В.В. Подбельский

«Программирование на C#»

Preprocessor Directives
Custom Attributes
SOLID

Препроцессорные директивы
Пользовательские атрибуты
SOLID

1. Общие правила

Нет точки с запятой

↓

```
#define PremiumVersion                  // OK
```

Пробелы в начале

↓

```
   #define BudgetVersion                // OK
#          define MediumVersion         // OK
```

↑

Пробелы между

Многострочные комментарии запрещены

↓

```
#define PremiumVersion        /* так нельзя */
```

Однострочные - разрешены

↓

```
#define BudgetVersion     // урезанная версия
```

2. Препроцессорные директивы (1)

Директива	Назначение
#define identifier	Определяет препроцессорный символ
#undef identifier	Отменяет препроцессорный символ
#if выражение	Если выражение истинно, компилирует последующий раздел
#elif выражение	Если выражение истинно, компилирует последующий раздел
#else	Если предшествующие выражения в #if и #elif ложны, компилирует последующий раздел
#endif	Обозначает конец конструкции #if

3. Препроцессорные директивы (2)

Директива	Назначение
#region name	Marks the beginning of a region of code; has no compilation effect
#endregion name	Marks the end of a region of code; has no compilation effect
#warning message	Displays a compile-time warning message
#error message	Displays a compile-time error message
#line indicator	Changes the line numbers displayed in compiler messages
#pragma text	Specifies information about the program context

4. Директивы #define и #undef

```
#define PremiumVersion
#define EconomyVersion
...
#undef PremiumVersion
```

Ошибки в применении директив:

```
using System;                   // First line of C# code
#define PremiumVersion          // Error
namespace Eagle
{
#define PremiumVersion          // Error
...
```

5. Условная компиляция. Синтаксис

```
#if   условие
#else
#elif  условие
#endif
```

Операции в выражениях условий:

== сравнение на равенство
!= сравнение на неравенство
! отрицание
|| дизъюнкция
&& конъюнкция

Операнды выражений условий:

Символы компиляции
Лексемы **true** и **false**
Выражения в круглых скобках

6. Условная компиляция. Пример

выражение

↓

```
#if !DemoVersion
...
#endif
```

выражение

↓

```
#if (LeftHanded && OemVersion) || FullVersion
...
#endif
#if true // следующий раздел всегда скомпилируется
...
#endif
```

7. Конструкции #if и #else

```
#if Condition
    CodeSection
#endif
```

```
#if Condition
    CodeSection1
#else
    CodeSection2
#endif
```

8. Конструкция #elif

```
#if Cond1
    CodeSection1
#elif Cond2
    CodeSection2
#elif Cond3
    CodeSection3
#endif
```

```
#if Cond1
    CodeSection1
#elif Cond2
    CodeSection2
    ...
#else
    CodeSectionE
#endif
```

9. Диагностические Директивы

#warning ***Message***
#error ***Message***

Пример кода:

```
#define RightHanded
#define LeftHanded

#if RightHanded && LeftHanded
#error Can't build for both RightHanded and LeftHanded
#endif

#warning Remember to come back and clean up this code!
```

10. Директивы Line Number

Синтаксис директив:

```
#line integer          // Sets line number of next line
                       // to value of integer
#line "filename"       // Sets the apparent filename
#line default          // Restores real line number and filename
#line hidden           // Hides the following code
                       // from stepping debugger
#line                  // Stops hiding from debugger
```

11. Директивы Line Number (2)

```
#line 226
x = y + z; // компилятор рассматривает это как 226 строку
...
#line 330 "SourceFile.cs"    // изменяем сообщаемый номер
                             // строки и имя файла
var1 = var2 + var3;
...
#line default // восстанавливаем исходные (истинные)
                //значения
```

12. Директивы Region

```
#region Constructors
MyClass()
{
...
}
MyClass(string s)
{
...
}
#endregion
```

```
static void Main( )
{
          ...
#region first

#region second
     ...
#endregion

#region third
     ...
#endregion

#endregion
}
```

13. Директивы #pragma warning

Warning messages to turn off

↓

#pragma warning disable 618, 414

... Messages for the listed warnings are off in this section of code.

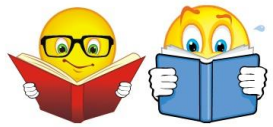
#pragma warning restore 618

#pragma warning disable

... All warning messages are turned off in this section of code.

#pragma warning restore

... All warning messages are turned back on in this section of code

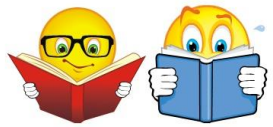


Метаданные и отражение

Метаданные – это данные о программах и используемых в них типах, хранимые в скомпилированных сборках.

Отражение – механизм, позволяющий программе во время своего исполнения считывать метаданные сборок (чужих и своих).

Классы по работе с отражением содержатся в пространстве имен **System.Reflection**.

