

Исходники в телеграме <http://ksergey.ru/profcsharp/>
<https://www.donationalerts.com/r/ksergei>

Первое видео

SOLID: SRP Принцип единственной ответственности (The Single Responsibility Principle) – Каждый класс должен иметь одну и только одну причину для изменений
<https://youtu.be/1nvfOcz-1Ss>

Второе видео

SOLID: OCP Принцип открытости/закрытости (The Open Closed Principle) - программные сущности должны быть открыты для расширения, но закрыты для модификации <https://youtu.be/tJkbThtl1D0>

Третье видео

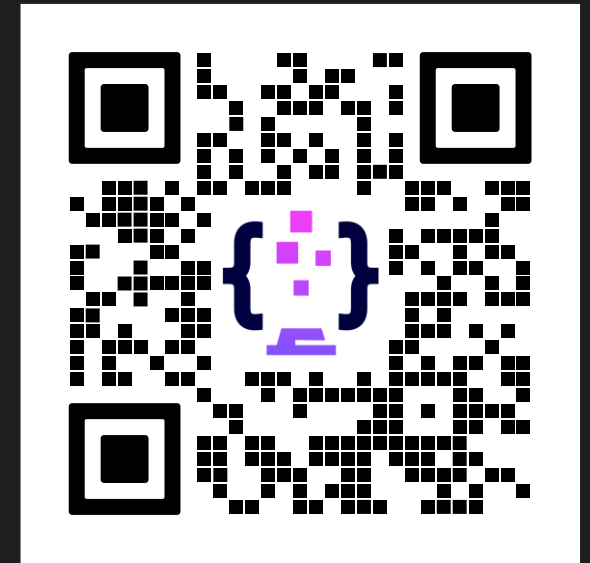
SOLID: LSP Принцип подстановки Барбары Лисков (The Liskov Substitution Principle) объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы <https://youtu.be/K0SHGO96C34>

Четвертое видео

SOLID: ISP Принцип разделения интерфейсов (The Interface Segregation Principle) https://youtu.be/_VCO4leZUd0

Пятое видео

SOLID: DIP Принцип инверсии зависимостей (The Dependency Inversion Principle) Зависимость на Абстракциях. Нет зависимости на что-то конкретное <https://youtu.be/u1IGDTinJv4>



Принципы SOLID

Сергей Камянецкий

Суть

Инкапсуляция сущностей с целью организации архитектуры приложения, которую будет легко поддерживать и расширять в течение всего промежутка эксплуатации

Предыстория

Роберт Мартин (*Дядя Боб*)

Консультант и автор в области разработки ПО

В 2001 организовал встречу группы, которая создала гибкую методологию разработки из техник экстремального программирования

**обобщил*

Предыстория

Роберт Мартин (*Дядя Боб*)

Консультант и автор в области разработки ПО

В 2001 организовал встречу группы, которая создала гибкую методологию разработки из техник экстремального программирования

**обобщил*

SOLID

Single responsibility — принцип единственной ответственности

Open-closed — принцип открытости / закрытости

Liskov substitution — принцип подстановки Барбары Лисков

Interface segregation — принцип разделения интерфейса

Dependency inversion — принцип инверсии зависимостей

SOLID: Single responsibility

Single responsibility — принцип единственной ответственности

SOLID: Single responsibility

Принцип единственной ответственности (англ. single-responsibility principle, SRP) — принцип ООП, обозначающий, что каждый объект должен иметь одну ответственность и эта ответственность должна быть полностью инкапсулирована в класс. Все его поведения должны быть направлены исключительно на обеспечение этой ответственности

SOLID: Single responsibility

Single responsibility — принцип единственной ответственности

SOLID: Single responsibility

Single responsibility — принцип единственной ответственности

Каждый объект должен иметь строго одну обязанность

SOLID: Single responsibility

Задача

Требуется разработать класс по работе с изображениями

SOLID: Single responsibility

```
public class Image
{
    private int width;
    private int height;
    public int Width => width;
    public int Height => height;
    private Image(int width, int height)
    public static Image CreateImage(int width, int height)

}
```

SOLID: Single responsibility

```
public class Image
{
    private int width;
    private int height;
    public int Width => width;
    public int Height => height;
    private Image(int width, int height)
    public static Image CreateImage(int width, int height)

    public ImageSize GetSize()

}
```

SOLID: Single responsibility

```
public class Image
{
    private int width;
    private int height;
    public int Width => width;
    public int Height => height;
    private Image(int width, int height)
    public static Image CreateImage(int width, int height)

    public ImageSize GetSize()
    public void SaveToFile(string path)

}
```

SOLID: Single responsibility

```
public class Image
{
    private int width;
    private int height;
    public int Width => width;
    public int Height => height;
    private Image(int width, int height)
    public static Image CreateImage(int width, int height)

    public ImageSize GetSize()
    public void SaveToFile(string path)
    public void SendToEmail(string email, string text, string subject)

}
```

SOLID: Single responsibility

```
public class Image
{
    private int width;
    private int height;
    public int Width => width;
    public int Height => height;
    private Image(int width, int height)
    public static Image CreateImage(int width, int height)

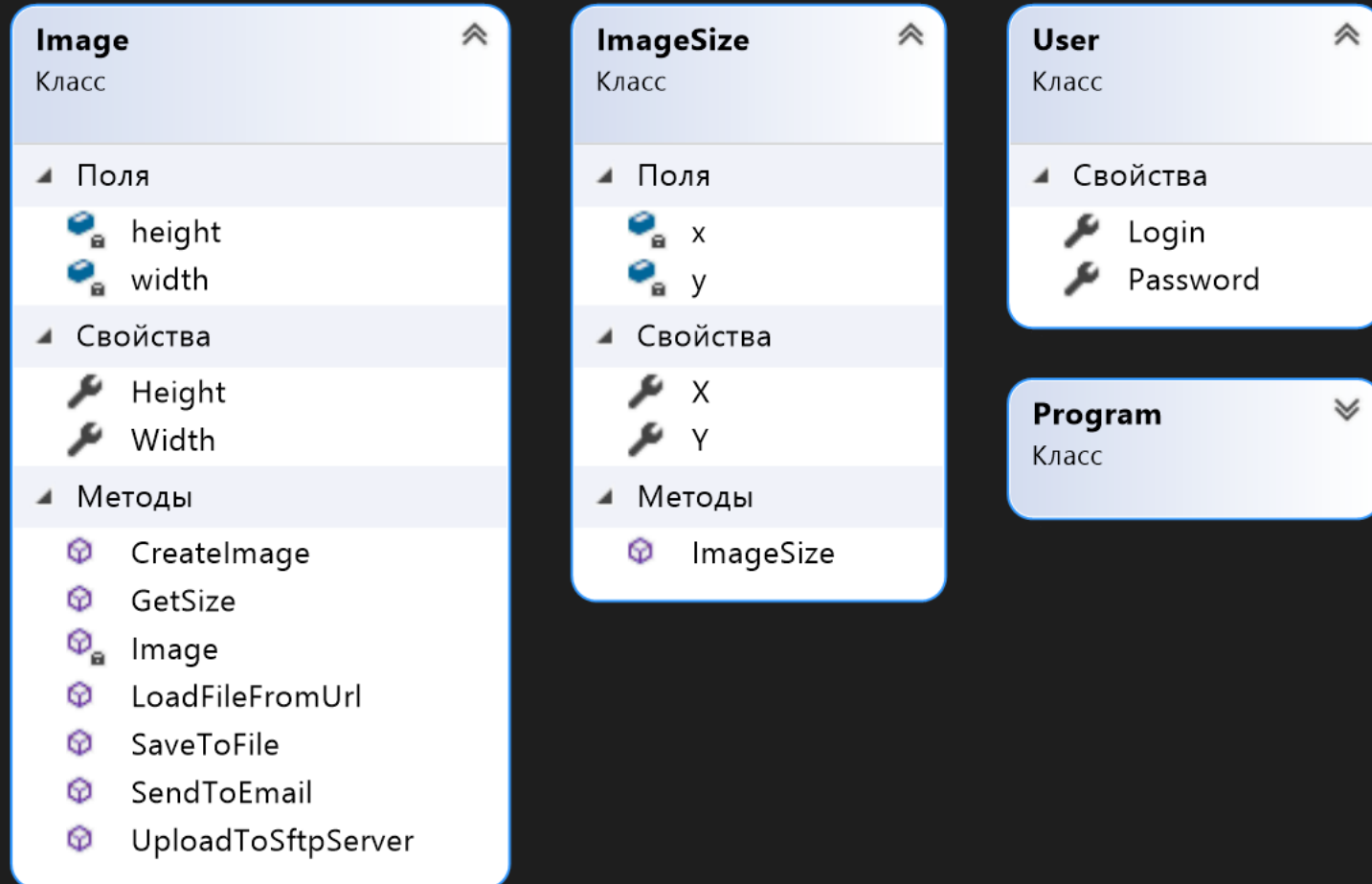
    public ImageSize GetSize()
    public void SaveToFile(string path)
    public void SendToEmail(string email, string text, string subject)
    public void UploadToSftpServer(string sftp, int port, Guid token, User user)
}
```


SOLID: Single responsibility

```
public class Image
{
    private int width;
    private int height;
    public int Width => width;
    public int Height => height;
    private Image(int width, int height)
    public static Image CreateImage(int width, int height)

    public ImageSize GetSize()
    public void SaveToFile(string path)
    public void SendToEmail(string email, string text, string subject)
    public void UploadToSftpServer(string sftp, int port, Guid token, User user)
    public void LoadFileFromUrl(string url)
}
```

SOLID: Single responsibility



SOLID: Single responsibility

Следовать принципу — декомпозировать

SOLID: Single responsibility

```
public class Image : Attach
{
    private int width;
    private int height;
    public int Width => width;
    public int Height => height;
    private Image(int width, int height)
    public static Image CreateImage(int width, int height)
    public ImageSize GetSize()
    public void SaveToFile(string path)
}
```

SOLID: Single responsibility

```
public class EmailService
{
    private string email;
    private string text;
    private string subject;
    private Attach[] attach;
    public EmailService(string email,
                        string text = "",
                        string subject= "",
                        params Attach[] args)
    public void SendTo(string email, string text, string subject)
}
```

