Объектно-ориентированное проектирование

Где мы находимся?

- Мы уже рассмотрели:
  - Основные идеи и понятия OOA, OOD, OOP
  - Увидели на примере языка C++ как используется объектно-ориентированный подход в разработке ПО.
  - Основы языка UML
  - Познакомились с подходом RDD и принципами GRASP
- Что впереди:
  - Более детально поговорим о теоретических принципах OOD, и, имея уже некоторый опыт OOP, посмотрим, что такое «хороший» и «плохой дизайн».
  - Познакомимся с критериями SOLID
  - Рассмотрим OOA и OOD на конкретном примере

Базовые принципы OOD

(*повторение – мать не только заикания…*)

- **Абстракция** – выделение отдельных взаимодействующих объектов и их классов.
- **Инкапсуляция** – сокрытие внутренних деталей реализации и состояния объекта, и выставление наружу только важных для окружающих свойств или методов.
- **Наследование** – способ повторного использования кода, путем выделения частных случаев классов в подклассы, наследующие структуру и поведение классов-предков.
- **Полиморфизм** – обеспечение множественности вариантов реализации однотипного поведения различными классами-потомками общего предка. Т.е. все потомки умеют делать то, что умел делать предок, но каждый потомок может делать это по-своему.

Абстракция и DRY

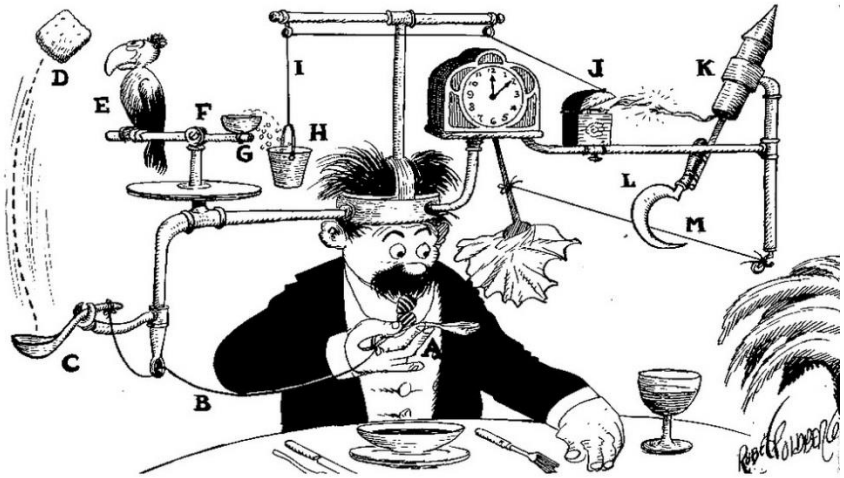
- **Абстракция** – это выделение в окружающем мире отдельных сущностей, которые мы будем моделировать.
- Производится через выделение **ключевых характеристик объекта**, отличающих его от других, и, таким образом, четкое определение границ объектов.
- Следствие – при грамотном абстрагировании (выделении) сущностей, каждая значительная часть функциональности системы описывается только **один раз** в исходном коде, что уменьшает затраты на реализацию и улучшает поддерживаемость кода.
- Принцип **DRY – Don’t Repeat Yourself – «Не повторяйся»** -
  - Когда принцип DRY применяется успешно, изменение единственного элемента системы не требует внесения изменений в другие, логически не связанные элементы. Те элементы, которые логически связаны, изменяются предсказуемо и единообразно.
  - Антипринцип – **WET – Write Everything Twice / We Enjoy Typing («Пиши все дважды»/ «Мы любим печатать»)** ☺

Инкапсуляция и связность/зацепление

- **Инкапсуляция** – обеспечение доступности главного, выделение основного содержания путем помещения всего мешающего, второстепенного в некую условную капсулу («черный ящик»).
- С правильной реализацией инкапсуляции тесно связаны характеристики **связности** и **зацепления** классов:
  - **Связность**, или **прочность** (*cohesion*) — способ и степень, в которой задачи, выполняемые некоторым классом, связаны друг с другом; мера силы взаимосвязанности элементов внутри модуля.
  - **Зацепление**, или **сцепление** (*coupling*) — способ и степень взаимозависимости между классами; сила взаимосвязей между классами; мера того, насколько связаны классы или модули.

