Наследование, композиция, агрегация, полиморфизм. SOLID

Подготовительное отделение C/C++ (открытый курс)

образование

Виды отношений между классами. Использование

Использование – отношение между классами, при котором один класс в своей реализации использует в той или иной форме реализацию объектов другого класса.

Примеры:

- Имя одного класса используется в сигнатуре метода другого класса;

```
class X;
class Y {
    void f(X*);
};
```

- В теле метода одного класса создаётся локальный объект другого класса;

```
class X { ... };
class Y {
    void f() { ...; X x; ... }
};
```

- Метод одного класса обращается к методу другого класса.

```
class X {
    static void f();
};

class Y {
    void f() { ...; X::f(); ... }
};
```

Виды отношений между классами. Ассоциация

Ассоциация показывает, что объекты одного класса связаны с объектами другого класса таким образом, что можно перемещаться от объектов одного класса к другому. Является общим случаем композиции и агрегации.

- **Агрегация** — это разновидность ассоциации при отношении между целым и его частями, при котором «части» могут существовать без «целого».

```
class Human;


class University {
    Human* students;
    size_t students_size;
};
```

- **Композиция** имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.

```
class Square { ... };


class Cube {
    Square[6] sides;
};
```

Виды отношений между классами. Наследование

```
class Stack {
    ...
public:
    void push(int elem);
    size_t size() const;
    int top() const;
    void pop();
};

class StackWithMaximum: public Stack {
    void update_max(int elem);
    void pop_max();
    ...
public:
    int get_max_element() const;

    void push(int elem) { Stack::push(elem); update_max(elem); }
    void pop() { Stack::pop(); pop_max(); }
};

...
StackWithMaximum s;
s.push(2);
s.push(1);
std::cout << s.top() << std::endl; // 1
std::cout << s.get_max_element() << std::endl; // 2
```

Наследование – отношение между классами, при котором один класс повторяет структуру и поведение другого класса. Класс, поведение и структура которого наследуется, называется базовым (родительским) классом, а класс, который наследует – производным классом.

- В производном классе структура и поведение базового класса (информационные члены и методы), дополняются и переопределяются.
- В производном классе указываются только дополнительные и переопределяемые члены класса.
- Производный класс является уточнением базового класса.

Одиночное наследование. Правила наследования

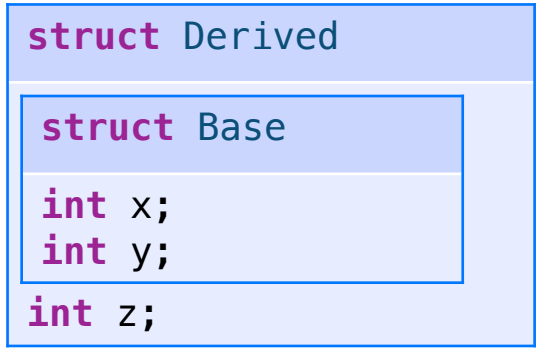
```
class
struct  имя_производного_класса: [ private | protected | public ] имя_базового_класса { ... }
```

Доступ в базовом типе \ Квалификатор доступа	private	protected	public
private	private	private	private
protected	private	protected	protected
public	private	protected	public

Для `class` квалификатор доступа по умолчанию `private`
Для `struct` квалификатор доступа по умолчанию `public`

Одиночное наследование. Правила наследования

```
struct Base {
    int x, y;
};

struct Derived: Base {
    int z;
};

...

Base b;
Derived d;

d.x = 1;
d.y = 2;
d.z = 3;

b = d;
```



Одиночное наследование. Правила наследования

```
class Base {
    int x, y;

public:
    Base(int x, int y): x(x), y(y) {}
};

class Derived: public Base {
public:
    Derived(): Base(0, 0) {}
    using Base::Base;
    using Base::operator=;
};

...

Derived d1;       // ОШИБКА, так как у базового класса нет конструктора по умолчанию
Derived d2(1, 2); // ОШИБКА, так как конструкторы и operator= не наследуются
Base b1(-1, -2);
d1 = b1;
```

Одиночное наследование. Преобразования указателей

Указатель на производный класс может быть неявно преобразован **в указатель на базовый класс**.

