C# Fortgeschrittenenschulung

Dependency Injection und Entwurfsmuster

Was ist Dependency Injection?

- **Definition:**
  - Ein Entwurfsmuster, das Objekten ihre Abhängigkeiten bereitstellt, anstatt diese selbst zu erstellen.
- **Ziel:**
  - Entkopplung von Komponenten für erhöhte Flexibilität und Testbarkeit.

Vorteile der Dependency Injection

- **Erhöhte Modulität:**
  - Klassen sind einfacher zu warten und zu verstehen.
- **Einfachere Testbarkeit:**
  - Test-Doubles (Mock-Objekte) können leichter bereitgestellt und verwendet werden.
- **Fördert lose Kopplung:**
  - Vermindert die direkte Abhängigkeit zwischen Klassen.

DI Muster: Constructor Injection

Codebeispiel

```
public interface IService { void Serve(); }

public class Client
{
    private readonly IService _service;

    public Client(IService service)
    {
        _service = service;
    }

    public void StartService()
    {
        _service.Serve();
    }
}
```

- **Beschreibung:**
  - Abhängigkeiten werden über den Konstruktor bereitgestellt.

DI Muster: Property Injection

Codebeispiel

```
public class Client
{
    public IService Service { get; set; }

    public void StartService()
    {
        Service?.Serve();
    }
}
```

- **Beschreibung:**
  - Abhängigkeiten werden durch Setzen einer öffentlichen Eigenschaft bereitgestellt.

DI Muster: Method Injection

Codebeispiel

```
public class Client
{
    public void StartService(IService service)
    {
        service.Serve();
    }
}
```

- **Beschreibung:**
  - Abhängigkeit wird als Parameter einer Methode bereitgestellt.

Vor- und Nachteile von Dependency Injection Ansätzen

Constructor Injection

Vorteile

- **Transparenz:** Alle Abhängigkeiten sind bei der Instanziierung sichtbar.
- **Unveränderlichkeit:** Abhängigkeiten können nach der Erstellung der Instanz nicht verändert werden.
- **Sicherstellung von Abhängigkeiten:** Fehlende Abhängigkeiten führen zu Kompilerfehlern.

Nachteile

- **Vielzahl an Abhängigkeiten:** Kann bei Klassen mit vielen Abhängigkeiten den Konstruktor überladen.
- **Teilweise Injektion:** Nicht geeignet, wenn einige Abhängigkeiten optional sind.

Property Injection

Vorteile

- **Flexibilität:** Abhängigkeiten können nach der Instanziierung gesetzt oder geändert werden.
- **Optionalität:** Macht Abhängigkeiten optional und ermöglicht festgelegte Standardwerte.

Nachteile

- **Sicherheitsrisiko:** Abhängigkeiten könnten vergessen werden gesetzt zu werden.
- **Mutierbarkeit:** Eingeführte Mutierbarkeit kann zu ungewollten Veränderung führen.

Method Injection

Vorteile

- **Kontextspezifität:** Abhängigkeiten werden aufgerufen und genutzt, wo benötigt.
- **Entkopplung:** Methodennutzung ohne feste Bindung während Instanziierung.

Nachteile

- **Komplexität:** Erhöht die Komplexität des Methodenaufrufs.
- **Verwirrung:** Abhängigkeiten sind nicht offensichtlich bei Instanziierung der Klasse.

Verwendung des .NET DI Containers

- **Konfiguration:**
  - `ServiceCollection` zur Registrierung von Diensten.
- **Entwicklung:**
  - Erstellen des ServiceProviders zur JavaScript-Funktion createFactory-Auswahl und Erstellung von Instanzen.

Beispiel

```
var services = new ServiceCollection();
services.AddSingleton<IService, MyServiceImplementation>();
var serviceProvider = services.BuildServiceProvider();

var client = serviceProvider.GetService<Client>();
client.StartService();
```

Service Registration Options

AddSingleton

- **Lebensdauer:** Singleton
- **Beschreibung:**
  - Eine einzige Instanz des Dienstes für die gesamte Lebensdauer der Anwendung.
  - Ideal für zustandsbehaftete Dienste oder ressourcenintensive Objekte, die geteilt werden können.

```
services.AddSingleton<IMyService, MyService>();
```

