
Principiile SOLID în C#

— cu Sebastian Ichim la PeakIT 2019 —

Gratuit - Interactiv - Exerciții practice

Hai să ne cunoaștem...

- Lucrez în IT din 1997
- Am experiență în programare din 1991
 - Sinclair -> ZC Spectrum, BASIC
 - IBM PC XT compatibil
 - Programare procedurală -> Programare Orientată pe Obiecte
 - Pascal, Turbo C, Visual Studio 1.0, Windows Game SDK
 - Desktop -> Mobile/Cloud
 - Cursuri la MI:
 - Grafică 3D: DirectX 12.0, OpenGL ES, Modern OpenGL cu Shadere 4.6
 - Șabloane de proiectare
 - Am venit la cursurile AgileHub în 2013
 - Despre Agile și Scrum



Ce așteptări am eu de la voi?

- Să fiți implicați și activi
- Să vă cunoașteți
- Să împărtășiți propria experiență legată de subiect
- Să mă întrerupeți ori de câte ori aveți întrebări
- Să nu promovați companiile din care proveniți

Sunteți pregătiți?

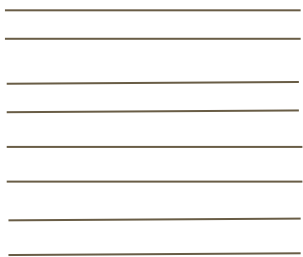
- Useful links:
 - https://codeshare.io/peakit2_SOLID
 - <https://github.com/ichimv/SolidPrinciples.git>
- Se vor face poze pentru promovarea PeakIT 02
- Wifi: user si parola
- Program:
 - Prima parte: 10:45-12:00 = 1h 15
 - A doua parte: 12:15-13:45 = 1h 30

Agenda

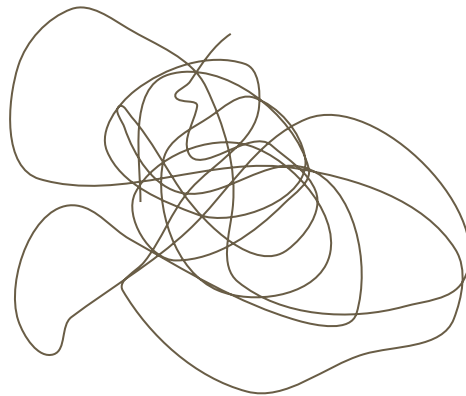
- Proiectarea codului
- Principiile SOLID
 - **S** - Single Responsibility Principle
 - **O** - Open/Closed Principle
 - **L** - Liskov Substitution Principle
 - **I** - Interface Segregation Principle
 - **D** - Dependency Inversion Principle
- Exerciții practice
- Concluzii

Simplitate și complexitate

- *“Sunt două moduri de proiectare software. Primul este de a-l face atât de simplu încât e evident că nu există deficiențe. Al doilea este de a-l face atât de complicat încât nu există deficiențe evidente. Prima metodă este mult mai dificilă” (Tony Hoare, 1981)*

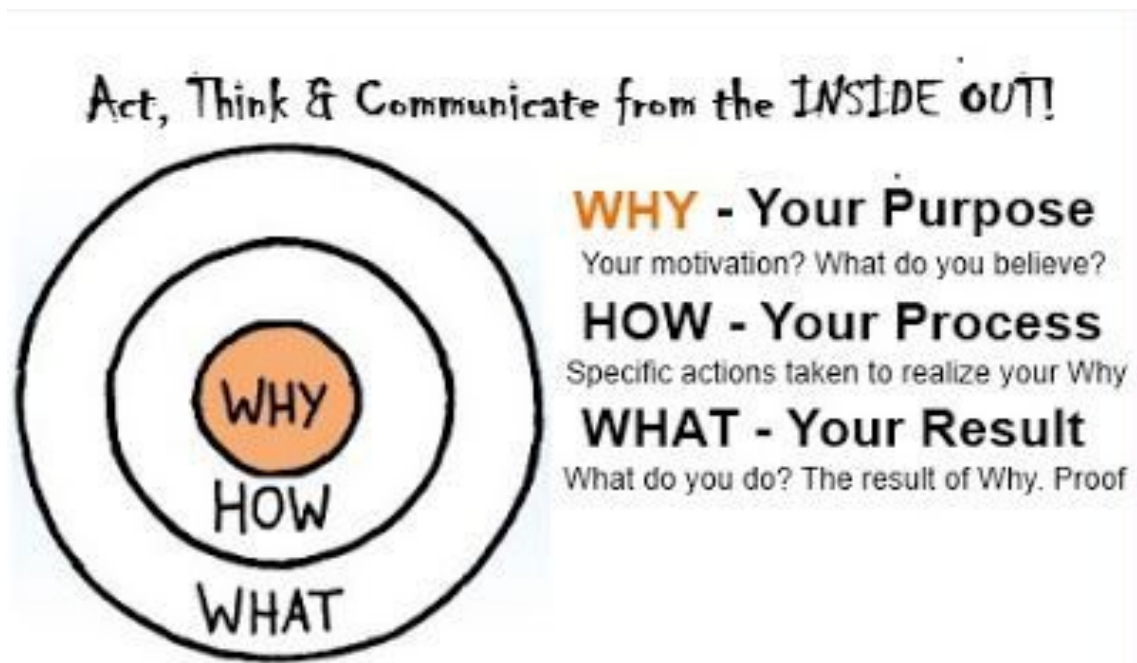


Lasania code vs Spaghetti code



Beneficiile codului proiectat corect

- Mai ușor de scris
- Mai ușor de înțeles
- Mai ușor de corectat
- Cu mai puține defecte
- Mai ușor de extins



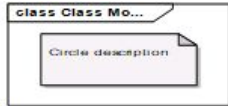
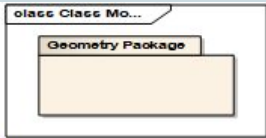
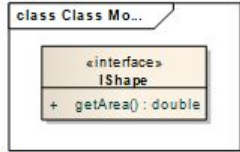
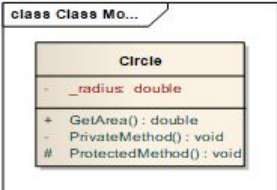
Nevoia calității designului

- Lipsa proiectării coerente conduce la scrierea unui cod haotic
 - Cu cât te apuci mai repede să scrii cod, cu atât termini mai târziu
 - Cu cât sistemul se bazează pe părți exotice, cu atât e mai intimidant să-l înțelegi
 - Datorită presiunii comerciale nu este timp să se proiecteze corect, dar este timp să se proiecteze de două ori

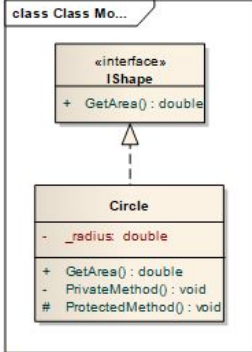
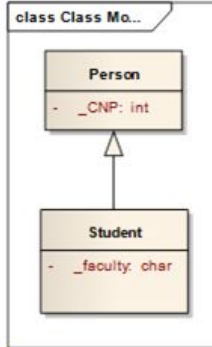
Calitatea Designului