Связность и зацепление

Слабая связность, сильное зацепление



Сильная связность, слабое зацепление

Закон Деметры

- Частный случай правила слабого зацепления.
  - объект должен иметь как можно меньше представления о структуре и свойствах чего угодно (включая собственные подкомпоненты).
  - Каждый программный модуль:
    - должен обладать ограниченным знанием о других модулях: знать о модулях, которые имеют «непосредственное» отношение к этому модулю;
    - должен взаимодействовать только с известными ему модулями «друзьями», не взаимодействовать с незнакомцами;
    - обращаться только к непосредственным «друзьям».
- Аналогия из жизни:
  - Если Вы хотите, чтобы собака побежала, глупо командовать её лапами, лучше отдать команду собаке, а она уже разберётся со своими лапами сама.

Закон Деметры

- Объект A не должен иметь возможность получить непосредственный доступ к объекту C, если у объекта A есть доступ к объекту B и у объекта B есть доступ к объекту C
- Метод M объекта O должен вызывать методы только следующих типов объектов:
  - собственно самого O
  - параметров M
  - других объектов, созданных в рамках M
  - прямых компонентных объектов O
  - глобальных переменных, доступных O, в пределах M
- «Используйте только одну точку»

Закон Деметры

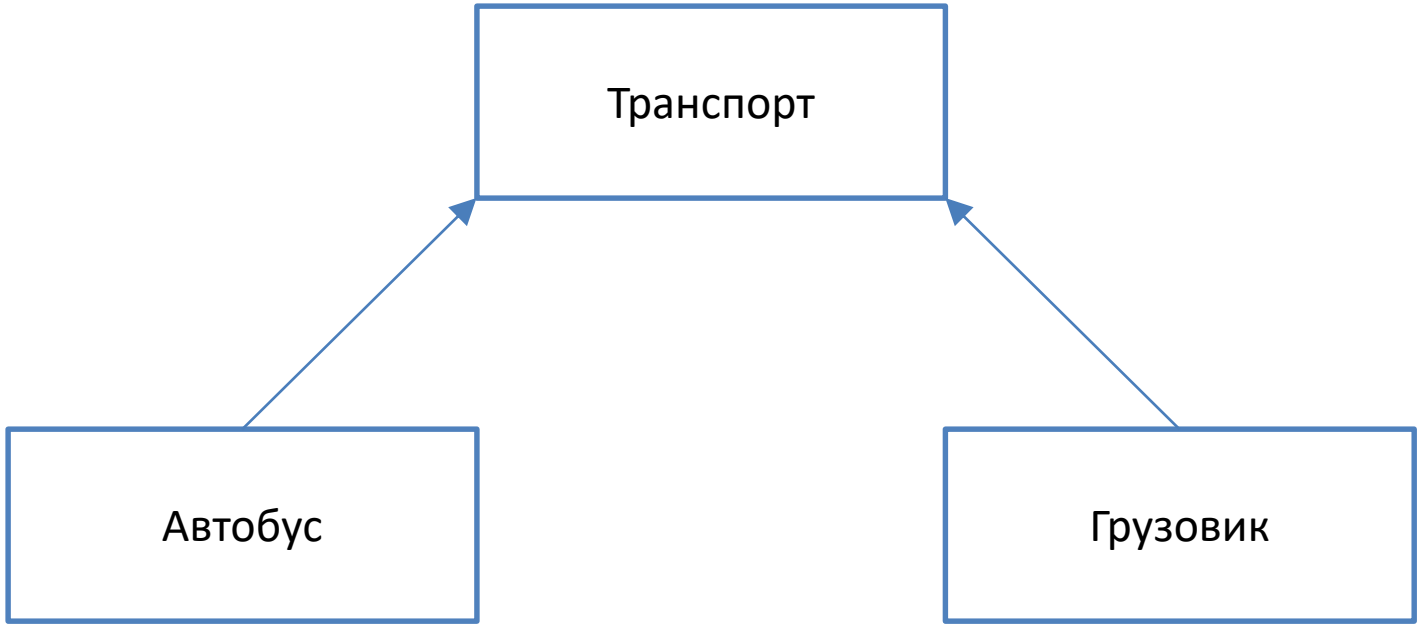
- Нарушение:
  - var v = anObject.GetThatComponent.GetSomeData();
- Выполнение:
  - var v = anObject.GetThatComponentsData();
- Преимущества:
  - Нет зависимости от реализации «ThatComponent»;
- Недостатки:
  - Необходимость реализации дополнительных методов;

