Проектируем структуру классов

ЛЕКЦИЯ №12

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:

Пример нарушения LSP
lsp_1.cpp

```
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

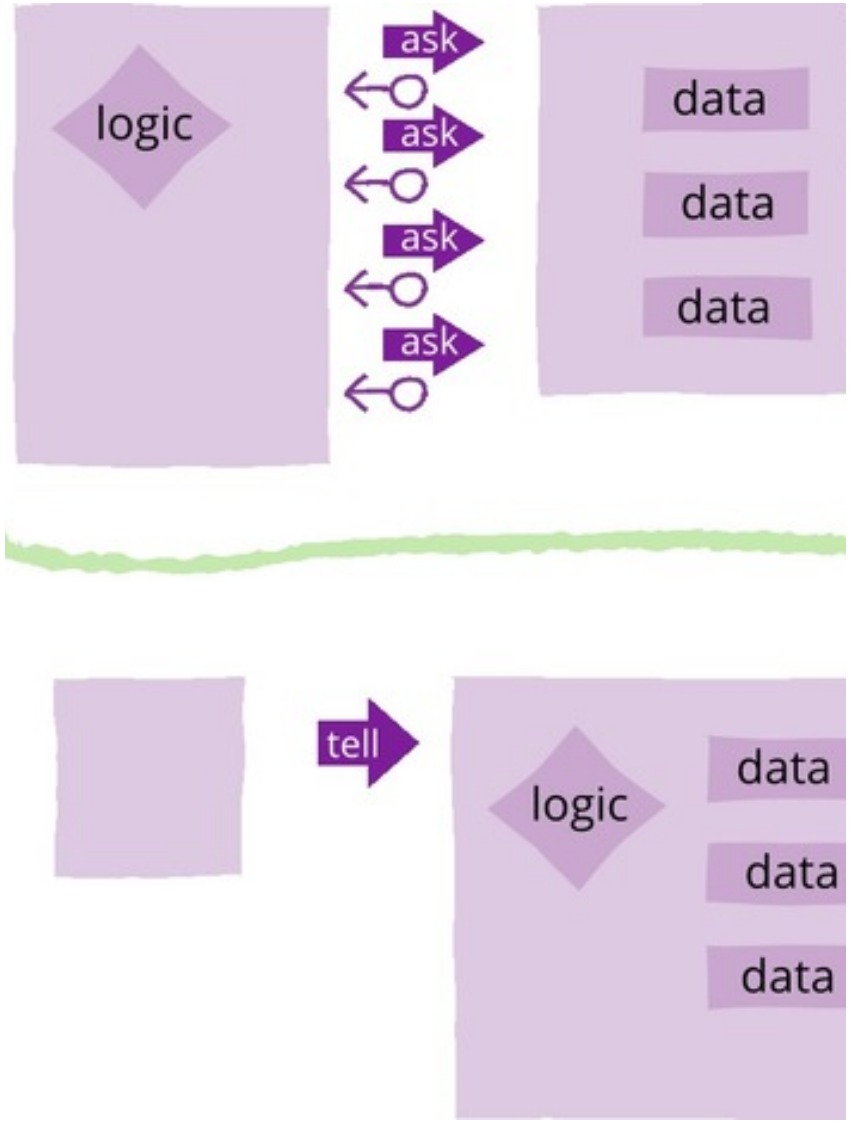
Как избежать нарушения принципа?

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 lsp_2.cpp

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

class Rectangle : public Figure {...}
class Square : public Figure { ... }
```



Tell Don’t Ask
tda.cpp

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 cqs_1.cpp

```
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 commad? Где query?

Пример: исправляем ситуацию cqs_2.cpp

```
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 посылает 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 зависить 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 раздутости интерфейса.

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

Пример: меню в ресторане isd_1.cpp

```
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

isp_2.cpp

```
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 связанны, 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 dip_1.cpp

```
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?**
   Зафисимость от реализации класса Menu.

Убираем зависимости dip_2.cpp

```
class ISearchItem {
public:
virtual bool isCookie( ) {return false;}
virtual bool accept(IItem*) {
return false;}
};
class ISearchInterface {
Public:
virtual bool SearchMenu(ISearchItem
*item) = 0;
};
class IItem : 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 сто если требования поменяются так, что это не пригодится?

А что если я усложняю программу зря?

Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