- Complexitate minimă
 - Obiectivul principal al designului: minimizarea complexității
 - Designul îți permite să ignori alte părți ale sistemului când ești cufundat într-o zonă specifică
- Tehnici standardizate
 - Principii SOLID, șabloane de proiectare, etc
 - Diagrame UML (Unified Modeling Language)

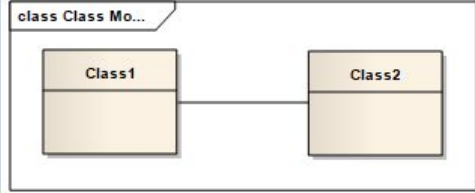
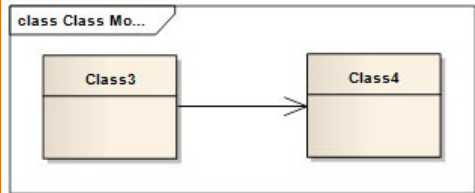
Elemente ale diagramelor UML (1)

Element din program	Element al diagramei	Descriere
Notă, comentariu		Orice text descriptiv
Pachet		Un grup de clase și interfețe
Interfață		Numele incepe cu "I". Se folosește și pentru clase abstracte.
Clasă		Accesul este indicat de: +(public), -(private) și #(protected)

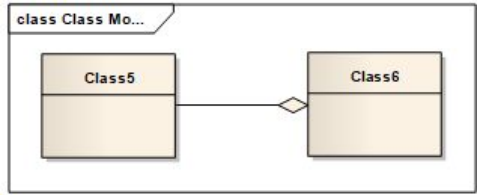
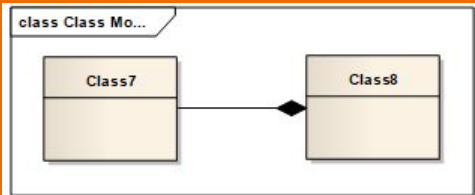
Elemente ale diagramelor UML (2)

Element din program	Element al diagramei	Descriere
Moștenire Realizare	 <pre> classDiagram class IShape { <<interface>> +GetArea() double } class Circle { -_radius double +GetArea() double -PrivateMethod() void #ProtectedMethod() void } IShape < .. Circle </pre>	Circle realizează interfața IShape
Moștenire Generalizare/ Specializare	 <pre> classDiagram class Person { -_CNP int } class Student { -_faculty char } Person < -- Student </pre>	Clasa Person generalizează clasa Student. Clasa student este o specializare a clasei Person.

Elemente ale diagramelor UML (3)

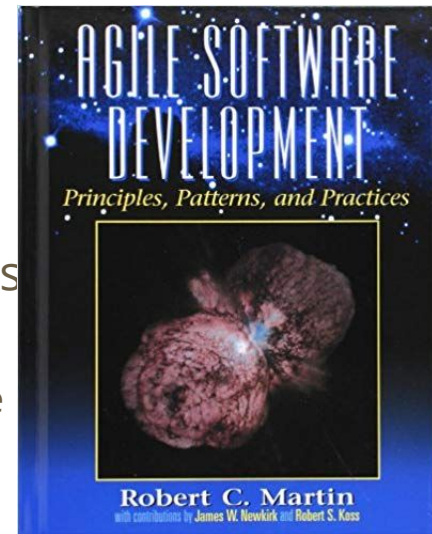
Element din program	Element al diagramei	Descriere
Asociere	 <p>The diagram shows a rectangular frame with a tab labeled 'class Class Mo...'. Inside, two class boxes labeled 'Class1' and 'Class2' are connected by a horizontal line, representing an association.</p>	Class1 și Class2 își apelează și își accesează reciproc elementele
Asociere directă	 <p>The diagram shows a rectangular frame with a tab labeled 'class Class Mo...'. Inside, a class box labeled 'Class3' is connected to a class box labeled 'Class4' by a directed association arrow pointing from Class3 to Class4.</p>	Class3 poate apela și accesa elementele lui Class4, dar nu și invers

Elemente ale diagramelor UML (4)

Element din program	Element al diagramei	Descriere
Agregare		Class6 are un atribut de tip Class5 și Class5 nu depinde de Class6
Compoziție		Class8 are un atribut de tip Class7 și Class7 depinde de Class8

Principiile S.O.L.I.D.

- Extinde POO și este un subset al principiilor formulate de Robert C. Martin în “Agile Software Development, Principles, Patterns, and Practices” publicată în 2002:
 - **S** - Single Responsibility Principle -> Principiul responsabilității unice
 - **O** - Open/Closed Principle -> Principiul deschis/închis
 - **L** - Liskov Substitution Principle -> Principiul Liskov al substituției
 - **I** - Interface Segregation Principle -> Principiul de segregare a interfețelor
 - **D** - Dependency Inversion Principle -> Principiul inversiunii dependențelor



Single Responsibility Principle (SRP)

- *Fiecare clasă trebuie să aibă o responsabilitate unică, iar aceasta trebuie să fie încapsulată în clasă*
- O clasă trebui să facă un singur lucru
- O clasă trebuie să aibă un singur motiv de schimbare
- Dacă poți găsi mai multe motive pentru a schimba o clasă înseamnă că are mai multe responsabilități



SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

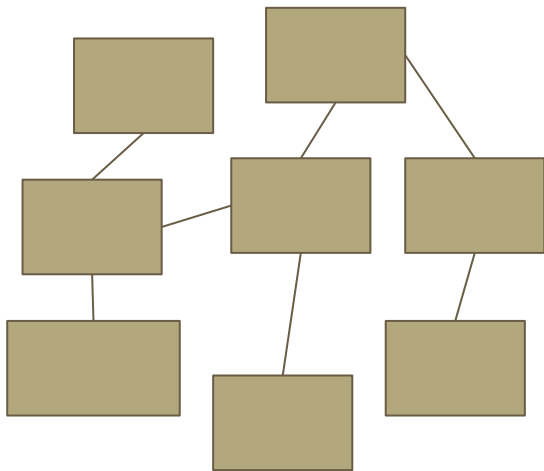
Coeziunea și cuplarea

- Coeziunea:
 - Măsura clarității responsabilității unui modul
 - Arată cât de apropiate sunt elementele aceluiasi modul
- Cuplarea:
 - Gradul în care fiecare modul al programului este legat direct de alte module
 - Cât trebuie să știm despre un modul pentru a-l înțelege pe altul
 - Măsura în care modificarea unui modul afectează alt modul

