SOLID

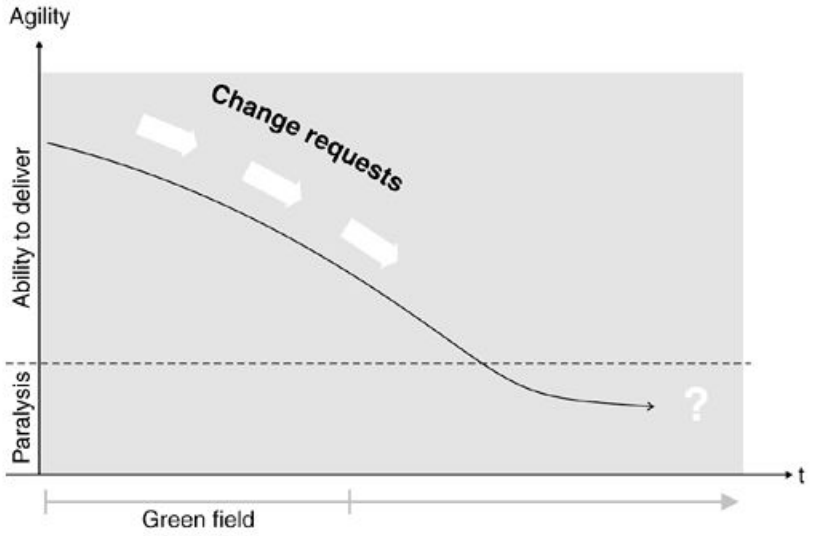
Принципы проектирования программ

О пользе проектирования

Изначально любая система обладает некоторой гибкостью (agility), другими словами, способностью к изменению. В процессе поступления новых требований от заказчика (change requests) ее гибкость падает до некоторого предела, называемого параличом (paralyses). Вход в стадию паралича означает:

- Вносить изменения стало слишком дорого;
- Вносить изменения стало невозможно из-за высокой сложности системы.

Паралича избежать невозможно, однако его можно максимально оттянуть, используя хорошую архитектуру.



S O L I D

S	Принцип единственной обязанности На каждый объект должна быть возложена одна единственная обязанность.
O	Принцип открытости/закрытости Программные сущности должны быть открыты для расширения, но закрыты для изменения.
L	Принцип подстановки Барбары Лисков Объекты в программе могут быть заменены их наследниками без изменения свойств программы.
I	Принцип разделения интерфейса Много специализированных интерфейсов лучше, чем один универсальный.
D	Принцип инверсии зависимостей Зависимости внутри системы строятся на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей, детали должны зависеть от абстракций.

Принцип единственной обязанности

Каждый интерфейс должен отвечать за что-то одно.

Индикатор ответственности – повод для изменений. Если есть несколько поводов для изменения, интерфейс совмещает несколько обязанностей.

Пример: класс для составления и печати отчета – должен быть один класс для составления и другой класс для печати.

То же относится и к методам – каждый метод должен делать что-то одно.

Принцип открытости-закрытости

Бертран Мейер

Модуль или класс должен быть закрыт для изменений и открыт для дополнений.
Иными словами, изменения программы должны происходить за счет дописывания нового кода, а не переписывания того, что уже работает.

Соблюсти принцип можно, добавляя новую функциональность в классы-наследники, при этом оставляя неизменными классы-предки (допускается и простое добавление новых методов в класс).

Роберт Мартин

Другая формулировка основана на использовании абстракций. **Класс A должен зависеть не от класса B, а от интерфейса IB, который реализуется классом B.**

Изменить поведение A можно, дав новую реализацию интерфейса IB, при этом код класса A не изменится.

Пример

```
class Figure
{
    public int X {set; get;}
    public int Y {set; get;}
}
```

Классы Фигура, Окружность, Прямоугольник и контейнер Рисунок.

