

Лабораторная работа №7. MVX Паттерны.

Учитывая цель уменьшения трудозатрат на разработку сложного программного обеспечения, предположим, что необходимо использовать готовые унифицированные решения. Ведь шаблонность действий облегчает коммуникацию между разработчиками, позволяет ссылаться на известные конструкции, снижает количество ошибок.

Паттерн (англ. design pattern) — повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Начнем с первого главного – Model-View-Controller. MVC — это фундаментальный паттерн, который нашел применение во многих технологиях, дал развитие новым технологиям и каждый день облегчает жизнь разработчикам.

1. MVC (Model-View-Controller)

Основная идея этого паттерна в том, что и контроллер и представление зависят от модели, но модель никак не зависит от этих двух компонент. Впервые паттерн MVC появился в языке SmallTalk. Разработчики должны были придумать архитектурное решение, которое позволяло бы отделить графический интерфейс от бизнес логики, а бизнес логику от данных. Таким образом, в классическом варианте, MVC состоит из трех частей, которые и дали ему название. Рассмотрим их:

Описание:

MVC разделяет приложение на три компонента:

Model (Модель): отвечает за бизнес-логику и данные приложения.

Под Моделью, обычно понимается часть содержащая в себе функциональную бизнес-логику приложения. Модель должна быть полностью независима от остальных частей продукта. Модельный слой ничего не должен знать об элементах дизайна, и каким образом он будет

отображаться. Достигается результат, позволяющий менять представление данных, то как они отображаются, не трогая саму Модель.

Модель обладает следующими признаками:

- Модель — это бизнес-логика приложения;
- Модель обладает знаниями о себе самой и не знает о контроллерах и представлениях;
- Для некоторых проектов модель — это просто слой данных (DAO, база данных, XML-файл);
- Для других проектов модель — это менеджер базы данных, набор объектов или просто логика приложения;

View (Представление): отвечает за отображение данных пользователю.

В обязанности Представления входит отображение данных полученных от Модели. Однако, представление не может напрямую влиять на модель.

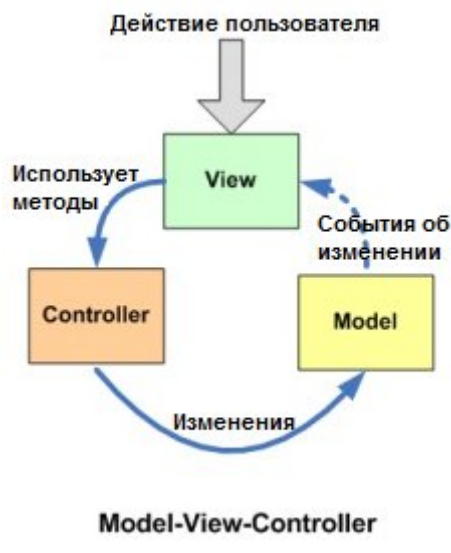
Можно говорить, что представление обладает доступом «только на чтение» к данным.

Представление обладает следующими признаками:

- В представлении реализуется отображение данных, которые получаются от модели любым способом;
- *В некоторых случаях, представление может иметь код, который реализует некоторую бизнес-логику.*

Примеры представления: HTML-страница, WPF форма, Windows Form.

Controller (Контроллер): посредник между Model и View, обрабатывает пользовательский ввод и управляет обновлением View.



Признаки контроллера

- Контроллер определяет, какое представление должно быть отображено в данный момент;
- События представления могут повлиять только на контроллер. Контроллер может повлиять на модель и определить другое представление.
- Возможно несколько представлений только для одного контроллера;

Контроллер перехватывает событие извне и в соответствии с заложенной в него логикой, реагирует на это событие изменяя Модель, посредством вызова соответствующего метода. После изменения Модель использует событие о том что она изменилась, и все подписанные на это события Представления, получив его, обращаются к Модели за обновленными данными, после чего их и отображают.

Преимущества:

- Четкое разделение ответственности.
- Облегчение тестирования.

Недостатки:

- При сложных интерфейсах Controller может перегружаться.

