A S.O.L.I.D. alapelvek

Jánosi-Rancz Katalin Tünde

Sapientia EMTE
tsuto@ms.sapientia.ro

A sikertelen alkalmazások okai

- nagyobb felelőséget fektetünk az osztályokra, (sok funkció nem kapcsolódik egy osztályhoz)
- az osztályok egymástól függenek (szorosan összekapcsolva), akkor az egyik változása hatással lesz a másikra
- ismétlődő kódok az alkalmazásban

S.O.L.I.D. alapelvek

- a kód karbantarthatóbb legyen, szülőatyja Robert C. Martin

- **S-** Single Responsibility Principle (Egy felelősség elve)
  - Egy osztály vagy modul egy, és csak egy felelősséggel rendelkezzen (azaz: egy oka legyen a változásra)
- **O-** Open/Closed Principle (Nyílt/zárt elv)
  - Egy osztály, vagy modul, legyen nyílt a kiterjesztésre, de zárt a módosításra.
- **L-** Liskov substitution principle (Liskov helyettesítési elv)
  - Minden osztály legyen helyettesíthető a leszármazott osztályával anélkül, hogy a program helyes működése megváltozna
- **I-** Interface segregation principle (Interface elválasztási elv)
  - Több specifikus interface jobb, mint egy általános
- **D-** Dependency inversion principle (Függőség megfordítási elv)
  - A kódod függjön absztrakcióktól, ne konkrét implementációktól

SOLID - S

S: Single Responsibility Principle (SRP) -Egy felelősség elve

- Egy osztály, vagy modul, egy, és csak egy felelősséggel rendelkezzen (azaz: egy oka legyen a változásra).

S: Single Responsibility Principle (SRP) - Egy felelősség elve

```
class Student {
public void addGrade(Subject subject, int grade) { }
public void setName(string name) { }
}
```

- a jegy beírást a tanár végzi, amíg a név változtatást a titkárság. Egy osztályban folyik össze két felelősség. Egy osztályt kell adott esetben módosítani két külön helyről érkező kérések alapján.
- Megoldás: Student osztály csak egy adattároló legyen, és a különböző feladatoknak külön osztályai legyenek:

```
class Student {
}
class GradeBook {
public void addGrade(Student student, int grade) { }
}
class StudentRecords {
public void changeStudentName(Student student, string newName) { }
}
```

S: Következtetés

- a felelősség lehet például:
  - Egy felhasználó vagy felhasználói csoport
  - Egy külső szolgáltatás (pl. adatbázis, API, fileba írás stb)
  - A felhasználói felület vagy azok elemei
  - … és még sok minden más.

NB: Az ORM vagy ActiveRecord pattern megsérti ezt az elvet!

SOLID - O

O: Open/Closed Principle - Nyílt/zárt elv

- Egy osztály, vagy modul, legyen nyílt a kiterjesztésre, de zárt a módosításra

O: Open/Closed Principle - Nyílt/zárt elv

```
public class Rectangle{
public double Height {get;set;}
public double Width {get;set; }
}
```

```
public class AreaCalculator {
public double TotalArea(Rectangle[] arrRectangles)
{
double area;
foreach(var objRectangle in arrRectangles)
{
area += objRectangle.Height * objRectangle.Width;
}
return area;
}
}
```

- Mi van ha Körnek vagy más alakzatnak akarok Területet számolni?

O: Open/Closed Principle - Nyílt/zárt elv

```
public class Rectangle{
public double Height {get;set;}
public double Width {get;set; }
 }
```

```
public class Circle{
public double Radius {get;set;}
 }
```

```
public class AreaCalculator
{
public double TotalArea(object[] arrObjects)
{
double area = 0;
Rectangle objRectangle;
Circle objCircle;
foreach(var obj in arrObjects)
{
if(obj is Rectangle)
{    area += obj.Height * obj.Width;  }
else
{   objCircle = (Circle)obj;
area += objCircle.Radius * objCircle.Radius * Math.PI;  }
}
return area;
}   }
```

Tipp: ha működést írsz le, használj interface-t! (Absztrakt)

```
public abstract class Shape
{  public abstract double Area();  }
```

```
 public class Rectangle: Shape
{
public double Height {get;set;}
public double Width {get;set;}
public override double Area()
{
return Height * Width;   }    }
```

```
public class Circle: Shape
{
public double Radius {get;set;}
public override double Area()
{
return Radius * Radus * Math.PI;
} }
```

```
public class AreaCalculator
{
public double TotalArea(Shape[] arrShapes)
{
double area=0;
foreach(var objShape in arrShapes)
{
area += objShape.Area();
}
return area;
} }
```

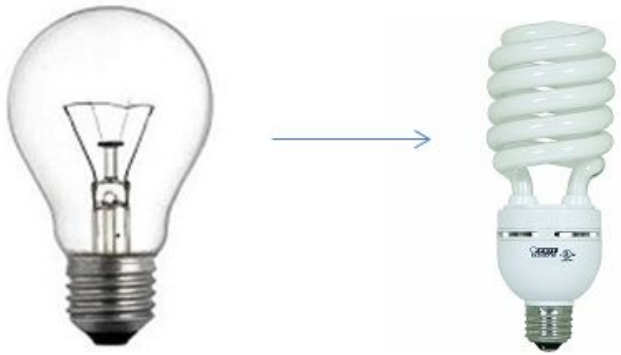
SOLID - L

L: Liskov Substitution Principle (LSP) - Liskov helyettesítési elv

- Minden osztály legyen helyettesíthető a leszármazott osztályával anélkül, hogy a program helyes működése megváltozna.

