Лабораторная работа №6. Введение в паттерны проектирования.

1. Singleton (Одиночка)

Описание:
Singleton обеспечивает создание единственного экземпляра класса и предоставляет глобальную точку доступа к нему.

Когда использовать:

- Для управления доступом к общему ресурсу (например, база данных, файл журнала).
- Чтобы предотвратить создание лишних объектов, особенно если они затратны по ресурсам.

```
class Singleton {
    private static Singleton instance;

    // Приватный конструктор
    private Singleton() {}

    // Метод доступа к экземпляру
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    public void showMessage() {
        System.out.println("Hello from Singleton!");
    }
}

public class Main {
    public static void main(String[] args) {
        Singleton singleton = Singleton.getInstance();
        singleton.showMessage();
    }
}
```

2. Factory Method (Фабричный метод)

Описание:
Factory Method предоставляет интерфейс для создания объектов, но позволяет подклассам изменять тип создаваемых объектов.

Когда использовать:

- Когда заранее неизвестен точный класс объектов, которые нужно создать.
- Для увеличения гибкости кода при добавлении новых типов объектов.

```
interface Animal {
    void makeSound();
}

class Dog implements Animal {
    public void makeSound() {
        System.out.println("Woof!");
    }
}

class Cat implements Animal {
    public void makeSound() {
        System.out.println("Meow!");
    }
}

class AnimalFactory {
    public Animal createAnimal(String type) {
        if (type.equalsIgnoreCase("dog")) {
            return new Dog();
        } else if (type.equalsIgnoreCase("cat")) {
            return new Cat();
        }
        return null;
    }
}

public class Main {
    public static void main(String[] args) {
        AnimalFactory factory = new AnimalFactory();
        Animal dog = factory.createAnimal("dog");
        dog.makeSound();

        Animal cat = factory.createAnimal("cat");
        cat.makeSound();
    }
}
```

3. Observer (Наблюдатель)

Описание:
Observer используется для оповещения нескольких объектов о произошедших изменениях в другом объекте.

Когда использовать:

- Когда объект должен уведомлять других о своем состоянии.
- Для реализации механизмов подписки/уведомления.

```
import java.util.ArrayList;
import java.util.List;

interface Observer {
    void update(String message);
}

class ConcreteObserver implements Observer {
    private String name;

    public ConcreteObserver(String name) {
        this.name = name;
    }

    public void update(String message) {
        System.out.println(name + " received update: " +
message);
    }
}

class Subject {
    private List<Observer> observers = new ArrayList<>();

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

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

    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Subject subject = new Subject();

        Observer observer1 = new ConcreteObserver("Observer1");
        Observer observer2 = new ConcreteObserver("Observer2");

        subject.addObserver(observer1);
        subject.addObserver(observer2);

        subject.notifyObservers("Update 1");
        subject.notifyObservers("Update 2");
    }
}
```

4. Command (Команда)

Описание:
Command превращает запросы в объекты, позволяя передавать их как аргументы, ставить в очередь или логировать. Паттерн инкапсулирует действия в виде объектов.

Когда использовать:

- Для реализации отмены/повтора действий (Undo/Redo).
- Когда нужно передавать операции между объектами.
- Для задач очередей или планировщиков.

```
// Интерфейс команды
interface Command {
    void execute();
}

// Получатель
class Light {
    public void turnOn() {
        System.out.println("The light is ON");
    }

    public void turnOff() {
        System.out.println("The light is OFF");
    }
}

// Конкретные команды
class TurnOnCommand implements Command {
    private Light light;

    public TurnOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOn();
    }
}

class TurnOffCommand implements Command {
    private Light light;

    public TurnOffCommand(Light light) {
        this.light = light;
    }
```

```
    public void execute() {
        light.turnOff();
    }
}

// Инициатор
class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}

public class Main {
    public static void main(String[] args) {
        Light light = new Light();
        Command turnOn = new TurnOnCommand(light);
        Command turnOff = new TurnOffCommand(light);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(turnOn);
        remote.pressButton();

        remote.setCommand(turnOff);
        remote.pressButton();
    }
}
```

5. Flyweight (Приспособленец)

Описание:
Flyweight минимизирует использование памяти путем совместного использования множества мелких объектов. Паттерн извлекает общие свойства в отдельные объекты, чтобы избежать дублирования.

