



Лекция 3

Наследование. Отношения объектов.
Принципы SOLID. GRASP

Наследование

- Наследование позволяет создавать новые классы, которые повторно используют, расширяют и изменяют поведение, определенное в других классах. Класс, члены которого наследуются, называется *базовым классом*, а класс, который наследует эти члены, называется *производным классом*.


```
public class Animal{
    private string _name;
    public string Greet(){
        return "Hello, I'm some sort of animal!";
    }
}

public class Dog : Animal{
    public string Bark(){
        return "Woof!"
    }
}
```




Абстрактные и виртуальные методы и свойства

- Если базовый класс объявляет метод виртуальным (virtual), производный класс может переопределить (override) метод с помощью своей собственной реализации. Если базовый класс объявляет метод абстрактным (abstract), этот метод не имеет реализации и **должен быть** переопределен в любом неабстрактном классе, который прямо наследует от этого класса. Если производный класс сам является абстрактным, то он наследует абстрактные члены, не реализуя их.
- Можно запретить изменение метода с помощью модификатора sealed.
- Абстрактный класс может содержать только абстрактные методы и свойства. Создавать экземпляры абстрактного класса нельзя.



```
public class Animal{  
    public virtual string Greet(){  
        return "Hello, I'm some sort of animal!";  
    }  
}
```

```
public class Dog : Animal{  
    public override string Greet(){  
        return "Hello, I'm a dog!";  
    }  
}
```



```
abstract class Animal{  
    public abstract string Greet();  
}
```


```
public class Dog : Animal{  
    public override string Greet(){  
        return "Hello, I'm a dog!";  
    }  
}
```



Интерфейсы



- Интерфейс определяет набор сигнатур методов и свойств. Любой класс или структура, реализующий этот интерфейс, должен предоставлять реализацию для членов, определенных в интерфейсе.
- Начиная с C# 8.0, интерфейс может определять реализацию по умолчанию для членов. Он также может определять статические члены, чтобы обеспечить единую реализацию для общих функциональных возможностей.



```
public interface IControl{  
    void Paint();  
}
```

```
public interface ISurface{  
    void Paint()  
}
```

```
public class SampleClass : IControl, ISurface{  
    public void Paint(){  
        Console.WriteLine("Paint method in SampleClass");  
    }  
}
```


Примеры интерфейсов в BCL

- IDisposable

```
public void Dispose()
```

- IComparable

```
public int CompareTo (object? obj);
```

- IEnumerable

```
public IEnumerator GetEnumerator();
```

- IEnumerator

```
public bool MoveNext();
```

```
public object Current {get;}
```

```
public void Reset();
```