L: Liskov Substitution Principle - Liskov helyettesítési elv

- Minden leszármazott osztály ugyanazt a funkcionalitást kell biztosítsa mint az ősösztály
- Jó



- Rossz

SOLID - I

I: Interface Segregation Principle -Interface elválasztási elv

- Több specifikus interface jobb, mint egy általános.

I: Interface Segregation Principle

```
public Interface ILead
{
void CreateSubTask();
void AssignTask();
void WorkOnTask();
}
```

```
public class TeamLead : ILead
{
public void AssignTask()
{
//Code to assign a task.
}
public void CreateSubTask()
{
//Code to create a sub task
}
public void WorkOnTask()
{
//Code to implement perform assigned task.
}
}
```

I: Interface Segregation Principle

```
public class Manager: ILead
{
public void AssignTask()
{
//Code to assign a task.
}
public void CreateSubTask()
{
//Code to create a sub task.
}
public void WorkOnTask()
{
throw new Exception(”Manager can’t work on Task”);
}
}
```

I: Interface Segregation Principle

```
public interface IProgrammer
{
void WorkOnTask();
}
```

```
public interface ILead
{
void AssignTask();
void CreateSubTask();
}
```

I: Interface Segregation Principle

```
public class Programmer: IProgrammer
{    public void WorkOnTask()
 {  //code to implement to work on the Task.  }  }
```

```
public class Manager: ILead
{
public void AssignTask()
{  //Code to assign a Task  }
public void CreateSubTask()
{  //Code to create a sub taks from a task. }  }
```

```
public class TeamLead: IProgrammer, ILead
{
public void AssignTask()
{  //Code to assign a Task  }
public void CreateSubTask()
{  //Code to create a sub task from a task.  }
public void WorkOnTask()
{  //code to implement to work on the Task.  }  }
```

SOLID - D

Dependency

Dependency Property

- (ezt mondtuk a Data Bindingnál: a célobjektum függőségi tulajdonság (**Dependency Property**) kell hogy legyen)
- **Dependency Property** egy olyan speciális tulajdonság, amelynek az értéke más objektumtól érkező adatok alapján kerül dinamikusan kiszámításra
- lehetővé teszi, hogy egy adott objektum tulajdonságait más objektumon keresztül definiáljuk, és úgy szabjunk rá értéket, hogy az környezettől függően változzon
- a tulajdonság értéke függhet a környezet sajátosságaitól is (pl. az operációs rendszer)
- olyan kód is kezelheti ezeket, amelyek nem ismerik a WPF –et

Dependency Property

- értékét nem feltétlenül a **szokásos** módon kapja, hanem:
  - adatkötésből
  - tartalmazó elemek tulajdonság-értékének „örökléséből”
  - animációból
  - stílusból, erőforrásból
  - rendszerbeállításból (témák, felhasználói beállítások)

Dependency Property

- a legtöbb tulajdonság WPF-ben függőségi tulajdonság
  - pl. lehetőséget ad a szülőelemek tulajdonságainak elérése, és beállítására

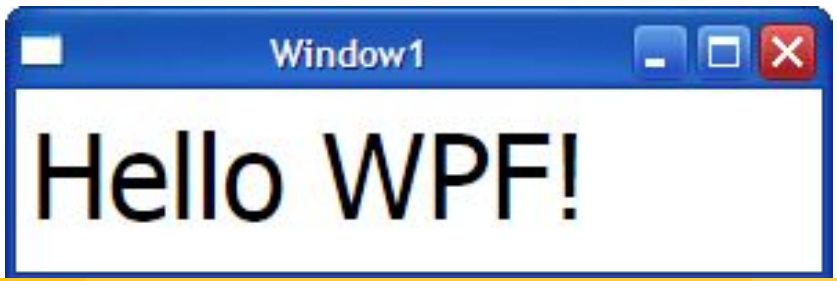
```
<Canvas> <!-- vászon -->
    <Label Content="Hello!" Canvas.Left="100" Canvas.Top="50" />
<!-- a címkében állítjuk be a vászonbeli pozíciót -->
```

Dependency Property

```
<Window
...
Title=”Window1” Height=”100” Width=”300”>
<Grid>
    <Label FontSize=”42”> Hello WPF! </Label>
</Grid>
</Window>
```

```
<Window
...
Title=”Window1” Height=”100” Width=”300” FontSize=”42”> //itt a FontSize
<Grid>
    <Label> Hello WPF! </Label>
</Grid>
</Window>
```

