

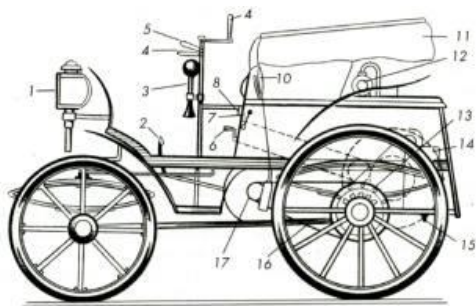


# **Принципы проектирования**

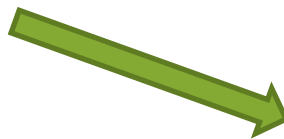
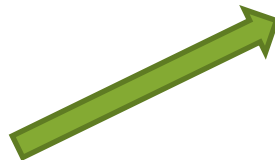
# Процесс разработки



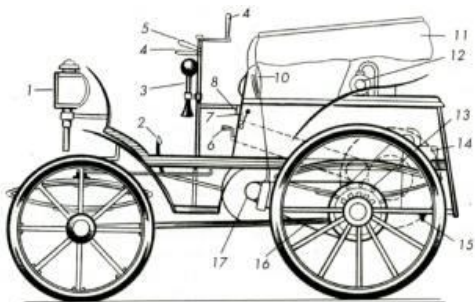
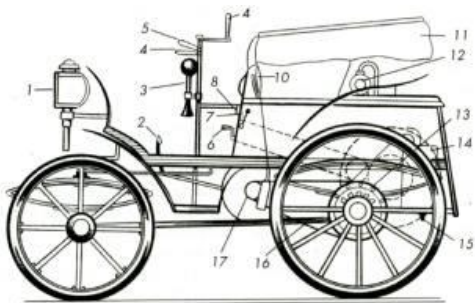
Клиент



Разработчики



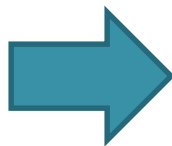
# С точки зрения клиента:



?

# Момент истины

- расширение приложения
- исправление ошибок
- написание тестов



- больше времени
- больше ошибок
- иногда очень сложно

- плохое внутреннее качество



- плохое внешнее качество



# Нарастание энтропии

- разные люди пишут код
- растет размер приложения
- растет связность между модулями, классами



Project timeline

# Технический долг

Технические долги включают ту работу в проекте, которую мы решаем не делать в данный момент, но которая будет мешать развитию проекта в дальнейшем, если не будет выполнена

- черновая разработка
- отсутствие проектирования

%  
% %

- увеличение времени на изменения
- рост количества ошибок



- рефакторинг
- улучшение качества дизайна и кода

Тело кредита

Вывод: технические долги == финансовые долги



# Свойства качественного кода

- расширяемость, гибкость (extensibility, agility)
- сопровождаемость (maintainability)
- простота (simplicity)
- читабельность, понятность (readability, clarity)
- тестируемость (testability)



Качественный код - это не просто какой-то абстрактный «красивый» код, а код, который обладает полезными внутренними свойствами.

# Как писать качественный код

## Чем руководствоваться?

- здравый смысл, опыт
- паттерны проектирования
- принципы проектирования
- правила рефакторинга
- модульные тесты

## Практики:

- парное программирование
- code review
- рефакторинг
- модульные тесты и TDD/BDD

## Когда нужно рефакторить?

- анти-паттерны проектирования
- code smell
- костыли
- большие временные затраты на изменения



# Иерархия. Принципы и паттерны





# Общие принципы проектирования

# SoC: Separation of Concerns

*Разделение системы на отдельные части (модули, звенья, слои, классы), которые будут как можно меньше связаны между собой.*

## **Достигается за счет:**

- разделение на звенья (tiers)
- разделение на слои (layers)
- модульность
- разделение на классы
- инкапсуляция

# DRY: Don't Repeat Yourself

*Каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы*

## **Свойства:**

- снижает затраты на поддержку/развитие/изменение.
- относится не только к дублированию кода, но и к дублированию других абстракций системы

## **Достигается за счет:**

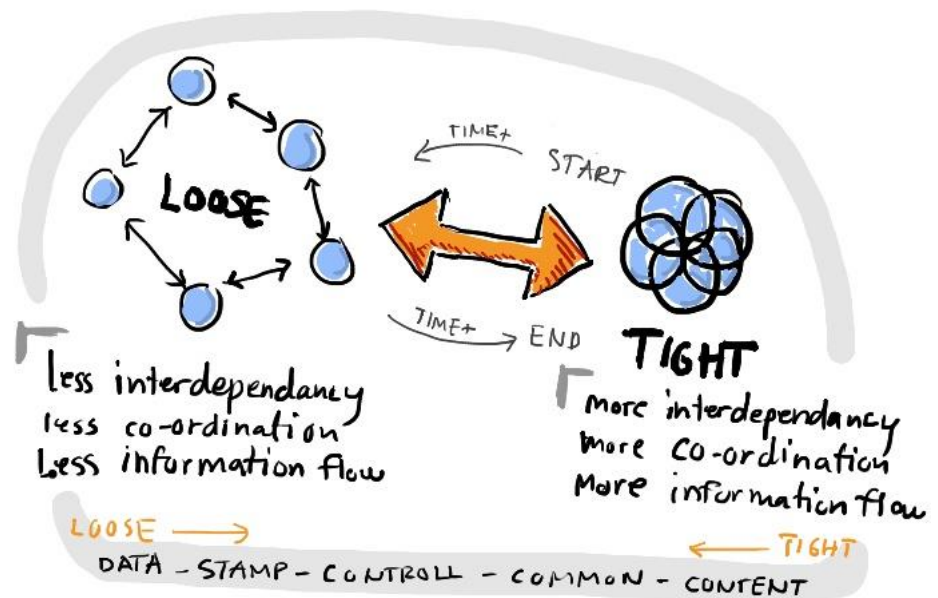
- отсутствие copy-paste
- повторное использование кода

# Low Coupling

*Coupling (связанность) – мера, определяющая, насколько жестко один элемент связан с другими элементами, или каким количеством данных о других элементах он обладает.*

**Достигается за счет:**

- однонаправленные связи
- зависимость от интерфейсов
- don't talk to the strangers



# High Cohesion

*Cohesion* (сцепленность, сфокусированность) – мера, определяющая связанность и сфокусированность обязанностей/ответственности элемента.

## Достигается за счет:

- сужение обязанностей элемента
- разделение ответственности между несколькими элементами
- группирование похожей ответственности в одном элементе





# KISS: Keep it simple, stupid!

*Простота системы является основной целью и ценностью.*

## **Связаны:**

- Бритва Оккама: *«Не следует плодить новые сущности без самой крайней на то необходимости»* или *«Объяснение, требующее наименьших допущений, с большей вероятностью является правильным»*
- Эйнштейн: *«Все должно быть предельно просто, но не проще»*

## **Достигается за счет:**

- прагматичный подход к проектированию
- чувство меры, опыт



# YAGNI: You Ain't Gonna Need It

*Не нужно добавлять функциональность пока в ней нет непосредственной нужды.*

## **Вытекает:**

- предварительная оптимизация вредна

## **Достигается за счет:**

- «ленивый» подход к проектированию

**Минус:** новый функционал может занимать много времени  
Важно быть прагматиком и учитывать будущие требования!

# Don't make me think

Код должен легко читаться и восприниматься с минимумом усилий, если код вызывает затруднения чтобы его понять, то вероятно его стоит упростить

## Write Code for the Maintainer

Практически любой код, который вы пишете, предстоит поддерживать в будущем вам или кому-то другому. В будущем, когда вы вернётесь к коду, обнаружите, что большая его часть совершенно вам незнакома, так что старайтесь писать как будто для другого.

