Паттерны проектирования

Лекция 9: Архитектурные решения

Преподаватель: [Ваше имя]

Группа: 203

Семестр: Осенний 2024

План лекции

1. **Что такое паттерны проектирования?**
2. **Singleton паттерн**
3. **Factory паттерн**
4. **Observer паттерн**
5. **Strategy паттерн**
6. **Практический пример: Паттерны в игре**

Что такое паттерны проектирования?

Определение:

Паттерн проектирования — это типичное решение часто встречающейся проблемы в проектировании программного обеспечения.

Преимущества:

- **Переиспользование** проверенных решений
- **Стандартизация** кода
- **Улучшение** читаемости и поддерживаемости
- **Упрощение** коммуникации между разработчиками

Классификация паттернов

По назначению:

- **Порождающие** — создание объектов
- **Структурные** — композиция объектов
- **Поведенческие** — взаимодействие объектов

По уровню:

- **Классовые** — работают с классами
- **Объектные** — работают с объектами

Singleton паттерн

Назначение:

Singleton гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к нему.

Применение:

- **Логирование** — один логгер для всего приложения
- **Конфигурация** — глобальные настройки
- **Кэширование** — общий кэш данных
- **Соединения с БД** — пул соединений

Реализация Singleton

Простая реализация:

```
public class GameLogger {
    private static GameLogger instance;
    private List<String> logMessages;

    // Приватный конструктор
    private GameLogger() {
        logMessages = new ArrayList<>();
    }

    // Глобальная точка доступа
    public static GameLogger getInstance() {
        if (instance == null) {
            instance = new GameLogger();
        }
        return instance;
    }

    public void log(String message) {
        logMessages.add(message);
        System.out.println("LOG: " + message);
    }
```

Thread-safe Singleton

Двойная проверка блокировки:

```
public class GameConfig {
    private static volatile GameConfig instance;
    private Properties config;

    private GameConfig() {
        config = new Properties();
        loadDefaultConfig();
    }

    public static GameConfig getInstance() {
        if (instance == null) {
            synchronized (GameConfig.class) {
                if (instance == null) {
                    instance = new GameConfig();
                }
            }
        }
        return instance;
    }

    private void loadDefaultConfig() {
        config.setProperty("board.size", "10");
        config.setProperty("max.units", "50");
        config.setProperty("auto.save", "true");
    }
```

Factory паттерн

Назначение:

Factory создает объекты без указания их точных классов.

Применение:

- **Создание юнитов** разных типов
- **Создание зданий** различных категорий
- **Создание оружия** и предметов
- **Создание эффектов** и анимаций

Простая Factory

```
public class UnitFactory {

    public static Unit createUnit(UnitType type, String name, Position position) {
        switch (type) {
            case WARRIOR:
                return new Warrior(name, 150, 25, 15, position);
            case ARCHER:
                return new Archer(name, 100, 30, 10, position);
            case MAGE:
                return new Mage(name, 80, 40, 5, position);
            default:
                throw new IllegalArgumentException("Неизвестный тип юнита: " + type);
        }
    }

    public static Unit createRandomUnit(String name, Position position) {
        UnitType[] types = UnitType.values();
        UnitType randomType = types[(int) (Math.random() * types.length)];
        return createUnit(randomType, name, position);
    }
}

// Использование
Unit warrior = UnitFactory.createUnit(UnitType.WARRIOR, "Aragorn", new Position(0, 0));
Unit randomUnit = UnitFactory.createRandomUnit("Hero", new Position(1, 1));
```

Abstract Factory

```
public abstract class AbstractUnitFactory {
    public abstract Unit createWarrior(String name, Position position);
    public abstract Unit createArcher(String name, Position position);
    public abstract Unit createMage(String name, Position position);
}

public class HumanUnitFactory extends AbstractUnitFactory {
    @Override
    public Unit createWarrior(String name, Position position) {
        return new HumanWarrior(name, 160, 26, 16, position);
    }

    @Override
    public Unit createArcher(String name, Position position) {
        return new HumanArcher(name, 110, 31, 11, position);
    }

    @Override
    public Unit createMage(String name, Position position) {
        return new HumanMage(name, 90, 41, 6, position);
    }
}

public class OrcUnitFactory extends AbstractUnitFactory {
    @Override
    public Unit createWarrior(String name, Position position) {
        return new OrcWarrior(name, 180, 28, 18, position);
    }

    @Override
    public Unit createArcher(String name, Position position) {
        return new OrcArcher(name, 120, 29, 12, position);
    }

    @Override
    public Unit createMage(String name, Position position) {
        return new OrcMage(name, 70, 45, 4, position);
    }
}
```

Observer паттерн

Назначение:

Observer определяет зависимость "один-ко-многим" между объектами так, что при изменении состояния одного объекта все зависимые от него объекты уведомляются автоматически.

