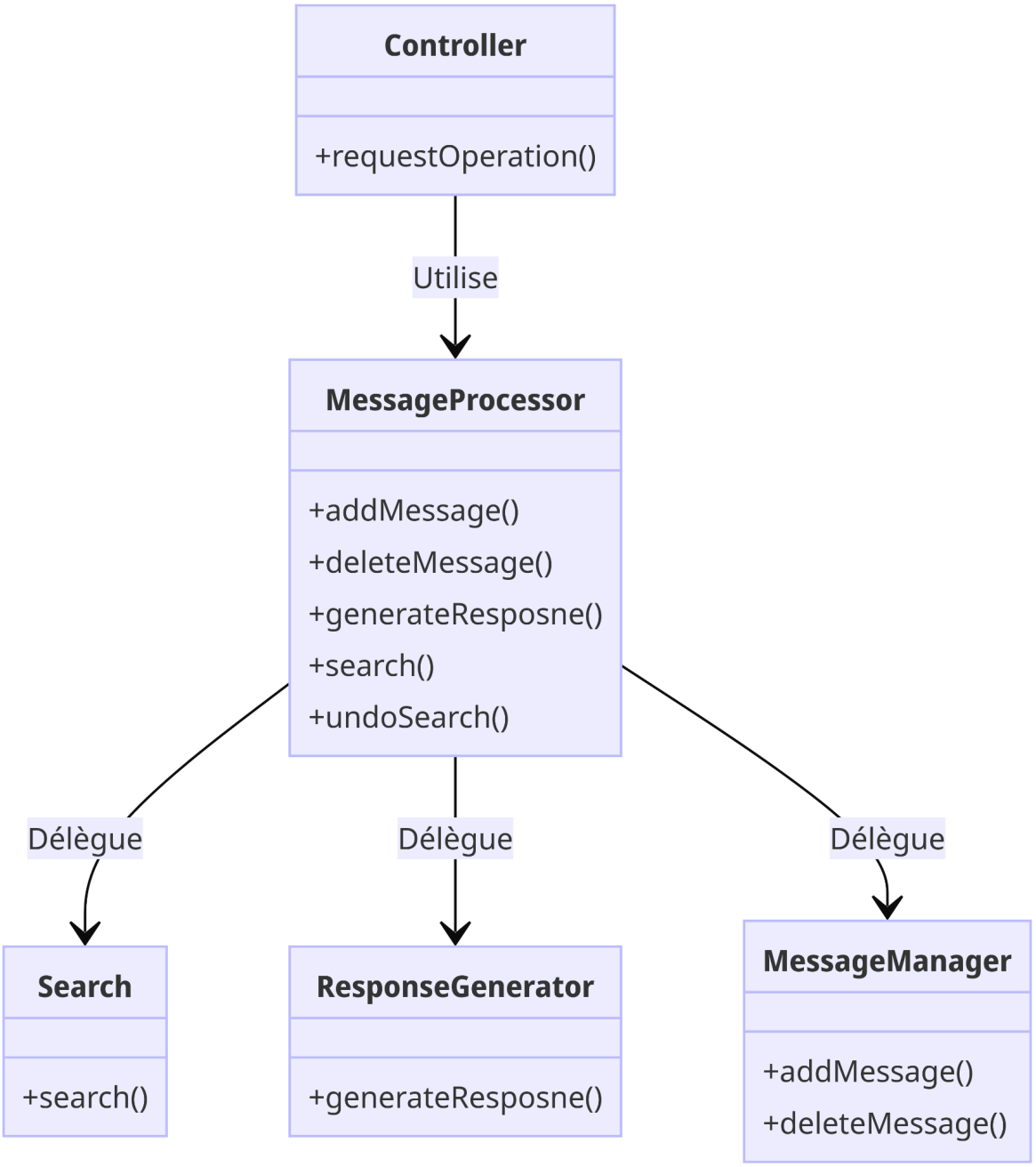
Cette conception permet un couplage lâche entre le Contrôleur et les Vues, rendant notre application plus flexible et plus facile à maintenir. Les changements dans l'état du Contrôleur sont efficacement propagés aux Vues sans que celles-ci aient besoin de vérifier constamment les changements. Ce patron est particulièrement efficace pour les applications avec plusieurs Vues qui doivent refléter les changements en temps réel.

