



Проектируем структуру классов

ЛЕКЦИЯ №11

Days 1 - 10

Teach yourself variables, constants, arrays, strings, expressions, statements, functions,...



Days 11 - 21

Teach yourself program flow, pointers, references, classes, objects, inheritance, polymorphism,



Days 22 - 697

Do a lot of recreational programming. Have fun hacking but remember to learn from your mistakes.



Days 698 - 3648

Interact with other programmers. Work on programming projects together. Learn from them.



Days 3649 - 7781

Teach yourself advanced theoretical physics and formulate a consistent theory of quantum gravity.



Days 7782 - 14611

Teach yourself biochemistry, molecular biology, genetics,...



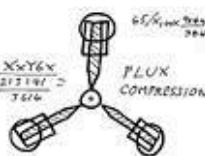
Day 14611

Use knowledge of biology to make an age-reversing potion.



Day 14611

Use knowledge of physics to build flux capacitor and go back in time to day 21.



Day 21

Replace younger self.



As far as I know, this is the easiest way to

"Teach Yourself C++ in 21 Days".

Software Design

проектирование программного обеспечения

1. Процесс который определяет **как** программа будет работать.
2. Процесс проектирования **не заканчивается** после написания программы. Он продолжается при внесении изменений в программу, при исправлении ошибок, при написании следующей версии и т.д.



Почему важно хорошее проектирование?



Martin Fowler

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Refactoring: Improving the Design of Existing Code, 1999

Хорошо спроектированную программу:

1. Проще и быстрее написать.
2. В него быстрее вносить изменения.
3. И главное, оно позволяет писать простой и понятный программный код!

Почему, вообще, появляются плохо спроектированные приложения?



Плохой дизайн классов появляется не только из-за неправильных решений принятых при написании программы. Он имеет особенность накапливаться при внесении рутинных изменений, таких как исправления ошибок.

Он как мусор, который выбрасывают из автомобилей: каждый выбрасывает по чуть-чуть и через какое-то время не видно обочины.

Можно ли качество программы померить?

- **Жесткость** - Как просто поменять структуру классов?
- **Хрупкость** – Как сложно испортить структуру программы?
- **Повторное использование**– Как сложно переиспользовать готовую структуру классов в других программах?

Как достигаем хорошей структуры класса?

1. Уменьшаем связанность кода.

Слабо-связанный код, позволяет вносить изменения в часть программы без влияния на остальной код.

2. Уменьшаем сложность кода.

3. Повышаем читаемость кода.

Делаем структуру классов однотипной.

4. Повышаем тестируемость кода.

Создаем методы классов таким образом, что бы их работу можно было легко протестировать.

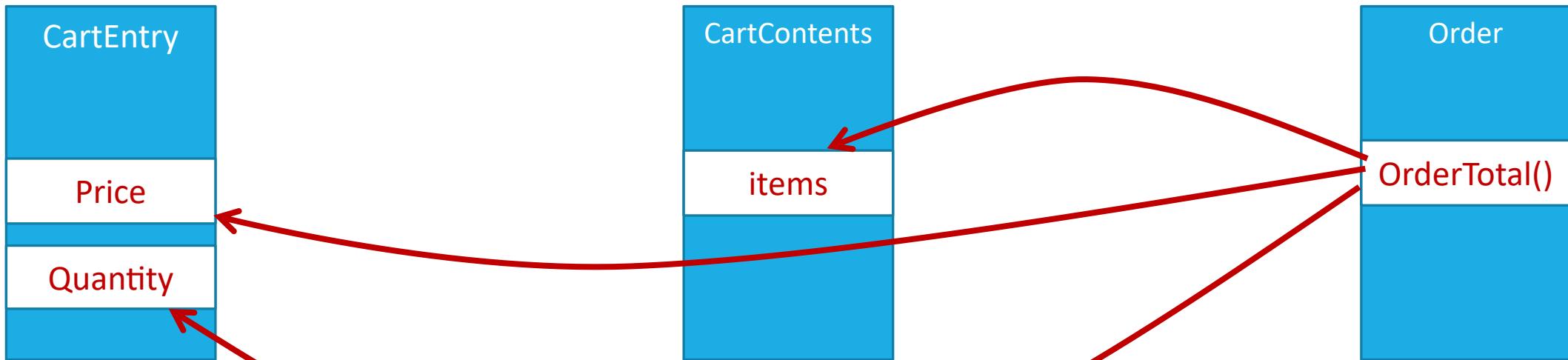


СИЛЬНО СВЯЗАННЫЙ КОД

```
1. struct CartEntry{
2.     float Price;
3.     int Quantity;
4. };
5. struct CartContents{
6.     std::vector<CartEntry> items;
7. };
8. class Order{
9.     private:
10.     CartContents _cart;
11.     float         _salesTax;
12.     public:
13.     Order(CartContents cart, float salesTax): _cart(cart), _salesTax(salesTax) {}
14.     float OrderTotal() {
15.         float cartTotal = 0;
16.         for (auto item:_cart.items) cartTotal += item.Price * item.Quantity;
17.         cartTotal += cartTotal*_salesTax;
18.         return cartTotal;
19.     }
20. };
```



