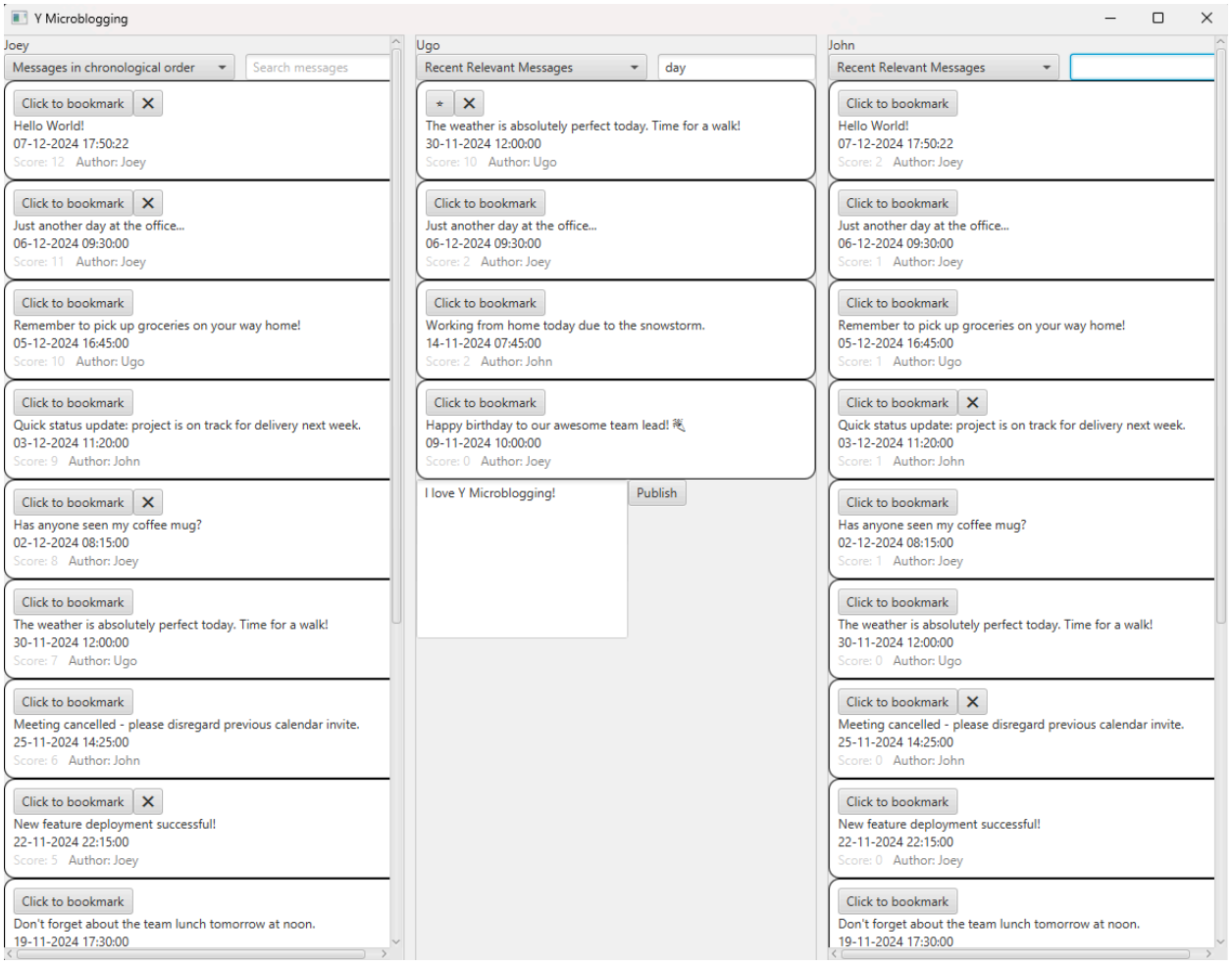
Y MICROBLOGGING

Mif01 - Gestion de projet et génie logiciel - rendu de projet



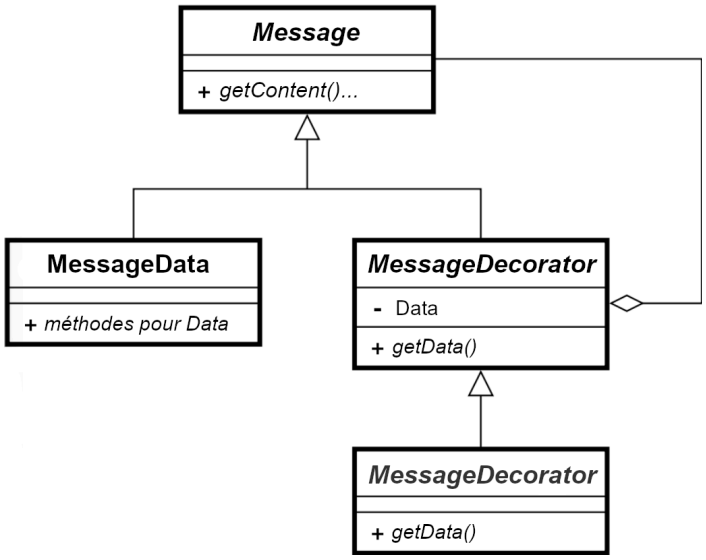
Ugo POUPON (p2020982), Joey DAVID (p2115771)

Présentation Globale du Projet

L'application Y microblogging cherche à répliquer une plateforme de messages instantanés accessibles à tous dans le style de X/Twitter. Son développement est un prétexte pour implémenter les concepts vus en cours de **mif01**, comprenant mais pas limités à l'*architecture modèle vue contrôleur*, aux *design patterns* et *principes grasp*, aux conventions de style, ainsi qu'aux tests automatiques. Nous sommes partis d'une base de code préexistante, que nous avons restructurée et sur laquelle nous avons implémenté différentes fonctionnalités au cours de 4 TP.

Design Patterns

1. *MessageDecorator* - Décorateur



La classe *MessageDecorator* nous permet d'étendre la définition de *Message* avec *MessageData*. On sépare désormais clairement les données du message (stockées dans l'objet *Message*) et ses métadonnées stockées dans *MessageData*, qui lui sont rattachées via *MessageDecorator*.

Ceci nous permet d'utiliser une liste de *MessageDecorators* dans Y au lieu d'une map de messages associés à leur *MessageData* respectifs. Elle respecte le principe de **single responsibility** en rendant plus nette la séparation entre le contenu du message et ses métadonnées.