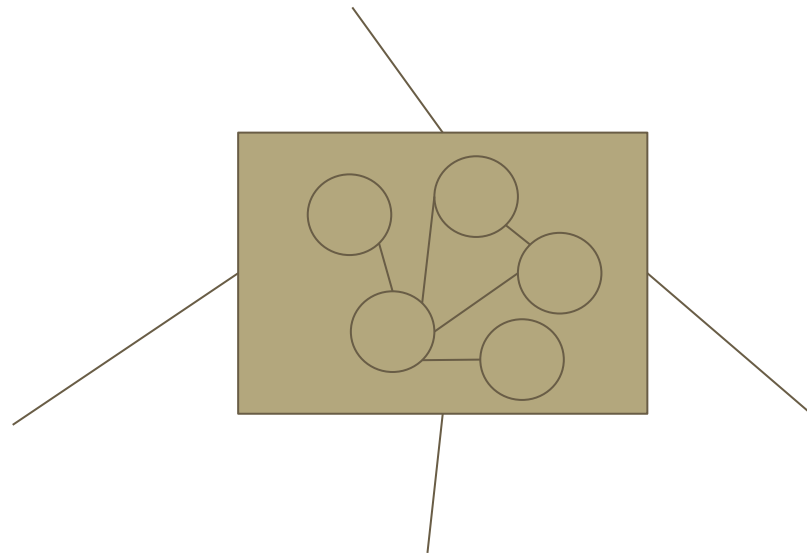
Coeziunea și cuplarea

- Coeziune slabă = Multe/neclare responsabilități care nu au multe în comun
 - **Coeziune ridicată** = Responsabilitate clară/ridicăta a modului
-
- Cuplare ridicată = Conexiuni multe între modulele programului
 - **Cuplare scăzută** = Conexiuni minime între modulele programului (Lego)

Complexitatea Designului



**Cuplarea între
module**



**Coeziunea în
modul**

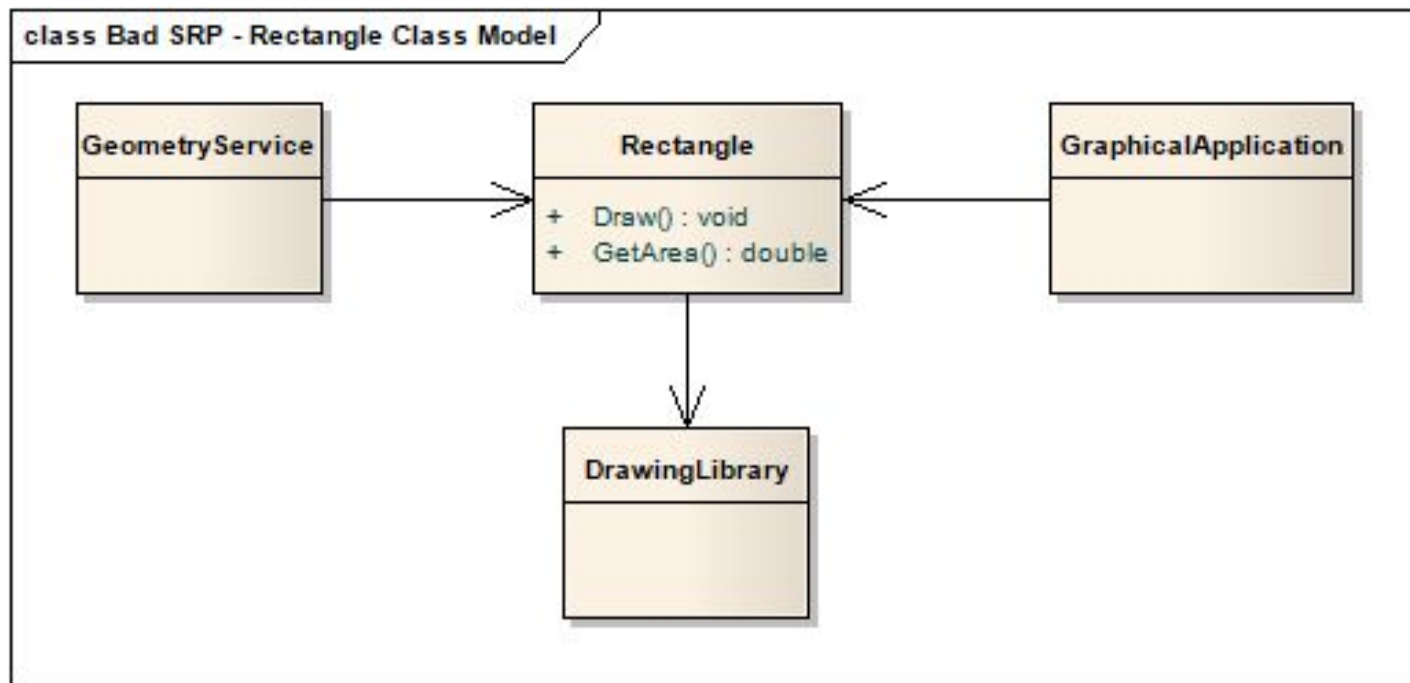
Responsabilitățile sunt motive de schimbare

- Cerințele noi se materializează în responsabilități noi
- Mai multe responsabilități = probabilitate mai mare de schimbare
- Legarea mai multor responsabilități în clasă = multe responsabilități
- Cu cât o clasă e schimbată mai des, cu atât cresc șansele introducerii erorilor