SOLID: Single responsibility

```
public class SftpService
{
    private readonly string ftp;
    private readonly int port;
    private readonly User user;
    private readonly Attach[] attach;
    private readonly Guid key;

    public SftpService(string ftp,
                       int port,
                       User user,
                       Guid token,
                       params Attach[] args)

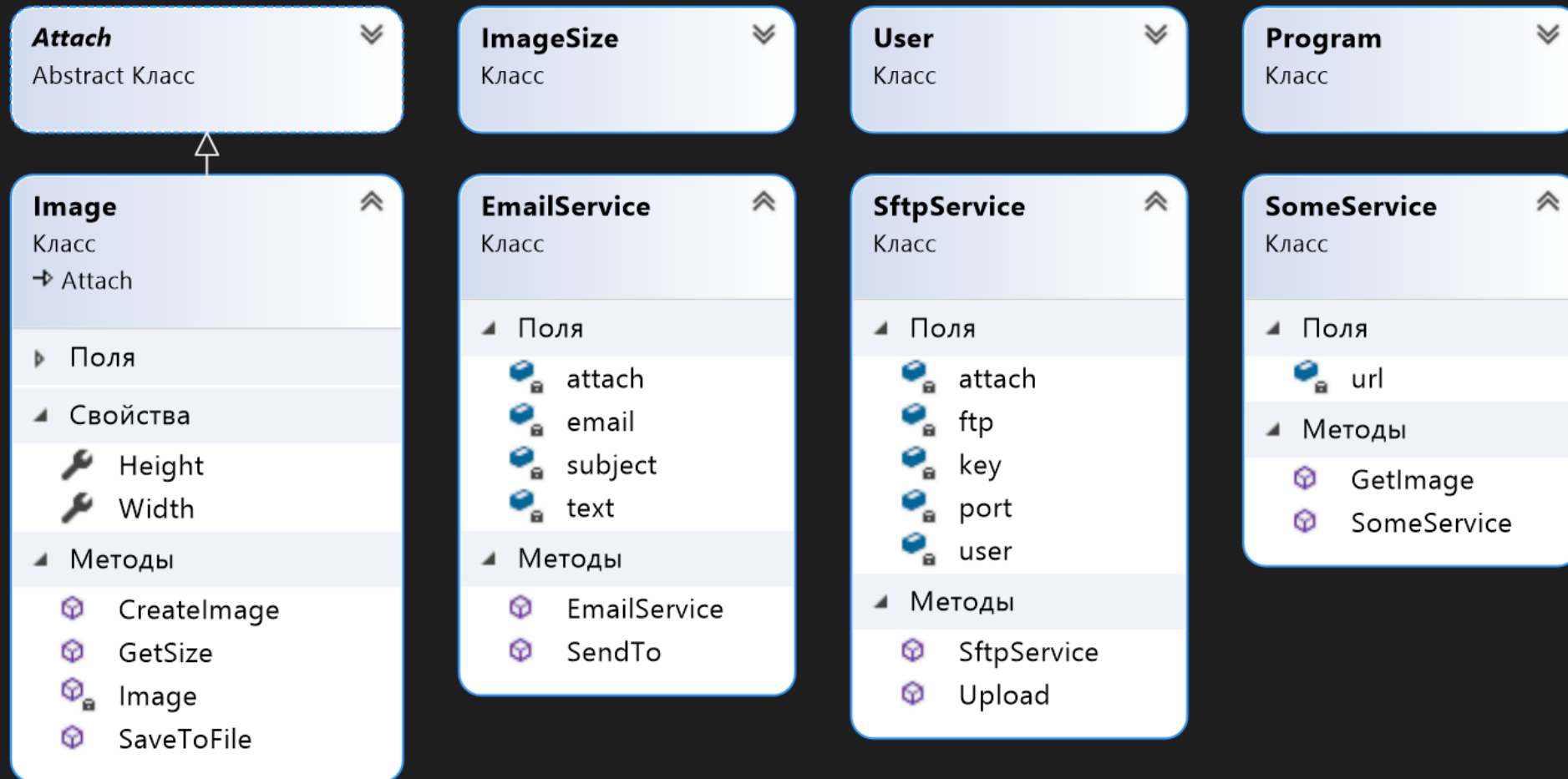
    public void Upload()
}
```

SOLID: Single responsibility

```
public class SomeService
{
    private string url;

    public SomeService(string url)
    public Image GetImage()
}
```

SOLID: Single responsibility



SOLID: Single responsibility. Замечание

Следовать принципу — декомпозировать

Объединять схожий функционал в одну сущность

не прощаюсь
все материалы
продолжаем обсуждение
<http://ksergey.ru/profcsharp/>



не прощаюсь
все материалы
продолжаем обсуждение
<http://ksergey.ru/profcsharp/>



Profcsharp

20 подписчиков

Принципы SOLID

Сергей Камянецкий

SOLID

Single responsibility — принцип единственной ответственности

Open-closed — принцип открытости / закрытости

Liskov substitution — принцип подстановки Барбары Лисков

Interface segregation — принцип разделения интерфейса

Dependency inversion — принцип инверсии зависимостей

SOLID: Open-closed

Open-closed — принцип открытости / закрытости

SOLID: Open-closed

Open-closed — принцип открытости / закрытости

Сформулирован Бертраном Мейером

SOLID: Open-closed

Open-closed — принцип открытости / закрытости

Сформулирован Бертраном Мейером

Каждая сущность закрыта для изменения
Нельзя вносить изменения в код, который используется

SOLID: Open-closed

Open-closed — принцип открытости / закрытости

Сформулирован Бертраном Мейером

Каждая сущность закрыта для изменения
Нельзя вносить изменения в код, который используется

Сущность открыта для модификаций
Поведение сущности может быть расширено

SOLID: Open-closed

```
public class Image : Attach
{
    private int width;
    private int height;
    public int Width => width;
    public int Height => height;
    private Image(int width, int height)
    public static Image CreateImage(int width, int height)
    public ImageSize GetSize()
    public void SaveToFile(string path)
}
```

SOLID: Open-closed

```
public class Image : Attach
{
    private int width;
    private int height;
    public int Width => width;
    public int Height => height;
    private Image(int width, int height)
    public static Image CreateImage(int width, int height)
    public ImageSize GetSize()
    public void SaveToFile(string path)
}
```

SOLID: Open-closed

```
public class Image : Attach
{
    private int width;
    private int height;
    public int Width => width;
    public int Height => height;
    private Image(int width, int height)
    public static Image CreateImage(int width, int height)
    public ImageSize GetSize()
    public void SaveToBMP(string path)
}
```

SOLID: Open-closed

```
public class Image : Attach
{
    private int width;
    private int height;
    public int Width => width;
    public int Height => height;
    private Image(int width, int height)
    public static Image CreateImage(int width, int height)
    public ImageSize GetSize()
    public void SaveToBMP(string path)
    public void SaveToJPG(string path)
}
```

SOLID: Open-closed

```
public class Image : Attach
{
    private int width;
    private int height;
    public int Width => width;
    public int Height => height;
    private Image(int width, int height)
    public static Image CreateImage(int width, int height)
    public ImageSize GetSize()
    public void SaveToBMP(string path)
    public void SaveToJPG(string path)
    public void SaveToPNG(string path)
}
```

SOLID: Open-closed

Как быть?

SOLID: Open-closed

Как быть?

```
public interface ISave
{
    void Save(string path, Image image);
}
```

SOLID: Open-closed

Как быть?

```
public interface ISave
{
    void Save(string path, Image image);
}
```

```
public abstract class SaveLogic
{
    void Save(string path, Image image);
}
```

SOLID: Open-closed

SOLID: Open-closed

```
public class SaveToBMP : ISave
{
    public void Save(string path, Image image)
}
```

SOLID: Open-closed

```
public class SaveToBMP : ISave
{
    public void Save(string path, Image image)
}
```

```
public class SaveToJPG : ISave
{
    public void Save(string path, Image image)
}
```

SOLID: Open-closed

```
public class SaveToBMP : ISave
{
    public void Save(string path, Image image)
}
```

```
public class SaveToJPG : ISave
{
    public void Save(string path, Image image)
}
```

```
public class SaveToPNG : ISave
{
    public void Save(string path, Image image)
}
```

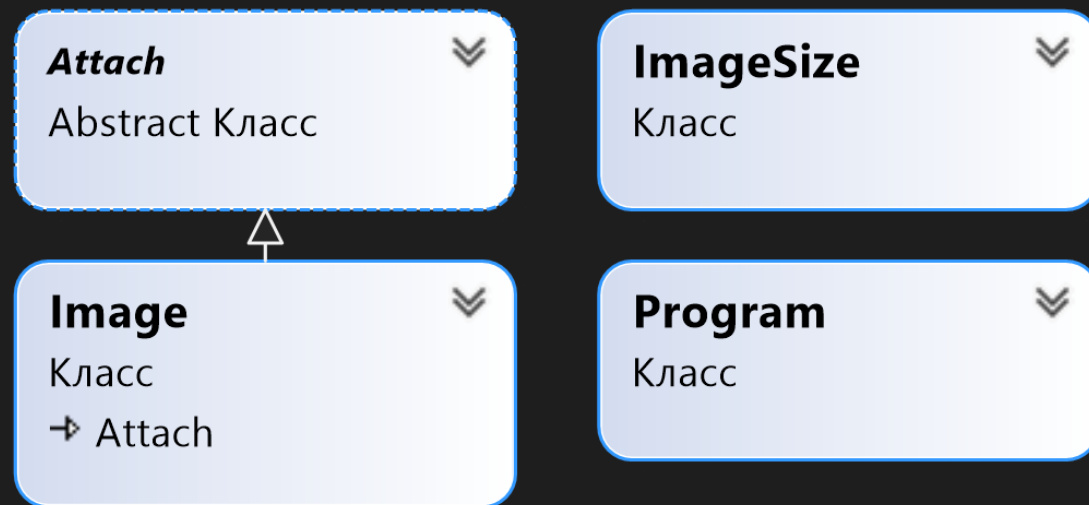
SOLID: Open-closed

Было

```
Image image = Image.CreateImage(28, 09);  
image.SaveToBMP("image.bmp");  
image.SaveToJPG("image.jpg");  
image.SaveToPNG("image.png");
```