Когда использовать:

- Когда в приложении есть множество объектов, имеющих общие свойства.
- Для оптимизации производительности и памяти в системах с ограниченными ресурсами.

```
import java.util.HashMap;
import java.util.Map;

// Flyweight
interface Shape {
    void draw(int x, int y);
}

// Конкретный Flyweight
class Circle implements Shape {
    private String color;

    public Circle(String color) {
        this.color = color;
    }

    public void draw(int x, int y) {
        System.out.println("Drawing " + color + " circle at (" +
x + ", " + y + ")");
    }
}

// Flyweight Factory
class ShapeFactory {
    private static final Map<String, Shape> shapes = new
HashMap<>();

    public static Shape getCircle(String color) {
        Shape shape = shapes.get(color);

        if (shape == null) {
            shape = new Circle(color);
            shapes.put(color, shape);
            System.out.println("Creating " + color + " circle");
```

```
        }

        return shape;
    }
}

public class Main {
    public static void main(String[] args) {
        Shape redCircle1 = ShapeFactory.getCircle("red");
        redCircle1.draw(10, 20);

        Shape redCircle2 = ShapeFactory.getCircle("red");
        redCircle2.draw(30, 40);

        Shape blueCircle = ShapeFactory.getCircle("blue");
        blueCircle.draw(50, 60);
    }
}
```

Задания по вариантам.

Для каждого паттерна необходимо создавать отдельный package.

I. Одиночка

1. **Реализовать Singleton для управления игровыми настройками.**
   Создайте класс GameSettings, в котором будут поля volume (громкость звука, от 0 до 100) и resolution (строка, например, "1920x1080"), а также методы getInstance(), setVolume(int volume), getVolume(), setResolution(String resolution), getResolution(). Настройки необходимо сохранять в файле с учетом времени изменения.

2. Реализовать Singleton для ведения логов игровых событий.
   Создайте класс Logger, в котором будет поле logFilePath (путь к файлу лога) и методы getInstance(), log(String message) для добавления строки в лог с отметкой времени, а также readLog() для чтения содержимого файла. Файл также должен хранить в себе время игрового события.

3. **Реализовать Singleton для сохранения и загрузки прогресса игры.**
   Создайте класс SaveManager, в котором будут поля для хранения данных (например, карта Map<String, Object> с ключами health, position, coins) и методы getInstance(), saveGame(String filename) для сохранения данных в файл и loadGame(String filename) для загрузки данных из файла. В файл нужно записывать время загрузки.
4. **Реализовать Singleton для управления подключением к базе данных.**
   Создайте класс DatabaseConnection, в котором будет поле connection (строка состояния подключения, например, "connected" или "disconnected") и методы getInstance(), connect(), disconnect(), getConnectionStatus(). База данных может выступать файлом. При создании необходимо в файле прописать время.

5. **Реализовать Singleton для управления звуками в игре.**
   Создайте класс AudioManager, в котором будет поле soundVolume (громкость звука, от 0 до 100) и методы getInstance(), setVolume(int volume), getVolume(), playSound(String soundName) для воспроизведения звука с заданным именем. При воспроизведении звука, его название записывается в файл и время, когда он был проигран.
6. **Реализовать Singleton для обработки игровых событий.**
   Создайте класс EventBus, в котором будут методы getInstance(), register(String event, Runnable callback), trigger(String event) для добавления событий и вызова зарегистрированных обработчиков. Каждое игровое событие записывается в файл с учетом времени вызова.
7. **Реализовать Singleton для отслеживания достижений игрока.**
   Создайте класс AchievementTracker, в котором будет поле achievements (список достижений, List<String>) и методы getInstance(), addAchievement(String achievement), getAchievements(). Каждое достижение и время его получения записываются в файл.
8. **Реализовать Singleton для обработки пользовательских вводов.**
   Создайте класс InputManager, в котором будут методы getInstance(), registerKeyBinding(String action, String key), getKeyForAction(String action) для назначения клавиш действиям. Пользовательский ввод и время ввода записываются в файл.
9. **Реализовать Singleton для управления игровой сессией.**
   Создайте класс SessionManager, в котором будут поля sessionId (строка) и startTime (дата/время начала) и методы getInstance(), startSession(), endSession(), getSessionInfo(). В файл необходимо записывать время начала и конца сессии.
10. **Реализовать Singleton для чтения конфигурации игры.**
    Создайте класс Configuration, в котором будут методы getInstance(),

loadConfig(String filename) для загрузки конфигурации из файла, а также get(String key) для получения значений по ключу. Загрузка конфига должна быть зафиксирована в файле с учетом времени.

