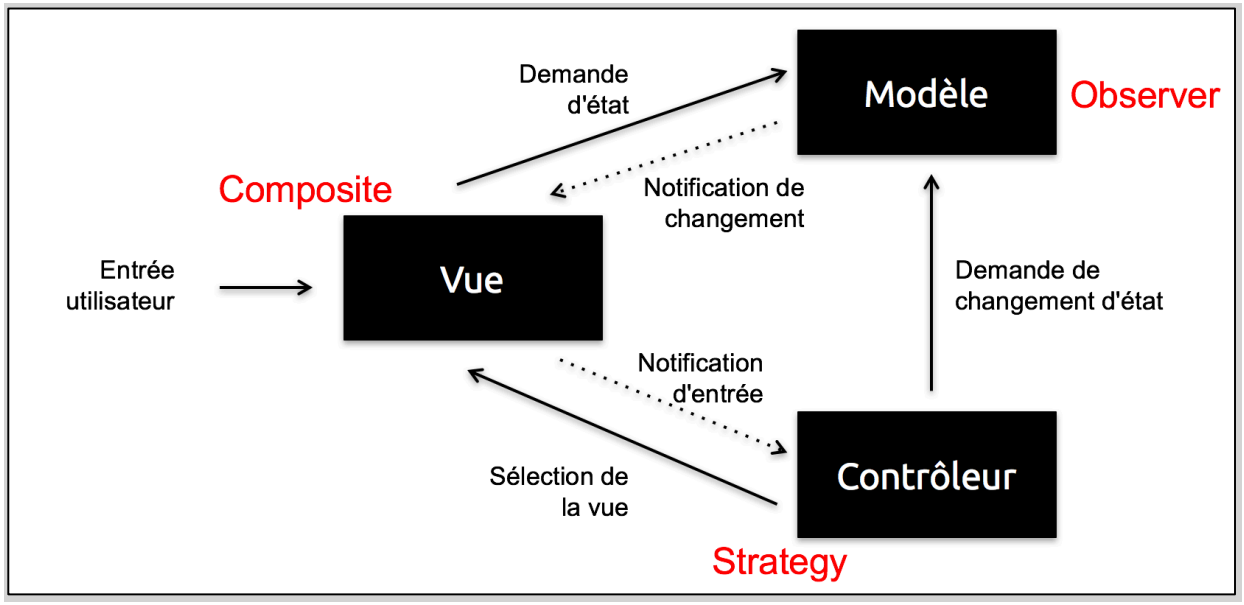
DESIGN PATTERNS

APPLICATION CHAT

L'objectif de cette session est de créer une application de chat (affichage console) en appliquant des design patterns :

- Composite
- Strategy
- Observer
- MVC
- Adapter

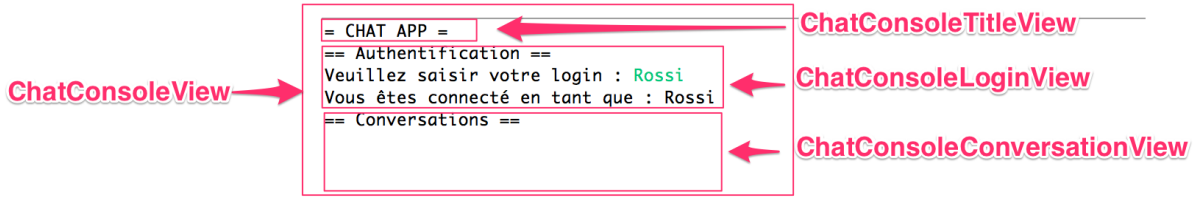
Présentation du pattern MVC



Etape 1 – Une vue (Pattern Composite)

Concentrons-nous dans un premier temps, sur la vue de l'application Chat.