Это **плохое** проектирование.

```
class Picture
{
  List<Figure> figures;

  public void Print()
  {
    foreach (var f in figures) {
      if (f is Rectangle)
       Console.Write((Rectangle)f.Info());

      if (f is Square)
       Console.Write((Square)f.Info());
    }

}
```

```
class Rectangle: Figure
{
    int w, h;
    public Rectangle(int x, int y, int w, int h)
    {
        X = x; Y = y; this.w = w; this.h = h;
    }
    public int S { get { return w * h; } }
    public string Info()
    {
        return string.Format("x={0} y={1} w={2} h={3} ",
           X, Y, w, h);
    }
}
```

```
class Square: Figure
{
    int w;

    public Square(int x, int y, int w)
    {
        X = x; Y = y; this.w = w;
    }
    public int S { get { return w * w; } }
    public string Info()
    {
        return string.Format("x={0} y={1} w={2}", X, Y, w);
    }
}
```

Пример (продолжение)

```
abstract class Figure
{
    public int X { set; get; }
    public int Y { set; get; }
    public int S { get; }
    public string Info();
}
```

Класс Picture зависит от абстракции. Изменение реализации конкретных фигур, даже появление новых фигур не меняют код Picture.

```
class Picture
{
    List<Figure> figures;

    public void PrintInfo()
    {
        foreach (var f in figures)
            Console.WriteLine(f.Info());
    }

    public int S()
    {
        return figures.Sum(f => f.S);
    }
}
```

```
class Rectangle : Figure
{
    int w, h;
    public Rectangle(int x, int y, int w, int h)
    {
        X = x; Y = y; this.w = w; this.h = h;
    }
    public override int S { get { return w * h; } }
    public override string Info()
    {
        return string.Format("x={0} y={1} w={2} h={3} ",
                             X, Y, w, h);
    }
}
```

```
class Square : Figure
{
    int w;

    public Square(int x, int y, int w)
    {
        X = x; Y = y; this.w = w;
    }
    public override int S { get { return w * w; } }
    public override string Info()
    {
        return string.Format("x={0} y={1} w={2}", X, Y, w);
    }
}
```

Принцип подстановки

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

Определение подтипа: **Тип S будет подтипом T тогда и только тогда, когда соблюдает принцип подстановки.**

Барбара Лисков

Язык C++ не поддерживает принцип подстановки, а C# и Java поддерживают.

Вопрос. Если при реализации интерфейса IList<T> метод Insert() будет добавлять не один, а два дубликата экземпляра, нарушится принцип подстановки?

Программирование по контракту

Контракт = Интерфейс + Предусловия
+ Постусловия
+ Инварианты

Предусловия – это ограничения, которые накладываются на входные параметры и внешние переменные методов, например, функция Gcd(a, b), которая находит НОД, требует, чтобы a >= 0, b >= 0 и a <=b.

Постусловия – это ограничения, которые накладываются на возвращаемые значения методов, выходные параметры, и состояние внешних переменных после завершения метода. Например, если r - наибольший общий делитель, то r > 0 && r <=b.

Инварианты – это условия, которые относятся к классу в целом и должны выполняться на всем протяжении жизни экземпляра класса. Например, в классе List объем захваченной памяти больше или равен объему памяти, занятому данными, а в классе SortedList данные к тому же всегда упорядочены. Инвариант – это и пред- и постусловие одновременно.
Кроме того, инварианты, как часть контракта, наследуются производными классами.