Связи



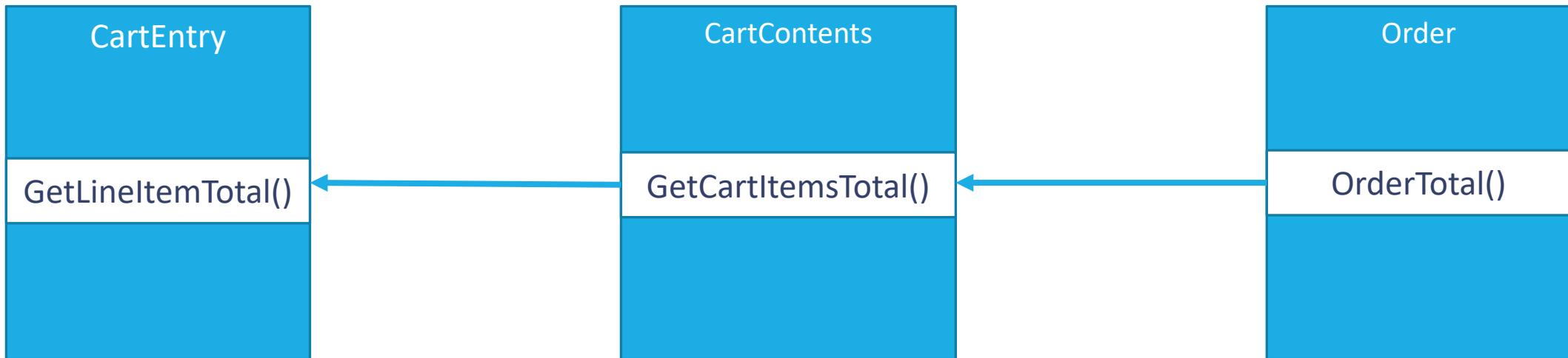
- Класс Order связан с несколькими классами;
- Класс Order связан с атрибутами классов;
- При изменении атрибутов в CartEntry или CartContents – придется менять Order;

Не так сильно связанный код

```
1. struct CartEntry{
2.     float Price;
3.     int Quantity;
4.     float GetLineItemTotal() { return Price * Quantity;      }
5. };
6. struct CartContents{
7.     std::vector<CartEntry> items;
8.     float GetCartItemsTotal() {
9.         float cartTotal = 0;
10.        for (auto item:items)    cartTotal += item.GetLineItemTotal();
11.        return cartTotal;
12.    }
13.};
14.class Order{
15. private:
16.     CartContents _cart;
17.     float          _salesTax;
18. public:
19.     Order(CartContents cart, float salesTax): _cart(cart), _salesTax(salesTax) {}
20.     float OrderTotal() {
21.         return _cart.GetCartItemsTotal() * (1.0f + _salesTax);
22.     }
23.};
```

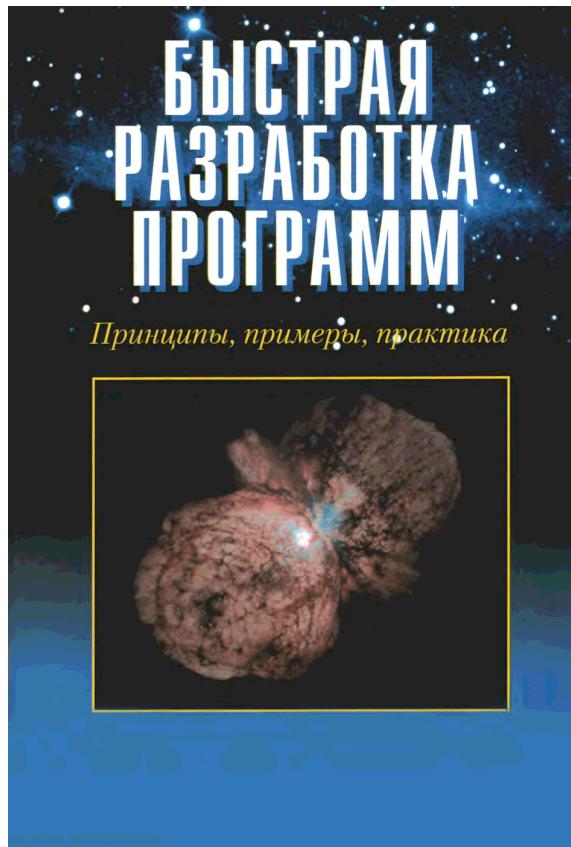


Связи



- Каждый класс связан только с одним классом;
- Классы связаны с методами, а не с атрибутами;
- При изменении атрибутов любого класса нужно менять только методы самого класса;

Роберт Сесил Мартин



*Быстрая разработка программ.
Принципы, примеры, практика.* —
Вильямс, 2004.