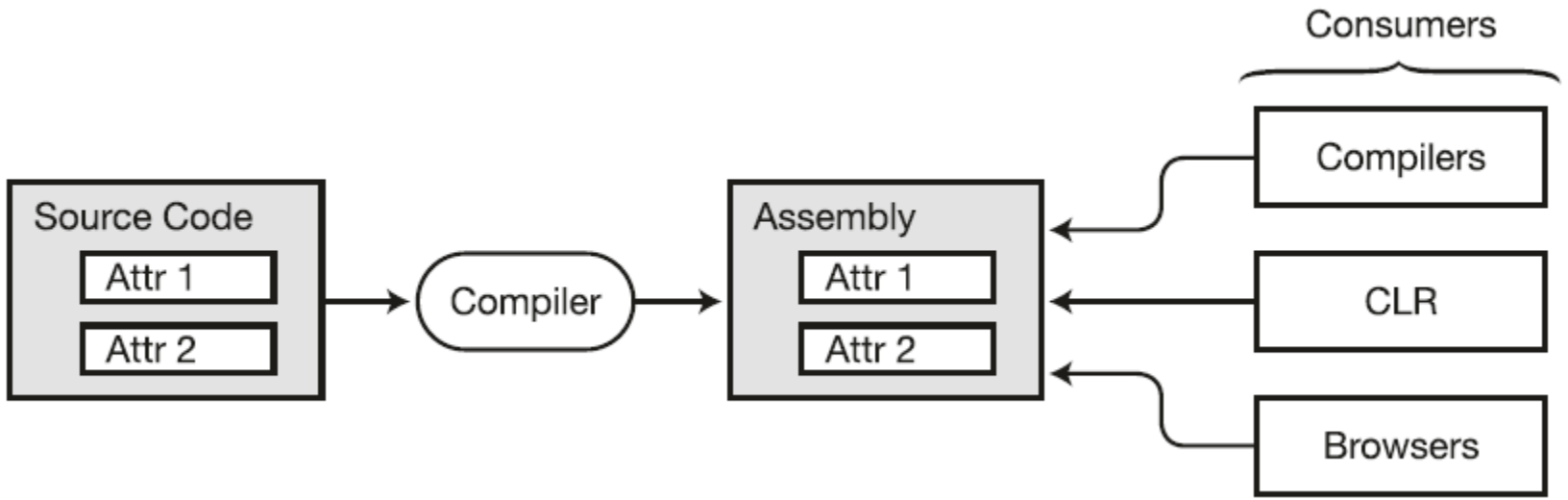


Что такое атрибут (attribute)?

Атрибут – это языковая конструкция, позволяющая добавлять метаданные к программной сборке.

Шаги:

1) Применение атрибута (библиотечного или самописного) к коду программы.
2) Компилятор добавляет метаданные об атрибутах в сборку
3) Программа во время выполнения может анализировать метаданные.



Использование нескольких атрибутов

Два варианта:

1. Стековый

```
[ Serializable ] // Stacked
[ MyAttribute("Simple class", "Version 3.57") ]
```

2. Через запятую

```
[ MyAttribute("Simple class", "Version 3.57"), Serializable ] // Comma separated
```

Применение к полям и методам

Применение к полям

```
[MyAttribute("Holds a value", "Version 3.2")]
public int MyField;
```

Применение к методам

```
[Obsolete]
[MyAttribute("Prints out a message.", "Version 3.6")]
public void PrintOut()
{
...
}
```

Явное указание цели атрибута

	event	**field**
	method	**param**
	property	**return**
	type	**typevar**
Глобальные:	**assembly**	**module**

```
[method: MyAttribute("Prints out a message.", "Version 3.6")]
[return: MyAttribute("This value represents ...", "Version 2.3")]
public long ReturnSetting()
{
...
}
```

Глобальные атрибуты

```
// Содержимое файла AssemblyInfo.cs

[assembly: AssemblyTitle("SuperWidget")]
[assembly: AssemblyDescription("Implements the SuperWidget product.")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("McArthur Widgets, Inc.")]
[assembly: AssemblyProduct("Super Widget Deluxe")]
[assembly: AssemblyCopyright("Copyright © McArthur Widgets 2012")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

Пользовательские атрибуты

имя атрибута

↓

```
public sealed class MyAttributeAttribute : System.Attribute
{
...
```

↑ суффикс

↑ базовый класс

Типичные члены класса-атрибута:

- поля;
- свойства;
- конструкторы.

Создание конструкторов

```
public MyAttributeAttribute(string desc, string ver)
{
        Description = desc;
        VersionNumber = ver;
}
```

Допускается:

- перегрузка конструкторов;
- пустой конструктор добавляется автоматически, если не указано ни одного конструктора.

Применение конструкторов

```
[MyAttribute("Holds a value")]        // конструктор с одним параметром
public int MyField;
[MyAttribute("Version 1.3", "Galen Daniel")]   // конструктор с двумя параметрами
public void MyMethod()
{ ...
```

Использование конструктора без параметров:

```
[MyAttr]
class SomeClass ...