Контракты в .NET

http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx

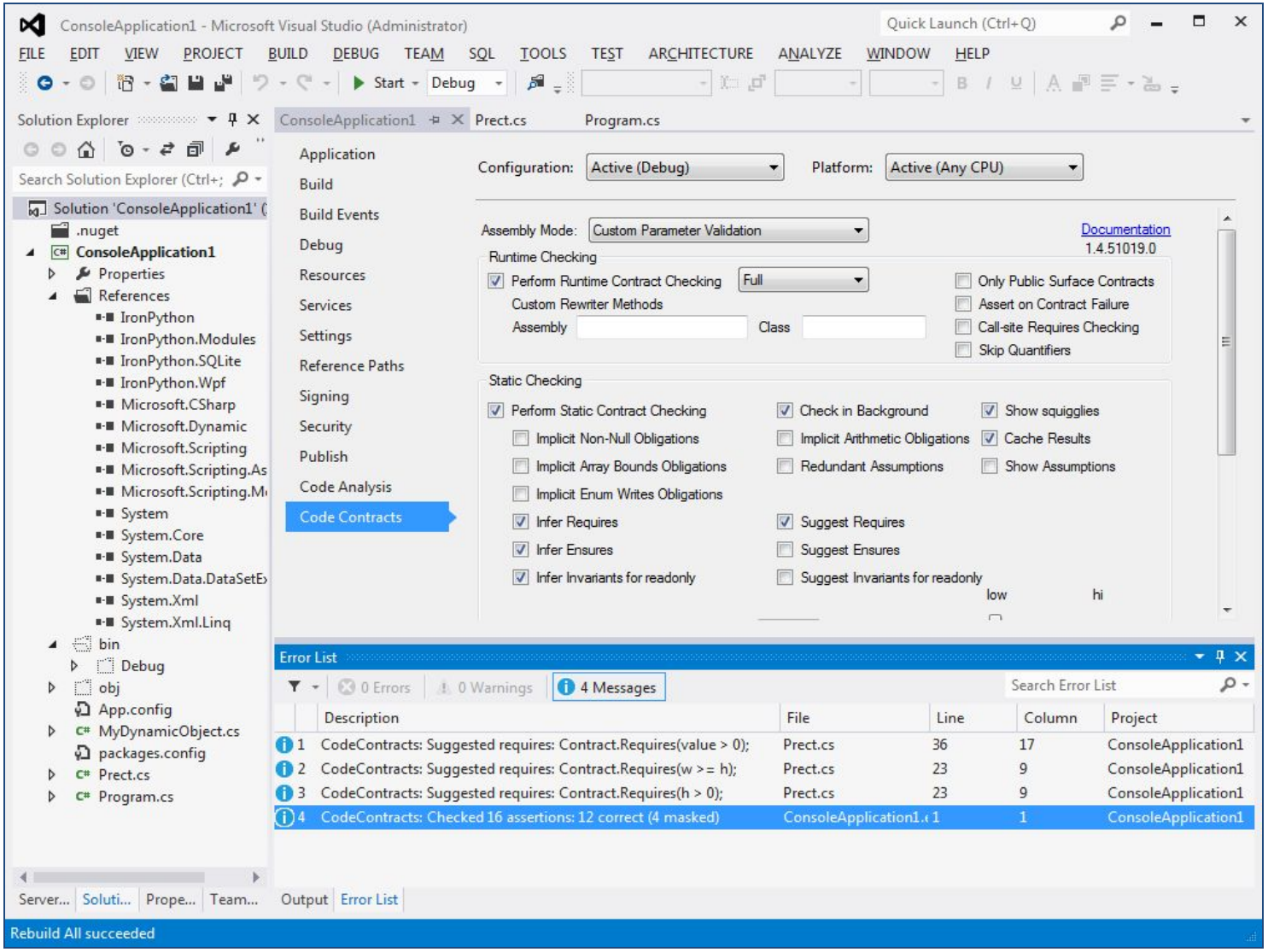
1. Библиотека и статический класс Contract.
2. Преобразователь кода – ccrewrite.exe.
3. Анализатор кода – cccheck.exe.

Первая часть это библиотека. Контракты кодируются с использованием вызова статических методов класса Contract (пространство имен System.Diagnostics.Contracts) из сборки mscorlib.dll. Контракты имеют декларативный характер, и эти статические вызовы в начале тела метода можно рассматривать как часть сигнатуры метода. Они являются методами, а не атрибутами, поскольку атрибуты очень ограничены, но эти концепции близки.

Вторая часть это binary rewriter, ccrewrite.exe. Этот инструмент модифицирует инструкции MSIL и проверяет контракт. Это инструмент дает возможность проверки выполнения контрактов, чтобы помочь при отладке кода. Без него, контракты просто документация и не включается в скомпилированный код.

Третья часть это инструмент статической проверки, cccheck.exe, который анализирует код без его выполнения и пытается доказать, что все контракты выполнены.

Code Contracts в Студии



Пример

```
// Прямоугольник с фиксированным периметром и шириной больше высоты.
    class MyRect
    {
        double p, w, h;
        [ContractInvariantMethod]
        void ObjectInvariant()
        {
            Contract.Invariant(p == w + h);
            Contract.Invariant(w >= h);
            Contract.Invariant(h > 0);
        }
        public MyRect(double w, double h)
        {
            this.w = w;
            this.h = h;
            p = w + h;
        }
        public double H
        {
            // Изменение высоты не должно превышать 10 единиц за раз.
            set {
                // Проверка в форме предусловия.
                Contract.Requires(Math.Abs(value - H) < 10.0, "Too large change.");
                h = value;
                w = p - h;
            }
            get { return h; }
        }
        public double W
        {
            // Изменение ширины не должно превышать 10 единиц за раз.
            set
            {
                // Проверка в форме постусловия.
                Contract.Ensures(Math.Abs(Contract.OldValue(w) - w) < 10.0, "Too large...");
                w = value;
                h = p - w;
            }
            get { return w; }
        }
        public double Square()
        {
            // Площадь всегда положительна.
            Contract.Ensures(Contract.Result<double>() > 0, "Squere must be positive");
```

Принцип разделения интерфейсов

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 будут меняться клиенты, которые этот метод не используют.

