

### Hinweis:

Diese Druckversion der Lerneinheit stellt aufgrund der Beschaffenheit des Mediums eine im Funktionsumfang stark eingeschränkte Variante des Lernmaterials dar. Um alle Funktionen, insbesondere Animationen und Interaktionen, nutzen zu können, benötigen Sie die On- oder Offlineversion. Die Inhalte sind urheberrechtlich geschützt.  
©2018 Beuth Hochschule für Technik Berlin

## DEP - Dependency Injection



## Lernziele und Überblick



### Lernziele

Ziel dieser Lerneinheit ist es, mit der Struktur und dem Sinn des Entwurfsmusters Dependency-Injection (DI) vertraut zu werden. Vorteile und Nachteile dieses Ansatzes sollen verstanden und aktiv vertreten werden können. Gleichzeitig soll sich der Student mit einem DI-Framework vertraut machen, diese testen und Mock-Objekte einbinden können.

Idealerweise wählt der Student - nach gründlicher Abwägung - ein Framework aus und setzt dies in seinem globalen Softwaretechnik-Projekt ein.

Schließlich soll die Bedeutung von Dependency-Injection in großen Softwaresystemen klar werden.



### Gliederung der Lerneinheit

Nach einer Einführung in die Begriffe und die Entstehung des Begriffs Dependency Injection werden die Problemstellung und ein erster Lösungsansatz behandelt.

Varianten werden in Kapitel 3 diskutiert und der Zusammenbau von Objekt-Bäumen als wichtiges Argument betrachtet.

Neben den Vor- und Nachteilen des Dependency Injection wird eine Abgrenzung am Beispiel des Service Locator Ansatzes dargestellt sowie das Lifecycle-Management behandelt.

Eine Dependency Injection Demo stellt das von Google entwickelte DI-Framework Google Guice an einem einfachen Beispiel vor und ein kurzer Blick auf die Beispiele mit „Spring“ und dem „PiccoContainer“ schließen das Thema ab.



### Zeitbedarf und Umfang

- Das theoretische und praktische Durcharbeiten der Lerneinheit dauert ca. 90 Minuten.
- Das Einbinden eines Frameworks in das eigene Projekt dauert (für 2-3 Abhängigkeiten) etwa 30 Minuten



Film

Webkonferenz zur Lerneinheit DEP

© Beuth Hochschule Berlin - Dauer: 03:34 Min. - Streaming Media 6.9 MB

Die Hinweise auf klausurrelevante Teile beziehen sich möglicherweise nicht auf Ihren Kurs.

Stimmen Sie sich darüber bitte mit ihrer Kursbetreuung ab.

## 1 Einführung

Im Englischen bezeichnet man eine Dependency als Abhängigkeit einer Komponente A vom dem Service einer Komponente B. Die Komponente, die die Abhängigkeit benötigt wird als Dependent bezeichnet. Der Begriff „injection“ wird häufig mit Injektion, Einspritzung oder Impfung übersetzt. Beides zusammen ergibt **Dependency Injection** und wird im Folgenden auch mit **DI** abgekürzt.

Bevor der Begriff Dependency Injection geprägt wurde sind ähnliche Ansätze unter dem Stichwort „Inversion of Control“ (IoC) erwähnt worden. Da man unter IoC jedoch das konkrete Hollywoodprinzip versteht, ist DI durchaus etwas Anderes. Unter dem Hollywoodprinzip („don't call us, we'll call you“) versteht man ein Framework welches die Programmkontrolle hat und dann registrierte Komponenten aufruft. Es geht hier also lediglich um den Kontakt - den Aufruf von Komponenten und nicht um deren Zusammenbau (engl. assembly).

 [http://de.wikipedia.org/wiki/Inversion\\_of\\_Control](http://de.wikipedia.org/wiki/Inversion_of_Control)

Aufgrund dieses Missverständnisses hat **MARTIN FOWLER** den Begriff Dependency Injection eingeführt, um einen anderen Entwurfsansatz - ein Entwurfsmuster - zu benennen. Der Begriff entstand u.a. aus einer Diskussion mit den Entwicklern von PicoContainer und **ROD JOHNSON** (Spring).



Definition

### Dependency Injection

Dependency Injection bezeichnet ein Entwurfsmuster, bei dem versucht wird, die Abhängigkeiten der Komponenten zu minimieren und diese mithilfe eines Frameworks oder sonstigen Mechanismus aufzulösen. Benötigt eine Komponente A also eine Komponente B, so soll A lediglich das Interface zu B kennen, und die konkrete Implementierung von einem Framework bekommen.

Der Begriff Dependency Injection lässt sich daraus ableiten, dass Komponenten ihre Abhängigkeiten von außen quasi „injiziert“ bekommen.

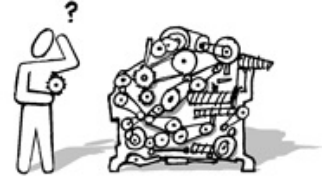
Dies hat viele Vorteile, die in den folgenden Kapiteln diskutiert werden sollen.

## 2 Problemstellung und Lösungsansatz

In der Lerneinheit ARC - Architektur wurde beschrieben wie wichtig die Punkte Wartbarkeit, Änderbarkeit sowie Testbarkeit für das Software-Engineering und insbesondere bei größeren Softwaresystemen sind. Die Idee der Dependency Injection ist auch aus der Beobachtung entstanden, dass große Softwaresysteme oftmals nicht mehr beherrscht werden können.

Zur Erinnerung hier einige Punkte / Probleme, die bei großen Systemen bekannt sind und doch immer wieder vorkommen:

- Das Softwaresystem wird zu groß. Es blickt keiner mehr durch.
- Es liegen Tausende von Klassen und Codefiles vor.
- Es entsteht der Eindruck, es handle sich um „Spaghetticode“ (Siehe: [www The Big Ball of Mud](#)).
- Die Wartung des Systems wird sehr aufwändig.



Den Entwicklern ist das Problem bereits bewusst, dass an vielen Stellen Änderungen vorgenommen werden müssten. Wiederum hört man auch von den Entwicklern, dass eigentlich ein komplettes „Rewrite“ des Systems nötig wäre - jedoch fehlt dafür wie so häufig, die Zeit. Jede Änderung im System bewirkt, dass an anderer Stelle viele neue Fehler auftreten. Die Entwickler sind somit ständig am Debuggen.

Was kann denn nun getan werden, um die Änderbarkeit eines Systems zu verbessern?