Créer la vue « ChatConsoleView » composée de 3 autres vues comme l'illustre la figure suivante :

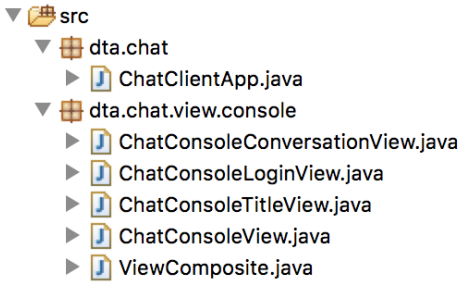


Appliquer le pattern Composite pour que tous les composants View aient une méthode « print() ».

L'appel « new ChatConsoleView().print() » provoque l'affichage complet.

Créer une application « ChatClientApp » qui permet d'afficher la vue « ChatConsoleView ».

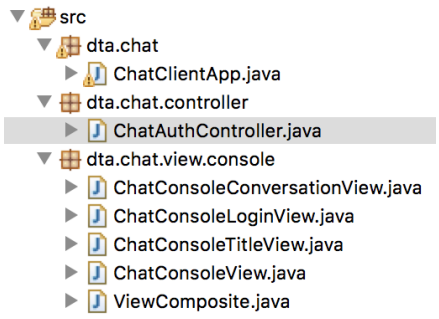
Les fichiers peuvent être organisés comme suit :



Etape 2 – Un contrôleur (Pattern Strategy)

L'objectif de cette étape est de créer un contrôleur qui va permettre de récupérer la saisie du login puis de la transmettre au composant « ChatConsoleConversationView ». Ce dernier pourra alors afficher un message de bienvenu.

Créer une interface contrôleur « ChatAuthController » qui possède une méthode d'authentification « authenticate ».




```
public interface ChatAuthController {
    void authenticate(String login);
}
```

Appliquer le pattern « Strategy » consiste à rendre l'implémentation du contrôleur paramétrable dans une vue (via par exemple un setter « setAuthController »).

Appliquer le pattern « Strategy » pour avoir le message de bienvenu dans le composant « ChatConsoleConversationView ».

Exemple d'utilisation globale :

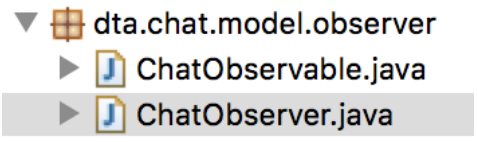
```
try(Scanner sc = new Scanner(System.in)) {

    final ChatConsoleView view = new ChatConsoleView(sc);
    view.setAuthController((login) -> {
        view.setLogin(login);
    });
    view.print();
}
```

Etape 3 – Un modèle (Pattern Observer)

- Créer une classe « dta.chat.model.ChatMessage » qui permet d'encapsuler les informations : login et text.

- Créer l'infrastructure permettant d'appliquer le pattern Observer :
    - Créer une interface « dta.chat.model.observer.ChatObserver<T> »
        - T correspond au type de l'objet de communication
        - Ajouter la méthode :
            - void update(ChatObservable<T> observable, T obj);

    - Créer une classe abstraite « dta.chat.model.observer.ChatObservable<T> »
        - Ajouter un champ List<ChatObserver<T>> observers.
        - Ajouter et implémenter les méthodes
            - public void addObserver(ChatObserver<T> observer)
            - public void removeObserver(ChatObserver<T> observer)
            - public void notifyObservers(T msg)



- Créer un modèle « dta.chat.model.ChatConversationModel »
    - Ce modèle est un observable (hérite de « ChatObservable »)
    - Y ajouter une méthode « public void setLogin(String login) »
        - Cette méthode notifie tous les observateurs.
    - Y ajouter une méthode « public void sendMessage(String msg) »
        - Cette méthode notifie tous les observateurs.