Применение:

- **Уведомления** о событиях игры
- **Обновление UI** при изменении состояния
- **Логирование** действий игроков
- **Синхронизация** между компонентами

Реализация Observer

```
public interface GameObserver {
    void onGameEvent(GameEvent event);
}

public abstract class GameEvent {
    private String type;
    private long timestamp;

    public GameEvent(String type) {
        this.type = type;
        this.timestamp = System.currentTimeMillis();
    }

    public String getType() { return type; }
    public long getTimestamp() { return timestamp; }
}

public class UnitMovedEvent extends GameEvent {
    private Unit unit;
    private Position from;
    private Position to;

    public UnitMovedEvent(Unit unit, Position from, Position to) {
        super("UNIT_MOVED");
        this.unit = unit;
        this.from = from;
        this.to = to;
    }

    // Геттеры...
}

public class GameSubject {
    private List<GameObserver> observers = new ArrayList<>();

    public void addObserver(GameObserver observer) {
        observers.add(observer);
    }

    public void removeObserver(GameObserver observer) {
        observers.remove(observer);
    }

    public void notifyObservers(GameEvent event) {
        for (GameObserver observer : observers) {
            observer.onGameEvent(event);
        }
    }
}
```

Использование Observer

```
public class GameEngine extends GameSubject {

    public void moveUnit(Unit unit, Position newPosition) {
        Position oldPosition = unit.getPosition();

        // Выполнение движения
        unit.setPosition(newPosition);

        // Уведомление наблюдателей
        UnitMovedEvent event = new UnitMovedEvent(unit, oldPosition, newPosition);
        notifyObservers(event);
    }
}

public class GameUI implements GameObserver {

    @Override
    public void onGameEvent(GameEvent event) {
        if (event instanceof UnitMovedEvent) {
            UnitMovedEvent moveEvent = (UnitMovedEvent) event;
            animateUnitMove(moveEvent.getUnit(), moveEvent.getFrom(), moveEvent.getTo());
        }
    }

    private void animateUnitMove(Unit unit, Position from, Position to) {
        // Анимация движения юнита
    }
}

// Использование
GameEngine engine = new GameEngine();
GameUI ui = new GameUI();
engine.addObserver(ui);

// При движении юнита UI автоматически обновится
engine.moveUnit(warrior, new Position(2, 2));
```

Strategy паттерн

Назначение:

Strategy определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми.

Применение:

- **Алгоритмы ИИ** для разных типов юнитов
- **Стратегии атаки** различных оружейных систем
- **Алгоритмы поиска пути** для разных местностей
- **Стратегии экономики** для разных фракций

Реализация Strategy

```
public interface AIStrategy {
    Position calculateNextMove(Unit unit, GameState gameState);
    Unit selectTarget(Unit unit, List<Unit> enemies);
    boolean shouldRetreat(Unit unit, GameState gameState);
}

public class AggressiveAIStrategy implements AIStrategy {
    @Override
    public Position calculateNextMove(Unit unit, GameState gameState) {
        // Агрессивная стратегия: движение к ближайшему врагу
        Unit nearestEnemy = findNearestEnemy(unit, gameState.getEnemyUnits());
        if (nearestEnemy != null) {
            return calculatePathToTarget(unit.getPosition(), nearestEnemy.getPosition());
        }
        return unit.getPosition();
    }

    @Override
    public Unit selectTarget(Unit unit, List<Unit> enemies) {
        // Выбор самого слабого врага
        return enemies.stream()
            .min(Comparator.comparingInt(Unit::getHealth))
            .orElse(null);
    }

    @Override
    public boolean shouldRetreat(Unit unit, GameState gameState) {
        // Агрессивные юниты не отступают
        return false;
    }
}
```