- eredmény ugyanaz



FontSize tulajdonság

- a statikus GetValue és SetValue metódusaival kezelhetőek a tulajdonság átadásával, amely statikus tulajdonságként definiált

```
[TypeConverter(typeof(FontSizeConverter)), Localizability(
     LocalizationCategory.None)]
public double FontSize
{
     get
     {
      return (double) base.GetValue(FontSizeProperty);
     }
     set
     {
      base.SetValue(FontSizeProperty, value);
     }
}

[CommonDependencyProperty]
public static readonly DependencyProperty FontSizeProperty;
```

FontSize tulajdonság -2

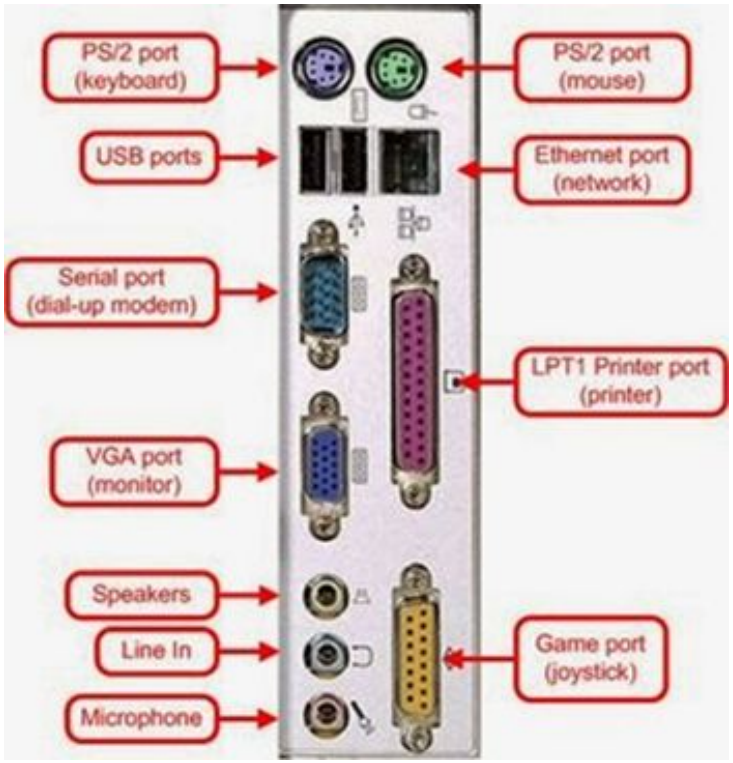
- a SetValue meghívásakor az alárendelt objektumok betűmérete is változik
- a Label betűmérete is megváltozik

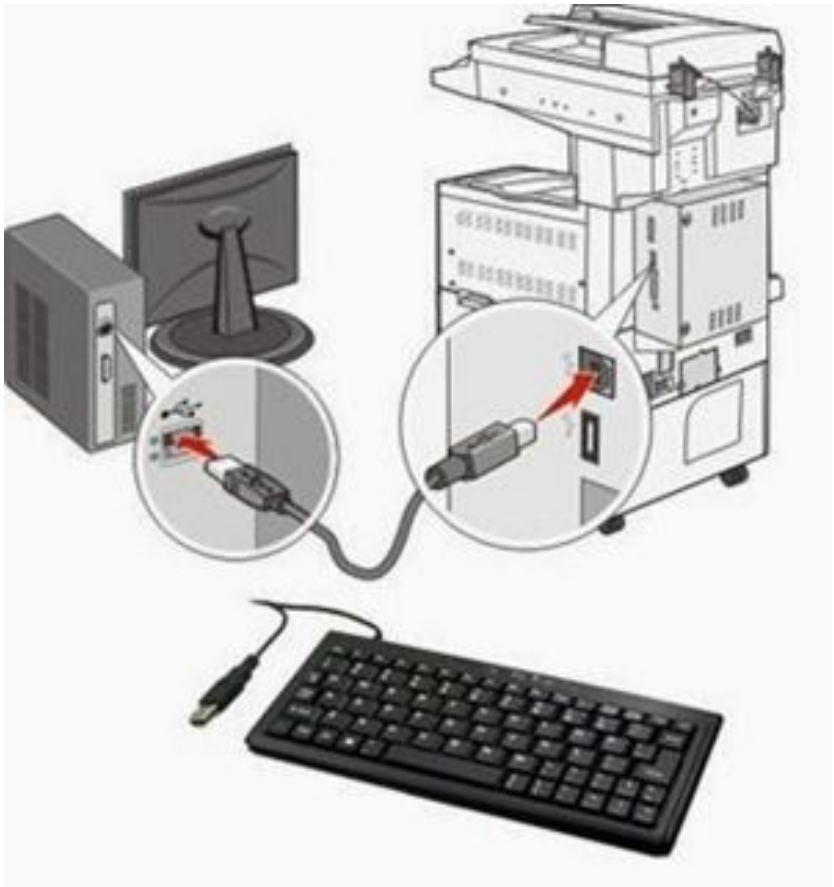
```
private void BtnChangeFontSize_Click(object sender, RoutedEventArgs e)
{
    window1.FontSize = 10;
}
```

