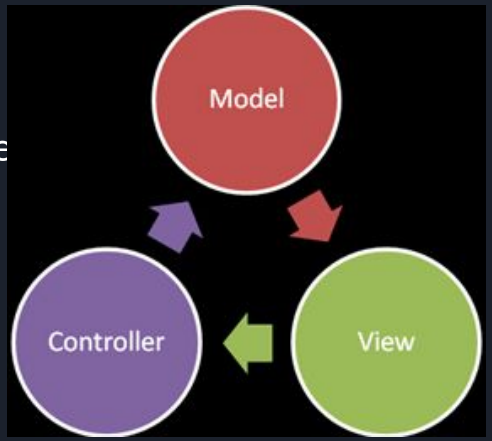
Model-View-Controller

MVC jest wzorcem projektowym, który polega na separacji odpowiedzialności za aplikację pomiędzy trzy jednostki; model, widok, kontroler.

Te trzy jednostki współpracują ze sobą w pełnej synergii by obsłużyć cały system wraz z jej logiką.

Żądanie / akcja użytkownika wprawia w ruch kontroler, który następnie je obsługuje w ścisłej współpracy z modelem by dostarczyć użytkownikowi odpowiedź w postaci odpowiednio zaprezentowanych danych, których z kole użytkownik oczekuje.



Model-View-Controller

MODEL

- Model jest komputerową reprezentacją rozpatrywanego problemu. Definiuje elementy, ich atrybuty i powiązania między nimi. Przykładowo, w aplikacji internetowej obsługującej sprzedaż biletów, model może określać reprezentację pojęć takich jak: bilet, wydarzenie, klient.

WIDOK

- Widok to część aplikacji odpowiedzialna za prezentację danych. Widoki mogą pobierać odpowiednie dane z modelu, lub modyfikować je. W odniesieniu do poprzedniego przykładu, w aplikacji rezerwującej bilety, widokiem może być strona WWW z formularzem rejestracyjnym dla klienta.

KONTROLER

- Rolą kontrolerów jest sterowanie aplikacją i komunikacja z użytkownikiem. Kontroler zwykle pośredniczy między przesyłem danych pomiędzy modelem, a widokiem. Kontroler odbierając informacje od użytkownika może zdecydować o dalszym działaniu aplikacji: wyświetlić widok, zmienić stan modelu, lub przesłać sterowanie do innego kontrolera.

Konsekwencje użycia

Zalety

- **Niezależność modelu** - model nie jest zależny od widoku i aplikacja może posiadać wiele niezależnych widoków dla tego samego modelu
- **Duża elastyczność widoków** - ze względu na na oddzielenie widoku od modelu, widoki mogą być modyfikowane częściej i niższym kosztem. Jest to szczególnie istotne, ponieważ w życiu systemu interfejs i warstwa prezentacji zmieniają się częściej niż logika biznesowa aplikacji

Wady

- **Złożoność** - architektura MVC nanosi dodatkową warstwę abstrakcji i nowe sposoby interakcji, co prowadzi do wzrostu jej złożoności. Zależność widoków od modeli i dodatkowa logika widoków czyni je szczególnie skomplikowane w testowaniu.
- **Mała elastyczność modelu** - modyfikacja modelu może w konsekwencji wymagać modyfikacji wielu widoków operujących na tym modelu (ponieważ model jest niezależny od widoku, ale widok zawsze opiera się na modelu)

Typy MVC

Typ I

W typie pierwszym, logika kontrolera jest zaimplementowana wewnątrz widoków. Przykładem jest JSF (Java servlet faces) - specjalny framework który umożliwia użytkownikowi osadzanie kodu Javy w dokumencie HTML

Typ II

W typie drugim, logika kontrolera zaimplementowana jest poza widokiem (w javie EE są to tak zwane servlety). Element widoku ma za zadanie jedynie renderowanie* danych przekazywanych przez servlety