UML-диаграммы классов



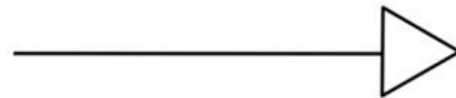
Ассоциация



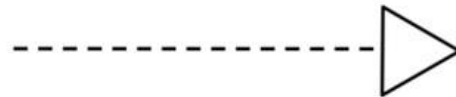
Агрегация



Композиция



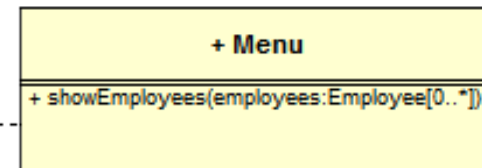
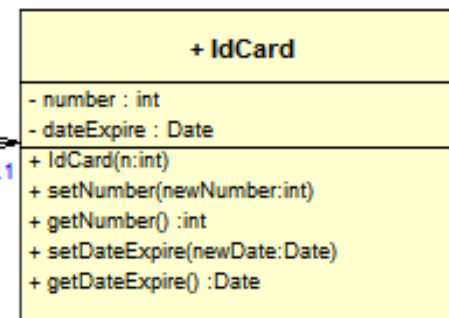
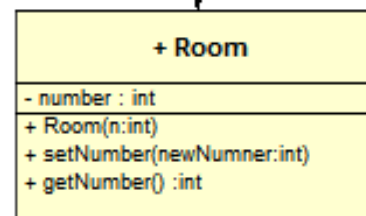
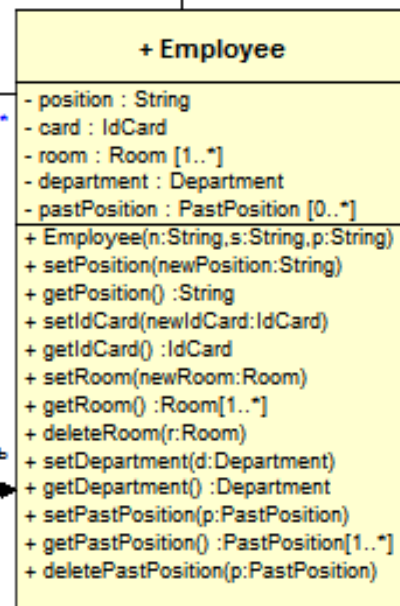
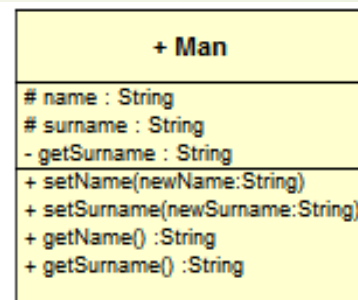
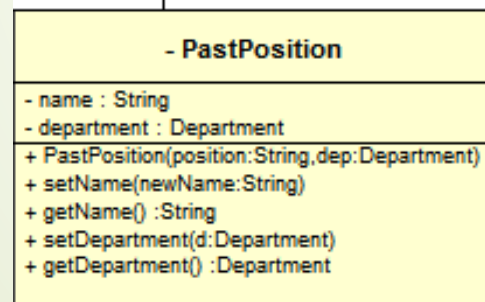
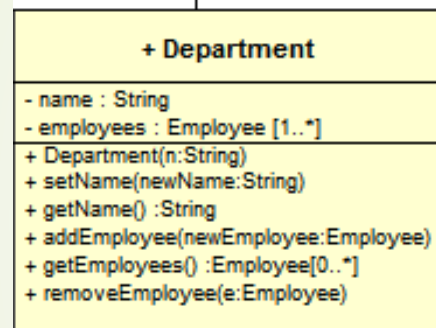
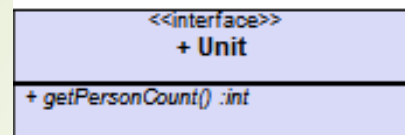
Обобщение



Реализация

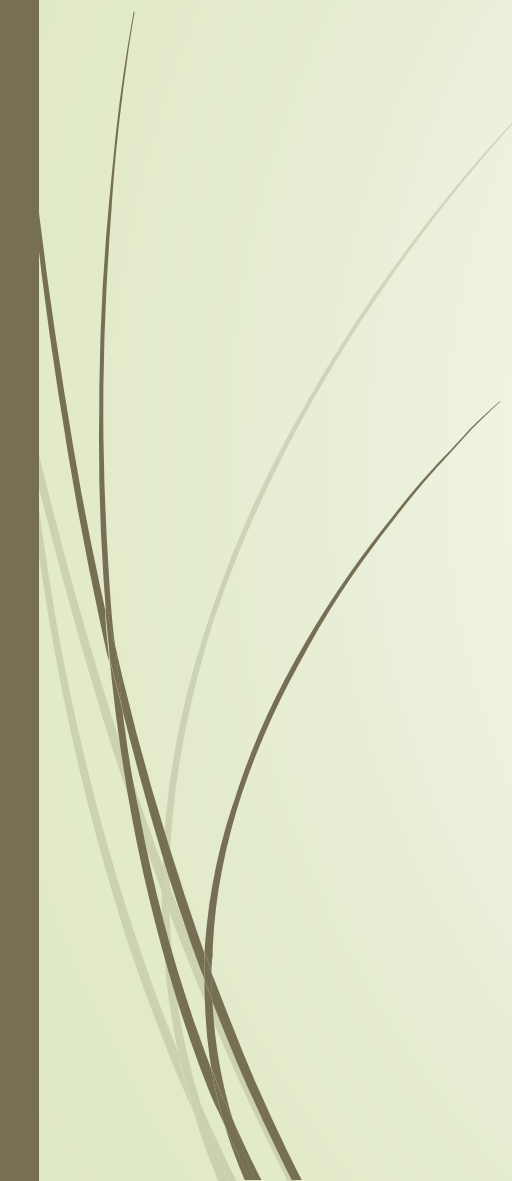


Зависимость






SOLID

- Single Responsibility Principle (Принцип единственной ответственности)
 - Open/Closed Principle (Принцип открытости/закрытости)
 - Liskov Substitution Principle (Принцип подстановки Лисков)
 - Interface Segregation Principle (Принцип разделения интерфейсов)
 - Dependency Inversion Principle (Принцип инверсии зависимостей)
- 