Ссылка на производный класс может быть неявно преобразована **в ссылку на базовый класс.**

```
class Base {
    int x;

public:
    Base(int x): x(x) {}

    int getX() const { return x; }
};

class Derived: public Base {
    int y;

public:
    Derived(int x, int y): Base(x), y(y) {}
};
```

```
void f(const Base& b) {
    std::cout << b.getX() << std::endl;
}

...

Derived d(1, 2);
f(d);  // 1

Base* base_ptr = &d;
```

Одиночное наследование. Квалификатор доступа *protected*

```
class Base {
    int x;
protected:
    void setX(int new_x) {
        std::cout << "x changed: " << new_x << std::endl;
        x = new_x;
    }
public:
    int getX() const { return x; }
};

class Derived: private Base {
public:
    void f() {
        x = 3;   // ОШИБКА, доступ к "закрытому" полю базового класса
        setX(3); // ОК, вызов "защищённого" метода базового класса
    }
};
...
Derived d;
d.setX(4); // ОШИБКА, доступ к "закрытому" методу класса
d.f();     // ОК, вызов "открытого" метода класса
d.getX();  // ОШИБКА, доступ к "закрытому" методу класса
```

Одиночное наследование. Перекрытие имён

```
class Base {
public:
    void f(const char*) const { std::cout << __PRETTY_FUNCTION__ << std::endl; }
    void f(int) const { std::cout << __PRETTY_FUNCTION__ << std::endl; }
};


class Derived: public Base {
public:
    void f(int) const { std::cout << __PRETTY_FUNCTION__ << std::endl; }
    using Base::f;
};


...

Derived d;
d.f(1);         // void Derived::f(int) const
d.f("hello");   // void Base::f(const char *) const
d.Base::f("hello");  // void Base::f(const char *) const
d.Base::f(1); // void Base::f(int) const
```

Одиночное наследование. *using* -объявление

```
class Base {
public:
    void f() {}
    void g() {}

protected:
    void h() {}
};


class Derived: private Base {
public:
    using Base::f;
    using Base::h;
};


...

Derived d;
d.f();
d.g(); // ОШИБКА
d.h();
```

Динамический полиморфизм, механизм виртуальных функций

```
class Assistant {
public:
    _vptr
    virtual const char* getName() const { return "?"; }
    virtual int getAge() const { return -1; }

    void greet() const {
        std::cout
            << "Hi! My name is " << getName() << "."  //
            << " I am " << getAge() << " years old."  //
            << std::endl;
    }
};
class Alex: public Assistant {
public:
    const char* getName() const { return "Alex"; }
    int getAge() const { return 30; }
};

...
Assistant a;
a.greet();  // Hi! My name is ?. I am -1 years old.
Alex alex;
alex.greet();  // Hi! My name is Alex. I am 30 years old.
```

```
(gdb) print a
$12 = {
  _vptr.Assistant = 0x402078 <vtable for Assistant+16>
}
(gdb) print *(void**)0x402078@2
$13 = {
  0x40121e <Assistant::getName() const>,
  0x40122e <Assistant::getAge() const>
}
(gdb) print alex
$14 = {
  <Assistant> = {
    _vptr.Assistant = 0x402058 <vtable for Alex+16>
  }, <No data fields>}
(gdb) print *(void**)0x402058@2
$15 = {
  0x4012da <Alex::getName() const>,
  0x4012ea <Alex::getAge() const>
}
```

Динамический полиморфизм, механизм виртуальных функций

Тип данных (класс), содержащий хотя бы одну виртуальную функцию, называется **полиморфным типом** (классом), а объект этого типа – **полиморфным объектом**.

При вызове виртуальной функции через указатель на полиморфный объект осуществляется динамический выбор тела функции в зависимости от типа объекта, а не от типа указателя. Тело функции в таком случае выбирается на этапе выполнения, а не компиляции. В этом и проявляется **динамический полиморфизм**.

Виртуальная функция объявляется описателем `virtual`. Во всех классах-наследниках наследуемая виртуальная функция остается таковой (виртуальной). Таким образом, все типы-наследники полиморфного типа являются полиморфными типами.