Наследование и полиморфизм

- Наследование следует применять в случаях, когда:
  - Между классами имеется смысловое отношение «является» (is-a), а не «включает» (has-a) – **простой схожести набора полей недостаточно**;
  - Есть возможность повторного использования кода (код базового класса применим к классу-потомку);
  - Есть необходимость использования полиморфизма (использование одного класса и методов для разных подтипов);
  - Иерархия классов остается на разумном уровне глубины, и добавление нового уровня вряд ли вынудит других разработчиков сильно увеличивать ее глубину;
  - Необходимо внести существенные изменения в классы-потомки, изменяя базовый класс.
- Применение наследования в других ситуациях не оправдано.
  - Зачастую, в спорных случаях предпочтительно использовать композицию, а не наследование.

Композиция вместо наследования

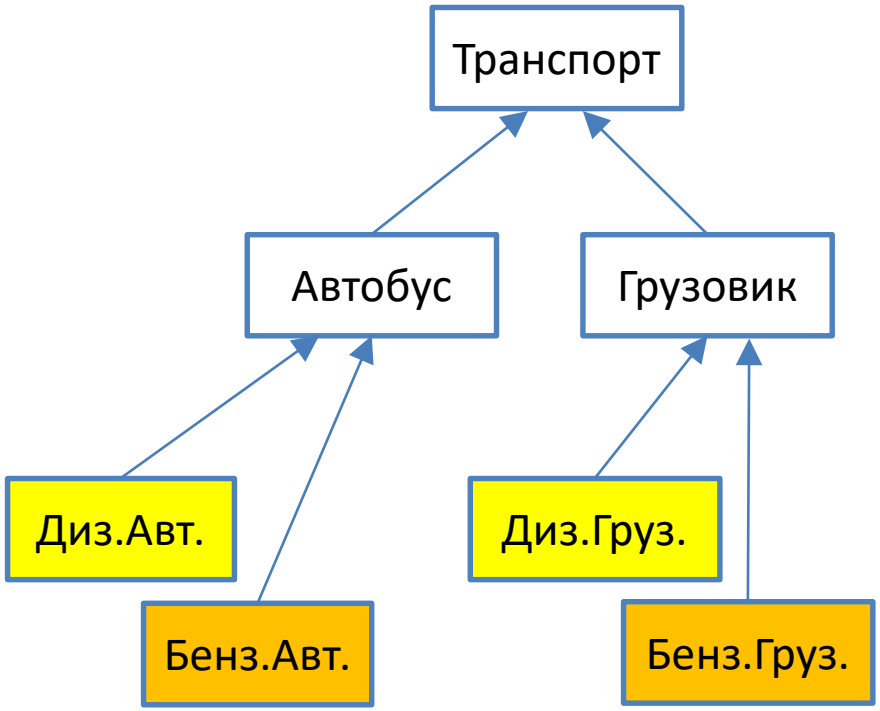
Простая иерархия по одному измерению - назначению транспортного средства