Single Responsibility Principle

- Класс должен выполнять какое-то одно действие и для его изменения должна быть только одна причина (влиять на спецификацию класса должно только какое-то одно потенциальное изменение в спецификации программы)

```
class Animal{  
    private string _name;  
    public Animal(string name);  
    public string Introduce(){  
        return "Hi, my name is " + this._name  
    }  
    public void SaveToFile(){  
        //  
    }  
}
```




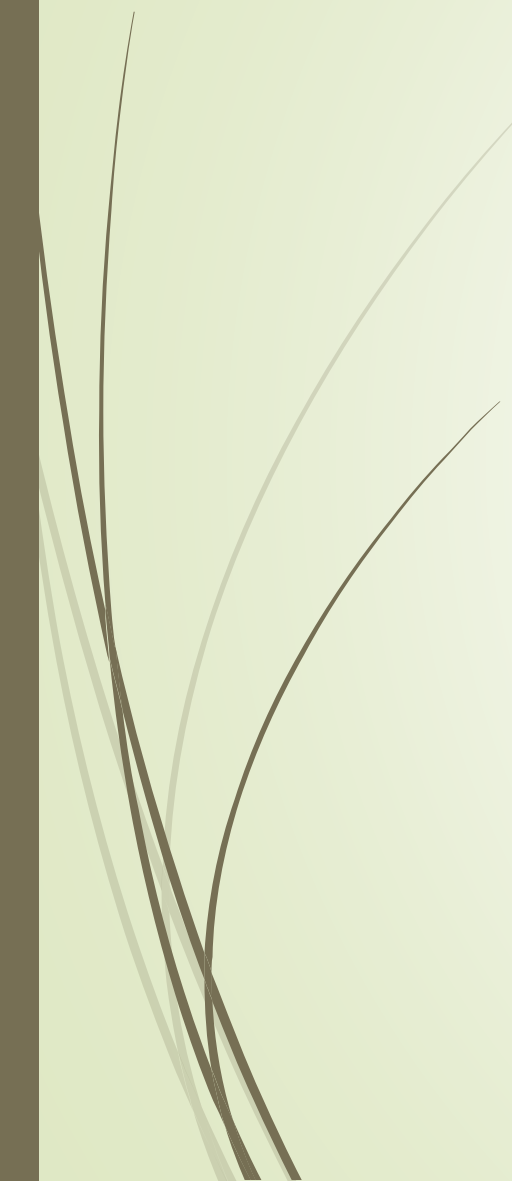

```
class Animal{
    private string _name;
    public Animal(string name);
    public string Introduce(){
        return "Hi, my name is " + this._name
    }
}
```

```
class AnimalSaver{
    private Animal _animal;
    public void SaveToFile(){
        ....
    }
}
```



Open/Closed Principle

- **Сущности программы должны быть открыты для расширения, но закрыты для изменения.**
 - Суть этого принципа состоит в том, что система должна быть построена таким образом, что все ее последующие изменения должны быть реализованы с помощью добавления нового кода, а не изменения уже существующего.
- 



```
interface IAnimalSaver{
    public void Save(Animal animal);
}


class AnimalSaverToXml : IAnimalSaver{
    private string _fileName;
    public AnimalSaverToXml(string fileName){
        this._filename = filename
    }
    public void Save(Animal animal){
        ....
    }
}

class AnimalSaverToJson : IAnimalSaver{
    ....
    public void SaveAnimal animal(){
        ....
    }
}
```




Liskov Substitution Principle

- Если у нас есть класс В, являющийся подклассом класса А, у нас должна быть возможность передать объект класса В любому методу, который ожидает объект класса А, причем этот метод не должен выдать в таком случае какой-то непредсказуемый результат.
- Дочерний класс расширяет поведение, но никогда не сужает его.
- Если класс не подчиняется принципу подстановки Барбары Лисков, это приводит к неприятным ошибкам, которые трудно обнаружить.



