Частное учреждение образования

«Колледж бизнеса и права»

УТВЕРЖДАЮ

Ведущий

методист колледжа

\_\_\_\_\_\_\_\_\_\_\_ Е.В. Паскал

«\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_202\_

|  |  |
| --- | --- |
| Специальность: 2-40 01 01 «Программное обеспечение информационных технологий» | Дисциплина: «Технология разработки программного обеспечения» |

**Лабораторная работа № 25**

**Инструкционно-технологическая карта**

Тема: Создание программной системы, использующей шаблон MVP

Цель: Научиться создавать программные системы, используя шаблон MVP

Время выполнения: 2 часа

**Контрольныевопросы**

1. Дайте определение понятию MVP.

2. Опишите назначение модели.

3. Опишите, в чем отличие Presenter от Controller.

4. Дайте определение понятию View (представление).

5. Опишите, как работает MVP.

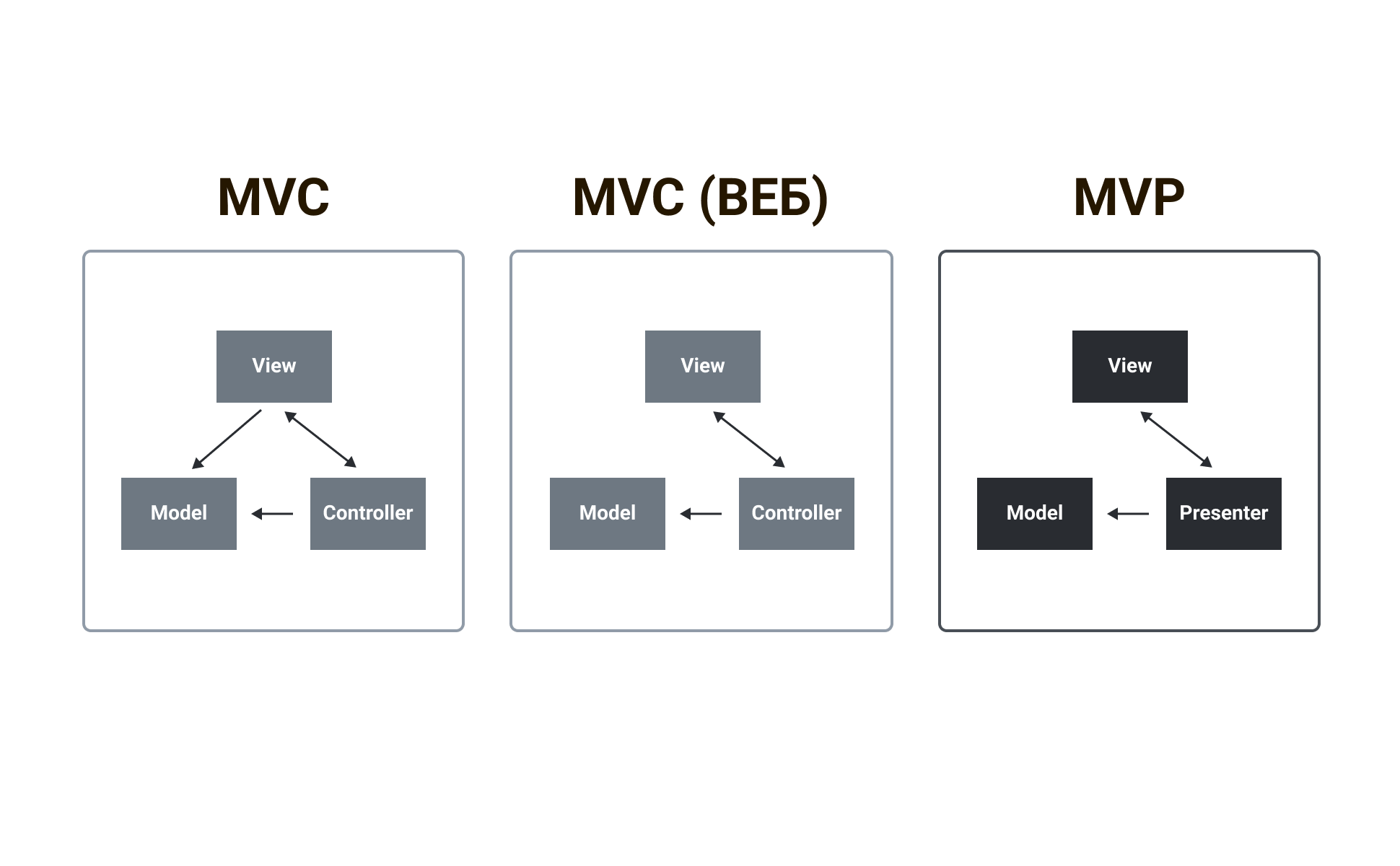
**Теоретические сведения для выполнения работы**