[MyAttr()]
class OtherClass ...
```

Важно: аргументы конструкторов должны быть известны в момент компиляции (константы)!

Императивный vs. декларативный стили

```
MyClass mc = new MyClass("Hello", 15);
```

Imperative Statement

```
[MyAttribute("Holds a value")]
```

Declarative Statement

Параметры конструкторов: позиционные и именованные

позиционный ↓ именованный ↓ равно ↑ именованный ↓ равно ↑

```
[MyAttribute("An excellent class", Reviewer="Amy McArthur", Ver="0.7.15.33")]
```

```
public sealed class MyAttributeAttribute : System.Attribute {
        public string Description;
        public string Ver;
        public string Reviewer;
        public MyAttributeAttribute(string desc)  { // единственный формальный параметр
                Description = desc;
        }
}
```

три аргумента ↓

```
[MyAttribute("An excellent class", Reviewer="Amy McArthur", Ver="7.15.33")]
class MyClass {
...
}
```

Ограничение использования атрибутов (AttributeUsage)

только для методов

↓

```
[ AttributeUsage( AttributeTargets.Method ) ]
public sealed class MyAttributeAttribute : System.Attribute
{ ...
```

Имя	Значение	Значение по умолчанию
ValidOn	Хранит список типов целей к которым может применяться атрибут. Первый параметр конструктора должен быть значением перечислимого типа AttributeTargets.	
Inherited	Булево значение, указывающее может ли атрибут наследоваться производными классами декорированного типа.	true
AllowMultiple	Булево значение, указывающее можно ли к цели применять одновременно несколько экземпляров атрибута текущего типа.	false

Конструктор AttributeUsage

```
[ AttributeUsage( AttributeTargets.Method | AttributeTargets.Constructor ) ]
public sealed class MyAttributeAttribute : System.Attribute
{ ...
```

Члены перечисления *AttributeTargets:*

All	Assembly	Class	Constructor
Delegate	Enum	Event	Field
GenericParameter	Interface	Method	Module
Parameter	Property	ReturnValue	Struct

Пример использования AttributeUsage

```
[ AttributeUsage( AttributeTargets.Class,       // Required, positional
                  Inherited = true,             // Optional, named
                  AllowMultiple = false ) ]     // Optional, named
public sealed class MyAttributeAttribute : System.Attribute
{ ...
```

Рекомендации для пользовательских атрибутов

• Класс атрибута должен представлять некоторое **состояние** цели, к которой он применяется.

• Если для атрибута требуются поля, добавьте **параметрический конструктор** для сбора значений (в него же добавьте опциональные параметры с умалчиваемыми значениями, если требуется).

• **Не реализовывайте** публичные методы или другие **функциональные члены** (за исключением свойств).

• В целях безопасности объявите класс атрибута опечатанным (**sealed**).

• Используйте атрибут **AttributeUsage** при объявлении собственного атрибута, чтобы явно указать множество целей вашего атрибута.

Пример пользовательского атрибута

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class ReviewCommentAttribute : System.Attribute
{
    public string Description { get; set; }
    public string VersionNumber { get; set; }
    public string ReviewerID { get; set; }

    public ReviewCommentAttribute(string desc, string ver)
    {
        Description = desc;
        VersionNumber = ver;
    }
}
```

Доступ к атрибуту. Метод IsDefined

```
[ReviewComment("Check it out", "2.4")]
class MyClass { }

public static void Main()
{
    MyClass mc = new MyClass(); // создаем объект
    Type t = mc.GetType(); // получаем тип объекта
    bool isDefined = // проверяем тип на наличие атрибута
    t.IsDefined(typeof(ReviewCommentAttribute), false);
    if (isDefined)
        Console.WriteLine($"ReviewComment применен к типу {t.Name}");
}
```

Результат работы программы:
ReviewComment применен к типу AttrDemo.MyClass

Использование метода GetCustomAttributes.

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class MyAttributeAttribute : Attribute
{
    public string Description { get; set; }
    public string VersionNumber { get; set; }
    public string ReviewerID { get; set; }
    public MyAttributeAttribute(string desc, string ver)
    {
        Description = desc;
        VersionNumber = ver;
    }
}

[MyAttribute("Check it out", "2.4")]
class MyClass
{
}
```

Использование метода GetCustomAttributes. Код Main

```
public static void Main() {
    Type t = typeof(MyClass);
    object[] AttArr = t.GetCustomAttributes(false);   // без наследования
    foreach (Attribute a in AttArr)
    {
        MyAttributeAttribute attr = a as MyAttributeAttribute;
        if (null != attr)
        {
          Console.WriteLine($"Description : { attr.Description }");
          Console.WriteLine($"Version Number : { attr.VersionNumber }");
          Console.WriteLine($"Reviewer ID : { attr.ReviewerID }");
        }
    }
}
```

Результат работы программы:

Description : Check it out
Version Number : 2.4
Reviewer ID :

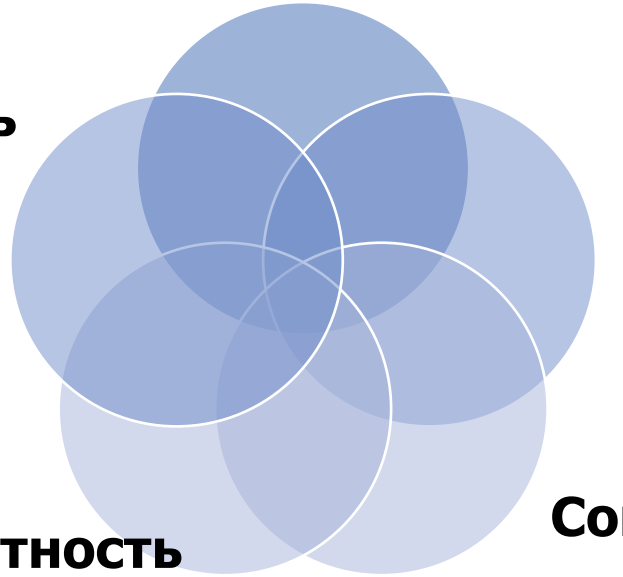
Принципы SOLID

- **SOLID** – принципы программирования, описывающие правильное применение объектно-ориентированного подхода к разработке ПО.
- Все шаблоны проектирования (паттерны) основаны на этих принципах.

Ценности качественного кода

Расширяемость, гибкость
(extensibility, agility)

Тестируемость
(testability)



Простота
(simplicity)

Читабельность, понятность
(readability, clarity)

Сопровождаемость
(maintainability)

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.

Single Responsibility Principle

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