“Пишите код так, как будто сопровождать его будет склонный к насилию психопат, который знает, где вы живете.” (Стив Макконнелл «Совершенный код»)

# Hide Implementation Details

Скрытие деталей реализации позволяет вносить изменения в код компонента с минимальными затрагиванием других модулей которые используют этот компонент

## Law of Demeter

Компоненты кода должны взаимодействовать только с их непосредственными связями (например, классы от которых они унаследованы, объекты, которые они содержат, объекты, переданные с помощью аргументов и т.д.)

# Avoid Premature Optimization


Даже не думайте об оптимизации, если ваш код работает, но медленней, чем вы хотите. Только потом можно начать задумываться об оптимизации, и только основываясь на полученном опыте.

Мы должны забыть про небольшие улучшения эффективности, скажем, около 97% времени: преждевременная оптимизация — корень всех бед.

© Дональд Кнут

## Code Reuse is Good

Не очень содержательный, но тоже хороший принцип как и все другие. Повторное использование кода повышает надежность и уменьшает время разработки



# Принципы объектно-ориентированного проектирования





# SOLID

Software Development is not a Jenga game

# SP Pr



## Just Because You Can, Doesn't Mean You Should

# SRP: Single Responsibility Principle

- 1. Не должно быть больше одной причины для изменения класса (Роберт Мартин Принципы, паттерны и практики гибкой разработки. — 2006 ).*
- 2. Каждый класс должен быть сфокусирован на своей области ответственности.*

## **Цель:**

- упростить внесение изменений
- защититься от побочных эффектов при изменениях

## **Достигается за счет:**

- правильное проектирование
- использование паттернов проектирования

# SRP:неправильный вариант

```
public class Account
{
    public string Number;
    public decimal CurrentBalance;
    public void Deposit(decimal amount) { ... }
    public void Withdraw(decimal amount) { ... }
    public void Transfer(decimal amount, Account recipient) { ... }
    public TaxTable CalculateTaxes(int year) { ... }
    public void GetByNumber(string number) { ... }
    public void Save() { ... }
}
```

# SRP: правильный вариант

```
public class Account
{
    public string Number;
    public decimal CurrentBalance;
    public void Deposit(decimal amount) { ... }
    public void Withdraw(decimal amount) { ... }
    public void Transfer(decimal amount, Account recipient) {
... }
}

public class AccountRepository
{
    public Account GetByNumber(string number) { ... }
    public void Save(Account account) { ... }
}

public class TaxCalculator
{
    public TaxTable CalculateTaxes(int year) { ... }
}
```

# Типичные примеры нарушения SRP

Типичными примерами нарушения SRP являются:

- **смешивание логики с инфраструктурой.** Бизнес-логика смешана с представлением, слоем персистентности, находится внутри WCF или Windows-сервисов. Должна быть возможность сосредоточиться на бизнес-правилах, не обращая внимания на второстепенные инфраструктурные детали;
- **слабая связность (low cohesion).** Класс/модуль/метод не является цельным и решает несколько несвязанных задач. Проявляются несколько групп методов, каждая из которых обращается к подмножеству полей, не используемых другими методами;
- **выполнение нескольких несвязанных задач.** Класс/модуль может быть цельным, но решать несколько несвязанных задач (вычисление заработной платы и построение отчета). Класс/модуль/метод должен быть сфокусированным на решении минимального числа задач;



# Типичные примеры нарушения SRP

- **решение задач разных уровней абстракции.** Класс/метод не должен отвечать за задачи разного уровня. Например, класс удаленного заместителя не должен самостоятельно проверять аргументы, заниматься сериализацией и шифрованием. Каждый из этих аспектов должен решаться отдельным классом
- **решение задач разных уровней абстракции.** Класс/метод не должен отвечать за задачи разного уровня. Например, класс удаленного заместителя не должен самостоятельно проверять аргументы, заниматься сериализацией и шифрованием. Каждый из этих аспектов должен решаться отдельным классом

# OCP: Open-Closed Principle



**OPEN CLOSED PRINCIPLE**

Open Chest Surgery Is Not Needed When Putting On A Coat

# OCP: Open-Closed Principle

*«Программные сущности (классы, модули, функции и т. п.) должны быть открытыми для расширения, но закрытыми для модификации» (Мартин Р. Принципы, паттерны и практики гибкой разработки. — 2006).*

## Цель:

- добиться гибкости системы
- избежать сильных переработок дизайна системы при изменениях – он должен быть простым и устойчивым к изменениям («не если, а когда»)

## Достигается за счет:

- правильное наследование
- инкапсуляция

# ОСР: Open-Closed Principle

- **Определение от Бертрана Мейера:** модули должны иметь возможность быть как открытыми, так и закрытыми. При этом понятия открытости и закрытости определяются так.
- Модуль называют открытым, если он еще доступен для расширения. Например, имеется возможность расширить множество операций в нем или добавить поля к его структурам данных.
- Модуль называют закрытым, если он доступен для использования другими модулями. Это означает, что модуль (его интерфейс — с точки зрения сокрытия информации) уже имеет строго определенное окончательное описание. На уровне реализации закрытое состояние модуля означает, что можно компилировать модуль, сохранять в библиотеке и делать его доступным для использования другими модулями (его клиентами).

# OCP: Open-Closed Principle



Заходите, мы  
*открыты*. Не стес-  
няйтесь, расширяйте наши классы  
любым нужным поведением. Если ваши  
потребности изменятся (а это наверняка  
произойдет), просто создайте собствен-  
ное расширение.