Пример:

Игровое приложение:

- Model — класс Player с полями health, score.
- View — интерфейс, отображающий здоровье и очки игрока.
- Controller — класс, обрабатывающий нажатия кнопок и обновляющий здоровье.

```
1  // Model
2  class Player {
3      private int health;
4
5      public Player(int health) {
6          this.health = health;
7      }
8
9      public int getHealth() {
10         return health;
11     }
12
13     public void takeDamage(int damage) {
14         health = Math.max(0, health - damage);
15     }
16
17     public void heal(int amount) {
18         health += amount;
19     }
20 }
```

```
22 // View
23 class PlayerView {
24     public void displayHealth(int health) {
25         System.out.println("Player's health: " + health);
26     }
27 }
28
```

```

29 // Controller
30 ▾ class PlayerController {
31     private Player model;
32     private PlayerView view;
33
34 ▾     public PlayerController(Player model, PlayerView view) {
35         this.model = model;
36         this.view = view;
37     }
38
39 ▾     public void takeDamage(int damage) {
40         model.takeDamage(damage);
41         updateView();
42     }
43
44 ▾     public void heal(int amount) {
45         model.heal(amount);
46         updateView();
47     }
48
49 ▾     public void updateView() {
50         view.displayHealth(model.getHealth());
51     }
52 }

```

```

54 // Main
55 ▾ public class Main {
56 ▾     public static void main(String[] args) {
57         Player player = new Player(100);
58         PlayerView view = new PlayerView();
59         PlayerController controller = new PlayerController(player, view);
60
61         controller.updateView();
62         controller.takeDamage(30);
63         controller.heal(20);
64     }
65 }
66

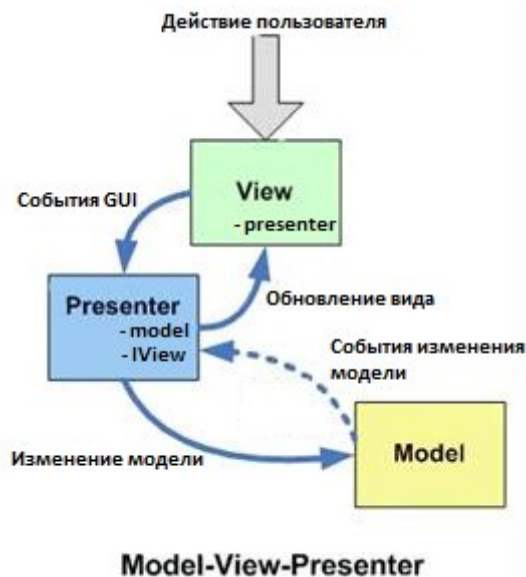
```

2. MVP (Model-View-Presenter)

Описание:

MVP также разделяет приложение на три компонента, но делает Presenter активным звеном:

- **Model (Модель):** хранит данные и логику.
- **View (Представление):** пассивный компонент, который только отображает данные, полученные от Presenter.
- **Presenter (Презентер):** активно управляет логикой, взаимодействует с Model и передает данные View.



Данный подход позволяет создавать абстракцию представления. Для этого необходимо выделить интерфейс представления с определенным набором свойств и методов. Презентер, в свою очередь, получает ссылку на реализацию интерфейса, подписывается на события представления и по запросу изменяет модель.

Признаки презентера:

- Двухсторонняя коммуникация с представлением;

- Представление взаимодействует напрямую с презентером, путем вызова соответствующих функций или событий экземпляра презентера;
- Презентер взаимодействует с View путем использования специального интерфейса, реализованного представлением;
- *Один экземпляр презентера связан с одним отображением.*

Каждое представление должно реализовывать соответствующий интерфейс. Интерфейс представления определяет набор функций и событий, необходимых для взаимодействия с пользователем (например, `IView.ShowErrorMessage(string msg)`). Презентер должен иметь ссылку на реализацию соответствующего интерфейса, которую обычно передают в конструкторе.

Логика представления должна иметь ссылку на экземпляр презентера. Все события представления передаются для обработки в презентер и практически никогда не обрабатываются логикой представления (в т.ч. создания других представлений).

Преимущества:

- Легкость тестирования Presenter.
- View становится максимально простым.

Недостатки:

- Presenter может стать сложным при масштабных проектах.

Пример:

Мобильное приложение:

- Model — класс `UserProfile` с именем и фото пользователя.
- View — экран, показывающий данные пользователя.