Разработал несколько прагматичных
принципов проектирования
программ, известных нам как SOLID.

SOLID

пять принципов хорошей структуры программы

- 1. Single Responsibility Principle**
- 2. Open/Close Principle**
- 3. Liskov Substitution Principle**
- 4. Interface Segregation Principle**
- 5. Dependency Inversion Principle**



Single Responsibility Principle

Принцип единственной обязанности.

- На каждый программный объект (это может быть как функция так и класс) должна быть возложена одна единственная ответственность.
 - Ответственность это то какую работу выполняет код, с точки зрения пользователя. Т.е. Это элементарная осмысленная под-задача в программе.
1. Требования к программе реализуются с помощью набора классов/функций.
 2. У каждого класса/функции есть «ответственность» в решении задачи.
 3. Чем больше ответственостей у класса/функции тем больше вероятность что его придется менять при изменении требований.
 4. Таким образом, появляется вероятность что при изменении одного класса мы повлияем на реализацию нескольких требований. Это может быть причиной ошибок.

Пример программируем модем



1. Позволяет звонить;
2. Позволяет передавать данные;
3. Позволяет принимать данные;
4. В конце сеанса связи – нужно разъединится;

Простой модем, для отправки текста

Example82_SRPModem1

```
1.class Modem {  
2.protected:  
3....  
4.public:  
5.Modem( ) {...}  
6.// Установка соединения  
7.bool Deal(const char* value) {...}  
8.// Проверка соединения  
9.bool Connected() {...}  
10.// Разрыв соединения  
11 void Hangup( ) {...}  
12.// Отправка данных  
13 void Send(char Data) {...}  
14.// Получение данных  
15 char Receive( ) {...}  
16.};
```

В классе есть две ответственности:

- Работа с соединением
- Передача и прием данных



Добро пожаловать, в реальный мир!



Изменяем задачу:

Вышла новая версия модема! Теперь есть модемы, которые умеют посыпать сразу целые числа!

Вторая версия программы Example83_SRP_Modem2

```
1.class Modem {  
2.protected:  
3....  
4.public:  
5.Modem( ) {...}  
6.bool Deal(const char* value) {...}  
7.bool Connected( ) {...}  
8 void Hangup( ) {...}  
9.// Поменялась сигнатура и реализация метода  
10 void Send(int Data) {...}  
11.// Новый метод (т.к. Мы теперь не можем определять конец по char == -1)  
12.bool NotEmpty ( ){...}  
13.// Поменялась сигнатура и реализация метода  
14.int Receive( ) {...}  
15.};
```



Ретроспектива

1. Пришлось **поменять** сигнатуру двух методов в классе Modem;
2. Пришлось **поменять** атрибуты класса modem;
3. Пришлось **поменять** реализацию конструктора ;
4. Пришлось **добавить** новый метод;
5. Пришлось **изменить** код проверки модема в функции main;

Изменения в коде, а тем более в сигнатуре метода приводят к тому, что мы ломаем существующие алгоритмы. Они перестают работать и требуют полной перепроверки! Это долго и сложно!

Рефакторинг: применяем SRP

Example84_SRP_Modem3

```
1. // Ответственность 1  
2. class IConnection {  
3. protected:  
4. ...  
5. public:  
6. IConnection() {...}  
7. virtual bool Deal(const char* value)  
{...}  
8. virtual bool Connected() {...}  
9. virtual void Hangup() {...}  
10.}
```

```
1. // Ответственность 2  
2. class ICharModem {  
3. protected:  
4. ...  
5. public:  
6. virtual void Send(char Data) {...}  
7. virtual bool NotEmpty() {...}  
8. virtual char Receive() {...}  
9. };
```

```
class Modem : public IConnection, public  
ICharModem {};
```

После внесения изменений

Example85_SRP_Modem4

```
class IIntModem {  
protected:  
...  
public:  
    virtual void Send(int Data) {...}  
    virtual bool NotEmpty() {...}  
    virtual int Receive() {...}  
};
```

1. Добавляем новый класс
2. Добавляем новую функцию для тестирования класса
3. Единственный код, который меняем:

```
class Modem : public  
IConnection, public IIntModem  
{};
```

Метрики качества программного кода

Cohesion – согласованность

Cohesion это свойство объекта / класса, определяющие насколько объект / класс занят своим делом.

Если **cohesion** низкое, то у класса слишком много ответственности, класс делает слишком много различных операций, отчего становится большим, а большой класс тяжело читать, тяжело расширять и т. д.

Чем выше **cohesion** (согласованность) – тем класс проще для понимания и изменения!