*renderowanie - graficzne przedstawienie treści zapisanej cyfrowo, właściwe dla danego środowiska

przykład użycia logiki kontrolera wewnątrz widoku w JSP - typ I

```
<html>
  <head><title>Hello World</title></head>

  <body>
    Hello World!<br/>
    <%
      out.println("Your IP address is " +
request.getRemoteAddr());
    %>
  </body>
</html>
```

przykład użycia typu II - w widoku nie ma żadnej logiki kontrolera, jedynie renderowana jest wiadomość ${message} przekazywana z kontrolera (Spring MVC)

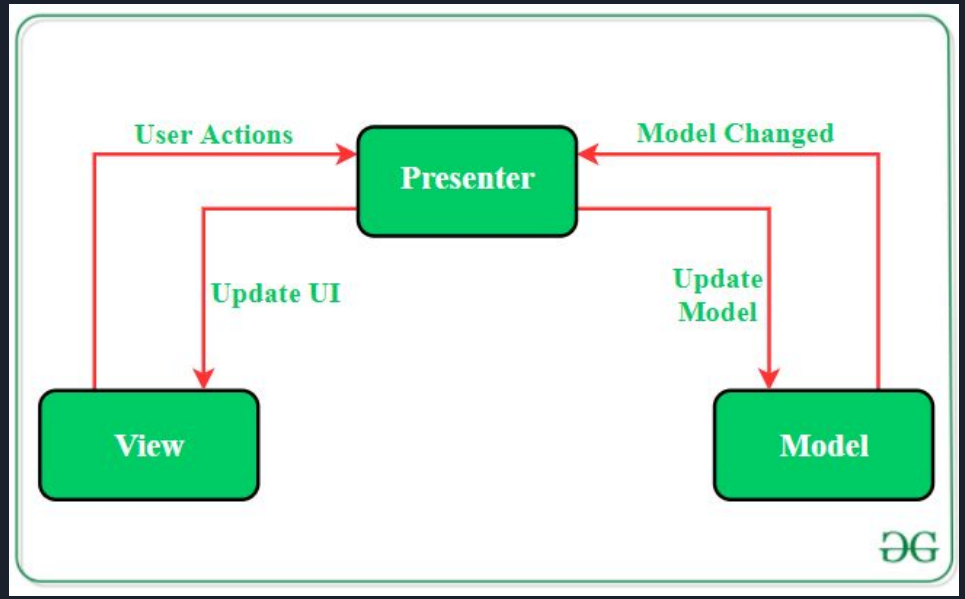
```
<html>
  <head>
    <title>Hello Spring MVC</title>
  </head>

  <body>
    <h2>${message}</h2>
  </body>
</html>
```

źródło: www.tutorialspoint.com

MVP - alternatywa dla MVC

MVP (Model View Presenter) - wzorzec projektowy będący alternatywą dla MVC, w którym całą komunikację między komponentami obsługuje komponent Presenter. Różni się on od MVC tym, że w jego obrębie nie występuje bezpośrednia zależność między Modelem i Widokiem



www.geeksforgeeks.org

Implementacja wzorca MVC

Polecenie jest behawioralnym wzorcem projektowym który zmienia żądanie w samodzielny obiekt zawierający wszystkie informacje o tym żądaniu. Taka transformacja pozwala na parametryzowanie metod przy użyciu różnych żądań. Oprócz tego umożliwia opóźnianie lub kolejkowanie wykonywania żądań oraz pozwala na cofanie operacji.

