Технологии программирования

Лекция 11 | ECS как альтернатива ООП

Лектор: Киселев Станислав Дмитриевич

ООП в контексте дизайна кода

Entity Centric Thinking и его проблемы

Мы здесь

От моделей к компонентам

Entity Component, плюсы и минусы

Важность производительности

Кэш-промахи, разница в типах памяти и варианты ускорения кода

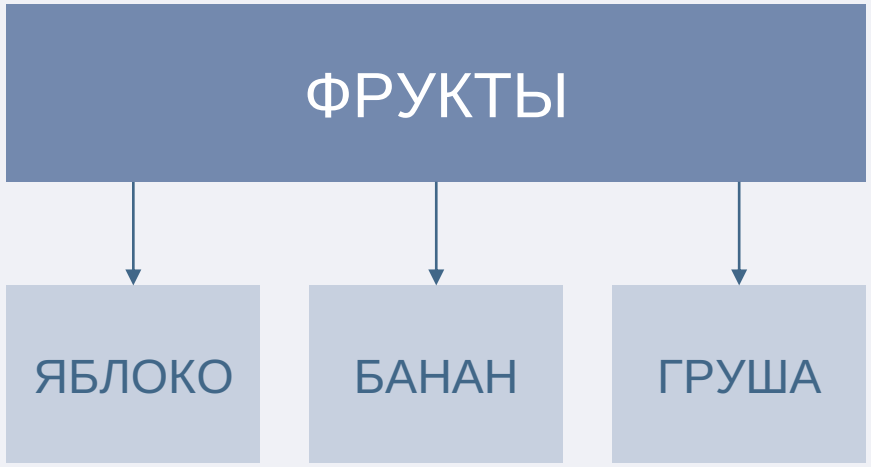
Data Oriented Design и Entity Component System

Что такое ECS, применение и отличия от ООП

Заключение

Сравнение подходов, когда и что выбирать

Вспомним ООП



Основные моменты:

- Программы – **совокупности объектов**, взаимодействующих между собой
- Каждый объект – экземпляр определенного класса, который **определяет свойства** и **поведение** объекта
- Условно основан на **моделировании реального мира**

Например, в программе управления библиотекой:

В ООП будет класс "Книга", который описывает свойства (название, автор, год издания) и методы (например, метод для выдачи книги)

Каждый экземпляр этого класса – конкретная книга

ООП в контексте дизайна кода

Под **дизайном кода (code design)** будем подразумевать способ организации структуры, взаимодействия и архитектуры приложения

Который определяет не только то, как будут реализованы функции, но и то, что они делают

Дизайн код ориентируется на следующие параметры:

Читаемость кода	Масштабируемость	Поддерживаемость	Производительность
как легко понять логику	как просто добавить новую функциональность	как быстро исправить баги или адаптировать код к изменениям	насколько эффективно работает система

В контексте нашей лекции ООП – это не парадигма, а выбор **архитектурного подхода** и **паттернов**, которые решают конкретные проблемы проектирования

Эволюция подходов

Entity-Centric (ООП-подход)

Сущности – самостоятельные объекты с данными и поведением классов

Entity Centric Thinking (ООП)

> **Entity-Centric (ООП) –** это подход к проектированию программных систем, в котором сущности (entities) рассматриваются как центральные и самостоятельные объекты, обладающие собственным поведением и данными

Этот подход тесно связан с классическим ООП (объектно-ориентированным программированием), где сущности часто реализуются через классы с методами (логикой) и полями (данными)

Основные характеристики:

Сущности как самодостаточные объекты

Наследование и иерархии

Жесткая связность

Основные характеристики Entity-Centric подхода

Сущности как самодостаточные объекты	Каждая сущность содержит и данные, и логику Например, класс Робота-Доставщика может иметь методы move(), delivery(), а класс Клиента – get_delivery(), move()
Наследование и иерархии	Часто используется глубокое наследование: Object → LivingEntity → RobotPlatform → MovableRobot → DeliveryRobot Это приводит к взрыву иерархий и сложности добавления новых функций
Жесткая связность	Поведение сущности заложено в её класс Например, если нужно добавить интеграцию со зданием, приходится: 1. Создавать подкласс RobotSomeBuildingIntegration 2. Использовать множественное наследование или интерфейсы

Проблемы ООП

```
// Класс " DeliveryRobot "
с наследованием
и собственной логикой

class DeliveryRobot: public
MovableRobot  {
    void move() { ... }
    void navigate() { ... }
    void delivery() { ...}
}
```