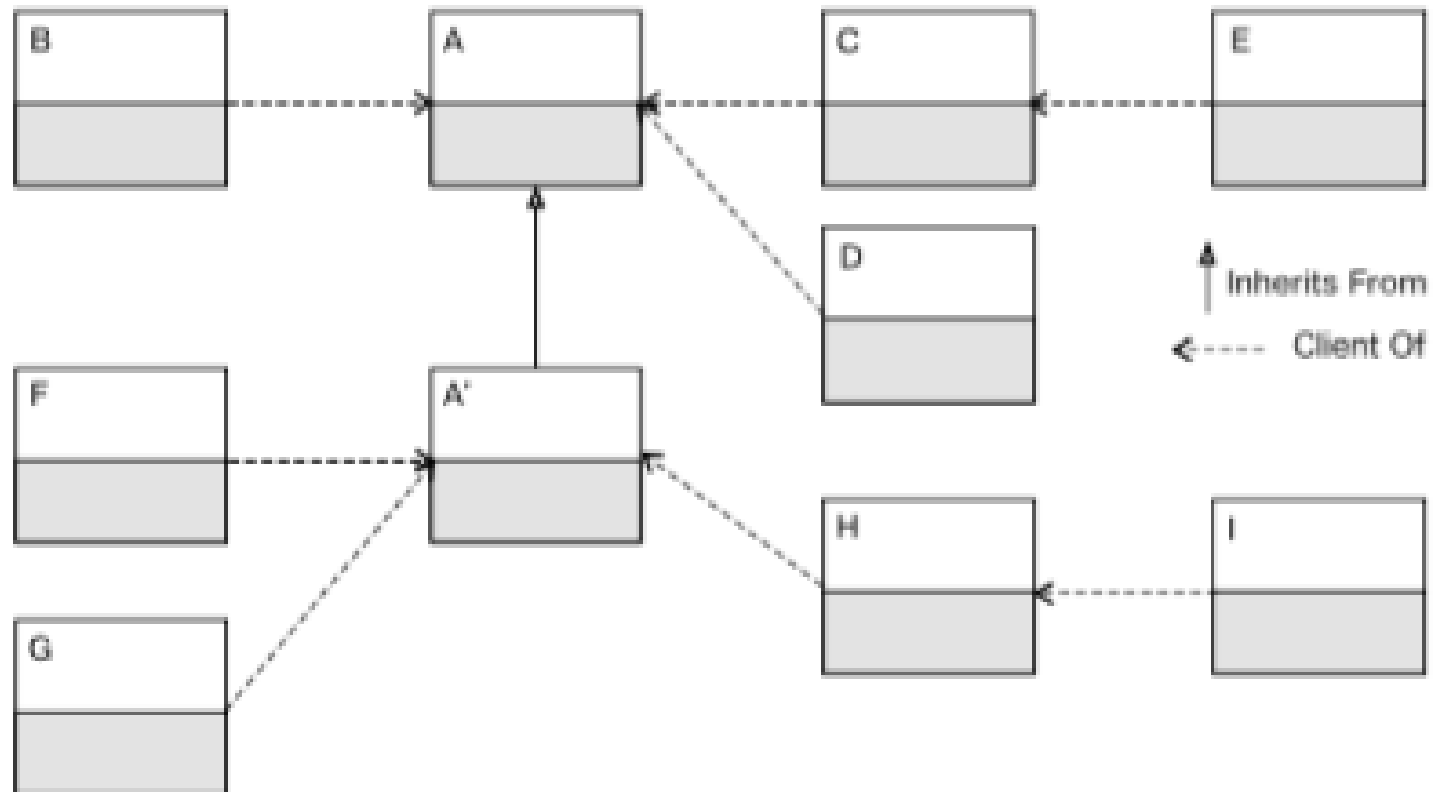
# OCP: Open-Closed Principle



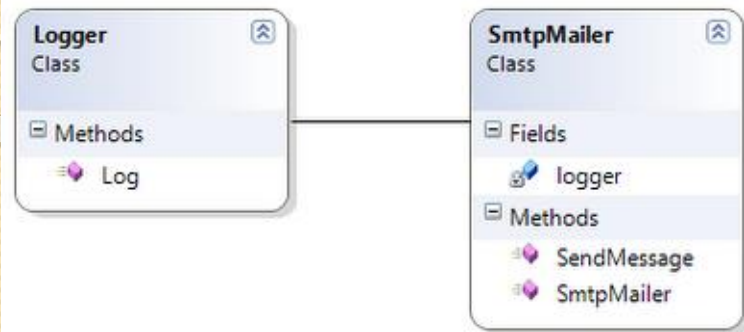
Извините, мы *закры-*  
*ты*. Мы потратили  
много времени на про-  
верку и отладку этого кода и не можем позво-  
лить вам изменять его. Код должен остаться  
закрытым для изменения. Если вас это не  
устраивает — обратитесь к директору.



# OCP: Open-Closed Principle



# ОСР: неправильный вариант



```
public class SmtplibMailer
{
    private readonly Logger logger;

    public SmtplibMailer()
    {
        logger = new Logger();
    }

    public void SendSmtplibMessage(string message)
    {
        // отправить сообщение
        logger.Log(message);
    }
}
```

Изменение: нужно писать лог в базу данных

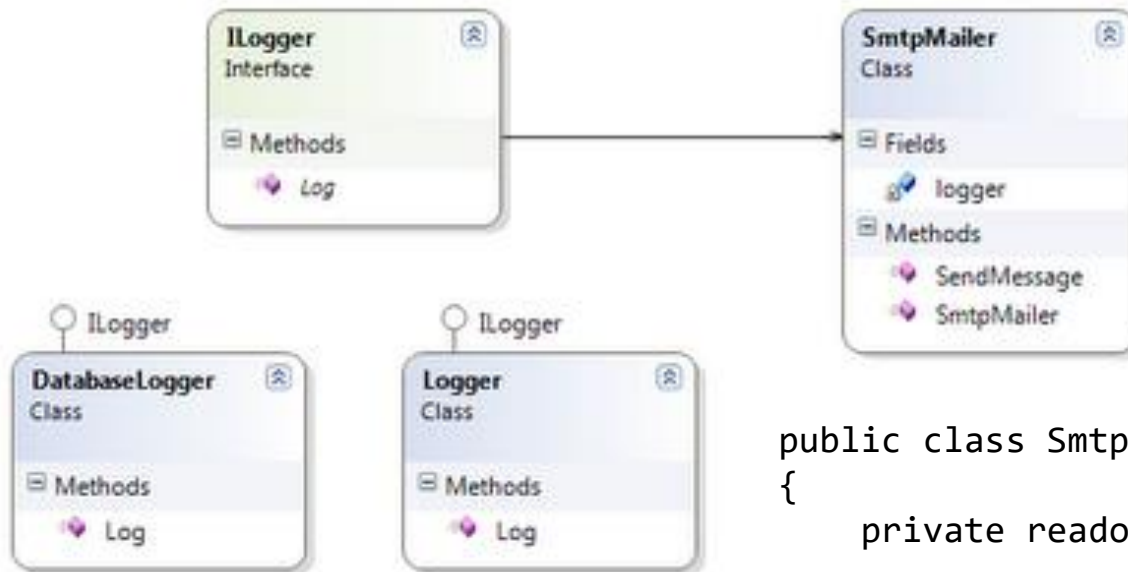
```
public class DatabaseLogger
{
    public void Log(string logText)
    {
        // сохранить лог в базе данных
    }
}

public class SmtplibMailer
{
    private readonly DatabaseLogger logger;

    public SmtplibMailer()
    {
        logger = new DatabaseLogger();
    }

    public void SendSmtplibMessage(string message)
    {
        // отправить сообщение
        logger.Log(message);
    }
}
```

