Design Pattern : Messagerie Collaborative

Creation

Factory

Factory est un patron de conception de création qui définit une interface (ensemble de méthodes) pour créer des objets dans une classe. Il permet de standardiser la création des certains objets dans l'ensemble de notre application. Il permet de restreindre la création des objets à son seul bénéfice pour éviter d'instancier certains objets à plusieurs endroits et de plusieurs manières ce qui rendrait la maintenabilité et l'évolutivité plus complexes.

De plus, en centralisant la création, cela permet de rajouter plus simplement un nouveau type de DAO à l'ensemble de l'application et de ce fait d'avoir à changer juste la DAO active pour changer de type de sauvegarde.

Nous avons ici 3 DAO différentes permettant chacune de sauvegarder en utilisant du json, du xml ou un système de base de données.

Singleton

Singleton est un patron de conception qui garantit que l'instance d'une classe n'existe qu'en un seul exemplaire, tout en fournissant un point d'accès global statique. Ici nous l'utilisons sur nos instances de connexions aux différents systèmes de sauvegarde pour nous assurer qu'il n'existe qu'une seule instance de connexion pour éviter de surcharger les différents systèmes.

Ce pattern nous permet également de venir faire en sorte d'instancier la DAO seulement quand l'on en a besoin et nous pourrions également nous assurer que seul le singleton puisse venir instancier nos différents objets pour que l'instanciation soit centraliser et de ce fait gagner en maintenabilité et en évolutivité.

Structure

Proxy

Le **proxy** est un patron de conception structurel qui vous permet d'utiliser un substitut pour un objet. Elle donne le contrôle sur l'objet original en ajoutant une couche de vérification supplémentaire.

Elle joue en quelque sorte le rôle de firewall dans notre cas nous permettant de ne pas charger la liste dans le cas où notre utilisateur serait dans un état absent. Elle permet également de ne pas recharger la liste si jamais aucune nouveauté n'a été notifiée par un observateur réseau (non représenté dans le schéma car la couche réseau n'était pas incluse dans le projet).

Dans ce cas, il nous permet aussi de venir contrôler le chargement pour le séquencer en fonction de la position du scroll de l'utilisateur et de ne pas charger tout d'un coup.

Comportement

Observer

L'Observer est un patron de conception qui permet de mettre en place un mécanisme de souscription dans des objets observés pour être notifié lors de l'apparition d'un certain évènement. Dans notre cas il est utile pour trois types d'actions:

- Notifier des différentes touches pouvant être pressé
- Notifier des différentes interactions avec l'interface.
- Notifier des différents changements dans la chatroom comme l'arrivée d'un utilisateur ou de nouveaux messages.

State

État est un patron de conception qui permet de modifier le comportement d'un objet lorsque son état interne change. L'objet donne l'impression qu'il change de classe.

Dans notre cas en fonction de l'état de présence (présent ou absent) cela nous permet de modifier le comportement d'envoi de message. Nous pourrions imaginer notifier l'utilisateur qu'il n'est pas possible pour lui d'envoyer des messages à cause de son statut, de lui proposer de changer de statut pour envoyer le message ou même de changer de statut à l'envoi d'un message directement.

Strategy

Strategy est un patron de conception qui permet de définir un contrat, qu'un ensemble de classes devra respecter et que nous pourrons donc manipuler comme étant des instances de cette interface. Elles seront donc interchangeables.

Ici l'ensemble des nos différents types de messages pourront être manipulés comme étant des iMessages et seront stockés dans une liste de IMessage pour former l'ensemble du chatMessage. L'interface déclarant les différentes méthodes que chaque classe qui l'implémentent devra avoir nous pourrons donc écrire le comportement d'affichage en fonction de classe enfant.

En exemple dans la classe de lien nous pourrons faire en sorte de rendre l'affichage cliquable, quand, dans la text nous rendrons l'affichage le plus simple possible mais toujours en manipulant dans la chatRoom des imessages.

Visitor

Visitor est un patron de conception qui nous permet de séparer les algorithmes et les objets sur lesquels ils opèrent. Ici cela nous permettra de venir appliquer différents filtres successivement sur notre chatMessage.

En effet le visitor(filtre) aura pour rôle de prendre le plaintext de notre chatMessage et de venir y effectuer tout un tas d'opérations (algorithmes) pour venir le séparer et le reconstruire sous forme d'une liste de classe concrète implémentant IMessage.

Chacun des filtres implémentant une même interface nous pourrons alors les lancer sur n'importe lequel de nos objets sans nous soucier de leur type concret mais en manipulant simplement des IchatMessageFilter.

Template Method

Patron de Méthode est un patron de conception qui permet de définir l'interface dans une classe mère en spécifiant des comportements par défauts, mais de laisser les sous-classes redéfinir ou non les différentes méthodes de l'interface.

Dans le cas de nos différentes DAO l'ensemble des comportements pourra être défini dans la classe mère DAO mais nous ne pourrons pas l'instancier, à la place nous allons la spécialiser dans des classes enfants et redéfinir des comportements au besoins pour traiter les problématiques liés à la spécialisation.

L'avantage est ici la factorisation de comportements ou d'attributs communs. Pour en profiter nos classes de DAO devront simplement étendre de la classe mère mais ce n'est pas le seul avantage car nous pourrons manipuler nos DAO spécialisés comme des DAOs génériques mais nous n'aurons alors accès qu'au méthode définie dans le classe mère mais avec le comportement défini dans les classes enfants.