Wie schon kurz angerissen, verfolgt die Idee des Dependency Injection den Ansatz, Wissen der Komponentenabhängigkeiten zu externalisieren. Dies bedeutet, dass die Komponente davon befreit werden sollte konkret zu wissen, mit welchen anderen Komponenten sie kommuniziert.

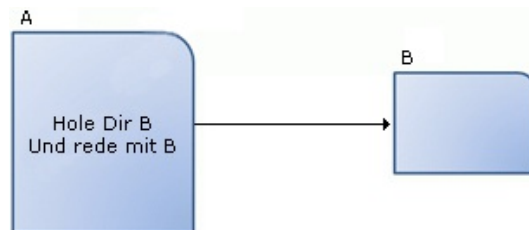


Abb.: Kommunikation der Komponenten

Es könnte Komponente A einfach mit Komponente B kommunizieren, indem diese instanziiert wird. Wir hätten dann:

```

class A {
    B myB = new B();
    myB.service();
}
  
```

### Nachteile

Das Problem bei dieser Variante ist, dass Komponente B fest verdrahtet ist. Dies hat folgende Nachteile:

1. Komponente B wird sich ändern. Warum sollte - wenn z. B. der Name von Komponente B geändert wird - sich auch Komponente A ändern? Es wäre ideal, wenn das nicht nötig wäre. Aber wenn sich der Name von B ändert, muss A auch geändert werden. Hier ist auch ein Refactoring in der IDE nicht immer die ideale Lösung, da Code betroffen sein kann, der nicht der Eigene ist.
2. Der Entwickler möchte die Komponente B austauschen und zu Testzwecken durch eine Attrappe ersetzen. Dies geht nicht ohne Code zu ändern. Der Code sollte aber nicht geändert werden.

Die Lösung liegt darin, eine weitere Indirektionsstufe im Wissen einzuführen, in der bestimmt wird, wer sich mit wem unterhalten darf.

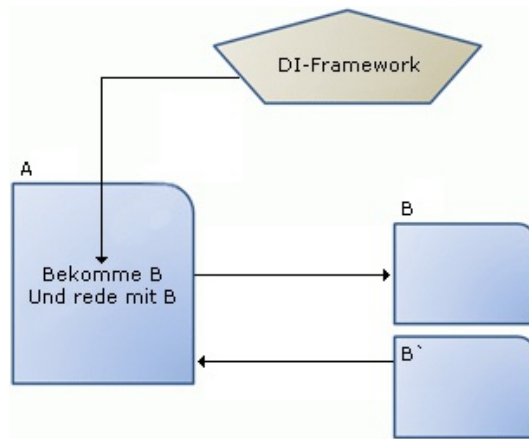


Abb.: Kommunikation mit DI

Wie die vorangegangene Abbildung zeigt, wäre es ideal, wenn Komponente A nur noch das Interface von B kennt. Die konkrete Implementierung - also welche der möglichen konkreten Klassen jetzt genau genommen werden soll - wird A von einem DI-Framework mitgeteilt. Die richtige und aktuelle Komponente wird instanziiert und übergeben.

Wie man sieht kann dies nicht nur das Interface konforme B, sondern auch B' oder B'' sein. Es können also Änderungen an der Klasse B vorgenommen werden - z. B. den Namen ändern - und A merkt nichts davon. Weil A die Komponente B entweder

1. über das **Interface** oder
2. über einen **selbstgewählten Namen** identifiziert.

In der Praxis gibt es Frameworks, die einen oder beide Mechanismen anbieten. In jedem Falle wird die Abhängigkeit von außen gesetzt, also „injected“ (engl. injiziert).

### 3 Dependency Injection Varianten

Die Vorteile des im vorigen Abschnitt genannten Mechanismus sind gewichtig:

- Wartbarkeit  
Änderungen am Code haben nicht mehr unmittelbar Auswirkungen auf andere Klassen. Die Indirektion federt quasi Änderungen ab. Entwickler werden daher eher motiviert Änderungen oder Refactorings durchzuführen.
- Änderbarkeit:  
Implementierungen von Interfaces können extrem einfach ausgetauscht werden.
- Testbarkeit:  
Um Komponenten besser testen zu können müssen oft lästige Abhängigkeiten ausgetauscht werden. Statt dieser Abhängigkeiten setzt man die bekannten Mock Objekte ein. Dies geht mit Dependency Injection natürlich viel einfacher.



Das DI Prinzip oder Entwurfsmuster ist also in der eigentlich eine gute Idee. Doch wie kann man es umsetzen?

#### 3.1 Möglichkeiten

##### Factories

Um eine Indirektion in Bezug auf Komponentenabhängigkeiten einzuführen, könnte man auf die Idee kommen, eine Factory zu nutzen. Dies kann durchaus eine gute Idee sein und es gibt Anwendungsfälle wo dies vorteilhaft ist. Wie würde das aussehen?



Beispiel

Dazu ein Beispiel:

Wir benötigen **(A)** ein Interface, **(B)** eine Implementierung und **(C)** ein Mock Objekt:

##### Interface

```
public Interface Service {
    void do();
}
```

##### Implementierung

```
public class ServiceImpl implements Service {
    public void do() {
        // Aufwendige Dinge...
    }
}
```

##### Mock Objekt

```
public class MockService implements Service {
    public void do() {
        // Einfache Realisierung. Z. B. einfache Rückgabe des Ergebnisses.
    }
}
```

##### Factory

Und nun einen Client der eine Factory verwendet:

```
public class Client {
    public void go() {
        Service service = ServiceFactory.getInstance();
        service.go();
    }
}
```

Die Factory könnte - ähnlich eines Singletons - so aussehen:

```
public class ServiceFactory {
    private ServiceFactory(){};
    private static Service instance = new ServiceImpl();

    private static Service getInstance(){
        return instance;
    }
    private static Service setInstance(){
        instance = service;
    }
}
```

Wie würde man das jetzt im Test nutzen?

```
public void TestClient() {
    MockService mock = ServiceFactory.getInstance();
    ServiceFactory.setInstance(mock);
    Client client = new Client();
    client.go();
    // assert etwas
    // Service zurücksetzen
}
```

#### Nachteile

Was sind die Nachteile dieses Ansatzes?

- Man muss den Factory Code immer wieder selbst und sogar für jeden Service neu schreiben, was nicht unbedingt das Ziel ist.
- Soll der Service als Singleton angeboten werden oder nicht? Diese Entscheidung soll vereinfacht werden. Entscheidet man sich gegen Singleton wäre der Nachteil eine umständliche Implementierung oder Konfiguration.
- Der Factory Code ist unter Umständen nicht Threadsafe.