Хрупкие иерархии

Добавление новой функциональности требует изменения существующих классов или создания новых подклассов

Проблемы производительности

Данные сущностей разрознены в памяти (например, объекты Robot, Obstacle и Client хранятся в разных местах), что вызывает **cache misses**

Сложность композиции

Невозможно гибко комбинировать поведение (например, без лишних классов создать объект, который и доставляет, и интегрирован со зданием)

Нарушение SRP (Single Responsibility Principle)

Классы обрастают несвязанной логикой (например, DeliveryRobot отвечает и за навигацию, и за доставку, и за оповещение клиента)

Какие проблемы со стороны архитектуры есть в данном коде?

```
class Enemy {
   protected:
      virtual void update() = 0;
}
class Dragon: public Enemy {
   private:
      int health;
      float positionX, positionY;
   public:
      void fly() { ... }
      void breatheFire() { ... }

      void update() override {
         fly();
         breatheFire();
         super.update();
      }
}
```

Какие проблемы со стороны архитектуры есть в данном коде?

```
class Enemy {
    protected:
        virtual void update() = 0;
}
class Dragon: public Enemy {
    private:
        int health;
        float positionX, positionY;
    public:
        void fly() { ... }
        void breatheFire() { ... }

        void update() override {
            fly();
            breatheFire();
            super.update();
        }
}
```

Если нужно добавить плавание, придется создавать класс SwimmingDragon или нарушать SOLID

Код дублируется между классами FlyingEnemy и SwimmingEnemy

Сложно создать гибрид
(например, дракон, который летает и плавает)

Общие проблемы Entity Centric

Негибкость для сложных систем

В играх или симуляциях с тысячами сущностей иерархии классов становятся неуправляемыми

Потребность в оптимизации

Современные процессоры требуют data locality (плотное хранение данных),
а ООП его нарушает

Появление альтернатив

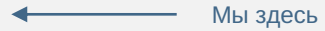
Data-Oriented Design предложил разделение данных и логики, динамическую композицию через компоненты и оптимизацию под кэш-память

ООП в контексте дизайна кода

Entity Centric Thinking и его проблемы

От моделей к компонентам

Entity Component, плюсы и минусы



Важность производительности

Кэш-промахи, разница в типах памяти и варианты ускорения кода

Data Oriented Design и Entity Component System

Что такое ECS, применение и отличия от ООП

Заключение

Сравнение подходов, когда и что выбирать

Причины перехода от моделей к компонентам

1
2
3
4

Эволюция сложности систем

Ранние проекты:

Простые объекты с четкими иерархиями (например, Player, Enemy)

Современные системы:

Тысячи взаимодействующих сущностей (игры, симуляции, IoT), где:

- Жесткие иерархии ООП стали неуправляемыми
- Требуется динамическая композиция
  (например, «объект, который и летает, и стреляет, и горит»)

Причины перехода от моделей к компонентам

Потребность в гибкости

Ранние проекты:

Ранее большинство приложений писались строго под узкую задачу и были заточены под неё

Современные системы:

Сейчас появились области знаний, при работе с которыми требуется большая гибкость и производительность с маштабируемостью

Например, симуляции реального мира для тестирования или ML

Причины перехода от моделей к компонентам

Требования к производительности

Ранние проекты:

Раньше не было обилия высоконагруженных систем, требующих минимальный отклик и полную отказоустойчивость

Для получения производительности код писался под конкретное железо, что давало большое преимущество

Современные системы:

Появилась необходимость в обработке миллиардов элементов при минимизации потребляемых ресурсов

Причины перехода от моделей к компонентам

1
2
3
4

Параллелизм и масштабируемость

Объекты с общим состоянием порождали гонки данных, требуя блокировок (мьютексы), что замедляло выполнение

Сложность в корректном расширении архитектуры

Современные системы:

Теперь компоненты системы обрабатываются как изолированные массивы, что позволяет запускать их в параллельных потоках без блокировок

Сущности динамически собираются из компонентов, а системы масштабируются линейно даже для миллионов объектов