SOLID: Open-closed

Было



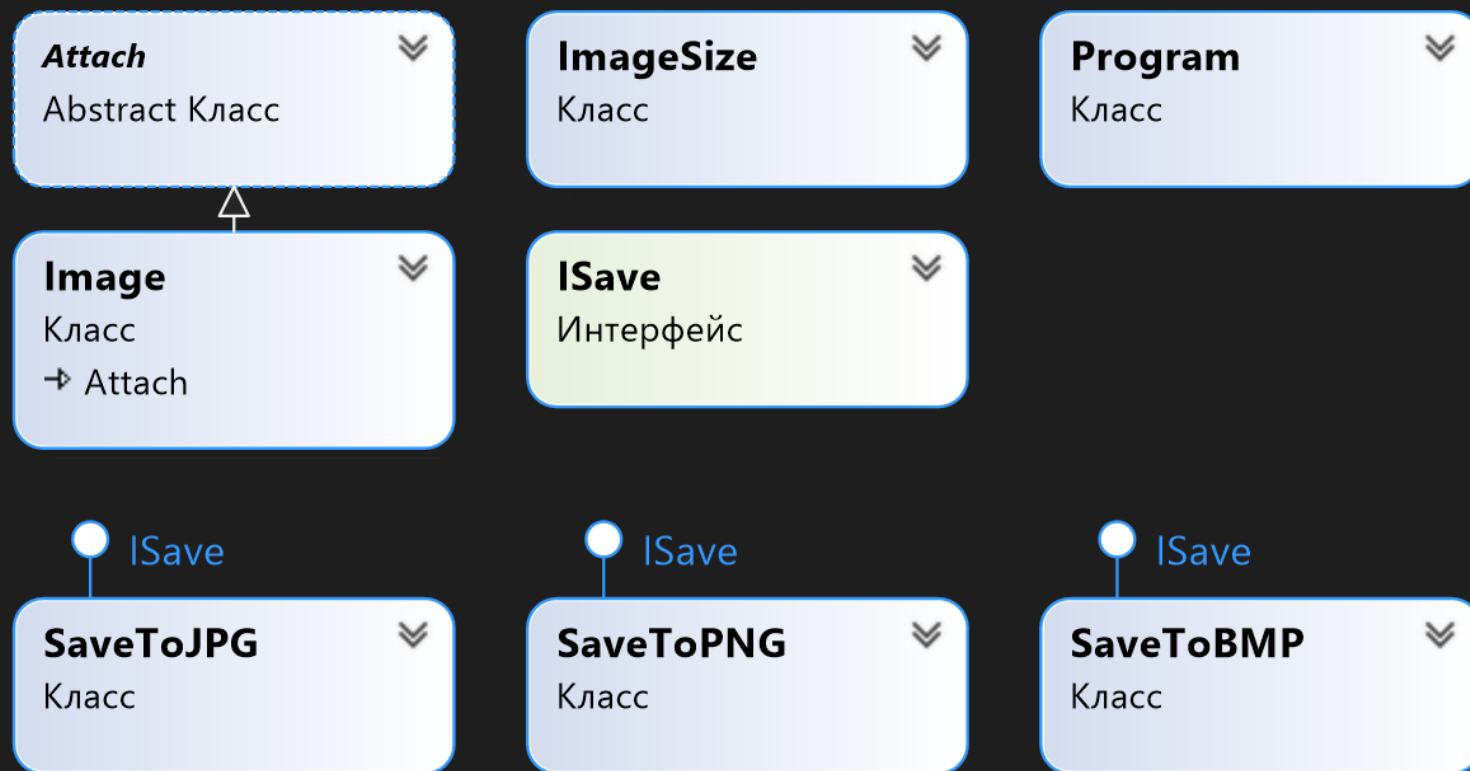
SOLID: Open-closed

Стало

```
Image[] pictures = new[]  
{  
    Image.CreateImage(28,09,new SaveToBMP()),  
    Image.CreateImage(19,90,new SaveToJPG()),  
    Image.CreateImage(15,06,new SaveToPNG()),  
};  
  
foreach (var pic in pictures)  
    pic.SaveTo($"filename_{DateTime.Now.Ticks}");
```

SOLID: Open-closed

Стало



SOLID: Open-closed

Методы расширения

SOLID: Open-closed

Методы расширения

```
public class Image : Attach
{
    private int width;
    private int height;
    public int Width => width;
    public int Height => height;
    private Image(int width, int height)
    public static Image CreateImage(int width, int height)
    public ImageSize GetSize()
}
```

SOLID: Open-closed

Методы расширения

```
static public class ImageExtensions
{
    public static void SaveToBMP(this Image image, string path)
    public static void SaveToJPG(this Image image, string path)
    public static void SaveToPNG(this Image image, string path)
}
```

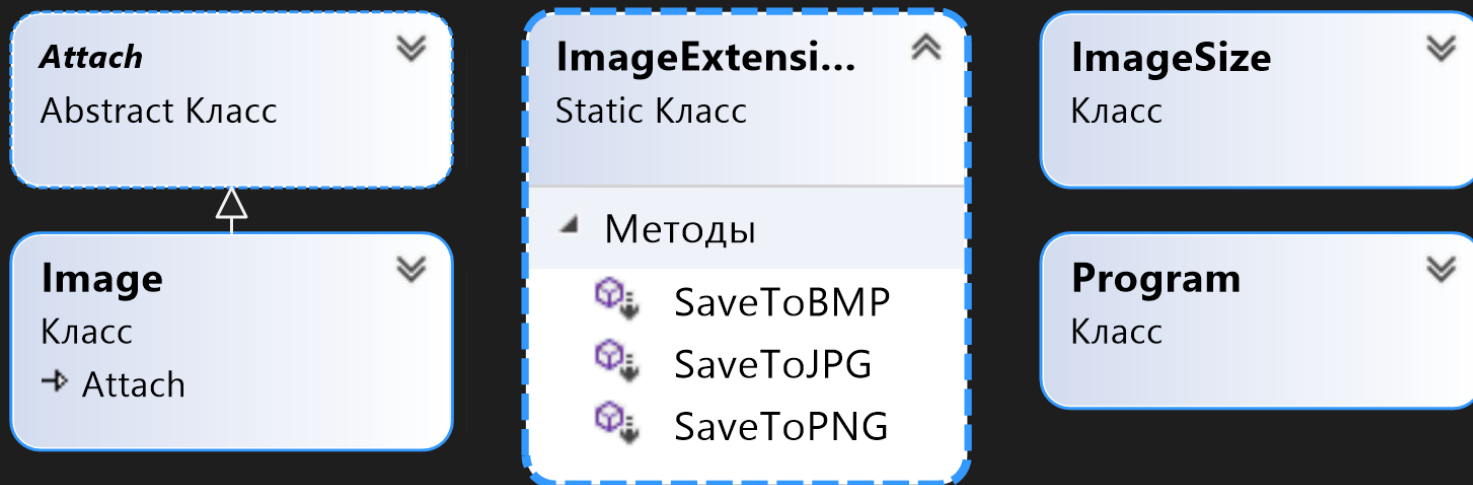
SOLID: Open-closed

Методы расширения

```
Image image = Image.CreateImage(28, 09);  
image.SaveToBMP("image.bmp");  
image.SaveToJPG("image.jpg");  
image.SaveToPNG("image.png");
```

SOLID: Open-closed

Методы расширения



не прощаюсь
все материалы
продолжаем обсуждение
<http://ksergey.ru/profcsharp/>



Принципы SOLID

Сергей Камянецкий

SOLID

Single responsibility — принцип единственной ответственности

Open-closed — принцип открытости / закрытости

Liskov substitution — принцип подстановки Барбары Лисков

Interface segregation — принцип разделения интерфейса

Dependency inversion — принцип инверсии зависимостей

SOLID: Liskov substitution

Liskov substitution — принцип подстановки Барбары Лисков

SOLID: Liskov substitution

Liskov substitution — принцип подстановки Барбары Лисков

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

Поведение классов-наследников не должно противоречить поведению, заданному базовым классом

SOLID: Liskov substitution

Liskov substitution — принцип подстановки Барбары Лисков

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

Поведение классов-наследников не должно противоречить поведению, заданному базовым классом

Пусть $Q(x)$ является свойством, верным относительно объектов x некоторого типа T . Тогда $Q(y)$ также должно быть верным для объектов y типа S , где S является подтипом типа T

SOLID: Liskov substitution

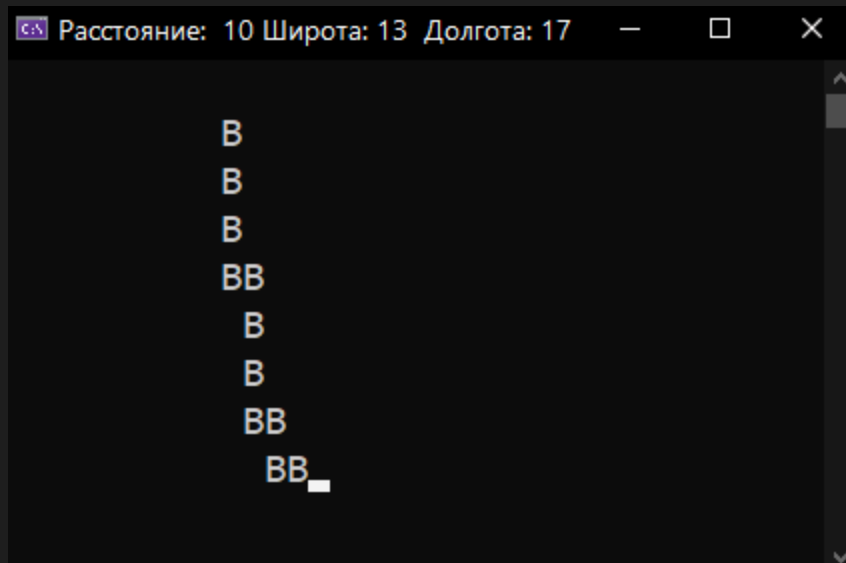
```
public class Bird
{
    public Coordinates Position { get; set; }
    protected int speed, spacing;
    public Bird() {...}
    protected void Mark(int x, int y) {...}
    private void Fly()
    {
        speed = 1;
        switch (rand.Next(2))
        {
            case 0: Position.Latitude += speed; break;
            default: Position.Longitude += speed; break;
        }
        spacing++;
    }
}
```

SOLID: Liskov substitution

```
public class CalculatingDistance
{
    int time;
    public CalculatingDistance(int time) {...}
    public void Calculate(Bird bird)
    {
        for (int i = 0; i < time; i++)
        {
            bird.Fly();
        }
        Console.Title = ($"\n\n\nРасстояние: {bird.Spacing} {bird.Position}");
    }
}
```


SOLID: Liskov substitution

```
Bird bird = new Bird();  
CalculatingDistance dist = new CalculatingDistance(10);  
dist.Calculate(bird);
```