# ОСР: правильный вариант



```
public class SmtpMailer
{
    private readonly ILogger logger;

    public SmtpMailer(ILogger logger)
    {
        this.logger = logger;
    }

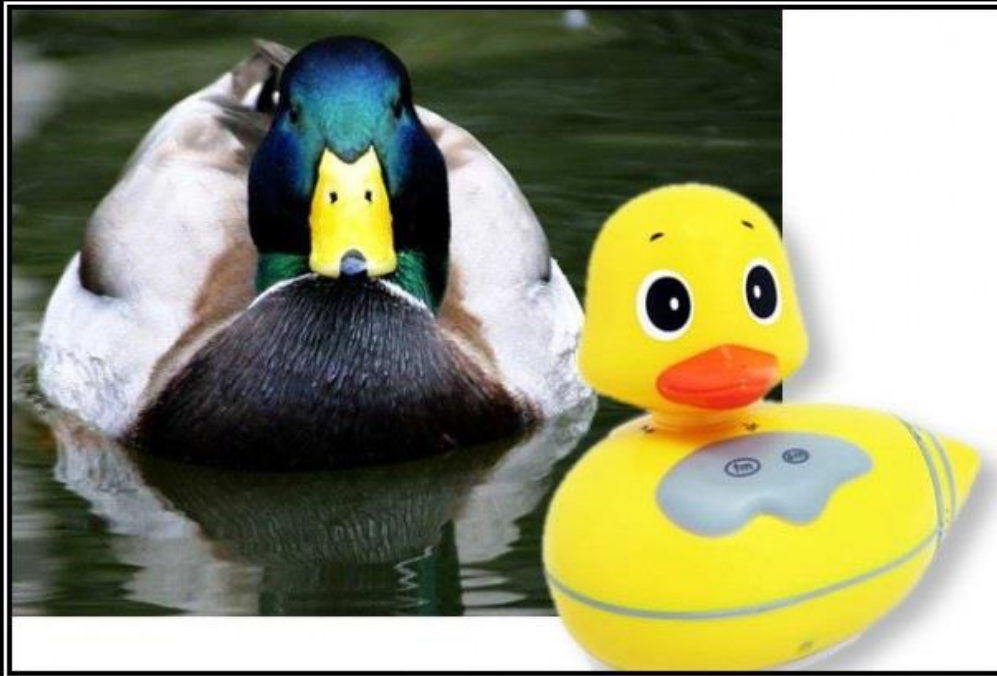
    public void SendMessage(string message)
    {
        // отправить сообщение
        logger.Log(message);
    }
}
```

# ОСР: типичные нарушения

Типичными примерами нарушения принципа «открыт/закрит» являются следующие.

- **Интерфейс класса является нестабильным.** Постоянные изменения интерфейса класса, используемого во множестве мест, приводят к постоянным изменениям во многих частях системы.
- **«Размазывание» информации об иерархии типов.** В коде постоянно используются понижающие приведения типов (downcasting), что «размазывает» информацию об иерархии типов по коду приложения. Это затрудняет добавление новых типов и усложняет понимание текущего решения.

# LSP: Liskov Substitution Principle



## LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You  
Probably Have The Wrong Abstraction

# LSP: Liskov Substitution Principle

- «Должна быть возможность вместо базового типа подставить любой его подтип» (Мартин Р. Принципы, паттерны и практики гибкой разработки. — 2006).
- «...Если для каждого объекта  $o_1$  типа  $S$  существует объект  $o_2$  типа  $T$  такой, что для всех программ  $P$ , определенных в терминах  $T$ , поведение  $P$  не изменяется при замене  $o_2$  на  $o_1$ , то  $S$  является подтипом (subtype) для  $T$ » (Лисков Б. Абстракция данных и иерархия. — 1988).

# LSP: Liskov Substitution Principle

- 1. Подтипы должны быть заменяемыми их исходными типами.*
- 2. Наследники должны соблюдать контракт предка*

## **Цель:**

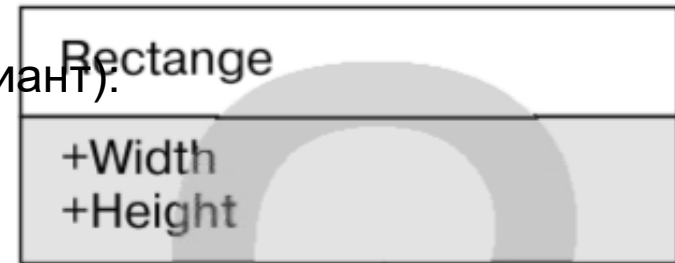
- избежать побочных эффектов и ошибок в существующем коде, работающем с базовыми классами, при добавлении наследников
- строить правильные иерархии наследования

## **Достигается за счет:**

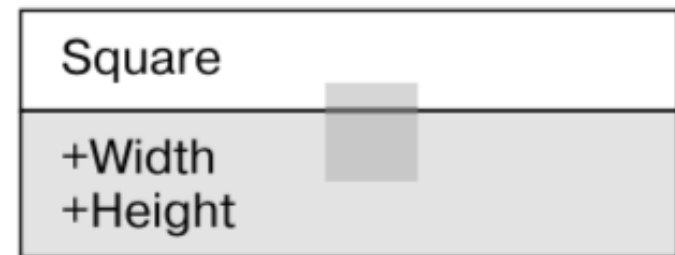
- правильное наследование классов

# LSP: Liskov Substitution Principle

**Контракт прямоугольника (инвариант):**  
ширина и высота положительны.



**Контракт квадрата (инвариант):**  
ширина и высота положительны,  
ширина и высота равны.



**Инвариант класса** — это утверждение, которое (должно быть) истинно применительно к любому объекту данного класса в любой момент времени (за исключением переходных процессов в методах объекта).



# LSP: неправильный вариант

```
class Rectangle
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }
    public int CalculateArea() { return Width * Height; }
}

class Square : Rectangle
{
    public override int Height
    {
        get { return base.Height; }
        set { base.Height = value; base.Width = value; }
    }
    public override int Width
    {
        get { return base.Width; }
        set { base.Width = value; base.Height = value; }
    }
}

class Program
{
    static void Main()
    {
        Rectangle r = new Square();
        r.Width = 3; r.Height = 2;
        Assert.AreEqual(6, r.CalculateArea());
    }
}
```

# LSP: правильный вариант

```
class Rectangle : IFigure
{
    public int Width { get; set; }
    public int Height { get; set; }
    public int CalculateArea() {
        return Width * Height;
    }
}

class Square : IFigure
{
    public int Side { get; set; }
    public int CalculateArea() {
        return Side * Side;
    }
}

class Program {
    static void Main() {
        Rectangle r = new Square(); // fail on compilation
        r.Width = 3;
        r.Height = 2;
        Assert.AreEqual(6, r.CalculateArea());
    }
}
```

