

The C# logo, consisting of a large white 'C' and a white '#' symbol, is positioned on the left side of the slide. It is set against a dark blue background that features a large, stylized, light blue geometric shape resembling a hexagon or a stylized 'A'.

Object-Oriented
Programming

4

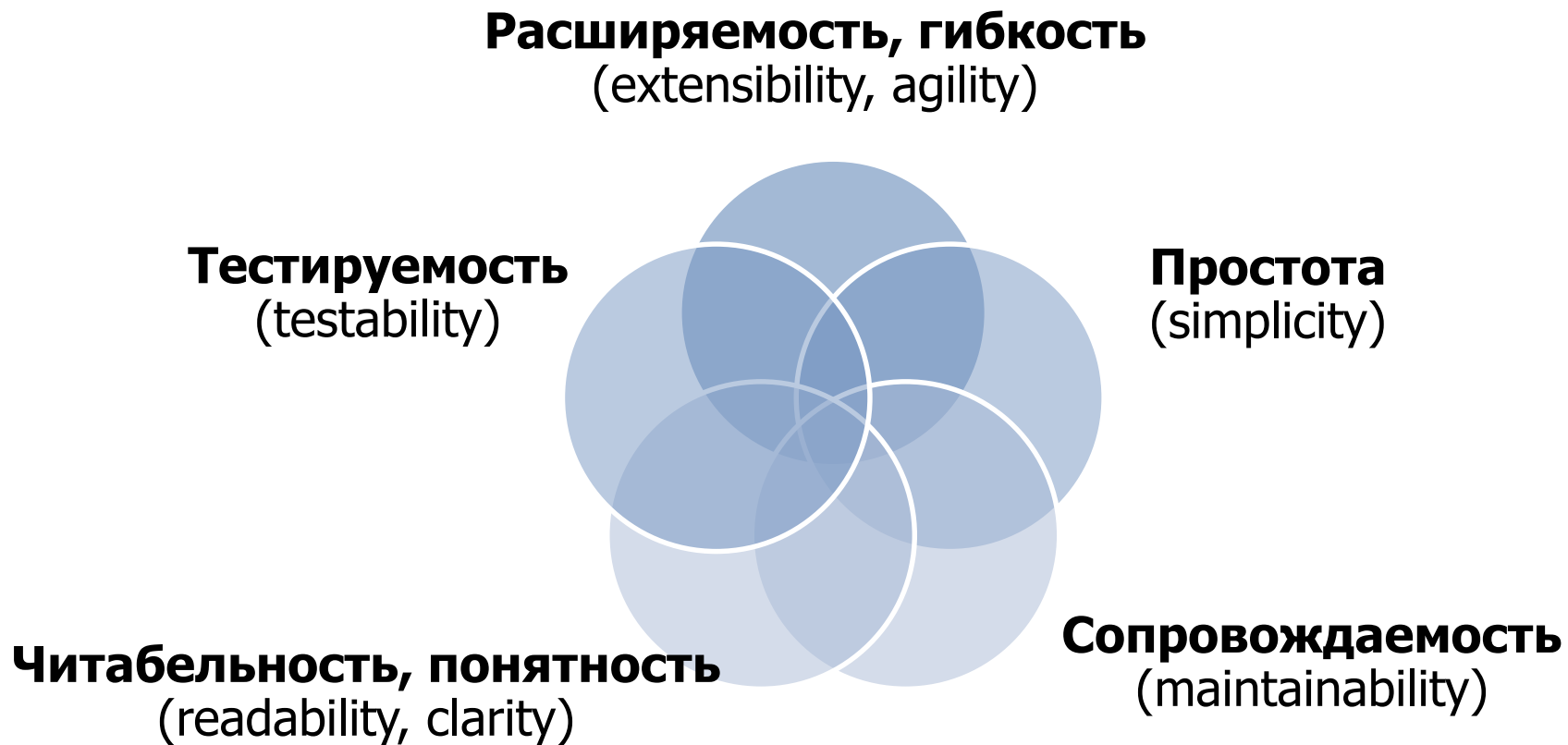
SOLID

д.т.н. Емельянов Виталий Александрович

✉: v.yemelyanov@gmail.com



Ценности качественного кода



Принципы SOLID

- ➔ **SOLID** – 5 принципов объектно-ориентированного программирования, описывающих архитектуру программного обеспечения.
- ➔ Все шаблоны проектирования (паттерны) основаны на этих принципах.



SRP – принцип единой ответственности

Смысл SRP: на каждый объект должна быть возложена одна единственная обязанность

Конкретный класс должен решать только конкретную задачу — ни больше, ни меньше.



SRP – принцип единой ответственности

- ➔ Каждый класс имеет свои обязанности в программе
- ➔ Если у класса есть несколько обязанностей, то у него появляется несколько причин для изменения
- ➔ Изменение одной обязанности может привести к тому, что класс перестанет справляться с другими.
- ➔ Такого рода связанность – причина хрупкого дизайна, который неожиданным образом разрушается при изменении



Хорошее разделение обязанностей выполняется только тогда, когда имеется полная картина того, как приложение должно работать.