- Presenter — класс, загружающий данные из Model и передающий их в View.

```
1 // Model
2 class UserProfile {
3     private String name;
4     private String email;
5
6     public UserProfile(String name, String email) {
7         this.name = name;
8         this.email = email;
9     }
10
11     public String getName() {
12         return name;
13     }
14
15     public String getEmail() {
16         return email;
17     }
18 }
```

```
20 // View
21 interface UserProfileView {
22     void showUserName(String name);
23     void showUserEmail(String email);
24 }
25
```

```
26 // Presenter
27 class UserProfilePresenter {
28     private UserProfile model;
29     private UserProfileView view;
30
31     public UserProfilePresenter(UserProfile model, UserProfileView view) {
32         this.model = model;
33         this.view = view;
34     }
35
36     public void updateView() {
37         view.showUserName(model.getName());
38         view.showUserEmail(model.getEmail());
39     }
40 }
```

```

42 // Implementation of View
43 ▾ class UserProfileConsoleView implements UserProfileView {
44     @Override
45     ▾ public void showUserName(String name) {
46         System.out.println("User Name: " + name);
47     }
48
49     @Override
50     ▾ public void showUserEmail(String email) {
51         System.out.println("User Email: " + email);
52     }
53 }
54
55 // Main
56 ▾ public class Main {
57     ▾ public static void main(String[] args) {
58         UserProfile user = new UserProfile("Alice", "alice@example.com");
59         UserProfileView view = new UserProfileConsoleView();
60         UserProfilePresenter presenter = new UserProfilePresenter(user, view);
61
62         presenter.updateView();
63     }
64 }
65

```

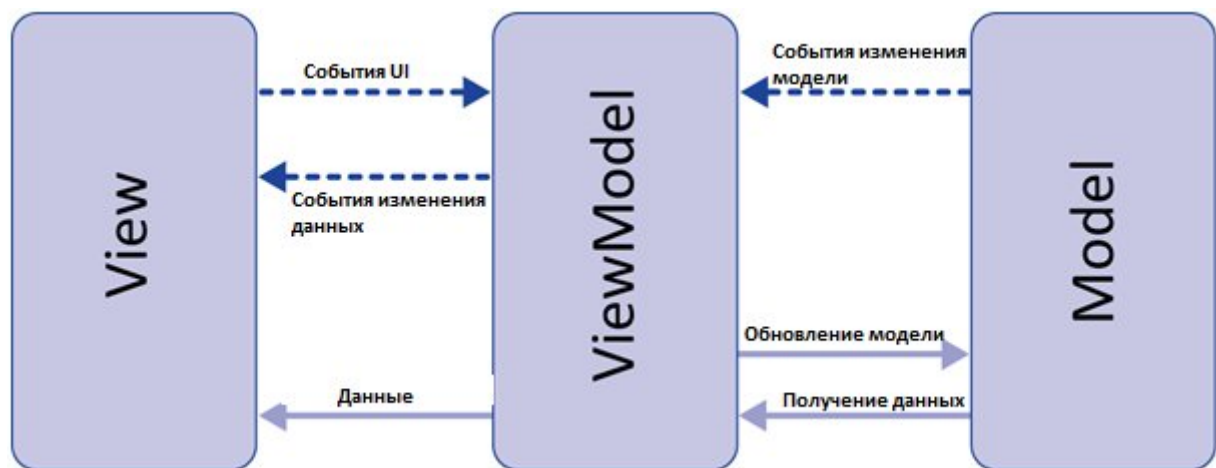
Дополнительно советую прочитать: <http://vector-sol.ru/Blog/8>

3. MVVM (Model-View-ViewModel)

Описание:

MVVM часто используется в приложениях с привязкой данных (например, WPF, Android):

- **Model (Модель):** отвечает за данные и их логику.
- **View (Представление):** отображает данные.
- **ViewModel:** связывает Model и View, содержит свойства, которые автоматически обновляют View благодаря механизму привязки данных.



Данный подход позволяет связывать элементы представления со свойствами и событиями View-модели. Можно утверждать, что каждый слой этого паттерна не знает о существовании другого слоя.