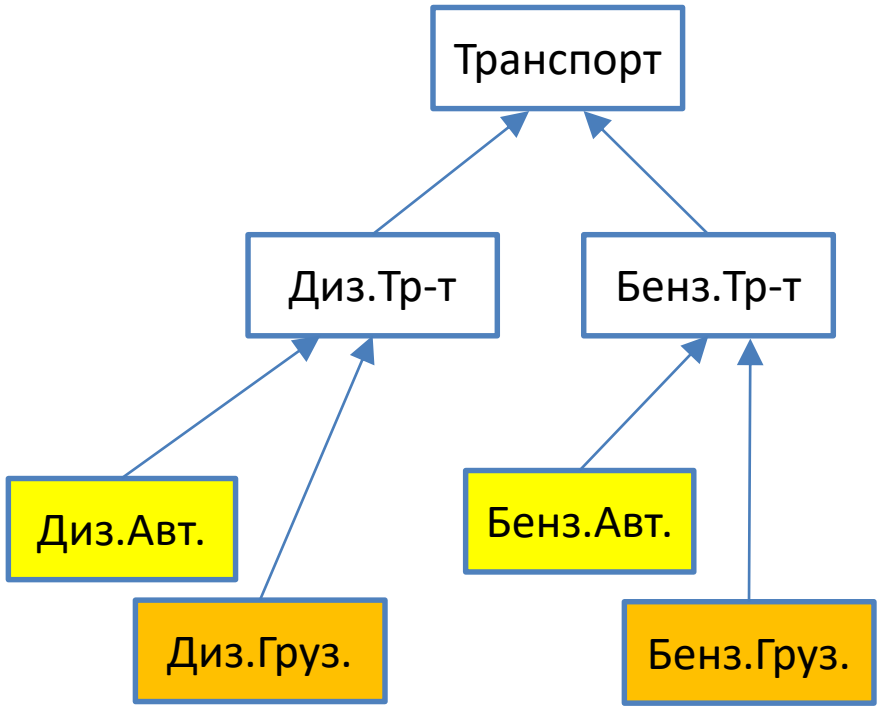
Композиция вместо наследования

Добавляем второе измерение – тип двигателя

Вариант 1



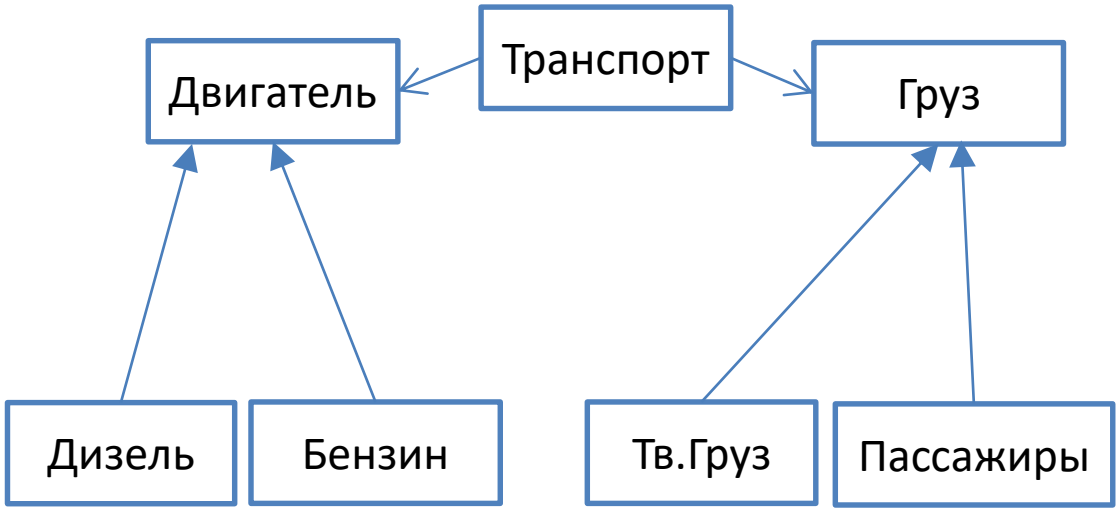
Вариант 2



Какой вариант выбрать?

Композиция вместо наследования

- Выделяем классы **Двигатель** и **Груз**
- Наследуем от Двигателя **Дизельный** и **Бензиновый** двигатели
- Наследуем от Груза **Твердый груз** и **Пассажиров**
- Добавляем прямо в класс **Транспорт** свойства **Двигатель** и **Груз** соответствующих типов



Критерии SOLID

SOLID by Uncle Bob (Robert Martin)

- **S** - Single responsibility principle

  На каждый класс должна быть возложена единственная обязанность.

- **O** - Open/closed principle

  Программные сущности должны быть открыты для расширения, но закрыты для изменения.