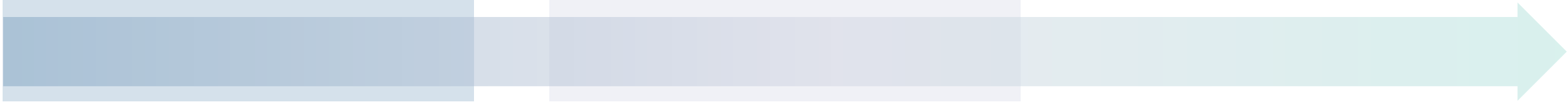
Эволюция подходов

Entity-Centric (ООП-подход)

Сущности – самостоятельные объекты с данными и поведением классов

Entity-Component (Гибридный подход)

Сущности – контейнеры компонентов, но логика может быть внутри сущностей или систем



Entity-Component (Гибридный подход)

Entity Component подразумевает архитектурный подход, в котором объекты (сущности) разбиваются на отдельные составляющие (компоненты), хранящие данные, с минимумом встроенной логикой – такие компоненты являются «кусочками» описания объекта

Ключевыми характеристиками являются:

Сущность фактически служит контейнером или идентификатором

Компоненты отвечают за состояние (данные), без собственной логики поведения

Логика предполагает быть встроенной непосредственно в сущность

Главным отличием от Entity Centric является использование агрегирования вместо наследования!

Пример симуляции | Роботы в заводском цеху

Сущности (Entity):

- Робот-сварщик
- Робот-погрузчик
- Конвейер

Компоненты (Component):

- Position (координаты)
- Battery (заряд батареи)
- Welder (мощность сварки)
- Movement (скорость)

```
// Компоненты
struct Position { float x, y; };
struct Battery { int charge; };
struct Welder { int power; };

// Сущность "Робот-сварщик"
Entity weldingRobot = world.create();
weldingRobot.add<Position>(10.0f, 5.0f);
weldingRobot.add<Battery>(100);
weldingRobot.add<Welder>(200);
// Система движения (логика)

void MovementSystem(World& world) {
   for (auto& [entity, pos, battery] : world.query<Position,
Battery>()) {
      if (battery.charge > 0) {
         pos.x += 1.0f; // Робот движется
         battery.charge -= 1;
      }
   }
}
```

Плюсы Entity-Component

Гибкость

Можно создать гибридного робота, добавив компоненты

```
Entity hybridRobot = world.create();
hybridRobot.add<Position>(...);
hybridRobot.add<Welder>(...);
hybridRobot.add<Movement>(...); // Теперь это "мобильный сварщик"
```

Отсутствие жестких иерархий

Нет проблем с множественным наследованием (в отличие от ООП)

Упрощенное тестирование

Компоненты изолированы (например, Battery можно тестировать отдельно)

Недостатки Entity-Component

Сложность управления зависимостями:

Если система Movement требует Position и Battery, а у сущности нет Battery → ошибка или проверки в runtime

Дублирование логики:

В EC логика может оставаться внутри сущностей (например, robot.update()), что нарушает принцип ECS

!

Производительность:

- Cache misses: Компоненты могут храниться в разных местах памяти
  Пример: Position и Battery робота лежат далеко друг от друга → процессор загружает данные медленнее
- Системы работают с разрозненными данными

ООП в контексте дизайна кода

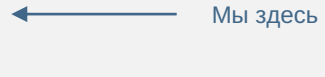
Entity Centric Thinking и его проблемы

От моделей к компонентам

Entity Component, плюсы и минусы

Важность производительности

Кэш-промахи, разница в типах памяти и варианты ускорения кода



Data Oriented Design и Entity Component System

Что такое ECS, применение и отличия от ООП

Заключение

Сравнение подходов, когда и что выбирать

Почему все упирается в производительность?

Железо не успевает за софтом

Закон Мура замедляется: CPU-частоты почти не растут, но объемы данных увеличиваются
Решение: Оптимизация кода (кэш-дружелюбные структуры, параллелизм)

Современные задачи требуют скорости

- Игры: 60+ FPS для VR/AR
- Наука: симуляции атомов, климата, ИИ
- Big Data: обработка миллионов запросов в секунду

Энергоэффективность

Мобильные устройства и облака экономят батарею и электричество

Плохой дизайн кода увеличивает энергопотребление CPU на 30-50%

Что такое кэш-промахи (Cache Misses)

> **Cache Miss (промах кэша)** случается, когда запрашиваемые данные отсутствуют в кэше и их нужно подгружать из основного источника

Процессоры имееют кэш, в котором хранится недавно прочитанные блоки оперативной памяти и обращение к кэшированным данным происходит существенно быстрее, чем «перекачивание» этих же данных из оперативной памяти на кристалл процессора

Почему это проблема?