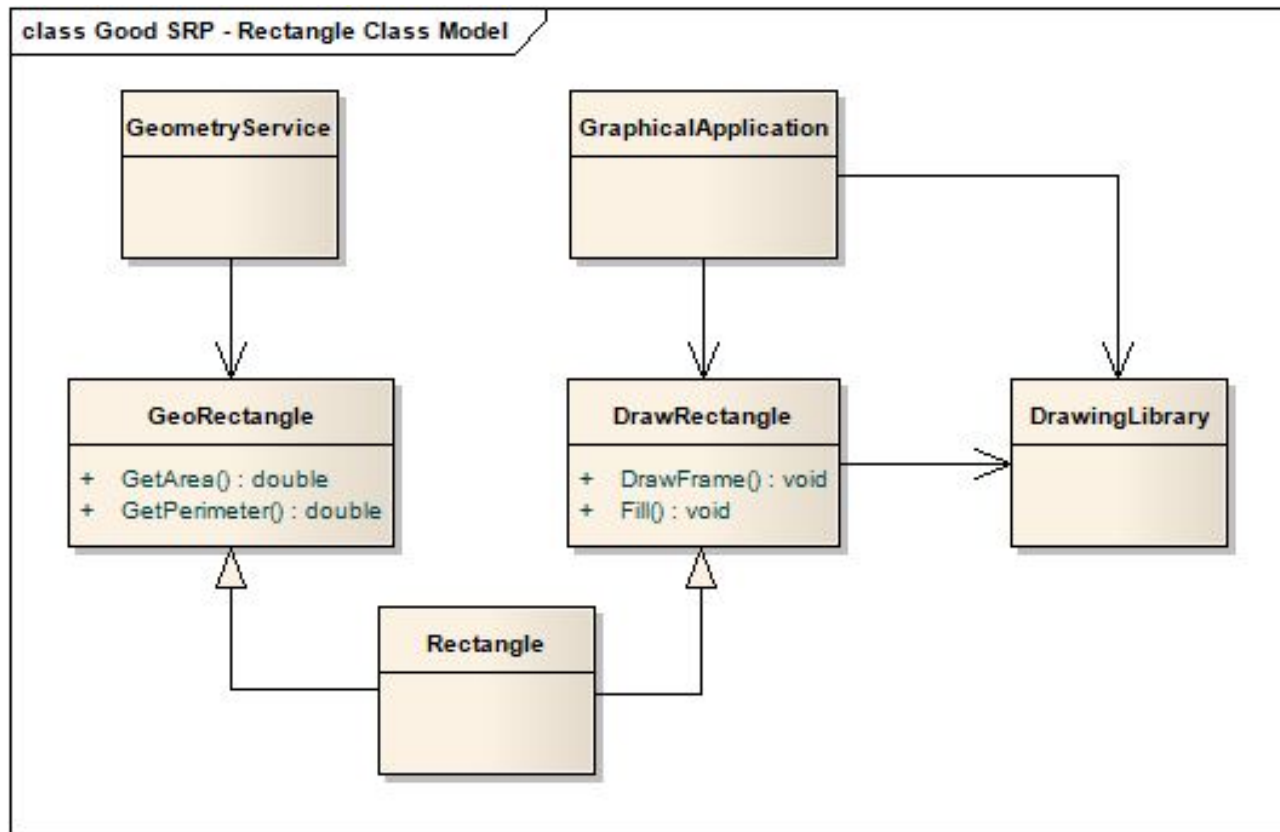
Exemple de responsabilități

- Logare
- Validare
- Notificare
- Formatare
- Parsare
- Randare
- Conversia datelor
- Error handling

Încălcare SRP - responsabilități multiple



Separarea responsabilităților



Exercițiu SRP - problema

```
class CustomerDb {  
  
    public void Add(string name) {  
  
        try {  
  
            // data base code  
  
        }  
  
        catch (Exception ex) {  
  
            System.IO.File.WriteAllText(@"c:\LogFile.log", ex.ToString());  
  
        }  
  
    }  
  
}
```


Exercițiu SRP - soluția

```
class CustomerDb {  
  
    private FileLogger _logger = new FileLogger();  
  
    public void Add(string name) {  
  
        try {  
  
            // data base code  
  
        }  
  
        catch (Exception ex) {  
  
            _logger.Handle(ex);  
  
        }  
  
    }  
  
}
```

```
class FileLogger {  
  
    public void Handle(Exception ex)  
  
    {  
  
        System.IO.File.WriteAllText(@"c:\LogFile.log", ex.ToString());  
  
    }  
  
}
```

Concluzie SRP

- Responsabilitate = “motiv pentru schimbare”
- Respectând SRP obținem **Cuplare scăzută** și **Coeziune ridicată**
- Mai multe clase cu responsabilități distincte determină un design mai flexibil

Open/Closed Principle (OCP) (Bertrand Mayer 1998)

- *Entitatea software (modul, clasă, metodă) trebuie să fie deschisă pentru extindere dar închisă pentru modificări*
- Extindem comportamentul entității fără a modifica codul existent
- Beneficii:
 - Clasa de bază este testată și nu se schimbă decât pentru fixare de bug-uri
 - Mentenabilitatea = Scade frecvența schimbărilor în codul existent



OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat

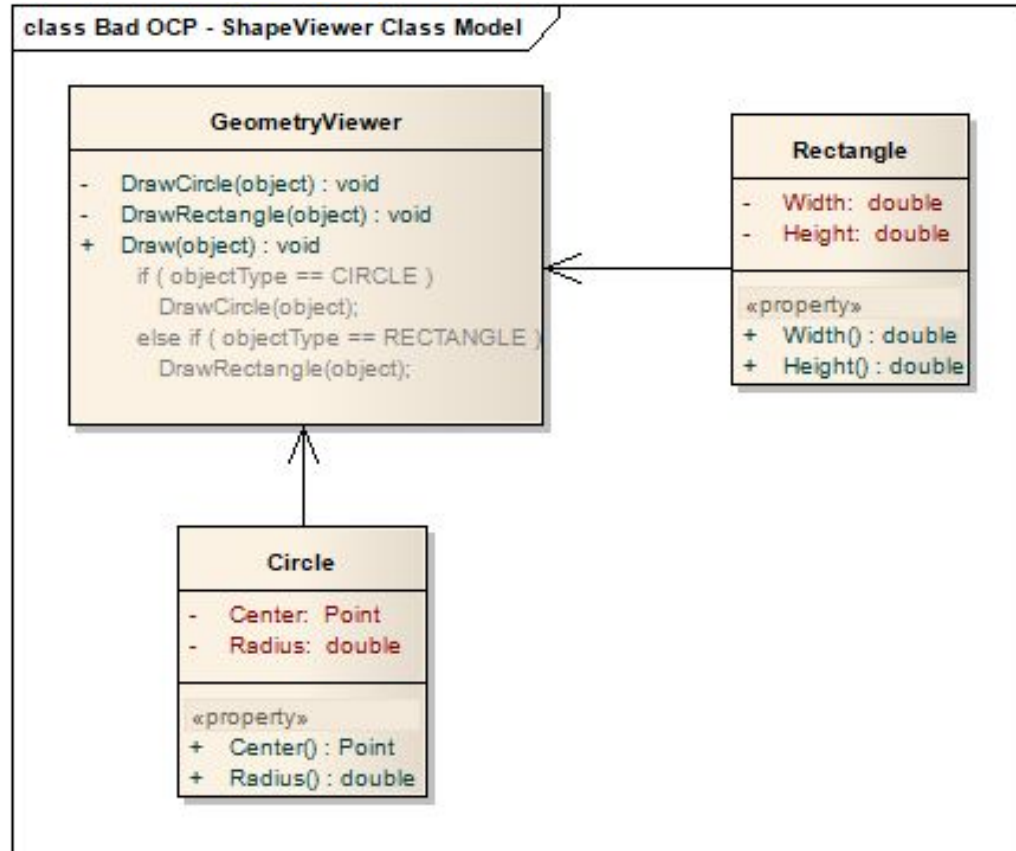
Principiul Deschis/Închis

- Deschis pentru extensii
 - Comportament nou se poate adăuga în viitor
- Închis pentru modificări
 - Modificări ale codului existent nu sunt necesare
 - Dacă schimbi o clasă trebuie să schimbi și clasele care depind de ea

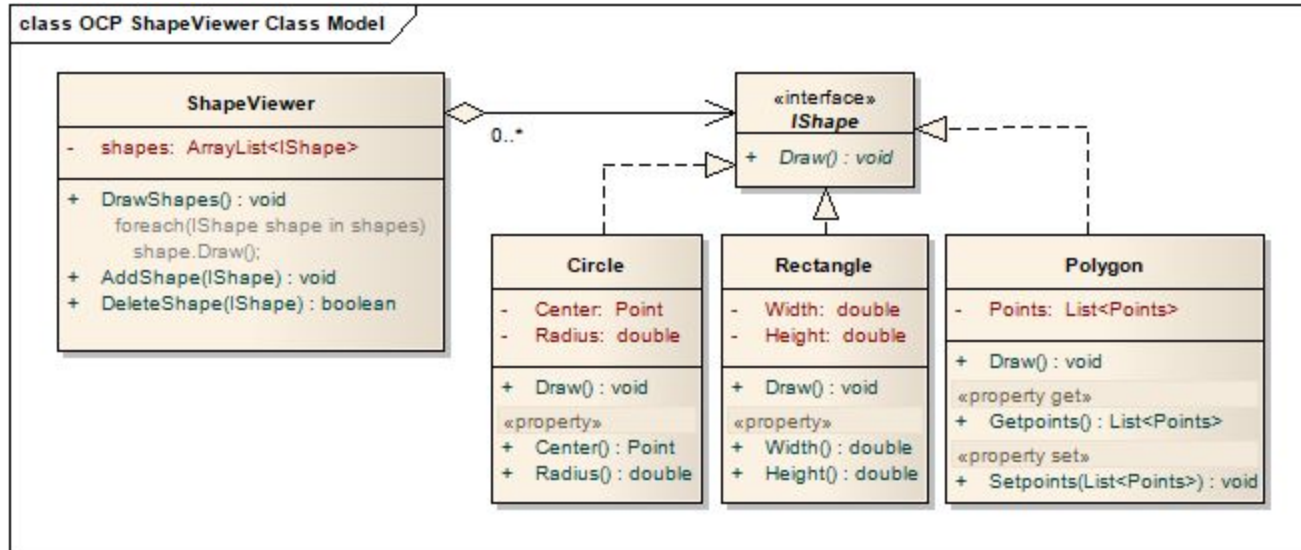
Schimbi comportamentul fără să schimbi codul?

