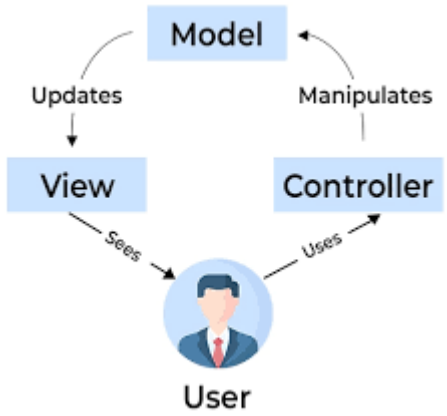
Il MVC (Model-View-Controller) Design Pattern, è un Pattern che divide il Software in tre componenti principali, facendo in modo che ogni parte del codice abbia una sola responsabilità e facilitando la manutenzione e l'estensione del codice.

Il Model è responsabile dei dati e delle operazioni su di essi, e non interagisce direttamente con la View, ad esempio in un centro di Adozione di Gatti, il Model potrebbe essere la lista di tutti i gatti presenti, con il loro nome, data di nascita e razza.

La View è la componente che si occupa di presentare i dati e le informazioni elaborate dal Model, aggiornandosi automaticamente quando i dati cambiano.

Il Controller fa da intermediario fra il Model e la View: può ricevere input utente dalla View e interagire con il Model per elaborare e recuperare nuove informazioni, successivamente aggiornando la View con i cambiamenti avvenuti.
Quindi il Controller fa interagire indirettamente la Model e la View; in un centro di Adozione di Gatti il Controller potrebbe essere la Richiesta di Adozione, che fa da intermediario fra la lista dei gatti presenti (Model) e la persona che li vuole adottare, ovvero l'user (View).



Il MVC Pattern rende quindi facile la manutenzione del codice, dato che si possono modificare le tre componenti singolarmente, senza dover modificare l'intero Software.

Per implementare il MVC Pattern in Java, è quasi fondamentale utilizzare l'Observer Pattern, che è un Pattern che forma una correlazione fra un insieme di oggetti (gli Observers) e il cambiamento di stato di un altro oggetto (l'Observable).

Possiamo fare l'esempio di un sistema di Notifica Automatica di un negozio online:
In questo caso, il catalogo dei prodotti è il Model (chiamiamolo StoreCatalogue), gli Utenti che si possono iscrivere al sistema di Notifica Automatica sono la View, (chiamiamoli User) e il Controller è ciò che viene notificato dallo StoreCatalogue quando c'è un nuovo prodotto, e ciò che di conseguenza notifica gli Utenti iscritti della novità (chiamiamolo StoreNotificationService).

Continua alla pagina seguente…

Incominciamo definendo un'Interfaccia NotificationService che delinea i metodi necessari di tutti i Sistemi di Notifica

```
public interface NotificationService {  no usages  1 implementation
    public void notifyall();  no usages  1 implementation
    public void subscribe(User user);  no usages  1 implementation
    public void unsubscribe(User user);  no usages  1 implementation
}
```

e anche un'Interfaccia User che delinea il metodo necessario update() di cui devono essere dotati tutti i tipi di Utenti, per poter essere aggiornati da un Sistema di Notifica.
(Noi non definiremo una Classe che implementa l'Interfaccia User, dato che per la spiegazione ci basta l'Interfaccia).

```
public interface User {
    public void update();  no usages
}
```

il metodo update() è il metodo che aggiornerà appunto la presentazione agli User, che sono la View.

Ora Implementiamo la Classe StoreNotificationService che implementa l'Interfaccia NotificationService, e sarà il nostro Controller.

```java
public class StoreNotificationService implements NotificationService {  no usages

    private List<User> subscribers;  4 usages

    public StoreNotificationService() {this.subscribers = new ArrayList<>();}  no usages

    @Override  1 usage
    public void notifyall() {
        subscribers.forEach(User::update);
    }
    @Override  no usages
    public void subscribe(User user) {
        subscribers.add(user);
    }
    @Override  no usages
    public void unsubscribe(User user) {
        subscribers.remove(user);
    }
}
```

Abbiamo una Lista contenente gli iscritti al Sistema di Notifica automatica, e i metodi subscribe() e unsubscribe() per iscrivere o rimuovere l'iscrizione ad un Utente.
E ovviamente abbiamo il metodo notifyall() che aggiornerà la view a tutti gli Utenti e verrà chiamato da StoreCatalogue (il Model) quando verrà aggiunto un nuovo prodotto al catalogo.

Implementiamo StoreCatalogue

```java
public class StoreCatalogue {  no usages

    private NotificationService notificationService;  2 usages
    private List<String> catalogue;  2 usages

    public StoreCatalogue() {  no usages
        catalogue = new ArrayList<>();
        notificationService = new StoreNotificationService();
    }

    public void addItem(String item) {  no usages
        catalogue.add(item);
        notificationService.notifyall();
    }
}
```

StoreCatalogue ha come campi: notificationService che è il suo Sistema di Notifica scelto, e catalogue che è il catalogo dei Prodotti disponibili.
Nel costruttore inizializziamo catalogue ad una ArrayList vuota e settiamo notificationService ad una nuova istanza di StoreNotificationService().

Vi è il metodo addItem() per aggiungere un nuovo prodotto al catalogo.

Quando viene aggiunto un nuovo prodotto al catalogo chiamando il metodo addItem(), il Sistema di Notifica scelto chiama il metodo notifyall(), che chiamerà il metodo update() di ogni Utente iscritto al Sistema di Notifica, aggiornandoli del cambiamento.

Abbiamo così implementato il MVC Pattern in Java utilizzando l'Observer Pattern:
Model: StoreCatalogue, elabora i dati aggiungendo prodotti al suo catalogo
View: User, riceve aggiornamenti quando il catalogo cambia
Controller: StoreNotificationService, fa da intermediario fra StoreCatalogue e gli User; viene informato quando viene aggiunto un nuovo prodotto al catalogo, e aggiorna gli User di conseguenza.