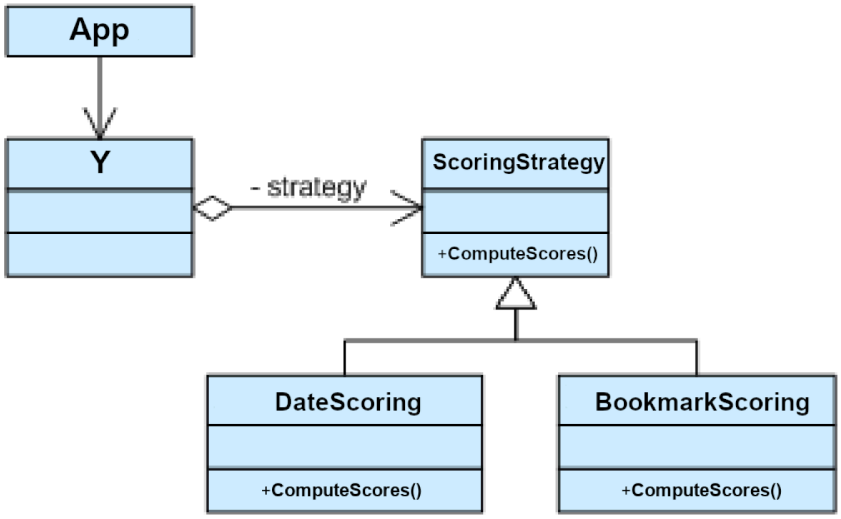
Cette structure facilite également l'évolution du système : si nous devons ajouter de nouvelles métadonnées (comme des tags ou des réactions), nous pouvons simplement étendre *MessageData* sans toucher à la logique de *Message* ou modifier l'interface existante de *MessageDecorator*.

2. *ScoringStrategy* - Stratégie

La classe *ScoringStrategy* implémente le pattern **Stratégie** pour encapsuler les différentes techniques de *scoring* des messages. Ceci est un choix judicieux, étant donné l'implémentation de diverses façons de noter les messages - on souhaite pouvoir ajouter ou enlever des méthodes de notation sans modifier la structure globale de l'application, selon les principes de faible couplage.

Cette interface définit donc une méthode *computeScores* qui prend une liste de MessageDecorator en paramètre. Chaque implémentation concrète (comme *DateScoring* et *BookmarkScoring*) fournit sa propre logique de calcul des scores. On peut modifier dynamiquement (pendant que l'application tourne) la stratégie de *scoring* via *setScoringStrategy*.

Cette approche respecte le principe **SOLID** de **Open-Closed** en permettant d'étendre les comportements de scoring sans modifier les classes existantes.