SOLID: Liskov substitution

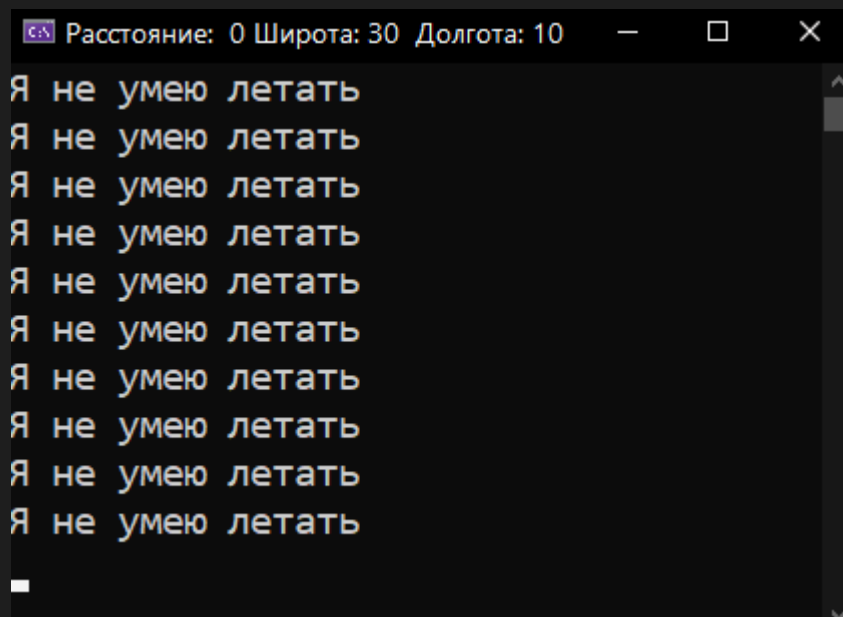
```
public class Kiwi : Bird
{
    public Kiwi() {...}
    public override void Fly() { Console.WriteLine("Я не умею летать"); }
    public void Run()
    {
        speed = 1;
        switch (rand.Next(4))
        {
            case 0: Position.Latitude += speed; break;
            case 1: Position.Latitude -= speed; break;
            case 2: Position.Longitude += speed; break;
            default: Position.Longitude -= speed; break;
        }
        spacing++;
        Mark(Position.Latitude, Position.Longitude);
    }
}
```

SOLID: Liskov substitution

```
public class CalculatingDistance
{
    int time;
    public CalculatingDistance(int time) {...}
    public void Calculate(Bird bird)
    {
        for (int i = 0; i < time; i++)
        {
            bird.Fly();
        }
        Console.Title = ($"\n\n\nРасстояние: {bird.Spacing} {bird.Position}");
    }
}
```

SOLID: Liskov substitution

```
Kiwi kiwi = new Kiwi();  
CalculatingDistance dist = new CalculatingDistance(10);  
dist.Calculate(kiwi);
```



SOLID: Liskov substitution

Решение — правильное проектирование

SOLID: Liskov substitution

Решение — правильное проектирование

```
public class Bird
{
    public Bird() {...}
    protected void Mark(int x, int y) {...}
    private void Fly() {...}
    public virtual void Move()
    {
        this.Fly();
    }
}
```

SOLID: Liskov substitution

Решение — правильное проектирование

```
public class CalculatingDistance
{
    int time;
    public CalculatingDistance(int time) {...}
    public void Calculate(Bird bird)
    {
        for (int i = 0; i < time; i++)
        {
            bird.Move();
        }
        Console.Title = ($"\n\nРасстояние: {bird.Spacing} {bird.Position}");
    }
}
```

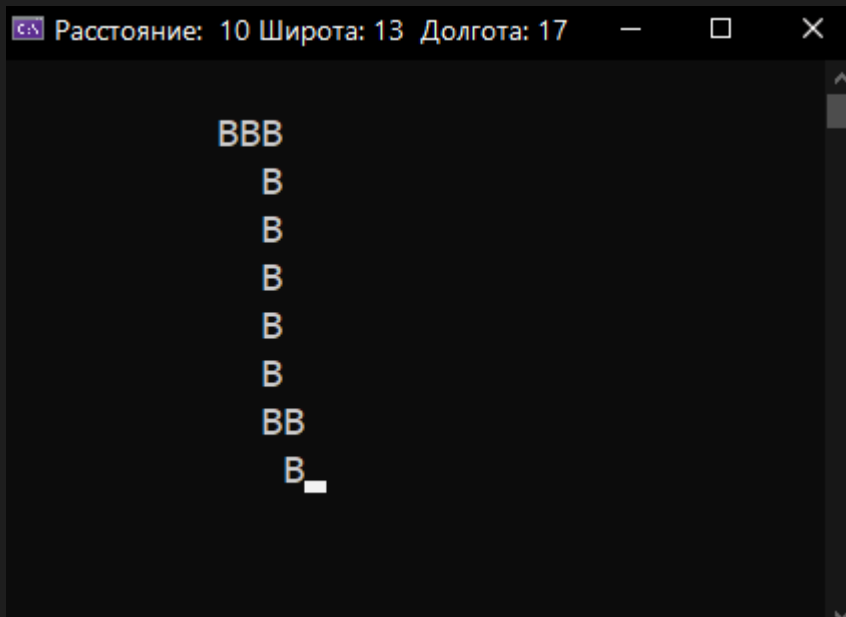
SOLID: Liskov substitution

Решение — правильное проектирование

```
public class Kiwi : Bird
{
    public Kiwi() {...}
    public override void Fly() { Console.WriteLine("Я не умею летать"); }
    private void Run() {...}
    public override void Move()
    {
        this.Run();
    }
}
```

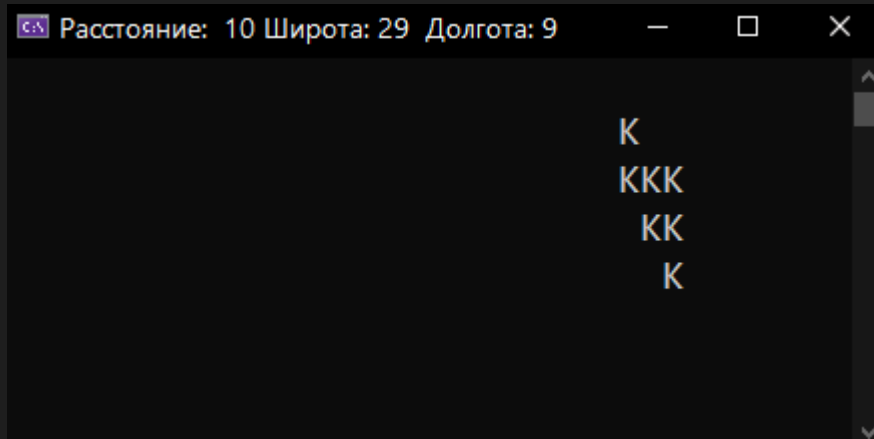

SOLID: Liskov substitution

```
Bird bird = new Bird();  
CalculatingDistance dist = new CalculatingDistance(10);  
dist.Calculate(bird);
```



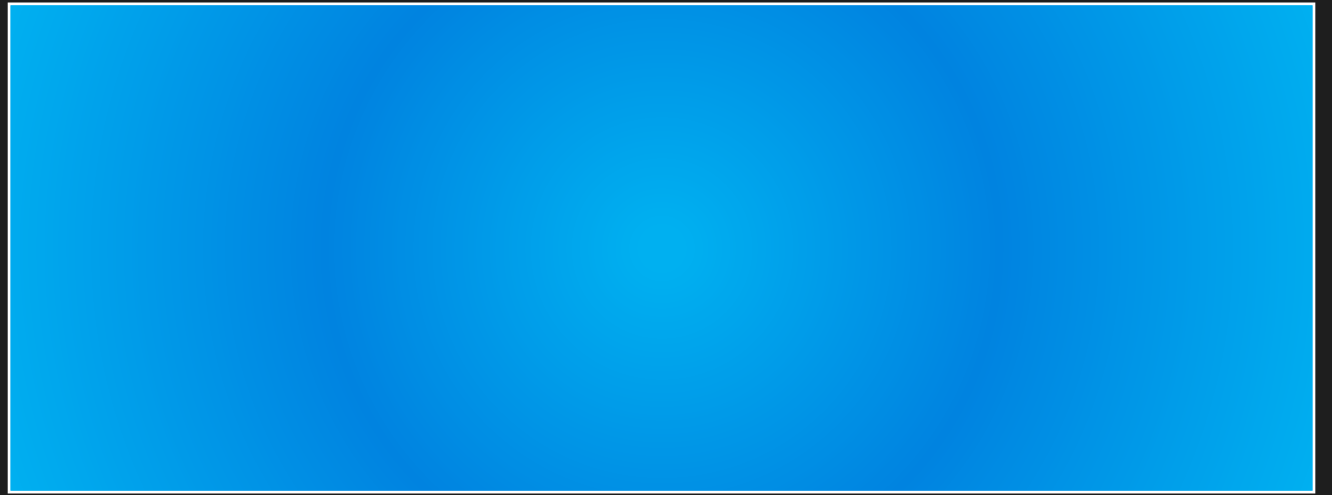
SOLID: Liskov substitution

```
Kiwi kiwi = new Kiwi();  
CalculatingDistance dist = new CalculatingDistance(10);  
dist.Calculate(kiwi);
```

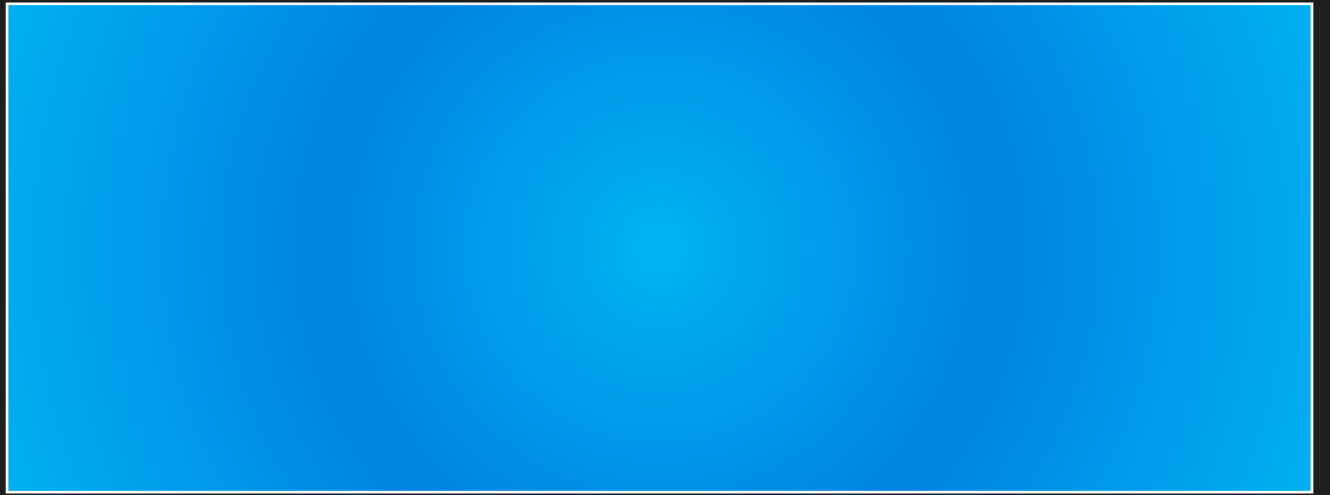
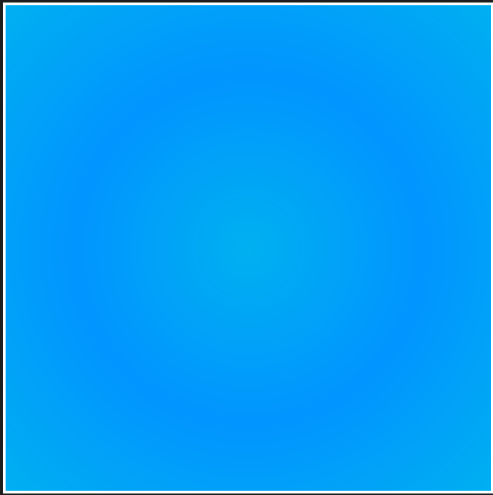


SOLID: Liskov substitution. Классика

SOLID: Liskov substitution. Классика



SOLID: Liskov substitution. Классика



SOLID: Liskov substitution. Вопросик

Ваш пример с сохранением изображения очень простой и в принципе при разработке этого класса можно предвидеть подобные изменения и сразу вынести сохранение отдельно с использованием интерфейса.