Другие стратегии ИИ

```
public class DefensiveAIStrategy implements AIStrategy {
    @Override
    public Position calculateNextMove(Unit unit, GameState gameState) {
        // Защитная стратегия: движение к союзникам
        if (unit.getHealth() < unit.getMaxHealth() * 0.3) {
            Unit nearestAlly = findNearestAlly(unit, gameState.getAllyUnits());
            if (nearestAlly != null) {
                return calculatePathToTarget(unit.getPosition(), nearestAlly.getPosition());
            }
        }
        return unit.getPosition();
    }

    @Override
    public Unit selectTarget(Unit unit, List<Unit> enemies) {
        // Выбор врага с наименьшей атакой
        return enemies.stream()
            .min(Comparator.comparingInt(Unit::getAttack))
            .orElse(null);
    }

    @Override
    public boolean shouldRetreat(Unit unit, GameState gameState) {
        // Отступление при низком здоровье
        return unit.getHealth() < unit.getMaxHealth() * 0.2;
    }
}

public class BalancedAIStrategy implements AIStrategy {
    @Override
    public Position calculateNextMove(Unit unit, GameState gameState) {
        // Сбалансированная стратегия
        if (unit.getHealth() < unit.getMaxHealth() * 0.4) {
            return new DefensiveAIStrategy().calculateNextMove(unit, gameState);
        } else {
            return new AggressiveAIStrategy().calculateNextMove(unit, gameState);
        }
    }

    // Другие методы...
}
```

Использование Strategy

```
public class AIUnit extends Unit {
    private AIStrategy aiStrategy;

    public AIUnit(String name, int health, int attack, int defense, Position position) {
        super(name, health, attack, defense, position);
        this.aiStrategy = new BalancedAIStrategy(); // По умолчанию
    }

    public void setAIStrategy(AIStrategy strategy) {
        this.aiStrategy = strategy;
    }

    public void performAITurn(GameState gameState) {
        // Вычисление следующего хода
        Position nextMove = aiStrategy.calculateNextMove(this, gameState);
        if (!nextMove.equals(getPosition())) {
            moveTo(nextMove);
        }

        // Выбор цели для атаки
        Unit target = aiStrategy.selectTarget(this, gameState.getEnemyUnits());
        if (target != null && canAttack(target)) {
            attack(target);
        }

        // Проверка необходимости отступления
        if (aiStrategy.shouldRetreat(this, gameState)) {
            retreat();
        }
    }
}

// Использование
AIUnit orcWarrior = new AIUnit("Orc", 180, 28, 18, new Position(5, 5));
orcWarrior.setAIStrategy(new AggressiveAIStrategy());
orcWarrior.performAITurn(gameState);
```

Комбинирование паттернов

```
public class GameManager {
    private static GameManager instance; // Singleton
    private GameEngine gameEngine;
    private UnitFactory unitFactory;
    private List<GameObserver> observers;

    private GameManager() {
        gameEngine = new GameEngine();
        unitFactory = new HumanUnitFactory(); // По умолчанию
        observers = new ArrayList<>();
    }

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

    public void setUnitFactory(AbstractUnitFactory factory) {
        this.unitFactory = factory;
    }

    public Unit createUnit(UnitType type, String name, Position position) {
        Unit unit = unitFactory.createWarrior(name, position); // Factory
        gameEngine.addUnit(unit);

        // Observer: уведомление о создании юнита
        UnitCreatedEvent event = new UnitCreatedEvent(unit);
        notifyObservers(event);

        return unit;
    }

    public void addObserver(GameObserver observer) {
        observers.add(observer);
    }

    private void notifyObservers(GameEvent event) {
        for (GameObserver observer : observers) {
            observer.onGameEvent(event);
        }
    }
}
```

Другие полезные паттерны

Command паттерн:

```
public interface GameCommand {
    void execute();
    void undo();
}

public class MoveUnitCommand implements GameCommand {
    private Unit unit;
    private Position from;
    private Position to;

    public MoveUnitCommand(Unit unit, Position to) {
        this.unit = unit;
        this.from = unit.getPosition();
        this.to = to;
    }

    @Override
    public void execute() {
        unit.setPosition(to);
    }

    @Override
```

Лучшие практики использования паттернов

✅ Что делать:

- **Использовать паттерны** для решения конкретных проблем
- **Комбинировать** паттерны для сложных решений
- **Документировать** использование паттернов
- **Тестировать** паттерны отдельно

❌ Чего избегать:

- **Применять паттерны** везде без необходимости
- **Усложнять** простые решения
- **Игнорировать** производительность
- **Забывать про** рефакторинг

Домашнее задание

Задача 1:

Реализовать Factory для создания различных типов зданий

Задача 2:

Создать Observer для уведомления о событиях игры

Задача 3:

Реализовать Strategy для различных алгоритмов поиска пути

Что дальше?

На следующей лекции:

- **Архитектура приложения**
- **MVC паттерн**
- **Разделение ответственности**
- **Модульность**

Подготовка:

- Изучить главу 17-18 из учебника
- Выполнить домашнее задание
- Подготовить вопросы по текущей теме

Вопросы?

Контакты:

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

Следующая лекция: Архитектура приложения