MVP – это паттерн программирования графических интерфейсов. В нём приложение делится на три компонента:

* Model (Модель) работает с данными, проводит вычисления и руководит всеми бизнес-процессами.
* View (Вид или представление) показывает пользователю интерфейс и данные из модели.
* Presenter (Представитель) служит прослойкой между моделью и видом.

Как и другие подобные паттерны (MVC, MVVM), MVP позволяет ускорить разработку и разделить ответственность разных специалистов; приложение удобнее тестировать и поддерживать.

Чаще всего его используют разработчики мобильных приложений, однако он гораздо популярнее, чем может показаться. Дело в том, что этот паттерн активно применяется в вебе, хотя там его и называют MVC:



Как работает MVP

На схеме выше видно, что приложение, созданное по принципам MVP, работает с помощью связи модели, вида и представителя. Происходит это так:

1. Вид строит интерфейс и добавляет в него данные из модели.
2. Пользователь видит информацию и взаимодействует с интерфейсом.
3. Вид перехватывает события и передаёт (делегирует) их представителю.
4. Представитель обрабатывает данные (не всегда) и передаёт их модели.
5. Модель выполняет какие-то операции и обновляется (меняет те или иные свойства).
6. Представитель получает обновлённую модель и передаёт её виду.
7. Вид строит интерфейс с новыми данными.

Основное отличие MVP и MVC в том, что в MVC обновлённая модель сама говорит виду, что нужно показать другие данные. Если же этого не происходит и приложению нужен посредник в виде представителя, то паттерн стоит называть MVP.

Всё это можно сравнить с работой издательства:

1. Автор готовит текст (модель).
2. Текст получает издатель (представитель).
3. Если с текстом всё в порядке, издатель передаёт его в отдел вёрстки (вид).
4. Верстальщики готовят книгу, которую начинают продавать читателям (пользователи).
5. Если пользователи как-то реагируют на книгу, например, пишут письма в издательство, то работа может начаться заново. Допустим, кто-то может заметить в книге неточность, тогда издатель передаст информацию автору, автор её обновит и так далее.

Конечно, это не точный алгоритм работы издательства, но для иллюстрации принципов MVP его достаточно.

Пример MVP-приложения

Так как MVP служит для упрощения разработки графических интерфейсов, рассмотреть его можно на примере WPF-приложения. В качестве вида будут выступать файлы MainWindow.xaml (разметка интерфейса) и MainWindow.xaml.cs (обработчик событий).

Начать можно с создания интерфейса авторизации:

<Border HorizontalAlignment="Center" VerticalAlignment="Center" Width="250">

<StackPanel>

<TextBlock Text="Login form" TextAlignment="Center"/>

<DockPanel LastChildFill="True">

<TextBlock Text="Login:"/>

<TextBox Name="LoginTextBox"/>

</DockPanel>

<DockPanel LastChildFill="True">

<TextBlock Text="Password:"/>

<PasswordBox Name="PassBox"/>

</DockPanel>

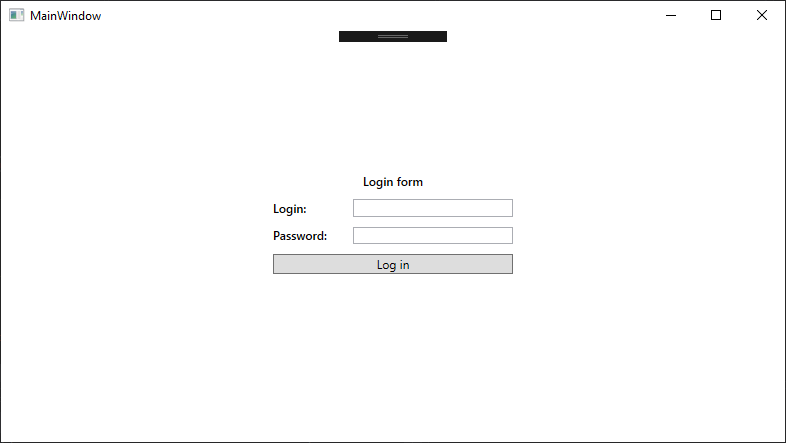
<Button Content="Log in" Name="LoginButton" Click="LoginButton\_Click" IsDefault="True"/>