Но на реальных проектах часто классы гораздо сложнее и отсюда вопрос: согласны ли вы с тем, что

**не стоит сразу выносить все и вся с целью удовлетворить принципу,
а написать сначала простое и понятное решение без лишних абстракций**

**и только в будущем когда станет ясно, что класс действительно часто меняется
переработать его так чтобы он начал удовлетворять этому принципу?**

Конечно если на этапе проектирования нельзя четко сказать, что это ТОЧНО будет меняться и мы не хотим делать абстракции ради абстракций.

SOLID: Liskov substitution. Классика

```
public class Image : Attach
{
    private int width;
    private int height;
    public int Width => width;
    public int Height => height;
    private Image(int width, int height)
    public static Image CreateImage(int width, int height)
    public ImageSize GetSize()
    public void SaveToFile(string path)
}
```

SOLID: Liskov substitution. Классика

```
public class Image : Attach
{
    private int width;
    private int height;
    public int Width => width;
    public int Height => height;
    private Image(int width, int height)
    public static Image CreateImage(int width, int height)
    public ImageSize GetSize()
    public void SaveToFile(string path)
}

public class ImaginaryImage : Image { ... }
```


SOLID: Liskov substitution. Классика

```
public class Image : Attach
{
    private int width;
    private int height;
    public int Width => width;
    public int Height => height;
    private Image(int width, int height)
    public static Image CreateImage(int width, int height)
    public ImageSize GetSize()
    public void SaveToFile(string path)
}
```

```
public class ImaginaryImage : Image { ... }
```

```
public class RealImage : Image { ... }
```

не прощаюсь
все материалы
продолжаем обсуждение
<http://ksergey.ru/profcsharp/>



Принципы SOLID

Сергей Камянецкий

SOLID

Single responsibility — принцип единственной ответственности

Open-closed — принцип открытости / закрытости

Liskov substitution — принцип подстановки Барбары Лисков

Interface segregation — принцип разделения интерфейса

Dependency inversion — принцип инверсии зависимостей

SOLID: Interface segregation

Interface segregation — принцип разделения интерфейса

SOLID: Interface segregation

Interface segregation — принцип разделения интерфейса

суть принципа заключается в том, что клиенты не должны зависеть от методов, которые они не используют

SOLID: Interface segregation

Interface segregation — принцип разделения интерфейса

суть принципа заключается в том, что клиенты не должны зависеть от методов, которые они не используют

Следовать принципу — значит использовать необходимый минимум реализаций методов

SOLID: Interface segregation

Interface segregation — принцип разделения интерфейса

суть принципа заключается в том, что клиенты не должны зависеть от методов, которые они не используют

Следовать принципу — значит использовать необходимый минимум реализаций методов

От реализации избыточных интерфейсов следует отказаться в пользу более специфичных

SOLID: Interface segregation

SOLID: Interface segregation

```
public abstract class Car
{
    public string Model { get; set; }
    public string Brand { get; set; }

    public Car(string model, string brand)
    {
        this.Model = model;
        this.Brand = brand;
    }
}
```

SOLID: Interface segregation

```
public abstract class Car
{
    public string Model { get; set; }
    public string Brand { get; set; }

    public Car(string model, string brand)
    {
        this.Model = model;
        this.Brand = brand;
    }
}

public abstract class Lada : Car
{
    public Lada(string model, string brand)
        : base(model, brand) { }
}
```

SOLID: Interface segregation

```
public abstract class Lada : Car
{
    public Lada(string model, string brand)
        : base(model, brand) { }
}

public abstract class Vesta : Lada
{
    public Vesta(string model, string brand)
        : base(model, brand) { }
}
```

SOLID: Interface segregation

```
public abstract class Vesta : Lada
{
    public Vesta(string model, string brand)
        : base(model, brand) { }
}

public interface IVesta
{
    Characteristic BodyWheelArrangementTractionWheels { get; }
    Characteristic BodyEngineLocation { get; }
    Characteristic BodyTypeNumberOfDoors { get; }
    Characteristic Bodyseats { get; }
    Characteristic BodyLengthWidthHeight { get; }
    Characteristic BodyWheelbase { get; }
    Characteristic BodyFrontRearTrack { get; }
    Characteristic BodyRoadClearance { get; }
    Characteristic BodyRearTrunkCapacity { get; }
    Characteristic SuspensionFront { get; }
    Characteristic SuspensionBack { get; }
    Characteristic SteeringControlSteeringMechanism { get; }
    Characteristic TiresGeneralProportions { get; }
}
```

SOLID: Interface segregation

```
public class Vesta122hp : Vesta, IVesta
{
    public Vesta122hp(string model, string brand)
        : base(model, brand) { }

    public Characteristic BodyWheelArrangementTractionWheels => ImplementedSomehow;
    public Characteristic BodyEngineLocation => ImplementedSomehow;
    public Characteristic BodyTypeNumberOfDoors => ImplementedSomehow;
    public Characteristic BodySeats => ImplementedSomehow;
    public Characteristic BodyLengthWidthHeight => ImplementedSomehow;
    public Characteristic BodyWheelbase => ImplementedSomehow;
    public Characteristic BodyFrontRearTrack => ImplementedSomehow;
    public Characteristic BodyRoadClearance => ImplementedSomehow;
    public Characteristic BodyRearTrunkCapacity => ImplementedSomehow;
    public Characteristic SuspensionFront => ImplementedSomehow;
    public Characteristic SuspensionBack => ImplementedSomehow;
    public Characteristic SteeringControlSteeringMechanism => ImplementedSomehow;
    public Characteristic TiresGeneralProportions => ImplementedSomehow;
}
```

SOLID: Interface segregation

```
public class Vesta113hp : Vesta, IVesta
{
    public Vesta113hp(string model, string brand)
        : base(model, brand) { }

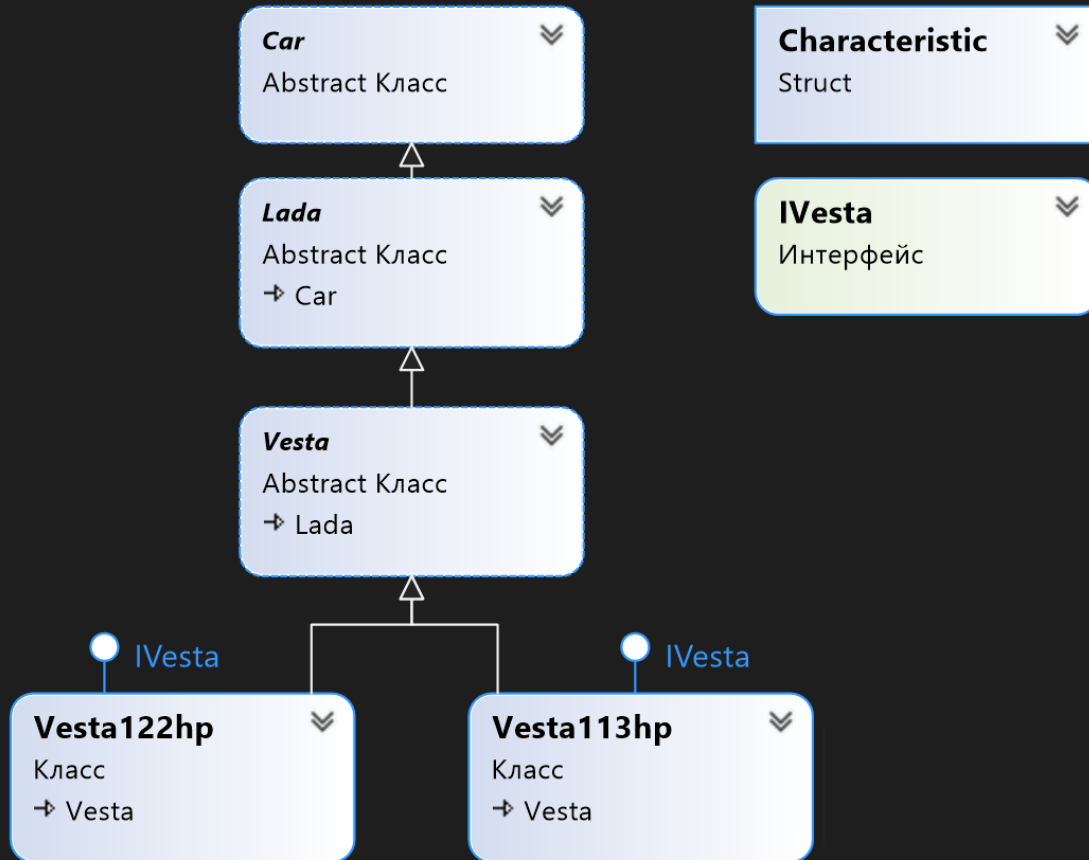
    public Characteristic BodyWheelArrangementTractionWheels => throw new NotImplementedException();
    public Characteristic BodyEngineLocation => throw new NotImplementedException();
    public Characteristic BodyTypeNumberOfDoors => throw new NotImplementedException();
    public Characteristic Bodyseats => ImplementedSomehow;
    public Characteristic BodyLengthWidthHeight => ImplementedSomehow;
    public Characteristic BodyWheelbase => ImplementedSomehow;
    public Characteristic BodyFrontRearTrack => ImplementedSomehow;
    public Characteristic BodyRoadClearance => ImplementedSomehow;
    public Characteristic BodyRearTrunkCapacity => ImplementedSomehow;
    public Characteristic SuspensionFront => throw new NotImplementedException();
    public Characteristic SuspensionBack => throw new NotImplementedException();
    public Characteristic SteeringControlSteeringMechanism => throw new NotImplementedException();
    public Characteristic TiresGeneralProportions => ImplementedSomehow;
}
```