NB. saját Dependency Property is létrehozható, lásd Code/MVVMPart2

Dependency Injection - Függőség injektálás

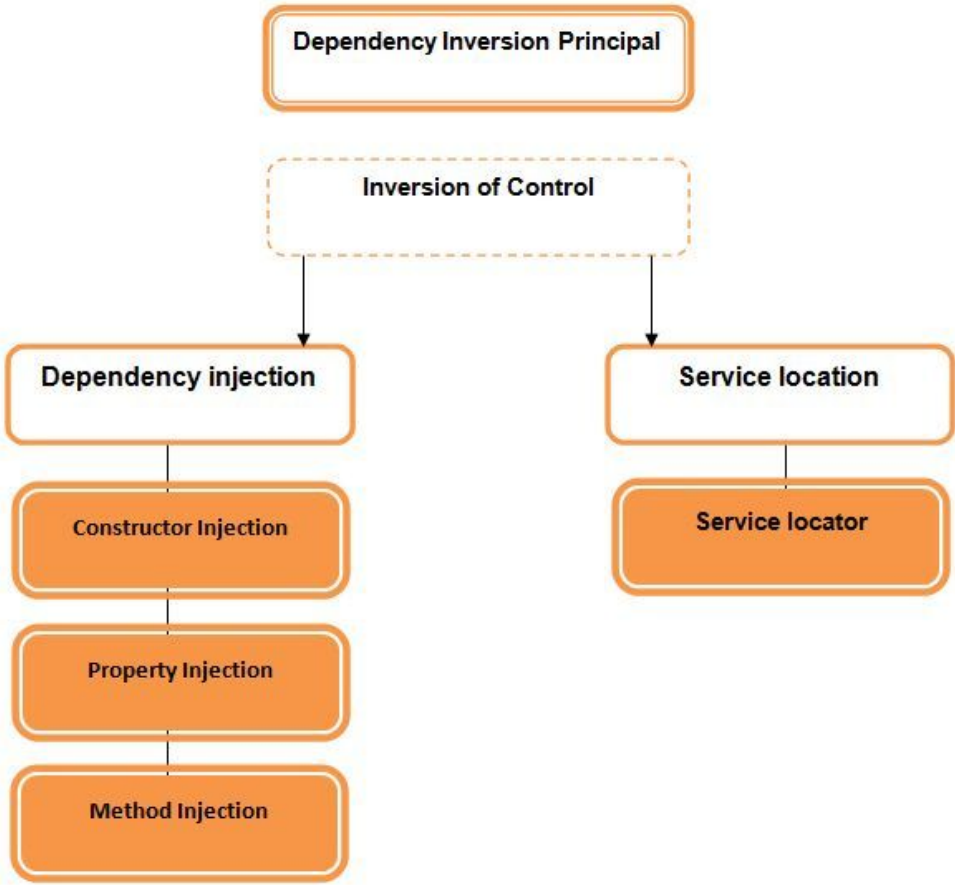
Dependency Inversion Principle





Dependency Injection

- az architektúra akkor megfelelő, ha az egyes rétegek között minél kisebb a függőség (**loose coupling**)
  - egyik réteg sem függhet a másik konkrét megvalósításától, és nem avatkozhat be a másik működésébe
  - ennek eléréséhez függőség befecskendezést (**dependency injection**) használunk
  - nem az adott komponens, hanem a környezet dönti el, hogy a függőségek mely megvalósításai kerülnek alkalmazásra (**Inversion of Control, IoC**)



Dependency Injection

- a **Dependency** két osztály közötti kapcsolatra utal, például az ”Egyik” osztályunk függ a ”Masik” osztályunktól
- a Dependency nem más, mint egy másik objektum, amelynek az egyik funkciójára az osztályodnak szüksége van
- Injection: ”Masik” osztályt az ”Egyik” osztályba fogjuk beinjektálni
- függőséget beinjektálni annyit jelent, hogy a függőség be van ”push”-solva, be van hozva az osztályunkba kívülről
- **Dependency Injection** azt jelenti, hogy egy osztály függőségeinek létrehozását és azok konfigurálását leválasztjuk az osztályról

Dependency Injection

- az injektálás és maga a DI több féle módon megvalósítható:
  - Konstruktor paraméterezésében
  - Setter metódusokkal
- nem a model osztályomban inicializálom a külső objektumot new operátor segítségével, hanem ehelyett egy már kész függőség objektumot kapok bemeneti paraméter, vagy egy setter változó révén