- Se realizează prin abstractizare
- Extinderea prin moștenire fără modificarea codului existent
 - Interfețe
 - Clase abstracte

Exemplu OCP - desenarea formelor geometrice



Exemplu OCP - suport pentru triunghiuri



Exercițiu OCP

Liskov Substitution Principle (LSP)

- *Orice obiect de tipul unei clase derivate trebuie să poată substitui obiecte de tipul clasei de bază fără ca programul să fie afectat*
- Introdus de Barbara Liskov pe 4 octombrie 1987
- Metodele care folosesc instanțe ale claselor de bază trebuie să poată folosi și instanțele claselor derivate cu același efect
- O clasă derivată trebuie să facă același lucru ca și clasa ei de bază și încă ceva în plus



LISKOV SUBSTITUTION PRINCIPLE



If it looks like a Duck, quacks like a Duck, but needs Batteries
- You probably have The Wrong Abstraction!

Substituibilitatea

- Clasele derivate nu trebuie să:
 - Schimbe comportamentul clasei lor de bază
 - Încalce constrângerile definite sau asumate în clasa de bază ([setter](#) [abstract](#))
 - Apeleze cod ca să știe că sunt diferite față tipurile lor de bază

Moștenirea și relația “is-a”

- În POO clasele derivate sunt în relația “is-a” cu clasa de bază
- Regula “is-a” are sens în “lumea reală” dar nu este aplicabilă întotdeauna în software design -> “square is a kind of rectangle”
- Conform LSP relația “is-a” trebuie înlocuită cu relația “is-substitutable-for”

Pătratul nu poate substitui un dreptunghi

```
class Rectangle {
```

```
    private double itsWidth;
```

```
    private double itsHeight;
```

```
    public Rectangle(double w, double h) {
```

```
        SetWidth(w); SetHeight(h);
```

```
    }
```

```
    public virtual void SetWidth(double w) { itsWidth = w; }
```

```
    public virtual void SetHeight(double h) { itsHeight = h; }
```

```
    public double GetHeight() { return itsHeight; }
```

```
    public double GetWidth() { return itsWidth; }
```

```
}
```

```
class Square : Rectangle {
```

```
    public Square(double side) : base(side, side) {}
```

```
    public override void SetWidth(double w) {
```

```
        base.SetWidth(w); base.SetHeight(w);
```

```
    }
```

```
    public override void SetHeight(double h) {
```

```
        base.SetHeight(h); base.SetWidth(h);
```

```
    }
```

```
}
```

```
class SquareTest
```

```
    void ValidateRectangleArea(Rectangle r)
```

```
    {
```

```
        r.SetWidth(5);
```

```
        r.SetHeight(4);
```

```
    }
```

```
}
```

```
        Debug.Assert((r.GetWidth() * r.GetHeight() == 20.0));
```

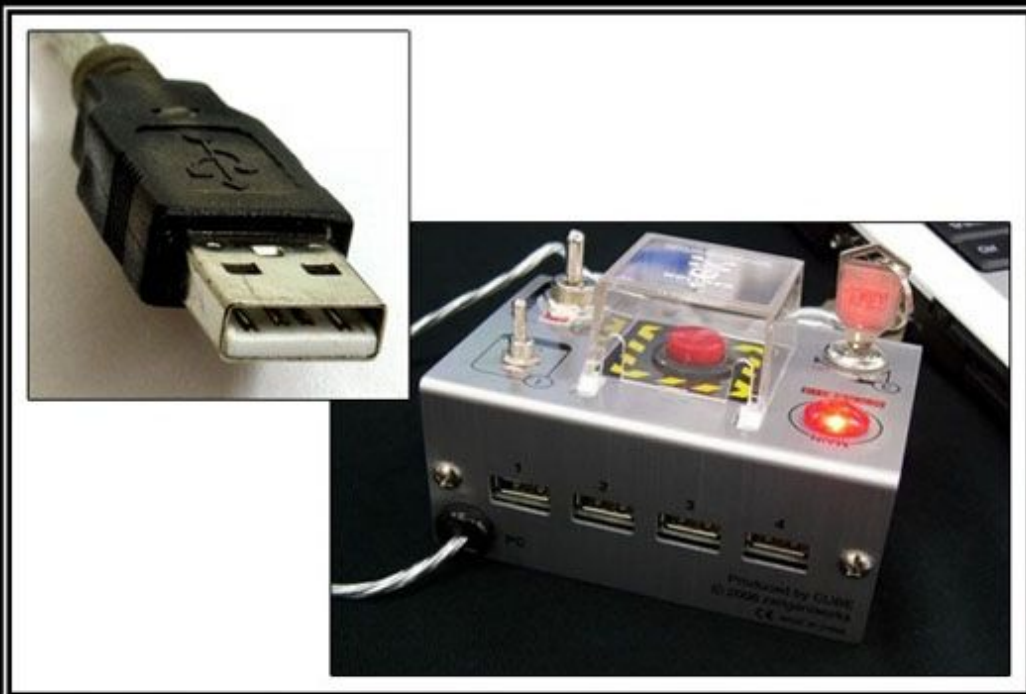
Exemplu LSP

Concluzie LSP

- Conformarea la LSP necesită o utilizare corectă a polimorfismului și produce cod mai ușor de întreținut
- Nu uitați: “is-substitutable-for” în loc de “is-a”

Interface Segregation Principle (ISP)

- *Nici un client nu trebuie forțat să depindă de metode pe care nu le utilizează*
- Mai multe interfețele mici, specifice pentru clienți, coezive sunt mai bune decât interfețe mari cu responsabilități multiple
- Segregarea interfețelor conduce la decuplarea funcționalităților