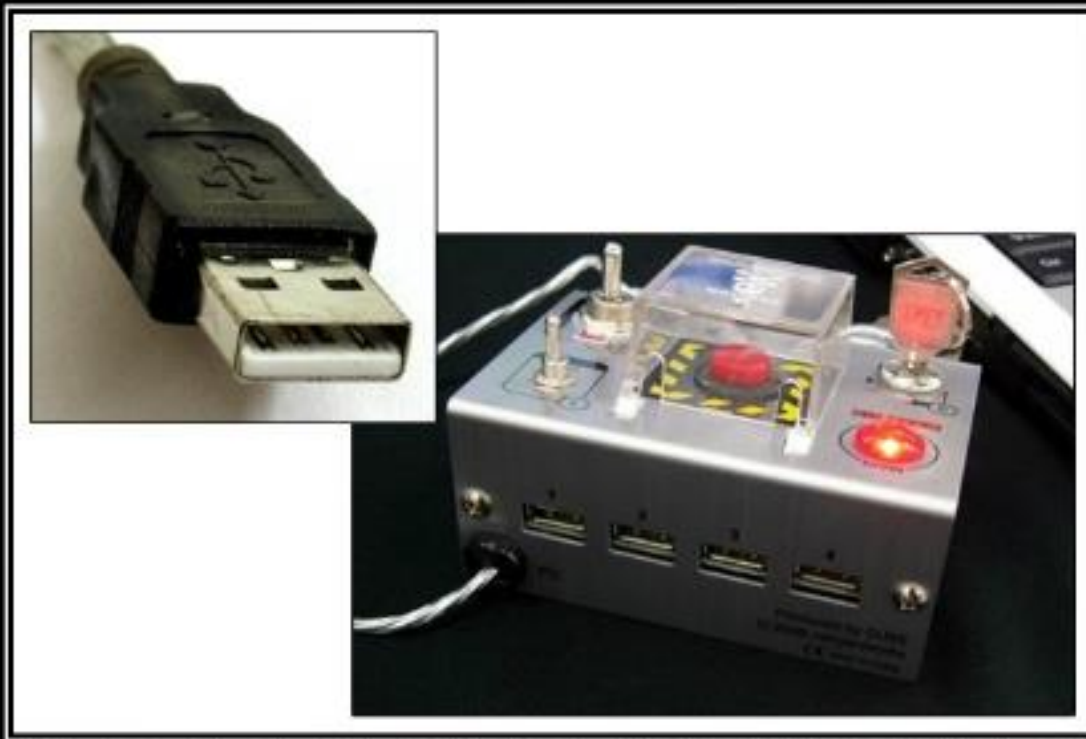
# LSP и контракты

- Производные классы не должны усиливать предусловия (не должны требовать большего от своих клиентов).
- Производные классы не должны ослаблять постусловия (должны гарантировать как минимум то же, что и базовый класс).
- Производные классы не должны нарушать инварианты базового класса (инварианты базового класса и наследников суммируются).
- Производные классы не должны генерировать исключения, не описанные базовым классом.

# Типичные примеры нарушения LSP

- Производные классы используются полиморфным образом, но их поведение не согласуется с поведением базового класса: генерируются исключения, не описанные контрактом базового класса, или не выполняются действия, предполагаемые контрактом базового класса.
- Контракт базового класса настолько нечеткий, что реализовать согласованное поведение наследником просто невозможно.

# ISP: Interface Segregation Principle



INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

# ISP: Interface Segregation Principle

- 1. *«Клиенты не должны вынужденно зависеть от методов, которыми не пользуются» (Мартин Р. Принципы, паттерны и практики гибкой разработки. — 2006).*
- 2. *Зависимость между классами должна быть ограничена как можно более «узким» интерфейсом.*

## Цель:

- ограничить знание одного класса о другом (принцип наименьшего знания)
- уменьшить зависимость между классами
- уменьшить количество методов для реализации при наследовании

## Достигается за счет:

- фокусирование интерфейсов на своей ответственности
- наследование от нескольких интерфейсов, а не от одного

# ISP: Interface Segregation Principle

Зависимости, стабильность которых уменьшается от очень стабильной до нестабильной:

- примитивные типы;
- объекты-значения (неизменяемые пользовательские типы);
- объекты со стабильным интерфейсом и поведением (пользовательские типы, интерфейс которых стабилен, а поведение не зависит от внешнего окружения);
- объекты с изменчивыми интерфейсом и поведением (типы расположены на стыке модулей, которые постоянно подвергаются изменениям, или типы, которые работают с внешним окружением: файлами, базами данных, сокетами и т. п.).

# ISP: неправильный вариант

```
public interface IBird
{
    void Eat();
    void Sing();
    void Fly();
}

public class Nightingale: IBird
{
    // здорово
}

public class Pigeon : IBird
{
    // ну, я не очень хорошо пою, но ладно
}

public class Penguin : IBird
{
    // хм... а я вообще птица?
}
```



# ISP: правильный вариант

```
public interface ICommonBird
{
    void Eat();
}
public interface ISingingBird
{
    void Sing();
}
public interface IFlyingBird
{
    void Fly();
}
public class Nightingale: ICommonBird, ISingingBird, IFlyingBird
{
    // хм, ничего не изменилось
}
public class Pigeon : ICommonBird, IFlyingBird
{
    // о, так лучше, я могу не петь
}
public class Penguin : ICommonBird
{
    // так намного лучше! хотя я еще и плавать могу ☺
}
```

# Типичные примеры нарушения ISP

- Метод принимает аргументы производного класса, хотя достаточно использовать базовый класс.
- У класса два или более ярко выраженных вида клиентов.
- Класс зависит от более сложной зависимости, чем нужно: принимает интерфейс провайдера вместо результатов его работы и т. п.
- Класс зависит от сложного интерфейса, что делает его зависимым от всех типов, используемых в этом интерфейсе.

# DIP: Dependency Inversion Principle



Принцип инверсии зависимостей на  
практике

# DIP: Dependency Inversion Principle



**DEPENDENCY INVERSION PRINCIPLE**

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

# DIP: Dependency Inversion Principle

*«Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те и другие должны зависеть от абстракций.*

*Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций» (Мартин Р. Принципы, паттерны и практики гибкой разработки. — 2006).*

## Цель:

- уменьшить связанность (low coupling)

## Связан:

- «Hollywood principle»: *Don't call us, we'll call you*

## Достигается за счет:

- паттерны Dependency Injection, Service Locator
- зависимость от абстракций
- DI-frameworks

# DIP: Dependency Inversion Principle

## Цель:

1. Дизайн является жестким (rigid), если его тяжело изменить, поскольку любое изменение влияет на слишком большое количество других частей системы.
2. Дизайн является хрупким (fragile), если при внесении изменений неожиданно ломаются другие части системы.
3. Дизайн является неподвижным (immobile), если код тяжело использовать повторно в другом приложении, поскольку его слишком сложно «выпутать» из текущего приложения.