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

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



```
void Foo()
```

```
{
```

```
    var rect = new Square();
```

```
    TestRectangleArea(rect);
```

```
}
```

```
public static void TestRectangleArea(Rectangle rect)
```

```
{
```

```
    rect.Height = 5;
```

```
    rect.Width = 10;
```

```
    if (rect.GetArea() != 50)
```

```
        throw new Exception("Некорректная площадь!");
```



```
}
```





Проектирование по контракту


- Предусловия – это требования подпрограммы, т.е. то, что обязано быть истинным для выполнения подпрограммы. Если данные предусловия нарушены, то подпрограмма не должна вызываться ни в коем случае. Вся ответственность за передачу «правильных» данных лежит на **вызывающей программе**.
- Постусловия выражают состояния «окружающего мира» на момент выполнения подпрограммы. Т.е. это условия, которые гарантируются самой подпрограммой.
- Инварианты – это глобальные свойства класса. Класс гарантирует, что данное условие всегда истинно с точки зрения вызывающей программы.
- Если клиент, вызывающий подпрограмму, выполняет все предусловия, то вызываемая подпрограмма обязуется, что после ее выполнения все постусловия и инварианты будут истинными

- 
- 
- Предусловия (Preconditions) не могут быть усилены в подклассе. Другими словами подклассы не должны создавать больше предусловий, чем это определено в базовом классе, для выполнения некоторого поведения
 - Постусловия (Postconditions) не могут быть ослаблены в подклассе. То есть подклассы должны выполнять все постусловия, которые определены в базовом классе.

Interface Segregation Principle

- Клиенты не должны вынужденно зависеть от методов, которыми не пользуются (много клиентоориентированных интерфейсов лучше, чем один интерфейс общего назначения. Клиенты не должны принуждаться к реализации функций, которые им не нужны)

```
interface IMessage
{
    void Send();
    string Text { get; set;}
    string Subject { get; set;}
    string ToAddress { get; set; }
    string FromAddress { get; set; }
}
```



```
interface IMessage{  
    void Send();  
    string ToAddress { get; set; }  
    string FromAddress { get; set; }  
}
```

```
interface IVoiceMessage : IMessage{  
    byte[] Voice { get; set; }  
}
```

```
interface ITextMessage : IMessage{  
    string Text { get; set; }  
}
```

```
interface IEmailMessage : ITextMessage{  
    string Subject { get; set; }  
}
```


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


```
class Book{  
    public string Text { get; set; }  
    public ConsolePrinter Printer { get; set; }  
    public void Print(){  
        Printer.Print(Text);  
    }  
}  
  
class ConsolePrinter{  
    public void Print(string text){  
        Console.WriteLine(text);  
    }  
}
```

```
interface IPrinter{
    void Print(string text);
}
class Book{
    public string Text { get; set; }
    public IPrinter Printer { get; set; }
    public Book(IPrinter printer) {
        this.Printer = printer;
    }
    public void Print() {
        Printer.Print(Text);
    }
}
class ConsolePrinter : IPrinter{
    public void Print(string text){
        Console.WriteLine("Печать на консоли");
    }
}
```



Dependency Inversion Principle

Would you solder a lamp directly
to the electrical wiring in a wall?