Einer der wichtigsten Nachteile (der „nested Hierarchien“) wird auf der nächsten Seite beschrieben.

#### Selbst Abhängigkeiten setzen

Man kann natürlich die Abhängigkeit einfach selbst im Konstruktor übergeben:

```
public class Client()
    private final Service service;

    public Client(Service service) {
        this.service = service,
    }
    public void work() {
        service.do();
    }
}
```

Der Mock Test würde nun einfacher aussehen:

```
public void testClient() {
    MockService mock = new MockService();
    Client client = new Client(mock);
    client.do();
    //assert something + cleanup
}
```


Falls man vergisst eine Instanz zu übergeben, könnte man beim ersten Zugriff die Variable überprüfen, ob diese null ist und dann eine Default-Implementierung übergeben.

In Ruby könnte man dies sehr einfach notieren:

```
service = Service.new if service.nil?

oder noch einfacher

service ||= service.new
```

Dazu ein komplettes  [Beispiel als Code \(Siehe Anhang\)](#).

Der Entwickler könnte an dieser Stelle denken, dass alles gelöst ist. Beim Erstellen einer neuen Komponente mit **new** braucht nichts mehr getan werden.

Es wird immer die Default-Abhängigkeit genommen. Werden andere Abhängigkeiten benötigt, müssen diese von Hand geschrieben und per Konstruktor übergeben werden.

#### Diskussion

Warum sind die Varianten 1 und 2 dennoch nicht optimal?

Eines der wichtigsten Argumente ist der Zusammenbau von Objekt-Bäumen!

Man stelle sich vor, dass der Service selbst fünf Abhängigkeiten besitzt, die ebenfalls wiederum fünf weitere Abhängigkeiten haben, die ganze zehn Stufen tief gehen. Dann möchte sich der Entwickler wohl kaum darum kümmern, diesen ganzen Baum selbst zusammenzubauen. Diese Aufgabe kann ein Framework übernehmen, das sich mit dem Zusammenbau von Objekthierarchien auskennt und dies rekursiv tun kann.

Der Anwender möchte also definieren: **IA** wird jetzt gerade von **ImyA** implementiert - und dies als **singleton** oder nicht als **singleton**! Dann soll der Rest automatisch geschehen. So wird **DI** von modernen Frameworks realisiert und macht die Arbeit mit Abhängigkeiten sehr einfach. Wie bereits erwähnt, wird dadurch zusätzlich das Ändern und Testen stark motiviert.



Hinweis

Gute DI Frameworks kennen noch mehr Scopes als nur **singleton** oder **No-Scope** (also immer ein frisches neues Objekt). Besonders für Web-Anwendungen gibt es viele weitere Scopes wie (**http**) **Request**, **Session**, **Flash**, **Conversation**, **Transaction** Scopes. Sogar eigene Scopes können dann mit Hilfe von Interfaces selbst definiert und an einen Kontext gebunden werden.

### 3.2 Dependency Injection Typen

Man unterscheidet mehrere Varianten, wie die Auflösung von Abhängigkeiten realisiert werden kann:

Typ	Beschreibung
Typ 0	Hier findet ein direkter Aufruf statt. In den Komponenten sind direkte Abhängigkeiten vorhanden (z. B. mit new).
Typ 1	Hier wird die Abhängigkeit über einen Service Manager ermittelt (JNDI ist so ein Dienst; in JavaEE ist dies der Fall)
Typ 2	Setter Injection mit einem Framework
Typ 3	Konstruktor Injection mit einem Framework

Tab.: Varianten von Abhängigkeiten

Die meisten fortschrittlichen Frameworks bieten Typ 2 und 3 DI an sowie weitergehende Konzepte, bei denen kein Setter oder Konstruktor benötigt wird.





**Nachteile der Konstruktor Injection:**

- Kann nur einmal aufgerufen werden.
- Kann nicht anders benannt werden.
- Bei vielen Feldern wird der Konstruktor sehr lang. Hier müssen evtl. viele Objekte in “**method objects**” transformiert werden. Dies passt dann aber u. U. nicht mehr mit den Fähigkeiten des Frameworks zusammen.

Letztere Variante bezeichnet man als Interface Injection. Hier wird eine Methode per Interface definiert, die vom Framework aufgerufen werden kann und die alle Abhängigkeiten übergibt.

Konstruktor Injection aber dennoch von einigen Entwicklern bevorzugt, da es eine sauber notierte Variante darstellt und in der Regel auch nicht mehr als eine Hand voll Abhängigkeiten vorliegen sollten. Ansonsten würde etwas mit dem Design der Klasse nicht stimmen.

**Diskussion der Setter Injection**

Insbesondere aufgrund der bereits genannten Nachteile wird Setter Injection häufiger angewendet. Hauptsächlich dann wenn weder die Setter selbst geschrieben werden müssen noch die Abhängigkeiten einfach per Annotation übergeben werden können. In diesem Fall spricht man von Field Injection. Google Guice nutzt dieses Verfahren und setzt die Felder der Abhängigkeiten direkt!


In existierenden Frameworks gibt es also mehrere Vorgehensweisen, wie DI realisiert werden kann:

- **(A)** Die Abhängigkeit wird als Feld definiert und von außen übergeben (Konstruktor oder Setter)
- **(B)** Die Abhängigkeiten werden irgendwo gesetzt (z. B. in XML) und dann aus einer speziellen Factory-Methode eines Frameworks geholt.

Erstere Variante **(A)** liegt z. B. bei Google Guice vor und ist auch per Annotation möglich. Das heißt, ich kann die Interfaces so annotieren, dass ich schreibe = annotiere, welche Implementierung denn gerade dieses Interface implementiert. Dann kann die konkrete Klasse vom Framework erstellt und geliefert werden.

In Spring werden die Abhängigkeiten meist in XML gesetzt und dann aus einer Factory geholt. Dies entspricht dem zweiten Fall **(B)**.

Weitere bekannte Vertreter sind:

- Apache Avalon.  
Eins der ersten DI Frameworks, das mittlerweile in viele Subframeworks aufgesplittet worden ist.
- Pico- und Nanocontainer  
 <http://picocontainer.org/>.

Picocontainer wird später noch kurz erwähnt und natürlich gibt es für fast alle Sprachen leistungsfähige DI-Frameworks.

