Архитектура приложения

Лекция 10: Структура и организация кода

Преподаватель: [Ваше имя]

Группа: 203

Семестр: Осенний 2024

План лекции

1. **Что такое архитектура приложения?**
2. **MVC паттерн**
3. **Разделение ответственности**
4. **Модульность и слои**
5. **Dependency Injection**
6. **Практический пример: Архитектура игры**

Что такое архитектура приложения?

Определение:

Архитектура приложения — это структура и организация компонентов программного обеспечения, определяющая их взаимодействие и отношения.

Цели:

- **Читаемость** и понимание кода
- **Поддерживаемость** и расширяемость
- **Тестируемость** компонентов
- **Переиспользование** кода
- **Масштабируемость** системы

Принципы хорошей архитектуры

SOLID принципы:

- **S** — Single Responsibility (единственная ответственность)
- **O** — Open/Closed (открыт для расширения, закрыт для изменения)
- **L** — Liskov Substitution (подстановка Лисков)
- **I** — Interface Segregation (разделение интерфейсов)
- **D** — Dependency Inversion (инверсия зависимостей)

Дополнительные принципы:

- **DRY** — Don't Repeat Yourself
- **KISS** — Keep It Simple, Stupid
- **YAGNI** — You Aren't Gonna Need It

MVC паттерн

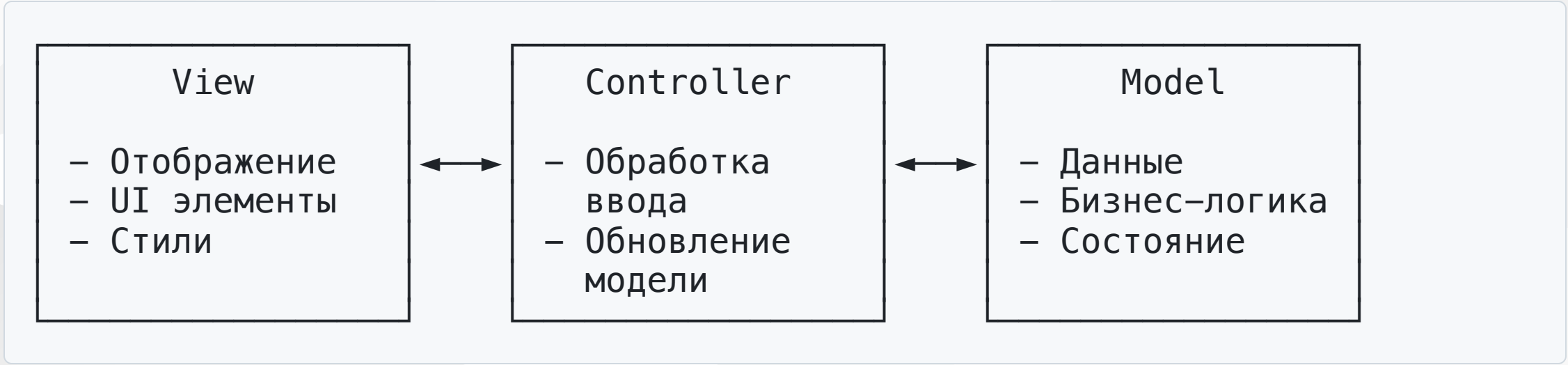
Что такое MVC?

MVC (Model-View-Controller) — архитектурный паттерн, разделяющий приложение на три основных компонента.

Компоненты:

- **Model** — данные и бизнес-логика
- **View** — представление данных пользователю
- **Controller** — обработка пользовательского ввода

Структура MVC



Взаимодействие:

1. **Пользователь** взаимодействует с View
2. **View** передает действия в Controller
3. **Controller** обновляет Model
4. **Model** уведомляет View об изменениях

Model (Модель)

Ответственность:

- **Хранение данных** приложения
- **Бизнес-логика** и правила
- **Валидация** данных
- **Уведомление** об изменениях

```
public class GameModel {
    private GameState gameState;
    private List<GameObserver> observers;

    public GameModel() {
        this.gameState = new GameState();
        this.observers = new ArrayList<>();
    }

    public void moveUnit(Unit unit, Position newPosition) {
        // Проверка возможности движения
        if (isValidMove(unit, newPosition)) {
            // Обновление состояния
            gameState.moveUnit(unit, newPosition);
```