AddScoped

- **Lebensdauer:** Scoped
- **Beschreibung:**
  - Eine Instanz pro Anforderung.
  - Geeignet für Dienste, die zustandsbehaftete Daten während einer Anfrage benötigen.

```
services.AddScoped<IMyService, MyService>();
```

AddTransient

- **Lebensdauer:** Transient
- **Beschreibung:**
  - Eine neue Instanz wird bei jeder Anforderung erstellt.
  - Geeignet für leichtgewichtige, zustandslose Dienste.

```
services.AddTransient<IMyService, MyService>();
```

Singleton mit Factory

- **Beschreibung:**
  - Verwenden einer Factory-Methode zur Initialisierung.
  - Nützlich für komplexe Initialisierungen.

```
services.AddSingleton<IMyService>(provider => new MyService());
```

Conditional Registration

- **Beschreibung:**
  - Dienste werden abhängig von bestimmten Bedingungen oder der Umgebung registriert.
  - Nützlich in Szenarien, in denen unterschiedliche Implementierungen basierend auf der Laufzeitumgebung benötigt werden.
- **Implementierung:**

```
if (Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT") == "Development")
{
    services.AddSingleton<IMyService, DevelopmentService>();
}
else
{
    services.AddSingleton<IMyService, ProductionService>();
}
```

- **Praktische Anwendung:**
  - Kann verwendet werden, um Debugging-Dienste oder Mock-Implementierungen in Entwicklungsumgebungen bereitzustellen.

Best Practices für Dependency Injection

- **Kleine, fokussierte Klassen:**
  - Nur benötigte Abhängigkeiten bereitstellen.
- **Verwendung von Schnittstellen:**
  - Definieren Sie Verträge zwischen Komponenten.
- **Minimaler Konstruktor:**
  - Vermeiden Sie, dass Klassen direkt große Komplexität erhalten.

Zusammenfassung

- Dependency Injection steigert Flexibilität und Testbarkeit.
- Durch Verwendung von DI Containern wird die Verwaltung komplexer Abhängigkeiten vereinfacht.
- Konstruktor-, Property- und Methodeninjection bieten verschiedene Ansätze zur DI-Implementierung.

Übung: Dependency Injection

Implementierung einer einfachen DI:

Erstellen Sie einen DI-Container für das Erzeugen von Objekten und deren Abhängigkeiten.

Entwurfsmuster: Singleton

Was ist das Singleton-Muster?

- **Definition:**
  - Stellt sicher, dass eine Klasse nur eine Instanz besitzt und bietet einen globalen Zugriffspunkt darauf.
- **Verwendungszweck:**
  - Nützlich für Ressourcen, die geteilt werden müssen, etwa Konfigurationsobjekte oder Verbindungspools.

Umsetzung des Singleton-Musters

```
public sealed class Singleton
{
    private static readonly Singleton instance = new Singleton();

    private Singleton() { }

    public static Singleton Instance
    {
        get { return instance; }
    }
}
```

Eigenschaften

- **Einschränkung:** Der Konstruktor ist privat oder geschützt.
- **Thread-Sicherheit:** Statische Initialisierung stellt Thread-Sicherheit sicher.

Entwurfsmuster: Factory

Was ist das Factory-Muster?

- **Definition:**
  - Bietet eine Methode zur Erzeugung von Objekten, wobei die konkreten Implementierungsklassen verschleiert werden.
- **Verwendungszweck:**
  - Nützlich, wenn die genaue Klasse unbekannt ist oder sich leicht ändern kann.

Umsetzung des Factory-Musters

```
public interface IProduct
{
    void DoSomething();
}

public class ConcreteProductA : IProduct
{
    public void DoSomething() { Console.WriteLine("Product A"); }
}

public class ProductFactory
{
    public IProduct CreateProduct(string type)
    {
        return type switch
        {
            "A" => new ConcreteProductA(),
            _ => throw new ArgumentException("Unknown product type", nameof(type))
        };
    }
}
```

Vorteile und Nachteile

Singleton

- **Vorteile:**
  - Globale Zugriffskontrolle auf die einzige Instanz.
  - Spart Ressourcen, da Wiederverwendung bevorzugt wird.
- **Nachteile:**
  - Kann Testbarkeit erschweren und Abhängigkeiten verstecken.
  - Konflikte bei Multithreading möglich, wenn nicht richtig implementiert.