Gute Frameworks können zyklische Referenzen erkennen. Zyklische Referenzen bauen ein „Henne-Ei“-Problem auf, bei dem eine Komponente nicht ohne die andere aufgebaut werden kann. Es wird dann versucht, diese mittels verschiedenen Mechanismen (Wechsel des Injection Mechanismus oder Proxies) aufzulösen.



Wichtig



Hinweis

## 4 Vorteile des DI Ansatzes

An dieser Stelle seien nochmal die Vorteile des Dependency Injection Ansatzes aufgezählt:

- Einfacheres Testen durch einfaches "injecten" von Mocks.
- Trennung / Decoupling von Code durch Einführen der weiteren Indirektionsstufe, wodurch eine geringere Kohäsion möglich wird.
- Es können Einsparungen bei sich wiederholendem Code erzielt werden, wenn bei der Umsetzung auf Factory oder Hand "injections" verzichtet wird
- Die Wartbarkeit des Codes erhöht sich. Änderungen am Code haben nicht mehr zur Folge, dass der Code bricht ("broken Windows").
- Man kann extrem einfach entscheiden, ob die Klasse / Abhängigkeit als Singleton genutzt werden soll oder nicht. Dies spart viel Singleton Code im Projekt.

## 5 Nachteile des DI-Ansatzes

Dependency Injection kann unter Umständen folgende Nachteile beinhalten:

1. In kleinen Projekten kann Dependency Injection einen unnötigen Overhead bedeuten. Ein zusätzliches Framework mit einzubeziehen ist mit (wenn auch geringerem) Aufwand verbunden.
2. Die Konfiguration eines DI-Frameworks kann aufwendig sein. Es können z. B. viele XML-Dateien zu pflegen sein, was auch eine erhöhte Fehleranfälligkeit bedeutet.
3. DI macht den Code eventuell komplizierter. Dies gilt allerdings für fast jedes Entwurfsmuster.

```
ClassB myb = new ClassB();
```

Die obere Zeile ist relativ einfach.

```
Interface B myb = ... magicDIFramework();
```

Die zweite Zeile ist komplexer. In einigen Fällen kann auch mit Annotations gearbeitet werden, was wiederum deutlich einfacher ist. Nur muss dann das Konzept der Definition von Feldern für die Abhängigkeiten in Klassen erst einmal verstanden und konstant richtig angewendet werden.

## 6 Service Locator / Service Discovery

Der Dependency Injection Ansatz unterscheidet sich stark vom Ansatz des einfachen Auffindens von Diensten / Services. Wir stellen an dieser Stelle einmal den Service Locator Ansatz vor, um dann am Ende auf die bedeutenden Unterschiede zum echten Dependency Injection hinzuweisen.

So bieten sowohl Java und .NET entsprechende Dienste an, um Komponenten und deren Services zu finden. In vielen Fällen geht es hier aber nicht um komplette Dependency Injection (Zusammenbauen von Objektbäumen) sondern um das einfache Holen der Komponenten. Ist die Komponente erst einmal da, wird ein Service (i. d. R. eine Methode) verwendet bzw. aufgerufen.

Unter Java gibt es dies z. B. für J2EE Dienste:

[www.oracle.com](http://www.oracle.com)

Die APIs sehen hier ähnlich aus:

```
class ServiceLocator {
    private ServiceLocator()
    public static ServiceLocator getInstance()
    public EJBObject getService(String id)
    protected String getId(EJBObject session)
    public EJBHome getHome(String name, Class clazz)
}
```

Es gibt Methoden, um Services zu registrieren und zu holen. Auch unter C# gibt es sowohl Frameworks, als natürlich auch die Möglichkeit dieses selbst zu implementieren und - z. B. über eine Collection - selbst zu verwalten.

```
public class ServiceLocator {
    ...
    public void AddService(String ServiceName, IService Service){...}
    public IService GetService(String ServiceName){...}
}
```

Der Java JNDI-Service ist eines der bekanntesten Beispiele für einen Service Locator. Dienste können dort in einem baumartigen Namensraum abgelegt werden. Häufig werden Ressourcen oder Data Sources dort initial gespeichert und zur Laufzeit der Programme wieder dynamisch geladen.

Ein weiteres Beispiel ist in der Jakarta Commons Bibliothek zu finden. Hier kann die Implementierung eines Interfaces einfach über eine Referenz in einer Datei gefunden werden. Betrachten Sie die folgende Architektur:

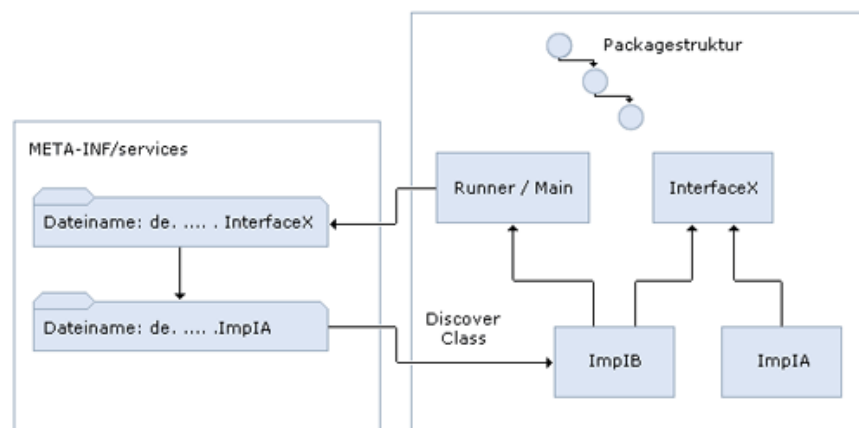


Abb.: Implementierung eines Interfaces

Wir sehen:

1. Es gibt ein InterfaceX
2. Es kann dazu verschiedene Implementierungen geben: ImplA und implB
3. In der Main-Anwendung wird mit Hilfe der Apache Commons Discovery Bibliothek einfach im Verzeichnis META-INF/services gesucht.
4. Dort findet das Framework die Datei mit dem Namen des Interface.
5. Der Inhalt der Datei ist dann der Name der Klasse die das Interface implementiert und die gerade gewählt werden soll.