11. **Реализовать Singleton для загрузки игровых ресурсов.**
    Создайте класс ResourceLoader, в котором будут методы getInstance(), loadTexture(String filename), loadModel(String filename), loadAudio(String filename) для загрузки ресурсов. При успешной загрузке, в файл записывается время.

12. Реализовать Singleton для управления игровой физикой.
    Создайте класс PhysicsEngine, в котором будут методы getInstance(), simulateStep(float deltaTime), addObject(Object obj), removeObject(Object obj) для работы с физическими объектами. При добавлении и удалении объекта фиксируется время в файле.

13. Реализовать Singleton для управления игровыми уведомлениями.
    Создайте класс NotificationManager, в котором будут методы getInstance(), showNotification(String message), clearNotifications(). При уведомлении, в файл записывается время.

14. **Реализовать Singleton для отслеживания текущего состояния игры.**
    Создайте класс StateTracker, в котором будет поле currentState (строка, например, "running", "paused") и методы getInstance(), setState(String state), getState(). При попытке изменить состояние, в файл записывается время.

15. Реализовать Singleton для управления сетевым соединением.
    Создайте класс NetworkManager, в котором будут поля isConnected (логическое) и методы getInstance(), connect(), disconnect(), getConnectionStatus(). При подключении и отключении, файл записывает время.

II. Фабрика

1. **Реализовать фабрику для создания игровых персонажей.**
   Создайте класс CharacterFactory, в котором будет метод createCharacter(String type). Для типа "warrior" возвращается объект класса Warrior, а для "mage" — объект класса Mage. Классы персонажей должны реализовать общий интерфейс Character с методом attack().
2. **Реализовать фабрику для создания оружия.**
   Создайте класс WeaponFactory, в котором будет метод createWeapon(String type). Для "sword" создается объект класса Sword, а для "bow" — объект класса Bow. Оба класса должны реализовать интерфейс Weapon с методом use().

3. Реализовать фабрику для создания врагов.
   Создайте класс EnemyFactory, в котором будет метод createEnemy(String type). Для "zombie" создается объект класса Zombie, а для "skeleton" — объект класса Skeleton. Классы врагов реализуют интерфейс Enemy с методом attack().

4. **Реализовать фабрику для создания игровых уровней.**
   Создайте класс LevelFactory, в котором будет метод createLevel(String difficulty). Для "easy" создается объект класса EasyLevel, а для "hard" — объект класса HardLevel. Уровни должны содержать метод generate().

5. **Реализовать фабрику для создания игровых зданий.**
   Создайте класс BuildingFactory, в котором будет метод createBuilding(String type). Для "castle" возвращается объект класса Castle, а для "barracks" — объект класса Barracks. Классы зданий должны реализовать интерфейс Building с методом build().

6. **Реализовать фабрику для создания предметов.**
   Создайте класс ItemFactory, в котором будет метод createItem(String rarity). Для "common" создается объект класса CommonItem, а для "legendary" — объект класса LegendaryItem. У предметов должен быть метод use().

7. **Реализовать фабрику для создания заклинаний.**
   Создайте класс SpellFactory, в котором будет метод createSpell(String type). Для "fire" создается объект класса FireSpell, а для "ice" — объект класса IceSpell. Заклинания должны реализовать интерфейс Spell с методом cast().

8. **Реализовать фабрику для создания транспорта.**
   Создайте класс VehicleFactory, в котором будет метод createVehicle(String type). Для "car" создается объект класса Car, а для "boat" — объект класса Boat. Транспорт должен содержать метод move().

9. **Реализовать фабрику для создания игровых ресурсов.**
   Создайте класс ResourceFactory, в котором будет метод createResource(String type). Для "wood" создается объект класса Wood, а для "stone" — объект класса Stone. Ресурсы должны содержать метод collect().

10. **Реализовать фабрику для создания питомцев.**
    Создайте класс PetFactory, в котором будет метод createPet(String type). Для "dog" создается объект класса Dog, а для "cat" — объект класса Cat. Питомцы должны содержать метод play().

11. **Реализовать фабрику для создания боевых машин.**
    Создайте класс BattleMachineFactory, в котором будет метод

createMachine(String type). Для "tank" создается объект класса Tank, а для "plane" — объект класса Plane. У машин должен быть метод attack()

12. **Реализовать фабрику для создания ландшафта.**
    Создайте класс TerrainFactory, в котором будет метод createTerrain(String type). Для "forest" создается объект класса Forest, а для "desert" — объект класса Desert. Ландшафт должен содержать метод render().

