Основи на ООП дизайна

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

д-р Тодор Цонков
todort@uni-sofia.bg

Връзки между класовете

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

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

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

Какво е dependency?

Dependency = клас A използва клас B временно

Характеристики:
краткотрайна
без ownership
най-слабата форма на връзка

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
}
```

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

Dependency - продължение

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);
    }
};
```

```
class Engine;

class Car {
    Engine* engine;  // по-слаба зависимост
};
```

Тук използваме forward declaration.
Това намалява зависимостта, защото:

Не ни трябва пълната дефиниция на Engine
Намаляваме include-овете
Подобряваме compile-time

Ownership - добри практики

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.

Ownership - добри практики

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. Чиста опционалност се ползва с std::optional. std::span - само ползва без ownership.

Пример за модерен шаблон в ООП дизайна - Dependency Injection:

```
class Logger { };
class Engine {
    Logger& logger;   // non-owning
public:
    Engine(Logger& l) : logger(l) {}
}; // Engine не трябва да унищожава Logger.
```

Композиция

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)

```
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)

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

```
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

Композиция vs Агрегация

Ако обектът не може да съществува без друг → композиция

Ако само го използва → агрегация

Недостатъци и рискове на агрегация:
1)нужда от проверки за nullptr
2)неясен ownership
3)възможни dangling pointers

```
class Student {
private:
    std::string name;
    Course* course;   // агрегация – не притежаваме обекта

public:
    Student(const std::string& name, Course* course)
        : name(name), course(course) {}

    void print() const {
        if (course) {
            std::print("Student: {}, Course: {}\n", name, course->getName());
        } else {
            std::print("Student: {}, No course assigned\n", name);
        }
    }
};
```

Кое кога да изберем?

Временно използване: Dependency

Дълготрайна връзка без ownership: association

Ясен собственик: unique_ptr
Родител → дете Композиция

Raw pointers са лош дизайн за ownership

```
class Logger {
public:
   void log() {}
};
```

```
class Engine {
   Logger* logger;   // non-owning?
public:
   Engine(Logger* l) : logger(l) {}

   void run() {
      logger->log();
   }
};

int main() {
   Engine* e;

   {
      Logger logger;
      e = new Engine(&logger);
   } // logger умира тук

   e->run();  // 💀 dangling pointer
}
```

Добри практики за дизайн

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.

.

Добри практики за дизайн

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, не в документацията.
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.

Примери за дизайн

Временно използване: Dependency

Дълготрайна връзка без ownership: association

Ясен собственик: unique_ptr
Родител → дете Композиция

Raw pointers са лош дизайн за ownership

```
class Logger {
public:
   void log() {}
};
```

```
class Engine {
   Logger* logger;   // non-owning?
public:
   Engine(Logger* l) : logger(l) {}

   void run() {
      logger->log();
   }
};

int main() {
   Engine* e;

   {
      Logger logger;
      e = new Engine(&logger);
   } // logger умира тук

   e->run();  // 💀 dangling pointer
}
```

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

std::variant

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.

```
int main() {
   std::variant<int, double, std::string> v; // може да бъде int, double или string
   v = 10;          // в момента е int
   v = 3.14;        // сега е double
   v = "Hello"s;    // сега е std::string
}
```

std:variant - използване

```
std::variant<int, std::string> v = "hi";
std::print("v = {}\n", std::get<std::string>(v));
// работи
```

Ако опитате std::get<int>(v) — ще получите std::bad_variant_access.

```
std::variant<int, double, std::string> v = 42;
std::print("index = {}\n", v.index()); // index = 0 (int е първи тип)
```

index() връща позицията на текущия тип в списъка от типове.
Първият тип има индекс 0, вторият 1 и т.н.

```
std::variant<int, double, std::string> v = 3.14;

std::visit([](auto&& arg) {
    std::print("v = {}\n", arg);
}, v);
```

auto&& arg позволява да хващаме всякакъв тип от variant.

Типо-безопасност: за разлика от union, компилаторът знае какво съдържа.

Яснота: работи добре с std::visit вместо сложни switch или if проверки.

Безопасен при move/copy: автоматично извиква правилния конструктор/деструктор на текущия тип.

Visitor Design pattern

```
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

Пример за Visitor pattern:

```
#include <string>
#include <print>

struct Circle { double radius; };
struct Rectangle { double width, height; };

struct ShapeVisitor {
    void operator()(const Circle& c) const {
        std::print("Circle area = {}\n", 3.1415 * c.radius *
c.radius);
    }
    void operator()(const Rectangle& r) const {
        std::print("Rectangle area = {}\n", r.width *
r.height);
    }
};

int main() {
    std::variant<Circle, Rectangle> shape = Circle{2.0};

    std::visit(ShapeVisitor{}, shape);
  return 0;
}
```

Пример с класа Student

```cpp
#pragma once
#include <string>
#include <vector>
#include <iostream>

class Student {
public:
    static constexpr double maxGPA = 4.0;

    // Конструктор с параметри
    Student(const std::string& name, int age)
        : name_(name), age_(age)
    {
        ++studentCount_;
    }

    // Забраняваме copy и move
    Student() = delete;
    ~Student() = default;

    // Getter-и
    const std::string& getName() const { return name_; }
    int getAge() const { return age_; }

    // Setter-и
    void setAge(int age) { age_ = age; }

    static int getStudentCount() { return studentCount_; }

private:
    std::string name_;
    int age_;
    static int studentCount_;
};
```

Пример с класа Student

```cpp
// Клас Student
class Student {
private:
    std::string name;
    StudentID id;               // std::variant пример
    std::unique_ptr<Address> address; // ownership (composition)
    std::vector<Course*> courses;     // aggregation – не притежава курса

public:
    // Constructor
    Student(std::string n, StudentID sid, std::unique_ptr<Address> addr)
        : name(std::move(n)), id(std::move(sid)), address(std::move(addr)) {}

    // Dependency: добавяне на курс (не ownership)
    void enroll(Course* course) {
        courses.push_back(course);
    }

    void print_info() const {
        std::print("Student: {}\n", name);

        // std::variant посетител
        std::visit([](auto&& arg) { std::print("ID: {}\n", arg); }, id);

        if (address) {
            std::print("Address: {}, {}, {}\n",
                       address->street, address->city, address->country);
        }

        std::print("Enrolled courses: {}\n", courses.size());
    }
};
```

Пример с класа Student

```
class Course {
private:
    std::string title;
    Grade grade; // може да се използва за оценка

public:
    Course(const std::string& t, Grade g) : title(t), grade(g) {}
    std::string get_title() const { return title; }
    Grade get_grade() const { return grade; }
};

int main() {
    auto addr = std::make_unique<Address>("Main St 1", "Sofia", "Bulgaria");
    Student s1("Todor", 12345, std::move(addr));

    Course math("Mathematics", Grade::A);
    Course physics("Physics", Grade::B);

    s1.enroll(&math);     // aggregation – студентът не притежава course
    s1.enroll(&physics);

    s1.print_info();
}
```

Обяснение на примера

Dependency – Student зависи от Course, защото използва pointer към него, но не го притежава.

Ownership / Composition – Student притежава Address чрез std::unique_ptr. Когато студентът се изтрие, адресът също се унищожава.

Aggregation – Student държи pointer към Course, но не притежава Course. Курсовете могат да съществуват независимо.

Design guidelines –

Използваме std::unique_ptr за ownership.

std::variant за гъвкав ID тип.

Конструкторите и методите са ясни и минимално зависими.

std::variant – показва как може да се използва за ID, което може да бъде int или string

Въпроси?

Благодаря за вниманието!