SRP – принцип единой ответственности

С#

```
1 public class Employee
2 {
3     public int ID { get; set; }
4     public string FullName { get; set; }
5
6
7     //метод Add() добавляет в БД нового сотрудника
8     //emp – объект (сотрудник) для вставки
9
10    public bool Add(Employee emp)
11    {
12        //код для добавления сотрудника в таблицу БД
13        return true;
14    }
15
16    // метод для создания отчета по сотруднику
17    public void GenerateReport(Employee em)
18    {
19        //Генерация отчета по деятельности сотрудника
20    }
21 }
```

ПЛОХО: Класс **Employee** не соответствует принципу SRP

? ПОЧЕМУ

Класс несет 2 ответственности:

- ➔ добавление сотрудника в БД
- ➔ создание отчета.

Класс **Employee** не должен нести ответственность за отчетность, т.к. если вдруг надо будет предоставить отчет в формате Excel или изменить алгоритм создания отчета, то потребуется изменить класс **Employee**.

SRP – принцип единой ответственности

Согласно SRP, необходимо написать отдельный класс для ответственности по генерации отчетов:

C#

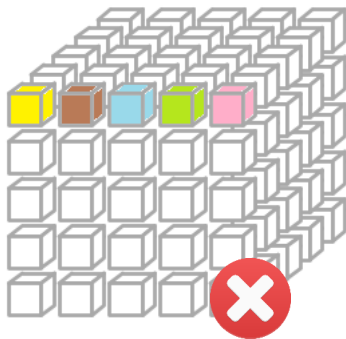
```
1 public class Employee
2 {
3     public int ID { get; set; }
4     public string FullName { get; set; }
5
6     public bool Add(Employee emp)
7     {
8         // Вставить данные сотрудника в таблицу БД
9         return true;
10    }
11 }
12
13 public class EmployeeReport
14 {
15     public void GenerateReport(Employee em)
16     {
17         // Генерация отчета по деятельности сотрудника
18     }
19 }
```

ОСР – принцип открытости/закрытости

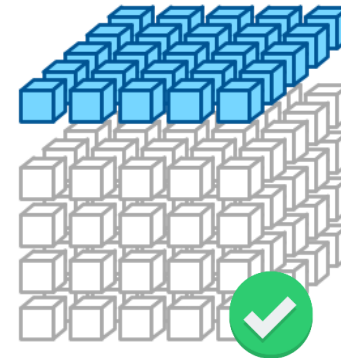
Смысл ОСР: Классы (модули) должны быть:

- ➔ **открыты для расширений** – модуль должен быть разработан так, чтобы новая функциональность могла быть добавлена только при создании новых требований.
- ➔ **закрыты для модификации** – означает, что мы уже разработали класс, и он прошел модульное тестирование. Мы не должны менять его, пока не найдем ошибки.

Модификации внутри:



Расширение:



ОСР – принцип открытости/закрытости

Принцип ОСР рекомендует проектировать систему так, чтобы в будущем **изменения можно было реализовать:**

- ✓ **путем добавления нового кода,**
- ✗ **а не изменением уже работающего кода.**



КАК ЭТО ВООБЩЕ ВОЗМОЖНО

ОСР – принцип открытости/закрытости

Принцип ОСР можно реализовать с помощью **интерфейсов** или **абстрактных классов**.

1. Интерфейсы фиксированы, но на их основе можно создать неограниченное множество различных поведений:

- ➔ *поведения* – это *производные классы от абстракций*.
- ➔ они могут манипулировать абстракциями.

2. Интерфейсы (абстрактные классы):

- ➔ могут быть **закрыты** для модификации – являются фиксированными;
- ➔ но их поведение можно расширять, создавая новые производные классы.

ОСР – принцип открытости/закрытости

C#

```
1 public class EmployeeReport
2 {
3     //свойство - тип отчета
4     public string TypeReport { get; set; }
5
6     //метод для отчета по сотруднику (объект em)
7     public void GenerateReport(Employee em)
8     {
9         if (TypeReport == "CSV")
10        {
11            // Генерация отчета в формате CSV
12        }
13
14        if (TypeReport == "PDF")
15        {
16            // Генерация отчета в формате PDF
17        }
18    }
19 }
```

ПЛОХО: Класс `EmployeeReport` не соответствует принципу ОСР



ПОЧЕМУ



Проблема в классе в том, что если надо внести новый тип отчета (например, для выгрузки в Excel), тогда надо добавить новое условие `if`. Т.е. необходимо изменить код уже работающего метода класса `EmployeeReport`.

ОСР – принцип открытости/закрытости

C#

```
1 public class IEmployeeReport
2 {
3     public virtual void GenerateReport(Employee em)
4     {
5         //Базовая реализация, которую нельзя модифицировать
6     }
7 }
8
9
10 public class EmployeeCSVReport : IEmployeeReport
11 {
12     public override void GenerateReport(Employee em)
13     {
14         //Генерация отчета в формате CSV
15     }
16 }
17
18 public class EmployeePDFReport : IEmployeeReport
19 {
20     public override void GenerateReport(Employee em)
21     {
22         //Генерация отчета в формате PDF
23     }
24 }
```

Класс **IEmployeeReport** закрыт от модификаций, но доступен для расширений.