Абстрактные классы. Чистые виртуальные функции

```
class Set {
public:
    virtual void add(int elem);= 0;
    virtual void del(int elem);= 0;
    virtual bool contains(int elem) const;= 0;
};

class TreeSet: public Set {
    ...
public:
    void add(int elem) override { ... }
    void del(int elem) override { ... }
    bool contains(int elem) const override { ... }
};

class HashSet: public Set {
    ...
public:
    void add(int elem) override { ... }
    void del(int elem) override { ... }
    bool contains(int elem) const override { ... }
};
```

```
void f(Set* s) {
    s->add(1);
    s->add(2);
    ...
}


TreeSet tree_set;
f(&tree_set);


HashSet hash_set;
f(&hash_set)


Set set; // ОШИБКА! Невозможно создать объект
         // абстрактного класса
```

Виртуальные деструкторы

```
class Set {
public:
    virtual ~Set() = default;
    virtual void add(int elem) = 0;
    ...
};
```

```
class HashSet: public Set {
    struct HashTableRecord {
        ...
    };

    size_t table_size;
    HashTableRecord* table;

public:
    HashSet(size_t init_size = 1024):
        table_size(init_size),
        table(new HashTableRecord[init_size])
    { ... }
    ~HashSet() { delete[] table; }
    void add(int elem) override { ... }
    ...
};
```

```
void f(Set* s) {
    s->add(1);
    s->add(2);
    ...
    delete s;
}

f(new HashSet);
```

Динамический полиморфизм, механизм виртуальных функций

```
class Base {
    int i = f(); // Base
public:
    virtual int f() const { std::cout << "Base" << std::endl; return 0; }
    Base() { f(); } // Base
    ~Base() { f(); } // Base
};

class Derived: public Base {
    int j = f(); // Derived
public:
    virtual int f() const { std::cout << "Derived" << std::endl; return 0; }
    Derived() { f(); } // Derived
    ~Derived() { f(); } // Derived
};

...

{
    Derived d;
}
```

Динамический полиморфизм, механизм виртуальных функций

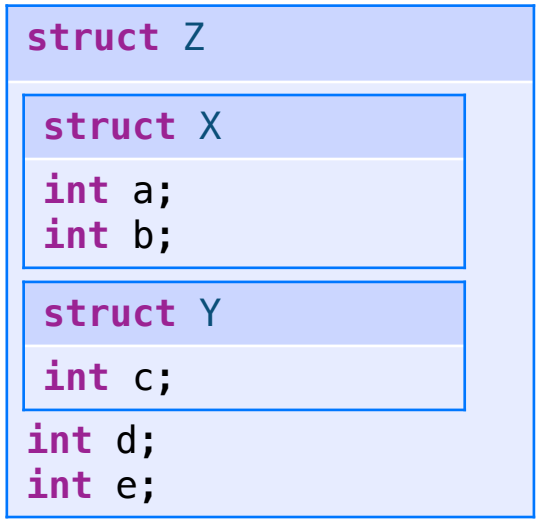
```
class Vehicle {
public:
    int speed, passenger_capacity, max_distance, price;

    virtual void print() const {
        std::cout << "speed: " << speed << std::endl;
        std::cout << "passenger capacity: " << passenger_capacity << std::endl;
        std::cout << "max distance: " << max_distance << std::endl;
        std::cout << "price: " << price << std::endl;
    }
};

class Plane: public Vehicle {
public:
    int engines_count, engine_type;

    void print() const override {
        Vehicle::print();
        std::cout << "engines count: " << engines_count << std::endl;
        std::cout << "engine type: " << engine_type << std::endl;
    }
};
```

Множественное наследование

```
class Encodable {
public:
    virtual ~Encodable() = default;
    virtual void Serialize(std::ostream& out) const = 0;
};


class Decodable {
public:
    virtual ~Decodable() = default;
    virtual void Deserialize(std::istream& in) = 0;
};


void save(const Encodable& enc, const char* filename) { ... }
void load(Decodable& dec, const char* filename) { ... }


class Vector: public Encodable, public Decodable {
    ...
public:
    ...
    void Serialize(std::ostream& out) const override { ... }
    void Deserialize(std::istream& in) override { ... }
};
```