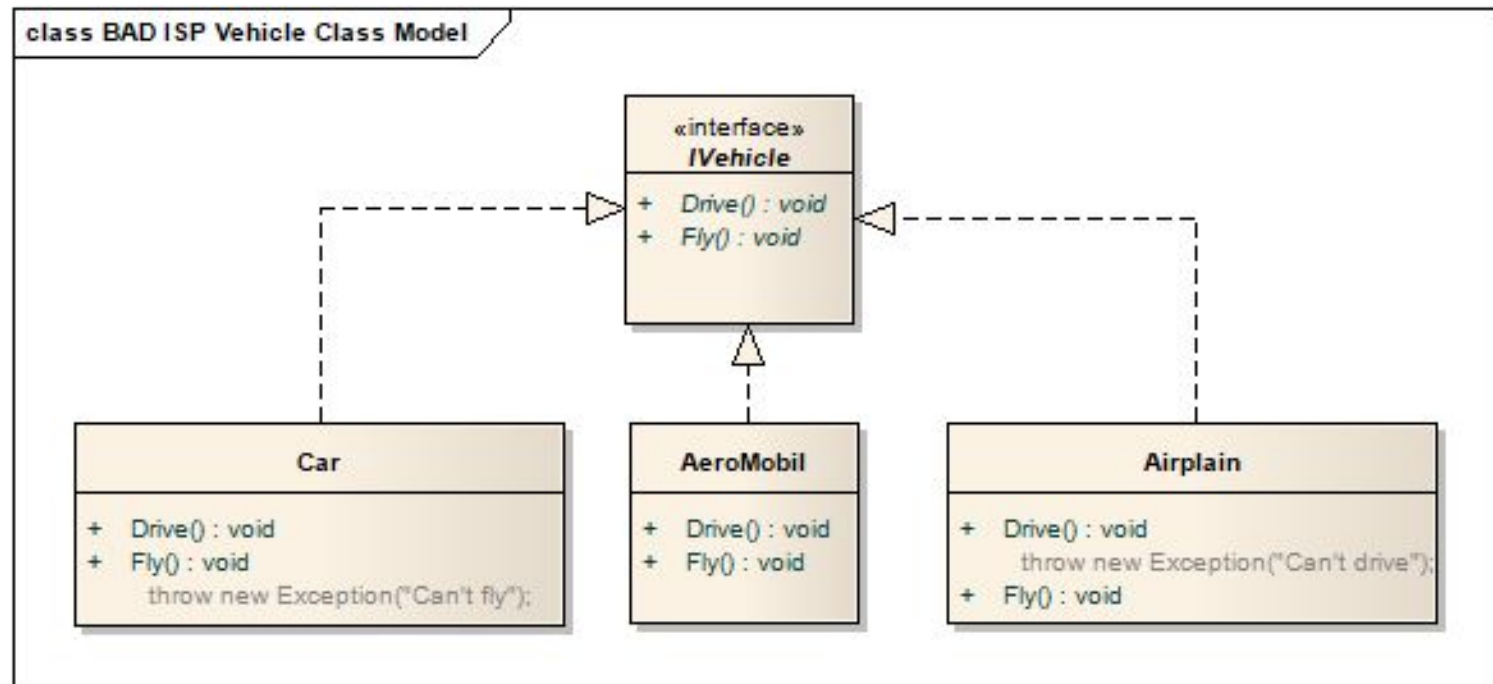
INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

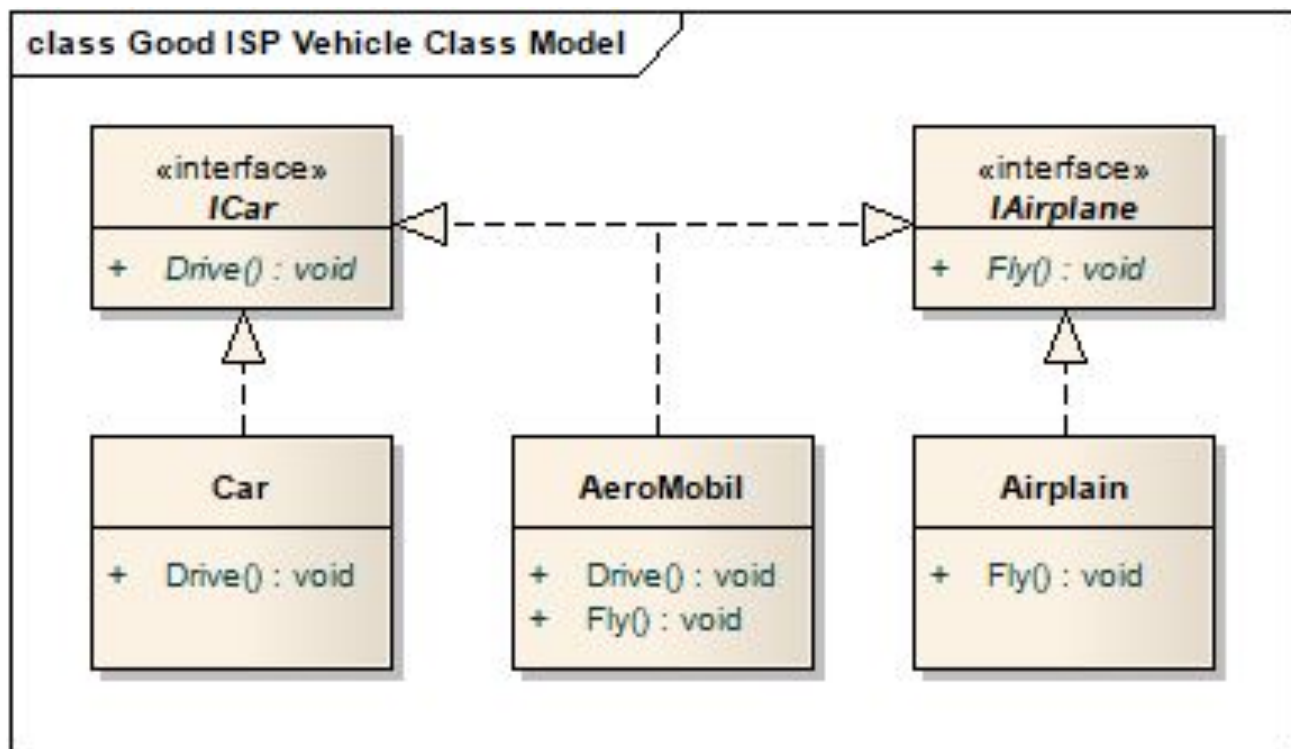
Ce este interfața?