Le diagramme de classe suivant illustre la logique du patron Observer



Facade Pattern

Le patron de conception Façade, tel que nous l'avons utilisé dans notre application avec la classe MessageProcessor, joue un rôle crucial en simplifiant les interactions complexes entre le contrôleur et les différents modèles.
Le patron Façade a pour but de fournir une interface simplifiée vers un ensemble de sous-systèmes ou de classes complexes. Il ne change pas les sous-systèmes eux-mêmes, mais offre une manière plus simple et plus directe de les utiliser. Cette simplification est bénéfique pour réduire la complexité de l'architecture et pour isoler les clients (dans ce cas, le contrôleur) de la multitude des interactions avec les classes du modèle.
Le diagramme de classe suivant illustre le patron Facade.



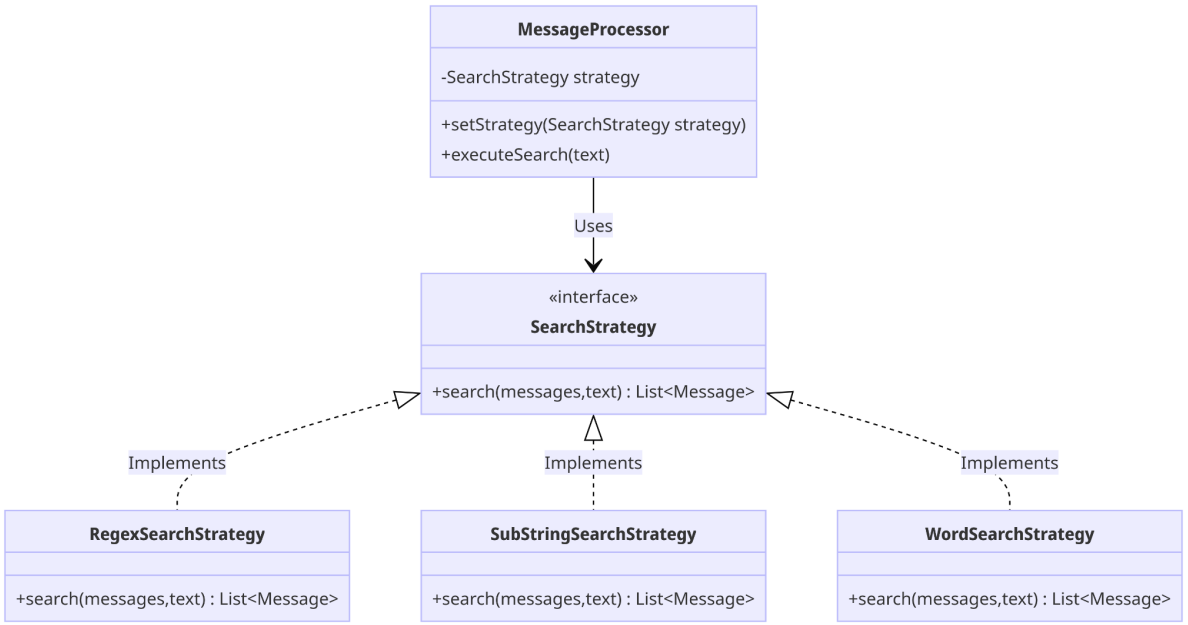
Strategy Pattern

Pour appliquer le patron strategy nous avons défini une interface **SearchStrategy** qui représente un contrat pour implémenter différentes stratégies de recherche de messages au sein du système de chatbot.
Elle déclare une méthode search(List<Message> messages, String text), qui est responsable de la recherche de messages contenant un texte spécifié dans une liste de messages.

Ensuite nous avons défini plusieurs stratégies de recherche concrètes tels que **RegexSearchStrategy** , **SubStringSearchStrategy** et **WordSearchStrategy** qui implémentent l'interface **SearchStrategy.**

Grâce à cette approche, notre application peut facilement changer de stratégie de recherche en fonction du contexte, sans avoir à modifier le code qui utilise ces stratégies. Cela rend notre modèle de recherche extrêmement flexible et adaptable à différents besoins de recherche.

Le diagramme de classe suivant illustre le patron Strategy.



Singleton Pattern

Le patron de conception Singleton est utilisé pour s'assurer qu'une classe n'a qu'une seule instance dans toute l'application et qu'il existe un point d'accès global à cette instance.

L'Application dans nos Stratégies de Recherche :

Instance Unique : Dans notre application, chaque stratégie de recherche, comme RegexSearchStrategy, SubStringSearchStrategy, et WordSearchStrategy, est implémentée comme une classe Singleton. Cela garantit qu'il n'existe qu'une seule instance de chaque stratégie dans l'application.