```
public class Employee
{
    public int ID { get; set; }
    public string FullName { get; set; }


    //метод Add() добавляет в БД нового сотрудника
    //emp – объект (сотрудник) для вставки

    public bool Add(Employee emp)
    {
        //код для добавления сотрудника в таблицу БД
        return true;
    }

    // метод для создания отчета по сотруднику
    public void GenerateReport(Employee em)
    {
        //Генерация отчета по деятельности сотрудника
    }
}
```

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:

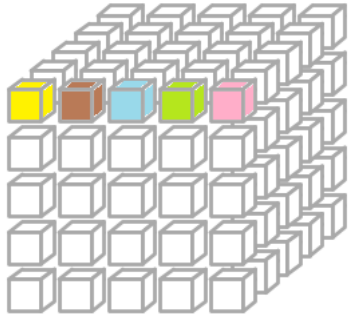
```
public class Employee
{
    public int ID { get; set; }
    public string FullName { get; set; }

    public bool Add(Employee emp)
    {
        // Вставить данные сотрудника в таблицу БД
        return true;
    }
}

public class EmployeeReport
{
    public void GenerateReport(Employee em)
    {
        // Генерация отчета по деятельности сотрудника
    }
}
```

OCP – принцип открытости/закрытости

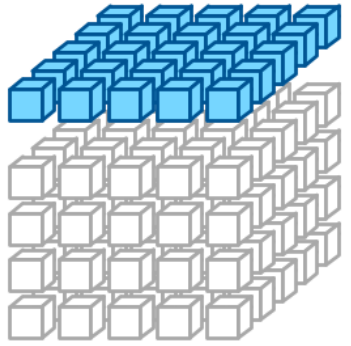
Смысл OCP: Классы (модули) должны быть:

- **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:



Open–Closed Principle

OCP – принцип открытости/закрытости

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

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

КАК ЭТО ВОЗМОЖНО

OCP – принцип открытости/закрытости

Принцип OCP 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.

OCP – принцип открытости/закрытости

```
public class EmployeeReport
{
    //свойство - тип отчета
    public string TypeReport { get; set; }

    //метод для отчета по сотруднику (объект em)
    public void GenerateReport(Employee em)
    {
        if (TypeReport == "CSV")
        {
            // Генерация отчета в формате CSV
        }

        if (TypeReport == "PDF")
        {
            // Генерация отчета в формате PDF
        }
    }
}
```

ПЛОХО: **Класс `EmployeeReport` не соответствует принципу OCP**

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`.

OCP – принцип открытости/закрытости

```
public interface IEmployeeReport
{
    public void GenerateReport(Employee em)
    {
      //Базовая реализация, которую нельзя модифицировать
    }
}


public class EmployeeCSVReport : IEmployeeReport
{
    public void GenerateReport(Employee em)
    {
      //Генерация отчета в формате CSV
    }
}