DI példa

- adott n darab pont a metrikus térben, keressük meg azt a két pontot, amelyek távolsága a legkisebb

```
public class Point {
    private double[] coordinates;
    public Point(double[] coordinates) {
        this.coordinates = coordinates;
    }
    public double[] getCoordinates() {
        return coordinates;
    }}
```

```
public class EuclideanMetric {

    public double metric(double[] x, double[] y) {
        if (x.length != y.length) {
            ...
        }
        double metric = 0.0;
        for (int i = 0; i < x.length; i++) {
            double diff = x[i] - y[i];
            metric += diff * diff;
        }
        metric = Math.sqrt(metric);
        return metric;
    }}
```

DI példa-2

```
public class ClosestPairAlgorithm {

     EuclideanMetric metric = new EuclideanMetric(); //saját maga példányosítja!
    //nem adtunk lehetoséget arra, hogy az EuclideanMetric mellet más távolság számító osztályok
         is létezhessenek
    //mi van ha ez ABContext, hogy tesztelem?

    public ClosestPair calculate(List<Point> points) {
        ClosestPair closestPair = null;
        double minDist = Double.POSITIVE_INFINITY;
        for (int i = 0; i < points.size() - 1; i++) {
             Point p = points.get(i);
            for (int j = i + 1; j < points.size(); j++) {
                 Point q = points.get(j);
                 double dist = dist(p, q);
                if (dist < minDist) {
                    minDist = dist;
                    closestPair = new ClosestPair(p, q);
                } }  }
        return closestPair;
    }
    private double dist(Point p, Point q) {
        return metric.metric(p.getCoordinates(), q.getCoordinates());
    }
    public static class ClosestPair {
        private Point a, b;
        public ClosestPair(Point a, Point b) {
            this.a = a;
            this.b = b;
        }
        public Point getA() {
            return a;
        }
        public Point getB() {
            return b;
       } }}
```

Megoldás - lépés 1

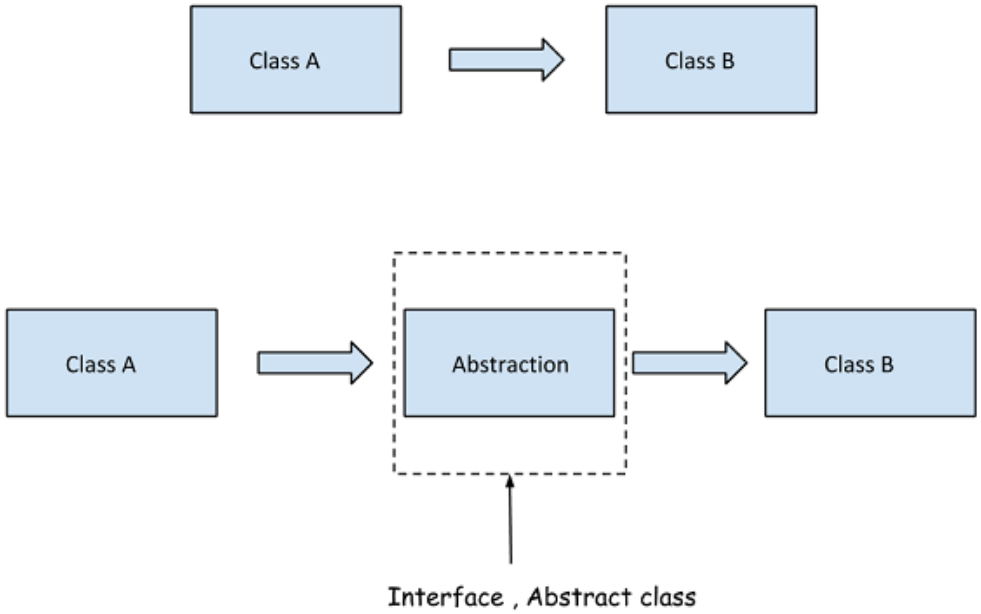
- ha nem ClosestPairAlgorithm végzi az EuclideanMetric példányosítását, hanem konstruktorban várja

```
public class ClosestPairAlgorithm {

    private EuclideanMetric metric;

    public ClosestPairAlgorithm(EuclideanMetric metric) {
        this.metric = metric;
    }
// ...
}
```

Megoldás - lépés 2



Megoldás - lépés 2

- ha követjük a program to interface not implementation elvet és készítünk egy Metric nevű interface-t:

```
public interface Metric {

    double metric(double x[], double y[]);
}
public class EuclideanMetric : Metric {

    @Override
    public double metric(double x[], double y[]) {
        // ...
    }}
public class ClosestPairAlgorithm {

    private Metric metric;

    public ClosestPairAlgorithm(Metric metric) {
        this.metric = metric;
    }
// ...}
```