Die Implementierung kann also von außerhalb zur Laufzeit gewechselt werden. Das Ganze noch einmal in einem einfachen Codebeispiel:

```
package de.vhf.swt.di;
public interface EinInterface {
    public int wichtigeMethode(String arg); // strlen
}
```

Das obige Interface gilt es zu implementieren. Dort ist eine wichtige Methode enthalten, die von verschiedenen Klassen durchaus verschieden implementiert werden kann. Oftmals gilt es, diese Services / Methode überhaupt zu finden und dann dynamisch austauschen zu können. Genau dies soll im Folgenden demonstriert werden.

Wir benötigen zwei Implementierungen des Interfaces:

```
package de.vhf.swt.di;
public class ImplementierungA implements EinInterface {
    public int wichtigeMethode(String arg) {
        return arg.length; //line before deleted!
    }
}
```

Und die Zweite:

```
package de.vhf.swt.di;
public class ImplementierungB implements EinInterface {
    public int wichtigeMethode(String arg) {
        char[] arr = arg.toCharArray();
        return arg.length; } }
}
```

Beide Methoden liefern die Länge des übergebenden Strings zurück. Nur auf unterschiedliche Art und Weise. Dennoch soll uns der Inhalt der Methode nicht interessieren. Wir wollen diesen Service finden und austauschen:

```
package de.vhf.swt.di;
import org.apache.commons.discovery.resource.ClassLoaders;
import org.apache.commons.discovery.tools.DiscoverClass;

public class DiscoverEtwas {
    private void findeEtwas() {
        ClassLoaders loaders = ClassLoaders.getAppLoaders(
            de.vhf.swt.di.EinInterface.class,
            getClass(), false);
        DiscoverClass discover = new DiscoverClass(loaders);
        Class implClass = discover.find(de.vhf.swt.di.EinInterface.class);
        System.out.println("Verwendete Klasse " + implClass.getName());
        try { // Jetzt als echte Instanz
            EinInterface inst = (EinInterface) implClass.newInstance();
            int result = inst.wichtigeMethode("vier"); // Result ausgeben = 4 !
        }
        catch (Exception e) {
            e.printStackTrace();// Real swt'lers do it better here...
        }
    }
}
```

Was hier passiert: Zu Beginn wird eine „**discover**“-Klasse benötigt, die die gesuchte Klasse mit dem „**finder**“ findet. Danach müssen diese noch in die echte Klasse umgewandelt werden sodass die „**wichtigeMethode**“ der gewünschten Klasse aufgerufen werden kann. Dazu hat der **finder** im Dateibaum das Interface gesucht und die Klasse gefunden.

Was lernen wir daraus und wie unterscheiden sich Service Locator nun von Dependency Injection?

1. Service Locator sind ideal wenn ich einfach nur einen Dienst **finden** will und nicht noch mehr Objekt-Magic haben möchte.
2. Ein Service Locator macht kein Dependency Injection. Ein vollständiger **Objektbaum** wird in der Regel **nicht** aufgebaut!
3. Ein Service Locator findet nur. Er macht **kein Lifecycle-Management** und stellt normalerweise **kein Singleton** zur Verfügung.

Machen Sie sich die Unterschiede bewusst und entscheiden Sie aufgrund dieser Unterscheidungsmerkmale, welches Werkzeug ihrer Meinung nach sinnvoller ist.

Dependency Injection Werkzeuge sind häufig deutlich mächtiger, als ein Service Locator, was sowohl ein Vor- als auch ein Nachteil sein kann. In einigen Umgebungen reicht ein Servicedienst völlig aus. Viele Softwareframeworks und Anwendungen haben eine einfache Registry bereits integriert.

Dennoch sind DI-Frameworks wie Google-Guice performant und Dank optionaler Annotations, extrem einfach, wodurch diese eine Vielzahl von weiteren Möglichkeiten des Objekthandlings bereitstellen.

## 7 Lifecycle-Management

Der Lebenszyklus eines Objektes hängt eng mit dem Scope eines Objektes zusammen. Das muss er aber nicht zwingend der Fall sein. Der Lebenszyklus eines Objektes kann auch in verschiedenen Formen, unabhängig vom Scope sein.

Betrachten wir welche Ereignisse es im Leben eines Objektes geben kann:



- **Object Construction**  
Mit dem Aufruf des Konstruktors ist ein Startpunkt definiert.
- **Object Destruction**  
Entweder implizit wie in Java oder explizit (finalizer in Java) wie in vielen anderen Sprachen (C++: `class X {~X()};`)
- **Object Initialization**  
Das Setzen der Abhängigkeiten. Hier wird auch oft von einem **start**-Zustand gesprochen.
- In vielen Frameworks gibt es Objekte mit bestimmten Bedeutungen. Dann können Objekte auch **no idle** / **paused** (nicht arbeitend), **detached** (nicht angebunden) oder in anderen Zuständen sein.

Die Idee des Lifecycle-Managements ist es, Objekte bei Events zu benachrichtigen. Diese Eigenschaft ist eine wichtige Hilfe in vielen Anwendungsfällen.



Beispiel

```
public class Car implements Startable { // (A) oder
public class Car { // (B) Variante mit Reflection

    public void start(){
        // Ressourcen aufbauen, Netzwerkverbindungen öffnen, etc.
    }
    public void stop(){
        // Ressourcen abbauen, Netzverbindungen schliessen, etc.
    }
}
```

Es gibt viele Gründe, warum ein zusätzlicher Mechanismus im Rahmen eines DI-Frameworks sinnvoll ist. Beispielsweise gibt es Aufgaben, deren Erledigung in einem Konstruktor oder einem Destruktor nicht so einfach möglich sind.

- **Scopes**  
Aktivierung von **start()** und **stop()** bei verschiedenen Kontextevents (z. B. http-Session).
- **Lazy loading**  
Bei einigen Frameworks gibt es die Möglichkeit, Objekte lazy also „faul“ zu laden. Das bedeutet, dass unter Umständen gar nicht alle abhängigen Objekte zur Instanziierungszeit vorhanden sein müssen. Es gibt Szenarien wo eine schnelle Startup-Time benötigt wird und daher dieses Feature wichtig ist. Im lazy loading-Fall kann eine Benachrichtigung der Objekte ebenfalls wichtig sein.
- **Post processing**  
Für die Validierung von instanziierten Objekten kann eine Methode sinnvoll sein die nach der Initialisierung aufgerufen wird. Spring bietet dafür beispielsweise den Bean PostProcessor an.
- **Rebinding**  
Was passiert wenn Abhängigkeiten geändert werden? Diese Fragestellung ist nicht trivial.