SOLID: Interface segregation

```
public class Vesta113hp : Vesta, IVesta
{
    public Vesta113hp(string model, string brand)
        : base(model, brand) { }

    public Characteristic BodyWheelArrangementTractionWheels => throw new NotImplementedException();
    public Characteristic BodyEngineLocation => throw new NotImplementedException();
    public Characteristic BodyTypeNumberOfDoors => throw new NotImplementedException();
    public Characteristic BodySeats => ImplementedSomehow;
    public Characteristic BodyLengthWidthHeight => ImplementedSomehow;
    public Characteristic BodyWheelbase => ImplementedSomehow;
    public Characteristic BodyFrontRearTrack => ImplementedSomehow;
    public Characteristic BodyRoadClearance => ImplementedSomehow;
    public Characteristic BodyRearTrunkCapacity => ImplementedSomehow;
    public Characteristic SuspensionFront => throw new NotImplementedException();
    public Characteristic SuspensionBack => throw new NotImplementedException();
    public Characteristic SteeringControlSteeringMechanism => throw new NotImplementedException();
    public Characteristic TiresGeneralProportions => ImplementedSomehow;
}
```

SOLID: Interface segregation



SOLID: Interface segregation

SOLID: Interface segregation

```
public abstract class Car
{
    public string Model { get; set; }
    public string Brand { get; set; }

    public Car(string model, string brand)
    {
        this.Model = model;
        this.Brand = brand;
    }
}
```

SOLID: Interface segregation

```
public abstract class Car
{
    public string Model { get; set; }
    public string Brand { get; set; }

    public Car(string model, string brand)
    {
        this.Model = model;
        this.Brand = brand;
    }
}

public abstract class Audi : Car
{
    public Audi(string model, string brand)
        : base(model, brand) { }
}
```

SOLID: Interface segregation

```
public abstract class Audi : Car
{
    public Audi(string model, string brand)
        : base(model, brand) { }
}

public abstract class AudiA5 : Audi
{
    public AudiA5(string model, string brand)
        : base(model, brand) { }
}
```

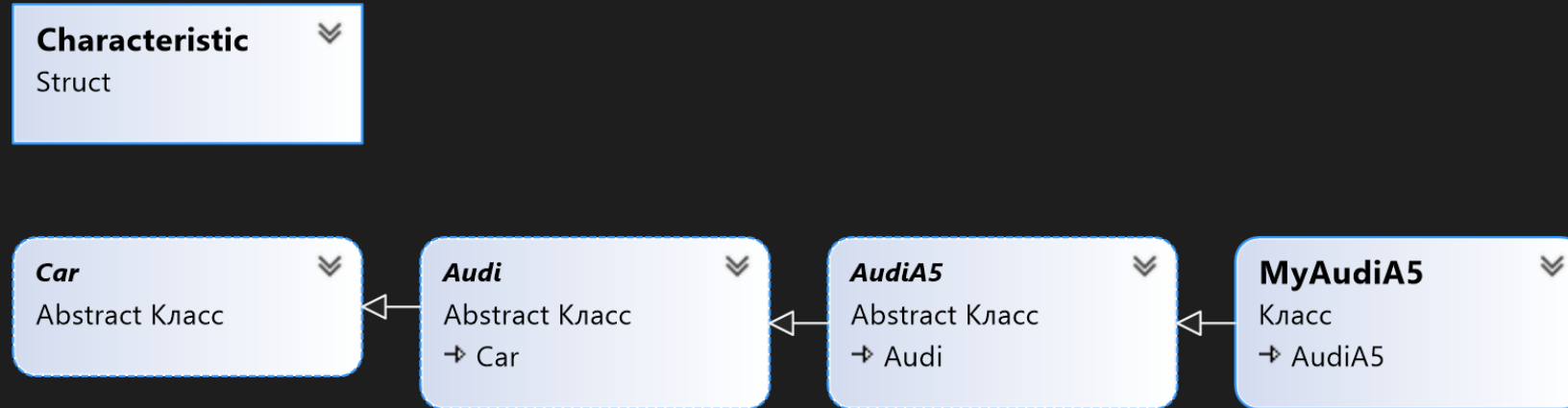
SOLID: Interface segregation

```
public abstract class Audi : Car
{
    public Audi(string model, string brand)
        : base(model, brand) { }
}

public abstract class AudiA5 : Audi
{
    public AudiA5(string model, string brand)
        : base(model, brand) { }
}

public class MyAudiA5 : AudiA5
{
    public MyAudiA5(string model, string brand)
        : base(model, brand) { }
}
```

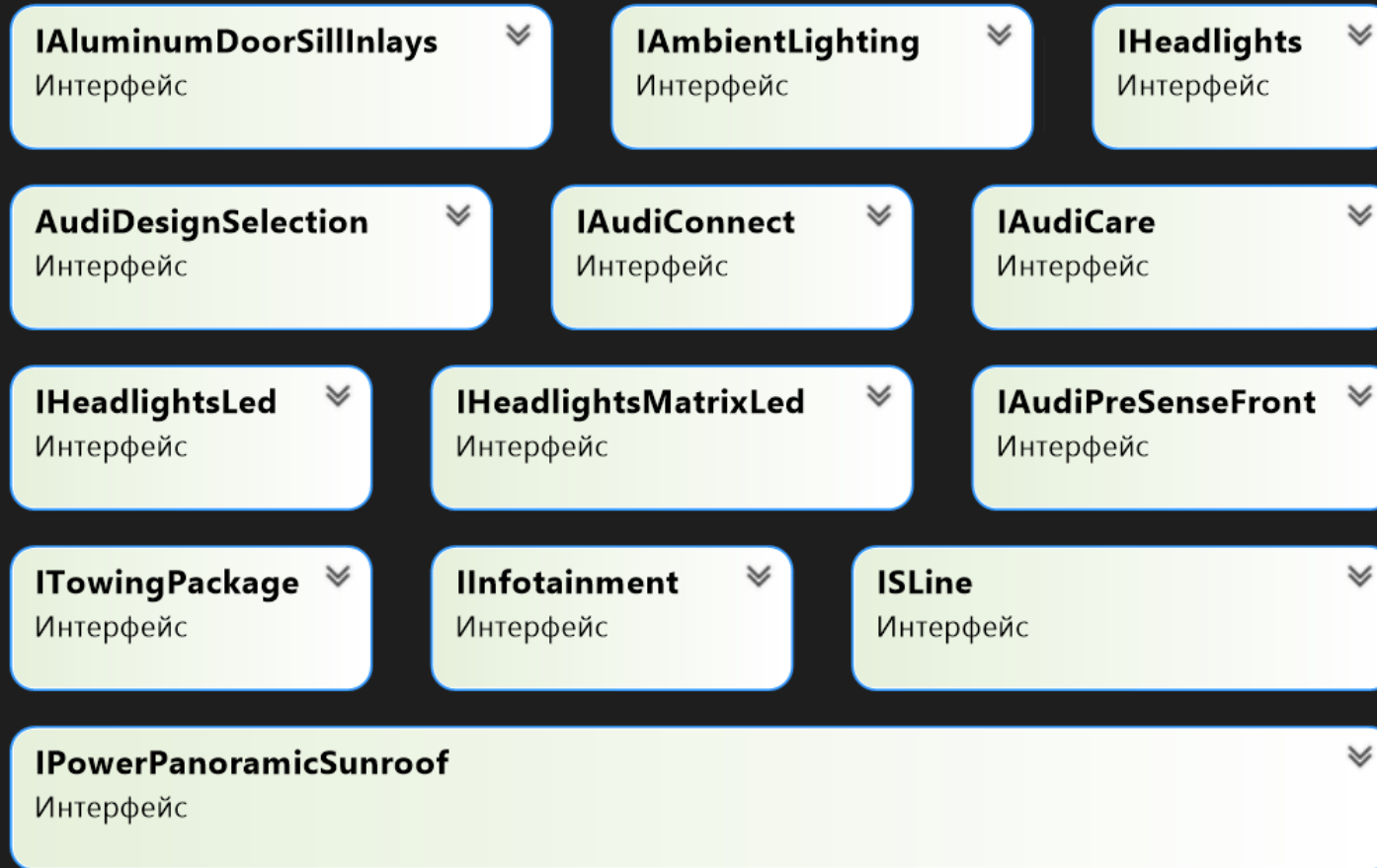
SOLID: Interface segregation



SOLID: Interface segregation

```
public class MyAudiA5 : AudiA5
{
    public MyAudiA5(string model, string brand)
        : base(model, brand) { }
}
```