Связанность (coupling)

1. **Связанность (coupling)** – степень того, насколько сильно модули зависят друг от друга.
2. **Как измерять:** можно посчитать количество и исходящих связей (вместе или по отдельности) между пакетами, классами, отдельными функциями.
3. **Loose coupling** (принцип): связанность должны быть слабой, т.е. связей между программными сущностями должно быть как можно меньше, и они должны быть как можно слабее.
4. **Сильно зависимый код** приводит к тому, что изменения в одном классе/методе ведут к необходимости вносить изменения в другие классы/методы.
5. Классы имеющие высокий **cohesion** обычно **слабо-связанные**;

7 Видов связности [1/2]

1. **Функционально связный класс/функция** содержит атрибуты/переменные, предназначенные для решения одной единственной задачи.
2. В **последовательно связанном классе** его объекты охватывают подзадачи, для которых выходные данные одной из подзадач являются входными для другой (открыть файл – прочитать запись – закрыть файл).
3. **Информационно связный класс** содержит объекты, использующие одни и те же входные или выходные данные. Так, по ISBN книги, можно узнать ее название, автора и год издания. Эти три процедуры (определить название, определить автора, определить год издания) связаны между собой тем, что все они работают с одним и тем же информационным объектом – ISBN.
4. **Процедурно связный класс** – это такой класс, объекты которого включены в различные (возможно, несвязанные) подзадачи, в которых управление переходит от одной подзадачи к следующей (сделать зарядку, принять душ, позавтракать, одеться, отправится на работу). В отличие от последовательно связанного модуля, в котором осуществляется передача данных, в процедурно связанном модуле выполняется передача управления.

7 Видов связности [2/2]

5. **Класс с временной связностью** – это такой класс, в котором объекты модуля привязаны к конкретному промежутку времени. Примером может являться класс, осуществляющий инициализацию системы. Элементы данного класса почти не связаны друг с другом за исключением того, что должны выполняться в определенное время.
6. **Класс с логической связностью** – это такой класс, объекты которого содействуют решению одной общей подзадачи, для которой эти объекты отобраны во внешнем по отношению к классу мире. Так, например, альтернативы: поехать на автомобиле, на метро, на автобусе – являются средством достижения цели: добраться в какое-то определенное место, из которых нужно выбрать одну.
7. **Класс со связностью по совпадению** содержит объекты, которые слабо связаны друг с другом (сходить в кино, поужинать, посмотреть телевизор, проверить электронную почту).

Какие «связанности» хорошие?

В программных системах должны присутствовать модули, имеющие следующие три меры связности: **функциональная, последовательная и информационная**, так как другие типы связности являются крайне нежелательными и осложняют понимание и сопровождение системы.

Как найти плохую связанность?

1. В классе есть методы, которые не вызывают других методов класса;
2. Атрибуты класса используются только в одном методе класса;
3. Класс приходится изменять, при любых изменениях в требованиях;

Open/Closed Principle (OCP)

Принцип открытости/закрытости

Программные сущности (классы, модули, функции и т.д.) должны быть открыты для расширения и закрыты для модификации.
(Б. Мейер, 1988)

В случае изменения требований, мы не должны закапываться в программе для поиска места где внести изменения, вместо этого мы должны добавлять новые классы, расширяющие функциональность программ.



Какие цели преследует OCP?

1. Помогает делать базовые алгоритмы – неизменными, т.е. независящими от вносимых изменений.
2. Новые классы-расширения вряд ли добавят ошибок в существующие алгоритмы.
3. Уменьшается количество проверок, которые нужно сделать для новых тестов.

Смысл метода в управлении алгоритмами через параметры-контексты. Расширяя такие параметры (например, через наследование) мы будем добиваться необходимых изменений в программе.

Пример с модемом

Теперь нам нужно уметь подключаться не только по номеру телефона, но и по IPv4 адресу. А это уже не строка, а 4 байта.

Нам придется менять IConnection, либо изменяя логику Deal (например, запрягов IP адрес в строку “10.1.12.45”) или добавлять еще один метод в интерфейс и менять структуру хранения данных в IConnection.

У нас плохой дизайн!

```
class IConnection {  
    ...  
    virtual bool Deal(const char* value) {  
        number = value;  
        connected = true;  
        return connected;  
    };  
    ...  
}
```