# DIP: неправильный вариант

```
class Developer
{
    public void WriteApplication()
    {
        IApplication helloWorld = new HelloWorldApp();
        WriteCode(helloWorld);
    }

    private void WriteCode() { ... };
}

public interface IApplication
{
    string GetCode();
}

public class HelloWorldApp : IApplication
{
    // реализация
}

public class VeryBigApp : IApplication
{
    // реализация
}
```

# DIP: правильный вариант

```
class Developer
{
    public void WriteApplication(IApplication app)
    {
        WriteCode(app);
    }

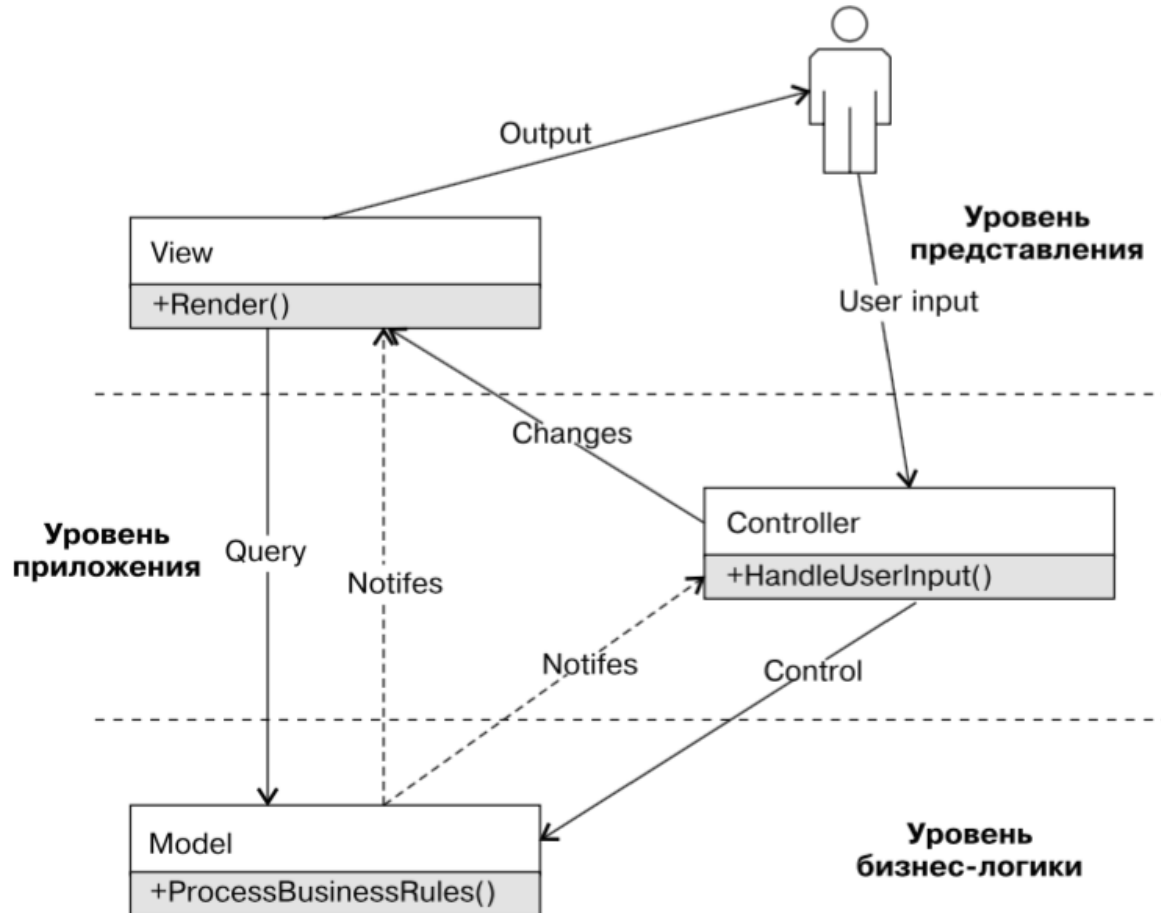
    private void WriteCode() { ... };
}

public static void Main()
{
    Developer dev = new Developer();
    IApplication app = new VeryBigApp(); // теперь мы умеем писать что угодно
    dev.WriteApplication(app);
}
```

Это шаблон Dependency Injection (Method Injection), но можно использовать и другие шаблоны (например, Service Locator), а также IoC Frameworks

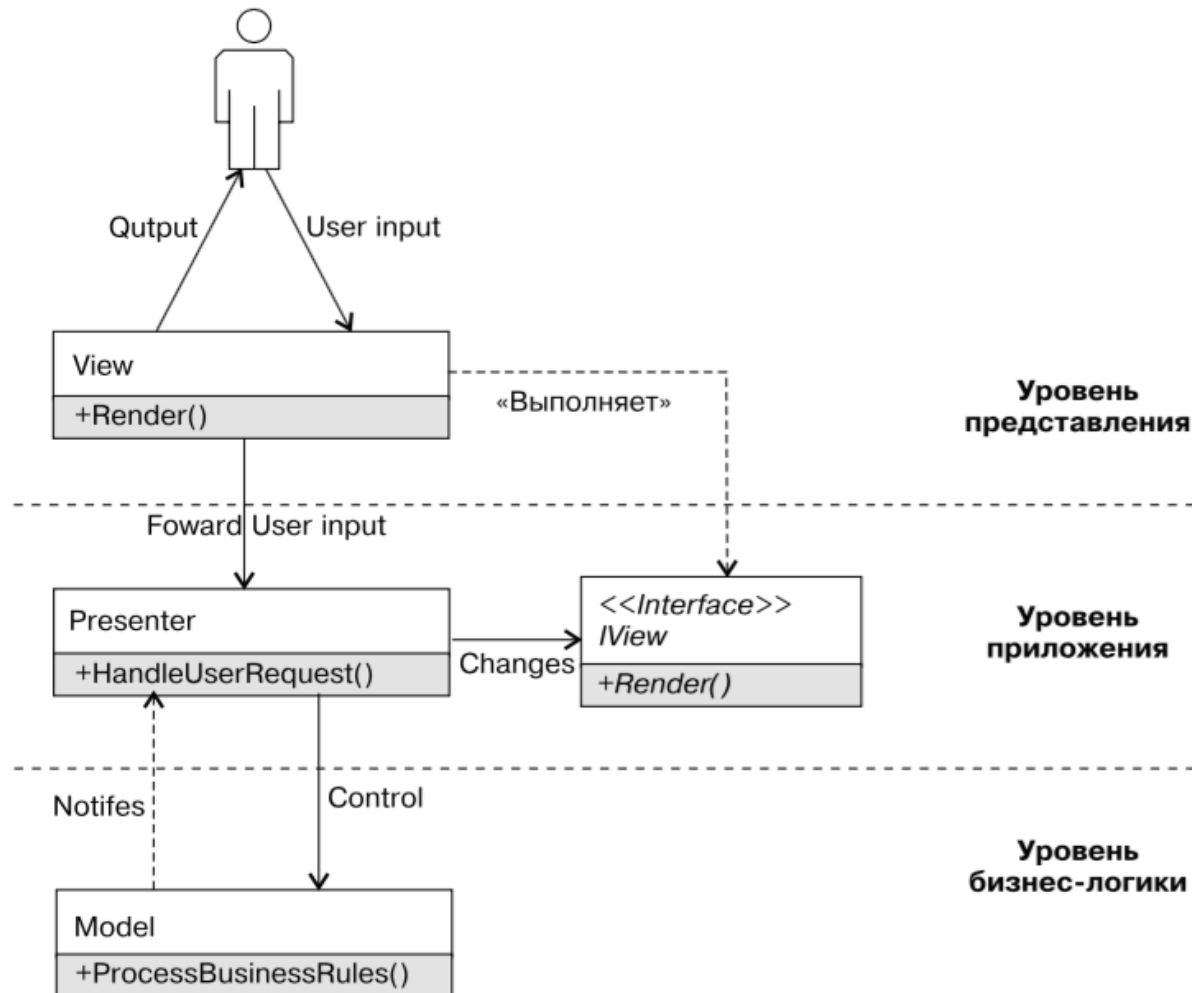


# DIP: Dependency Inversion Principle



Основная идея этого паттерна в том, что и контроллер и представление зависят от модели, но модель никак не зависит от этих двух компонент