13. **Реализовать фабрику для создания эффектов.**
    Создайте класс EffectFactory, в котором будет метод createEffect(String type). Для "explosion" создается объект класса ExplosionEffect, а для "lightning" — объект класса LightningEffect. Эффекты должны содержать метод play().

14. **Реализовать фабрику для создания игровых карт.**
    Создайте класс MapFactory, в котором будет метод createMap(String type). Для "city" создается объект класса CityMap, а для "dungeon" — объект класса DungeonMap. Карты должны содержать метод generate().

15. **Реализовать фабрику для создания игровых костюмов.**
    Создайте класс CostumeFactory, в котором будет метод createCostume(String type). Для "knight" создается объект класса KnightCostume, а для "wizard" — объект класса WizardCostume. Костюмы должны содержать метод equip().

III. Наблюдатель

1. **Реализовать систему уведомлений для подписчиков изменений погоды.**
   Создайте класс WeatherStation с методами addObserver(Observer observer) и removeObserver(Observer observer) для добавления и удаления подписчиков, а также notifyObservers(). Подписчики, реализующие интерфейс Observer, получают уведомление с текущей температурой.

2. **Реализовать систему подписки на игровые события.**
   Создайте класс GameEventManager для добавления и удаления подписчиков с методами subscribe(String event, Observer observer) и notify(String event). Подписчики получают уведомление о событии, например, "gameOver".

3. **Реализовать систему обновления цен на бирже.**
   Создайте класс StockMarket с методами addStock(String stock) и updateStock(String stock, float price). Подписчики, реализующие интерфейс Observer, получают уведомление об изменении цен.

4. **Реализовать систему оповещения о статусе игроков в сети.**
   Создайте класс OnlineStatusManager, в котором подписчики получают уведомления о статусе "online" или "offline" для каждого игрока.

5. **Реализовать систему уведомлений об обновлении заданий.**
   Создайте класс QuestManager, подписчики которого получают уведомления при добавлении новых заданий или завершении текущих.

6. **Реализовать систему наблюдателей за состоянием здоровья персонажа.**
   Создайте класс HealthTracker, в котором подписчики уведомляются об изменении здоровья (например, снижение до критического уровня).

7. **Реализовать систему уведомлений о текущем времени.**
   Создайте класс Clock с методом tick(), который уведомляет всех подписчиков каждые 60 секунд.

8. **Реализовать систему рассылки новостей.**
   Создайте класс NewsPublisher, подписчики которого уведомляются о новых статьях, добавленных через метод publish(String article).

9. **Реализовать систему управления состоянием игрового босса.**
   Создайте класс BossStateManager, который уведомляет подписчиков об изменении состояния босса (например, "rageMode" или "stunned").

10. **Реализовать систему мониторинга производства ресурсов.**
    Создайте класс ResourceFactoryMonitor, подписчики которого получают уведомления о завершении производства новых партий ресурсов.

11. Реализовать систему мониторинга игрового времени.
    Создайте класс GameTimeTracker, который уведомляет подписчиков об изменении игрового времени, например, наступлении дня или ночи.

12. **Реализовать систему наблюдения за крафтом предметов.**
    Создайте класс CraftingStation, в котором подписчики уведомляются о завершении процесса крафта.

13. **Реализовать систему подписки на апгрейды базы.**
    Создайте класс BaseUpgradeNotifier, который отправляет подписчикам уведомления о завершении апгрейдов базы.

14. **Реализовать систему уведомлений для объектов в поле зрения.**
    Создайте класс VisionSystem, подписчики которого уведомляются о появлении новых объектов в поле зрения игрока.

15. **Реализовать систему мониторинга количества очков команды.**
    Создайте класс ScoreTracker, подписчики которого получают уведомления об изменении очков в командной игре.

IV. Команда.

1. Реализовать управление персонажем через команды.
   Создайте интерфейс Command с методом execute(). Реализуйте классы MoveCommand, AttackCommand, DefendCommand и вызовите их из класса PlayerController.

2. **Реализовать систему управления умным домом.**
   Создайте команды TurnOnLights, TurnOffLights и SetTemperature для управления освещением и термостатом через класс SmartHomeController.

3. **Реализовать очередь команд для ритуалов в игре.**
   Создайте команды StartRitual, SummonMonster и FinishRitual, добавьте их в очередь выполнения через класс RitualExecutor.

4. **Реализовать систему отмены действий игрока.**
   Добавьте в команды методы undo() для отмены последних выполненных действий, таких как движение и атака.

