ООП в играх: Как программируют Mortal Kombat, World of Warcraft и GTA

1. Класс = Шаблон персонажа в игре

Представь, что ты создаёшь нового героя в RPG-игре. У тебя есть **шаблон (класс)** - например, “Эльф-лучник”. Это не сам персонаж, а его описание:

```
// Это КЛАСС - шаблон для всех эльфов-лучников
class ElfArcher {
    string имя;
    int здоровье = 100;
    int урон = 25;
    int стрелы = 30;

    void Стрелять() { ... }
    void Перезарядить() { ... }
};
```

Парадигма: Абстракция - мы создаём упрощённую модель реальности, скрывая сложные детали.

2. Объект = Конкретный персонаж

Когда ты создаёшь **конкретного персонажа** по шаблону - это **объект**:

```
// Создаём ОБЪЕКТЫ (конкретных персонажей)
ElfArcher леголас;    // Леголас - объект класса ElfArcher
ElfArcher левлон;     // Левлон - другой объект того же класса

леголас.имя = "Леголас";
леволас.имя = "Левлон";
```

Пример из игр:

- В **Dota 2** есть класс Hero, а объекты - это конкретные герои: Pudge, Invoker, Crystal Maiden

- Каждый имеет одинаковые параметры (здоровье, манна), но разные значения

3. Инкапсуляция = Скрытые статы персонажа

В игре ты видишь только **полоску здоровья**, но не знаешь точное число HP. Это **инкапсуляция** - скрытие внутренних данных:

```
class Персонаж {
private:    // Закрытая зона - только для игры
   int точноеЗдоровье = 150;
   int скрытыйУрон = 45;

public:     // Открытая зона - что видит игрок
   void ПолучитьУрон(int урон) {
      точноеЗдоровье -= урон;
      // Игра сама решает, как обработать урон
   }

   int ПоказатьЗдоровье() {
      return точноеЗдоровье; // Игрок видит только результат
   }
};
```

В играх это:

- **Skyrim**: Ты не видишь формулы расчёта урона, только результат

- **World of Warcraft**: Характеристики (сила, ловкость) скрыто влияют на урон

- **Cyberpunk 2077**: Внутренние расчёты хакинга скрыты от игрока

4. Наследование = Эволюция классов персонажей

В **Diablo** есть базовый класс Воин, а от него наследуются специализации:

```
// Базовый класс - общие свойства всех воинов
class Воин {
protected:
   int здоровье;
   int сила;
   void Атаковать() { ... }
};

// Наследуем - берём всё от Воина и добавляем своё
class Паладин : public Воин {
   void Лечение() { ... }  // Добавили новую способность
   void СвященныйЩит() { ... }
};

class Варвар : public Воин {
   int ярость;  // Новая характеристика
```

```
    void Вихрь() { ... }    // Новая способность
};
```

Дерево классов в играх:

5. Полиморфизм = Разные эффекты одной кнопки

На кнопку **Q** разные герои реагируют по-разному:

```
class Герой {
public:
    virtual void НажмиQ() = 0; // Виртуальный метод
};

class Маг : public Герой {
    void НажмиQ() override {
        ВыпуститьОгненныйШар();  // У мага - огненный шар
    }
};

class Лучник : public Герой {
    void НажмиQ() override {
        ВыстрелитьСтрелой();    // У лучника - стрела
    }
};

class Лекарь : public Герой {
    void НажмиQ() override {
        ВылечитьСоюзника();     // У лекаря - лечение
```

```
    }
};
```

В играх:

- **League of Legends**: Кнопка Q у каждого чемпиона делает что-то уникальное

- **Overwatch**: Shift - уникальная способность для каждого героя

- **The Witcher 3**: Одна кнопка атаки, но разные анимации для каждого типа оружия

6. Конструктор = Создание персонажа в меню

Когда ты создаёшь нового персонажа в начале игры:

```
class Персонаж {
public:
    // Конструктор - вызывается при создании
    Персонаж(string имя, string раса, string класс) {
        this->имя = имя;
        this->раса = раса;
        this->класс = класс;
        здоровье = 100;
        урон = 10;
        уровень = 1;
        cout << "Персонаж " << имя << " создан!";
    }
};

// В меню создания персонажа:
Персонаж мойГерой("Артас", "Человек", "Паладин");
```

Аналогия:

- **Skyrim**: Меню создания персонажа = конструктор

- **Dark Souls**: Выбор класса в начале = передача параметров в конструктор

7. Геттеры/Сеттеры = Настройка персонажа

```
class Персонаж {
private:
    int уровеньСложности;

public:
    // Геттер - получить значение
```

```
    int ПолучитьСложность() {
        return уровеньСложности;
    }

    // Сеттер - установить значение с проверкой
    void УстановитьСложность(int сложность) {
        if (сложность >= 1 && сложность <= 10) {
            уровеньСложности = сложность;
        } else {
            cout << "Некорректная сложность!";
        }
    }
};
```