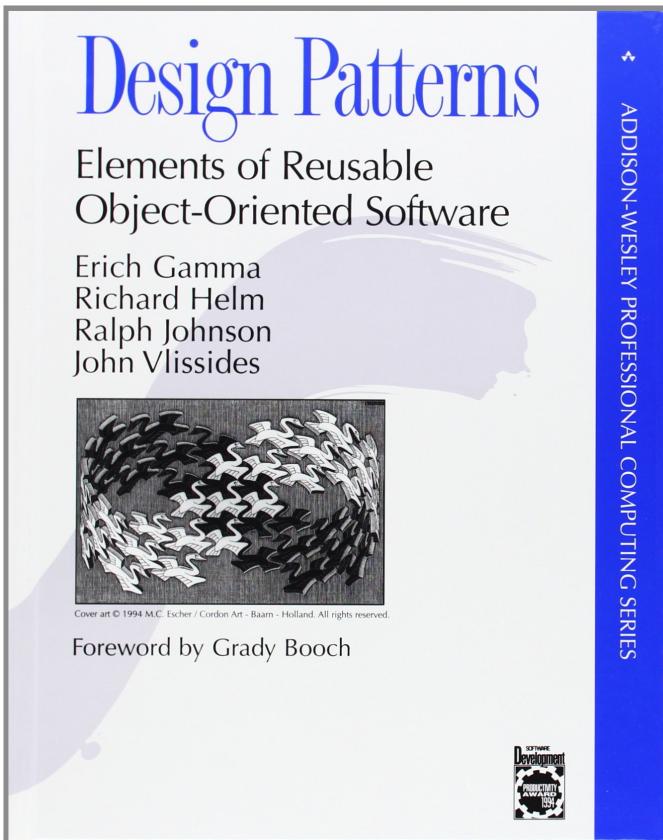
OCP: Выносим адрес в контекст!

Example86_OCP

1. Создаем отдельный класс `IAddress`;
2. Делаем в нем виртуальную функцию для получения адреса в виде удобном для драйвера модема (воображаемого);
3. Делаем у него два наследника `AddressPhone` и `AddressIP`;
4. В наследниках описываем структуру хранения адреса;
5. Теперь изменение типа адреса сводится просто к добавлению нового класса-наследника. Алгоритмы не меняются!

```
class IConnection {  
    ...  
    virtual bool  
    Deal(std::shared_ptr<IAddress> value) {  
        number = value;  
        connected = true;  
        return connected;  
    };  
    ...  
}
```

Книга про паттерны проектирования



- 1. Имя** – однозначно идентифицирует проблему и способ ее решения;
- 2. Задача** – Описание того когда надо применять паттерн
- 3. Решение** – Описание того из чего состоит паттерн (структура решения)
- 4. Результат** – Последствия применения паттерна

Паттерн TemplateMethod

Example87_TemplateMethod

1. Скелет алгоритма определяется в базовом классе. Алгоритм использует еще не определенные методы (которые будут описаны в наследниках)
2. Классы-наследники переопределяют методы, используемые в алгоритме.
3. Таким образом, мы можем менять элементы алгоритма только созданием новых классов-потомков.

```
class Mailer{  
protected:  
    virtual const char* Greeting() = 0;  
    virtual const char* Sign() = 0;  
    ...  
    const char* ComposeMail(){  
        static std::string mail;  
        mail = Greeting();  
        mail += msg;  
        mail += Sign();  
        return mail.c_str();  
    };
```

Теперь без виртуальных функций

CRTP:Example88_TemplateMethodCRTP

```
template <class T> class Mailer {  
...  
const char* ComposeMail() {  
    static std::string mail;  
    mail = ((T*)this)->Greeting();  
    mail += msg;  
    mail += ((T*)this)->Sign();  
    return mail.c_str();  
}  
};
```