<TextBlock Name="MessageBlock"/>

</StackPanel>

</Border>

Вот что должно получиться:



Теперь можно приступить к файлу MainWindow.xaml.cs:

public partial class MainWindow : Window

{

//Создание связи с другими компонентами

private Model model;

private Presenter presenter;

public MainWindow() // Инициализация компонентов

{

InitializeComponent();

model = new Model();

presenter = new Presenter(model);

}

private void LoginButton\_Click(object sender, RoutedEventArgs e) //Обработчик события нажатия на кнопку входа

{

this.model = this.presenter.Login(LoginTextBox.Text, PassBox.Password); //Отправка данных представителю и получение обновлённой модели

Update(); //Вызов метода обновления интерфейса

}

private void Update()

{

MessageBlock.Text = this.model.Message; //Вывод сообщения из модели

}

}

Тут уже заложена необходимая связь между компонентами. Дальше стоит рассмотреть код представителя:

public class Presenter

{

private Model model; //Связь с моделью

public Presenter(Model model)

{

this.model = model;

}

public Model Login(string login, string password) //Получение данных от вида

{

login = login.Trim(); //Обработка полученной информации

password = password.Trim();

this.model.Login(login, password); //Обновление модели

return this.model; //Передача обновлённой модели

}

}

Он получает данные, обрабатывает их и передаёт модели. Модель обновляется, а представитель возвращает её виду.

Вот что происходит в модели:

public class Model

{

private List<User> users;

private User loggedUser;

public Model()

{

users = new List<User>(); //Создание списка тестовых пользователей

users.Add(new User("Name1","Login1","password1"));

users.Add(new User("Name2", "Login2", "password2"));

users.Add(new User("Name3", "Login3", "password3"));

users.Add(new User("Name4", "Login4", "password4"));

loggedUser = null;

}

public void Login(string login, string password) //Метод авторизации

{

bool hasLogged = false;

foreach (User user in this.users)

{

if (user.Login == login && user.Password == password) //Поиск совпадений полученных данных с пользовательскими

{

this.loggedUser = user; //Обновление модели

hasLogged = true;

break;

}

}

if (!hasLogged)

{

this.loggedUser = null;

}

}

public string Message

{

get

{

return this.loggedUser != null ? $"Nice to see you {this.loggedUser.Name}!" : "Wrong login or password!"; //Вывод сообщения в зависимости от того, успешно ли был авторизован пользователь

}

}

}

Тут появляется новая сущность – User. Этот и подобные классы используются в качестве данных, с которыми работает модель. Таких классов может быть сколько угодно: работа с ними ведётся так же, как и вне паттерна MVP. Вот и сам класс User:

public class User

{

private string name;

private string login;

private string password;

public User(string name, string login, string password)

{

this.name = name;

this.login = login;

this.password = password;

}

public string Name

{

get

{

return this.name;

}

}

public string Login

{

get

{

return this.login;

}

}

public string Password

{

get

{

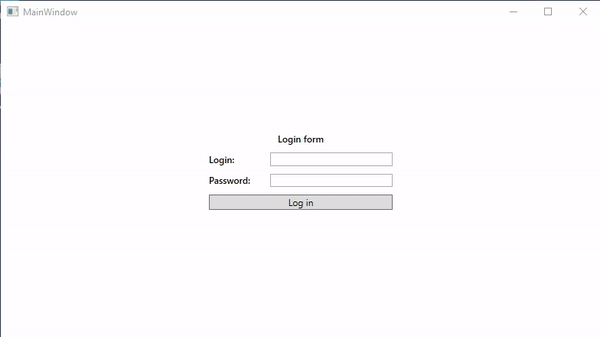
return this.password;

}

}

}

Теперь можно проверить, как это всё работает:



**Порядок выполнения работы**

1. Изучите теоретические сведения.

2. Реализовать программу использующую MVP по теме своего индивидуального задания, представленной в практической работе №1.

**Литература**

Орлов С.А Технологии разработки программного обеспечения: Учебник, 5-е изд. - СПб.: Питер, 2016 с. 245-250.

Преподаватель \_\_\_\_\_\_\_\_ А.С. Комаровский

Рассмотрено на заседании цикловой

комиссии ПОИТ №

Протокол № от «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 202\_

Председатель ЦК \_\_\_\_\_\_\_\_ В.Ю. Михалевич