2. Проектирование ПО

2.2. Объектно-ориентированный подход

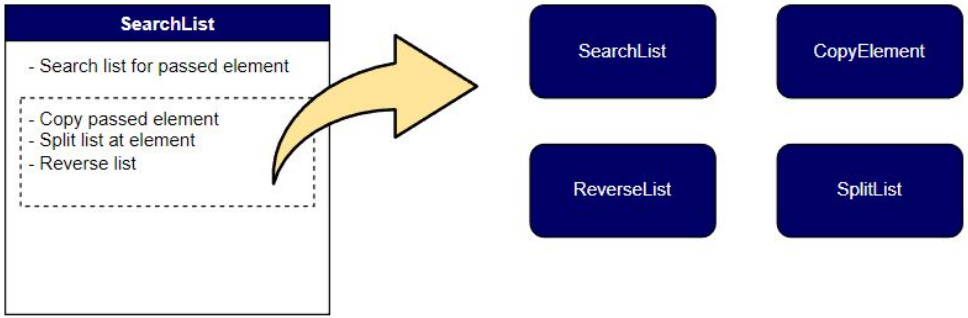
SOLID

Принципы “красивого” ООП-дизайна, и не более того…

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

Single Responsibility

- **Каждый объект должен иметь одну обязанность [причину для изменения]**
- **Эта обязанность должна быть полностью инкапсулирована в объект**



SRP: Splitting a non-SRP class

“A class should only have a single responsibility, that is, only changes to one part of the software’s specification should be able to affect the specification of the class.” -Robert C. Martin

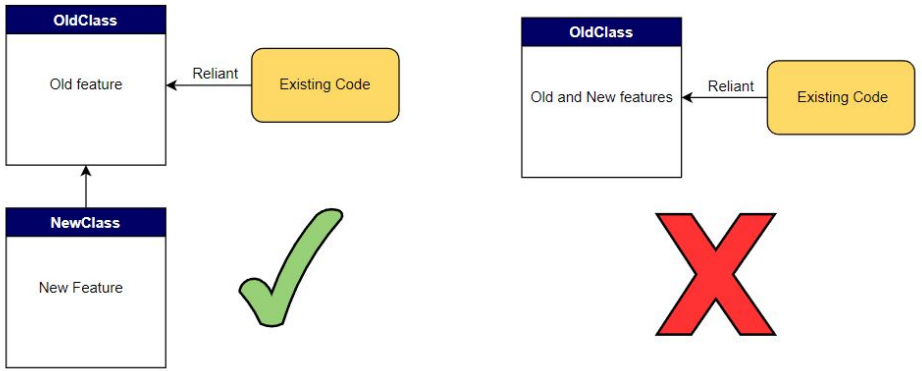
Изображения: https://www.educative.io/blog/solid-principles-oop-c-sharp

Open/Closed

- **Программные сущности (классы, модули, функции, и др) должны быть открыты для расширения, но закрыты для изменения**
  - **переиспользование через наследование --> полиморфизм**
  - **неизменные интерфейсы**

То есть мы определяем однозначные точки для расширения, и их меняем.

- **В интерфейсной части систем используется меньше конкретных классов -- только виртуальные (интерфейсы) и сырые данные.**



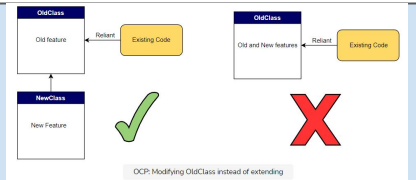
OCP: Modifying OldClass instead of extending

“Software entities … should be open for extension, but closed for modification.” Robert C. Martin

Изображения: https://www.educative.io/blog/solid-principles-oop-c-sharp

Open/Closed: Расширение функциональности

- **Новые методы обработки старых данных**
- **Новые данные**
- **Переиспользование через наследование --> полиморфизм**
  - **Неизменные интерфейсы**
- **Композиция**
  - **Только внешние интерфейсы для доступа снаружи -- инкапсуляция**



“Software entities … should be open for extension, but closed for modification.” Robert C. Martin

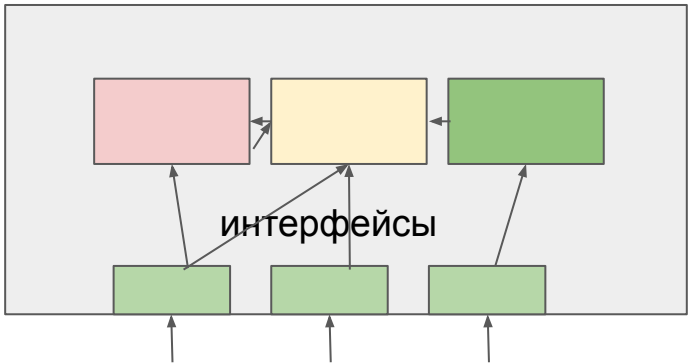
То есть мы определяем однозначные точки для расширения, и их меняем.

- **В интерфейсной части систем используется меньше конкретных классов -- только виртуальные (интерфейсы) и сырые данные.**
- **Перекрытие базовых методов**
- **Множественное наследование**

Наследование

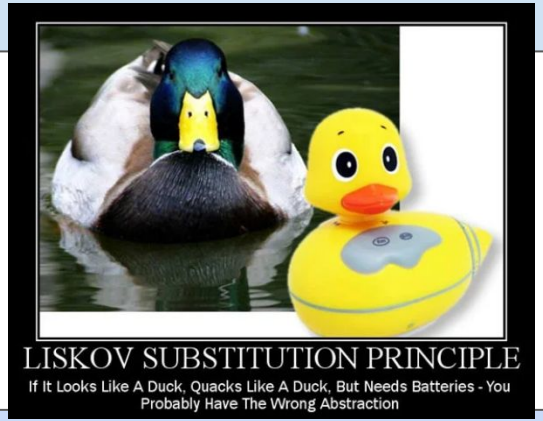
+ Новые данные	
	Данные
+ Новый код	Код

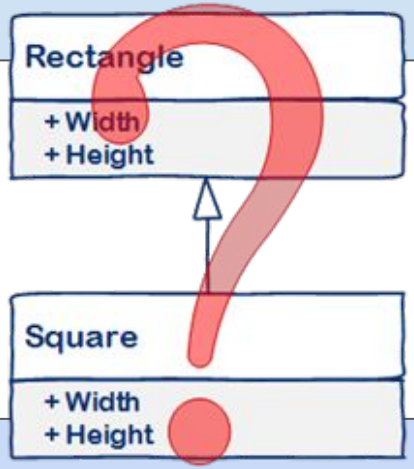
Агрегирование



Liskov Substitution

- **Это скорее определение наследования -- функции, которые используют базовый тип, должны использовать подтипы базового типа, не зная об этом**
- **Если это потомок, и предок где-то используется, то и потомок может использоваться там же. И инварианты предка должны выполняться в потомке.**





В терминологии контрактного программирования:

1. Производные классы не должны усиливать предусловия (не должны требовать большего от своих клиентов).
2. Производные классы не должны ослаблять постусловия (должны гарантировать, как минимум тоже, что и базовый класс).
3. **Инварианты базового класса и наследников суммируются**
4. Производные классы не должны генерировать исключения, не описанные базовым классом.