Признаки View-модели:

- Двухсторонняя коммуникация с представлением;
- View-модель — это абстракция представления. Обычно означает, что свойства представления совпадают со свойствами View-модели / модели
- View-модель не имеет ссылки на интерфейс представления (IView).
Изменение состояния View-модели автоматически изменяет

представление и наоборот, поскольку используется механизм связывания данных (Bindings)

- *Один экземпляр View-модели связан с одним отображением.*

При использовании этого паттерна, представление не реализует соответствующий интерфейс (IView).

Представление должно иметь ссылку на источник данных (DataContext), которым в данном случае является View-модель. Элементы представления связаны (Bind) с соответствующими свойствами и событиями View-модели. В свою очередь, View-модель реализует специальный интерфейс, который используется для автоматического обновления элементов представления. Примером такого интерфейса в WPF может быть INotifyPropertyChanged.

Преимущества:

- Удобство привязки данных.
- Хорошая поддержка двусторонней связи между View и ViewModel.

Недостатки:

- Может быть избыточным для простых приложений.
- Сложность тестирования ViewModel с привязкой данных.

Пример:

Приложение для задач:

- Model — класс Task с полями title и isCompleted.
- View — интерфейс, отображающий список задач.
- ViewModel — класс, содержащий свойства tasks, которые обновляются в View при изменении данных.

```

1 import java.util.Scanner;
2 import java.util.ArrayList;
3 import java.util.List;
4 // Model
5 class Task {
6     private String title;
7     private boolean completed;
8
9     public Task(String title, boolean completed) {
10         this.title = title;
11         this.completed = completed;
12     }
13
14     public String getTitle() {
15         return title;
16     }
17
18     public boolean isCompleted() {
19         return completed;
20     }
21
22     public void setCompleted(boolean completed) {
23         this.completed = completed;
24     }
25 }

```

```

27 // ViewModel
28 class TaskViewModel {
29     private List<Task> tasks;
30
31     public TaskViewModel() {
32         tasks = new ArrayList<>();
33     }
34
35     public void addTask(String title) {
36         tasks.add(new Task(title, false));
37     }
38
39     public List<String> getTaskTitles() {
40         List<String> titles = new ArrayList<>();
41         for (Task task : tasks) {
42             titles.add(task.getTitle() + (task.isCompleted() ? " (completed)" : ""));
43         }
44         return titles;
45     }
46
47     public void completeTask(int index) {
48         if (index >= 0 && index < tasks.size()) {
49             tasks.get(index).setCompleted(true);
50         }
51     }
52 }

```

```

54 // View
55 class TaskView {
56     public void displayTasks(List<String> tasks) {
57         for (int i = 0; i < tasks.size(); i++) {
58             System.out.println((i + 1) + ". " + tasks.get(i));
59         }
60     }
61 }

63 // Main
64 public class Main {
65     public static void main(String[] args) {
66         TaskViewModel viewModel = new TaskViewModel();
67         TaskView view = new TaskView();
68         Scanner scanner = new Scanner(System.in);
69
70         viewModel.addTask("Finish homework");
71         viewModel.addTask("Go to the gym");
72         viewModel.addTask("Read a book");
73
74         while (true) {
75             System.out.println("Tasks:");
76             view.displayTasks(viewModel.getTaskTitles());
77             System.out.println("Enter task number to complete (or 0 to exit):");
78             int choice = scanner.nextInt();
79
80             if (choice == 0) break;
81             viewModel.completeTask(choice - 1);
82         }
83     }
84 }

```

Сравнение паттернов

Паттерн	Особенности	Преимущества	Недостатки
MVC	View активно обновляет данные	Простота реализации	Controller может перегружаться
MVP	View пассивное, Presenter активное	Легкость тестирования	Сложность Presenter
MVVM	Привязка данных через ViewModel	Удобство в сложных интерфейсах	Сложность реализации

Задания по вариантам.

1. MVC (Model-View-Controller)

Тема: Создание простого приложения, где интерфейс пользователя обновляется в соответствии с изменением данных модели через контроллер. В главном классе необходимо протестировать обработку нескольких событий.

1. Учёт здоровья персонажа:

Создать класс **Player** (модель) с полями **health** (здоровье) и методами **takeDamage(damage)** и **heal(amount)**. Реализовать **PlayerView** для отображения здоровья и **PlayerController** для управления изменениями здоровья.