В играх:

- **Настройки графики**: нельзя поставить 200 FPS, если монитор на 144 Гц

- **Характеристики**: сила не может быть отрицательной

Принципы SOLID для геймдизайна:

S - Single Responsibility (Единая ответственность)

Каждый класс делает **одну вещь**: - ЗвуковойДвижок - только звуки - ГрафическийДвижок - только графика
- ИгроваяЛогика - только правила игры

O - Open/Closed (Открыт для расширения, закрыт для изменений)

Добавляем новых персонажей **без изменения старых кодов**:

```
// Можно добавить нового героя, не меняя бой
class НовыйГерой : public Герой {
    // Добавляем новые способности
};
```

L - Liskov Substitution (Подстановки Лисков)

Любого наследника можно подставить вместо родителя: - Если Тролль наследуется от Враг, то везде где нужен Враг, можно использовать Тролль

I - Interface Segregation (Разделение интерфейсов)

Лучше много маленьких интерфейсов, чем один большой:

```
// Плохо: Воин должен уметь всё
class УниверсальныйГерой {
    void Атаковать();
    void Колдовать();
    void Лечить();
    void Красться();
};

// Хорошо: Разделили на интерфейсы
class МожетАтаковать { void Атаковать(); };
class МожетКолдовать { void Колдовать(); };
class МожетЛечить { void Лечить(); };

class Воин : public МожетАтаковать { ... };
class Маг : public МожетАтаковать, public МожетКолдовать { ... };
```

D - Dependency Inversion (Инверсия зависимостей)

Зависи от абстракций, а не конкретных классов:

```
// Плохо: Зависим от конкретного врага
void Битва(Орк враг) { ... }

// Хорошо: Зависим от интерфейса
void Битва(Враг враг) { ... }
// Можем передать Орка, Тролля, Дракона - любого врага
```

Реальный пример: Система заклинаний в Hogwarts Legacy

```
// Базовый класс заклинания
class Заклинание {
protected:
    string название;
    int урон;
    int стоимостьМаны;

public:
    virtual void Произнести() = 0;  // Абстрактный метод
};

// Наследники - конкретные заклинания
class Экспеллиармус : public Заклинание {
    Экспеллиармус() {
        название = "Экспеллиармус";
        урон = 25;
        стоимостьМаны = 10;
```

```
    }

    void Произнести() override {
        cout << "Экспеллиармус!";  // Полиморфизм!
        // Уникальная логика заклинания
    }
};

class Люмос : public Заклинание {
    Люмос() {
        название = "Люмос";
        урон = 0;  // Не наносит урон
        стоимостьМаны = 5;
    }

    void Произнести() override {
        cout << "Люмос!";
        ОсветитьОбласть();  // Совсем другая логика
    }
};

// Инкапсуляция - палочка управляет заклинаниями
class ВолшебнаяПалочка {
private:
    vector<Заклинание*> известныеЗаклинания;

public:
    void ВыучитьЗаклинание(Заклинание* заклинание) {
        известныеЗаклинания.push_back(заклинание);
    }

    void Произнести(string названиеЗаклинания) {
        // Находим нужное заклинание и произносим
        for (auto заклинание : известныеЗаклинания) {
            if (заклинание->название == названиеЗаклинания) {
                заклинание->Произнести();  // Полиморфизм!
                return;
            }
        }
    }
};
```

Твоя игра - твоя

ООП-практика:
Начни с простого: Класс Герой с полями здоровье, урон
Добавь инкапсуляцию: Сделай поля приватными, добавь методы ПолучитьУрон(), Лечить()
Создай наследников: Воин, Маг, Лучник от Герой
Примени полиморфизм: Метод Умение() по-разному работает для каждого класса
Собери всё вместе: Создай несколько объектов и заставь их сражаться

Почему это круто:

- **Переиспользование кода**: Написал класс Враг один раз - создавай орков, гоблинов, драконов
  Лёгкая поддержка: Исправил баг в Герое - он исправился у всех наследников
  Гибкость: Добавил нового босса за 5 минут, унаследовав от Враг
  Читаемость: Код выглядит как описание игры: герой.Атаковать(враг)

Задание-челлендж:

Создай мини-игру с классами: 1. Персонаж (базовый) 2. Игрок и Враг (наследники) 3. Воин, Маг, Лучник (наследники Игрок) 4. Орк, Тролль, Дракон (наследники Враг)

Сделай так, чтобы: - Каждый класс имел уникальную способность - Можно было легко добавить нового персонажа - Код читался как сценарий битвы

Помни: Каждая игра, в которую ты играешь - это тысячи классов и объектов, взаимодействующих по правилам ООП! Ты уже понимаешь, как работают твои любимые игры изнутри