View (Представление)

Ответственность:

- **Отображение данных** пользователю
- **UI элементы** и их стили
- **Обработка пользовательского ввода**
- **Обновление** при изменении данных

```
public class GameView implements GameObserver {
    private Stage primaryStage;
    private GridPane gameBoard;
    private VBox infoPanel;
    private VBox actionPanel;

    public GameView() {
        initializeUI();
    }

    private void initializeUI() {
        primaryStage = new Stage();
        primaryStage.setTitle("Пошаговая стратегия");

        BorderPane root = new BorderPane();

        // Создание игрового поля
        gameBoard = createGameBoard();
        root.setCenter(gameBoard);

        // Создание информационной панели
        infoPanel = createInfoPanel();
```

Controller (Контроллер)

Ответственность:

- **Обработка пользовательского ввода**
- **Координация** между Model и View
- **Валидация** входных данных
- **Вызов** методов модели

```
public class GameController {
    private GameModel model;
    private GameView view;

    public GameController(GameModel model, GameView view) {
        this.model = model;
        this.view = view;

        // Подписка на события модели
        model.addObserver(view);

        // Настройка обработчиков событий
        setupEventHandlers();
    }

    private void setupEventHandlers() {
        // Обработка кликов по игровому полю
        view.getGameBoard().setOnMouseClicked(e -> {
            Position clickedPosition = getPositionFromMouseEvent(e);
            handleCellClick(clickedPosition);
        });

        // Обработка кнопок действий
```

Разделение ответственности

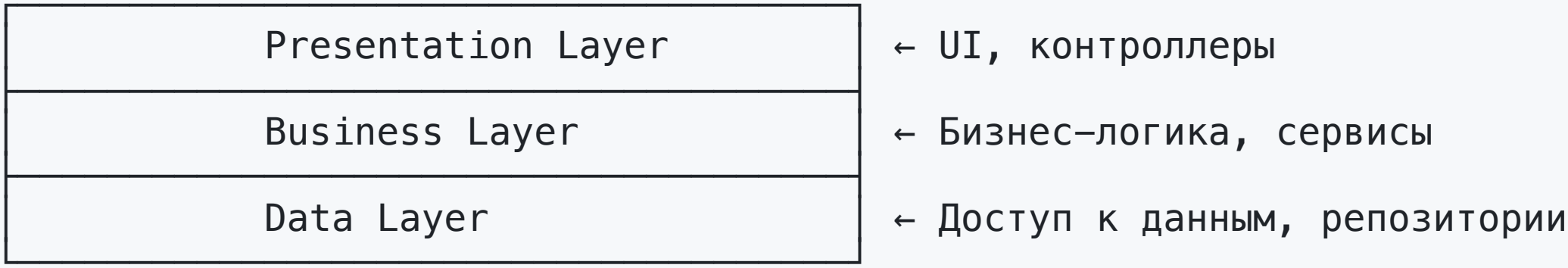
Принцип единственной ответственности:

Каждый класс должен иметь только одну причину для изменения.

```
// ❌ Плохо: класс делает слишком много
public class GameManager {
    public void moveUnit() { /* ... */ }
    public void attackUnit() { /* ... */ }
    public void saveGame() { /* ... */ }
    public void loadGame() { /* ... */ }
    public void renderUI() { /* ... */ }
    public void handleInput() { /* ... */ }
}

// ✅ Хорошо: каждый класс имеет одну ответственность
public class UnitMovementService {
    public void moveUnit(Unit unit, Position newPosition) { /* ... */ }
}

public class CombatService {
    public void attackUnit(Unit attacker, Unit target) { /* ... */ }
}

public class SaveLoadService {
    public void saveGame(GameState state, String filename) { /* ... */ }
    public GameState loadGame(String filename) { /* ... */ }
}
```

Модульность и слои

Архитектура по слоям:



Преимущества:

- **Изоляция** изменений
- **Тестируемость** компонентов
- **Переиспользование** кода

Пример слоистой архитектуры

```
// Presentation Layer
public class GameUI {
    private GameController controller;

    public void showGameBoard(GameState state) {
        // Отображение игрового поля
    }

    public void showUnitInfo(Unit unit) {
        // Отображение информации о юните
    }
}

// Business Layer
public class GameService {
    private UnitRepository unitRepository;
    private CombatService combatService;

    public void moveUnit(int unitId, Position newPosition) {
        Unit unit = unitRepository.findById(unitId);
        if (unit != null && isValidMove(unit, newPosition)) {
            unit.setPosition(newPosition);
            unitRepository.save(unit);
        }
    }

    public void attackUnit(int attackerId, int targetId) {
        Unit attacker = unitRepository.findById(attackerId);
        Unit target = unitRepository.findById(targetId);

        if (attacker != null && target != null) {
            combatService.performAttack(attacker, target);
            unitRepository.save(attacker);
            unitRepository.save(target);
        }
    }
}

// Data Layer
public class UnitRepository {
    private List<Unit> units = new ArrayList<>();

    public Unit findById(int id) {
        return units.stream()
            .filter(unit -> unit.getId() == id)
            .findFirst()
            .orElse(null);
    }

    public void save(Unit unit) {
        // Сохранение юнита
    }
}
```

Dependency Injection

Что такое DI?

Dependency Injection — паттерн, при котором зависимости объекта передаются извне, а не создаются внутри объекта.

Преимущества:

- **Слабая связанность** между компонентами
- **Легкое тестирование** с mock объектами
- **Гибкость** в конфигурации
- **Переиспользование** компонентов

Реализация Dependency Injection

```
// Интерфейсы для сервисов
public interface UnitService {
    void moveUnit(Unit unit, Position newPosition);
    void attackUnit(Unit attacker, Unit target);
}

public interface SaveLoadService {
    void saveGame(GameState state, String filename);
    GameState loadGame(String filename);
}

// Реализации сервисов
public class UnitServiceImpl implements UnitService {
    @Override
    public void moveUnit(Unit unit, Position newPosition) {
        // Реализация движения
    }

    @Override
    public void attackUnit(Unit attacker, Unit target) {
        // Реализация атаки
    }
}

public class SaveLoadServiceImpl implements SaveLoadService {
    @Override
    public void saveGame(GameState state, String filename) {
        // Реализация сохранения
    }

    @Override
    public GameState loadGame(String filename) {
        // Реализация загрузки
    }
}

// Контроллер с внедрением зависимостей
public class GameController {
    private final UnitService unitService;
    private final SaveLoadService saveLoadService;

    public GameController(UnitService unitService, SaveLoadService saveLoadService) {
        this.unitService = unitService;
        this.saveLoadService = saveLoadService;
    }

    public void handleMoveUnit(Unit unit, Position newPosition) {
        unitService.moveUnit(unit, newPosition);
    }

    public void handleSaveGame(String filename) {
        GameState currentState = getCurrentGameState();
        saveLoadService.saveGame(currentState, filename);
    }
}
```

Конфигурация зависимостей

```
public class DependencyContainer {
    private static DependencyContainer instance;
    private Map<Class<?>, Object> services;

    private DependencyContainer() {
        services = new HashMap<>();
        initializeServices();
    }

    public static DependencyContainer getInstance() {
        if (instance == null) {
            instance = new DependencyContainer();
        }
        return instance;
    }

    private void initializeServices() {
        // Регистрация сервисов
        services.put(UnitService.class, new UnitServiceImpl());
        services.put(SaveLoadService.class, new SaveLoadServiceImpl());
        services.put(CombatService.class, new CombatServiceImpl());
        services.put(AIService.class, new AIServiceImpl());
    }

    @SuppressWarnings("unchecked")
    public <T> T getService(Class<T> serviceClass) {
        return (T) services.get(serviceClass);
    }
}

// Использование
public class GameApplication {
    public static void main(String[] args) {
        DependencyContainer container = DependencyContainer.getInstance();

        UnitService unitService = container.getService(UnitService.class);
        SaveLoadService saveLoadService = container.getService(SaveLoadService.class);

        GameController controller = new GameController(unitService, saveLoadService);
        GameView view = new GameView();

        // Запуск приложения
        Game game = new Game(controller, view);
        game.start();
    }
}
```