Если надо добавить новый тип отчета, просто надо создать новый класс и унаследовать его от **IEmployeeReport**

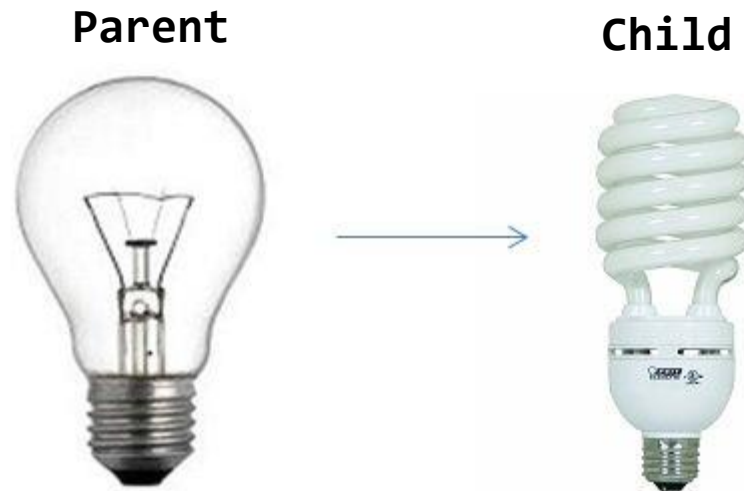
ОСР – принцип открытости/закрытости

Применение ОСР позволяет:

- ➡ создавать системы, которые будут сохранять стабильность при изменении требований;
- ➡ создать систему, которая будет существовать дольше первой версии.

LSP – принцип подстановки Барбары Лисков

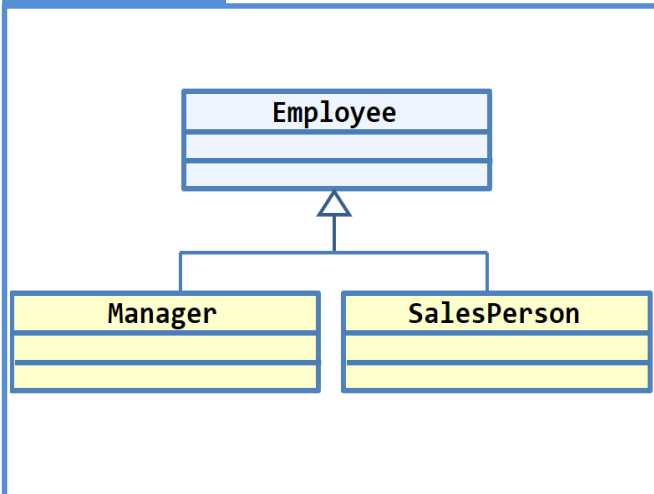
Смысл LSP: «вы должны иметь возможность использовать любой производный класс вместо родительского класса и вести себя с ним таким же образом без внесения изменений».



LSP – принцип подстановки Барбары Лисков

Согласно LSP, классы-наследники (**Manager** и **SalesPerson**) ведут себя также, как класс-родитель (**Employee**)

UML

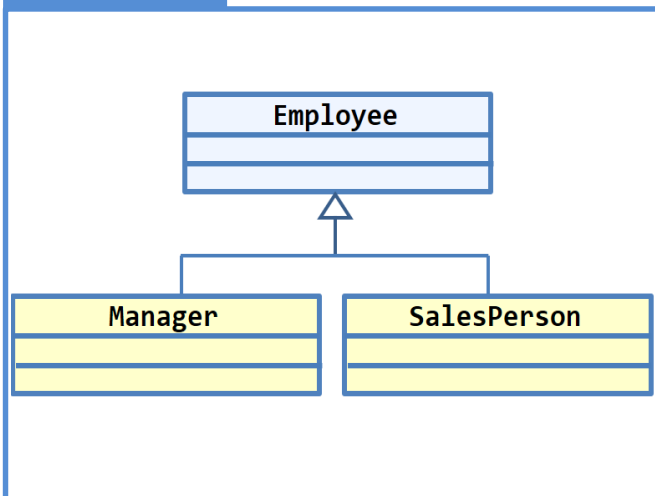


C#

```
1 public abstract class Employee
2 {
3     public virtual string GetWorkDetails(int id)
4     {
5         return "Base Work";
6     }
7
8     public virtual string GetEmployeeDetails(int id)
9     {
10        return "Base Employee";
11    }
12 }
```

LSP – принцип подстановки Барбары Лисков

UML



Плохой код. ПОЧЕМУ?

C#

```
13 public class Manager : Employee
14 {
15     public override string GetWorkDetails(int id)
16     {
17         return "Manager Work";
18     }
19
20     public override string GetEmployeeDetails(int id)
21     {
22         return "Manager Employee";
23     }
24 }
25
26 public class SalesPerson : Employee
27 {
28     public override string GetWorkDetails(int id)
29     {
30         throw new NotImplementedException();
31     }
32
33     public override string GetEmployeeDetails(int id)
34     {
35         return "SalesPerson Employee";
36     }
37 }
```


LSP – принцип подстановки Барбары Лисков

C#

```
38 static void Main(string[] args)
39 {
40     List<Employee> list = new List<Employee>();
41
42     list.Add(new Manager());
43     list.Add(new SalesPerson());
44
45     foreach (Employee emp in list)
46     {
47         emp.GetEmployeeDetails(985);
48     }
49 }
```

ПРОБЛЕМА:

для `SalesPerson` невозможно вернуть информацию о работе, поэтому получаем необработанное исключение, что нарушает принцип LSP.