- újrafelhasználható kódot kaptunk

Dependency Injection előnyei

- csökkenti az osztályok közötti csatolást
  - azáltal, hogy nem az osztály gondoskodik függőségeinek példányosításáról a szoftverünk osztályai lazábban lesznek csatoltak
- növeli a kód újrafelhasználhatóságot
  - mivel az osztályunk a függőségeinek konfigurálásától is mentes, így különféle helyzetekben másként tudjuk felhasználni
- növeli a kód karbantarthatóságát
  - az osztály függőségei könnyedén felfedezhetőek, ezáltal áttekinthetőbb, olvashatóbb kódot kapunk
- elősegíti a tesztelést
  - a függőségek így a tesztekben könnyen mockolhatókká válnak, s a teszt függetlenedhet az osztály függőségeinek működésétől (a függőségek implementációjában történő változtatások, nem vezetnek az osztályunk tesztjének bukásához, lokalizáltá válik a teszt)

Dependency Injection hátrányai

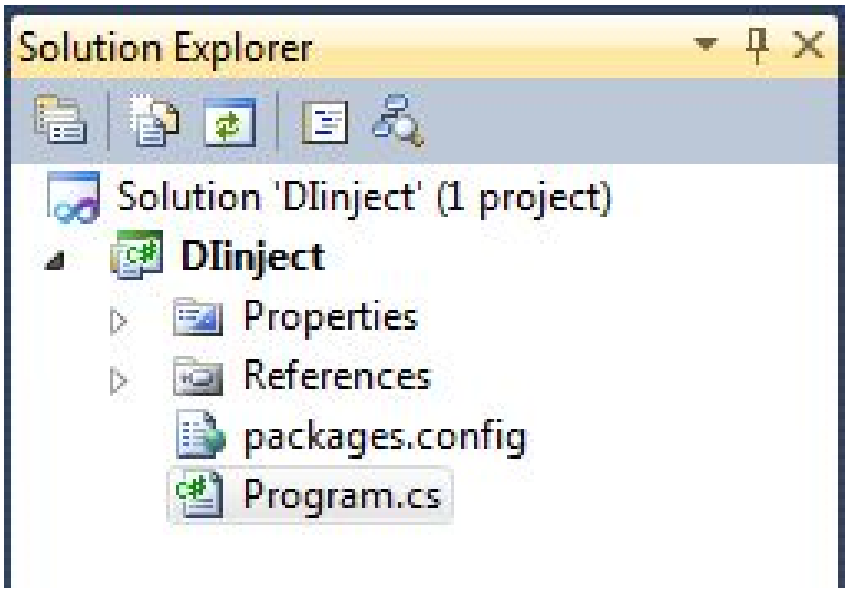
- Növeli a kód komplexitást
- komplikálja a debugolást

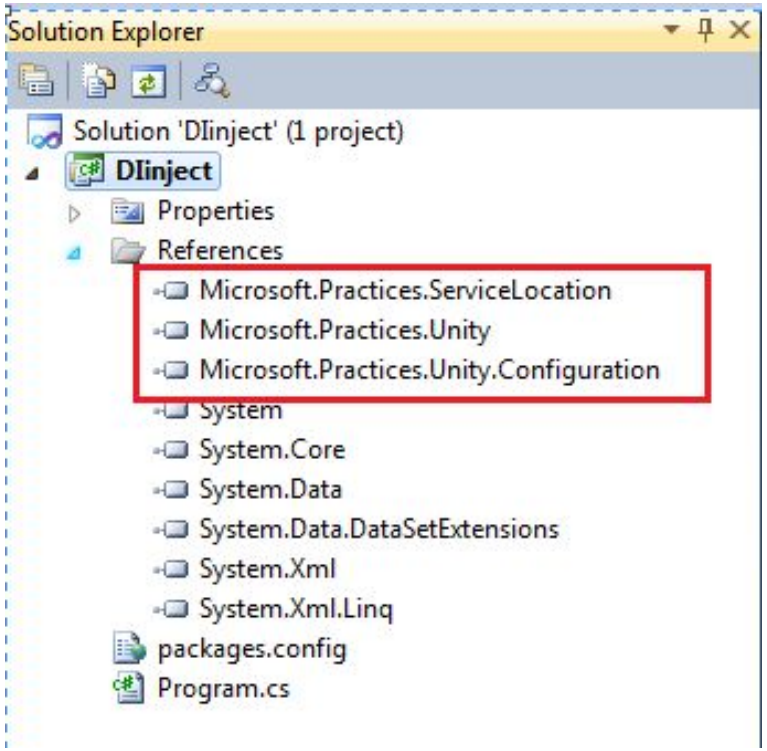
DI keretrendszerek

- DI keretrendszerek: Microsoft Unity, Ninject, Caliburn, Managed Extensibility Framework

Dependency Injection Unity frameworkot használva

- Console application **DIinject**
- Adding Reference to Microsoft Unity Framework





Dependency Injection Unity frameworkot használva

- **BL** Business Logic Layer hozzáadása
- **DL** Data Access Layer hozzáadása

BL.cs*

DInject.BL

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5
6  namespace DIinject
7  {
8      public class BL
9      {
10
11
12     }
13 }
14
```

DL.cs*

DInject.DL

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5
6  namespace DIinject
7  {
8      public class DL
9      {
10
11     }
12 }
13
```