- **L** - Liskov substitution principle

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

- **I** - Interface segregation principle

  Много специализированных интерфейсов лучше, чем один универсальный.

- **D** - Dependency inversion principle

  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.

Single responsibility principle

- **Принцип единственной обязанности** - каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс. Все его сервисы должны быть направлены исключительно на обеспечение этой обязанности.
- Пример – генератор отчетов. Имеет две обязанности – выборка данных для отчета и формирование отчета. Должен быть разделен на два класса.
- Причина – большая устойчивость к изменениям. Меняем формат отчета – не трогаем выборку.

SRP – пример нарушения

```
// Класс, моделирующий объект Заказчик
class Customer
{
    // ... - часть кода пропущена для краткости

    void Add()
    {
        try
        {
            // Код, сохраняющий Заказчика в БД
        }
        catch (...)
        {
            std::ofstream fout("c:\Error.txt");
            fout << "Не получилось сохранить Заказчика в БД";
            fout.close();
        }
    }
};
```

Класс вынужден знать детали механизма логирования, например, имя лог-файла

SRP – скорректированный пример

```
class FileLogger
{
    public:
    void Handle(string error)
    {
        ofstream fout("c:\Error.txt");
        fout << error;
        fout.close();
    }
};

class Customer
{
    //...
    virtual void Add()
    {
        try
        {
            // Код, сохраняющий Заказчика в БД
        }
        catch (...)
        {
            new FileLogger()->Handle("Не получилось сохранить Заказчика в БД");
        }
    }
};
```

Open/closed principle

- **Принцип открытости/закрытости** – «программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения» - это означает, что такие сущности могут позволять менять свое поведение без изменения их исходного кода.
- Пример – любой случай наследования, расширяющего возможности (например Заказ и ЗаказПоТелефону)
- Причина - код, подчиняющийся данному принципу, не изменяется при расширении и поэтому не требует дополнительных трудозатрат.

O/CP – пример нарушения

```
class Customer
{
    //...

    int custType; // Тип заказчика (1 = «золотой» клиент, 0 = обычный)

    // Расчет скидочной цены
    double GetDiscountPrice(double totalSales)
    {
        if (custType == 1) //Для «золотого» клиента делаем скидку 15%
        {
            return totalSales * 0.85;
        }
        else
        {
            return totalSales;
        }
    }
};
```

А что, если потом понадобится ввести «серебряного» или «платинового» клиента?

O/CP – скорректированный пример

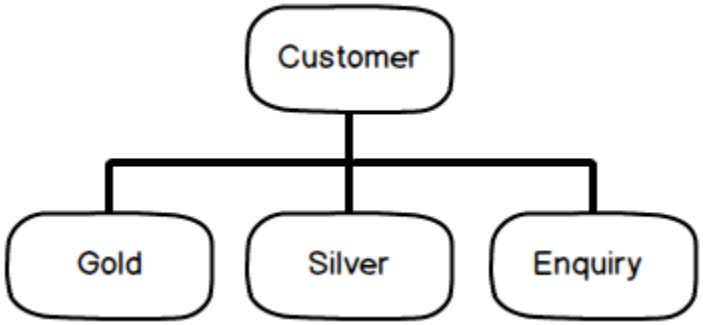
```
class Customer
{
    //...
    virtual double GetDiscountPrice(double totalSales)
    {
        //Для обычного клиента скидок не предусмотрено
        return totalSales;
    }
};


class GoldenCustomer : public Customer
{
    //...
    double GetDiscountPrice(double totalSales)
    {
        //Для «золотого» клиента делаем скидку 15% относительно базовой
        return Customer::GetDiscountPrice(totalSales)*0.85;
    }
};
```

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»
  - – т.е. поведение наследуемых классов не должно противоречить поведению, заданному базовым классом, то есть поведение наследуемых классов должно быть ожидаемым для кода, использующего переменную базового типа.
- Причина - код, использующий иерархию классов с нарушениями принципа Лисков, помимо оперирования ссылкой на базовый класс, оказывается также вынужден знать и о подклассе. Подобная функция нарушает принцип открытости/закрытости, поскольку она требует модификации в случае появления в системе новых производных классов.

