Dependency Inversion Principe in C#

Dependency Inversion Principe in C# met voorbeeld

1. **Wat is het Dependency Inversion Principe in C#?**
2. **Voorbeeld zonder Dependency Inversion Principe.**
3. **Voorbeeld met Dependency Inversion Principe.**
4. **Voordelen van implementatie van Dependency Inversion Principe.**

Wat is de Dependency Inversion Principe in C#?

Het **Dependency Inversion Principle** (DIP) zegt dat **hoger-niveau modules/classen niet afhankelijk mogen zijn van lager-niveau modules/classes. Beiden moeten afhankelijk zijn van abstracties.**
Abstractions mogen niet afhankelijk zijn van details. Details moeten afhankelijk zijn van abstracties.
Probeer dus steeds een hoger-niveau module en lager-niveau module zo los mogelijk aan elkaar te koppelen.
Dit kan je doen door ze beiden afhankelijk te maken van abstracties (interfaces /abstracte classes) in plaats van ze rechtstreeks naar elkaar te doen verwijzen.

Wat is "Tight-Coupling" in Software Design?

Tight coupling betekent dat klassen en objects afhankelijk zijn van elkaar
Als een class afhankelijk is van een andere concrete class, dan is er een "Tight coupling" tussen de 2 classes. Dwz, indien we een afhankelijke class wijzigen, moeten we ook de classen wijzigen die afhangen van een object van deze klasse. Deze soort code is moeilijk te testen of te onderhouden.

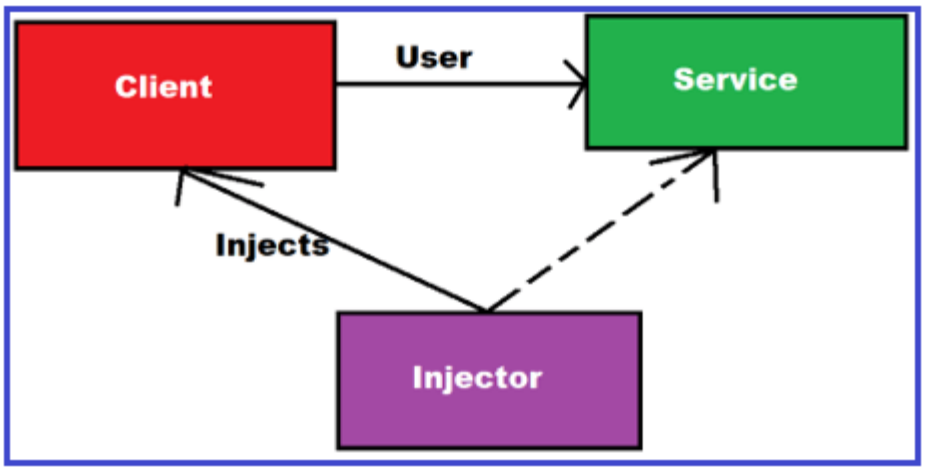
Wat is "Loose Coupling" in Software Design?

Dit betekent dat classes en objects onafhankelijk zijn. Als we een class wijzigen, dan zal dat geen andere class beïnvloeden. Dit wordt "Loose coupling" genoemd. Deze soort code is gemakkelijk te testen en te onderhouden.

Wat is het Dependency Injection Design Pattern in C#?

Het Dependency Injection Design Pattern in C# is de implementatie waarbij we een object of class injecteren in een class die ervan afhankelijk is.Het Dependency Injection design pattern is het meest gebruikte design pattern dat tegenwoordig wordt gebruikt om afhankelijkheden tussen object en classes te verwijderen.
Dependency Injection (DI) is een design pattern die het ***Inversion of Control principe (IoC)*** implementeert. Het laat toe om dependency objecten te creëren buiten een class en levert de objecten aan deze class op verschillende manieren.
Door middle van Dependency Injection (DI), verplaatsen we de creatie en binding van het afhankelijke object buiten de class die ervan afhankelijk is.
Dependency Injection pattroon betrekt 3 types van classen:

1. **Client Class:** de Client class (afhankelijke class) is een class die afhankelijk is van de service class.
2. **Service Class:** de Service class (dependency) is een class die service levert aan de client class.
3. **Injector Class:** de Injector class injecteert het service class object in de client class.



De injector class maakt een object van de service class en injecteert dit object in de client class. Op deze manier zorgt het Dependency Injection patroon dat de verantwoordelijkheid van objectcreatie van de service class buiten de client class gebeurt.

Verschillende Types van Dependency Injection in C#?

De injector class kan het dependency object in een class injecteren op 3 verschillende manieren

Constructor Injection: Wanneer de Injector het dependency object (d.i. service) injecteert via de client class constructor, dit wordt ***Constructor Injection*** genoemd.
Property Injection: Wanneer de Injector het dependency object (d.i. service) injecteert via een public property van de client class, dit wordt ***Property Injection*** genoemd of ***Setter Injection***.
Method Injection: Wanneer de Injector het dependency object (d.i. service) injecteert via een public methode van de client class, dit wordt ***Method Injection*** genoemd.

Constructor injection is de meest gebruikte manier van Dependency injection

Dependency Inversion Principe in C# met een voorbeeld

Maak een console app (.Net Core)

Maak 3 klassen aan **Employee.cs, EmployeeDAL.cs** and **EmployeeBL.cs**

Employee.cs

```
public class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }
    public string Department { get; set; }
    public int Salary { get; set; }
}
```

EmployeeBL.cs

```
public class EmployeeBL
{

    public IEmployeeDAL employeeDAL;
    public EmployeeBL(IEmployeeDAL employeeDAL)
    {
        this.employeeDAL = employeeDAL;
    }
    public List<Employee> GetAllEmployees()
    {
        return employeeDAL.SelectAllEmployees();
    }

}
```

EmployeeBL.cs

```
public interface IEmployeeDAL
{
    List<Employee> SelectAllEmployees();
}
public class EmployeeDAL: IEmployeeDAL
{
    public List<Employee> SelectAllEmployees()
    {
        List<Employee> ListEmployees = new List<Employee>();
        //Get the Employees from the Database
        //for now we are hard coded the employees
        ListEmployees.Add(new Employee() { ID = 1, Name = "Jos", Department = "IT" });
        ListEmployees.Add(new Employee() { ID = 2, Name = "Jan", Department = "HR" });
        ListEmployees.Add(new Employee() { ID = 3, Name = "Piet", Department = "Boekhouding" });
        return ListEmployees;
    }
}
```

Program.cs

```
class Program
{
        static void Main(string[] args)
        {
            EmployeeBL employeeBL = new EmployeeBL(new EmployeeDAL());
            List<Employee> ListEmployee = employeeBL.GetAllEmployees();
            foreach (Employee emp in ListEmployee)
            {
                Console.WriteLine("ID = {0}, Naam = {1}, Afdeling = {2}", emp.ID, emp.Name,
emp.Department);
            }
            Console.ReadKey();
        }
}
```

Referenties

https://www.tutorialsteacher.com/ioc

https://dotnettutorials.net/lesson/introduction-to-inversion-of-control/

https://dotnettutorials.net/lesson/dependency-injection-design-pattern-csharp/

https://www.yogihosting.com/aspnet-core-configurations/#project

https://dotnettutorials.net/lesson/dependency-inversion-principle/

https://www.yogihosting.com/aspnet-core-dependency-injection/#views