LSP – принцип подстановки Барбары Лисков

Для решения этой проблемы в C# необходимо просто разбить функционал на два интерфейса `Iwork` и `IEmployee`:

C#

```
1
2 public interface IEmployee
3 {
4     string GetEmployeeDetails(int Id);
5 }
6
7 public interface IWork
8 {
9     string GetWorkDetails(int Id);
10 }
11
12
13 public class SalesPerson : IEmployee
14 {
15     public string GetEmployeeDetails(int Id)
16     {
17         return "SalesPerson Employee";
18     }
19 }
```

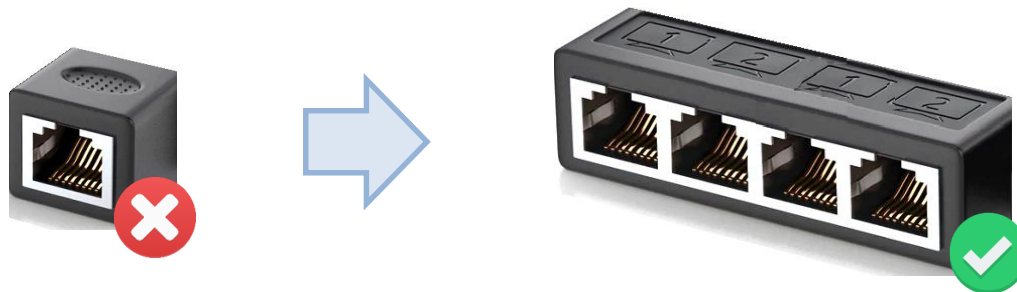
```
20
21 public class Manager : IWork, IEmployee
22 {
23     public string GetWorkDetails(int Id)
24     {
25         return "Manager Work";
26     }
27
28     public string GetEmployeeDetails(int Id)
29     {
30         return "Manager Employee";
31     }
32 }
33
34
35
36
37
38
```

Теперь `SalesPerson` требует реализации только `IEmployee`, а не `IWork`. При таком подходе будет поддерживаться принцип LSP

ISP – принцип разделения интерфейсов

Смысл ISP: много специализированных интерфейсов лучше, чем один универсальный

- ➔ Соблюдение этого принципа необходимо для того, чтобы классы-клиенты использующий/реализующий интерфейс знали только о тех методах, которые они используют, что ведёт к уменьшению количества неиспользуемого кода.



ISP – принцип разделения интерфейсов

Пусть есть одна база данных (БД) для хранения данных всех типов сотрудников (типы сотрудников: *Junior* и *Senior*)

- ➔ Необходимо реализовать возможность добавления данных о сотрудниках в БД.
- ➔ Возможный вариант интерфейса для сохранения данных по сотрудникам:

C#

```
1
2 public interface IEmployee
3 {
4     bool AddDetailsEmployee();
5 }
6
```

ISP – принцип разделения интерфейсов

Допустим все классы **Employee** наследуют интерфейс **IEmployee** для сохранения данных в БД. Теперь предположим, что в компании однажды возникла необходимость читать данные только для сотрудников в должности **Senior**.

➔ Что делать?

➔ Просто добавить один метод в интерфейс?

ПЛОХО: Интерфейс **IEmployee** не соответствует принципу ISP

С#

```
1
2 public interface IEmployee
3 {
4     bool AddDetailsEmployee();
5     bool ShowDetailsEmployee(int id);
6 }
7
```



ПОЧЕМУ



Потому что мы что-то ломаем. Мы вынуждаем объекты **JuniorEmployee** показывать свои данные из базы данных.

ISP – принцип разделения интерфейсов

Согласно ISP, решение заключается в том, чтобы передать новую ответственность другому интерфейсу:

C#

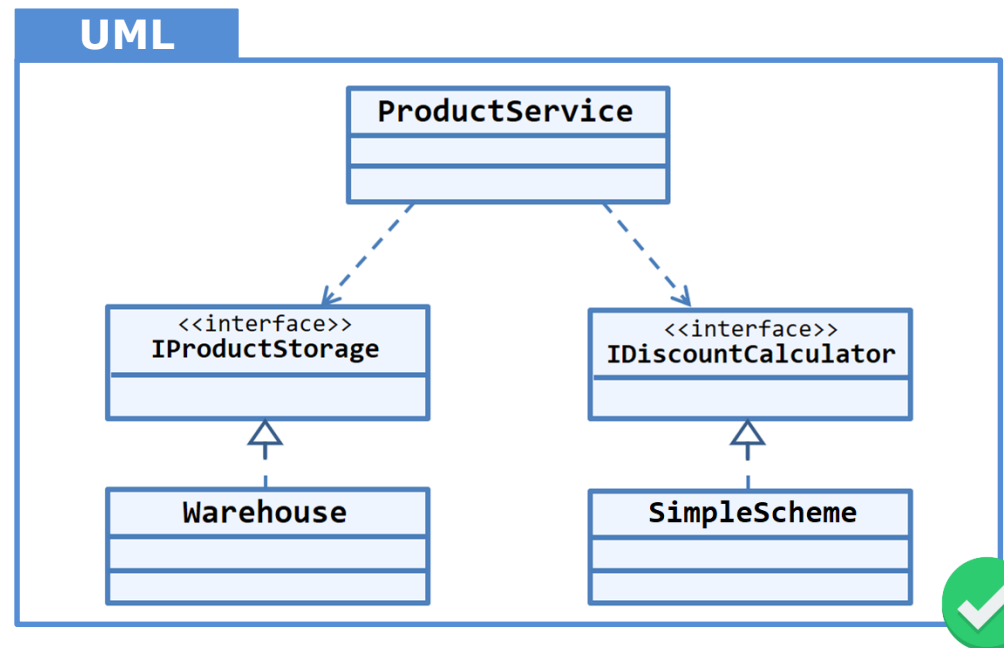
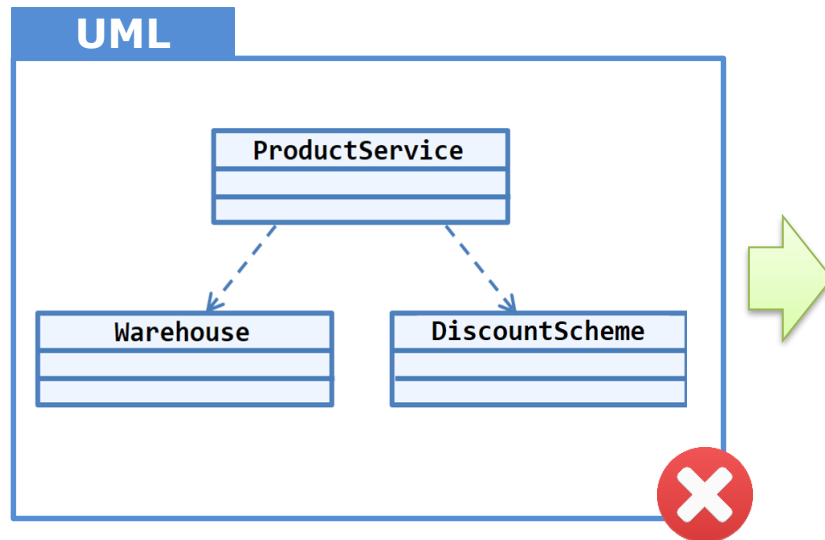
```
1 public interface IOperationAdd
2 {
3     bool AddDetailsEmployee();
4 }
5
6 public interface IOperationGet
7 {
8     bool ShowDetailsEmployee(int id);
9 }
```

РЕЗУЛЬТАТ: теперь, класс **JuniorEmployee** будет реализовывать только интерфейс **IOperationAdd**, а **SeniorEmployee** оба интерфейса. Таким образом обеспечивается разделение интерфейсов.

DIP – принцип инверсии зависимостей

Смысл DIP: «зависеть от абстракций, а не от деталей»

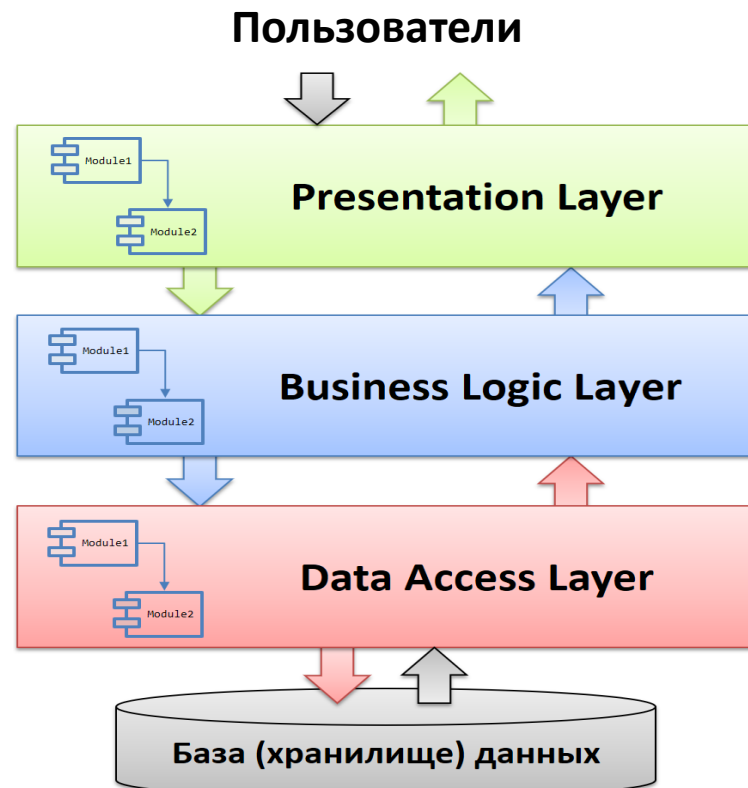
1. Модули верхних уровней не должны зависеть от модулей нижних уровней. Модули обоих уровней должны зависеть от абстракций.
2. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.



DIP – принцип инверсии зависимостей

Многослойная архитектура ПО:

- ➔ В любой хорошо структурированной объектно-ориентированной архитектуре можно выделить ясно очерченные слои архитектуры ПО.