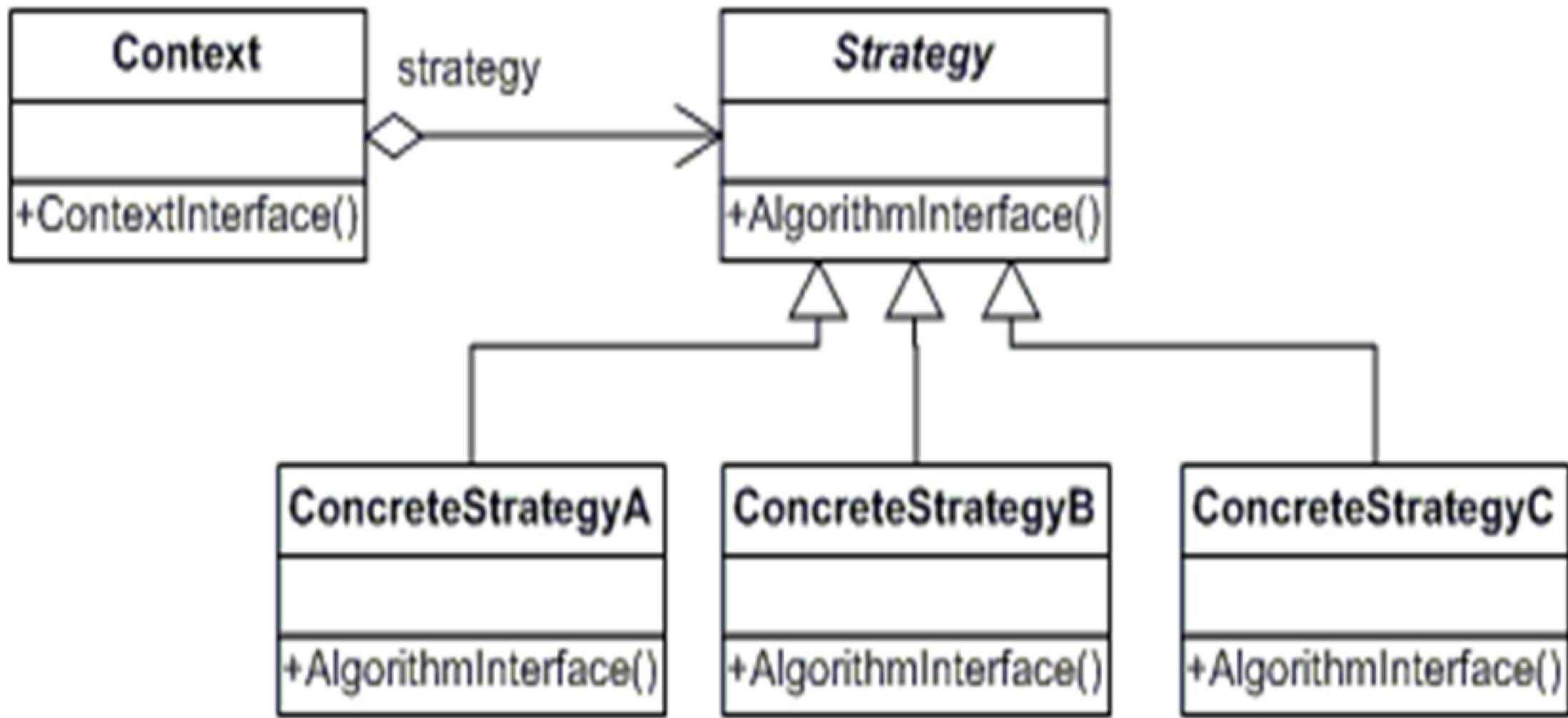
Strategy

Необходимо иметь много алгоритмов решения одной задачи (например, с какими-то особенностями).

Фактически, это механизм подключения plugi-in.

Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

1. **Strategy (Compositor)** - стратегия:
- объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс Context пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в классе ConcreteStrategy;
2. **ConcreteStrategy** - конкретная стратегия:
- реализует алгоритм, использующий интерфейс, объявленный в классе Strategy;
3. **Context (Composition)**
 - Контекст:
 - конфигурируется объектом класса ConcreteStrategy;
 - хранит ссылку на объект класса Strategy;
 - может определять интерфейс, который позволяет объекту Strategy получить доступ к данным контекста.



Пример

Example89 _ Strategy

```
template <class StrategyIF> class Context {  
protected:  
    // Reference to the strategy.    StrategyIF  
    strategy;  
public:  
    // Register reference to //strategy on  
    //construction.  
    Context(StrategyIF param) : strategy(param) {}  
    void contextInterface(const char *parm) {  
        strategy.algorithmInterface(parm);  
    }  
};
```

```
class ConcreteStrategy1 : public StrategyIF {  
public:  
    void algorithmInterface(const char* param)  
    override {  
        ...  
    }  
}  
  
class ConcreteStrategy2 : public StrategyIF {  
public:  
    void algorithmInterface(const char* param)  
    override {  
        ...  
    }  
}
```

Open/Close Principle

1. Помогает упростить внесение изменений в код.
2. Усложняет программу.
3. Стратегия применения: Если понадобилось внести изменение в код – просто его вносим. Если потребовалось внести изменение второй раз – переделываем код под ОСР.

Два раза на одни грабли не наступим!



Liskov Substitution Principle

Принцип подстановки Барбары Лисков

Объекты в программе могут быть заменены их наследниками без изменения свойств принципов работы программы.

Т.е. Классы наследники должны наследовать не только сигнатуры но и поведение класса- родителя!

1. Классы-наследники не должны убирать/скрывать поведение базового класса.
2. Классы-наследники не должны нарушать инварианты классов-родителей.
3. Классы-наследники не должны генерировать новых исключений по отношению к классам-родителям.
4. Классы-наследники могут свободно добавлять новые методы, которые расширяют поведение класса-родителя.

Смысл принципа достаточно прост:

Если НЕЧТО выглядит как утка,
крякает как утка, двигается
как утка и на вкус как утка,
почему это может
быть не уткой?

Мы определили для
себя утку...



Atkritka.com

Пример нарушения LSP

Example90_LSP

```
class Rectangle {  
protected:  
    int width, height;  
public:  
    Rectangle(int w, int h) : width(w), height(h)  
    {};  
    virtual void SetWidth(int value) {  
        width = value;}  
    virtual void SetHeight(int value) {  
        height = value;}  
    virtual int GetSquare() {return width *  
height;}};
```

```
class Square: public Rectangle {  
public:  
    Rectangle(int w, int h) :  
    Rectangle(w,w) {};  
virtual void SetWidth(int value) {  
    width = height =value;}  
virtual void SetHeight(int value)  
{  
    height = width = value;}  
    virtual int GetSquare() {return  
width * height;}};
```

Что же тут не так?