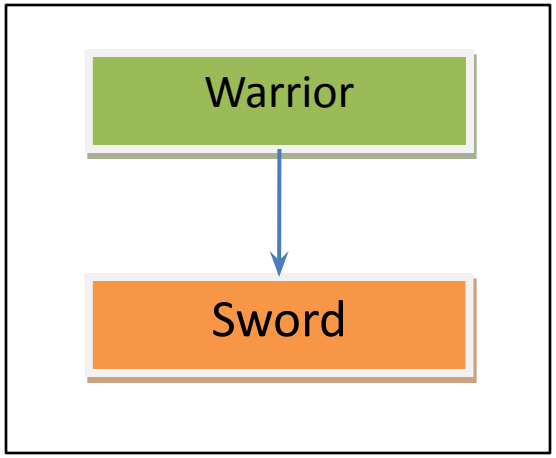
Принцип инверсии зависимостей

1. Зависимости внутри системы строятся на основе абстракций (т.е. интерфейсов).

2. Модули верхнего уровня не зависят от модулей нижнего уровня.

Пример зависимости

Есть два класса – Воин и Меч. Воин владеет мечем, а значит, зависит от него.




```
class Program
{
    static void Main()
    {
        Warrior warrior = new Warrior(new Sword());
        warrior.Kill();
    }
}
```

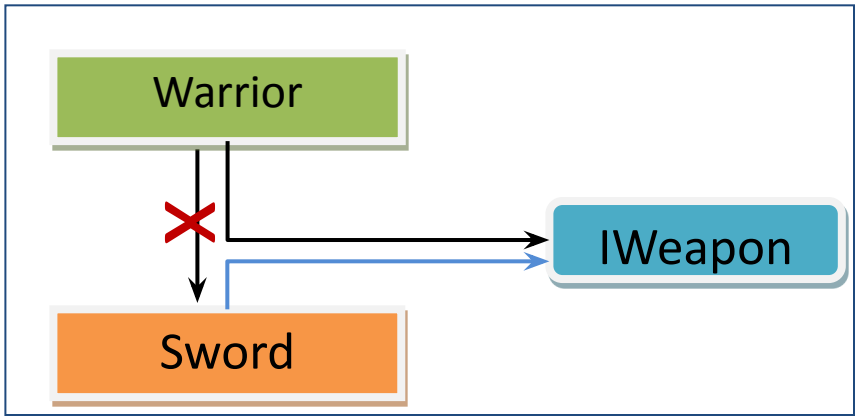
```
// Воин владеет оружием
public class Warrior
{
    readonly Sword weapon;

    // Оружие получает при рождении
    public Warrior(Sword weapon)
    {
        this.weapon = weapon;
    }
    // При помощи оружия может и убить
    public void Kill()
    {
        weapon.Kill();
    }
}
```

```
// Меч способен убивать
public class Sword
{
    public void Kill()
    {
        Console.WriteLine("Chuk-chuck");
    }
}
```

Инверсия зависимости

Воин зависит от абстракции.
И меч зависит от абстракции.
От меча воин не зависит.




```
// Оружие способно убивать
public interface IWeapon
{
    void Kill();
}
```

```
// Меч – оружие, поэтому способен убивать
public class Sword : IWeapon
{
    public void Kill()
    {
        Console.WriteLine("Chuk-chuck");
    }
}
```

```
// Базука – оружие, поэтому способна убивать
public class Bazuka : IWeapon
{
    public void Kill()
    {
        Console.WriteLine("BIG BADABUM!");
    }
}
```

Код после инверсии зависимости

```
public class Warrior
{
    readonly IWeapon weapon;

    public Warrior(IWeapon weapon)
    {
        this.weapon = weapon;
    }
    public void Kill()
    {
        weapon.Kill();
    }
}
```

```
public interface IWeapon
{
    void Kill();
}
```

```
public class Sword : IWeapon
{
    public void Kill()
    {
        Console.WriteLine("Chuk-chuck");
    }
}
```

