Основи на ООП дизайна.

титуляр на курса: д-р Тодор Цонков (ttsonkov@gmail.com)

От какво ще се състои настоящата лекция?

Dependency. Ownership. Композиция и агрегация. Design guidelines. std::variant.

Какво представляват връзките между обекти?

В обектно-ориентирания дизайн не е важно само какво прави класът, а и:

Как е свързан с други класове?
Кой притежава кого?
Колко силна е зависимостта?

Основни понятия:
Dependency
Ownership
Composition
Aggregation

Dependency

```
class Logger {
public:
    void log(const std::string& msg);
};

class StudentService {
public:
    void enrollStudent(const std::string& name, Logger&
logger) {
        logger.log("Enrolling " + name);
    }
};
```

Dependency = клас A използва клас B временно

Характеристики:
краткотрайна
без ownership
най-слабата форма на връзка

Dependency - добри практики

Добри практики:
Dependency чрез параметри
Dependency чрез абстракции (interfaces)
Минимална видимост (forward declaration)

Лоши практики:
#include навсякъде
Скрити зависимости
Създаване на обекти вътре в метода

```
void foo() {
    Logger l; // ❌ hard dependency
}
```

Детайли

Какво наистина е Dependency?

Dependency означава:
Клас A не може да бъде компилиран / използван / тестван без клас B

Важно:
Dependency НЕ означава ownership
Dependency НЕ означава дълготрайна връзка
Dependency е използване, не притежание

```
//Antipattern
class StudentService {
public:
    void enroll(const std::string& name) {
        Logger logger;  // ❌
        logger.log(name);
    }
};
```

Ownership

Дефиниция

В C++ ownership (собственост) описва кой обект или променлива е отговорен за живота на даден ресурс и кой трябва да го освободи.

Класическият пример е динамично разпределена памет: ако един обект притежава указател, той трябва да се грижи за извикването на delete или да използва смарт указател (std::unique_ptr, std::shared_ptr) за автоматично управление на ресурса. Правилното управление на ownership е ключово за избягване на memory leaks, dangling pointers и двойно изтриване.

Пример

```
class School {

std::vector<std::unique_ptr<Stud
ent>> students_;
public:  void
addStudent(std::unique_ptr<Stud
ent> s) {
students_.push_back(std::move(s
));
 }
};
```

Smart Pointers

std::unique_ptr → един собственик

std::shared_ptr → споделена собственост

raw pointer / reference → НЕ притежава

Ownership - последователност

Златно правило: Винаги да се започва от най-простото и добавя сложност само при нужда. Ако искаме да имаме член данна на класа я правим:

1)Стойност (Value): Винаги, когато обектът е малък или лесен за местене. Това е най-бързият и безопасен код.

2)unique_ptr: Когато обектът е голям, полиморфен или трябва да живее извън живота на класа, но има само един собственик.

3)shared_ptr: Само ако обектът има повече от един собственик (например в графи или кешове или асинхронно програмиране). Ако се ползва "за всеки случай", вероятно има проблем с дизайна.

4)Референция (T&) / string_view: Когато функцията само „ползва“ обекта за малко и той гарантирано съществува.

5)Raw Pointer(T*)/Observer_ptr(C++23) : Когато е „опционален“ (може да е nullptr), но не го притежаваш, а друг се грижи за живота му.

Композиция и агрегация

Какво е композиция?

1)Обектът притежава другия обект

2)Животът на „частта“ е строго зависим

3)Унищожаването на цялото ⇒ унищожава и частите

4)Най-често се реализира чрез стойност (value member)

Основна идея

```
class Address {
    std::string city;
};

class Student {
    Address address;
};
```

Address се създава заедно със Student
Не може да бъде nullptr
Не се споделя с други обекти
Унищожава се автоматично (RAII)

Композиция и агрегация

Какво е агрегация?

1)Обектът НЕ притежава другия

2)Всеки има собствен жизнен цикъл

3)Често се реализира чрез:
указател, референция

4)Позволява споделяне на обекта

Основна идея

```
class Course {};

class Student {
    Course* course; // агрегация
};
```

Student не създава Course
Student не го унищожава
Course може да се споделя между много студенти
Възможно е course == nullptr

Предимства и недостатъци

Ако обектът не може да съществува без друг → композиция

Ако само го използва → агрегация

Недостатъци и рискове:

1)нужда от проверки за nullptr
2)неясен ownership
3)възможни dangling pointers

```
class Course {
public:
    void print() const {
        std::cout << "OOP course\n";
    }
};
class Student {
    Course* course;   // агрегация
public:
    Student(Course* c) : course(c)
{}
    void showCourse() const {
        // потенциален проблем
        course->print();     }
};

Student* s;

{
    Course c;
    s = new Student(&c);
} // c е унищожен тук

// undefined behavior
s->showCourse();
```

Кое кога да изберем?

Пример

Временно използване: Dependency

Дълготрайна връзка без ownership: association

Ясен собственик: unique_ptr

Родител → дете Композиция

Лош дизайн

```
class School {
std::vector<Student*> students;
public:
   void addStudent(const
std::string& name) {
      students_.push_back(new
Student(name)); // ❌
   }
};
```

Проблеми:

1)неясен ownership
2)leak
3)трудно тестване
4)няма RAII

Добър дизайн

```
class School {
    std::vector<std::unique_ptr<Student>> students_;
public:
    void addStudent(std::string name) {
        students_.push_back(
            std::make_unique<Student>(std::move(name))
        );
    }
};
```

Design Guidelines

Добър дизайн означава код, който е лесен за разбиране, поддръжка и разширяване. Ето няколко практични design guidelines, които се доказват в реални проекти:

1) Ясна отговорност (Single Responsibility)
Всеки клас и функция трябва да има една основна причина да се променя. Ако класът „знае твърде много“ – време е за разделяне.