DIP – принцип инверсии зависимостей

- ➔ **Presentation Layer** (уровень представления) – уровень, с которым непосредственно взаимодействует пользователь. Этот уровень включает компоненты пользовательского интерфейса, механизм получения ввода от пользователя и т.д.
- ➔ **Business Logic Layer** (уровень бизнес-логики): содержит набор компонентов, которые отвечают за обработку полученных от уровня представлений данных, реализует всю необходимую логику приложения, все вычисления, взаимодействует с базой данных и передает уровню представления результат обработки.
- ➔ **Data Access Layer** (уровень доступа к данным): хранит модели, описывающие используемые сущности, также здесь размещаются специфичные классы для работы с разными технологиями доступа к данным, например, класс контекста данных Entity Framework. Здесь также хранятся репозитории, через которые уровень бизнес-логики взаимодействует с базой данных.

DIP – принцип инверсии зависимостей

1. Классы (модули) высокого уровня реализуют бизнес-правила или логику в системе (приложении).
2. Низкоуровневые классы (модули) занимаются более подробными операциями, другими словами, они могут заниматься записью информации в базу данных или передачей сообщений в ОС и т.п.



В ЧЕМ ПРОБЛЕМА:

ЕСЛИ высокоуровневый класс имеет зависимость от дизайна и реализации другого класса, **ВОЗНИКАЕТ РИСК ТОГО, ЧТО ИЗМЕНЕНИЯ В ОДНОМ КЛАССЕ НАРУШАТ ДРУГОЙ КЛАСС.**

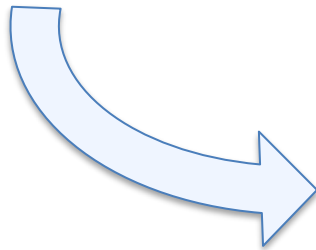
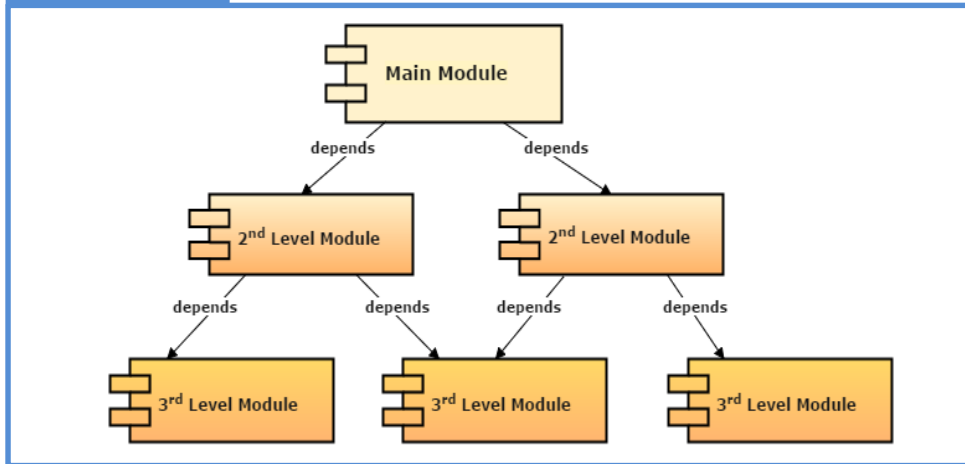


РЕШЕНИЕ:

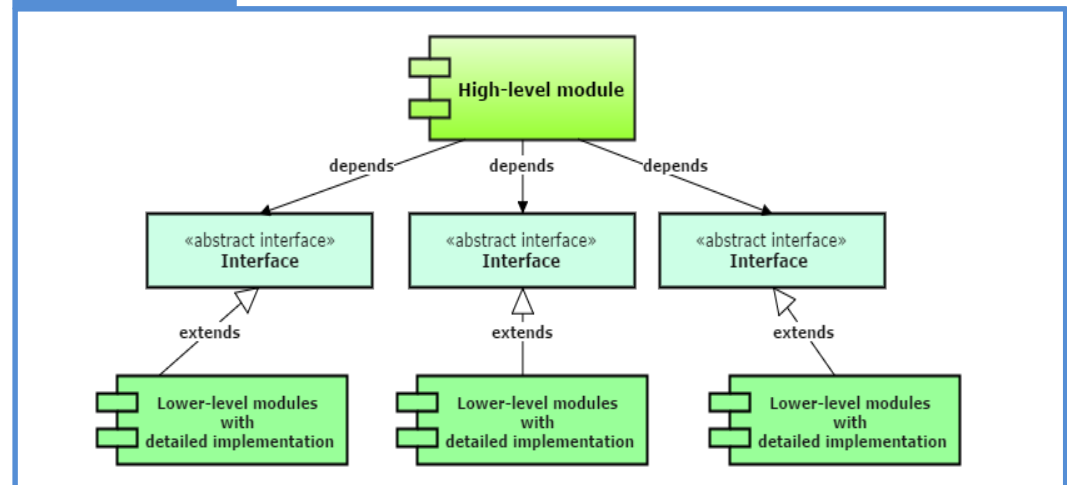
Держать высокоуровневые и низкоуровневые классы слабо связанными. Для этого необходимо сделать их зависимыми от абстракций, а не друг от друга.