public class EmployeePDFReport : IEmployeeReport
{
    public void GenerateReport(Employee em)
    {
      //Генерация отчета в формате PDF
    }
}
```

IEmployeeReport закрыт от модификаций, но доступен для расширений.

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

OCP – принцип открытости/закрытости

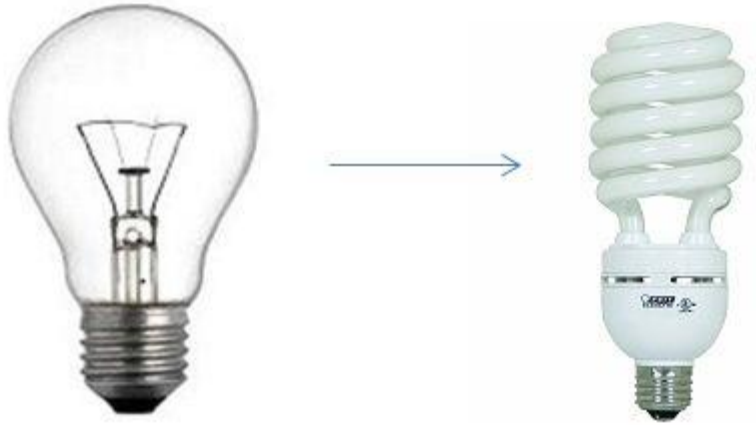
Применение OCP позволяет:

- создавать системы, которые будет 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».**

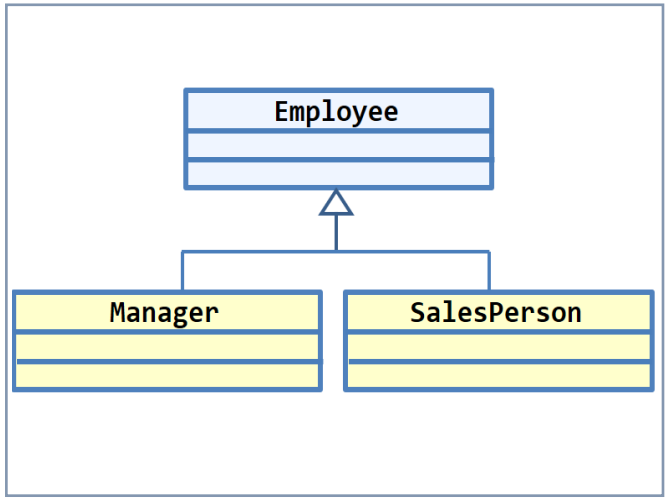
родитель **потомок**



Liskov Substitution Principle

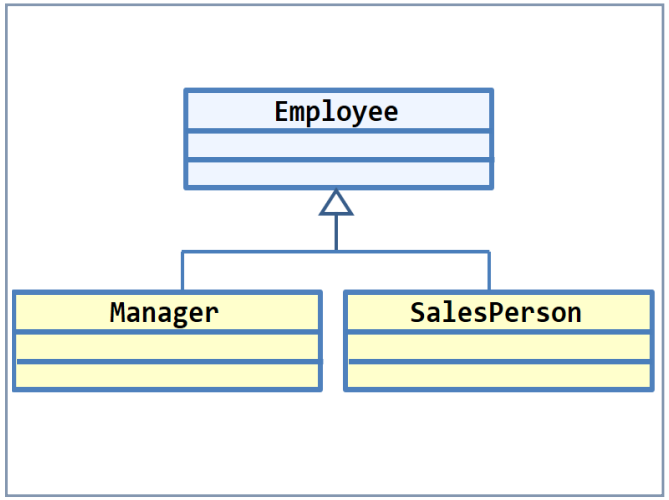
LSP – принцип подстановки Барбары Лисков

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




```
public class Employee
{
    public virtual string GetWorkDetails(int id)
    {
        return "Base Work";
    }

    public virtual string GetEmployeeDetails(int id)
    {
        return "Base Employee";
    }
}
```

LSP – принцип подстановки Барбары Лисков



Плохой код. ПОЧЕМУ?

```
public class Manager : Employee
{
    public override string GetWorkDetails(int id)
    {
        return "Manager Work";
    }

    public override string GetEmployeeDetails(int id)
    {
        return "Manager Employee";
    }
}

public class SalesPerson : Employee
{
    public override string GetWorkDetails(int id)
    {
        throw new NotImplementedException();
    }

    public override string GetEmployeeDetails(int id)
    {
        return "SalesPerson Employee";
    }
}
```

LSP – принцип подстановки Барбары Лисков

```
static void Main(string[] args)
{
    List<Employee> list = new List<Employee>();

    list.Add(new Manager());
    list.Add(new SalesPerson());

    foreach (Employee emp in list)
    {
        emp.GetWorkDetails(1234);
    }
}
```

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 разбить функционал на два интерфейса `IWork` и `IEmployee`:

```
public interface IEmployee
{
    string GetEmployeeDetails(int Id);
}

public interface IWork
{
    string GetWorkDetails(int Id);
}


public class SalesPerson : IEmployee
{
    public string GetEmployeeDetails(int Id)
    {
        return "SalesPerson Employee";
    }
}
```

```
public class Manager : IWork, IEmployee
{
    public string GetWorkDetails(int Id)
    {
        return "Manager Work";
    }

    public string GetEmployeeDetails(int Id)
    {
        return "Manager Employee";
    }
}
```

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

LSP – принцип подстановки Барбары Лисков

```
static void Main() {
    List<IWork> lstWork = new List<IWork>();
    lstWork.Add(new Manager());
    // lstWork.Add(new SalesPerson());
    foreach (var emp in lstWork)
    {
        Console.WriteLine(emp.GetWorkDetails(1234));
    }

    List<IEmployee> lstEmployee = new List<IEmployee>();
    lstEmployee.Add(new Manager());
    lstEmployee.Add(new SalesPerson());
    foreach (var emp in lstEmployee)
    {
        Console.WriteLine(emp.GetEmployeeDetails(1234));
    }
}
```

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.

Interface Segregation Principle

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:

```
public interface IEmployee
{
    bool AddDetailsEmployee();
}
```

ISP – принцип разделения интерфейсов

Допустим все классы **Employee** реализуют 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