2) Слаба свързаност, силна кохезия
Класовете трябва да са възможно най-независими един от друг, но вътрешно логично свързани. Dependency ≠ Ownership.

3) Изразителни интерфейси
API-то трябва да „казва истината“ за поведението си. Ако функцията може да се провали – трябва да се покаже в типа на функцията (`std::expected`, error codes), не в документацията.

Design Guidelines

Продължение...

4) Грешките трябва да са очевидни.
По-добре компилационна грешка, отколкото runtime изненада.
Да се използват `const`, `=delete`, strong typing и RAII.

5) Композицията е за предпочитане пред наследяването
Наследяването създава твърди връзки. Композицията дава гъвкавост и по-лесно тестване.

6) Проектиране за промяна, не за текущия момент
Най-лошият дизайн е този, който „работи идеално... засега“. Важно е как кодът ще изглежда след 2 години.

Пример със School и Student

```
class School {
public:
    std::string name() const { return "High School #1"; }
};

class Student {
    std::string name_;
public:
    explicit Student(std::string name) :
name_(std::move(name)) {}

    void printSchool(const School& school) const {
        std::cout << name_ << " studies at " <<
school.name();
    }
};

//Още по-добър и професионален пример ще разгледаме в
следващите лекции с абстракция.

//Antipattern:

class Student {
    School* school_; // ❌ кой притежава?
};
```

std::variant

std::variant е type-safe обединение (discriminated union / sum type) в C++, въведено в C++17.
Позволява една променлива да съдържа точно една стойност от предварително зададен набор типове, като компилаторът следи кой тип е активен. Вместо: void*, union, йерархии с наследяване, std::any се ползва:

```
std::variant<int, double, std::string>
✔ type-safe
✔ без heap allocation
✔ без RTTI / virtual
✔ compile-time гаранции
```

variant вътрешно съхранява:
1)памет, достатъчна за най-големия тип
2)индекс (кой тип е активен)

Само един тип е активен в даден момент
При смяна → старият обект се унищожава, новият се конструира.

std::variant - пример

```cpp
#include <variant>
#include <iostream>
#include <string>

int main() {
    std::variant<int, double, std::string> v;

    v = 10;
    std::cout << std::get<int>(v) << "\n";

    v = 3.14;
    std::cout << std::get<double>(v) << "\n";

    v = "hello";
    std::cout << std::get<std::string>(v) << "\n";
}
```

Детайли

```cpp
using Data = std::variant<A, B, C>;

struct Node {
    Data data;
};
```

- Node е асоцииран с алтернативи, не с конкретен тип
- асоциацията е explicit и затворена
- няма nullptr, няма dynamic lifetime

📌 Design insight
Variant изразява "точно едно от тези поведения / данни".

Compile time dependency

```
Runtime dependency (лошо)
struct Processor {
    virtual void run() = 0;
};

1)включва headers
2)изисква inheritance
3)RTTI / virtual

using Processor = std::variant<Fast, Safe,
Debug>;

void run(const Processor& p) {
    std::visit([](auto& x){ x.run(); }, p);
}

✔ dependency-то е compile-time
✔ всички зависимости са видими в type-а
✔ няма скрити поведения
```

Цялостен пример на наученото досега - student.h

```cpp
#pragma once
#include <string>
#include "Address.h"
#include "Course.h"

class Student {
    std::string name;      // RAII
    int facultyNumber;
    Address address;       // композиция
    Course* course;        // агрегация (НЕ
притежава)
public:
    Student(const std::string& n, int fn,
            const Address& a, Course* c);

    // Rule of Three
    Student(const Student& other); // copy ctor
    // copy assignment operator
    Student& operator=(const Student& other);
    ~Student(); // destructor

    const std::string& getName() const;
    const Course* getCourse() const;
};
```

Цялостен пример досега - student.cpp

```cpp
#include "Student.h"
#include <iostream>
Student::Student(const std::string& n, int fn,
                 const Address& a, Course* c)
    : name(n), facultyNumber(fn), address(a), course(c) {}

Student::Student(const Student& other)
    : name(other.name),
      facultyNumber(other.facultyNumber),
      address(other.address),
      course(other.course) {} // shallow copy
Student& Student::operator=(const Student& other) {
    if (this != &other) {
        name = other.name;
        facultyNumber = other.facultyNumber;
        address = other.address;
        course = other.course;
    }
    return *this;
}
Student::~Student() {
    // НЕ трием course → не го притежаваме
}

const std::string& Student::getName() const {
    return name;
}
const Course* Student::getCourse() const {
    return course;
}
```

Цялостен пример досега на Java

```java
public class Student {
    private final String name;
    private final int facultyNumber;
    private final Address address; // композиция
    private final Course course;   // агрегация (референция)

    public Student(String name, int fn, Address address, Course course)
{       this.name = name;
        this.facultyNumber = fn;
        this.address = new Address(address); // defensive copy
        this.course = course; // не притежаваме
    }

    // copy constructor
    public Student(Student other) {
        this.name = other.name;
        this.facultyNumber = other.facultyNumber;
        this.address = new Address(other.address);
        this.course = other.course;
    }

    public String getName() {
        return name;
    }

    public Course getCourse() {
        return course;
    }
}
```

Въпроси?

Благодаря за вниманието!