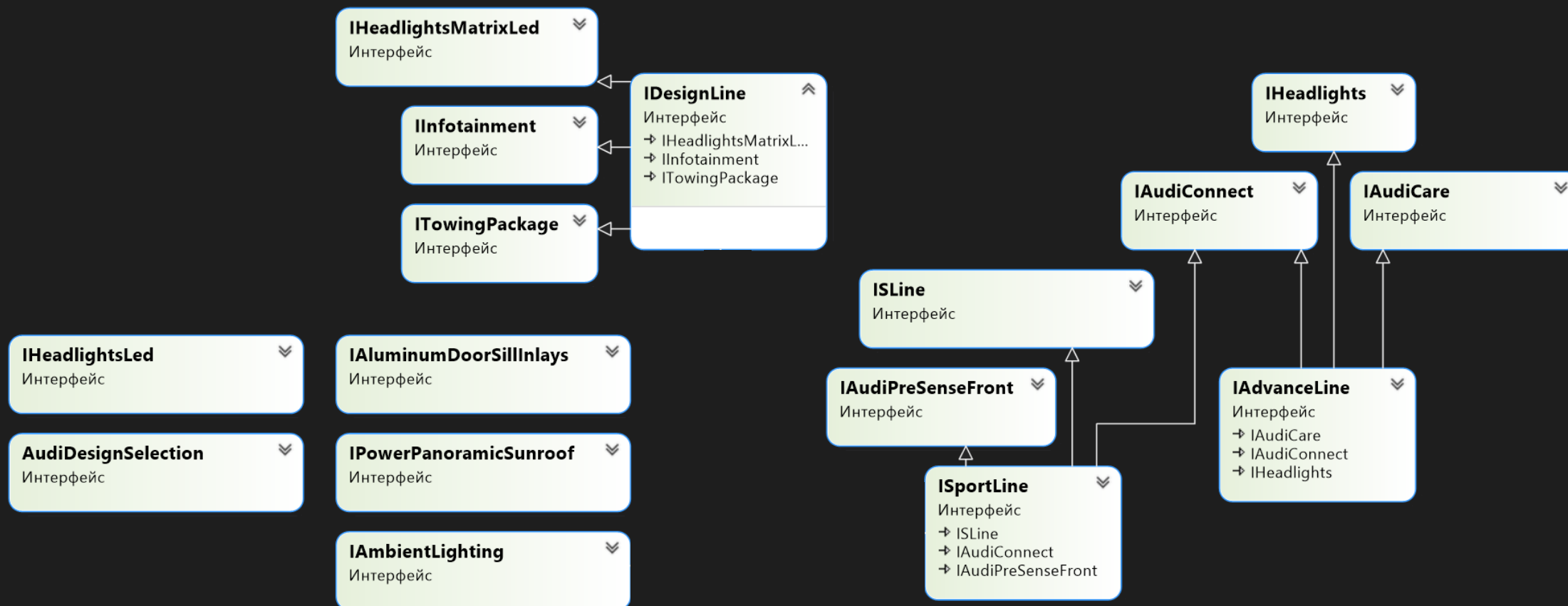
SOLID: Interface segregation



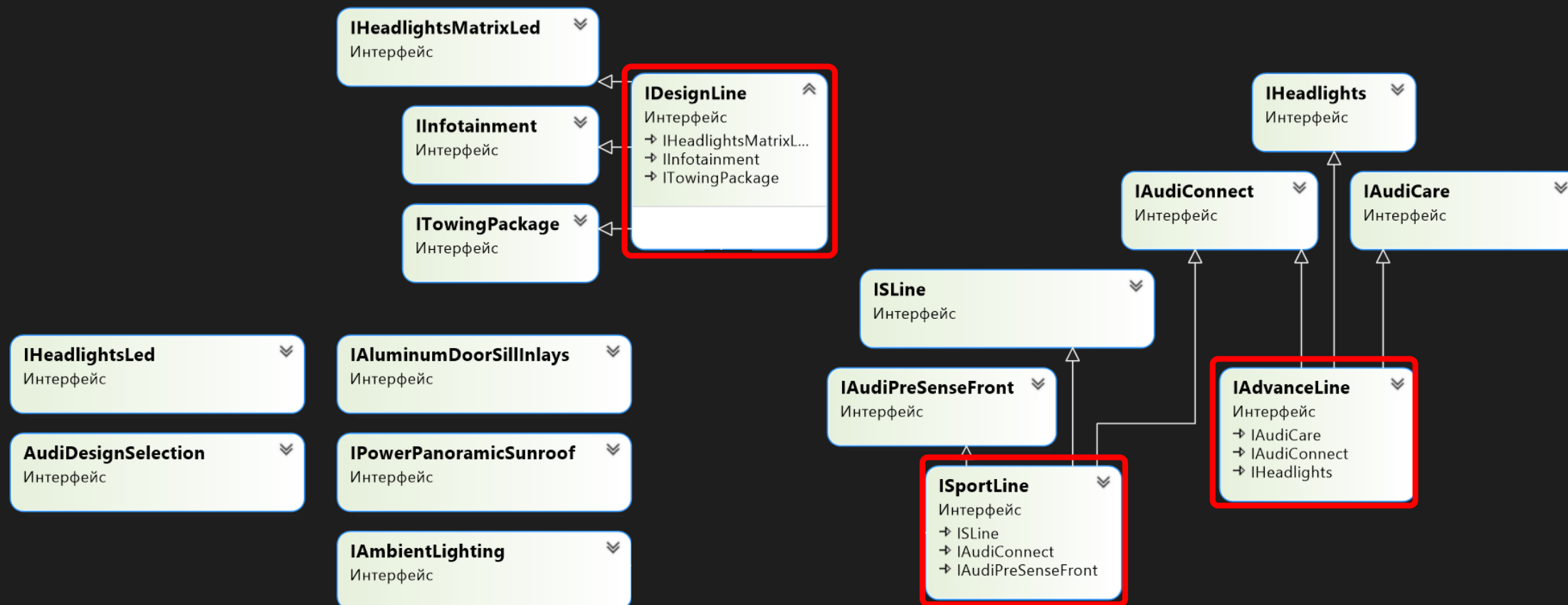
SOLID: Interface segregation

```
public class MyAudiA5 : AudiA5 , IAmbientLighting, IHeadlightsMatrixLed, ISLine
{
    public MyAudiA5(string model, string brand)
        : base(model, brand) { }
    public Characteristic AmbientLightingPackage { get; }
    public Characteristic HeadlightsMatrixLedPackage { get; }
    public Characteristic SLinePackage { get; }
}
```

SOLID: Interface segregation



SOLID: Interface segregation



SOLID: Interface segregation

SOLID: Interface segregation

```
public class MyAudiA5 : AudiA5, ISportLine
{
    public MyAudiA5(string model, string brand)
        : base(model, brand) { }
    public Characteristic SLinePackage { get; }
    public Characteristic AudiConnectPackage { get; }
    public Characteristic AudiPreSenseFrontPackage { get; }
}
```

SOLID: Interface segregation

```
public class MyAudiA5 : AudiA5, ISportLine
{
    public MyAudiA5(string model, string brand)
        : base(model, brand) { }
    public Characteristic SLinePackage { get; }
    public Characteristic AudiConnectPackage { get; }
    public Characteristic AudiPreSenseFrontPackage { get; }
}
```

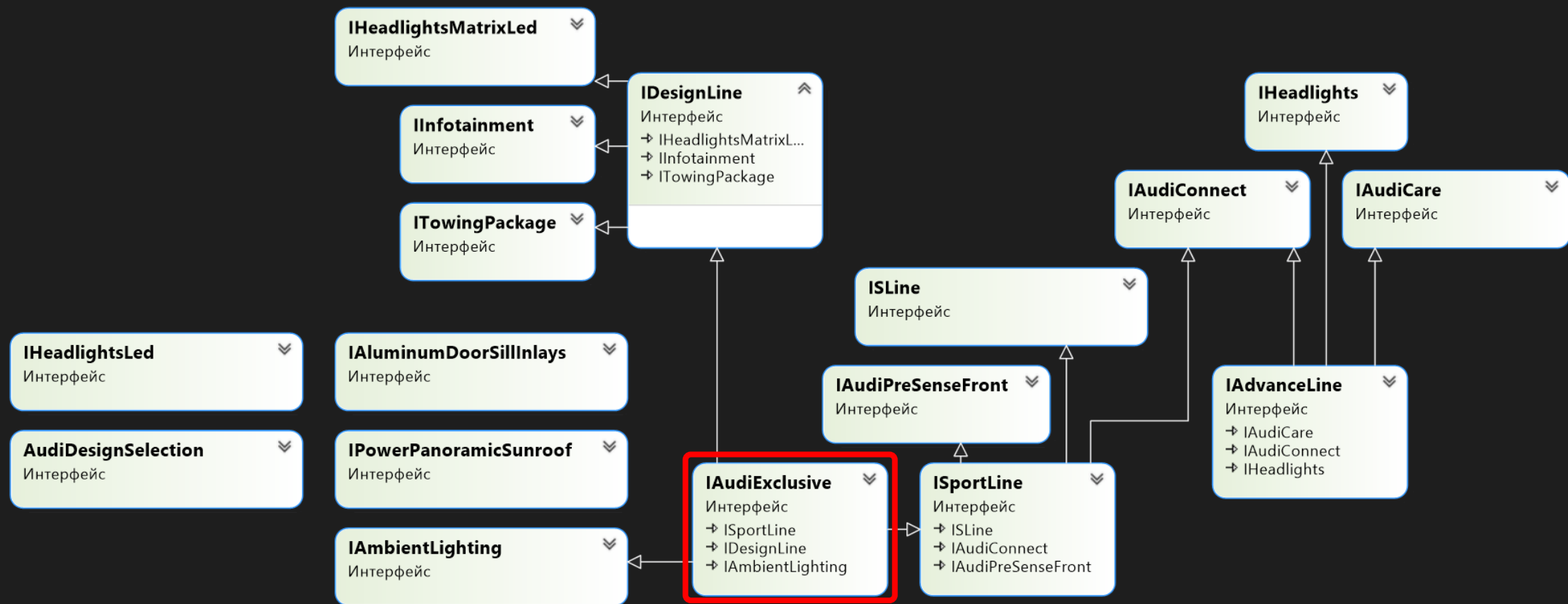
```
public class MyAudiA5 : AudiA5, IDesignLine
{
    public MyAudiA5(string model, string brand)
        : base(model, brand) { }
    public Characteristic HeadlightsMatrixLedPackage { get; }
    public Characteristic InfotainmentPackage { get; }
    public Characteristic TowingPackagePackage { get; }
}
```


SOLID: Interface segregation

```
public class MyAudiA5 : AudiA5, ISportLine
{
    public MyAudiA5(string model, string brand)
        : base(model, brand) { }
    public Characteristic SLinePackage { get; }
    public Characteristic AudiConnectPackage { get; }
    public Characteristic AudiPreSenseFrontPackage { get; }
}
```

```
public class MyAudiA5 : AudiA5, IDesignLine, IAmbientLighting
{
    public MyAudiA5(string model, string brand)
        : base(model, brand) { }
    public Characteristic HeadlightsMatrixLedPackage { get; }
    public Characteristic InfotainmentPackage { get; }
    public Characteristic TowingPackagePackage { get; }
    public Characteristic AmbientLightingPackage { get; }
}
```

SOLID: Interface segregation



SOLID: Interface segregation

```
public class MyAudiA5 : AudiA5, IAudiExclusive
{
    public MyAudiA5(string model, string brand)
        : base(model, brand) { }
    public Characteristic SLinePackage { get; }
    public Characteristic AudiConnectPackage { get; }
    public Characteristic AudiPreSenseFrontPackage { get; }
    public Characteristic HeadlightsMatrixLedPackage { get; }
    public Characteristic InfotainmentPackage { get; }
    public Characteristic TowingPackagePackage { get; }
    public Characteristic AmbientLightingPackage { get; }
}
```

не прощаюсь
все материалы
продолжаем обсуждение
<http://ksergey.ru/profcsharp/>



Принципы SOLID

Сергей Камянецкий

SOLID

Single responsibility — принцип единственной ответственности

Open-closed — принцип открытости / закрытости

Liskov substitution — принцип подстановки Барбары Лисков

Interface segregation — принцип разделения интерфейса

Dependency inversion — принцип инверсии зависимостей

SOLID: Dependency Inversion

Dependency inversion — принцип инверсии зависимостей

SOLID: Dependency Inversion

Dependency inversion — принцип инверсии зависимостей используется для уменьшения связности сущностей между собой

SOLID: Dependency Inversion

Dependency inversion — принцип инверсии зависимостей используется для уменьшения связности сущностей между собой