Kontroler oraz plik web.xml

```
package com.devchronicles.mvc.plain;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class FrontController extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)
                    throws ServletException, IOException {

     Action action =
          AbstractActionFactory.getInstance().getAction(request);
     String view = action.execute(request, response);
          getServletContext().getRequestDispatcher(view).forward(request,
          response);
     }

}
```

```
<servlet>
    <servlet-name>FrontController</servlet-name>
    <servlet-class>com.devchronicles.mvc.plain.FrontController</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>FrontController</servlet-name>
    <url-pattern>/users/*</url-pattern>
</servlet-mapping>
```

Klasa Factory

```
public class AbstractActionFactory {

    private final static ActionFactory instance = new ActionFactory();

    public static ActionFactory getInstance() {
        return instance;
    }
}
```

```
public class ActionFactory {

    private Map<String, Action> actions = new HashMap<>();
    private Action action;

    public ActionFactory() {
        actions.put("GET/users", new HomeAction());
        actions.put("GET/users/listusers", new ListUsersAction());
    }

    public synchronized Action getAction(HttpServletRequest request) {
        String path = request.getServletPath() + request.getPathInfo();
        String actionKey = request.getMethod() + path;
        System.out.println(actionKey);
        action = actions.get(actionKey);
        if(action == null){
            action = actions.get("GET/users");
        }

        return action;
    }
}
```

Klasa Action

```
public class ListUsersAction implements Action {
    public String execute(HttpServletRequest request, HttpServletResponse
                          response) {

        List<String> userList = new ArrayList<>();
        userList.add("John Lennon");
        userList.add("Ringo Starr");
        userList.add("Paul McCartney");
        userList.add("George Harrison");
        request.setAttribute("listusers", userList);
        return "/WEB-INF/pages/listusers.jsp";
    }
}
```

Implementacja MVC w Java EE

W celu implementacji modelu MVC w Jave'e EE niezbędne jest napisanie klas kontrolerów, które będą odpowiadać za przyjmowanie zapytań od klientów oraz zwracanie odpowiednich widoków. W najnowszych wersjach Javy EE ten schemat został uproszczony przez wprowadzenie FacesServlet, które zajmują się zarządzaniem zapytaniami oraz dostarczaniem widoków do klienta, dzięki temu sama implementacja MVC została mocno uproszczona i programista musi się skupić głównie na stworzeniu odpowiednich widoków oraz opracowaniem dobrze odwzorowanych modeli.

Przykład kontrolera w JAVE EE

Klasa kontrolera musi być oznaczona adnotacją “@Controller”. Kontroler może mieć domyślną ścieżkę zapytania zapisaną w adnotacji “@Path” nad klasą oraz tak samo domyślny widok w adnotacji “@View”.

Nad metodami odpowiadającymi za zarządzanie przychodzącymi zapytaniami też należy przypisać kilka adnotacji. Jedną z takich adnotacji jest adnotacja odpowiadająca za typ zapytania, na tym przykładzie użyto “@Get” obsługujące zapytanie typu GET.

Kolejne adnotacje możliwe do użycia to “@Path” oraz “@View”, które pozwalają na konfigurację zwracanego widoku oraz ścieżki zapytania dla danej metody.

Adnotacja “@Produces” służy do określenia jaki typ danych zostanie zwrócony do klienta.

```
@Controller
@Path("user")
public class UserController {
    @Inject
    private User user;
    @POST
    public String post() {
        user.setName("John Doe");
        return "redirect:/submit";
    }
    @GET
    public String get() {
        return "success.jsp";
    }
}
```

JAVA EE MVC - Model i widok

Widok

- Przesyłanym widokiem do klienta może być zwykły plik html lub wykorzystanie plików jsp. Strony jsp są to rozbudowane pliki html, pozwalają na używanie kodu java, daje to większe możliwości w zarządzaniu danymi po stronie widoku.

Model

- Modelem może być każda klasa dla której można stworzyć obiekt, najlepiej żeby było to zwykłe klasy POJO.

```java
public class UserDTO {
    private String username;
    private String password;

    public UserDTO(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public String getUsername() { return username; }

    public void setUsername(String username) { this.username = username; }

    public String getPassword() { return password; }

    public void setPassword(String password) { this.password = password; }
}
```

FacesServlet

- pełni rolę kontrolera
  - zarządza zadaniami użytkowników i dostarcza odpowiednie widoki
  - zarządza cyklem życia strony internetowej.
- używa specjalnego języka deklaracji widoku dla JSF - facelets
- konfiguracja zapisywana jest w pliku **faces-config.xml**, natomiast w wersji JSF 2.2+ doszły adnotacje
- bazowa konfiguracja jest bardzo uniwersalna - dopiero najbardziej zaawansowane aplikacje wymagają zmian w configu