LSP – пример нарушения

```
// Класс, моделирующий заказчика, приведенного рекламной
// кампанией, но не сделавшего еще ни одного заказа
class Enquiry : public Customer
{
    public override double GetDiscount(double totalSales)
    {
        return Customer::GetDiscount(totalSales)*0.95; //Скидка 5%
    }

    public override void Add()
    {
        throw “Нельзя сохранять Enquiry!"; //не сохраняем в БД
    }
}
```



LSP – пример нарушения – где возникает проблема

```
Customer *customers[3];
customers[0] = new Customer();
customers[1] = new GoldenCustomer();
customers[2] = new Enquiry();

// …

for(int i=0; i<3; i++)
{
    customers[i]->Add(); //вот тут мы упадем
}
```

А все дело в том, что, на самом деле Enquiry, хоть и кажется Customer’ом – это не настоящий Customer

LSP – скорректированный пример

```
class IDiscount //интерфейс вычисления скидки
{
    public:
    virtual double GetDiscount(double totalSales) = 0;
};

class IDatabase //интерфейс сохранения в БД
{
    public:
    virtual void Add() = 0;
};

class Enquiry : public IDiscount //реализует только IDiscount
{
    //...
    double GetDiscount(double totalSales)
    {
        return totalSales*0.95;
    }
};
```

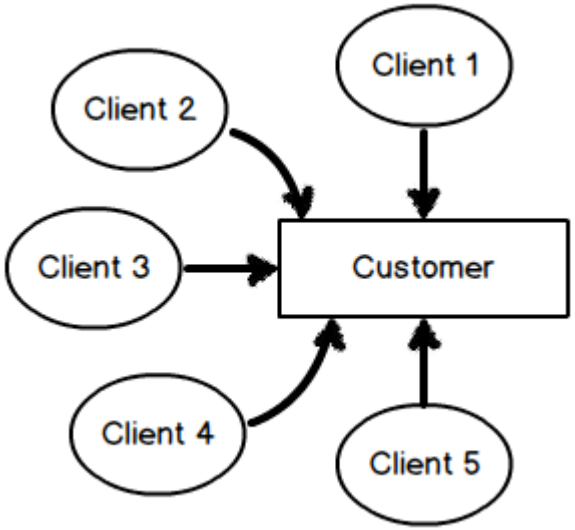
LSP – скорректированный пример

```
class Customer : public IDiscount, public IDatabase
{
    //...
    virtual void Add() {/*...*/ };

    virtual double GetDiscount(double totalSales)
    {
        return totalSales;
    }
};

IDiscount* leads[2];
leads[0] = new Customer();
leads[1] = new Enquiry();
// …
for(int i=0; i<2; i++)
{
   std::cout << leads[i]->GetDiscount(100));
   //leads[i].Add() вызвать уже не получится на этапе компиляции
}
```

Interface segregation principle

- **Принцип разделения интерфейса:** «Клиенты не должны зависеть от методов, которые они не используют.»
- Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе.
  - В итоге, при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют.
- Причина – устойчивость к изменениям.

ISP – пример нарушения




```
// расширяем интерфейс
class IDatabase
{
public:
    // старые клиенты используют этот метод
    virtual void Add() = 0;
    // этот метод добавлен для новых клиентов
    virtual void Read() = 0;
       // но поддерживать его придется всем
}
```

ISP – скорректированный пример

```
class IDatabaseReader // Интерфейс чтения из БД
{
public:
    virtual void Read() = 0;
}

class ReadableCustomer : public IDatabase, public IDatabaseReader
{

    void Add()
    {
        // Реализует запись в БД
    }

    void Read()
    {
        // Реализует чтение из БД
    }
}
```

Dependency inversion principle

- **Принцип инверсии (обращения) зависимостей:**
  - 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 абстракций.
- Причина - позволяет уменьшить **зацепление (coupling)** модулей, упростив их модификацию в будущем.
- Пример из жизни – чтобы сварить две железяки, мне нужен любой сварщик (абстрактный класс), а не именно электросварщик (конкретный класс) по имени Вася (конкретный объект)