Множественное наследование. Представление объектов в памяти

```
struct X {
    int a, b;
};


struct Y {
    int c;
};


struct Z: X, Y {
    int d, e;
};
```

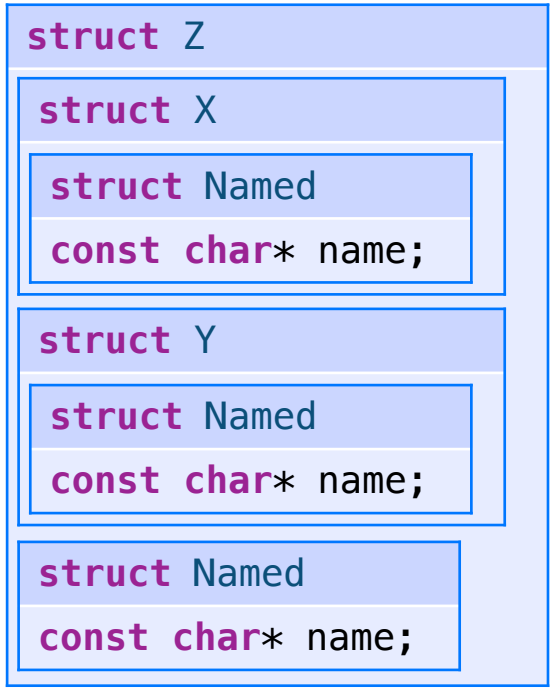


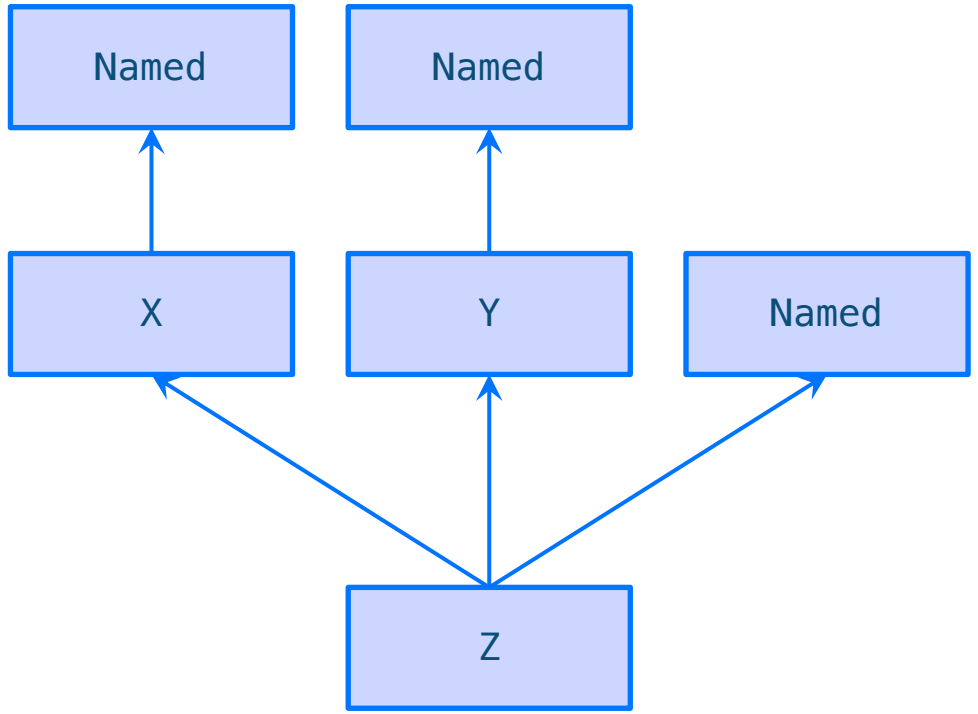
Видимость при множественном наследовании

```
struct X {
    int n;
    void f(int) {}
};

struct Y {
    const char* n;
    void f(const char*) {}
};

struct Z: X, Y {
    using X::n;
    using X::f;
    using Y::f;
};

...

Z z;
z.n = 5;        // ОШИБКА! Имя n содержится в нескольких базовых классах
z.f("hello");   // ОШИБКА! Имя f содержится в нескольких базовых классах
z.X::n = 5;
z.Y::n = "world";
z.X::f(1);
z.Y::f("hello");
```