2. Калькулятор:

Создать класс **Calculator** (модель) с методами для базовых операций: **add**, **subtract**, **multiply**, **divide**. Реализовать **CalculatorView** для отображения результата и ввода данных. Создать **CalculatorController**, который обрабатывает пользовательский ввод и вызывает соответствующие методы модели.

3. Магазин оружия в игре:

Создать класс **Weapon** с полями **name** и **damage**. Реализовать модель списка оружия, вид для отображения доступных предметов и контроллер для добавления и удаления оружия.

4. Список задач:

Реализовать модель **Task** с полями **title** и **isCompleted**. Вид должен показывать задачи, а контроллер — добавлять, удалять и отмечать задачи как выполненные.

5. Трекер финансов:

Создать модель **Expense** с полями **amount** и **category**. Вид должен показывать общий расход и список категорий, а контроллер — добавлять новые расходы.

6. Управление инвентарём:

Создать модель **InventoryItem** с полями **name** и **quantity**. Вид должен отображать список предметов, а контроллер — добавлять, убирать или изменять количество предметов.

7. Учёт игровых достижений:

Реализовать модель **Achievement** с полями **name** и **isUnlocked**. Вид должен показывать список достижений, а контроллер — разблокировать их.

8. Музыкальный плеер:

Создать модель **Song** с полями **title**, **artist** и **isPlaying**. Вид отображает текущую песню, а контроллер переключает треки и управляет воспроизведением.

9. Погода в реальном времени:

Реализовать модель **WeatherData** с полями **temperature** и **humidity**. Вид отображает данные о погоде, а контроллер обновляет их.

10. Симуляция урона:

Создать модель **Enemy** с полями **health** и **name**. Вид отображает состояние врага, а контроллер наносит урон.

11. Трекер прогресса:

Модель **Progress** с полями **currentValue** и **maxValue**. Вид показывает шкалу прогресса, а контроллер увеличивает или сбрасывает прогресс.

12. Чат-приложение:

Модель **Message** с полями **sender** и **content**. Вид отображает сообщения, а контроллер добавляет их в список.

13. Таймер обратного отсчёта:

Модель **Timer** с полями **duration** и **isRunning**. Вид отображает оставшееся время, а контроллер запускает или останавливает таймер.

14. Управление транспортом:

Модель **Vehicle** с полями **type** и **status**. Вид отображает список транспорта, а контроллер изменяет статус транспорта (в пути/на стоянке).

15. Рейтинг игроков:

Модель **PlayerScore** с полями **name** и **score**. Вид отображает таблицу рейтингов, а контроллер добавляет новые результаты.

2. MVP (Model-View-Presenter)

Тема: Разработка приложений с разделением логики через Presenter. Интерфейс для View обязателен.

1. Калькулятор для обмена валют:

Создать модель **CurrencyConverter** с методом **convert(amount, fromCurrency, toCurrency)**. Интерфейс View должен позволять вводить данные (сумму и валюту) и отображать результат. Presenter будет обрабатывать ввод и запрашивать конвертацию.

2. Менеджер заказов:

Реализовать модель **Order** с полями **id**, **productName**, **quantity**. View предоставляет интерфейс для добавления/удаления заказов, а Presenter отвечает за взаимодействие с моделью.

3. Профиль пользователя:

Создать модель **UserProfile** с полями **name**, **email**, **age**. Интерфейс View включает формы для редактирования профиля. Presenter обновляет модель и уведомляет View.

4. Планировщик задач:

Модель **Task** с полями **title**, **dueDate**, **isCompleted**. View позволяет создавать задачи и отмечать их выполненными. Presenter управляет добавлением задач и обновлением состояния.

5. Бронирование билетов:

Реализовать модель **Ticket** с полями **id**, **destination**, **price**. View отображает доступные билеты, а Presenter обрабатывает бронирования и подтверждения.

6. Трекер шагов:

Модель **StepTracker** с полем **steps**. View показывает текущие шаги и позволяет сбрасывать счётчик. Presenter обрабатывает логику обновления и сброса.