DIP – принцип инверсии зависимостей

UML

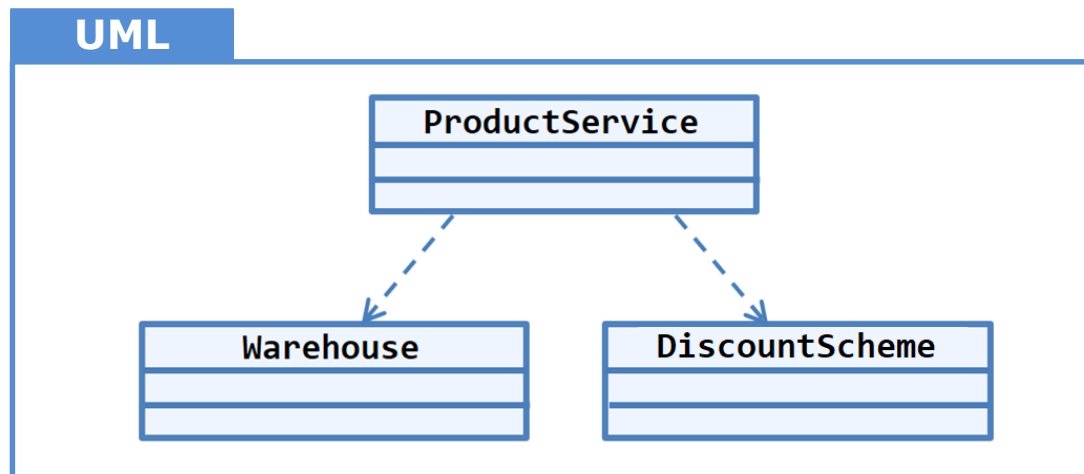


UML



DIP – принцип инверсии зависимостей

ЗАДАЧА: Требуется составить программу для расчета суммарной скидки товара, который хранится на складе, по определенной карте скидок.



1. **ProductService** – класс с методом для расчета суммарной скидки товара
2. Класс **ProductService** зависит от реализации классов:
 - ➡ **Warehouse** – склад, на котором хранится товар
 - ➡ **DiscountScheme** – схема начисления скидки

DIP – принцип инверсии зависимостей

C#

```
1 public class Product
2 {
3     public double Cost { get; set; }
4     public String Name { get; set; }
5     public uint Count { get; set; }
6 }
7
8
9 public class Warehouse
10 {
11     public IEnumerable<Product> GetProducts()
12     {
13         return new List<Product> { new Product {Cost=140, Name = "Tyres", Count=1000},
14                                     new Product {Cost=160, Name = "Disks", Count=200},
15                                     new Product {Cost=100, Name = "Tools", Count=100}
16                                     };
17     }
18 }
```

DIP – принцип инверсии зависимостей

C#

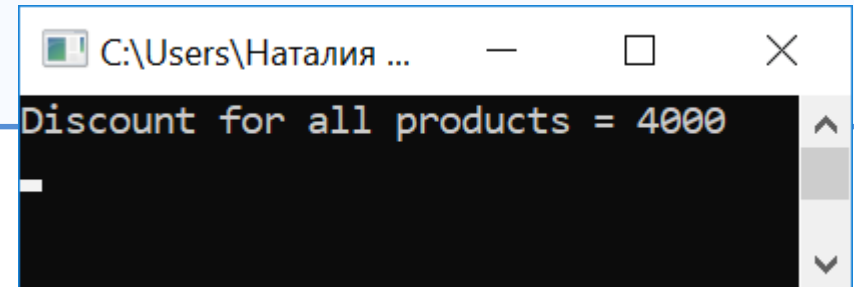
```
19
20 public class DiscountScheme
21 {
22     public double GetDiscount(Product p)
23     {
24         switch(p.Name)
25         {
26             case "Tyres": return 0.01;
27             case "Disks": return 0.05;
28             case "Tools": return 0.1;
29             default: return 0;
30         }
31     }
32 }
```

```
38
39 public class ProductService
40 {
41     public double GetAllDiscount()
42     {
43         double sum = 0;
44
45         Warehouse wh = new Warehouse();
46
47         IEnumerable<Product> products = wh.GetProducts();
48
49         DiscountScheme ds = new DiscountScheme();
50
51         foreach (var p in products)
52             sum += p.Cost * p.Count * ds.GetDiscount(p);
53
54         return sum;
55     }
56 }
```

DIP – принцип инверсии зависимостей

C#

```
57 class Program
58 {
59     static void Main(string[] args)
60     {
61         ProductService ps = new ProductService();
62         Console.WriteLine("Discount for all products = " + ps.GetAllDiscount());
63
64         Console.ReadKey();
65     }
66 }
```