Interface Segregation

- **Клиенты не должны зависеть от методов, которые они не используют**
  - **слишком “толстые” интерфейсы необходимо разделять на более “мелкие” и специфические**
- **Удобство: не перекомпилировать клиента 2, если что-то изменилось для клиента 1**

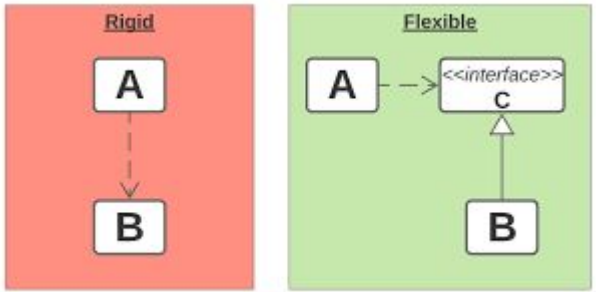
```
struct Signal;

struct IMachine {
    virtual void gsm(Signal &bytes) = 0;
    virtual void wifi(Signal &bytes) = 0;
    virtual void streaming(Signal &bytes) = 0;
};

struct MultimediaModem : IMachine {
    void gsm(Signal &bytes) override { }
    void wifi(Signal &bytes) override { }
    void streaming(Signal &bytes) override { }
};

struct WIFIRouter : IMachine {                    // Not OK
    void gsm(Signal &bytes) override { }
    void wifi(Signal &bytes) override { // do connect }
    void streaming(Signal &bytes) override { }
};
```

```
struct IMultimediaModem {
    virtual void gsm(Signal &bytes) = 0;
    virtual void streaming(Signal &bytes) = 0;
};

struct IWIFIRouter {
    virtual void wifi(Signal &bytes) = 0;
};

struct MultimediaModem : IMultimediaModem {
    void gsm(Signal &bytes) override { }
    void streaming(Signal &bytes) override { }
};

struct WIFIRouter : IWIFIRouter {
    void wifi(Signal &bytes) override {  }
};

struct IMachine : IWIFIRouter, IMultimediaModem { };

struct Machine : IMachine {
    IWIFIRouter&   m_router;
    IMultimediaModem&   m_modem;
    Machine(IWIFIRouter &p, IMultimediaModem &s) : m_router{p}, m_modem{s} { }
    void gsm(Signal &bytes) override {m_modem.gsm(bytes); }
    void streaming(Signal &bytes) override {m_modem.streaming(bytes); }
    void wifi(Signal &bytes) override { m_router.wifi(bytes); }
};
```

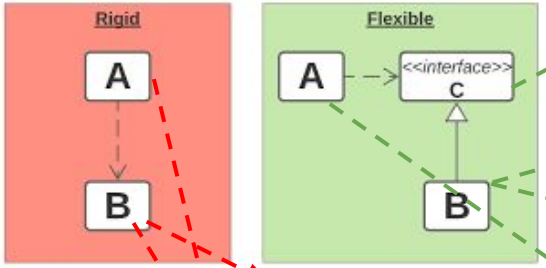
Dependency Inversion

- **Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.**
- **Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракции.**

- **Модули (или классы) верхнего уровня** = классы, которые выполняют операцию при помощи инструмента
- **Модули (или классы) нижнего уровня** = инструменты, которые нужны для выполнения операций
- **Абстракции** – представляют интерфейс, соединяющий два класса
- **Детали** = специфические характеристики работы инструмента



Dependency Inversion

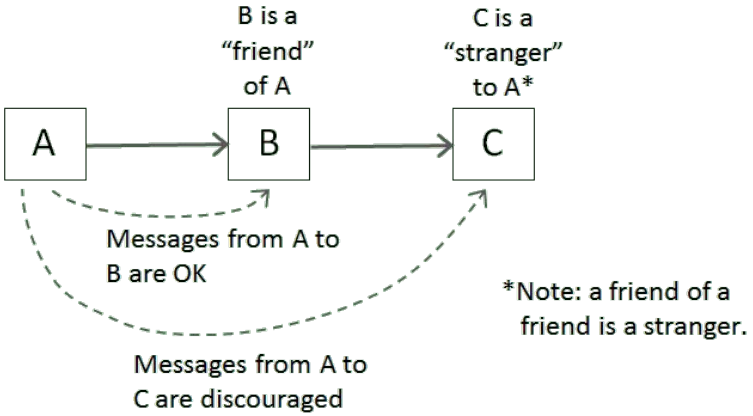



```
class FrontEndDeveloper {
 public:
  void developFrontEnd();
};

class BackEndDeveloper {
 public:
  void developBackEnd();
};

class Project {
 public:
  void deliver() {
    front_resp.developFrontEnd();
    back_resp.developBackEnd();
  }
 private:
  FrontEndDeveloper front_resp;
  BackEndDeveloper back_resp;
};
```

```
class Developer {
 public:
  virtual ~Developer() = default;
  virtual void develop() = 0;
};

class FrontEndDeveloper : public Developer {
 public:
  void develop() override { developFrontEnd(); }
 private:
  void developFrontEnd();
};

class BackEndDeveloper : public Developer {
 public:
  void develop() override { developBackEnd(); }
 private:
  void developBackEnd();
};

class Project {
 public:
  using Developers = std::vector<std::unique_ptr<Developer>>;
  explicit Project(Developers developers)
      : developers_{std::move(developers)} {}

  void deliver() {
    for (auto &developer : developers_) {
      developer->develop();
    }
  }

 private:
  Developers developers_;
};
```

+ Закон деметры

- Закон Деметры (Law of Demeter, LoD) — это принцип проектирования программного обеспечения, который рекомендует минимизировать связи между объектами, ограничивая доступ к внутренним компонентам объектов. Основная идея заключается в том, что объекты должны взаимодействовать друг с другом только через свои непосредственные зависимости, избегая глубоких цепочек доступа.




```cpp
class A {
public:
    void doSomething() {
        // Directly accessing a method
        b.getC().doSomethingElse();
    }

private:
    B b;
};

class B {
public:
    C& getC() {
        return c;
    }

private:
    C c;
};

class C {
public:
    void doSomethingElse() {
        // Some implementation
    }
};
```

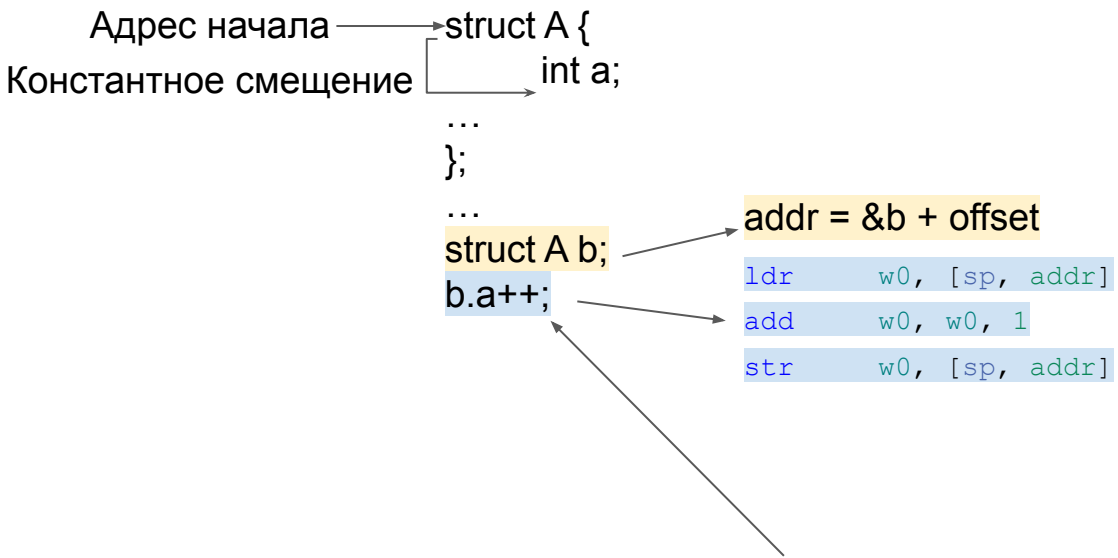
```cpp
class A {
public:
    void doSomething() {
        // A only interacts with B
        b.doSomething();
    }

private:
    B b;
};

class B {
public:
    void doSomething() {
        // B interacts with C
        c.doSomethingElse();
    }

private:
    C c;
};

class C {
public:
    void doSomethingElse() {
        // Some implementation
    }
};
```

2. Проектирование ПО

2.3. Некоторые технические особенности ООП на C++

C/C++

В чём принципиальное отличие языков класса C и C++ от других ЯП?