```
public interface IEmployee
{
    bool AddDetailsEmployee();
    bool ShowDetailsEmployee(int id);
}
```

ПОЧЕМУ

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

ISP – принцип разделения интерфейсов

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

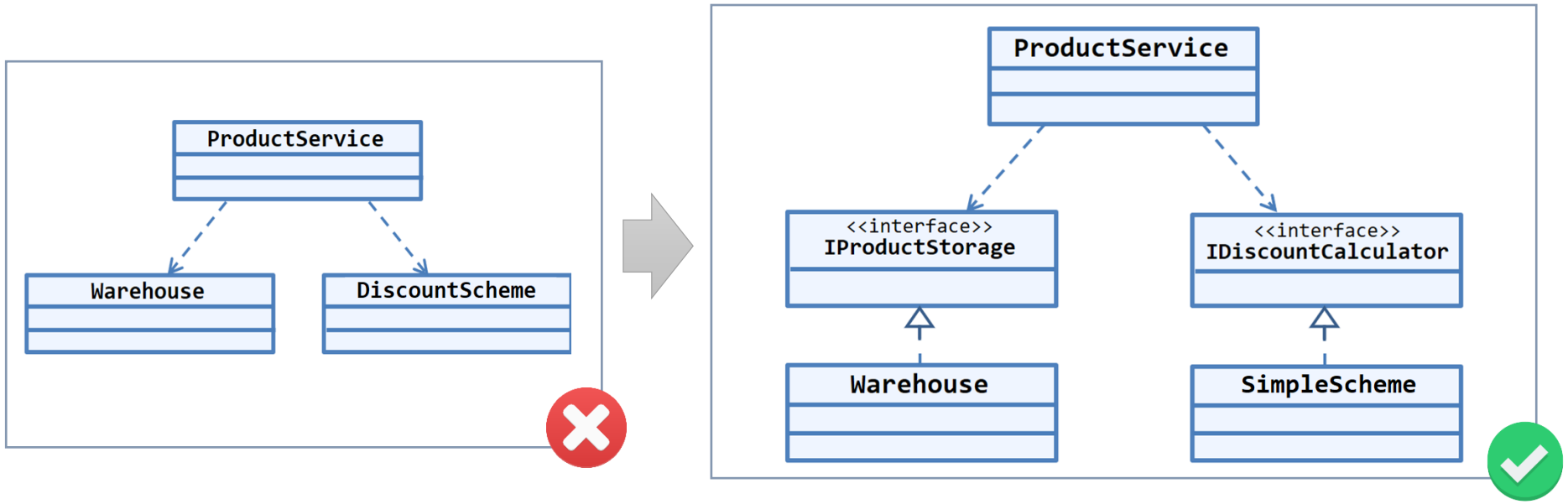
```
public interface IOperationAdd
{
    bool AddDetailsEmployee();
}