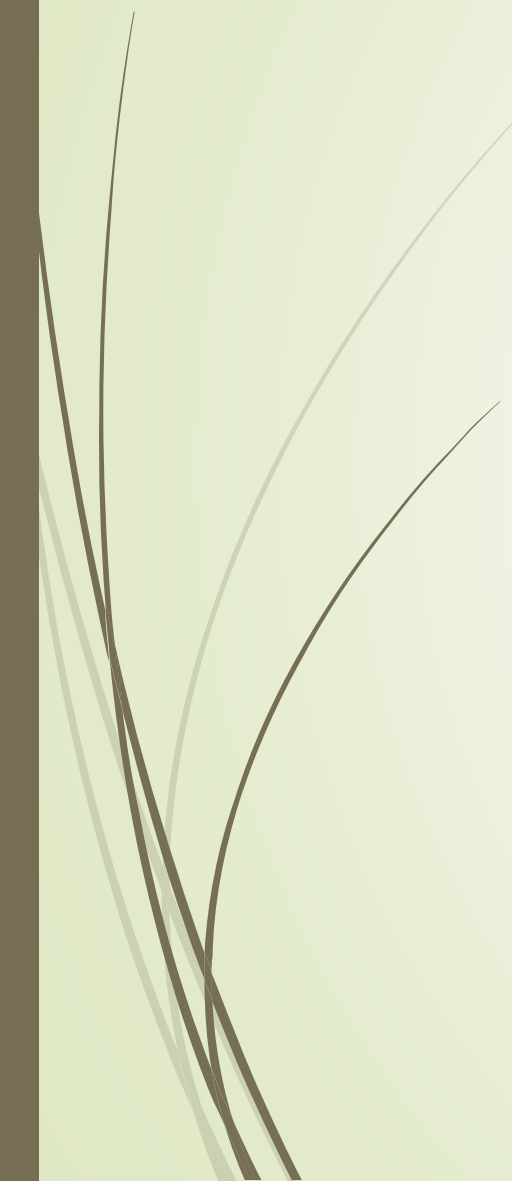

GRASP — General Responsibility Assignment Software Patterns

- Creator
- Controller
- Pure Fabrication
- Information Expert
- High Cohesion
- Indirection
- Low Coupling
- Polymorphism
- Protected Variations

Information Expert

- Ответственность должна быть назначена тому, кто владеет максимумом необходимой информации для исполнения — информационному эксперту (информация должна обрабатываться там, где она содержится!)

```
private int getOrderPrice(Order order) {  
    var orderItems = order.getOrderItems();  
    int result = 0;  
    for (var orderItem : orderItems) {  
        var amount = orderItem.getAmount();  
        var good = orderItem.getGood();  
        var price = good.getPrice();  
        result += price * amount;  
    }  
    return result;  
}
```



```
public class Order {  
    private List<OrderItem> orderItems;  
    private String destinationAddress;  
    public int getPrice() {  
        int result = 0;  
        for(var orderItem : orderItems) {  
            result += orderItem.getPrice();  
        }  
        return result;  
    }  
}
```


```
public class OrderItem {  
    private Good good;  
    private int amount;  
    public int getPrice() {  
        return amount * good.getPrice();  
    }  
}
```



Creator

- Класс должен создавать экземпляры тех классов, которые он может:
 - Содержать или агрегировать;
 - Записывать;
 - Использовать;
 - Инициализировать, имея нужные данные

```
public class Client {  
    public void doSmth() {  
        var good = new Good("name", 2);  
        var orderItem = new OrderItem(good, amount);  
        var orderItems = new ArrayList<>();  
        orderItems.add(orderItem);  
        var order = new Order(orderItems, "abc");  
    }  
}
```



```
public class Order {  
    private List<OrderItem> orderItems = new ArrayList<>();  
    private String destinationAddress;  
    public Order(String destinationAddress) {  
        this.destinationAddress = destinationAddress;  
    }  
    public int getPrice() { ... }  
    public void addOrderItem(int amount, String name, int price) {  
        orderItems.add(new OrderItem(amount, name, price));  
    }  
}
```

```
public class OrderItem {  
    private Good good;  
    private int amount;  
    public OrderItem(int amount, String name, int price) {  
        this.amount = amount;  
        this.good = new Good(name, price);  
    }  
    public int getPrice() { ... }  
}
```




Controller

- Обязанности по обработке входящих системных сообщений необходимо делегировать специальному объекту Controller'у. Controller — это объект, который отвечает за обработку системных событий, и при этом не относится к интерфейсу пользователя