Процессор работает в 100 раз быстрее RAM

↓

Каждый промах – это простои

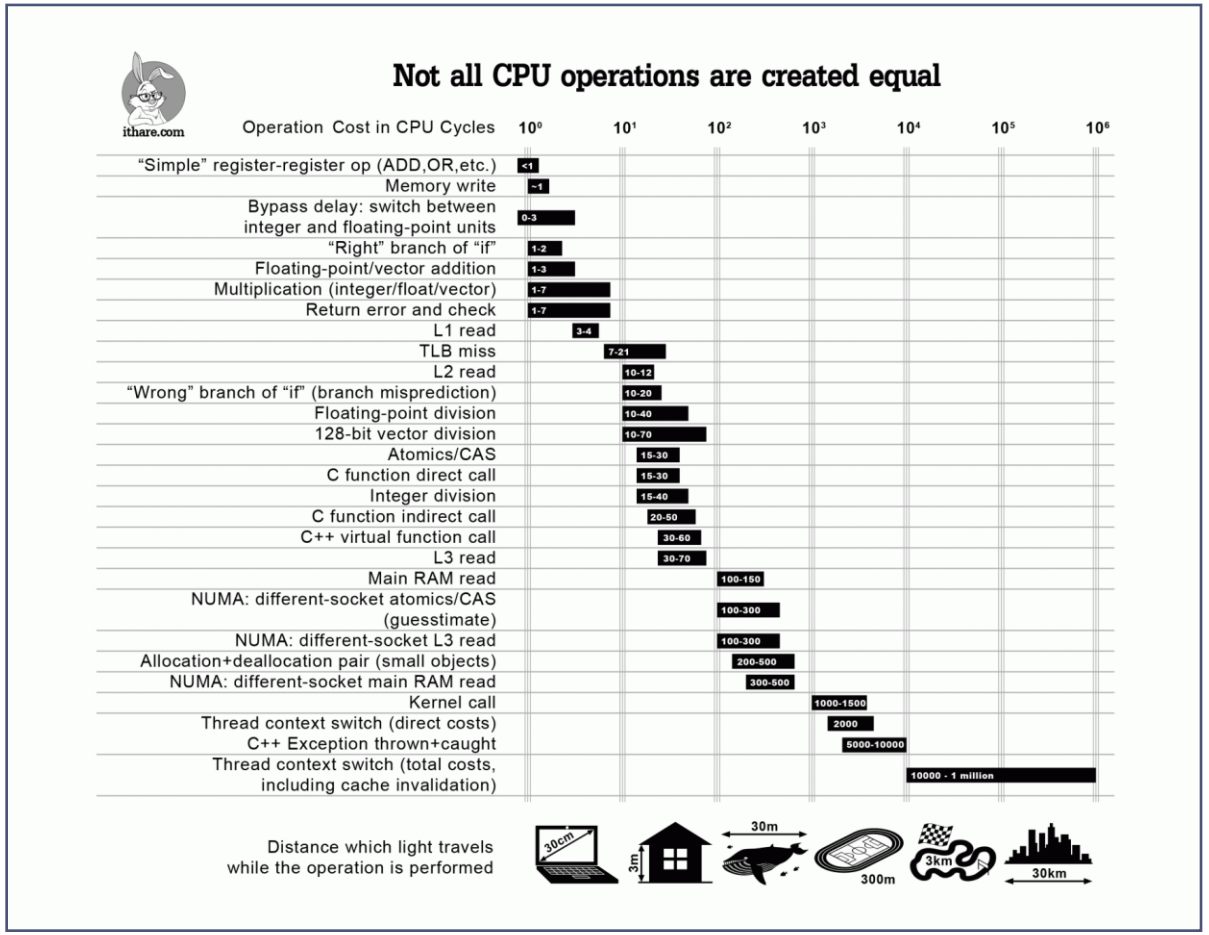
Пример: Если игра обрабатывает 10 000 объектов, и у каждого промах – FPS падает

Что такое кэш-промахи (Cache Misses)

```
const int count  = 1000000;
int *staticRefs[count];
int *dynamicRefs[count];
int staticArray[count];

int main() {
    for(int i = 0; i < count; i++)  {
        staticArray[i] = 0;
        staticRefs[i] = &staticArray[i];

        int *val = new int(0);
        dynamicRefs[i] = val;
    }
```

```
    //первый цикл
        for(int i = 0; i < count; i++)  { (*staticRefs[i])++; }

    //второй цикл
        for(int i = 0; i < count;
    i++)  {    (*dynamicRefs[i])++;  }

    }
```