Vorteile und Nachteile

Factory

- **Vorteile:**
  - Baut Komplexität der Objektinitialisierung aus dem Client-Code aus.
  - Erleichtert den Wechsel der Produktfamilien.
- **Nachteile:**
  - Erhöhung der Abstraktion und Komplexität bei der Handhabung zahlreicher Produkttypen.
  - Kann zu "God Object" führen, wenn zu viel Logik in die Factory ausgelagert wird.

Übungen: Entwurfsmuster

1. **Singleton-Muster:**
    - Erstellen Sie ein Singleton für eine Konfigurationsklasse und prüfen Sie die Instanziierung.
2. **Factory-Muster:**
    - Implementieren Sie eine Factory, die verschiedene Fahrzeugobjekte erstellt und verwendet.

Entwurfsmuster: Observer

Was ist das Observer-Muster?

- **Definition:**
  - Erlaubt einem Objekt, über Änderungen an einem anderen Objekt informiert zu werden.
- **Verwendungszweck:**
  - Geeignet für Systeme, bei denen es wichtig ist, die Zustandsänderungen eines Objekts zu beobachten.

Umsetzung des Observer-Musters

```
public interface IObserver
{
    void Update();
}

public class Subject
{
    private List<IObserver> observers = new List<IObserver>();

    public void Attach(IObserver observer) => observers.Add(observer);
    public void Notify() => observers.ForEach(o => o.Update());
}
```

Eigenschaften

- **Lose Kopplung:** Ermöglicht benachrichtigte Änderungen ohne stark gekoppelte Abhängigkeiten.
- **Flexibilität:** Unterstützt offenes Abonnieren und Abbestellen der Beobachter.

Entwurfsmuster: Strategy

Was ist das Strategy-Muster?

- **Definition:**
  - Definiert eine Familie von Algorithmen, die austauschbar in entsprechenden Kontext eingesetzt werden können.
- **Verwendungszweck:**
  - Ermöglicht die Wahl des Algorithmus zur Laufzeit.

Umsetzung des Strategy-Musters

```
public interface IStrategy
{
    void Execute();
}

public class Context
{
    private IStrategy strategy;
    public Context(IStrategy strategy) { this.strategy = strategy; }
    public void ExecuteStrategy() => strategy.Execute();
}
```

Eigenschaften

- **Flexibilität:** Algorithmen können zur Laufzeit gewechselt werden.
- **Entkopplung:** Trennt die Ausführung von Algorithmen vom Kontext, in dem sie verwendet werden.

Vorteile und Nachteile

Observer

- **Vorteile:**
  - Locker gekoppelte Kommunikation zwischen Absender und Empfänger.
  - Dynamisches Abonnieren/Abbestellen möglich.
- **Nachteile:**
  - Potentiell viele Benachrichtigungen können zu Leistungsengpässen führen.
  - Komplexität bei der Fehlerverfolgung in komplexen Netzwerken.

Vorteile und Nachteile

Strategy

- **Vorteile:**
  - Austauschbare Algorithmen ohne Modifikation des Kontextes.
  - Einfache Erweiterbarkeit durch Hinzufügen neuer Strategien.
- **Nachteile:**
  - Verwaltung vieler Strategien kann komplex werden.
  - Erhöhte Anzahl von Klassen bei vielen Algorithmen.

Zusammenfassung

Schlüsselkonzepte

- **Dependency Injection:**
  - Fördert lose Kopplung und Testbarkeit.
  - Konstruktor-, Property- und Methodeninjektion als Ansätze.
- **Entwurfsmuster:**
  - **Singleton:** Eine einzige Instanz einer Klasse und ein globaler Zugriffspunkt.
  - **Factory:** Erzeugt Objekte, ohne die konkreten Klassen zu spezifizieren.
  - **Observer:** Ermöglicht Abonnenten, auf Ereignisse oder Zustandsänderungen zu reagieren.
  - **Strategy:** Austauschbare Algorithmen, die zur Laufzeit angepasst werden können.

Übungen: Entwurfsmuster

1. **Observer-Muster:**
    - Entwickeln Sie ein einfaches Benachrichtigungssystem für eine Wetterstation.
2. **Strategy-Muster:**
    - Implementieren Sie mehrere Sortieralgorithmen und verwenden Sie das Strategy-Muster, um die Auswahl zur Laufzeit zu treffen.