Accès Global : Chaque classe Singleton fournit une méthode statique (généralement nommée getInstance) qui permet d'accéder à son instance unique. Si l'instance n'existe pas encore, elle est créée lors du premier appel à cette méthode. Les appels ultérieurs retournent la même instance.

Avantages :

Contrôle de Ressources : Cela permet de contrôler l'utilisation des ressources, car la même instance est réutilisée, évitant ainsi l'allocation inutile de mémoire ou d'autres ressources.
Cohérence : Comme il n'y a qu'une seule instance de chaque stratégie, cela garantit une cohérence dans leur utilisation à travers l'application.
Performance : La réutilisation de l'instance unique améliore les performances en réduisant le besoin de création fréquente d'objets.
Utilisation dans le Modèle de Recherche :

Lorsque notre modèle de recherche a besoin d'utiliser une stratégie spécifique, il invoque la méthode getInstance sur la classe de stratégie appropriée. Cela assure que le modèle utilise toujours la même instance de la stratégie, maintenant la cohérence et optimisant l'utilisation des ressources.

En résumé, l'implémentation des stratégies de recherche en tant que classes Singleton dans notre application Eliza GPT offre un moyen efficace et optimisé de gérer les différentes méthodes de recherche, en assurant que ces stratégies sont gérées de manière cohérente et performante à travers l'application.

Chain of responsibility Pattern

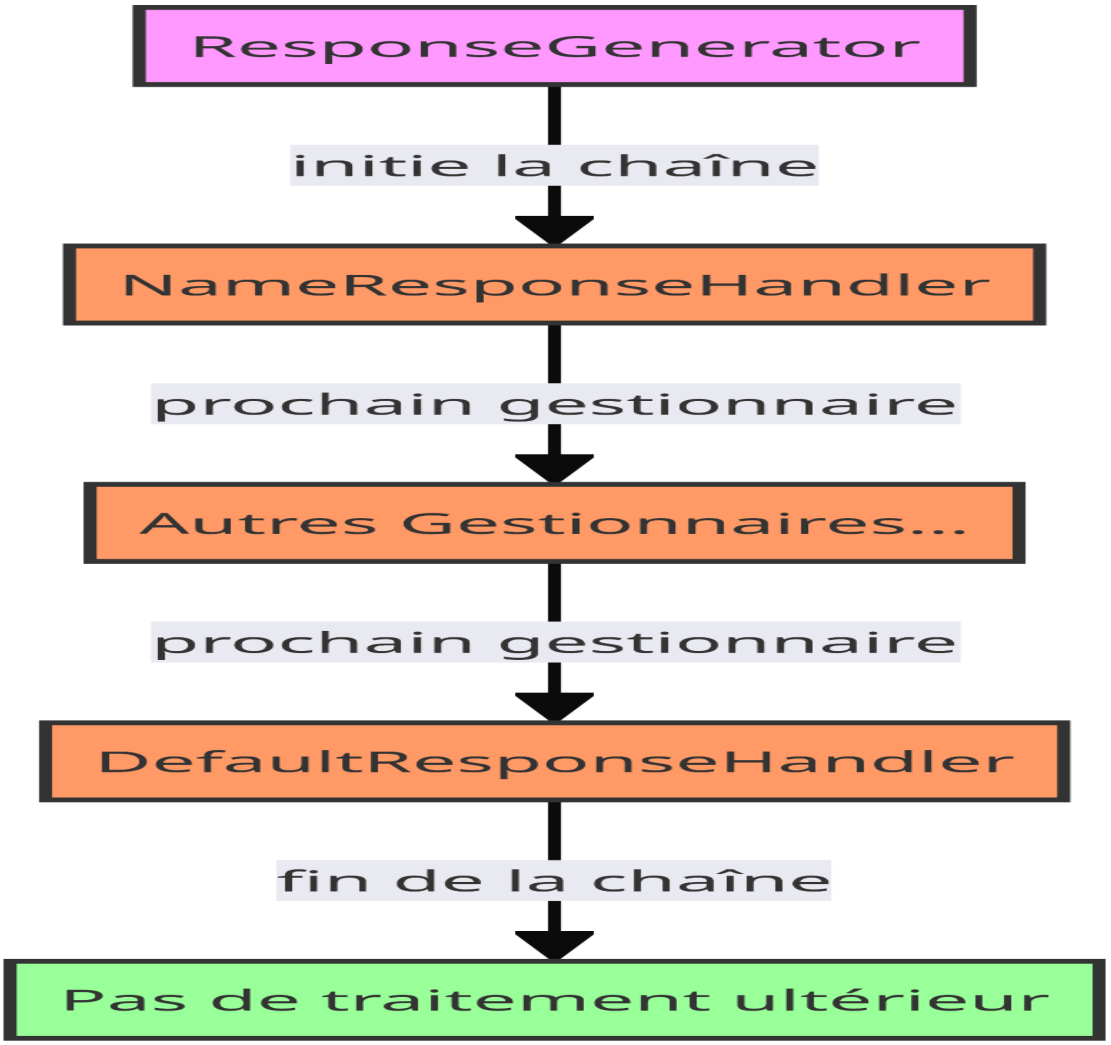
Pour implémenter ce pattern nous avons défini une interface ResponseHandler qui définit le contrat pour gérer les réponses dans Eliza GPT. Chaque implémentation de cette interface doit fournir une logique pour générer des réponses en fonction de l'entrée de l'utilisateur.
Elle inclut également une méthode **setNextHandler** pour définir le prochain gestionnaire dans la chaîne, permettant ainsi de former une chaîne de gestionnaires de réponses.

