

The background features a complex network of thin grey lines and dots, forming a web-like structure. Scattered throughout are various triangles of different sizes and orientations, some with solid black dots at their vertices. The overall aesthetic is minimalist and technical.

PATTERN OF THE WEEK

Week 9: Dependency Injection und Design Pattern

SINGLETON PATTERN
Singleton und
Double-Check-Locking

INTERCEPTOR &
CHAIN OF RESPONSIBILITY

DEPENDENCY INJECTION
Edge Cases of
Dependency Injection

01

02

03

TABLE OF CONTENTS

SERVICE LOCATOR

01

SINGLETON PATTERN

Und Double-Check-Locking



SINGLETON PATTERN

- Genau eine Instanz einer Klasse
- Kapselt den Erstellungsprozess (privater Konstruktor)
- "get only" Zugriff auf die Instanz
- Problem: Nicht thread-safe

```
1 public sealed class Singleton1 {  
2     private Singleton1() {}  
3     private static Singleton1 instance = null;  
4     public static Singleton1 Instance {  
5         get {  
6             if (instance == null) {  
7                 instance = new Singleton1();  
8             }  
9             return instance;  
10        }  
11    }  
12 }
```

DOUBLE-CHECK-LOCKING

- 1. Check denn lock() kostet viel Zeit
- lock() um die Ausführung zu sperren
- 2. Check falls zwei Threads im lock() warten

```
if (instance == null)           // First check
{
    lock (syncRoot)
    {
        if (instance == null) // Second check
            instance = new Singleton();
    }
}
return instance;
```

C# IMPLEMENTIERUNG

- Lazy<T> implementiert Singleton
- .Value → Instance

```
public sealed class Singleton
{
    private static readonly Lazy<Singleton> lazy =
        new Lazy<Singleton>(() => new Singleton());

    public static Singleton Instance { get { return lazy.Value; } }

    private Singleton()
    {
    }
}
```

AUFGABE

SINGLETON MIT DCL IMPLEMENTIEREN

Singleton Logger ohne DCL
Konstruktur mit Sleep 1sec
Zwei Threads erzeugen Singleton
"Singletons" prüfen

DCL implementieren



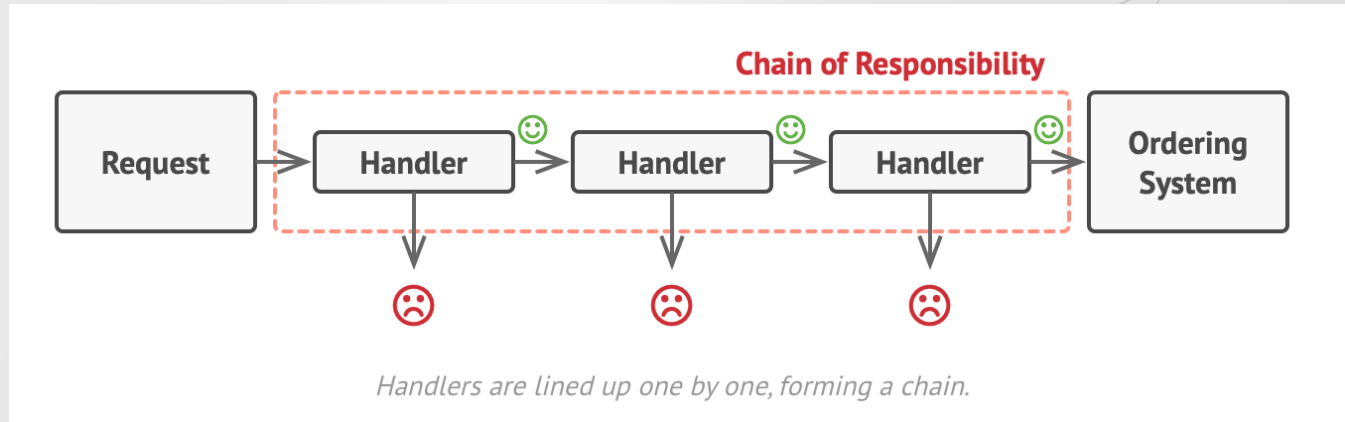


02

INTERCEPTOR & CHAIN OF RESPONSIBILITY

CLASS-TO-CLASS HAND-OVER

- “before” Code (or abort)
- Hand-Over
- “after” Code (or abort)