Множественное наследование. Виртуальные базовые классы

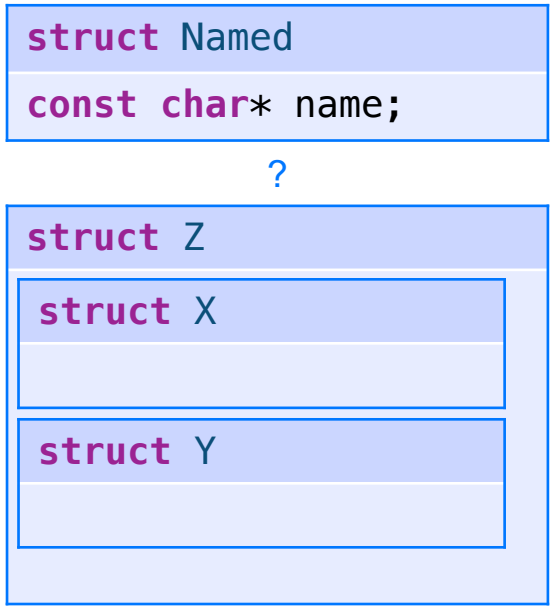
```
class Named {
    const char* name;

public:
    Named(const char* name): name(name) {}
    const char* getName() const { return name; }
};

struct X: Named {
    X(): Named("X") {}
};

struct Y: Named {
    Y(): Named("Y") {}
};

struct Z: X, Y, Named {
    Z(): Named("Z") {}
};

...

Z z;
std::cout << z.getName() << std::endl;  // ОШИБКА! Имя getName найдено в нескольких базовых классах
```

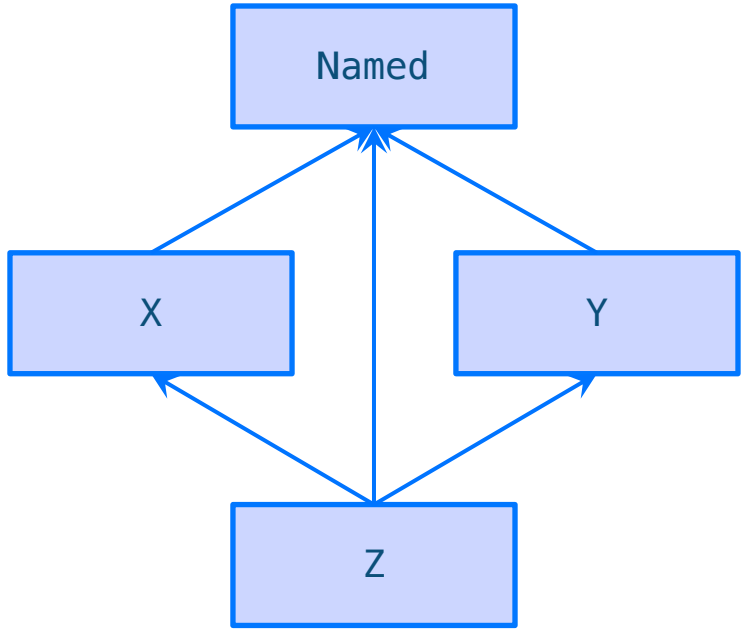




Множественное наследование. Виртуальные базовые классы

```
class Named {
    const char* name;

public:
    Named(const char* name): name(name) {}
    const char* getName() const { return name; }
};

struct X: virtual Named {
    X(): Named("X") {}
};

struct Y: virtual Named {
    Y(): Named("Y") {}
};

struct Z: X, Y, virtual Named {
    Z(): Named("Z") {}
};

...

Z z;
std::cout << z.getName() << std::endl;  // Z
```





Динамическая информация о типе (RTTI)

```cpp
#include <iostream>
#include <typeinfo>


struct Base {
    virtual const char* who_is_that() const { return "Base"; }
    virtual ~Base() = default;
};

struct Derived: public Base {
    const char* who_is_that() const override { return "Derived"; }
};

void f(const Base& base) {
    std::cout << base.who_is_that() << std::endl;

    const std::type_info& info = typeid(base);
    std::cout << "name: " << info.name()
              << ", hash code: " << info.hash_code() << std::endl;
    return;
}
```

```cpp
Base b;
f(b);
// Base
// name: 4Base, hash code: 4294996542


Derived d;
f(d);
// Derived
// name: 7Derived, hash code: 4294996580
```

Runtime type identification работает только с полиморфными типами данных!