Вычисления расстояния до этого поля в статике позволяет языку быть условно быстрым и “близким к ассемблеру”.

C/C++

Фундаментальный шаг C++ по сравнению с C -- ввести код в объекты

В простейшем случае для объекта класса A его невиртуальный метод A::f -- такая же C-функция (в том числе и по скорости вызова), но:

- Имена манглятся (см. Лекцию 2)
- Если метод не статический, первым параметром (неявно) принимается указатель **this** нужного типа.

Любое обращение к полям и методам класса явно или неявно использует **this**.

C - **struct**
typedef struct {int m;int n;} Data;
Data d = {4,2}; (**до C99**)
C99: Data d = {.n=2,.m=4};

C++ **class (struct это тоже класс)**

Для создания / уничтожения объектов используются специальные методы класса

- Наследование == включение
- Инкапсуляция
- Полиморфизм

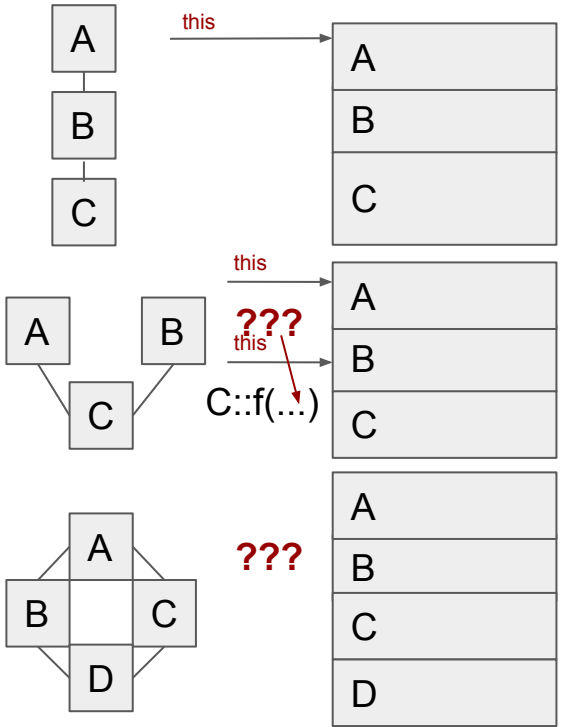
…

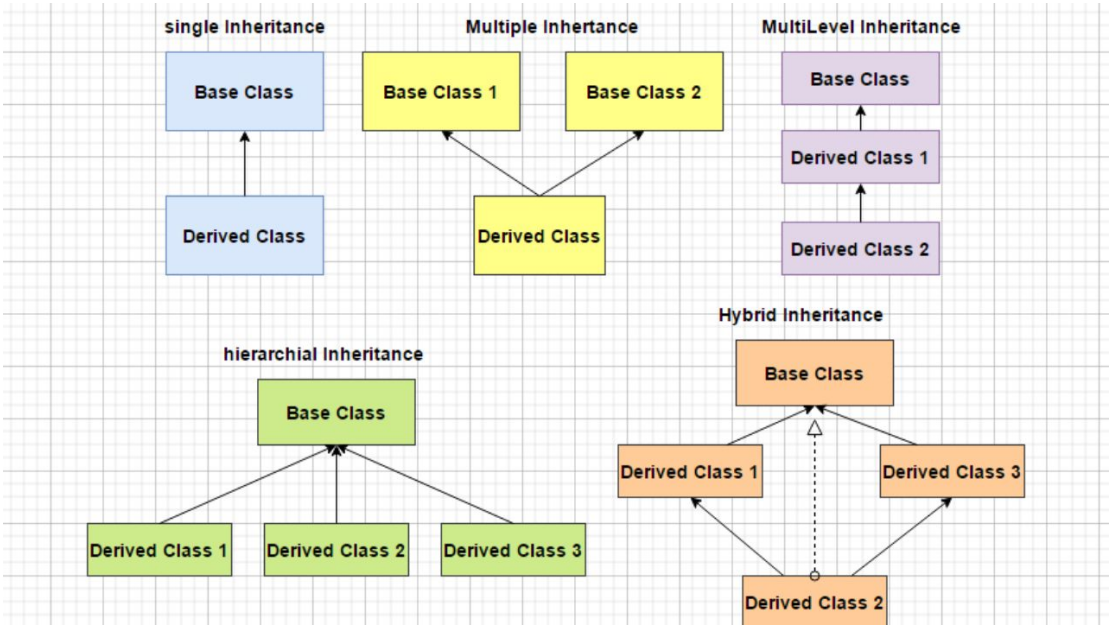
Статический метод класса

Что делать, если конструктор не сможет создать объект (например, по независящим от него причинам)?

Множественное наследование

- **Различные механизмы приведения типов**
- **RTTI**





Шаблоны

- **Препроцессор с проверкой типа**
- **Код объекта создаётся только при инстанцировании**
- **На каждый тип -- по экземпляру -- код сильно раздувается**
- **Проблемы ‘by design’**

```
#include<iostream>

template<typename T>
T add(T num1, T num2) {
  return (num1 + num2);
}

int main() {
  ... ... ...

  result1 = add<int>(2,3);
  ... ... ...

  result2 = add<double>(2.2,3.3);
  ... ... ...

}
```

```
int add(int num1, int num2) {
    return (num1 + num2);
}
```

```
double add(double num1, double num2) {
    return (num1 + num2);
}
```

```
template <typename T>
class Heap {
  public:
    void push( const T &val );
    T pop();
    bool empty() const { return h_.empty(); }
  private:
    std::vector<T> h_;
};
```

Специализация шаблонов

- **Позволяет специфицировать логику под конкретные типы**
- **Косвенно, приводит к меньшему разрастанию кода при инстанцировании**
- **Полная специализация**
- **Частичная специализация -- только для классов**

Исключения

- **Throw/Catch, раскрутка стека**
- **Обработка нештатных событий с нелокальной передачей управления**

Исключения: 2 вида

- “Внешние” -- исключения уровня ОС или аппаратуры (например, Page Fault)
- **“Внутренние” -- исключения на уровне языка -- о них и поговорим.**

Исключения

Обработка ошибок в C-стиле:

- enum error_t {E_OK, E_ERR1, E_ERR2};
  error_t f(...); // return ret_code as error / no error
- Вернуть код ошибки по указателю, переданному в списке параметров
  error_t f(..., error_t *errcode); // error_t*
- errno
  - thread-local

Исключения

Обработка ошибок в C-стиле:

- enum error_t {E_OK, E_ERR1, E_ERR2};
  error_t f(...); // return ret_code as error / no error
- Вернуть код ошибки по указателю, переданному в списке параметров
  error_t f(..., error_t *errcode); // error_t*
- errno
  - thread-local

Недостатки?