7. Учёт расходов:

Модель **Expense** с полями **amount**, **category**, **date**. View отображает общий расход и категории, а Presenter управляет добавлением расходов.

8. Редактор заметок:

Модель **Note** с полями **title** и **content**. View позволяет создавать, редактировать и удалять заметки, а Presenter управляет действиями.

9. Интернет-магазин:

Модель **Product** с полями **name**, **price**, **stock**. View показывает список товаров и позволяет добавлять их в корзину. Presenter управляет списком и расчётом итоговой стоимости.

10. Учёт посещаемости студентов:

Модель **StudentAttendance** с полями **name**, **isPresent**. View отображает список студентов, а Presenter позволяет отмечать посещаемость.

11. Управление освещением в умном доме:

Модель **Light** с полями **roomName**, **isOn**. View позволяет включать/выключать свет в разных комнатах, а Presenter управляет состоянием света.

12. Трекер прогресса чтения:

Модель **Book** с полями **title**, **pagesRead**, **totalPages**. View отображает прогресс чтения книги. Presenter управляет обновлением прогресса.

13. Управление контактами:

Модель **Contact** с полями **name**, **phone**, **email**. View позволяет добавлять/редактировать контакты, а Presenter управляет действиями.

14. Игра «Счётчик очков»:

Модель **Score** с полями **playerName** и **points**. View отображает текущий счёт, а Presenter управляет увеличением и сбросом очков.

15. Менеджер файлов:

Модель **File** с полями **name**, **size**, **type**. View отображает список файлов, а Presenter управляет добавлением, удалением и поиском файлов.

3. MVVM (Model-View-ViewModel)

Тема: Создание приложений с использованием двустороннего биндинга (где возможно) или обновления View через ViewModel.

1. Калькулятор BMI:

Модель **BMI** с полями **height**, **weight** и методом **calculateBMI()**. View позволяет вводить данные и автоматически отображает результат через ViewModel.

2. Список покупок:

Модель **Item** с полями **name**, **quantity**, **isBought**. View показывает список покупок, а ViewModel обновляет их в реальном времени.

3. Учёт времени на задачи:

Модель **Task** с полями **title**, **duration**, **timeSpent**. View показывает прогресс выполнения задач, а ViewModel обновляет время в процессе выполнения.

4. Курсы валют:

Модель **CurrencyRate** с полями **currencyName**, **exchangeRate**. View показывает актуальные курсы валют, которые обновляются через ViewModel.

5. Учёт тренировок:

Модель **Workout** с полями **name**, **reps**, **completedReps**. View позволяет вводить количество повторений и отмечать завершённые, а ViewModel синхронизирует изменения.

6. Таймер обратного отсчёта:

Модель **CountdownTimer** с полями **duration**, **remainingTime**. View отображает оставшееся время, обновляемое ViewModel в реальном времени.

7. Музыкальный плейлист:

Модель **Track** с полями **title**, **artist**, **isPlaying**. ViewModel управляет переключением треков и синхронизирует их состояние с View.

8. Управление бюджетом:

Модель **Budget** с полями **income**, **expenses**, **balance**. ViewModel рассчитывает баланс в реальном времени и обновляет View.

9. Трекер калорий:

Модель **FoodItem** с полями **name**, **calories**. View показывает суммарное потребление калорий, синхронизируемое через ViewModel.

10. Магазин приложений:

Модель **App** с полями **name**, **rating**, **downloads**. ViewModel управляет фильтрацией и сортировкой приложений, отображаемых в View.

11. Редактор профиля пользователя:

Модель **User** с полями **name**, **email**, **phone**. ViewModel связывает данные с формой редактирования, обновляемой в реальном времени.

12. Прогноз погоды:

Модель **Weather** с полями **temperature**, **condition**, **location**. View показывает погоду, обновляемую ViewModel при смене местоположения.

13. Трекер сна:

Модель **SleepRecord** с полями **date**, **hoursSlept**. View отображает график сна, синхронизируемый ViewModel.

14. Чат в реальном времени:

Модель **Message** с полями **sender**, **content**, **timestamp**. ViewModel управляет добавлением сообщений в View при их поступлении.

15. Шахматная игра:

Модель **ChessBoard** с полями для представления позиций фигур. ViewModel управляет ходами и обновлением доски в View.