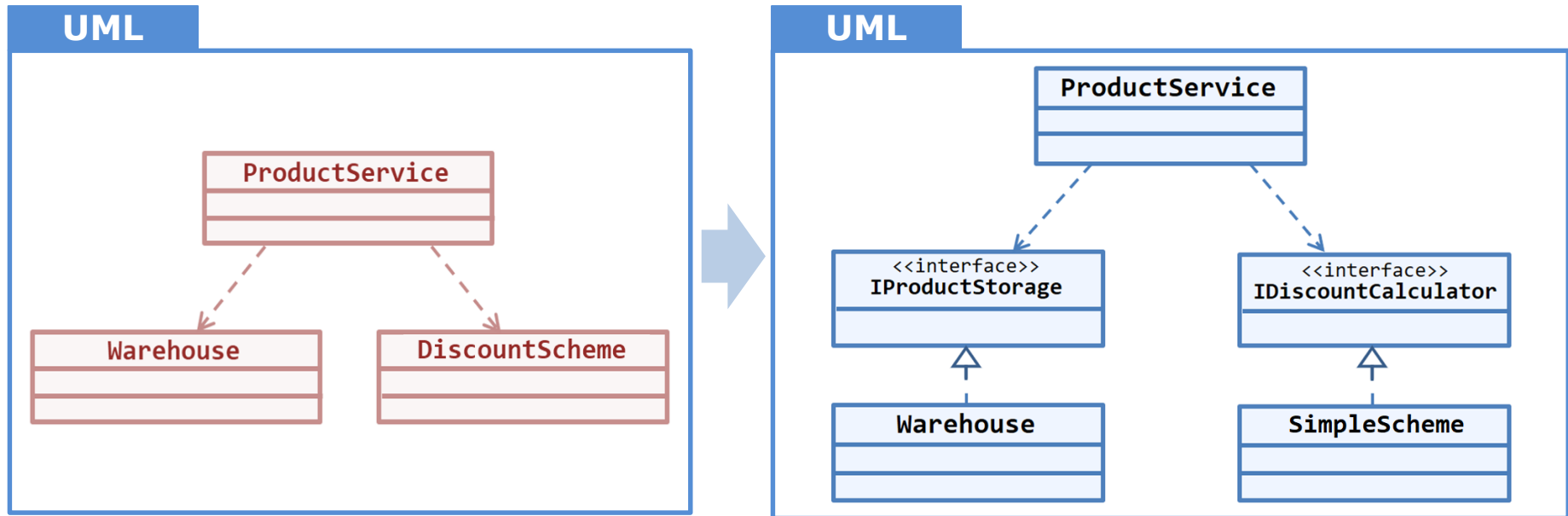
A screenshot of a Windows console window. The title bar shows the path 'C:\Users\Наталья ...'. The console output displays the text 'Discount for all products = 4000' on a single line. The cursor is positioned at the end of the line.

ПРОБЛЕМЫ:

1. По факту мы не можем без изменения `ProductService` рассчитать скидку на товары, которые могут быть не только на складе `Warehouse`.
2. Так же нет возможности подсчитать скидку по другой карте скидок (с другим `Disctount Scheme`).

DIP – принцип инверсии зависимостей

Применяем DIP:



Стрелки на диаграмме классов от **Warehouse** и **SimpleScheme** поменяли направление (инверсия зависимости). Теперь от **Warehouse** и **SimpleScheme** (**DiscountScheme**) ничего не зависит. Наоборот - они зависят от абстракций (интерфейсов).

DIP – принцип инверсии зависимостей

C#

```
1
2 public interface IProductStorage
3 {
4     IEnumerable<Product> GetProducts();
5 }
6
7 public interface IDiscountCalculator
8 {
9     double GetDiscount(Product products);
10 }
11
12 public class Product
13 {
14     public double Cost { get; set; }
15     public String Name { get; set; }
16     public uint Count { get; set; }
17 }
18
```

DIP – принцип инверсии зависимостей

C#

```
19 public class Warehouse : IProductStorage
20 {
21     public IEnumerable<Product> GetProducts()
22     {
23         return new List<Product> { new Product {Cost=140, Name="Tyres", Count= 1000},
24                                     new Product {Cost=160, Name="Disks", Count= 200},
25                                     new Product {Cost=100, Name="Tools", Count= 100}};
26     }
27 }
28
29 public class SimpleScheme : IDiscountCalculator
30 {
31     public double GetDiscount(Product p)
32     {
33         switch (p.Name)
34         {
35             case "Tyres": return 0.01;
36             case "Disks": return 0.05;
37             case "Tools": return 0.1;
38             default: return 0;
39         }
40     }
41 }
```

DIP – принцип инверсии зависимостей

C#

```
42 public class ProductService
43 {
44     public double GetAllDiscount(IProductStorage storage,
45                                 IDiscountCalculator discountCalculator)
46     {
47         double sum = 0;
48         foreach (var p in storage.GetProducts())
49             sum += p.Cost * p.Count * discountCalculator.GetDiscount(p);
50
51         return sum;
52     }
53 }
54
55 class Program
56 {
57     static void Main(string[] args)
58     {
59         ProductService ps = new ProductService();
60         Console.WriteLine("Discount for all products = " +
61                           ps.GetAllDiscount(new Warehouse(), new SimpleScheme()));
62         Console.ReadKey();
63     }
64 }
```

DIP – принцип инверсии зависимостей

Проблемы архитектуры ПО, которые устраняются с применением DIP:

- ➔ **Жесткость:** изменение одного модуля ведет к изменению других модулей
- ➔ **Хрупкость:** изменения приводят к неконтролируемым ошибкам в других частях программы
- ➔ **Неподвижность:** модуль сложно отделить от остальной части приложения для повторного использования

SOLID упрощенно:

Single Responsibility

– *делай модули меньше (1 ответственность)*

Open/Closed

– *делай модули расширяемыми*

Liskov Substitution

– *наследники ведут себя так же, как родители*

Interface Segregation

– *дели слишком сложные интерфейсы*

Dependency Inversion

– *используй интерфейсы*