03

DEPENDENCY INJECTION

Edge Cases of Dependency Injection

DEMO

DEPENDENCY INJECTION EXAMPLES

Mit Castle Windsor



FIRST COME, FIRST SERVE

- Erste Registrierung wird immer als erstes genommen
- Überschreiben
 - a. mit "IsFallback" (um eine vorherige Registrierung zu deaktivieren)
 - b. und "IsDefault" (um eine spätere Registrierung zu bevorzugen)

```
[TestFixture]  
public class LetsTalkAboutDifferentBehaviour
```


LIFESTYLES/SCOPES

- Ein Scope ist ein Dictionary mit bereits erstellten Instanzen
- Standard Scopes sind "immer neu" oder im "container dictionary"
- Singleton: Im "container scope" wird die Instanz gespeichert
- Transient: Es wird eine neue Instanz erzeugt
- Nicht vergessen: Wenn ein Singleton einen Transient als ctor-Parameter bekommt, wird der Transient natürlich nur mit dem Singleton einmalig erzeugt. Ein anderes Singleton bekommt natürlich eine neue Instanz

```
[Test]  
public void SingletonAndTransient()
```

SCOPES

- Ein neuer "Scope" öffnet ein neues Dictionary mit Instanzen
- Lifestyle "Scoped" erwartet ein `container.BeginScope()`
- Gleich Scope, gleiche Instanz
- Neuer Scope, neue Instanz



```
[Test]  
public void AddingScopes_NewScopeNewLogger()
```

SCOPES

- Manche DI Container machen ein "BeginScope()" bei jedem "Resolve"
- Transient ist u.U. automatisch ein "Scoped"
- Scope "AlwaysUnique" ist dann ein "Transient"
- Wichtig: Hier unterscheiden sich DI Container. Es empfiehlt sich ein Unit Test mit dem gewünschten Verhalten zu machen, im Falle von Breaking Changes.

```
[Test]  
public void ResolveScoped()
```

TYPISCHE LIFESTYLES

- PerThread
 - PerHttpRequest
 - PerResolve
 - PerUser
-
- Ein Lifestyle ist also an etwas gebunden, in dem Singletons erwünscht sind und während der "Session" geteilt werden.

```
[TestFixture]  
public class LetsTalkAboutDifferentBehaviour
```


DESIGN PATTERN

- DI Container implementieren Unterstützung für manche Design Pattern
- Singleton (auch z.B. per Thread) [siehe oben]
- Decorator Pattern
- Interceptor Pattern (Chain of Responsibility)
- Factory Pattern
- Lazy Loading
- ...und viele mehr



DECORATOR PATTERN

- First come, first serve
- Implementierung wie gehabt, ctor mit Interface auf "sich selbst"

```
public class AwesomeMailsDecorator : ISendMailService
{
    private readonly ISendMailService _sendMailService;

    public AwesomeMailsDecorator(ISendMailService sendMailService)
    {
        _sendMailService = sendMailService;
    }
}
```

- Die letzte Registrierung ist der innerste Decoratee, also die eigentliche Logik
- Tipp: Innersten Decoratee mit IsFallback() registrieren

```
// first come first serve. The first one will be the outermost decorator
container.Register(Component.For<SendMailServices.ISendMailService>().ImplementedBy<SendMailServices.BlockConfidentialMailsDecorator>());
container.Register(Component.For<SendMailServices.ISendMailService>().ImplementedBy<SendMailServices.SendMailService>().IsFallback());
container.Register(Component.For<SendMailServices.ISendMailService>().ImplementedBy<SendMailServices.AwesomeMailsDecorator>());
```

INTERCEPTOR

- Implementiert "Chain of Responsibility"
- Wichtig: Weiterleiten ans nächste Kettenglied nicht vergessen
- Interceptor müssen auch am Container registriert werden (DI in den Interceptor)
- Im Gegensatz zum Decorator sind Interceptor absolut generisch und die Logik entsprechend abstrakter

```
container.Register(Component  
    .For<SendMailServices.ISendMailService>()  
    .ImplementedBy<SendMailServices.SendMailService>()  
    .Interceptors<LogInterceptor>()  
    .Interceptors<UppercaseParameterInterceptor>());
```

LAZY LOADING

- Automatisches Injection von Lazy<T>, statt T
- Kann für das lösen zirkuläre Abhängigkeiten verwendet werden
- Wichtig: Den LazyOfComponentLoader nicht vergessen

```
// this is required to allow lazy to be loaded automatically  
container.Register(Component.For<ILazyComponentLoader>().ImplementedBy<LazyOfTComponentLoader>());
```