Динамическая информация о типе. *dynamic_cast*

```
struct Base {
    virtual const char* who_is_that() const { return "Base"; }
    virtual ~Base() = default;
};

struct Derived: public Base {
    const char* who_is_that() const override { return "Derived"; }
    void f() const { std::cout << "Derived::f" << std::endl; }
};

void f(const Base& base) {
    const Derived& derived = dynamic_cast<const Derived&>(base);
    derived.f(); // Derived::f
    return;
}

...

Derived d;
f(d); // OK

Base b;
f(b); // ОШИБКА во время исполнения программы, аварийное завершение
```

SOLID

Сокращение от:

- **S**ingle responsibility,
- **O**pen-closed,
- **L**iskov substitution,
- **I**nterface segregation и
- **D**ependency inversion

— мнемонический акроним, введённый Майклом Фэзерсом для первых пяти принципов, названных Робертом Мартином в начале 2000-х, которые означали 5 основных принципов объектно-ориентированного программирования и проектирования.

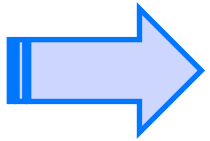
Single responsibility principle

A class should have only one reason to change.

Robert C. Martin

Принцип единственной ответственности — принцип ООП, обозначающий, что каждый объект должен иметь одну ответственность и эта ответственность должна быть полностью инкапсулирована в класс. Все его поведения должны быть направлены исключительно на обеспечение этой ответственности.

```
class Document {
public:
    const char* getTitle() const;
    const char* getHeader() const;
    const char* getBody() const;
    const char* getFooter() const;
    const char* toJSON() const;
};
```



```
class DocumentStructure {
public:
    const char* getTitle() const;
    const char* getHeader() const;
    const char* getBody() const;
    const char* getFooter() const;
};

class JSONDocumentEncoder {
    JSONDocumentEncoder(const DocumentStructure*);
    const char* toJSON() const;
};
```

Open–closed principle

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification

Bertrand Meyer

Принцип открытости/закрытости Мейера: программные сущности (классы, модули, функции) должны быть открыты для расширения, но не для модификации.

Полиморфный принцип открытости/закрытости: программные сущности должны быть:

- открыты для расширения, то есть поведение сущности может быть расширено путём создания новых типов сущностей;
- закрыты для изменения: в результате расширения поведения сущности, не должны вноситься изменения в код, который эту сущность использует.

Liskov substitution principle

Принцип подстановки Барбары Лисков:

Пусть *q(x)* является свойством, верным относительно объектов *x* некоторого типа *T*. Тогда *q(y)* также должно быть верным для объектов *y* типа *S*, где *S* является подтипом типа *T*.

Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

Роберт С. Мартин

Поведение классов-наследников не должно противоречить поведению, заданному базовым классом, то есть должно быть ожидаемым для кода, использующего переменную базового типа.

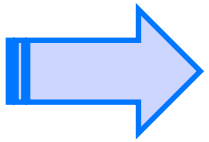
«Подкласс не должен требовать от вызывающего кода больше, чем базовый класс, и не должен предоставлять вызывающему коду меньше, чем базовый класс».

Interface segregation principle

Программные сущности *не должны зависеть от методов, которые они не используют.*

Роберт С. Мартин

```
class Persistable {
public:
    virtual ~Persistable() = default;
    virtual void Serialize(std::ostream& out) const = 0;
    virtual void Deserialize(std::istream& in) = 0;
};


void save(const Persistable& enc, const char* filename);
void load(Persistable& dec, const char* filename);
```