Доступ к разной памяти в тактах процессора



Статья о тактах процессора

Варианты ускорения системы

Замена линейного поиска O(n) на бинарный O(log n)

Использование хеш-таблиц с доступом O(1) вместо списков

Сортировка данных для быстрого доступа

размер / сложность	10	20	30	40	50	60
n	0,00001 сек.	0,00002 сек.	0,00003 сек.	0,00004 сек.	0,00005 сек.	0,00005 сек.
n^2	0,0001 сек.	0,0004 сек.	0,0009 сек.	0,0016 сек.	0,0025 сек.	0,0036 сек.
n^3	0,001 сек.	0,008 сек.	0,027 сек.	0,064 сек.	0,125 сек.	0,216 сек.
n^5	0,1 сек.	3,2 сек.	24,3 сек.	1,7 минут	5,2 минут	13 минут
2^n	0,0001 сек.	1 сек.	17,9 минут	12,7 дней	35,7 веков	366 веков
3^n	0,059 сек.	58 минут	6,5 лет	3855 веков	$2x10^8$ веков	$1,3x10^{13}$ веков

Варианты ускорения системы

Уменьшение cache misses
за счет локализации данных

Обработка массивов
последовательно
(а не вразброс)

Хранение компонентов
отдельными массивами
(Position[], Health[])

Struct of Arrays (SoA)

```
// Плохо (AoS): данные разрознены
struct Object { float x, y; int health; };
Object objects[1000];

// Хорошо (SoA): данные плотно упакованы
struct Objects { float x[1000], y[1000]; int health[1000]; };
```

Варианты ускорения системы

Memory Pool

Заранее выделенный пул памяти для объектов одного типа

Плюсы:

- Исключает фрагментацию и аллокацию / освобождение в runtime
- Данные хранятся плотно → меньше промахов доступа к кэшу

```
class MemoryPool {
    std::vector<Enemy> pool; // Заранее выделенная память
    std::stack<Enemy*> freeList; // Свободные объекты

public:
    Enemy* create() {
        Enemy* obj = freeList.top();
        freeList.pop();
        return obj;
    }

    void destroy(Enemy* obj) { freeList.push(obj); }
};
```

Варианты ускорения системы

Современные задачи, такие как симуляции реального мира (климатические модели, физические системы) и обработка Big Data (анализ транзакций, машинное обучение), требуют обработки миллионов сущностей с минимальными задержками

Классический Entity-Centric подход (ООП) оказался **неспособен масштабироваться** для таких сценариев из-за фундаментальных проблем в дизайне кода

Это заставило индустрию развивать подходы, которые переосмысливают работу с данными и логикой, так что классические подходы уступают место новым:

- Data-Oriented Design (оптимизация под кэш)
- Entity Component (разделение данных и логики)
- Функциональному программированию (иммутабельность + параллелизм)

Эволюция подходов

Entity-Centric (ООП-подход)

Сущности – самостоятельные объекты с данными и поведением классов

Entity-Component (Гибридный подход)

Сущности – контейнеры компонентов, но логика может быть внутри сущностей или систем

Data-Oriented Design (ECS)

Сущности = уникальные ID

Компоненты = чистые данные (структуры)

Системы = логика, обрабатывающая компоненты

ООП в контексте дизайна кода

Entity Centric Thinking и его проблемы

От моделей к компонентам

Entity Component, плюсы и минусы

Важность производительности

Кэш-промахи, разница в типах памяти и варианты ускорения кода

Data Oriented Design и Entity Component System

Что такое ECS, применение и отличия от ООП

Заключение

Сравнение подходов, когда и что выбирать

Data-Oriented Design (DOD)

Data-Oriented Design (DOD) – это подход к проектированию кода, который:

- Фокусируется на данных, а не на объектах
- Оптимизирует работу памяти для минимизации задержек доступов
- Обрабатывает данные пачками (например, все Position сразу), а не по одному объекту

Ключевой принцип:

«Не спрашивай что умеет объект, спрашивай что нужно сделать с данными»