- Un set de metode care trebuie implementate de o clasă care realizează interfața
- Metodele publice ale unei clase
- Interfața este cea ce poate vedea și folosi clientul

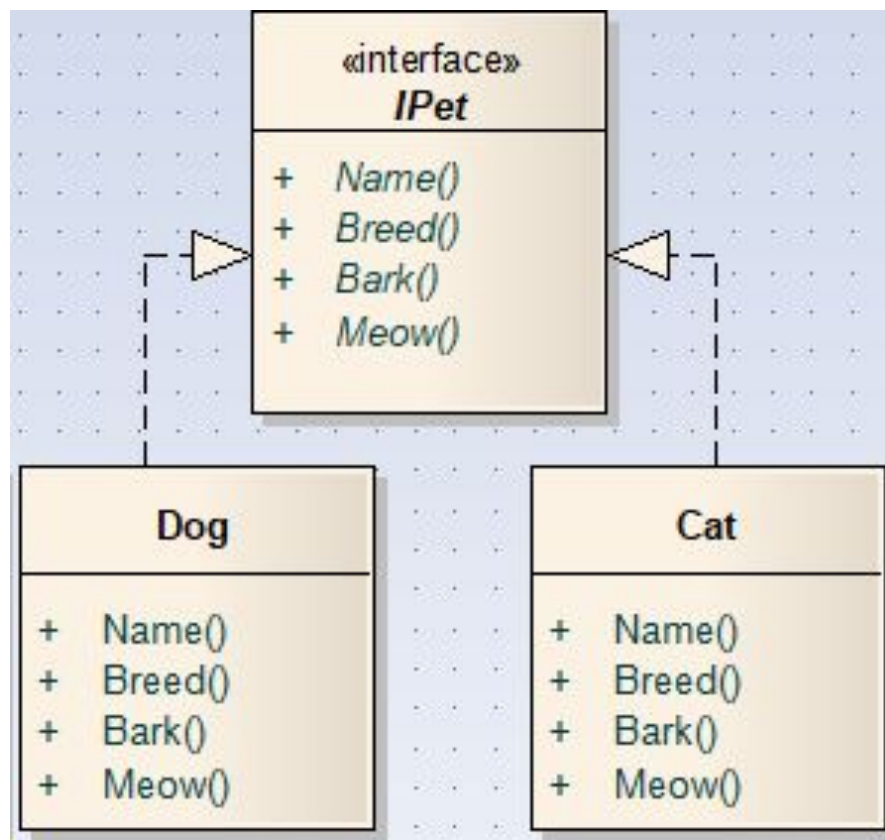
Mașina zburătoare



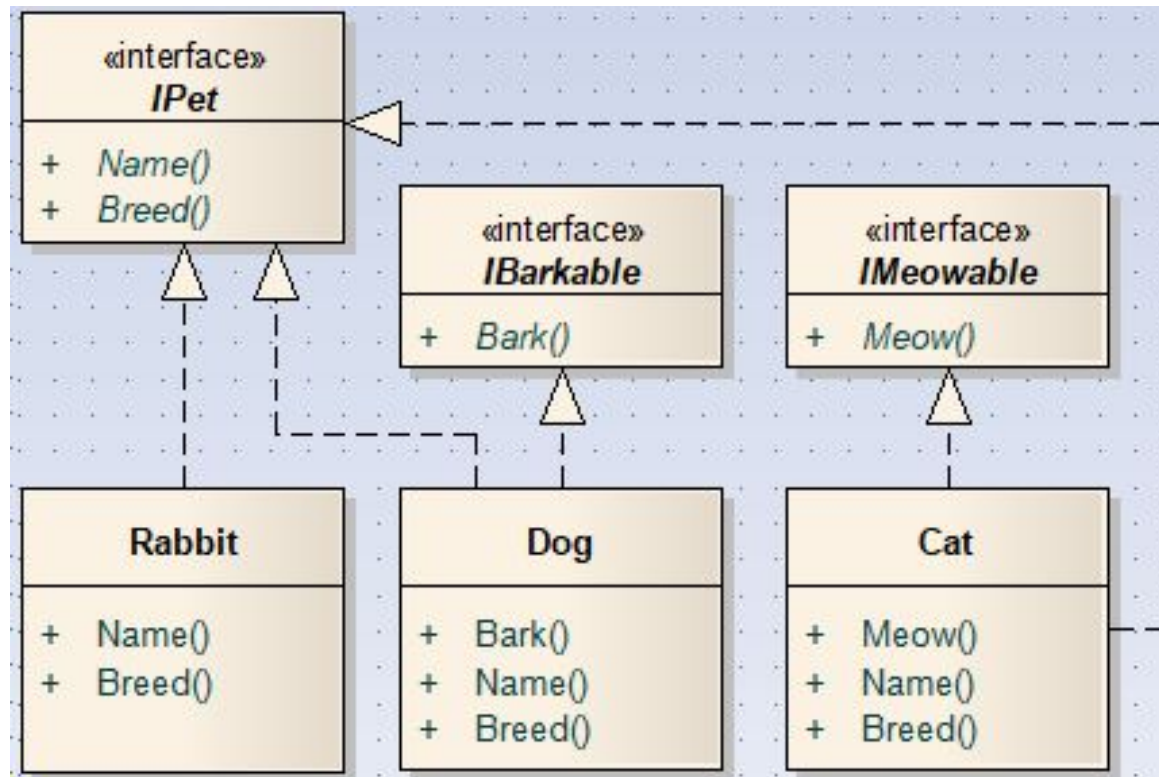
Mașina zburătoare



Pisica care latră



Pisica miaună și câinele latră



Exemplu ISP

Concluzie ISP

- Nu forța codul client să depindă de clase de care nu e nevoie
- Păstrează interfețele lean și focusate
- Refactorizează interfețele “mari” și “spargele” în interfețe mai mici

Dependency Inversion Principle (DIP)

- *Modulele de nivel înalt nu ar trebui să depindă de modulele de nivel scăzut, și ambele ar trebui să depindă de abstractizare*
- Abstractizarea nu trebuie să depindă de detalii
- Detaliile trebuie să depindă de abstractizare
- Abstractizarea decuplează modulele



DEPENDENCY INVERSION PRINCIPLE

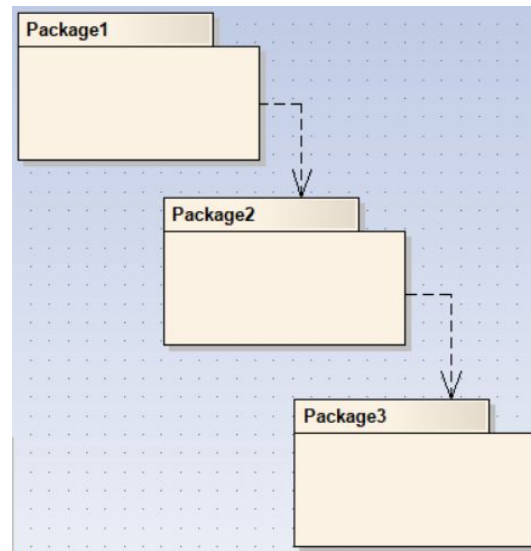
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Ce sunt dependențele?