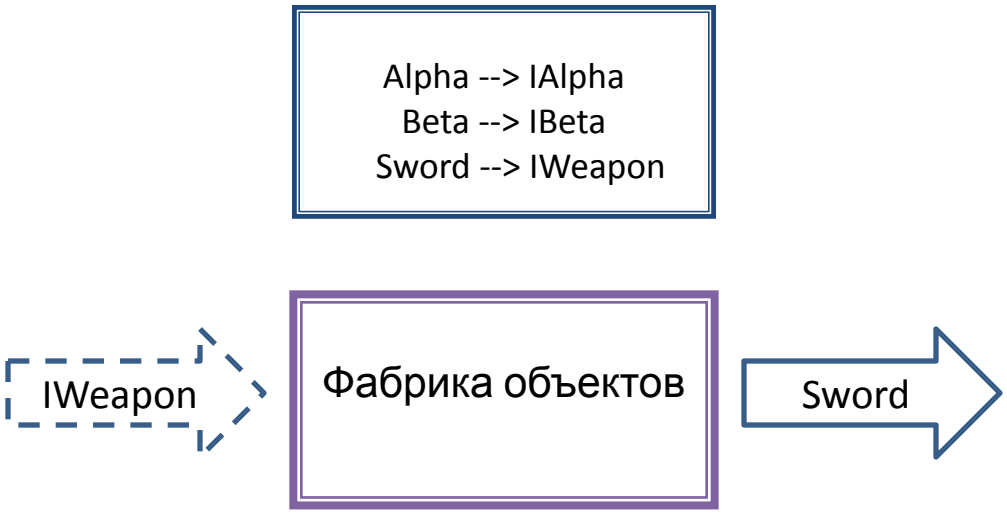
```
class Program
{
    static void Main()
    {
        Warrior warrior = new Warrior(new Sword());
        warrior.Kill();
    }
}
```

IoC-контейнер

IoC контейнер - это служба для управления созданием объектов.

Составные части контейнера:

1. Регистратор реализаций
2. Фабрика объектов



Пакет Ninject и его установка

Меню: TOOLS / Library Package Manager / Package Manager Console

или

Команда в PM-консоли: PM> Install-Package Ninject

Ninject в настольном приложении

```
using Ninject.Modules;

public class WeaponNinjectModule : NinjectModule
{
    public override void Load() {
        // Регистрация реализации
        this.Bind<IWeapon>().To<Sword>();
    }
}
```

```
using Ninject;

class Program
{
    public static IKernel AppKernel;

    static void Main()
    {
        // Фабрика объектов
        AppKernel = new StandardKernel(new WeaponNinjectModule());
        var warrior = AppKernel.Get<Warrior>();
    }
}
```

Атрибут [Inject]

Версия класса Warrior с полем weapon

```
public class Warrior
{
    readonly IWeapon weapon;

    public Warrior(IWeapon weapon)
    {
        this.weapon = weapon;
    }

    public void Kill()
    {
        weapon.Kill();
    }
}
```

Версия класса Warrior со свойством Weapon

```
public class Warrior
{
    [Inject]
    public IWeapon Weapon { set; get; }

    public void Kill()
    {
        Weapon.Kill();
    }
}
```

Ninject инициализирует автоматические свойства, если они открытые и помечены атрибутом [Inject]

Самостоятельно

```
class ScheduleManager
{
    public String GetSchedule()
    {
        return "1,2,3";
    }
}

class ScheduleViewer
{
    ScheduleManager _scheduleManager;

    public ScheduleViewer(ScheduleManager scheduleManager)
    {
        _scheduleManager = scheduleManager;
    }

    public string RenderSchedule()
    {
        return "<" + _scheduleManager.GetSchedule() + ">";
    }
}

class Program
{
    static void Main(string[] args) {
        ScheduleViewer sv = new ScheduleViewer(new ScheduleManager());
        Console.WriteLine(sv.RenderSchedule());
    }
}
```

Объект ScheduleViewer показывает расписание, придав ему эстетический вид.

Он получает расписание в сыром виде при помощи объекта ScheduleManager

Преобразуйте заданный код, внедрив зависимость от интерфейса и применив IoC-контейнер для создания объектов.