1. `Square` определяет новый инвариант, равенство сторон прямоугольника.
2. Проблема вызвана наличием мутаторов в контракте `Rectangle`.
3. С точки зрения геометрии (в математики все объекты неизменяемы) наследование корректно.

Инвариант

1. Утверждение о классе, выраженное пред или пост-условием
2. Условия которые описаны не в коде, а в тестах программы.
3. Безусловные утверждения о коде

Как избежать нарушения принципа?

1. Tell, Don't Ask

Не нужно запрашивать у объектов их внутреннее состояние, вместо этого – его нужно передавать в качестве параметра. Это делает объекты более универсальными.

2. Создавайте новые базовые типы.

Когда два объекта кажутся похожими но не являются наследниками друг-друга, то нужно создать общий родительский класс.

3. Создавайте тесты на проверку пост-условий выполнения методов!

Именно тест на проверку площади фигуры и нашел нашу ошибку.

Исправляем ситуацию

Example91_LSP2

```
class Figure {  
public:  
    virtual int GetSquare( ) = 0;  
};  
  
class Rectangle : public Figure {...}  
class Square : public Figure { ... }
```



Tell Don't Ask

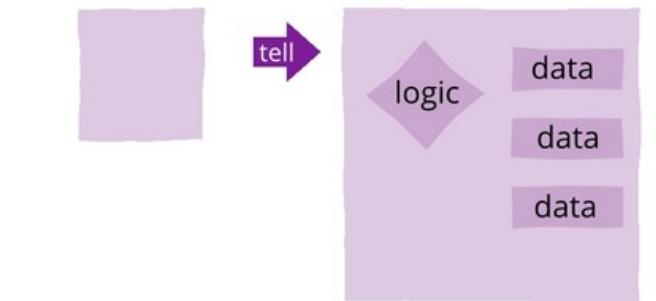
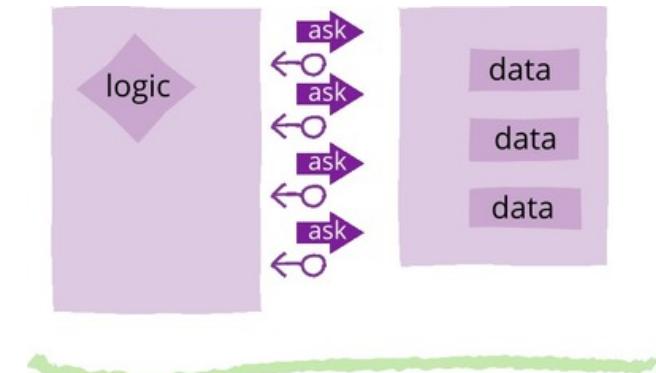
Example98_TDA

Один из основополагающих принципов ООП: Необходимо делегировать объекту действия, вместо того, что запрашивать его детали реализации. Это помогает достичь многократного использование класса (поскольку ни кто не знает его деталей реализации).

Command Query Separation – принцип разделение методов, которые выполняют какие-либо действия (tell) и методов, которые осуществляют запросы данных (ask).

Law of Demeter – объект может вызывать методы только:

- Себя
- Своих параметров
- Объектов, созданных внутри метода



Побочные эффекты и чистые выражения

1. **Чистое выражение** (referentially-transparent expression) – выражение, которое может быть заменено на свое значение без влияния на выполнение программы.
2. **Побочные эффекты** – любые изменения состояния программы, помимо порождения результата (или возбуждения исключения).
3. К побочным эффектам относятся любые изменения состояния объектов, глобальных переменных, ввод-вывод.
4. **Чистые выражения** = выражения, не имеющие **побочных эффектов**.

Принцип Command Query Separation

Операция либо имеет побочные эффекты (команда), либо возвращает значение (запрос), являясь чистой функцией.

(Б. Мейер, 1988)

Принцип говорит о том, что не должно быть функций, которые и возвращают результат запроса и меняют состояние объекта (имеют побочный эффект).

Пример: «одноразовое» хранилище

Example92_CQS1

```
1. class MessageStore {  
2. public:  
3.     const std::string Save(size_t index, const char* msg) {  
4.         ...  
5.     }  
6.     void Read(size_t index) {  
7.         ...  
8.     }  
9. }
```

Где command? Где query?

Пример: исправляем ситуацию

Example92_CQS2

```
1. class MessageStore {  
2. public:  
3.     void Save(size_t index, const char* msg) {  
4.         ...  
5.     }  
6.     const std::string Read(size_t index) {  
7.         ...  
8.     }  
9. }
```

В Read не должно быть побочного эффекта

Принцип надежности

Принцип также известен как **закон Постеля** после интернет-пионера **Джона Постеля**, который написал в ранней спецификации протокола TCP что:

TCP должны следовать за общим принципом надежности: будьте консервативны в том, что Вы делаете, быть либеральными в том, что Вы принимаете от других.