Gestionnaires de Réponses (Handlers) :
Des classes comme **NameResponseHandler**, **QuestionResponseHandler**, **DefaultResponseHandler** … etc implémentent ResponseHandler. Chaque classe est responsable de traiter un type spécifique d'entrée utilisateur.

Fonctionnement de la Chaîne :
Lorsqu'une requête est reçue, elle est d'abord traitée par le premier gestionnaire (**NameResponseHandler**, par exemple). Si ce gestionnaire ne peut pas traiter la requête, la requête est passée au gestionnaire suivant dans la chaîne.
Ce processus continue jusqu'à ce qu'un gestionnaire puisse traiter la requête ou jusqu'à ce que la requête atteigne le **DefaultResponseHandler**, qui sert de filet de sécurité.
ResponseGenerator :
Notre classe **ResponseGenerator** initialise et gère la chaîne de responsabilité. Elle configure la séquence des gestionnaires et déclenche le traitement de la réponse en passant le message de l'utilisateur au premier gestionnaire de la chaîne.
Avantages :
Cette approche permet une grande flexibilité dans la gestion des réponses. Les gestionnaires peuvent être ajoutés, retirés ou réorganisés facilement sans modifier le reste du code.
Elle favorise également le principe de responsabilité unique, chaque gestionnaire se concentrant sur un type spécifique de traitement de message.

En résumé, notre implémentation du patron Chaîne de Responsabilité dans la génération de réponses permet à l'application Eliza GPT de traiter divers types d'entrées utilisateur de manière organisée et extensible, en déléguant la responsabilité de la réponse à une série de gestionnaires spécialisés.

Le diagramme suivant illustre la logique du patron Chain Of Responsibility



Ⅲ. Ethique

Introduction :

Dans le paysage en constante évolution de l'éducation et de la recherche, les chatbots et les systèmes d'intelligence artificielle (IA) ont émergé comme des outils puissants. Cependant, leur intégration s'accompagne d'un ensemble de considérations éthiques qui demandent notre attention. Cet article explore les défis et les limites associés à l'utilisation éthique des chatbots dans l'éducation et la recherche.[1]
Corps de l'article :

1. Biais et Discrimination :

L'une des principales préoccupations éthiques tourne autour du potentiel de renforcement des biais et de la discrimination par les systèmes d'IA. Comme les chatbots reposent sur des algorithmes préprogrammés, ils peuvent involontairement perpétuer des biais existants dans les données qu'ils utilisent, impactant la précision et la fiabilité de leurs réponses.[1]

Par exemple, des tests réalisés par des informaticiens et documentés dans une récente publication ont révélé que les chatbots d'IA populaires présentent des biais politiques distincts. Selon ces résultats, ChatGPT d'OpenAI et son nouveau modèle GPT-4 ont été identifiés comme les chatbots les plus orientés politiquement à gauche, tandis que LLaMA de Meta penchait le plus à droite et était le plus autoritaire. Ces découvertes soulignent que les modèles linguistiques pré entraînés ont des inclinations politiques qui renforcent la polarisation présente dans les corpus d'entraînement, propageant ainsi des biais sociaux dans la prédiction de discours haineux et les détecteurs de désinformation.[3]