Note: dans ce diagramme, on a choisi de ne représenter que les méthodes de scoring via Date et Bookmark par souci de clarté, mais d'autres méthodes sont bien sûr implémentées dans la version finale de notre application.

3. MessageFactory - Fabrique

La Classe *MessageFactory* implémente le pattern **Fabrique**, pour centraliser et encapsuler la création des messages. Cette implémentation nous a notamment permis de rendre plus "propre" la création des messages existants au lancement de l'application, que nous conservons dans des fichiers *.txt* stockés dans src/main/resources/messages.

MessageFactory, qui comporte trois méthodes de création, permet de **découpler** la création des messages de leur utilisation dans la classe Y.

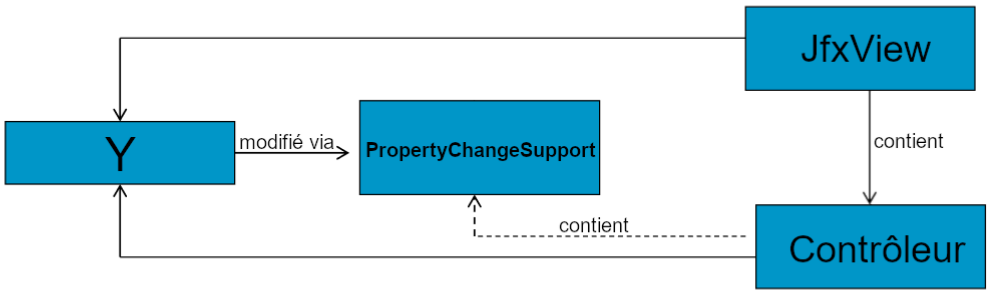
4. Observateur (comptabilisé dans MVC)

La classe *Y* implémente le pattern **Observateur** en utilisant un objet *PropertyChangeSupport pcs*, qui notifie une liste de *listeners* stockés en son sein. On utilise *addPropertyChangeListener* et *removePropertyChangeListener* pour y ajouter ou en retirer des éléments.

Lorsqu'un événement de notre choix se produit, on utilise la fonction membre

firePropertyChange pour avertir tous les listeners d'un changement. Cette fonction possède trois champs :

→ *propertyName* contient une string indiquant le type de changement
→ *oldValue* contient l'ancienne valeur modifiée (possiblement nulle)
→ *newValue* contient la nouvelle valeur déclarée (possiblement nulle)



Ce pattern permet de maintenir le **principe Grasp** du **faible couplage** - les composants intéressés par les changements d'état peuvent s'abonner aux notifications de Y sans avoir besoin d'être directement intégrés dans sa logique métier. Au passage, le **contrôleur** gère les listeners, ce qui va avec notre implémentation du **pattern MVC**.

Implémentations bonus

Nous avons implémenté une fonction de recherche pour chaque utilisateur, permettant aux usagers de rechercher des messages par mots contenus dans les messages. Nous avons aussi mis en place une configuration dynamique de l'application (cf *YConfiguration.java*) afin d'avoir des messages initiaux aux dates calculées dynamiquement à l'exécution, ce qui permet de montrer notre système de scoring en partie basé sur les dates.

Ethique

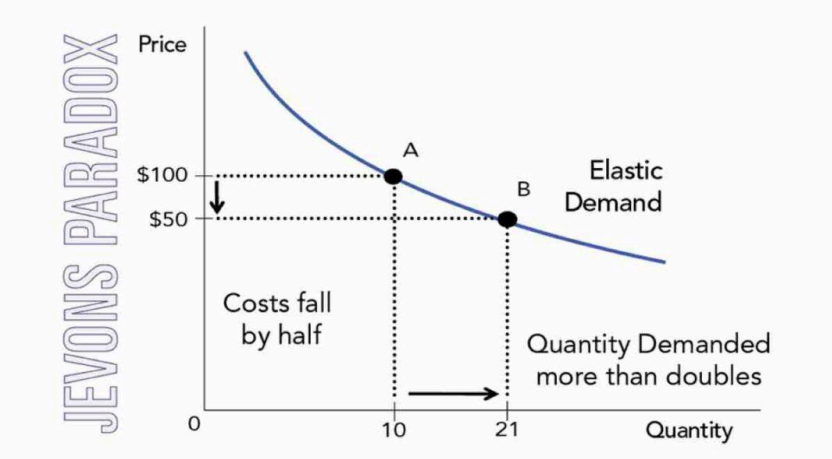
La plupart des questions éthiques soulevées par le développement d'un réseau social comme Y, le sujet de ce TP, ont déjà été explorées en profondeur - pollution via l'ouverture de datacenters massifs, facilitation du harcèlement en ligne, polarisation idéologique et *echo chambers* (PNAS - The Echo Chamber Effect), récolte abusive de données sur les utilisateurs...