5. **Реализовать систему управления меню.**
   Создайте команды OpenMenuCommand, CloseMenuCommand и SelectItemCommand для управления элементами интерфейса.

6. Реализовать систему очереди действий для строительных ботов. Создайте команды BuildWall, RepairBuilding и GatherResources, добавьте их в очередь для выполнения через класс BotController.

7. **Реализовать систему управления атакой и защитой башни.** Создайте команды ActivateTurrets и DeployShield для управления башнями через класс TowerController.

8. **Реализовать макросы для серии команд в редакторе карт.** Создайте команды AddObject, RemoveObject и ChangeTerrain. Объедините их в макрос, чтобы выполнять их последовательно.

9. **Реализовать систему автоматизации действий в шахте.** Создайте команды StartDrilling, CollectOres и StopDrilling для управления процессами через класс MiningAutomation.

10. **Реализовать управление боевым дроном.** Создайте команды MoveToTarget, ScanArea и FireWeapon для управления дроном через класс DroneController.

11. **Реализовать систему редактирования текстового документа.** Создайте команды InsertText, DeleteText и UndoEdit для управления текстовым документом через класс DocumentEditor.

12. **Реализовать систему кастов заклинаний через команды.** Создайте команды CastFireball, CastShield и HealAlly для управления магией через класс SpellCaster.

13. **Реализовать систему управления фермой.** Создайте команды PlantCrop, HarvestCrop и WaterField для фермерского симулятора через класс FarmController.

14. **Реализовать управление флотом кораблей.**
    Создайте команды MoveFleet, EngageEnemy и DefendPosition для управления флотом через класс FleetCommander.

15. **Реализовать систему управления парком аттракционов.**
    Создайте команды StartRide, StopRide и PerformMaintenance для управления аттракционами через класс ParkManager.

V. Приспособленец.

1. **Реализовать пул объектов для деревьев на карте.**
   Создайте класс Tree с параметрами позиции и типа. Используйте фабрику для переиспользования объектов.

2. **Реализовать оптимизированное хранение текстур.**
   Создайте класс Texture с полями для пути к текстуре и фабрику для работы с ограниченным числом экземпляров.

3. **Реализовать пул объектов для врагов.**
   Создайте фабрику, которая возвращает объекты Enemy с общими характеристиками, но разной позицией.

4. **Реализовать пул объектов для зданий на игровой карте.**
   Создайте класс Building, содержащий информацию о типе здания и фабрику для переиспользования объектов с одинаковыми характеристиками.

5. **Реализовать систему оптимизированного хранения пуль.**
   Создайте класс Bullet, который переиспользуется через пул объектов.

Добавьте поля для позиции и фабрику для управления созданием объектов.

6. Реализовать оптимизированное хранение анимаций.
   Создайте класс Animation с полем для имени анимации и используйте фабрику для хранения уникальных анимаций.

7. **Реализовать пул объектов для NPC.**
   Создайте класс NPC с общими характеристиками, такими как модель и текстуры, но с уникальной позицией. Реализуйте фабрику для управления их созданием.

8. **Реализовать систему кэширования ресурсов.**
   Создайте класс ResourceCache, который управляет созданием и переиспользованием объектов ресурсов, таких как текстуры или звуковые файлы.

9. **Реализовать пул объектов для кристаллов в игре.**
   Создайте класс Crystal, в котором общие свойства (цвет, форма) кэшируются, а уникальные параметры (позиция, размер) задаются динамически.

10. **Реализовать систему оптимизации для частиц.**
    Создайте класс Particle, который кэширует визуальные свойства частиц. Реализуйте фабрику для переиспользования объектов в системе частиц.

11. Реализовать пул объектов для облаков.
    Создайте класс Cloud с общими параметрами (текстура, скорость), которые кэшируются, и фабрику для создания объектов с уникальной позицией.

12. **Реализовать систему оптимизации для противников в уровне.**
    Создайте класс EnemyType для хранения характеристик (модель, текстуры) и фабрику, которая переиспользует объекты.

13. **Реализовать пул объектов для текстовых подсказок.**
    Создайте класс TextHint, в котором общий стиль текста кэшируется, а содержимое задается динамически.

14. **Реализовать оптимизированное хранение декораций.**
    Создайте класс Decoration с полями для типа (например, камень или куст) и позиции, а также фабрику для кэширования объектов.

15. Реализовать пул объектов для стрел в бою.
    Создайте класс Arrow с общими параметрами (модель, текстуры), кэшируемыми через фабрику, и уникальной позицией.