- Tester l'ensemble.
    - Exemple :

```
try(Scanner sc = new Scanner(System.in)) {
    ChatConversationModel model = new ChatConversationModel();
    final ChatConsoleView view = new ChatConsoleView(sc);
    view.setAuthController((login) -> {
        model.setLogin(login); // notifie les vues abonnées
    });

    model.addObserver(view);

    view.print();

    model.sendMessage("Bonjour");
    model.sendMessage("C'est moi !");

}
```

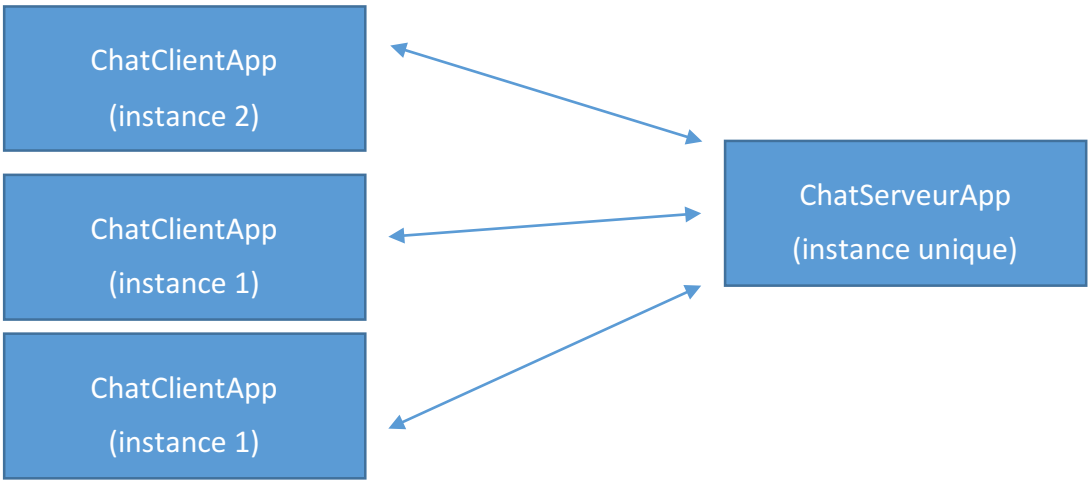
    - Résultat attendu :

```
= CHAT APP =
== Authentification ==
Veuillez saisir votre login : Rossi
Welcome : Rossi
== Conversations ==
Rossi : Bonjour
Rossi : C'est moi !
```

- Il est possible de créer une classe dédiée au contrôle de l'authentification qui implémente l'interface « ChatAuthController ». Implémenter cette variante.

Etape 4 – Communication Socket (Pattern Adapter)

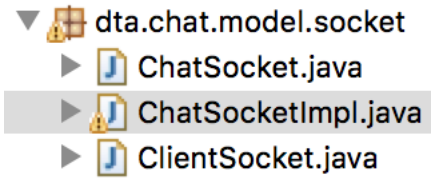
Cette consiste à finaliser l'application « Chat » qui permet de créer une discussion de groupe à travers le réseau. La communication se fera via le protocole TCP => architecture client-serveur.



- **Serveur Chat**
  - Intégrer la classe « ChatServerApp » dans votre projet.
  - Lancer ce serveur (le port de communication est configurable).

- **Client Socket**
  - Intégrer la classe « ClientSocket » qui est une classe utilitaire permettant de recevoir et d'envoyer un objet à travers le réseau.
  - Un exemple d'utilisation est présenté dans la méthode « main » de cette classe.
  - Tester la communication en procédant comme suit :
    - Lancer le serveur (ou vérifier qu'il est lancé).
    - Lancer le client.
    - Vérifier le message reçu dans la console du serveur

- **Pattern Adapter**
  - Appliquer le pattern Adapter pour que la communication soit utilisable via l'interface « ChatSocket »




```
public interface ChatSocket extends AutoCloseable {
    void sendMessage(ChatMessage msg) throws ChatClientException;
    ChatMessage readMessage() throws ChatClientException;;
}
```

- Modifier la méthode « main » de la classe « ClientSocket » pour utiliser l'interface ChatSocket et valider le fonctionnement.

- **Lecture du Chat**
    - Compléter l'application pour qu'elle puisse afficher les conversations du réseau.

Etape 5 – Envoyer un message

Donner la possibilité à un client de saisir un message et de l'envoyer aux autres clients.