Beispiel

Dazu noch ein Beispiel:

```
class Car {  
    public Car(Engine e, Pump p){  
        this.engine = e;  
        this.pump = p;  
        e.setRotationSpeed(...); // Initialisierung in Initialisierung  
        p.calculatePressure(...); // Initialisierung in Initialisierung  
    }  
}
```

Es gibt offenbar auch Initialisierungen die nicht idealerweise in den Konstruktor der Engine passen, sondern erst später aufgerufen werden müssen. Allerdings unbedingt bei der Initialisierung des **Autos**. Äquivalent kann man in Java beim Schließen des Objektes nicht garantieren, wann ein **finalizer** aufgerufen wird. Sicher nur vor dem neuen Erstellen eines Objektes. Es kann aber auch viel zu früh oder zu spät passieren. Im ungünstigen Fall wenn die Anwendung geschlossen wird. Wenn aber ein Objekt nur im **finalizer** z. B. Kommunikationsverbindungen schließt, kann dies zu Fehlern führen.

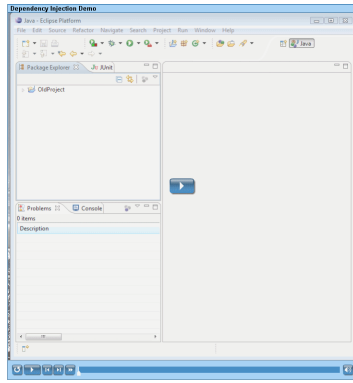
Gelingt es mit einem DI-Framework den Lebenszyklus in den Griff zu bekommen, können hier bessere Methoden aufgerufen werden.

Auch in anderen Frameworks / Architekturen wie OSGI - die auch ein Lifecycle-Management beinhalten (Siehe Lerneinheit ARC - Architektur) - gibt es das Problem der Änderung von Services. Die Frage ist, was dann mit bereits instanziierten und vom DI-Framework zusammen gebauten Objekten geschieht? Eine Möglichkeit wäre, dass einfach nichts passiert.

Besser ist es jedoch, wenn die Objekte benachrichtigt - und entsprechende „sichere“ Aktionen eingeleitet werden können.

## 8 Google Guice

Das von Google entwickelte DI-Framework hat den Jolt Award gewonnen und zeichnet sich durch eine hohe Performance aus. Es ist ideal für Entwickler, die kein XML konfigurieren möchten. Daher sei es an dieser Stelle etwas ausführlicher dargestellt.



Animation: Dependency Injection Demo (Siehe Anhang)



Beispiel

### Beispiel zu Dependency Injection mit Google Guice

Das folgende Beispiel zeigt Dependency Injection mit Google Guice an einem einfachen Beispiel:

- Ein Taschenrechner - im Folgenden Calculator genannt - möchte den Mittelwert zweier Zahlen berechnen
- Dazu benötigt er die Funktionalität „addieren“ aus der Komponente **Calc**
- Die Komponente **Calc** ist im Interface **ICalc** definiert
- Alternativ kann für Tests auch eine Mock Methode (**CalcMock**) benutzt werden, falls **calc** in der Realität nicht in den Test mit einbezogen werden könnte.
- Es gibt weiterhin eine TestMethode, die den Taschenrechner testen soll: **CalculatorTest**
- Es ist auch möglich - falls gewünscht - , dass die Konfiguration aus den Klassen in eine extra Datei ausgelagert wird: **CalculatorModule**.

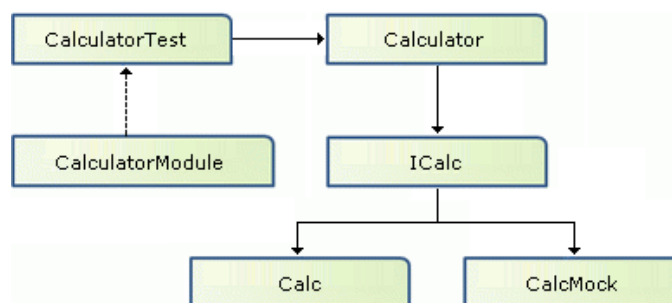


Abb.: DI mit Google Guice

Codebeispiele

Als erstes definieren wir das **Interface** für die Berechnung der Summe:

```

package de.vfh.swt.di;

import com.google.inject.ImplementedBy;
@ImplementedBy(Calc.class)
public interface ICalc {
    public int add(int a, int b);
}
  
```



Hier zeigt sich, dass sofort angegeben werden kann, wer dieses Interface definiert. Der Entwickler kann dies also als Annotation gleich bei dem Interface angeben oder bei Bedarf auch in ein Modul auslagern, so dass es nicht hier steht.

Wie sieht nun die **Rechen-Klasse** aus die dieses Interface implementiert?

```
public class Calc implements ICalc {
    public int add(int a, int b) {
        return a+b;
    }
}
```

Hier passiert nichts Überraschendes.

Alternativ kann statt der obigen Klasse eine andere ausgewählt werden, die das Interface implementiert. Dies könnte ein Mock sein, der einfacher aufgebaut ist:

```
package de.vfh.swt.di;
public class CalcMock implements ICalc {
    public int add(int a, int b){
        return 8;
    }
}
```

Simulation komplexer Vorgänge

In der Realität würde ein komplexer Vorgang ersetzt bzw. simuliert werden. Beispielsweise das Abbuchen von Geld auf einem Konto, oder Netzzugriffe. Also alles, was man in einem TestszENARIO nicht gerne ausblenden würde. In unserem Fall möchten wir die Mittelwertbildung testen und senden die Zahlen 2 und 6 hinein. Um die Addition auszublenden, können wir also einfach ein Mock verwenden, dass 8 zurückliefert ohne „kostbare Additionsressourcen“ zu verbrauchen.

Ziel dieser ganzen Aktion ist zu zeigen, dass mit Dependency Injection ganz leicht Implementierungen ausgetauscht werden können, ohne dass der eigentliche Quelltext im Calculator beeinflusst wird.