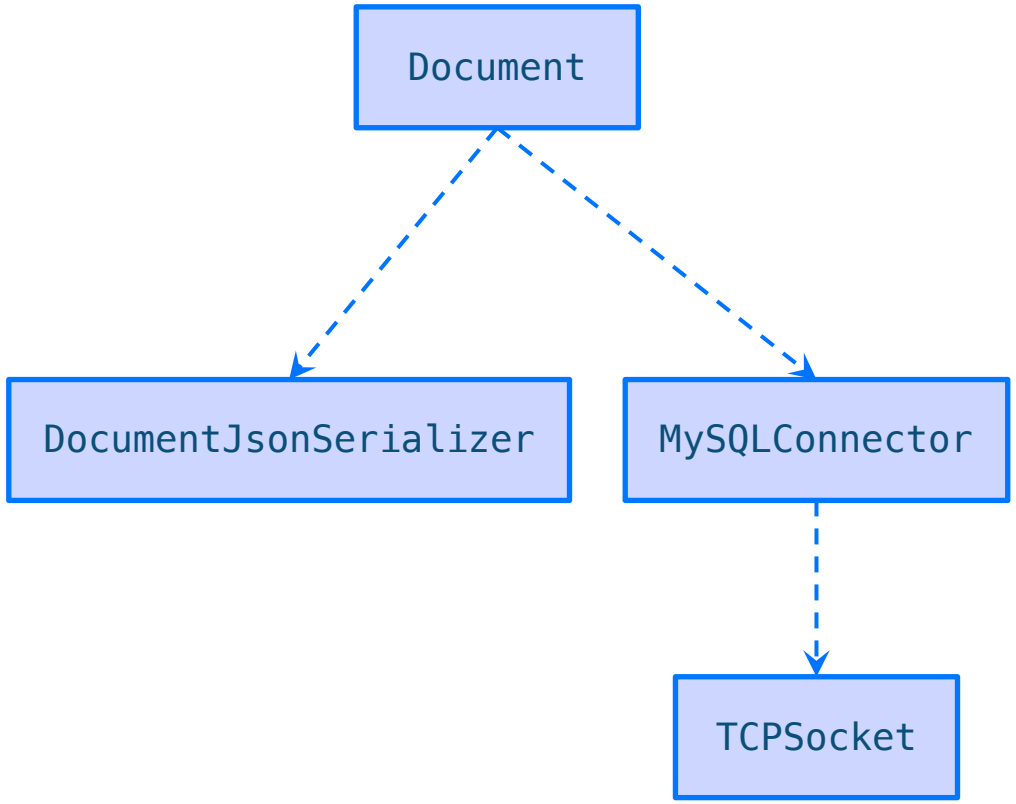
```
class Encodable {
public:
    virtual ~Encodable() = default;
    virtual void Serialize(std::ostream& out) const = 0;
};


class Decodable {
public:
    virtual ~Decodable() = default;
    virtual void Deserialize(std::istream& in) = 0;
};


void save(const Encodable& enc, const char* filename);
void load(Decodable& dec, const char* filename);
```