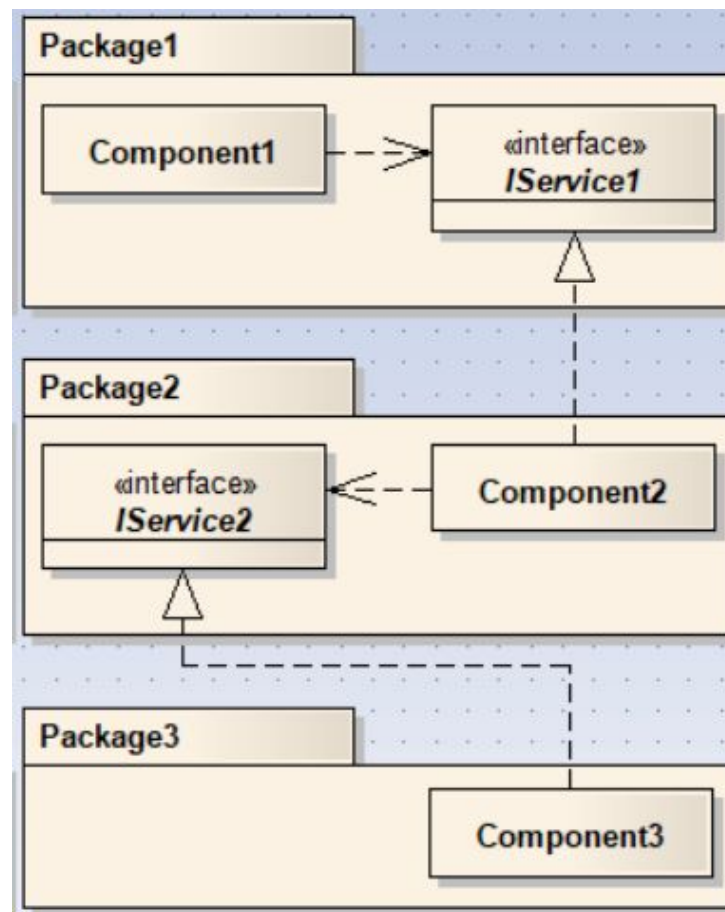
- Platforma
 - .NET Framework
 - .NET Core
 - Java SDK
 - Angular
- Biblioteci grafice
 - GDI
 - OpenGL
 - Direct3D
- Baza de date
- Sistemul de fișiere
- Email
- Web service
- Resurse de sistem
- Generatorul de numere random
- Bibliotecile externe
- API - Application Programming Interface
- SDK - Software Development Kit
 - ARCore
 - DirectX

Dependențele în programarea tradițională

- Modulele de nivel înalt apelează modulele de nivel scăzut
- Interfața cu utilizatorul depinde de:
 - Logica de business depinde de:
 - Module utilitare
 - Module de infrastructură
 - Baza de date
- Instanțierea claselor se face prin toate modulele și se încalcă SRP



Soluția problemei DIP



Dependency Injection

- Tehnică folosită pentru a permite codului apelant să injecteze dependența de care are nevoie o clasă atunci când este creată instanța
- Tehnici principale:
 - Injectia în constructor
 - Injectia în proprietate
 - Injectia în parametru

Fără injecție

```
class Sword {  
    public void Hit(string enemy) {  
        Console.WriteLine($"Spliced {enemy} in  
pieces");  
    }  
}
```

```
class Soldier {  
    private readonly Sword sword;  
    public Soldier() {  
        this.sword = new Sword();  
    }  
    public void Attack(string enemy){  
        this.sword.Hit(enemy);  
    }  
}
```


Injectia în constructor

- Dependențele sunt transmise pe constructor
- Pro:
 - Clasa e într-o stare validă după creare
- Cons:
 - Constructorul poate avea mulți parametri/dependențe
 - Unele funcționalități (Serializarea) pot necesita constructor default

Ex: Injecție în constructor

```
interface IWeapon {
```

```
    void Hit(string enemy);
```

```
}
```

```
class Sword : IWeapon {
```

```
    public void Hit(string enemy) {
```

```
        Console.WriteLine($"Spliced {enemy} in pieces");
```

```
    }
```

```
}
```

```
class Soldier {
```

```
    private readonly IWeapon weapon;
```

```
    public Soldier(IWeapon weapon) {
```

```
        this.weapon = weapon;
```

```
    }
```

```
    public void Attack(string enemy){
```

```
        this.weapon.Hit(enemy);
```

```
    }
```

```
}
```

Injectia în proprietate

- Dependențele sunt transmise folosind proprietăți
 - Se mai numește “setter injection”
- Pro:
 - Dependența se poate schimba oricând pe perioada existenței obiectului
- Cons:
 - Obiectul poate fi într-o stare invalidă între construcție și apelul setter-ului

Ex: Injecție în proprietate

```
interface IWeapon {
```

```
    void Hit(string enemy);
```

```
}
```

```
class Sword : IWeapon {
```

```
    public void Hit(string enemy) {
```

```
        Console.WriteLine($"Spliced {enemy} in pieces");
```

```
    }
```

```
}
```

```
class Soldier {
```

```
    private IWeapon weapon;
```

```
    public Soldier(IWeapon weapon) {
```

```
        SetWeapon(weapon);
```

```
    }
```

```
    public void SetWeapon(IWeapon weapon) {
```

```
        this.weapon = weapon;
```

```
    }
```

```
    public void Attack(string enemy){
```

```
        this.weapon.Hit(enemy);
```

```
    }
```

```
}
```

Injectia în parametru

- Dependențele sunt transmise folosind parametrul unei metode
- Pro:
 - Nu necesită schimbări pentru restul clasei
- Cons:
 - Schimbă semnătura metodei
 - Metoda poate avea prea mulți parametri

Ex: Injecție în parametru

```
interface IWeapon {
```

```
    void Hit(string enemy);
```

```
}
```

```
class Sword : IWeapon {
```

```
    public void Hit(string enemy) {
```

```
        Console.WriteLine($"Spliced {enemy} in pieces");
```

```
    }
```

```
}
```

```
class Soldier {
```

```
    public Soldier() {
```

```
    }
```

```
    public void Attack(IWeapon weapon, string enemy){
```

```
        weapon.Hit(enemy);
```

```
    }
```

```
}
```

Exemplu DIP

Concluzie DIP

- Să depinzi de abstractizare nu de clase concrete
- Nu forța modulele de nivel înalt să depindă de modulele de nivel scăzut prin instanțiere directă sau prin apeluri de metode statice
- Injectează dependența prin injecția în constructor, proprietate sau parametru

Beneficiile aplicării principiilor SOLID

- Se reduce complexitatea codului
- Crește lizibilitatea, extensibilitatea și mentenabilitatea
- Se reduce numărul erorilor
- Se reutilizează codul
- Se obține o mai bună testabilitate
- Se reduce cuplarea strânsă

Materiale suplimentare

Principles of Object Oriented Design by Uncle Bob

Design Principles: <https://www.oodeesign.com/design-principles.html>

SOLID Design Principles Explained

The SOLID principles of Object Oriented Design

SOLID Development Principles – In Motivational Pictures

Solid Design Principle in C#

Concluzii finale

- Aplicând principiile SOLID, obținem un cod robust, ușor de întreținut, reutilizabil, sustenabil, scalabil și ușor de testat, iar codul nu tinde să se degradeze când ceva se schimbă
- Aceste 5 principii esențiale sunt folosite de inginerii software din întreaga lume și, dacă doriți să construiți software "solid", ar trebui să începeți să aplicați aceste principii

Bibliografie

- Agile Principles, Patterns, and Practices in C# – Hardcover, Amazon.com – Robert C. Martin, Micah Martin
- <https://druss.co/wp-content/uploads/2013/10/Agile-Principles-Patterns-and-Practices-in-C.pdf>
- SOLID Principles of Object Oriented Design – Pluralsight Online Training – Steve Smith

Întrebări



Feedback

- Formular online: <http://bit.ly/peak-it-2019-feedback>

- Completați în sală

- Durează 2-3 minute

- Este anonim