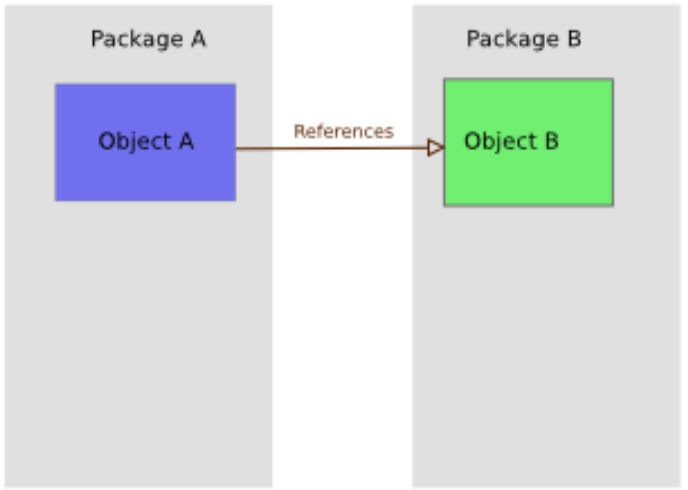


Figure 1

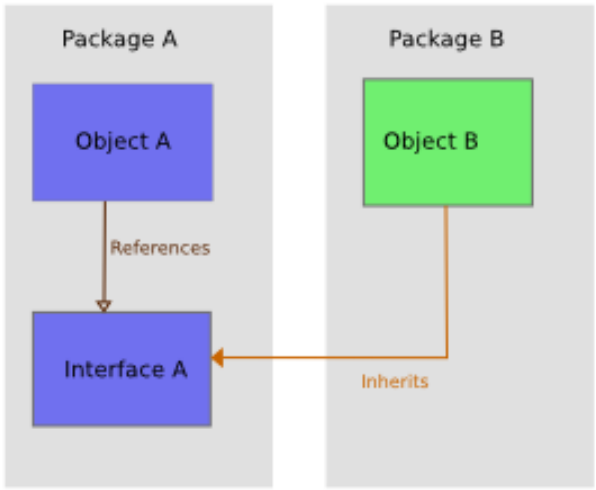


Figure 2

DIP – пример нарушения

```
class Customer
{
  //...
  FileLogger *logger = new FileLogger();
  virtual void Add()
  {
      try
      {
          // Код, сохраняющий Заказчика в БД, пропущен
      }
      catch (...)
      {
          logger->Handle("Не получилось сохранить Заказчика в БД");
      }
  }
}
```

А если я не хочу логировать в файл, а хочу в EventLog?

DIP – скорректированный пример

```
class ILogger //интерфейс логгера
{
public:
    virtual void Handle(string error) = 0;
};

class FileLogger : public ILogger
{
    void Handle(string error) {/*запись ошибки в файл*/ }
};

class EventViewerLogger : public ILogger
{
    void Handle(string error) {/*запись ошибки в Event Log */ }
};

// возможны и другие реализации (SNMP, Email, etc.)
```

DIP – скорректированный пример

```
class Customer : public IDiscount, public IDatabase
{
private:
    ILogger *logger;
public:
    // инжекция зависимости через конструктор
    Customer(ILogger *i) { logger = i; } // используем логгер, переданный клиентом

    virtual void Add()
    {
        try
        {
            // Код, сохраняющий Заказчика в БД пропущен
        }
        catch (...)
        {
            // логируем ошибку
            logger->Handle("Не получилось сохранить Заказчика в БД");
        }
    }
    // реализация IDiscount пропущена
}

Customer customer = new Customer(new EventViewerLogger()); // использование
```

SOLID и TDD

- TDD – Test Driven Development (Разработка через тестирование) – это техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки:
  - сначала пишется тест, покрывающий желаемое изменение,
  - затем пишется код, который позволит пройти тест,
  - затем проводится рефакторинг нового кода к соответствующим стандартам.
- Автор SOLID – Robert Martin – один из адептов методики TDD
  - Следование принципам SOLID делает код не только хорошо поддерживаемым и модифицируемым, но и удобно тестируемым.
    - Например, инверсия зависимости дает возможность инжекции объектов-заглушек (stub) или имитаторов (mock);