2. Manque de Transparence :

La transparence émerge comme une considération éthique cruciale. Les utilisateurs doivent être pleinement conscients des capacités du chatbot et des conséquences de leurs interactions. Des préoccupations de confidentialité surgissent car les données personnelles collectées pendant les conversations peuvent être sujettes à des abus, créant un déséquilibre éthique.[2]

3. Limitations de la Compréhension et de la Créativité :

Bien que les chatbots excellent dans l'analyse des données, ils pèchent par leur compréhension du contexte et leur capacité à générer des idées créatives, des éléments essentiels dans la recherche scientifique. Les chercheurs humains restent indispensables pour l'évaluation critique et l'interprétation des résultats, garantissant précision et normes éthiques.[1]
Selon un récent article de TechNews[4], ChatGPT représente une menace potentielle pour les systèmes éducatifs en raison de son accessibilité facile et de sa réactivité. Les étudiants l'utilisent de manière extensive, certains s'en remettant exclusivement à ChatGPT pour compléter leurs devoirs. Bien que l'article reconnaisse les avantages des chatbots, il souligne les impacts négatifs qui pourraient compromettre l'ensemble du cadre éducatif. Cela inclut la promotion de la malhonnêteté académique, la diminution des compétences de réflexion critique, la promotion de la paresse, l'impact sur la rétention de la mémoire, et l'aggravation de l'inégalité d'accès. La facilité et la prévalence de

l'utilisation de ChatGPT pourraient contribuer à une diminution de l'efficacité globale des systèmes éducatifs.[4]

4. Risque de Mauvais Usage :

Peut être le défi éthique le plus préoccupant est le risque de mauvais usage des chatbots. De la diffusion de désinformation à la manipulation de l'opinion publique, les chatbots peuvent être programmés pour servir des intérêts spécifiques, entraînant des conséquences néfastes. Ce mauvais usage s'étend à la recherche médicale, à la politique publique, à la recherche de marché et aux études académiques, impactant divers aspects de la société.[2]

Conclusion :

Au XXIe siècle, l'intégration des systèmes d'IA, y compris des chatbots, devient impérative. Cependant, il est crucial de reconnaître et de traiter leurs limites et leurs défis éthiques. La détection précoce et la compréhension de ces défis peuvent contribuer à atténuer les préjudices potentiels, en sauvegardant la confidentialité, la précision et l'équité dans l'éducation et la recherche.

Unit tests:

Nous avons réalisé des tests unitaires pour notre application en utilisant la bibliothèque Java JUnit. Cela impliquait de tester chaque fonction individuelle (unité) pour assurer leur exactitude et leur fiabilité. Pour améliorer notre approche de test, nous avons intégré la bibliothèque Mockito, nous permettant de créer des mocks pour les classes associées à nos fonctions. Cela garantit que nos tests unitaires restent efficaces même si les classes sous-jacentes subissent des changements ou rencontrent des erreurs.

Dans certains cas, nous avons jugé nécessaire de créer de nouveaux constructeurs pour certaines classes. Ces constructeurs acceptent des mocks comme arguments, permettant aux classes de les utiliser lors des tests. Dans notre fonction de configuration, exécutée avant chaque test, nous fournissons les instances de mocks aux constructeurs des classes que nous souhaitons tester.

Il est à noter que nous avons choisi de ne pas utiliser l'annotation spéciale Hamcrest, car nous avons jugé que les méthodes d'assertion fournies par JUnit (telles que assertEquals, assertTrue, assertSame) étaient suffisantes pour nos besoins de test.

Références:

[1]: Kooli, C. Chatbots in Education and Research: A Critical Examination of Ethical Implications and Solutions. *Sustainability* **2023**, *15*, 5614. https://doi.org/10.3390/su15075614
[2]: Ethics of Chatbots Codecademy Team https://www.codecademy.com/article/ethics-of-chatbots
[3]: You’re Not Imagining It: Top AI Chatbots Have Political Biases, Researchers Say
The peer-reviewed, award-winning paper says OpenAI’s ChatGPT leans left as Meta's LLaMA leans right. https://decrypt.co/151796/ai-political-bias-left-right-research

[4]: ChatGPT May Lead To The Downfall Of Education And Critical Thinking
https://www.techbusinessnews.com.au/blog/chatgpt-may-lead-to-the-downfall-of-eduction-and-critical-thinking/#:~:text=Diminishes%20critical%20thinking%3A%20By%20relying,their%20own%20ideas%20and%20perspectives.