Исключения

Обработка ошибок в C-стиле:

- enum error_t {E_OK, E_ERR1, E_ERR2};
  error_t f(...); // return ret_code as error / no error

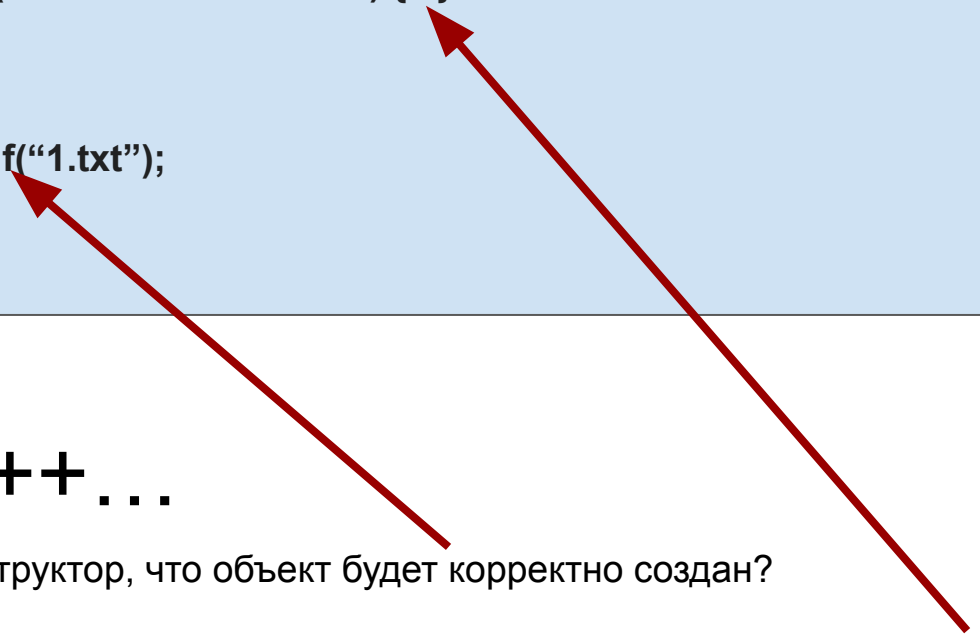
int atoi(const char*);

- В случае, если преобразование невозможно, вернёт 0 …
- В случае, если число слишком большое, вернёт HUGE_VAL и соответствующее значение в errno.

Они есть уже в C…

Исключения

```
Class FileObj {
        FileObj(const char *filename) {...}
};

int main() {
        FileObj f(“1.txt”);
        …
}
```



В C++…

А гарантирует ли конструктор, что объект будет корректно создан?

Как понять, что произошла ошибка, если конструктор ничего не возвращает?

Такие ошибки оставляют объект в неконстистентном состоянии -- нарушают инварианты класса.

Исключения

- Выход из некоторой вызванной функции в вызывающий код (в обход штатных средств возврата управления)
- При выходе должен быть сгенерирован объект (или данные), содержащий информацию о том, что привело к исключению
- Возврат должен быть “пойман” ниже по стеку вызовов и обработан
  - Таким образом, реализуется **нелокальный** выход из функции
  - По-хорошему, исключение **должно быть обработано** и **не должно** приводить к abort’у
    - Но и не обязано (может быть сгенерирован std::terminate)

Исключения: техническая реализация

- **throw** компилятор транслирует в пару вызовов функций **libstdc++**, которые размещают исключение и начинают раскручивание стека посредством **_Unwind*** из **`libgcc_s.so`** / **`libunwind`**.
- Для **каждого catch блока** компилятор добавляет информацию после тела метода, таблицу исключений, которые метод может отлавливать, таблицу очистки
- В процессе раскручивания стека вызывается из **libstdc++** “**personality routine**”,
- которая проверяет каждую функцию в стеке на исключения, которые та может отлавливать.
- Если не нашлось никого, кто мог бы отловить эту ошибку, вызывается **std::terminate**.
- Если кто-то все же нашелся, **раскрутка запускается снова с вершины стека**.
  - При этой повторной раскрутке запускается "**personality routine**" по очистке ресурсов для каждого метода.
  - Подпрограмма проверяет таблицу очистки для текущего метода.
  - Если в методе есть, что очистить, подпрограмма "прыгает" в текущий фрейм и запускает код очистки, который вызывает деструкторы для каждого из объектов, в текущей области видимости.
- Когда раскрутка натыкается на фрагмент стека, который может обрабатывать исключение, она "прыгает" в блок обработки исключения.
- После окончания обработки исключения, очищаем память исключения.

Исключения: throw

- **throw-объявление** будет транслировано компилятором в два вызова: **__cxa_allocate_exception** и **__cxa_throw**.
- -- __cxa_allocate_exception и __cxa_throw "из libstdc++.
- -- __cxa_allocate_exception выделяет память для нового исключения.
- -- __cxa_throw выполняет подготовку и отдает исключение в **_Unwind**, в набор функций, которые живут в **`libgcc_s.so`** / **`libunwind`** и производит реальную размотку стека (ABI определяет интерфейс этих функций).

- `__cxa_allocate_exception(size_t)' -- выделяет достаточное количество памяти для хранения исключения во время его пробрасывания.
- `__cxa_throw(void* thrown_exception, struct type_info *tinfo, void (*dest)(void*))' -- функция ответственна за вызов раскрутки стека.

Важный эффект: __cxa_throw никогда не предполагает возврат (return). Она так же передает управление подходящему catch-блок для обработки исключения либо вызывает (по-умолчанию) std::terminate, но никогда ничего не возвращает.

Исключения: catch

- Отлов исключений требует от программы рефлексии (исследования своего собственного кода):
  `__cxa_begin_catch'
  `__cxa_end_catch'
  `__gxx_personality_v0'

```
_Z18func_with_catch_blockv:
        .cfi_startproc
        .cfi_personality 0,__gxx_personality_v0
        .cfi_lsda 0,.LLSDA1
```

-- линкер использует инфу из этого пролога для спецификации **CFI** (call frame information);
-- **CFI** хранит информацию о фрейме вызова. Используется, в основном, для раскручивания стэка.
-- **LDSA** (language specific data area) — специальная область, используемая персональной функцией, чтобы узнать, какие исключения могут быть обработаны данной функцией.

Исключения: catch

- Отлов исключений требует от программы рефлексии (исследования своего собственного кода):
  _Unwind_Reason_Code **__gxx_personality_v0** (
  int version, _Unwind_Action actions, uint64_t exceptionClass,
  _Unwind_Exception* unwind_exception, _Unwind_Context* context);

-- **__gxx_personality_v0 -- “персональная подпрограмма” по очистке ресурсов для каждого метода.**
-- Подпрограмма проверяет таблицу очистки для текущего метода.
-- Если в методе есть, что очистить, подпрограмма "прыгает" в текущий фрейм и запускает код очистки, который вызывает деструкторы для каждого из объектов, размещенных в текущей области видимости.

Исключения: фазы 1,2

```
main()
  |
  +-- riskyFunction()
        |
        +-- __cxa_allocate_exception()  // Allocate memory for the exception
        |
        +-- __cxa_throw()               // Throw the exception
              |
              +-- _Unwind_RaiseException()  // Start unwinding (search phase)
                    |
                    +-- Personality Routine  // Called for each stack frame
                          |
                          +-- _Unwind_GetContext()  // Get context of the stack frame
                          |
                          +-- Decides if this frame can handle the exception
                          |
                          +-- If no match, continue searching up the stack
              |
              +-- If a matching catch block is found, search phase ends
```

“Search”

```
main()
  |
  +-- _Unwind_RaiseException()  // Resume unwinding (cleanup phase)
        |
        +-- Personality Routine  // Called again for each stack frame
              |
              +-- _Unwind_GetContext()  // Get context of the stack frame
              |
              +-- _Unwind_SetIP()       // Set instruction pointer for cleanup
              |
              +-- Destructors for local objects are called
              |
              +-- _Unwind_Resume()      // Continue unwinding
  |
  +-- __cxa_begin_catch()  // Enter the catch block
  |
  +-- Execute catch block logic
  |
  +-- __cxa_end_catch()    // Exit the catch block
```

“Cleanup”

Разделение на 2 фазы нужно для:

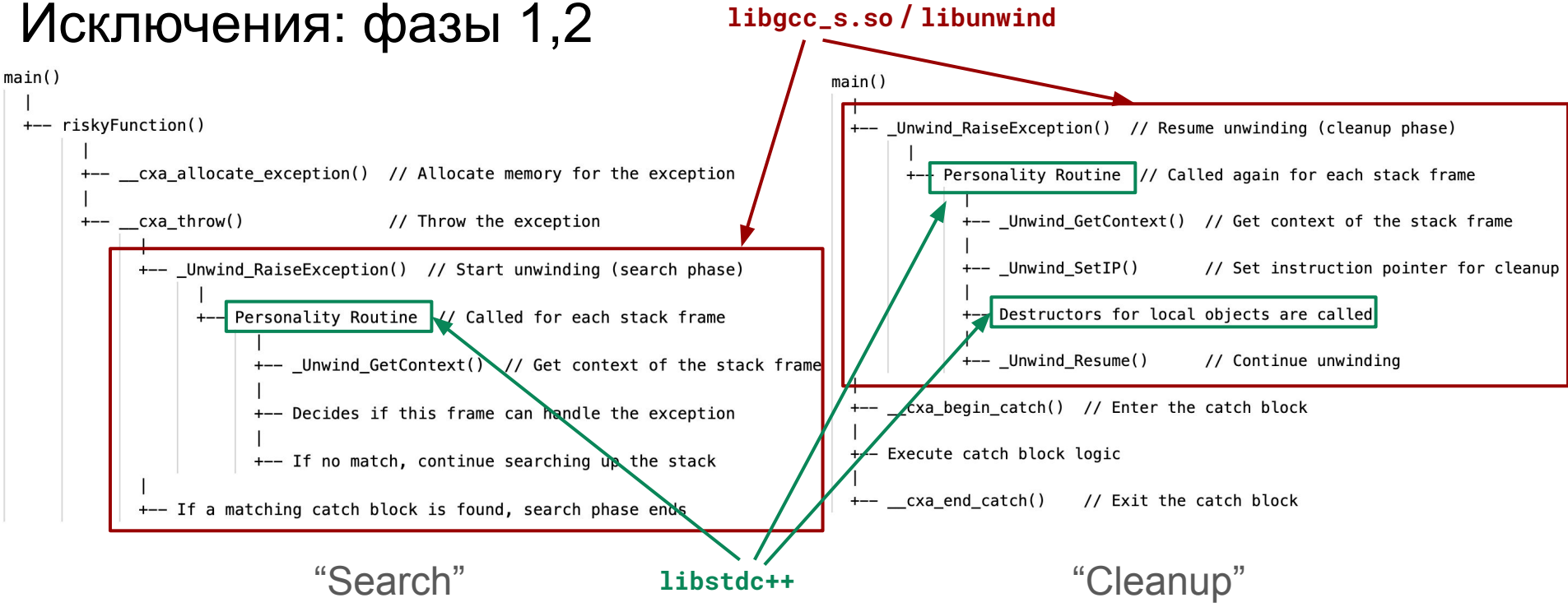
-- Разделения логики сбора стек-трейса до точки обработки и очистки стека

-- больше однозначности, меньше UB и т.д.

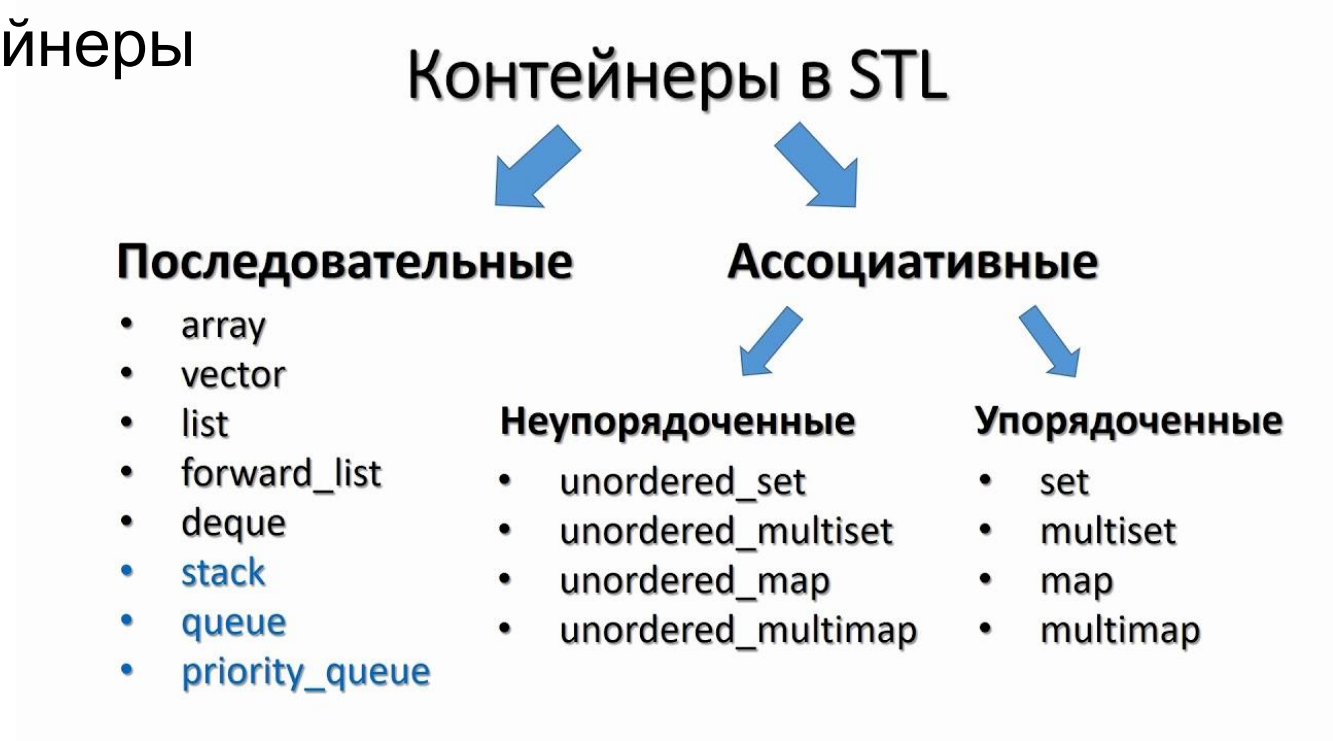
-- очистки ресурсов “по факту”

-- Обработки вложенных исключений

Исключения: фазы 1,2



Контейнеры



- последовательные контейнеры — вектор (vector), двусвязный список (list), дэка (deque);
- ассоциативные контейнеры — множества (set и multiset), хэш-таблицы (map и multimap);
- псевдо контейнеры — битовые маски (bitset), строки (string и wstring), массивы (valarray)...

Контейнеры

- Стандартная библиотека предоставляет различные контейнеры для хранения коллекций связанных объектов.
- Контейнеры — это шаблоны классов.
- При объявлении переменной контейнера указывается тип элементов, которые будет хранить контейнер.
- Контейнеры могут создаваться с использованием списков инициализаторов.
- Контейнеры имеют функции-члены для добавления и удаления элементов и выполнения других операций.

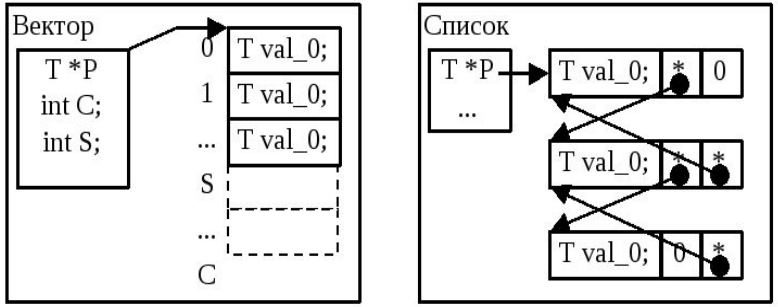
- последовательные контейнеры — вектор (vector), двусвязный список (list), дэка (deque);
- ассоциативные контейнеры — множества (set и multiset), хэш-таблицы (map и multimap);
- псевдо контейнеры — битовые маски (bitset), строки (string и wstring), массивы (valarray)...