Dependency inversion principle

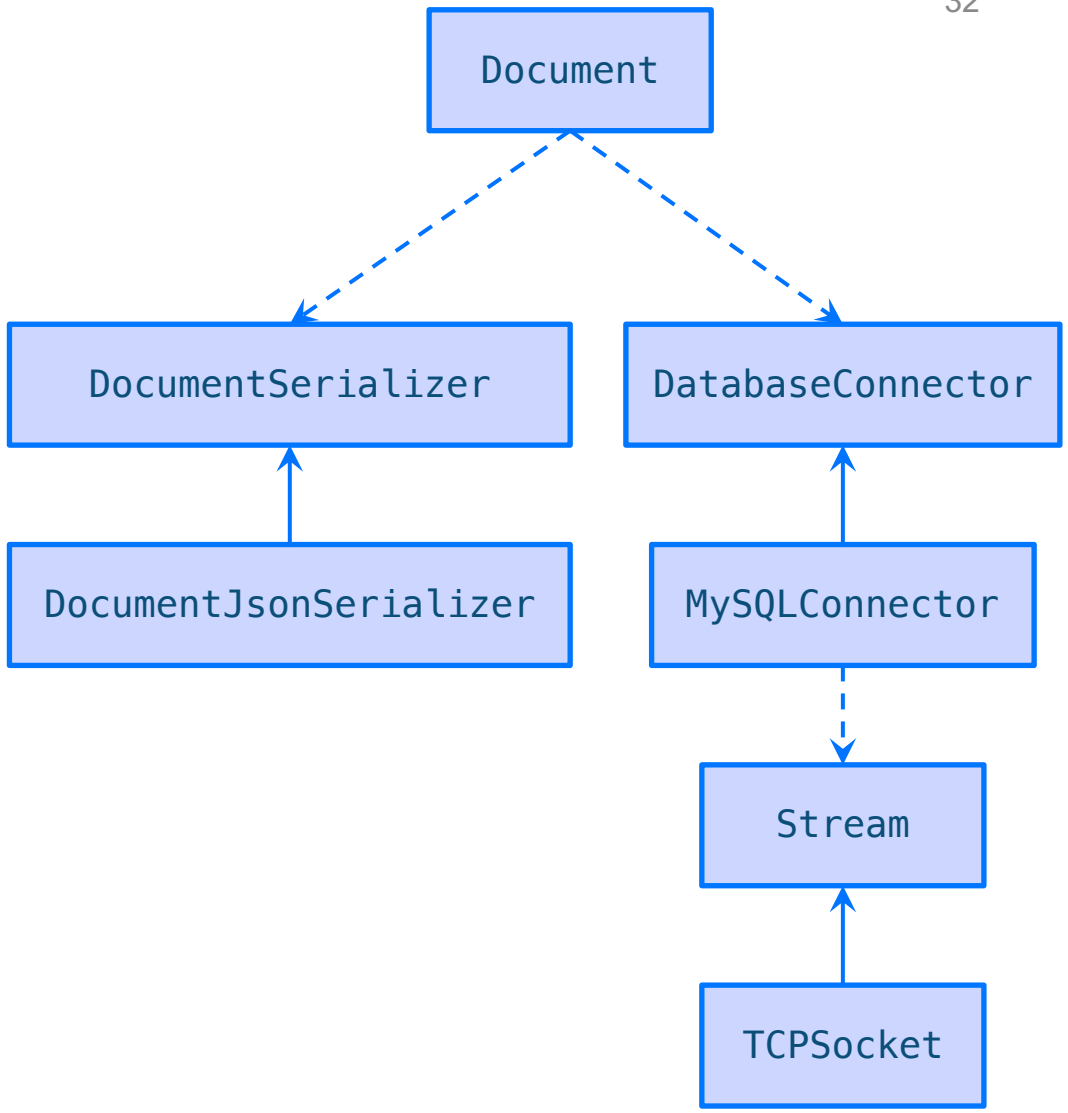
Принцип инверсии зависимостей:

1. Модули верхних уровней не должны импортировать сущности из модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
2. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

```
class Document {
public:
    int getID() const;
    void save() const;
};

class DocumentJsonSerializer {
public:
    const char* toJSON(const Document&) const;
};

class MySQLConnector {
public:
    void insert(int id, const char* data);
};

class TCPSocket {
public:
    void write(const char* data);
};


void Document::save() {
    MySQLConnector database;
    DocumentJsonSerializer json_serializer;
    database.insert(getID(), json_serializer.toJSON(*this));
}

void MySQLConnector::insert(int id, const char* data) {
    TCPSocket sock;
    ...
    sock.write(buf);
}
```



```
class Document;

class DocumentSerializer {
    virtual const char* serialize(const Document&) const = 0;
};

class DatabaseConnector {
public:
    virtual void insert(int id, const char* data) = 0;
};

class Document {
public:
    int getID() const;
    void save(DocumentSerializer& s, DatabaseConnector& db) const {
        db.insert(getID(), s.serialize(*this));
    }
};

class Stream {
public:
    virtual void write(const char* data) = 0;
};

class MySQLConnector: public DatabaseConnector {
    Stream& stream;
public:
    MySQLConnector(Stream& s): stream(s) {}
    void insert(int id, const char* data) override {
        stream.write(buf);
    }
};

class DocumentJsonSerializer: public DocumentSerializer {
public:
    const char* serialize(const Document&) const override;
};
```




```
class TCPSocket: public Stream {
public:
    void write(const char* data) override;
};
```

Спасибо за внимание!

E-mail: i.anferov@corp.mail.ru
Telegram: igor_anferov
GitHub: igor-anferov

Игорь Анфёров

образование