Практический пример: Архитектура игры

```
// Главный класс приложения
public class Game {
    private GameModel model;
    private GameView view;
    private GameController controller;
    private GameService gameService;

    public Game() {
        initializeComponents();
        setupDependencies();
    }

    private void initializeComponents() {
        // Создание компонентов
        model = new GameModel();
        view = new GameView();
        controller = new GameController();
        gameService = new GameService();
    }

    private void setupDependencies() {
        // Настройка зависимостей
        controller.setModel(model);
        controller.setView(view);
        controller.setGameService(gameService);

        view.setController(controller);
        model.addObserver(view);
    }

    public void start() {
        // Запуск игры
        view.show();
        model.startNewGame();
    }
}

// Структура пакетов
/*
game/
├── model/          # Модели данных
│   ├── GameState.java
│   ├── Unit.java
│   └── Position.java
├── view/           # Представление
│   ├── GameView.java
│   ├── GameBoard.java
│   └── UnitInfoPanel.java
├── controller/     # Контроллеры
│   ├── GameController.java
│   └── UnitController.java
├── service/        # Бизнес-логика
│   ├── GameService.java
│   ├── UnitService.java
│   └── CombatService.java
├── repository/     # Доступ к данным
│   ├── UnitRepository.java
│   └── GameRepository.java
└── util/           # Утилиты
    ├── GameLogger.java
    └── ConfigManager.java
*/
```

Тестирование архитектуры

```
// Тест контроллера с mock объектами
public class GameControllerTest {
    private GameController controller;
    private GameModel mockModel;
    private GameView mockView;

    @Before
    public void setUp() {
        mockModel = mock(GameModel.class);
        mockView = mock(GameView.class);
        controller = new GameController(mockModel, mockView);
    }

    @Test
    public void testMoveUnit() {
        // Arrange
        Unit unit = new Unit("Test", 100, 10, 5, new Position(0, 0));
        Position newPosition = new Position(1, 1);

        when(mockModel.isValidMove(unit, newPosition)).thenReturn(true);

        // Act
        controller.handleMoveUnit(unit, newPosition);

        // Assert
        verify(mockModel).moveUnit(unit, newPosition);
    }

    @Test
    public void testInvalidMove() {
        // Arrange
        Unit unit = new Unit("Test", 100, 10, 5, new Position(0, 0));
        Position newPosition = new Position(1, 1);

        when(mockModel.isValidMove(unit, newPosition)).thenReturn(false);

        // Act
        controller.handleMoveUnit(unit, newPosition);

        // Assert
        verify(mockModel, never()).moveUnit(any(), any());
        verify(mockView).showError("Невозможно выполнить ход");
    }
}
```

Лучшие практики архитектуры

✅ Что делать:

- **Разделять ответственности** между компонентами
- **Использовать интерфейсы** для слабой связанности
- **Внедрять зависимости** извне
- **Следовать принципам SOLID**
- **Тестировать** каждый слой отдельно

❌ Чего избегать:

- **Смешивать слои** архитектуры
- **Создавать сильные зависимости** между компонентами
- **Игнорировать** принципы SOLID

Домашнее задание

Задача 1:

Реорганизовать существующий код по MVC архитектуре

Задача 2:

Реализовать Dependency Injection для основных сервисов

Задача 3:

Создать тесты для каждого слоя архитектуры

Что дальше?

На следующей лекции:

- **Многопоточность**
- **Threads и Runnable**
- **Синхронизация**
- **Concurrent collections**

Подготовка:

- Изучить главу 19-20 из учебника
- Выполнить домашнее задание
- Подготовить вопросы по текущей теме

Вопросы?

Контакты:

- **Email:** [ваш.email@university.edu]
- **Telegram:** [@username]
- **Офис:** [номер кабинета]

Следующая лекция: Многопоточность