ООП vs DOD

ООП	DOD
«У каждого объекта есть методы update(), render()»	«Есть массив позиций – обработаем их все»
Данные разрознены (x, health, sprite в одном объекте)	Данные сгруппированы (x[], health[] отдельно)
Код думает об «абстракциях» (иерархии классов)	Код думает о «трансформациях данных» (системы)

Entity Component System

Entity-Component-System – это архитектурный паттерн, созданный специально для разработки игр, который отлично подходит для описания динамического виртуального мира

ECS возводит в абсолют принцип **Composition Over Inheritance** (композиция важнее наследования) и может являться частным примером **Data Oriented Design** (ориентированного на данные дизайна), однако это уже зависит от интерпретации паттерна конкретной реализацией

ECS можно и нужно комбинировать в другими подходами,
извлекая максимальную производительность и гибкость в настройки системы

Entity Component System

Entity	Сущность, максимально абстрактный объект Условный контейнер для свойств, определяющих чем будет являться эта сущность Зачастую представляется в виде идентификатора для доступа к данным
Component	Компонент, свойство с данными объекта Компоненты в ECS должны содержать исключительно **чистые данные, без единой капли логики**
System	Система, логика обработки данных Системы в ECS не должны содержать **никаких данных, только логика обработки данных**

Архитектура Entity Component System

Порядок исполнения:

1 Создание сущностей

Мир (World) генерирует ID и добавляет компоненты

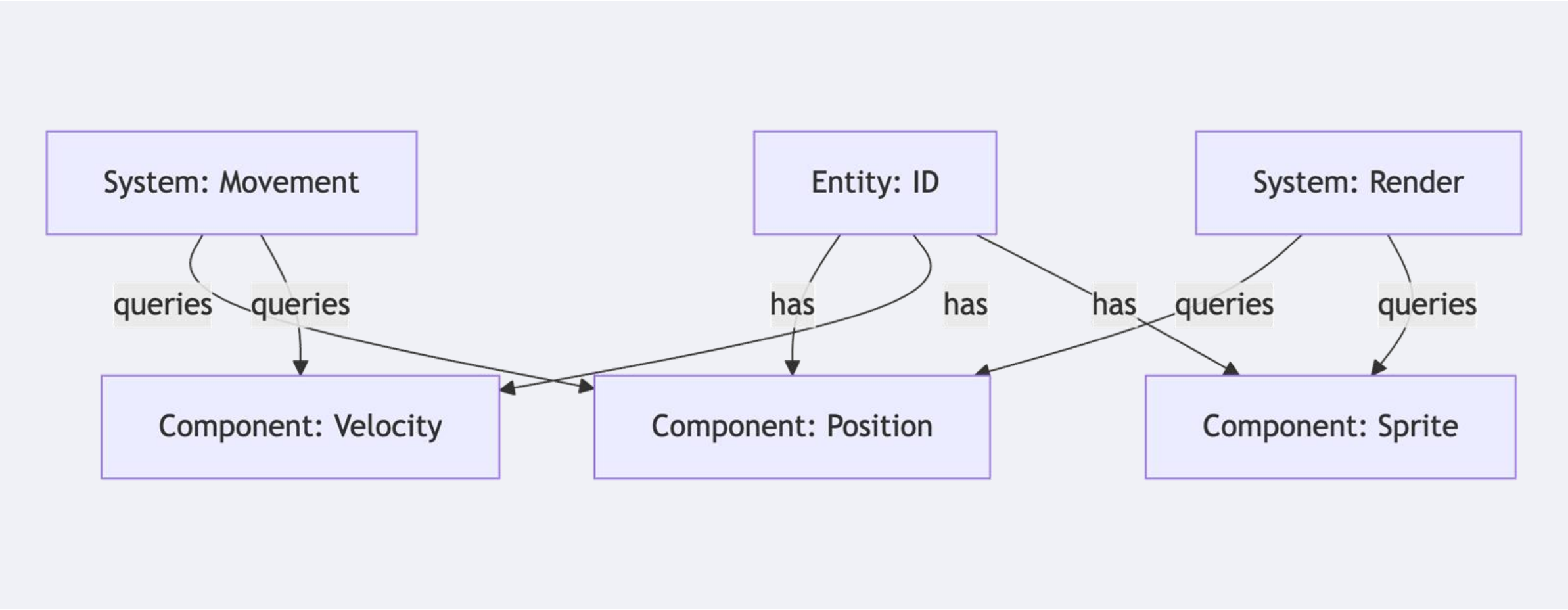
2 Запуск систем

Каждая система обрабатывает только нужные компоненты

3 Типичный кадр игры

```
# Псевдокод
world.update() {
    MovementSystem.run()
    # Обновляем позиции

    CollisionSystem.run()
    # Проверяем столкновения

    RenderSystem.run()
    # Рисуем спрайты
}
```