Dependency Injection Unity frameworkot használva

- **IProduct** interfész, **Insertdata()** metódussal
- **DL** (Data access layer) implementálja az IProduct interfészt

IProduct.cs*

DIinject.IProduct

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5
6  namespace DIinject
7  {
8      public interface IProduct
9      {
10         string Insertdata();
11     }
12 }
13
```

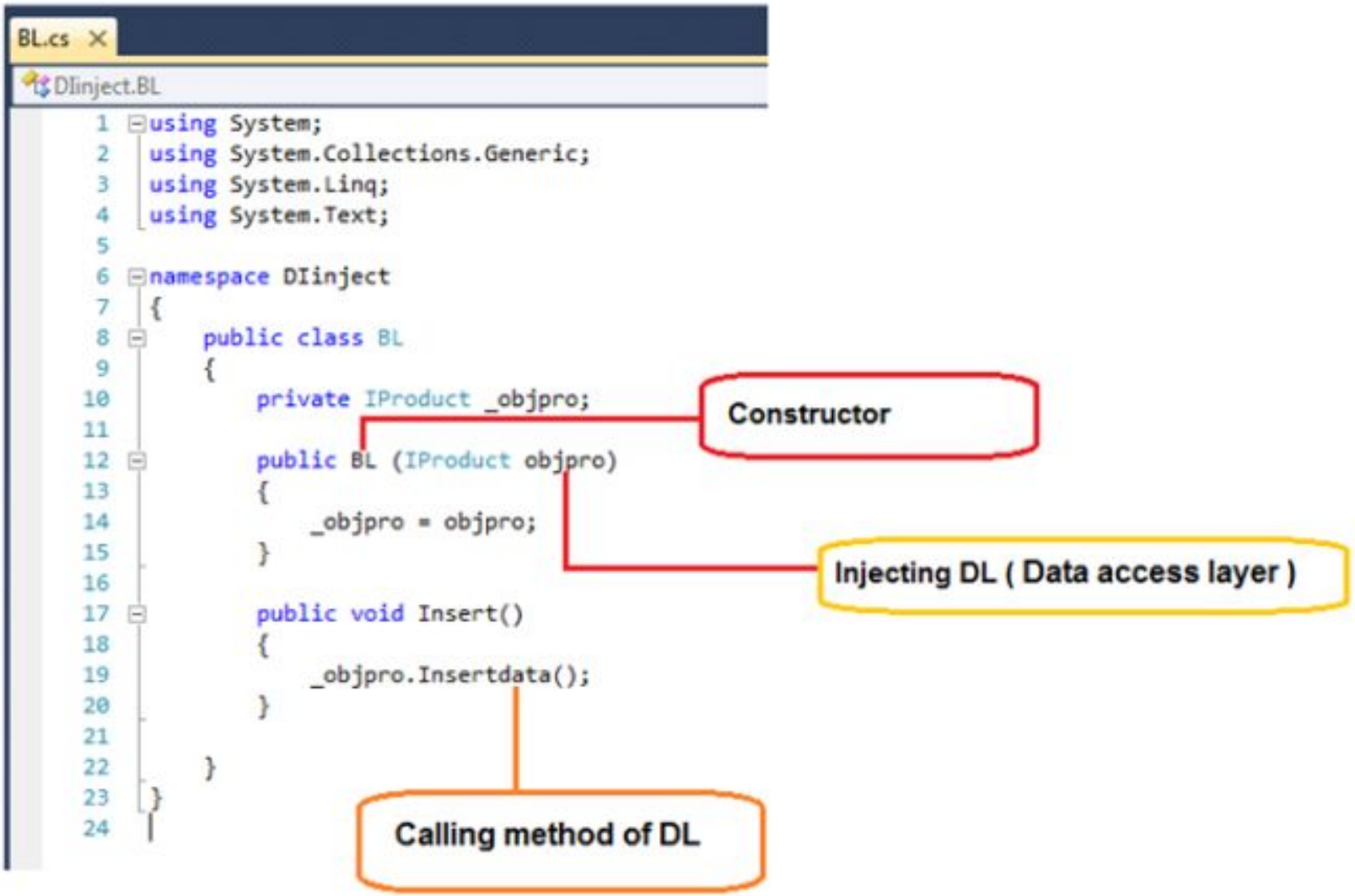
DL.cs

DIinject.DL

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5
6  namespace DIinject
7  {
8      public class DL : IProduct
9      {
10         public string Insertdata()
11         {
12             string val = "Dependency inject
13
14             Console.WriteLine(val);
15
16             return val;
17         }
18     }
19 }
20
```

Dependency Injection Unity frameworkot használva

- függőség injektálás a BL (Business Logic) konstruktorában



Unity Container konfigurálása

1. Unity Container létehozása

```
UnityContainer IU = new UnityContainer();
```

2. típus regisztrálása (Injektálni DA-t BL-be)

```
IU.RegisterType<BL>();

IU.RegisterType<DL>();
```