Użycie FacesServlet w MVC

JSF wykorzystuje koncept **backing beans**

- klasa, która jest powiązana ściśle z komponentami na poszczególnych stronach.
- do jej utworzenia należy użyć adnotacji **@Named** oraz **@RequestScope**. Można je również zastąpić z pomocą adnotacji **@Model.**
- do **backing beans** można odwoływać się bezpośrednio z plików JSF.

```
<h:form>
    <h2>Click to see a
        <h:commandLink value="list of subjects"
                       action="#{listSubjectsAction.execute}"/>
    </h2>
</h:form>
```

Jak stworzyć przykładową stronę - deklaracja backing beans

```
@RequestScoped
@Named
public class ListSubjectsAction {
    private List<String> subjectList = new ArrayList<>();

    public List<String> getSubjectList() {
        return subjectList;
    }

    public String execute() {
        subjectList.add("math");
        subjectList.add("polish");
        subjectList.add("science");
        subjectList.add("physics");
        subjectList.add("it");
        return "/WEB-INF/pages/listsubjects.xhtml";
    }
}
```

Jak stworzyć przykładową stronę - index.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">

<h:head><title>Welcome</title></h:head>
<h:body>

    <h1>Welcome to our site</h1>
    <h:form>
        <h2>Click to see a
            <h:commandLink value="list of subjects"
                           action="#{listSubjectsAction.execute}"/>
        </h2>
    </h:form>
</h:body>
</html>
```

Jak stworzyć przykładową stronę - listsubjects.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html">

<head>
    <meta charset="UTF-8">
    <title>List of subjects</title>
</head>
<body>
<h1>Our users are:</h1>
<ui:repeat value="#{listSubjectsAction.subjectList}" var="listsubjects">
    <h:outputText value="#{listsubjects}"/>
    <br/>
</ui:repeat>
```

Kiedy i gdzie używać wzorca MVC

Wzorzec MVC najczęściej wykorzystywany jest w aplikacjach webowych, lecz można go używać wszędzie tam, gdzie istnieje korzyść z odseparowania widoku od logiki biznesowej aplikacji.
Wykorzystanie wzorca MVC w architekturze aplikacji internetowych jest wszechobecne w wielu implementacjach i projektach programistycznych.

Rozdzielenie aplikacji na architekturę „model-widok-kontroler” sprawia, że różne części aplikacji mogą być rozwijane praktycznie niezależnie od siebie. Przykładowo jeden zespół może pracować nad logiką wyświetlania (widok), podczas gdy inny zespół może pracować nad logiką biznesową i przekazywaniem informacji do widoku (model, kontroler).
Takie zastosowanie pozwala uzyskać większą wydajność w tworzeniu aplikacji oraz zwiększa kontrolę nad kodem, co niewątpliwie jest kluczową zaletą użycia wzorca MVC.

Podsumowanie

Wzorzec MVC posiada wiele różnych implementacji oraz punktów widzenia, lecz jego głównym założeniem jest oddzielenie widoku aplikacji od logiki zawartej w modelu.

Koncepcja separacji widoku od logiki polega na utrzymaniu wyraźnego podziału między obiektami modelującymi daną akcję użytkownika, a prezentacją wyników tej akcji na ekranie. Wynik może być wyświetlany użytkownikowi w dowolnej formie i dowolnym formacie, a nawet w odrębnym pliku możliwym do pobrania z serwera.

Jeśli zaimplementowany kod będzie opierał się na powyższym założeniu, to znaczy, że wzorzec został użyty poprawnie.

Dziękujemy za uwagę!

Autorzy:

Jakub Grzegorzak
Maciej Kapitan
Bartosz Krawczyk
Arkadiusz Pajor
Dawid Ropa
Urszula Wąsowska