public interface IOperationGet
{
    bool ShowDetailsEmployee(int id);
}
```

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.

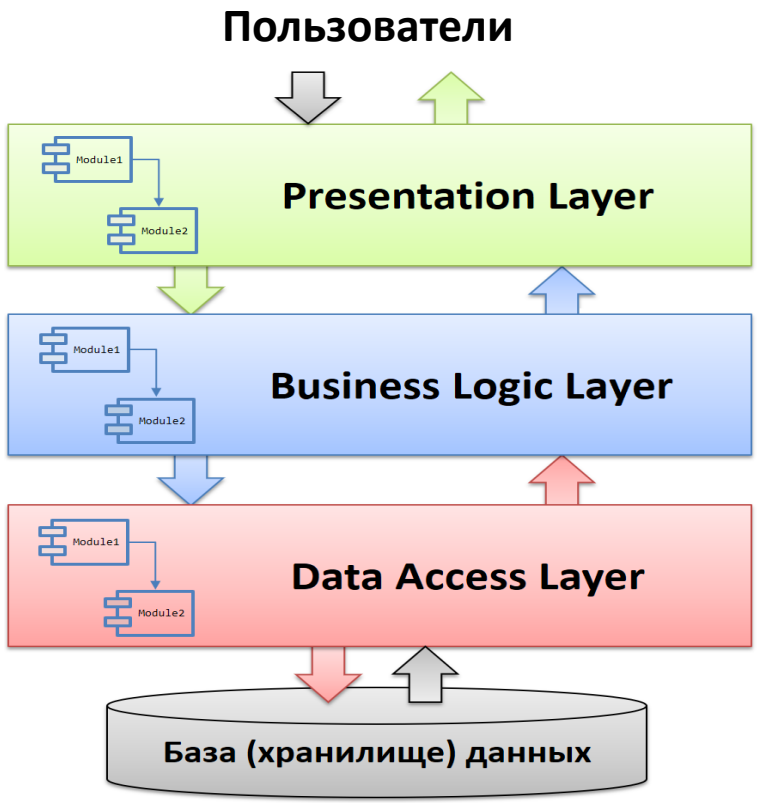


Dependency Inversion Principle

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.

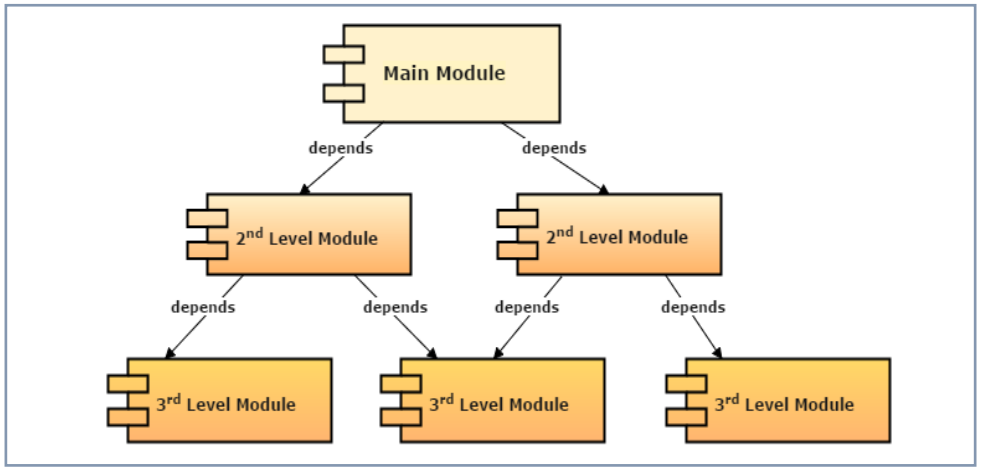
В ЧЕМ ПРОБЛЕМА:

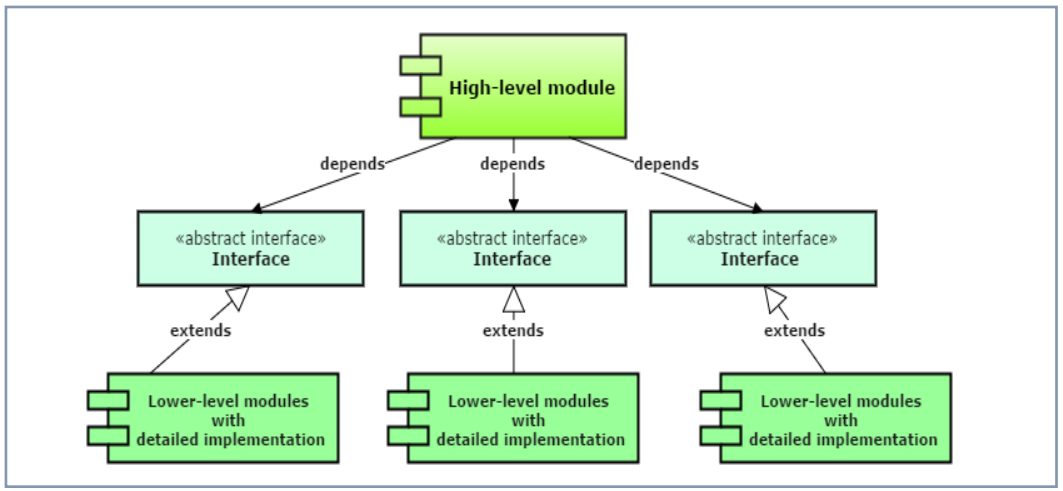
ЕСЛИ высокоуровневый класс имеет зависимость от дизайна и реализации другого класса, **ВОЗНИКАЕТ РИСК ТОГО, ЧТО ИЗМЕНЕНИЯ В ОДНОМ КЛАССЕ НАРУШАТ ДРУГОЙ КЛАСС.**

РЕШЕНИЕ:

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

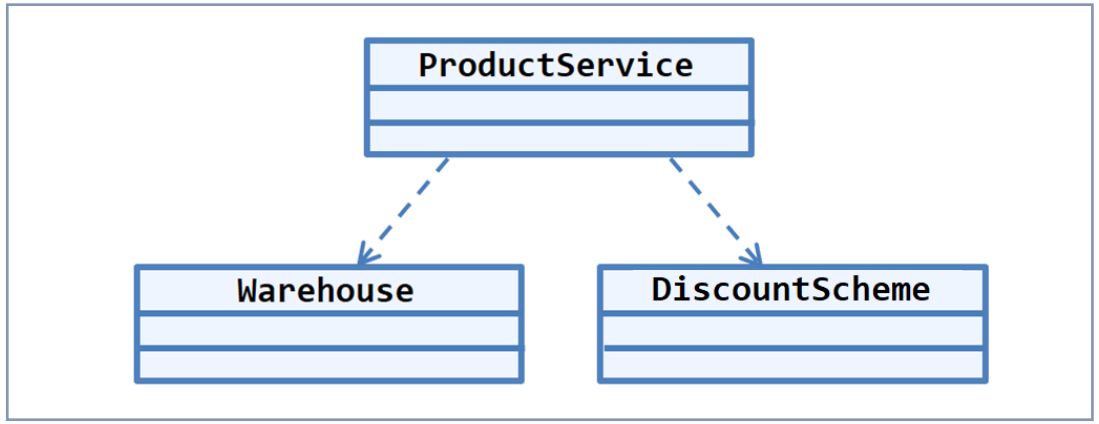
DIP – принцип инверсии зависимостей





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, в котором хранится товар;
   - ➡ `DiscountScheme` – схема начисления скидки.

DIP – принцип инверсии зависимостей

```
public class Product
{
    public double Cost { get; set; }
    public String Name { get; set; }
    public uint Count { get; set; }
}


public class Warehouse
{
    public IEnumerable<Product> GetProducts()
    {
      return new Product[] { new Product {Cost=140, Name = "Tyres", Count=1000},
                             new Product {Cost=160, Name = "Disks", Count=200},
                             new Product {Cost=100, Name = "Tools", Count=100}
                           };
    }
}
```

DIP – принцип инверсии зависимостей

```
public class DiscountScheme
{
   public double GetDiscount(Product p)
   {
      switch(p.Name)
      {
        case "Tyres":  return 0.01;
        case "Disks":  return 0.05;
        case "Tools":  return 0.1;
        default: return 0;
      }
   }
}
```