Другими словами, кодекс, который посыпает команды или данные к другим машинам (или к другим программам на той же самой машине) должен соответствовать полностью техническим требованиям, но кодекс, который получает вход, должен принять вход non-conformant, пока значение четкое.

Среди программистов, чтобы произвести совместимые функции, принцип популяризирован в форме **быть контравариантом во входном типе и ковариантный в типе продукции**.



Interface Segregation Principle

Принцип разделения интерфейса.

Много специализированных интерфейсов лучше, чем один универсальный.

Пользователи вашего класса не должны зависеть от методов, которые им не нужно использовать для решения своих задач.

Что плохого в «больших интерфейсах»?

1. Интерфейсы с большим числом методов трудно переиспользовать.
2. **Ненужные методы** приводят к увеличению связанности кода.
3. Дополнительные методы усложняют понимание кода.

Зато если у Вас много небольших интерфейсов можно быть уверенным, что каждый из них реализует одну ответственность!

Как найти «ненужный код»?

Что бы найти нарушения принципа нужно просто искать «неиспользуемый код»

1. Утилизация интерфейса

Если клиент, который использует класс, пользуется только частью методов – то скорее всего ваш класс предоставляет слишком «раздутые» интерфейсы.

2. Пустая реализация

Если есть методы, которые не реализованы (ни чего не делают) – то это то же признак раздущести интерфейса.

Следствие

Для каждого (типичного вида) клиента должны быть отдельная проекция контракта (т.е. свой интерфейс)

Пример: меню в ресторане

Example94 _ ISP1

```
class IBusinessMenu{  
protected:  
    virtual const char* GetFirstItem()=0;  
    virtual const char* GetSecondItem()=0;  
    virtual const char* GetCompot()=0;  
public:  
    void PrintMenu(){  
        ...  
    }  
};
```

- Нам нужно создавать меню для ресторана.
- Решаем что всегда меню состоит из трех блюд.
- А всегда ли это удобно?
Что если блюда два или четыре?

Собираем меню из простых интерфейсов

Example95_ISP2

```
template <class... Tail> class Menu {  
public:  
    void print() {}  
};  
  
template <class A, class ... Tail>  
class Menu<A, Tail...> : public Menu<Tail ...> {  
public:  
    A value;  
    Menu(A a, Tail ... tail) : value(a), Menu<Tail...>(tail...) {}  
    void print() {  
        std::cout << "Item:" << value.Value() << std::endl;  
        Menu < Tail...> &next = static_cast<Menu < Tail...>&> (*this);  
        next.print();  
    }  
};
```



Dependency Inversion Principle

зависимость от абстракции, а не от реализации

Принцип инверсии зависимостей.

Зависимости внутри системы строятся на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Уменьшаем сильные связи между классами:

1. Два класса сильно связаны, если они зависят друг от друга;
2. Изменения в одном, ведут к изменениям в другом;
3. Сильно связанные классы не могут работать отдельно друг от друга;

Пример: опять меню

Example96 _ DIP1

```
1. class ItemBulka : public IItem {  
2. ...  
3. };  
4. class ItemCoffe : public IItem {  
5. public:  
6. void print() override {  
7. std::cout << "Item coffe ";  
8. bool hasBulka = false;  
9. for (auto i : items) {  
10. if (dynamic_cast<ItemBulka*> (i.get()))  
11. std::cout << " for your cookie";  
12. }  
13. std::cout << std::endl;  
14. }  
15.};
```

Печатаем меню в ресторане.

Классы ItemBulka и ItemCoffe – элементы меню.

Если кофе подается с булкой то в названии кофе это отображаем.

Что тут не так:

1. А что если выпечка это не только ItemBulka?

Зависимость от класса ItemBulka.

2. А откуда взялась коллекция items?

Зависимость от реализации класса Menu.

Убираем зависимости

Example96 _ DIP2

```
class ISearchItem {  
public:  
    virtual bool isCookie() {return false;}  
    virtual bool accept(IIItem*) {  
        return false;}  
};  
class ISearchInterface {  
public:  
    virtual bool SearchMenu( ISearchItem  
*item) = 0;  
};  
class IIItem : public ISearchItem {  
public:  
    virtual void print( ISearchInterface  
*search) = 0;  
};
```

Убираем зависимости в абстрактные классы:

1. **ISearchItem** – класс для осуществления поиска элементов
2. Новое свойство базового класса – является ли он выпечкой (нарушили ли мы ISP?)
3. **ISearchInterface** – убираем зависимость от внутренней реализации коллекции (теперь это может быть не только vector)

Принцип Y.A.G.N.I. You Aren't Gonna Need It

Когда создается дизайн «про запас» или «для использования в будущем», подумайте – а что если требования поменяются так, что это не пригодится?

А что если я усложняю программу зря?





Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