\+ Итераторы к каждому классу контейнеров

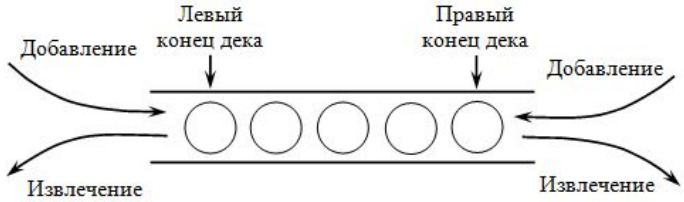
Последовательные контейнеры

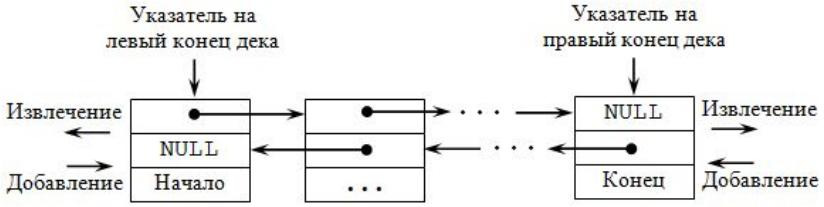
- **Вектор --** *контейнер*, который позволяет осуществлять доступ к элементам по индексам и вставку в конец за O(1).
- **Список -** *контейнер*, который позволяет вставлять и удалять элемент за O(1), и не позволяет осуществлять доступ по индексу.

Схема организации контейнеров: вектор и список.



- **Дека** (англ. *deque – double-ended queue*, двухсторонняя очередь) – контейнер, представляющий собой последовательность элементов, в который можно добавлять и удалять элементы с двух сторон за O(1). Первый и последний элементы дека соответствуют входу и выходу дека.

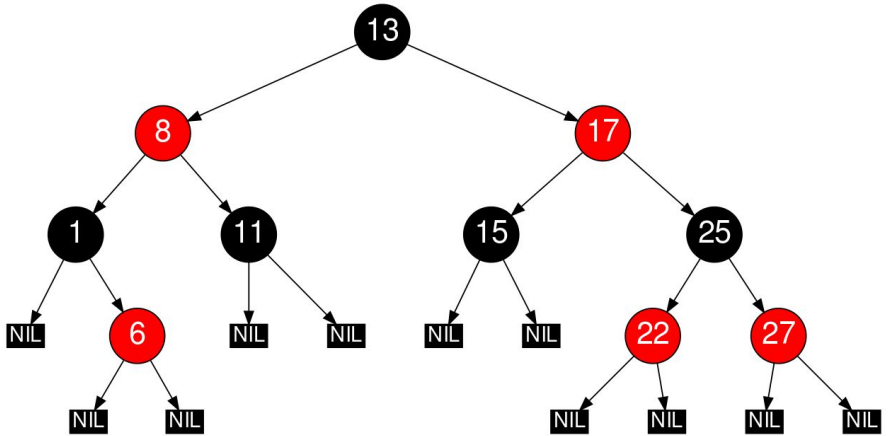




А ещё есть более простые контейнеры: однонаправленный список, очередь, стек…

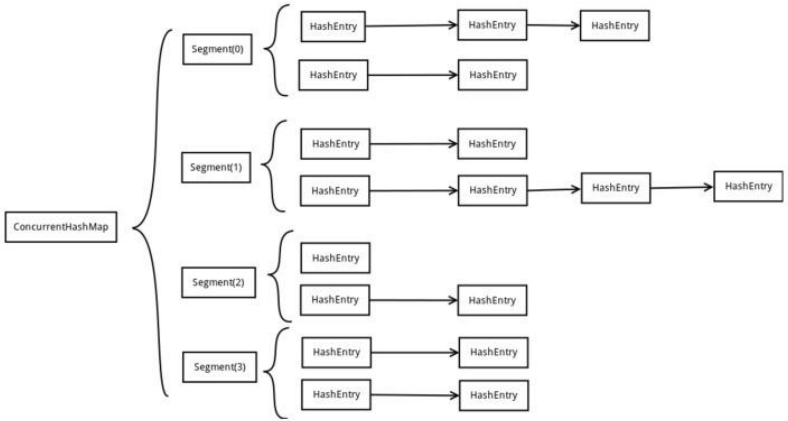
Ассоциативные контейнеры

map, set -- под контейнером -- RBTree (вариант сбалансированного дерева поиска)



- Map -- {k,v} -- хранилище, в случае упорядочивания -- упорядочивание по ключам, {k,v} -- std::pair
- Set -- {v}, значение и есть ключ

unordered_map/set -- под контейнером -- хеш таблица



- multi* -- обобщение set, map

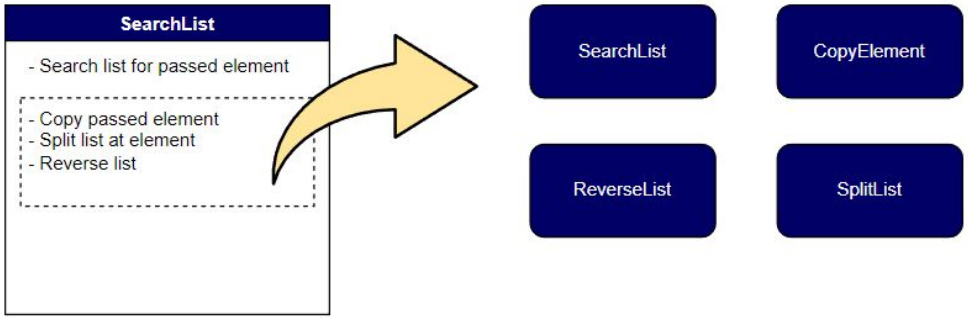
SOLID

Принципы “красивого” ООП-дизайна, и не более того…

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

Single Responsibility

- **Каждый объект должен иметь одну обязанность [причину для изменения]**
- **Эта обязанность должна быть полностью инкапсулирована в объект**



SRP: Splitting a non-SRP class

“A class should only have a single responsibility, that is, only changes to one part of the software’s specification should be able to affect the specification of the class.” -Robert C. Martin

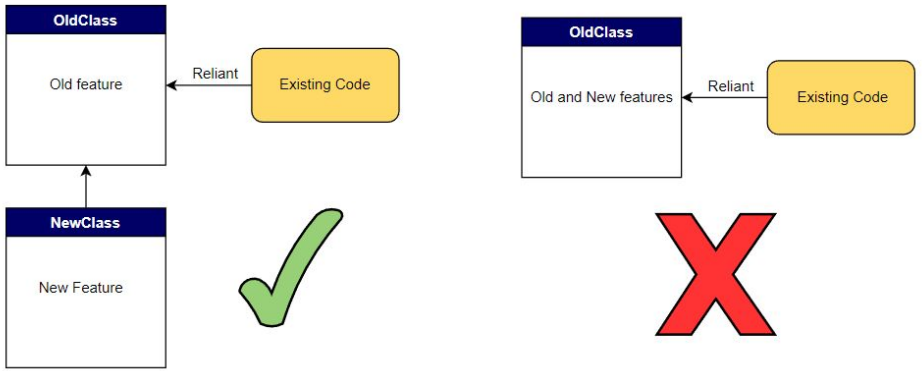
Изображения: https://www.educative.io/blog/solid-principles-oop-c-sharp

Open/Closed

- **Программные сущности (классы, модули, функции, и др) должны быть открыты для расширения, но закрыты для изменения**
  - **переиспользование через наследование --> полиморфизм**
  - **неизменные интерфейсы**

То есть мы определяем однозначные точки для расширения, и их меняем.