# DIP: Dependency Inversion Principle



# DI Patterns. Constructor Injection

**Назначение** Разорвать жесткую связь между классом и его **обязательными** зависимостями.

Передача зависимостей через конструктор является предпочтительным способом внедрения зависимостей, поскольку в этом случае проще всего обеспечить четкий контракт между классом и его клиентами

# DI Patterns. Constructor Injection

```
// Декораторы var ms = new MemoryStream();  
var bs = new BufferedStream(ms);
```

```
// Стратегия сортировки  
var sortedArray =  
    new SortedList<int,string>(new CustomComparer());
```

```
// Класс ResourceReader принимает Stream  
Stream ms = new MemoryStream();  
var resourceReader = new ResourceReader(ms);
```

```
// BinaryReader/BinaryWriter, StreamReader/StreamWriter  
// также принимают Stream через конструктор  
var textReader = new StreamReader(ms);
```

# DI Patterns. Property Injection

**Назначение** - Разорвать жесткую связь между классом и его **необязательными** зависимостями.

Передачу зависимости через свойство следует применять **только для необязательных зависимостей**, для которых существует разумная реализация по умолчанию, известная классу сервиса; при этом должна существовать возможность изменить зависимость во время исполнения сервиса без серьезных последствий (в противном случае должно генерироваться исключение).

# DI Patterns. Property Injection

```
// Dependency.dll
public interface IDependency { }

// CustomService.dll (!)
internal class DefaultDependency : IDependency { }

// CustomService.dll
public class CustomService
{
    public CustomService()
    {
        Dependency = new DefaultDependency();
    }
    public IDependency Dependency { get; set; }
}
```

# DI Patterns. Property Injection

```
public class CustomService
{
    private readonly IDependency _dependency;
    public CustomService()
        : this(new DefaultDependency())
    { }

    public CustomService(IDependency dependency)
    {
        _dependency = dependency;
    }
}
```

# DI Patterns. Method Injection

**Назначение** Предоставить классу сервиса дополнительную информацию для выполнения определенной задачи.

Зависимости, передаваемые через конструктор или свойство являются «статическими» зависимостями и требуются объекту на протяжении всего времени его жизни, и не изменяются от одной операции к другой. Когда зависимость (ее реальный тип или состояние) может быть разной от вызова к вызову или это единственный способ передачи зависимости, поскольку метод является статическим, тогда более подходящим является передача зависимости именно через метод.



# DI Patterns. Method Injection

**Метод является статическим и другие варианты не подходят.**

```
public interface ICurrencyRate
{
    int GetCurrencyRate(string currency);
}

// PaymentService
public static Money CalculatePayment(
    ICurrencyRate currencyRate)
{
    return new Money();
}
```

# DI Patterns. Method Injection

**Зависимость может изменяться от операции к операции**

// Задаёт "стратегию" форматирования отчёта

```
public interface IReportFormatter
{
    string GetFormatString();
}
```

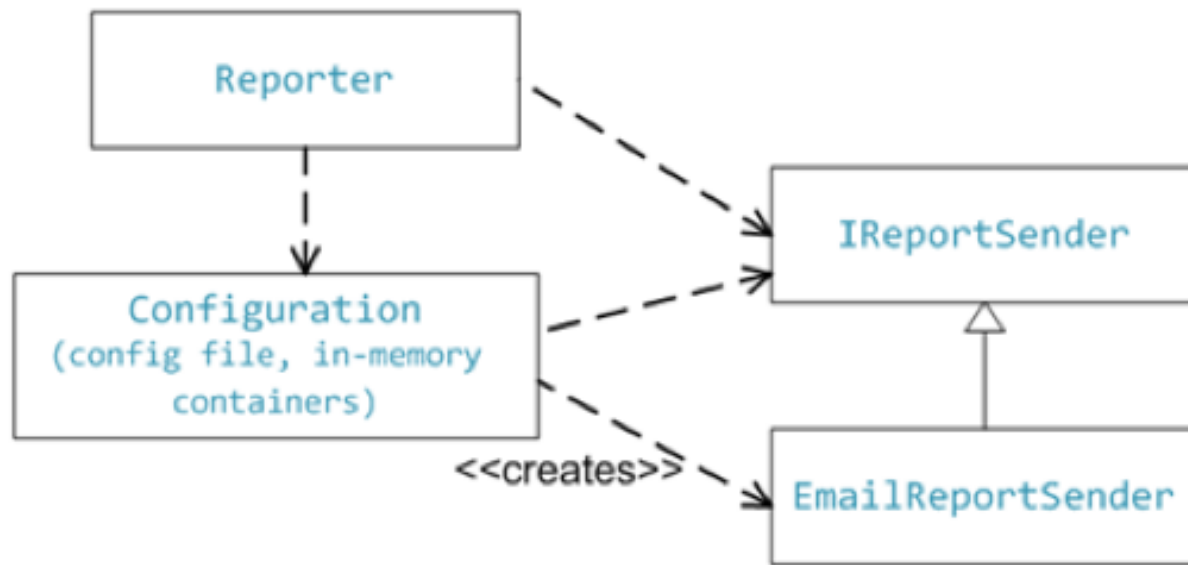
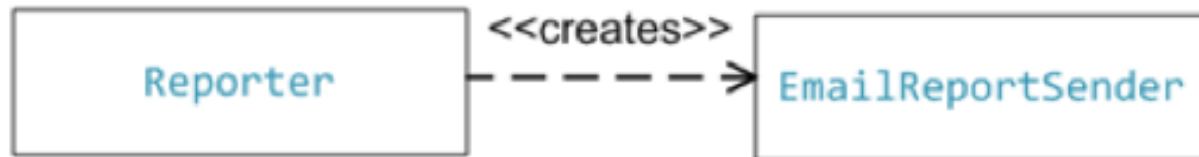
// ReportService public string