Low Coupling


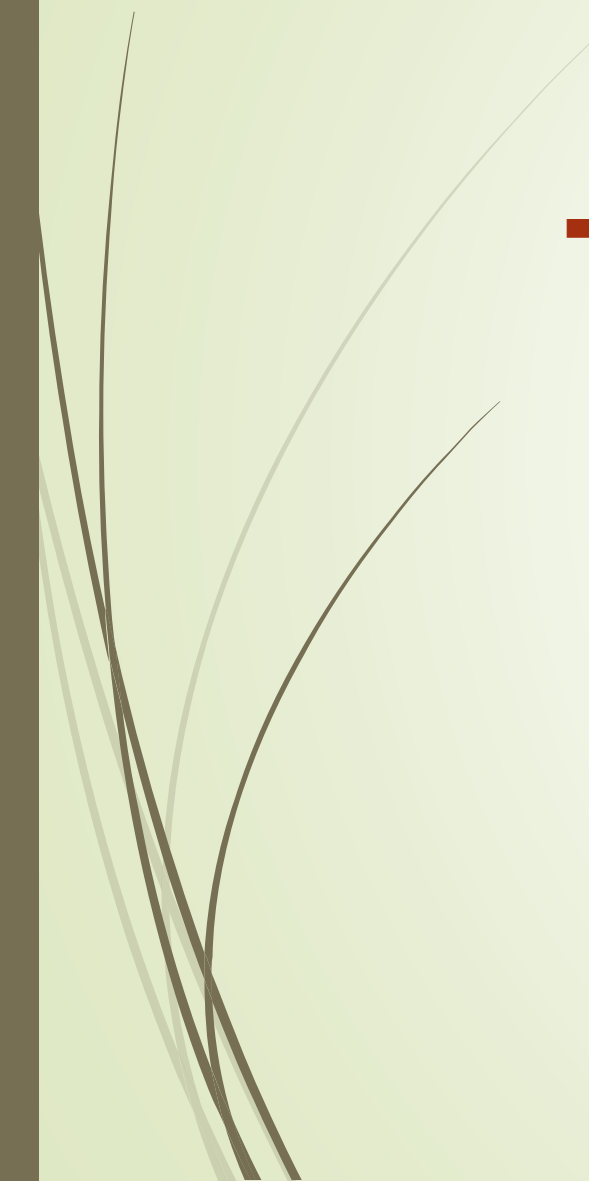
- **Связанность** — мера неотрывности элемента от других элементов
- Необходимо распределить ответственности между классами так, чтобы обеспечить **минимальную связанность**.

```
public class A {  
    private int a;  
    private B b;  
    public A(int a) {  
        this.a = a;  
        this.b = new B(this);  
    }  
}  
  
public class B {  
    private A a;  
    public B(A a) {  
        this.a = a;  
    }  
}
```

High Cohesion

- Связность класса — мера сфокусированности предметных областей его методов.
- Если возвести Low Coupling в абсолют, то достаточно быстро можно прийти к тому, чтобы разместить всю функциональность в одном единственном классе. В таком случае связей не будет вообще, но в этот класс попадет совершенно несвязанная между собой бизнес — логика.

```
public class Data {  
    private int temperature;  
    private int time;  
    private int calculateTimeDifference(int time) {  
        return this.time - time;  
    }  
    private int calculateTemperatureDifference(int temperature) {  
        return this.temperature - temperature;  
    }  
}
```

- 
- 
- Low Coupling и High Cohesion представляют из себя два связанных между собой паттерна, рассматривать которые имеет смысл только вместе. Их суть можно объединить следующим образом: система должна состоять из слабо связанных классов, которые содержат связанную бизнес — логику. Соблюдение этих принципов позволяет удобно переиспользовать созданные классы, не теряя понимания об их зоне ответственности



Pure Fabrication

- Необходимо обеспечивать low coupling и high cohesion. Для этой цели может понадобиться синтезировать искусственную сущность, не имеющую аналогов в предметной области
- Пример: Data Access Objects



Polymorphism

- Необходимо обрабатывать различные варианты поведения на основании типа, допуская замену частей системы.

Предлагается распределить обязанности между классами с использованием полиморфных операций, оставив каждой внешней системе свой интерфейс.

- Наличие в коде конструкции switch является нарушением данного принципа, switch'и подлежат рефакторингу.



Indirection

- Необходимо распределить обязанности между объектами, избежав прямого связывания. Для этого можно присвоить обязанности по обеспечению связи между компонентами или службами промежуточному объекту.
- Любой объект в коде необходимо вызывать через его интерфейс.



Protected Variations

- Необходимо спроектировать систему так, чтобы изменение одних ее элементов не влияло на другие.
- В качестве решения предлагается идентифицировать точки возможных изменений или неустойчивости и распределить обязанности таким образом, чтобы обеспечить устойчивую работу системы.



GRASP

- Information Expert — информацию обрабатываем там, где она содержится.
- Creator — создаем объекты там, где они нужны.
- Controller — выносим логику взаимодействия с пользователем в отдельный класс или компонент.
- Low Coupling, High Cohesion — проектируем классы с однородной бизнес-логикой и минимальным количеством связей между собой.
- Polymorphism — различные варианты поведения системы при необходимости оформляем в виде полиморфных вызовов.
- Pure Fabrication — не стесняемся создавать классы, не имеющие аналог в предметной области, если это необходимо для соблюдения Low Coupling и High Cohesion.
- Indirection — любой класс вызываем через его интерфейс.
- Protected Variations — применяя все вышесказанное, получаем устойчивый к изменениям код.