- **В интерфейсной части систем используется меньше конкретных классов -- только виртуальные (интерфейсы) и сырые данные.**



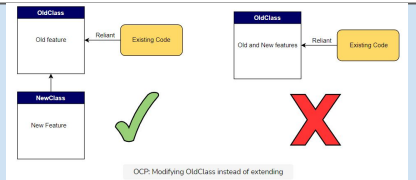
OCP: Modifying OldClass instead of extending

“Software entities … should be open for extension, but closed for modification.” Robert C. Martin

Изображения: https://www.educative.io/blog/solid-principles-oop-c-sharp

Open/Closed: Расширение функциональности

- **Новые методы обработки старых данных**
- **Новые данные**
- **Переиспользование через наследование --> полиморфизм**
  - **Неизменные интерфейсы**
- **Композиция**
  - **Только внешние интерфейсы для доступа снаружи -- инкапсуляция**



“Software entities … should be open for extension, but closed for modification.” Robert C. Martin

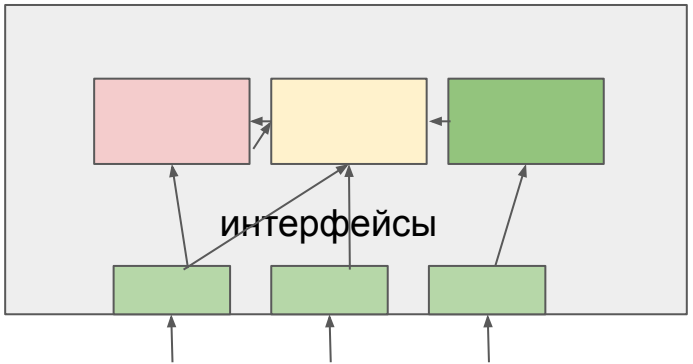
То есть мы определяем однозначные точки для расширения, и их меняем.

- **В интерфейсной части систем используется меньше конкретных классов -- только виртуальные (интерфейсы) и сырые данные.**
- **Перекрытие базовых методов**
- **Множественное наследование**

Наследование

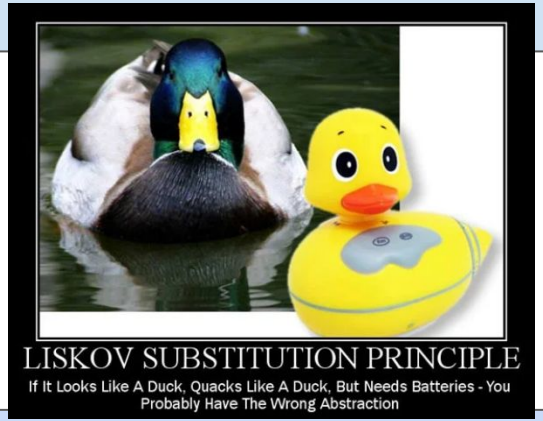
+ Новые данные	
	Данные
+ Новый код	Код

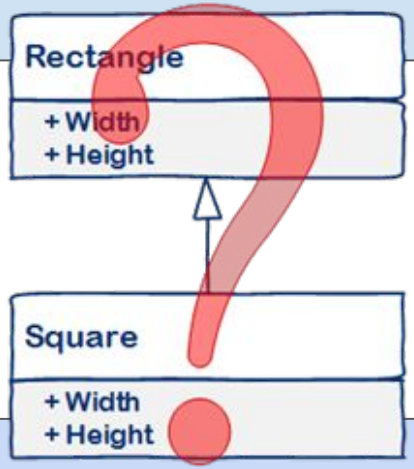
Агрегирование



Liskov Substitution

- **Это скорее определение наследования -- функции, которые используют базовый тип, должны использовать подтипы базового типа, не зная об этом**
- **Если это потомок, и предок где-то используется, то и потомок может использоваться там же. И инварианты предка должны выполняться в потомке.**





В терминологии контрактного программирования:

1. Производные классы не должны усиливать предусловия (не должны требовать большего от своих клиентов).
2. Производные классы не должны ослаблять постусловия (должны гарантировать, как минимум тоже, что и базовый класс).
3. **Инварианты базового класса и наследников суммируются**
4. Производные классы не должны генерировать исключения, не описанные базовым классом.

Interface Segregation

- **Клиенты не должны зависеть от методов, которые они не используют**
  - **слишком “толстые” интерфейсы необходимо разделять на более “мелкие” и специфические**
- **Удобство: не перекомпилировать клиента 2, если что-то изменилось для клиента 1**

```
struct Signal;

struct IMachine {
    virtual void gsm(Signal &bytes) = 0;
    virtual void wifi(Signal &bytes) = 0;
    virtual void streaming(Signal &bytes) = 0;
};

struct MultimediaModem : IMachine {
    void gsm(Signal &bytes) override { }
    void wifi(Signal &bytes) override { }
    void streaming(Signal &bytes) override { }
};

struct WIFIRouter : IMachine {                    // Not OK
    void gsm(Signal &bytes) override { }
    void wifi(Signal &bytes) override { // do connect }
    void streaming(Signal &bytes) override { }
};
```

```
struct IMultimediaModem {
    virtual void gsm(Signal &bytes) = 0;
    virtual void streaming(Signal &bytes) = 0;
};

struct IWIFIRouter {
    virtual void wifi(Signal &bytes) = 0;
};

struct MultimediaModem : IMultimediaModem {
    void gsm(Signal &bytes) override { }
    void streaming(Signal &bytes) override { }
};

struct WIFIRouter : IWIFIRouter {
    void wifi(Signal &bytes) override {  }
};

struct IMachine : IWIFIRouter, IMultimediaModem { };

struct Machine : IMachine {
    IWIFIRouter&   m_router;
    IMultimediaModem&   m_modem;
    Machine(IWIFIRouter &p, IMultimediaModem &s) : m_router{p}, m_modem{s} { }
    void gsm(Signal &bytes) override {m_modem.gsm(bytes); }
    void streaming(Signal &bytes) override {m_modem.streaming(bytes); }
    void wifi(Signal &bytes) override { m_router.wifi(bytes); }
};
```

Dependency Inversion

- **Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.**
- **Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракции.**

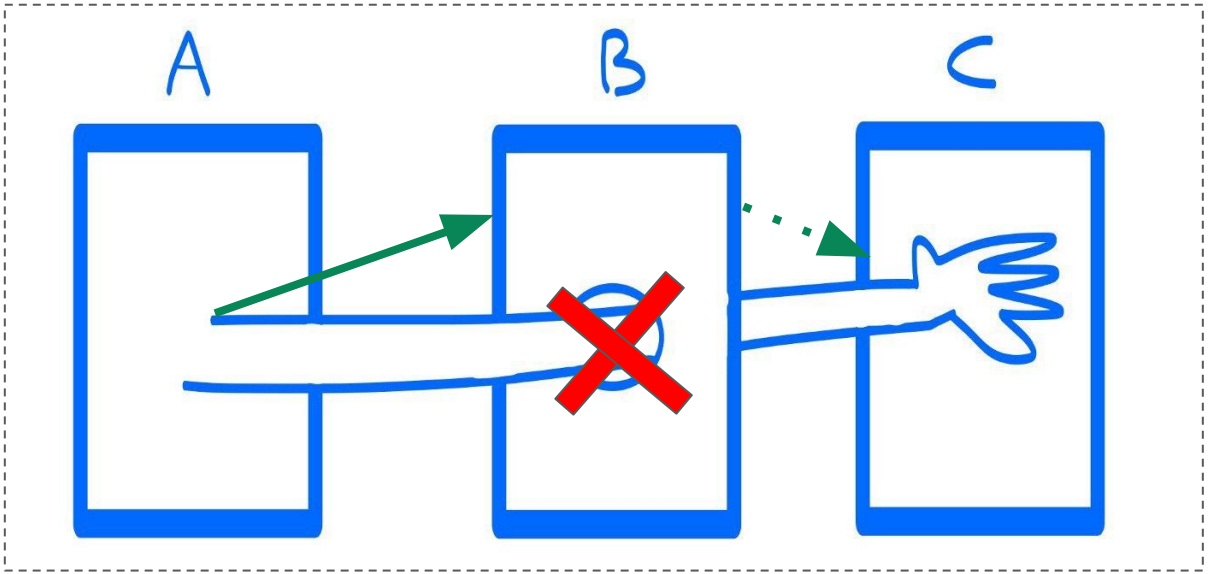
- **Модули (или классы) верхнего уровня** = классы, которые выполняют операцию при помощи инструмента
- **Модули (или классы) нижнего уровня** = инструменты, которые нужны для выполнения операций
- **Абстракции** – представляют интерфейс, соединяющий два класса
- **Детали** = специфические характеристики работы инструмента