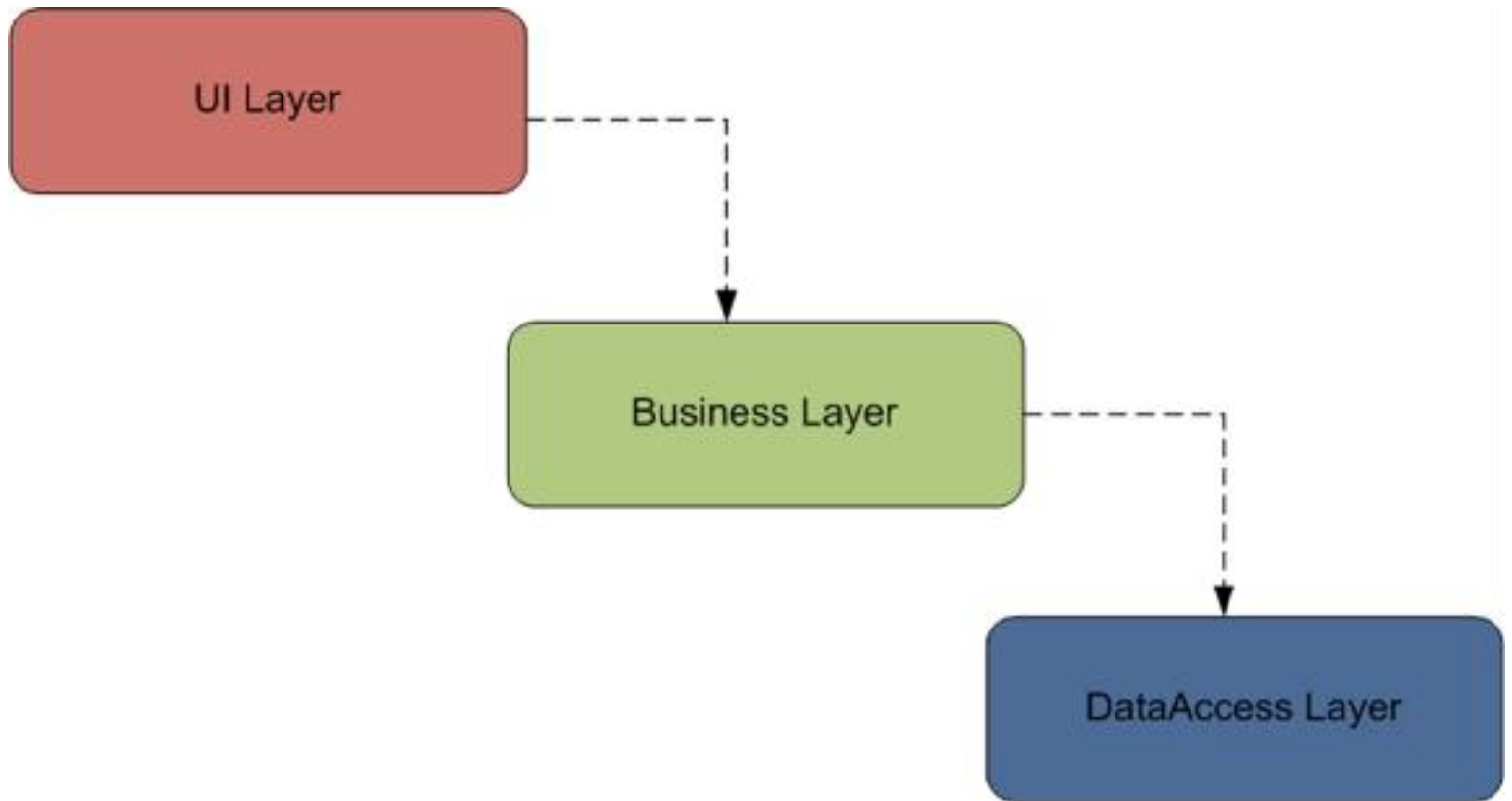
```
CreateReport(IReportFormatter reportFormatter)
{
    return default(string);
}
```

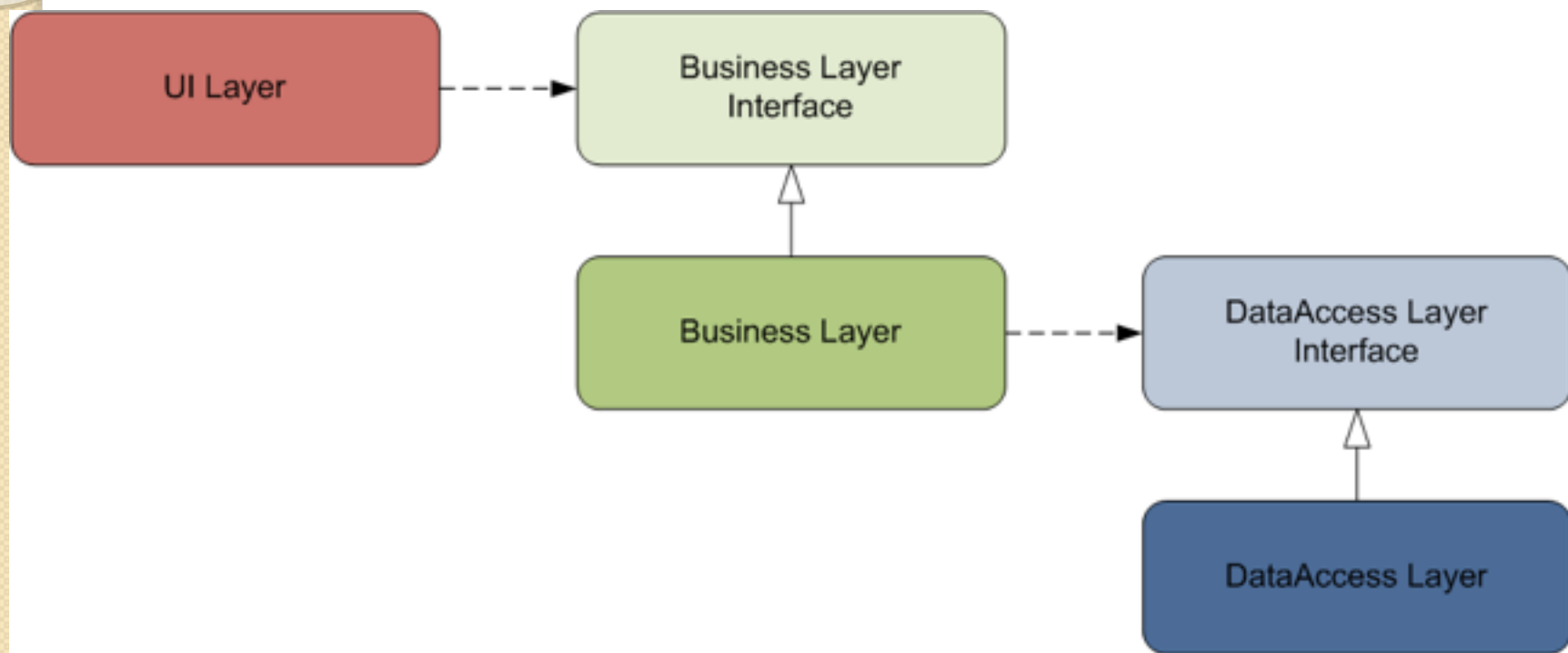
# Dependency Injection контейнеры

- StructureMap (AltDotNet)  
<http://structuremap.sourceforge.net/Default.htm>
- Castle Windsor (AltDotNet)  
<http://www.castleproject.org/container/index.html>
- Unity (Microsoft P&P)  
<http://www.codeplex.com/unity>
- Ninject (open source) <http://ninject.org>

# Dependency Injection контейнеры








# Dependency Injection in .NET

Mark Seemann  
Foreword by Guzik Bucci

 MANNING

LOOK  
INSIDE