Der eigentliche **Taschenrechner** sieht nun so aus:

```
package de.vfh.swt.di;

import com.google.inject.Inject;

// Constraint: sum will always be even
public class Calculator {
    private final Calc calci;

    @Inject
    public Calculator(ICalc aCalc) {
        calci = aCalc;
    }

    int average(int a, int b) {
        int sum = calci.add(a, b);
        return sum / 2;
    }
}
```



Hier passiert jetzt Wichtiges:

- In der Methode `Average` wird die Abhängigkeit von `calc` (hier `calci`) verwendet und dann die Hälfte zurückgeliefert (was dem Mittelwert zweier Zahlen entspricht).
- Um hier aber Dependency Injection zu aktivieren - also die Abhängigkeit der Implementierung von `ICalc` austauschen zu können - müssen wir die folgenden Dinge unternehmen:
  1. Wir definieren ein Feld `calci` vom Typ `Calc` (es kann ruhig `private` sein!), in welches die Abhängigkeit injiziert werden soll.
  2. Jetzt muss nur noch mit `@Inject` annotiert werden, dass Google Guice hier per Konstruktor **Injection** die Abhängigkeit setzen (injizieren) soll.

Damit können wir jetzt eine **Testklasse** schreiben, die den Taschenrechner verwendet:

```
package de.vfh.swt.di;

import junit.framework.Assert;
import org.junit.Test;

import com.google.inject.Guice;
import com.google.inject.Injector;
public class CalculatorTest {

    @Test
    public void testAverage() {
        Injector i = Guice.createInjector();
        Calculator calci = i.getInstance(Calculator.class);
        Assert.assertEquals(4, calci.average(2,6));
    }
}
```

In der Anwendung muss einmal initial ein Injector erstellt werden. Dies ist quasi die bessere Factory, mit der alle Klassen geholt werden können. Genau das wird danach auch getan mit `getInstance(Calculator.class)`

Der Test wird dann ausgeführt und er ist erfolgreich = grün. Es wurde von Google die Klasse `calc` injiziert und  $(2+6)/2$  ist gleich 4.

Jetzt könnte man leicht `Calc` mit `CalcMock` austauschen, den wir ja oben schon definiert haben:

```
package de.vfh.swt.di;

import com.google.inject.ImplementedBy;

@ImplementedBy(CalcMock.class)
public interface ICalc {
    public int add(int a, int b);
}
```

Wir müssen also nur das Interface anders annotieren und schon wird der Test mit der Mock-Abhängigkeit aus Calculator geladen.

Wem das Annotieren von Interfaces nicht gefällt und wer die Konfiguration an einer - zentralen und compilergeprüften (!) Stelle halten will, der kann ein Google Guice Modul verwenden:

```
package de.vfh.swt.di;

import com.google.inject.Binder;
import com.google.inject.Module;

public class CalculatorModule implements Module { // oder extends
    public void configure(Binder binder) {
        binder.bind(ICalc.class).to(Calc.class);
    }
}
```

Hier können also einfach Klassen an Interfaces „gebunden“ werden.

Features von  
Google Guice

Das Framework kann noch einiges mehr:

- Google Guice bietet sowohl Konstruktor-, als auch Setter- und Field-Injection an.
- Google Guice kümmert sich um den gesamten Baum und Zyklen
- Google Guice kann Klassen im Singleton Scope zur Verfügung stellen (das spart Code, den man selber nicht schreiben möchte).
- Module können organisiert werden.
- Google Guice stellt AOP bereit. Beispielsweise ein einfaches „**Method Interception**“, d. h. die Ausführung von vordefinierten Methoden beim Aufruf von bestimmten (oder ausgezeichneten) Methodengruppen.

### Weiterführende Literatur & Links

- **ROBBIE VANBRANT**, „Google Guice“, Apress, ISBN 978-1-59059-997-6
- [!\[\]\(79de0df6c6ddd2d4eb74f1cc5f48ec50\_img.jpg\) Google Guice Homepage](#)  
Im Downloadbereich ist die Dokumentation als PDF zu finden.
- Artikel in [!\[\]\(d4c9768318b38eff1042b07478e20b4c\_img.jpg\) Dr. Dobbs](#)

## 9 DI mit Spring

Das Spring Framework ist vor einigen Jahren aus der konzeptionellen Arbeit von **ROD JOHNSON** entstanden.

Eines der Kernelemente von Spring ist das Dependency injection Modul, für welches sogenannte Spring Beans definiert werden können - üblicherweise in XML. Dies ist für viele Unternehmen ein Vorteil, da Beans als Dateien gemanagt werden können. Entwickler dagegen empfinden die Konfiguration von XML oft als lästig. Es liegt daher an Ihnen zu entscheiden, welches Konzept sie bevorzugen.

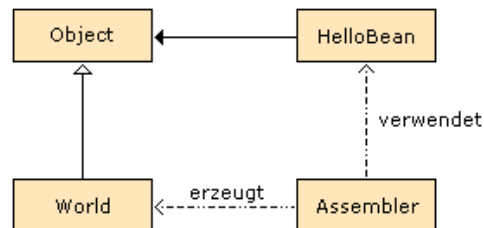


Abb.: Beispiel für Spring



Beispiel

Im nachfolgenden Code werden sowohl die **HelloBean** als auch die **WorldBean** geholt.

```
[package und import-Anweisungen]

public class Assembler {
    public static void main(String[] args) {
        ClassPathResource resource = new ClassPathResource(
            "helloworld/helloworld.xml");
        XmlBeanFactory factory = new XmlBeanFactory(resource);
        HelloBean helloBean = (HelloBean)factory.getBean("myHelloBean");
        Object bean = factory.getBean("myBean");
        helloBean.setBean(bean);
        helloBean.sayHello();
    }
}
```

Zur Veranschaulichung wird die **bean per Setter injection** in der **HelloBean** gesetzt, die dann ordentlich **HelloWorld** aufrufen kann. Hier gilt also das gleiche Prinzip: Über einen **Setter** kann das Feld der Abhängigkeit gesetzt werden.

Nun die eigentliche **HelloBean** mit **Setter** und privatem Feld:

```
package de.oreilly.springbasger.chapter2.helloworld2;

public class HelloBean {
    private Object bean;
    public void setBean(Object bean){
        this.bean = bean;
    }
    public void sayHello() {
        System.out.println("Hello " + this.bean);
    }
}
```

Es wäre kein fortschrittliches Dependency Injection Framework, wenn nicht beide Beans automatisch und daher implizit erzeugt werden könnten (sofern alle Beans in XML definiert worden sind!):

```
public static void main(String[] args) {
    ClassPathResource resource = new ClassPathResource(
        "helloworld/helloworld.xml");
    XmlBeanFactory factory = new XmlBeanFactory(resource);
    HelloBean helloBean = (HelloBean)factory.getBean("myHelloBean");
    helloBean.sayHello();
}
```