Une conséquence nocive plus immédiate de ces réseaux, qui démultiplie l'ampleur des conséquences précédentes et qui affecte tout utilisateur d'un smartphone est leur aspect chronophage. Le smartphone a été un tremplin indiscutable à l'ubiquitisation d'internet et de la communication fractionnée : ce nouveau mode de communication basé sur l'envoi de courts messages / données - particulièrement bien représenté par les sms, mais aussi les réseaux

sociaux modernes (Murray State's Digital Commons- Smartphones and Scatterbrains).

Écrire une lettre ou passer un coup de téléphone nécessite que l'utilisateur dédie toute son attention à la communication pendant un intervalle de temps considérable, sans interruption. C'est une action à prévoir, parfois coûteuse en temps, en énergie et en concentration, et certainement pas quelque chose qui deviendrait un automatisme.

Les réseaux sociaux ont marqué l'avènement du format court : on peut désormais écrire ou lire un message en quelques secondes, regarder une courte vidéo de moins d'une minute en attendant son prochain arrêt de métro, envoyer un tweet sur son voisin bruyant à 3 heures du matin, sur un coup de tête. Puisqu'il est beaucoup plus facile de communiquer, d'échanger et de s'informer instantanément et presque sans effort, on aurait pu s'attendre à ce que l'on y dédie moins de temps et d'attention. Mais de façon peu surprenante, le paradoxe de Jevons (Jevon's Paradox - Wikipedia) a entraîné l'augmentation du temps et de l'énergie dédiés à ces vecteurs de communication et de consommation.



le paradoxe de Jevons

L'utilisation d'applications de communication instantanée et des réseaux sociaux finit souvent par remplacer le moindre temps mort du quotidien, et déborde souvent même sur les temps censés être productifs. Notamment par l'optimisation des interfaces pour stimuler le plus de dopamine possible, ils rendent leurs alternatives - passe-temps non digitaux, etc - désagréables en comparaison. Ils prennent une place de plus en plus significative dans la vie de chacun, ce qui décuple l'impact de leurs autres risques et conséquences adverses (Speaking of Psychology - Attention Spans).

Les plus gros réseaux sociaux sont conçus et optimisés continuellement pour inviter tous leurs utilisateurs à passer le plus de temps possible sur leurs plateformes. Ne pas passer plus qu'un temps modéré sur ces plateformes nécessitant une discipline non négligeable, le développement d'une application comme Y doit soulever beaucoup de questions du côté de ses créateurs. Son option de classification des messages par pertinence peut-elle aller trop loin, et polariser ses utilisateurs ? Comment attirer de l'audimat parmi les utilisateurs des grosses

plateformes déjà établies, sans copier leur modèle construit sur l'exploitation cognitive ? Peut-on vraiment être en compétition avec eux sans sacrifier l'aspect éthique de la conception du produit, et si oui, comment ?

Tests

L'essentiel des tests manuels que nous avons réalisés étaient ceux portant sur la vue. Nous avons en effet pu automatiser le reste des tests (sur le contrôleur, le tri et les méthodes de scoring) via l'écriture de tests avec junit.jupiter.api.

- **Affichage des messages** : vérification que les messages nouvellement postés par un utilisateur apparaissent instantanément dans les timelines de tous les autres utilisateurs sans interférence.
- **Bookmarks et recherches** : assurance que l'ajout ou la suppression de bookmarks, ainsi que l'utilisation de la barre de recherche par un utilisateur, n'influence en rien les vues ou les données des autres utilisateurs.
- **Gestion des suppressions** : contrôle que seul l'auteur peut supprimer ses propres messages et que cette suppression est correctement répercutée sur les pages des autres utilisateurs.
- **Visualisation propre aux stratégies de notation** : confirmation que le choix de la stratégie de scoring d'un utilisateur affecte uniquement sa vue, sans modifier les scores affichés pour d'autres utilisateurs.
- **Création automatique de messages** : nous avons mis en place un fichier de configuration gérant automatiquement la création de messages plus ou moins récents afin de démontrer la validité de nos stratégies de scoring. Nous avons vérifié que les messages étaient créés avec les bonnes dates et heures.

Ces tests manuels se sont concentrés sur les interactions visuelles et les comportements en temps réel de l'interface, car les tests automatiques couvrent déjà les autres aspects du système.