Архитектура Entity Component System



Архитектура Entity Component System

Система (System) – это чистая логика, которая выборочно обрабатывает компоненты, не храня состояния

Определение и ключевые свойства:

Не имеет данных	Сфокусирована на одной задаче	Работает с группами сущностей
Только читает или изменяет компоненты	Например, MovementSystem двигает сущности, а RenderSystem рисует их	Выбирает все сущности с нужным набором компонентов

Примеры систем

MovementSystem – обновляет Position на основе Velocity

```
void MovementSystem(World& world) {
    for (auto& [entity, pos, vel] : world.query<Position, Velocity>()) {
        pos.x += vel.dx * deltaTime; pos.y += vel.dy * deltaTime;
    }
}
```

Примеры систем

RenderSystem - отрисовка всех сущностей с Position и Sprite

```
void RenderSystem(World& world) {
    for (auto& [entity, pos, sprite] : world.query<Position, Sprite>()) {
        drawSprite(sprite.texture, pos.x, pos.y);
    }
}
```

Примеры систем

CollisionSystem - проверка столкновений между Hitbox и Health

```
void CollisionSystem(World& world) {
    for (auto& [entity1, hitbox1] : world.query<Hitbox>()) {
        for (auto& [entity2, hitbox2, health] : world.query<Hitbox, Health>()) {
            if (checkCollision(hitbox1, hitbox2)) {
                health.value -= 10;
            }
        }
    }
}
```

Плюсы ECS

Производительность

- Данные хранятся плотно (SoA) → минимум кэш-промахов
- Системы параллелятся (например, Movement и AI работают одновременно)

Гибкость

Добавить полет = прикрепить Flyable компонент + FlightSystem

Масштабируемость

1000 или 1 000 000 сущностей — код не меняется

Минусы ECS

Сложность обучения

Неочевидно для разработчиков, привыкших к ООП

Оверхеды на организацию

Требуются менеджеры сущностей / компонентов (например, entt)

Дебаггинг

Нет «объектов» → сложнее трассировать связи

Реализация Entity Component System

```cpp
// Компоненты (чистые данные)
struct Position { float x, y; };
struct Velocity { float dx, dy; };
struct Sprite { std::string texture; };
struct Health { int value; };

// Сущность - просто ID
using Entity = uint32_t;

// Мир (хранит сущности и компоненты)
class World {
public:
    Entity create() {
        return nextEntityId++;
    }

    template<typename T>
    void add(Entity e, T component) {
        components<T>[e] = component;
    }

    template<typename T>
    T& get(Entity e) {
        return components<T>[e];
    }
    template<typename... Ts>
    auto query() {
        std::vector<std::tuple<Entity, Ts&...>> result;
        for (auto& [e, _] : components<Position>) { // Используем Position как "маркер"
            if ((has<Ts>(e) && ...)) { result.emplace_back(e, get<Ts>(e)...);  } }
        return result;
    }
private:
    Entity nextEntityId = 0;
    template<typename T>
    std::unordered_map<Entity, T> components;
    template<typename T>
    bool has(Entity e) {
        return components<T>.count(e);
    }
};
```

Реализация Entity Component System

```
// Система движения
class MovementSystem {
public:
    void run(World& world, float deltaTime) {
        for (auto& [entity, pos, vel] : world.query<Position,
Velocity>()) {
            pos.x += vel.dx * deltaTime;
            pos.y += vel.dy * deltaTime;
        }
    }
};

// Система рендеринга
class RenderSystem {
public:
    void run(World& world) {
        for (auto& [entity, pos, sprite] : world.query<Position,
Sprite>()) {
            std::cout << "Draw " << sprite.texture
                      << " at (" << pos.x << ", " << pos.y << ")\n";  } } };
```

```
// Система столкновений
class CollisionSystem {
public:
    void run(World& world) {
        auto entities = world.query<Position, Health>();
        for (auto& [e1, pos1, health1] : entities) {
            for (auto& [e2, pos2, health2] : entities) {
                if (e1 != e2 && checkCollision(pos1, pos2)) {
                    health1.value -= 10;
                    std::cout << "Collision! Health: " << health1.value <<
"\n";
                }
            }
        }
    }

private:
    bool checkCollision(Position& a, Position& b) {
        return std::abs(a.x - b.x) < 1.0 && std::abs(a.y - b.y) < 1.0;
    } };
```