Dependency Inversion

```
class FrontEndDeveloper {
 public:
  void developFrontEnd();
};

class BackEndDeveloper {
 public:
  void developBackEnd();
};

class Project {
 public:
  void deliver() {
    front_resp.developFrontEnd();
    back_resp.developBackEnd();
  }
 private:
  FrontEndDeveloper front_resp;
  BackEndDeveloper back_resp;
};
```

```
class Developer {
 public:
  virtual ~Developer() = default;
  virtual void develop() = 0;
};

class FrontEndDeveloper : public Developer {
 public:
  void develop() override { developFrontEnd(); }
 private:
  void developFrontEnd();
};

class BackEndDeveloper : public Developer {
 public:
  void develop() override { developBackEnd(); }
 private:
  void developBackEnd();
};

class Project {
 public:
  using Developers = std::vector<std::unique_ptr<Developer>>;
  explicit Project(Developers developers)
      : developers_{std::move(developers)} {}

  void deliver() {
    for (auto &developer : developers_) {
      developer->develop();
    }
  }

 private:
  Developers developers_;
};
```

Закон Деметры -- закон минимального (транз.) знания



В более точной формулировке закон Деметры гласит, что метод *f* класса *C* должен ограничиваться вызовом методов следующих объектов:

- *C*;
- объекты, созданные *f*;
- объекты, переданные *f* в качестве аргумента;
- объекты, хранящиеся в переменной экземпляра *C*.

Метод *не должен* вызывать методы объектов, возвращаемых любыми из разрешенных функций. Другими словами, разговаривать можно с друзьями, но не с чужаками.

2. Проектирование ПО

2.4. Об архитектуре и паттернах проектирования

Признаки удачно спроектированной архитектуры

Эффективность системы

Надёжность
Безопасность
Производительность
Масштабируемость
Отзывчивость
…

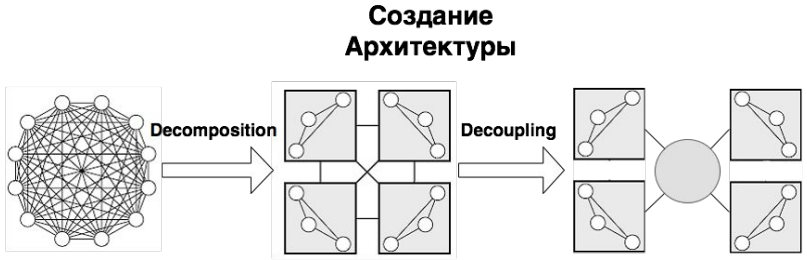
Гибкость системы

Легкость изменения текущей ф-ти
Исправление ошибок
Настройка системы
- Под пользователя
- Под различные сценарии

…

Поддерживаемость системы

Масштабируемость процесса разработки
Тестируемость
Обновляемость
Переиспользуемость
Обратная совместимость
…



Признаки неудачно спроектированной архитектуры

Жесткость

Тяжело модифицировать
Модификация одного модуля влечёт за собой (избыточные) модификации в других

…

Хрупкость

Легкость изменения текущей ф-ти
Изменение в одном модуле нарушают другие модули

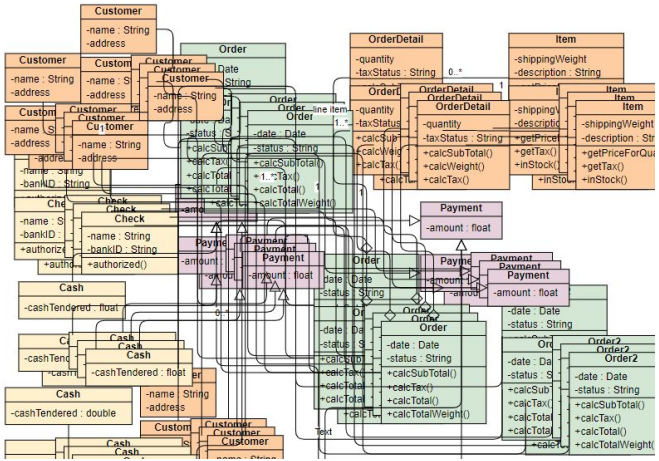
…

Неподвижность

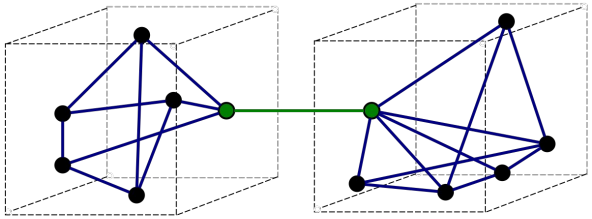
Тяжело “извлечь” модуль наружу

- Как правило, говорит о большом сопряжении и низкой связности некоторых модулей

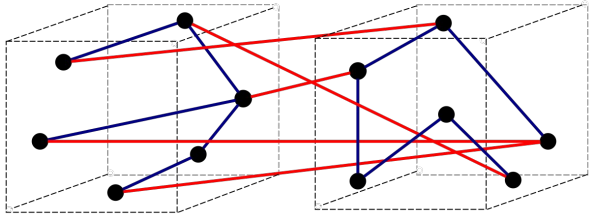
…



Модульность



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

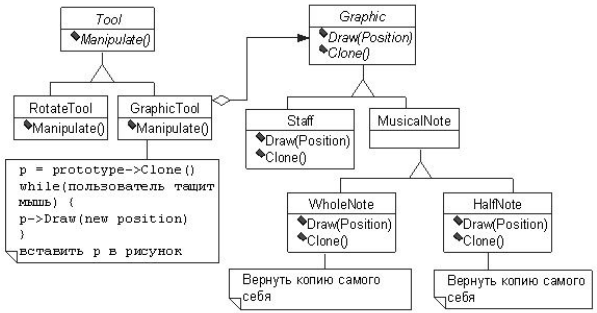
- Метрики взаимозависимости модулей
  - **Coupling (сопряжение)** -- мера того, насколько взаимозависимы разные модули в программе (по вызовам, как правило)
  - **Cohesion (связность)** -- степень, в которой задачи, выполняемые одним модулем, связаны друг с другом (в плане смысла)

Цель -- небольшое сопряжение и сильная связность (low coupling, high cohesion)

Паттерны

- Паттерн (от англ. Pattern) — образец, шаблон.
- В проектировании программ (и не только) -- это разумный, устоявшийся способ решения какой-либо задачи, который точно приведёт к намеченному результату*

- Паттерны ООП -- совсем тактические приёмы
- Архитектурные паттерны -- приёмы проектирования одной или нескольких подсистем
- Архитектурные стили -- что-то совсем глобальное …



- “Тропа умнее человека” (народное творчество)

Анти-паттерны

- В проектировании программ (и не только) -- это пример того, как не нужно решать те или иные задачи

- В ООП
- В кодировании
- Архитектурные
- Методологические

…