Модули верхних уровней не должны зависеть от модулей нижних уровней

Оба типа модулей должны зависеть от абстракций

SOLID: Dependency Inversion

Dependency inversion — принцип инверсии зависимостей используется для уменьшения связности сущностей между собой

Модули верхних уровней не должны зависеть от модулей нижних уровней

Оба типа модулей должны зависеть от абстракций

Абстракции не должны зависеть от деталей

Детали должны зависеть от абстракций

SOLID: Dependency Inversion

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
    ///
    ///
    ///
    ///
    ///
}
```

SOLID: Dependency Inversion

```
public class ListStorage
{
    private List<Person> storage;
    public ListStorage()
    {
        storage = new List<Person>();
    }
    public List<Person> GetPersons() => this.storage;
    public void Add(Person p) => storage.Add(p);
}
```

SOLID: Dependency Inversion

```
public class SearchByAge
{
    ListStorage storage;
    public SearchByAge(ListStorage storage) => this.storage = storage;
    public void Search()
    {
        foreach (var p in storage.GetPersons().Where(e => e.Age > 45))
        {
            Console.WriteLine($"{p.FirstName} {p.Age}");
        }
    }
}
```

SOLID: Dependency Inversion

```
public class SearchByFirstName
{
    ListStorage storage;
    public SearchByFirstName(ListStorage storage) => this.storage = storage;
    public void Search()
    {
        foreach (var p in storage.GetPersons().Where(e => e.FirstName.Contains("Name_3")))
        {
            Console.WriteLine($"{p.FirstName} {p.Age}");
        }
    }
}
```

SOLID: Dependency Inversion

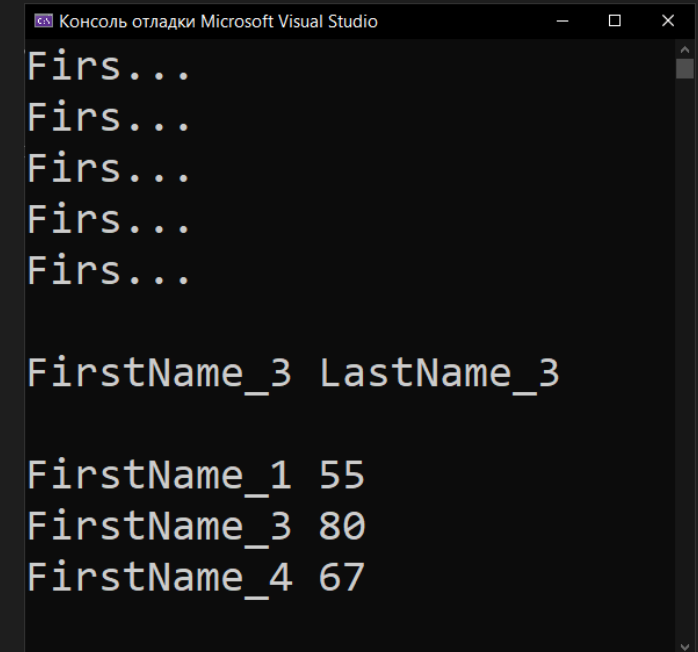
```
public class SearchByLastName
{
    ListStorage storage;
    public SearchByLastName(ListStorage storage) => this.storage = storage;
    public void Search()
    {
        foreach (var p in storage.GetPersons().Where(e => e.LastName.Contains("_")))
        {
            Console.WriteLine($"{p.FirstName.Substring(0,4)}..." );
        }
    }
}
```


SOLID: Dependency Inversion

```
var storage = new ListStorage();  
new SearchByLastName(storage).Search();  
new SearchByFirstName(storage).Search();  
new SearchByAge(storage).Search();
```

SOLID: Dependency Inversion

```
var storage = new ListStorage();  
new SearchByLastName(storage).Search();  
new SearchByFirstName(storage).Search();  
new SearchByAge(storage).Search();
```



Консоль отладки Microsoft Visual Studio

```
Firs...  
Firs...  
Firs...  
Firs...  
Firs...  
  
FirstName_3 LastName_3  
  
FirstName_1 55  
FirstName_3 80  
FirstName_4 67
```

SOLID: Dependency Inversion

```
public class DictionaryStorage
{
    private Dictionary<string, Person> storage;
    public DictionaryStorage()
    {
        storage = new Dictionary<string, Person>();
    }
    public Dictionary<string, Person> GetPersons() => this.storage;
    public void Add(string key, Person p) => storage.Add(key, p);
}
```

Проблемка

SOLID: Dependency Inversion

```
public class ListStorage
{
    private List<Person> storage;
    public ListStorage()
    {
        storage = new List<Person>();
    }
    public List<Person> GetPersons() => this.storage;
    public void Add(Person p) => storage.Add(p);
}
```

Проблемка

Было

SOLID: Dependency Inversion

```
public class DictionaryStorage
{
    private Dictionary<string, Person> storage;
    public DictionaryStorage()
    {
        storage = new Dictionary<string, Person>();
    }
    public Dictionary<string, Person> GetPersons() => this.storage;
    public void Add(string key, Person p) => storage.Add(key, p);
}
```

Проблемка

Стало

SOLID: Dependency Inversion

```
var storage = new ListStorage();  
new SearchByLastName(storage).Search();  
new SearchByFirstName(storage).Search();  
new SearchByAge(storage).Search();
```

```
//Ошибка CS1061 "KeyValuePair<string, Person>" не содержит определения "FirstName"  
//Ошибка CS1061 "KeyValuePair<string, Person>" не содержит определения "LastName"  
//Ошибка CS1061 "KeyValuePair<string, Person>" не содержит определения "Age"
```

SOLID: Dependency Inversion

Можно ли что-то сделать?

SOLID: Dependency Inversion

Можно ли что-то сделать?

Повысить уровень абстракции

SOLID: Dependency Inversion

Можно ли что-то сделать?

Повысить уровень абстракции

```
public interface IFindStorage
{
    List<Person> FindAll(Predicate<Person> predicate);
}
```

SOLID: Dependency Inversion

```
public class ListStorage : IFindStorage
{
    private List<Person> storage;
    public ListStorage()
    {
        storage = new List<Person>();
    }
    public List<Person> GetPersons() => this.storage;
    public void Add(Person p) => storage.Add(p);

    public List<Person> FindAll(Predicate<Person> predicate)
    {
        return storage.Where(e => predicate(e)).ToList();
    }
}
```

SOLID: Dependency Inversion

```
public class DictionaryStorage : IFindStorage
{
    private Dictionary<string, Person> storage;
    public DictionaryStorage()
    {
        storage = new Dictionary<string, Person>();
    }
    public Dictionary<string, Person> GetPersons() => this.storage;
    public void Add(string key, Person p) => storage.Add(key, p);

    public List<Person> FindAll(Predicate<Person> predicate)
    {
        return storage.Where(e => predicate(e.Value)).Select(e => e.Value).ToList();
    }
}
```

SOLID: Dependency Inversion

Что это дало?

SOLID: Dependency Inversion

```
public class SearchByAge
{
    IFindStorage storage;
    public SearchByAge(IFindStorage storage) => this.storage = storage;
    public void Search()
    {
        foreach (var p in storage.FindAll(e => e.Age > 45))
        {
            Console.WriteLine($"{p.FirstName} {p.Age}");
        }
    }
}
```

SOLID: Dependency Inversion

```
public class SearchByFirstName
{
    IFindStorage storage;
    public SearchByFirstName(IFindStorage storage) => this.storage = storage;
    public void Search()
    {
        foreach (var p in storage.FindAll(e => e.FirstName.Contains("Name_3")))
        {
            Console.WriteLine($"{p.FirstName} {p.Age}");
        }
    }
}
```

SOLID: Dependency Inversion

```
public class SearchByLastName
{
    IFindStorage storage;
    public SearchByLastName(IFindStorage storage) => this.storage = storage;
    public void Search()
    {
        foreach (var p in storage.FindAll(e => e.LastName.Contains("_")))
        {
            Console.WriteLine($"{p.FirstName} {p.Age}");
        }
    }
}
```

SOLID: Dependency Inversion

```
Console.WriteLine("ListStorage: ");  
var storage = new DictionaryStorage();  
new SearchByLastName(storage).Search();  
new SearchByFirstName(storage).Search();  
new SearchByAge(storage).Search();
```

```
Console.WriteLine("ListStorage: ");  
var listStorage = new ListStorage();  
new SearchByLastName(listStorage).Search();  
new SearchByFirstName(listStorage).Search();  
new SearchByAge(listStorage).Search();
```


SOLID: Dependency Inversion

```
Console.WriteLine("ListStorage: ");  
var storage = new DictionaryStorage();  
new SearchByLastName(storage).Search();  
new SearchByFirstName(storage).Search();  
new SearchByAge(storage).Search();
```

```
Console.WriteLine("ListStorage: ");  
var listStorage = new ListStorage();  
new SearchByLastName(listStorage).Search();  
new SearchByFirstName(listStorage).Search();  
new SearchByAge(listStorage).Search();
```

```
var storage = new ListStorage();  
new SearchByLastName(storage).Search();  
new SearchByFirstName(storage).Search();  
new SearchByAge(storage).Search();
```

SOLID: итоги

SOLID: итоги

Single responsibility — принцип единственной ответственности
антипаттерн «Божественный объект»

SOLID: итоги

Single responsibility — принцип единственной ответственности
антипаттерн «Божественный объект»

Open-closed — принцип открытости / закрытости
обратить внимание на паттерн «Декаратор»

SOLID: итоги

Single responsibility — принцип единственной ответственности
антипаттерн «Божественный объект»

Open-closed — принцип открытости / закрытости
обратить внимание на паттерн «Декаратор»

Liskov substitution — принцип подстановки Барбары Лисков
«Дети не ломают всё если заменить ими родителей»

SOLID: итоги

Single responsibility — принцип единственной ответственности
антипаттерн «Божественный объект»

Open-closed — принцип открытости / закрытости
обратить внимание на паттерн «Декаратор»

Liskov substitution — принцип подстановки Барбары Лисков
«Дети не ломают всё если заменить ими родителей»

Interface segregation — принцип разделения интерфейса
Не пишем всё в одном месте

SOLID: итоги

Single responsibility — принцип единственной ответственности
антипаттерн «Божественный объект»

Open-closed — принцип открытости / закрытости
обратить внимание на паттерн «Декаратор»

Liskov substitution — принцип подстановки Барбары Лисков
«Дети не ломают всё если заменить ими родителей»

Interface segregation — принцип разделения интерфейса
Не пишем всё в одном месте

Dependency inversion — принцип инверсии зависимостей
Высокоуровневый код не должен зависеть от низкоуровневого

<http://ksergey.ru/profcsharp/>