Реализация Entity Component System

```
World world;

// Создаем сущности
Entity player = world.create();
world.add(player, Position{0, 0});
world.add(player, Velocity{1, 0});
world.add(player, Sprite{"player.png"});
world.add(player, Health{100});

Entity enemy = world.create();
world.add(enemy, Position{10, 0});
world.add(enemy, Health{50});

// Системы
MovementSystem movement;
RenderSystem render;
CollisionSystem collision;
```

```
// Игровой цикл
for (int i = 0; i < 3; ++i) {
    movement.run(world, 1.0f); // deltaTime = 1.0
    collision.run(world);
    render.run(world);
}
```

Как это работает?

1 Создание сущностей

- player (ID 0) с компонентами Position, Velocity, Sprite, Health
- enemy (ID 1) с Position и Health

2 Запуск систем

- MovementSystem двигает игрока по Velocity.
- CollisionSystem проверяет столкновения (условно).
- RenderSystem рисует только сущности с Position и Sprite.

3 Типичный кадр игры

- За 3 итерации игрок перемещается в (3, 0), теряет здоровье при столкновении.

ООП в контексте дизайна кода

Entity Centric Thinking и его проблемы

От моделей к компонентам

Entity Component, плюсы и минусы

Важность производительности

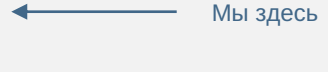
Кэш-промахи, разница в типах памяти и варианты ускорения кода

Data Oriented Design и Entity Component System

Что такое ECS, применение и отличия от ООП

Заключение

Сравнение подходов, когда и что выбирать



ЗАКЛЮЧЕНИЕ

Итоги

ECS – это специализированный инструмент, как GPU для графики

ООП остаётся стандартом для 90% задач

Идеальный выбор зависит от:

- Производительности (ECS для FPS/симуляций)
- Гибкости (ECS для динамических объектов)
- Сложности (ООП для быстрого прототипирования)

«Используйте ECS там, где он решает проблемы, а не потому что это модно»

ЗАКЛЮЧЕНИЕ

Когда выбирать ООП

1 Не критична производительность:

- Визуальные редакторы, инструменты
- Мобильные приложения без сложной логики

2 Четкие иерархии:

- GUI (кнопки, окна, формы)
- Бизнес-логика (банки, CRM)

Примеры:

Игры: 2D-платформеры (Mario)

Frontend: React-компоненты

AI: Простые NPC в RPG

Когда выбирать ECS

1 Высокие требования к скорости:

- 60+ FPS в играх (шутеры, MMO)
- Научные расчеты (физика, химия)

2 Динамические сущности:

- Объекты с меняющимся поведением (например, юнит, который может летать/плавать/невидим)
- Симуляции с случайными событиями (погода, катастрофы)

Примеры:

Игры: StarCraft II, Minecraft с модами

Симуляции: Климатические модели, робототехника

AI: Стратегии в реальном времени (RTS)

ЗАКЛЮЧЕНИЕ

Когда что лучше выбирать

Критерий	ООП (Классы/Объекты)	ECS (Сущности-Компоненты-Системы)
Игры	Простые игры (платформеры, визуальные новеллы)	AAA-игры, симуляции (1000+ сущностей)
Симуляции	Маломасштабные (физика 10 тел)	Крупные (погода, биология, трафик)
Frontend State	Подходит (React/Vue компоненты)	Избыточно (редко нужна оптимизация)
AI	Простые поведенческие деревья	Сложные ИИ-системы (рои, стратегии)
Производительность	Средняя (GC, cache misses)	Высокая (DOD, parallel systems)
Гибкость	Жесткие иерархии	Динамическая композиция компонентов
Сложность	Просто освоить	Высокий порог входа

Гибридные подходы

ООП + ECS:

- Игры: Unity
  (обычные GameObject + DOTS для оптимизации)
- Frontend: Redux
  (стейт-менеджмент похож на ECS)

Когда смешивать:

Основная логика на ООП, критичные части на ECS

Спасибо за внимание!
Ваши вопросы?