```
public class ProductService
{
  public double GetAllDiscount()
  {
     double sum = 0;

     Warehouse wh = new Warehouse();

     IEnumerable<Product> products = wh.GetProducts();

     DiscountScheme ds = new DiscountScheme();

     foreach (var p in products)
        sum += p.Cost * p.Count * ds.GetDiscount(p);

     return sum;
  }
}
```

DIP – принцип инверсии зависимостей

```
class Program
{
    static void Main(string[] args)
    {
        ProductService ps = new ProductService();
        Console.WriteLine("Discount for all products = " + ps.GetAllDiscount());

        Console.ReadKey();
    }
}
```

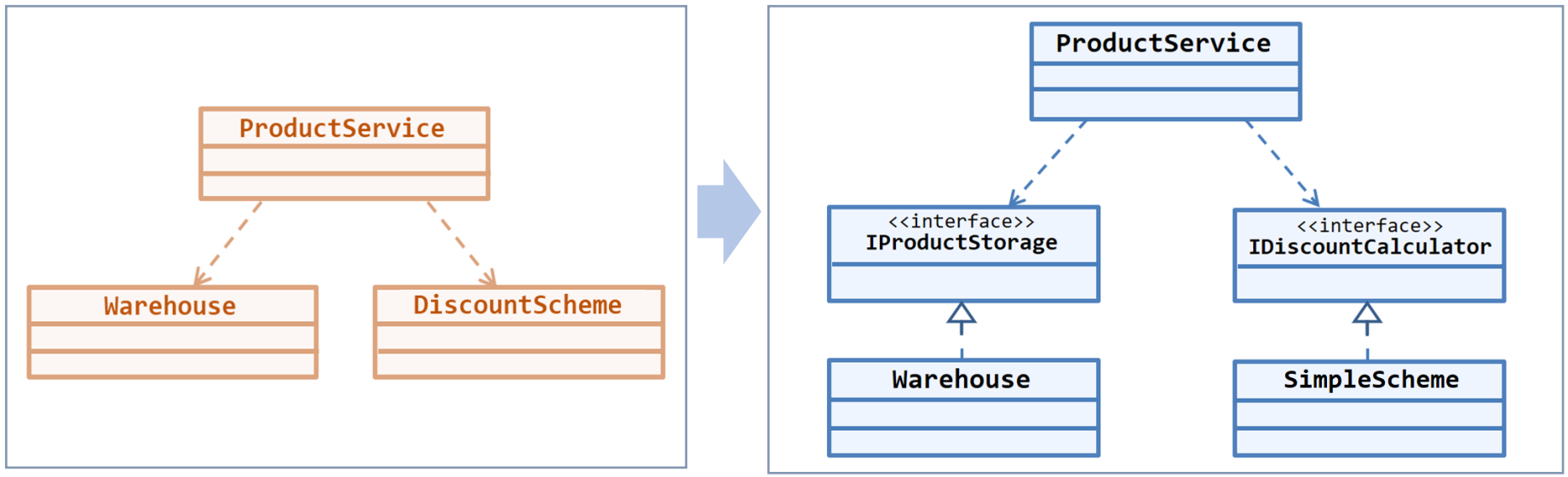
```
Discount for all products = 4000
```

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 другой реализацией `DiscountScheme`).

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

```
public interface IProductStorage
{
    IEnumerable<Product> GetProducts();
}

public interface IDiscountCalculator
{
    double GetDiscount(Product products);
}

public class Product
{
    public double Cost { get; set; }
    public String Name { get; set; }
    public uint Count { get; set; }
}
```

DIP – принцип инверсии зависимостей

```
public class Warehouse : IProductStorage
{
    public IEnumerable<Product> GetProducts()
    {
       return new Product[] { new Product {Cost=140, Name="Tyres", Count= 1000},
                              new Product {Cost=160, Name="Disks", Count= 200},
                              new Product {Cost=100, Name="Tools", Count= 100}};
    }
}

public class SimpleScheme : IDiscountCalculator
{
    public double GetDiscount(Product p)
    {
        switch (p.Name)
        {
            case "Tyres": return 0.01;
            case "Disks": return 0.05;
            case "Tools": return 0.1;
            default: return 0;
        }
    }
}
```

DIP – принцип инверсии зависимостей

```
public class ProductService
{
    public double GetAllDiscount(IProductStorage storage,
                    IDiscountCalculator discountCalculator)
    {
        double sum = 0;
        foreach (var p in storage.GetProducts())
            sum += p.Cost * p.Count * discountCalculator.GetDiscount(p);

        return sum;
    }
}

class Program
{
    static void Main(string[] args)
    {
        ProductService ps = new ProductService();
        Console.WriteLine("Discount for all products = " +
                          ps.GetAllDiscount(new Warehouse(), new SimpleScheme()));
        Console.ReadKey();
    }
}
```

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: подведение итогов

- **Single Responsibility** – делайте модули меньше (1 ответственность)
- **Open/Closed** – делайте модули расширяемыми
- **Liskov Substitution** – наследники должны вести себя так же, как родители
- **Interface Segregation** – делите слишком сложные интерфейсы
- **Dependency Inversion** – используйте интерфейсы