3. típus regisztrálása specifikus injektálandó taggal

```
IU.RegisterType<IProduct, DL>();
```

```
RegisterType<From , To>( );
```

4. Resolve, a kért típus példányának megoldása a tárolóból

```
BL objDL = IU.Resolve<BL>();
```

5. metódus meghívása

```
BL objDL = IU.Resolve<BL>();
objDL.Insert(); // BL Method
```

Program.cs

Program.cs*

DIinject.Program — Main(string[])

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using Microsoft.Practices.Unity;
6
7  namespace DIinject
8  {
9      class Program
10     {
11         static void Main(string[] args)
12         {
13
14             /* Creating microsoft unity Container*/
15             UnityContainer IU = new UnityContainer();
16
17             /* Register a type*/
18               IU.RegisterType<BL>();
19               IU.RegisterType<DL>();
20             /* Register a type with specific members to be injected. */
21
22             IU.RegisterType<IProduct, DL>();
23
24             BL objDL = IU.Resolve<BL>();
25             objDL.Insert(); // BL Method
26             Console.ReadLine();
27         }
28     }
```

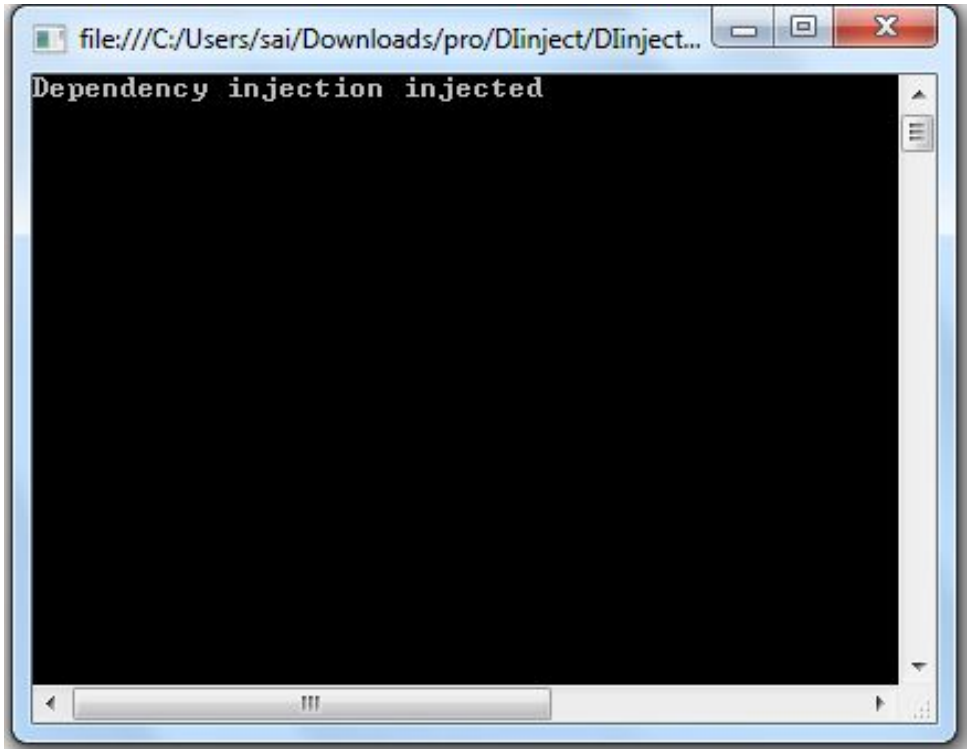
Futtatás

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5
6  namespace DIinject
7  {
8      public class BL
9      {
10         private IProduct _objpro;
11
12         public BL (IProduct objpro)
13         {
14             _objpro = objpro;
15         }
16
17         public void Insert()
18         {
19             _objpro.Insertdata();
20         }
21
22     }
23 }
24
25
```

objpro {DIinject.DL}

((DIinject.DL)(_objpro)) {DIinject.DL}

Eredmény


file:///C:/Users/sai/Downloads/pro/DIinject/DIinject...
Dependency injection injected

Zárszó

- A S.O.L.I.D. elvek nem eredményezik azt, hogy a kód mindörökké karbantartható lesz. Ahhoz emberi erőt is kell beletenni.
- Egy módosítási kérésnél meg kell állni, hogy gyorsan összedrótozzuk működőre. Ha már ott az összedrótozás, venni kell a fáradtságot, és szét kell szedni
- a kód mennyisége sokkal nagyobb lesz, de a minőség jobb

Forrás

- http://www.webstar.hu/2015/12/dependency-injection-a-step-to-orthogonal-code/
- https://www.c-sharpcorner.com/UploadFile/4d9083/dependency-injection-using-microsoft-unity-framework/
- https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/
- https://www.c-sharpcorner.com/UploadFile/pranayamr/overview-of-interface-segregation-principle/
- https://www.refaktor.hu/tiszta-kod-5-resz-a-s-o-l-i-d-alapelvek/