Spring Bean Definitionen sind in einer XML-Datei hinterlegt, die das Framework über die **ClassPath Resource** lesen kann:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
    [Bean Definitionen]
</beans>
```

Die Definitionen von Beans können folgendermaßen ausgestaltet werden:

```
<bean id="myBean" class="helloworld.class"/>

<bean id="Tom" class="model.Person">
    <constructor-arg value="Tom" />
</bean>

<bean id="admin" class="model.Person">
    <property name="firstname" value="Hermann" />
</bean>
```

Auch Spring liefert für die Bean-Konfiguration quasi alles, was auch Google Guice kann. So kann man mit Spring beispielsweise auch nur mit Annotations arbeiten und die XML-Dateien ebenfalls weglassen.

Weiterhin gibt es das Attribut **autowire="constructor"**. Damit wird die oben verwendete Spring Factory informiert, die Abhängigkeiten aus dem verwendeten Konstruktor per **Reflection** selbst zu finden. Dies erspart explizite Abhängigkeitsdefinitionen in der XML Datei.

---

## Literatur & Links

- Spring Homepage [www.springsource.org](http://www.springsource.org)
- **DANIEL OLTMANNS, STEFAN EDLICH** : Spring 2. BOD, ISBN 409764769798  
Frei downloadbar: [www.heise.de](http://www.heise.de))

## 10 PicoContainer

Abschließend soll kurz ein letztes bekanntes Framework für Java vorgestellt werden:

 PicoContainer.

Bei PicoContainer werden im einfachsten Fall, Klassen über Ihre `class` registriert und genauso auch wieder geholt:



```
PicoContainer pico = new DefaultPicoContainer();
pico.addComponent(Foo.class);
pico.getComponent(Foo.class).service();
```

Natürlich kennt auch PicoContainer viel mehr Verfahren und Möglichkeiten:

- Verwaltung von Singletons
- Verschiedene Injection Methoden
- Echtes Lifecycle-Management, das ein wenig an OSGi erinnert. Methoden können das Interface `Startable` implementieren. Dies beinhaltet die Aufrufe: (1) `start`, (2) `stop` und (3) `dispose`, die es dem Objekt ermöglichen, auf Ereignisse zu reagieren.

### Zusammenfassung

- Dependency Injection bezeichnet ein Entwurfsmuster, bei dem versucht wird, die Abhängigkeiten der Komponenten zu minimieren und diese mithilfe eines Frameworks oder anderen Mechanismen aufzulösen.
- Dependency Injection soll die Wartbarkeit, Änderbarkeit und Testbarkeit - insbesondere bei größeren Softwaresystemen - unterstützen
- Um eine Indirektion in Bezug auf Komponentenabhängigkeiten einzuführen, könnte man eine Factory nutzen oder die Abhängigkeit einfach so im Konstruktor übergeben. Beide Varianten haben Nachteile.
- Gute DI Frameworks kennen mehr Scopes als nur Singleton oder No-Scope (also immer ein frisches neues Objekt).
- Es gibt mehrere Varianten, wie die Auflösung von Abhängigkeiten realisiert werden kann.
- Gute Frameworks erkennen zyklische Referenzen.
- In kleinen Projekten kann Dependency Injection einen unnötigen Overhead bedeuten sodass die Konfiguration eines DI-Frameworks aufwendig sein kann.
- In der Regel sind Dependency Injection Werkzeuge deutlich mächtiger, als ein Service Locator, was sowohl ein Vor- als auch ein Nachteil sein kann.
- Gute DI-Frameworks integrieren ein Lifecycle Management.
- In Spring werden Abhängigkeiten üblicherweise in XML definiert.

---

Sie sind am Ende dieser Lerneinheit angelangt. Auf den folgenden Seiten finden Sie noch die Übung zur Wissensüberprüfung und die Literaturhinweise.

## Übung



Programmieren

### Übung DEP-01

#### DI-Framework untersuchen

Bitte wählen Sie ein Dependency Injection Framework aus, das Ihnen für Ihre Sprache und Umgebung geeignet erscheint. Erstellen Sie ein neues Projekt und verwenden Sie das Framework. Alternativ können Sie auch Ihr bisheriges Softwareprojekt auf DI umstellen.

Beantworten Sie die folgenden Fragen und bereiten Sie sich darauf vor, die Ergebnisse im Chat oder einem Web-Meeting vorzutragen. Stimmen Sie sich mit Ihrer Kursbetreuung über das Vorgehen und die Termine ab.

1. Was waren die Gründe für Ihre Entscheidung, welches Injection-Modell Sie verwenden? (Konstruktor, Setter?)
2. Was gefällt Ihnen an diesem Framework und was nicht?
3. Konnten Sie im Laufe der Zeit erkennen, dass Ihr Programm wartbarer oder änderbarer wurde?

Bearbeitungszeit: 60 Minuten

## Literatur

### Bücher

- **DHANJII R. PRASANNA**, „Dependency Injection“, Manning, July 2009
- **ROBBIE VANBRABRANT**, „Google Guice“, Apress, April 2008
- **CRAIG WALLS, RYAN BREIDENBACH**, „Spring in Action“, Manning, August 2007
- **THOMAS VAN DE VELDE, BRUCE SNYDER, CHRISTIAN DUPUIS, NAVEEN BALANI, SING LI, ANNE HORTON**, „Beginning Spring Framework 2“, Wiley & Sons, Dezember 2007
- **EBERHARD WORFF**, „Spring 2“, Dpunkt Verlag, April 2007
- **DANIEL OLTMANNS, STEFAN EDLICH**, „Spring 2 für Grünschnäbel“, BoD, Dezember 2007

### Links

- Erster Artikel von [www.Martin Fowler](#).
- Eintrag in [www.Wikipedia mit Werkzeugliste](#).
- Sehr guter Artikel auf [www.theserverside.com](#).
- Artikel in [www.Dr.Dobbs](#).
- Artikel von [www.Jakob Jenkov](#).
- Artikel im [www.Developer Magazin](#).
- Artikel in [www.MSDN](#).

### Podcasts

- se-radio.net: [www.Episode 2](#)



## Appendix

---

### Codebeispiel

```
class A
  def initialize a_dep
    @dep = a_dep
  end

  def do_sth
    @dep ||= Dependency.new
    puts "Hello" + @dep.work
  end
end

class Dependency
  def work
    return " world!"
  end
end

class Mock
  def work
    return " mock!"
  end
end

my_a = A.new nil
my_a.do_sth

my_b = A.new Mock.new
my_b.do_sth
```

### Demo zur Dependency Injection



Film

