FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Fachochschule Aachen, Campus Jülich

Fachbereich 09 - Medizintechnik und Technomathematik
Studiengang Angewandte Mathematik und Informatik

Seminararbeit

Dependency Injection in Java Spring

Autor:
Yannic Utens, 3531341

Betreuer:
Prof. Dr. Volker Sander
Michael Peter

Aachen, 25. Januar 2024

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema

“Dependency Injection in Java Spring”

Selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.
Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzu- bewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Ort, Datum	Yannic Utens

Diese Arbeit wurde beaufsichtigt von

Prof. Dr. Volker Sander	Michael Peter

Aachen, 25. Januar 2024

Zusammenfassung

Die folgende Arbeit behandelt die Verwendung von Dependency Injection (DI) und Inversion of Control (IoC) in Java Spring. Es bietet eine detaillierte Erklärung der Ziele von Spring, wie beispielsweise die Bereitstellung austauschbarer Komponenten auf verschiedenen Ebenen der Anwendung und die Schaffung intuitiver API-Zugänge. Des Weiteren wird ein praktisches Beispiel einer Todo-Liste-Applikation vorgestellt, die im Browser läuft und eine H2-Datenbank im Hintergrund verwendet. Die Anwendung benutzt dabei von Spring bereitgestellete Bibliotheken zum Verdeutlichen der beschriebenen Technologien.

Zusätzlich werden bei der Erläuterung von DI und IoC in Java Spring die Vorteile der Verwendung eines IoC-Containers hervorgehoben. Es wird auch diskutiert, wie DI die Wartung des Codes vereinfachen und die Verwendung derselben Klasseninstanz in der gesamten Anwendung ermöglichen kann. Darüber hinaus wird verdeutlicht, wie Spring das objektorientierte Programmieren, durch Einsetzen von 'design pattern', vereinfacht.

Abschließend behandelt die Arbeit die umfassende Natur von Spring als Framework, das auf DI und IoC basiert, und hebt die Freiheit hervor, die es Entwicklern bei der Strukturierung ihres Codes bietet. Es wird auch die Bedeutung des Verständnisses der zugrunde liegenden Technologien von Spring für eine effektive Code-Entwicklung unterstrichen. Insgesamt bietet die Arbeit eine detaillierte Erkundung von DI und IoC in Java Spring und bietet wertvolle Einblicke in deren praktische Anwendung und Vorteile.

Inhaltsverzeichnis

1 Motivation **1**

2 Inversion of Control **2**
2.1 Allgemeines . . . 2
2.2 Beispiel für Inversion of Control . . . 2
2.3 Model-View-Controller . . . 3

3 Dependency Injection **4**
3.1 Allgemeines . . . 4
3.2 Rollen bei Dependency Injection . . . 4
3.2.1 Services und Clients . . . 4
3.2.2 Interfaces . . . 4
3.2.3 Injectors . . . 5
3.3 Vor- und Nachteile . . . 5
3.3.1 Vorteile . . . 5
3.3.2 Nachteile . . . 5

4 DI und IoC in Java Spring **6**
4.1 Der Inversion of Control Container . . . 6
4.1.1 Spring Beans . . . 7
4.2 Dependency Injection . . . 8
4.2.1 Constructor Injection . . . 9
4.2.2 Setter Injection . . . 9
4.2.3 Property Injection . . . 10
4.2.4 Konfigurationsmethoden . . . 11
4.2.5 @Profile Annotation . . . 12

5 Todo-Liste **14**
5.1 Grundlage des Beispiels . . . 14
5.2 Aufbau . . . 14
5.2.1 H2-Datenbank . . . 14
5.2.2 Main Methode . . . 15
5.2.3 Data Objects . . . 15
5.2.4 Repository . . . 16
5.2.5 Service . . . 16

5.2.6 Controller 16
5.3 Zusammenfassung 17

6 Fazit **18**

A Literaturverzeichnis **19**

1 Motivation

Spring ist ein Framework für Java, welches Java flexibler aus Programmiersprache machen soll, so dass es in den verschiedensten Situationen eingesetzt werden kann. Spring wirbt mit 'out of the box solutions' und verspricht Entwicklern ihren Code zu vereinfachen. Die Kerntechnologien von Spring sind 'Inversion of Control' (IoC) oder auch 'Dependency Injection' (DI) genannt. Diese Programmiertechniken sollen dem Entwickler eine objektorientierte Basis bieten, um produktiver ordentlichen Code schreiben zu können.

Spring hat mehrere Ziele, die es durch das Einsetzen von DI versucht anzustreben. Zum Beispiel bietet es das Framework an, auf verschiedenen Ebenen der Applikation austauschbare Komponenten zu installieren. Die Komponenten sind dann austaschbar, ohne dass die Stellen von der Code eingesetzt wird, überarbeitet werden müssen. Außerdem bietet Spring intuitive API-Zugänge, die für Entwickler schnell zu verstehen sein sollen. Ein typisches Beipsiel für diese Funktionen ist das Austauschen von der im Hintergrund liegenden Datenbank durch Konfigurationsdatein. Dadurch ist es möglich die App je nach Kontext des Startens (Testlauf, Produktion, Entwicklung, etc.) mit anderen Randbedingungen und Komponenten laufen zu lassen.

Als Java Entwickler kommt man nicht daran vorbei, von Spring zu hören. Der Einstieg in Spring geht schnell, aber ohne die Technologien im Hintergrund zu verstehen, kann es schwierig sein, seinen Code in Spring zu überarbeiten. Diese Arbeit beschäftigt sich mit den genannten Technologien und soll einen Überblick bieten, wie Spring diese Technologien einsetzt. Außerdem wird ein Code-Beispiel gezeigt, welche verschiedene Praktiken des Spring Frameworks nutzt, die durch IoC and DI möglich werden.

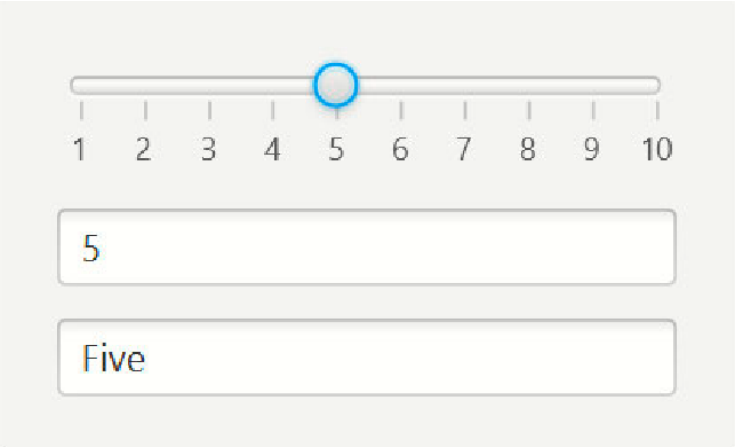
2 Inversion of Control

2.1 Allgemeines

Inversion of Control (IoC) ist ein weit verbreitetes Designmuster, das in vielen Frameworks eingesetzt wird, um den Fluss der Programmsteuerung umzukehren. Die Motivation dahinter ist es, die Interessen einzelner Programmabschnitte zu trennen(sepertation of concern) und Objekte lose zu verbinden(loose coupling) [Rob22]. Dadurch ist es möglich, umfangreiche Applikationen zu bauen, ohne das die Codekomplexität ein unüberblickbares Level erreicht.

2.2 Beispiel für Inversion of Control

Um das Konzept der Inversion of Control zu verdeutlichen, betrachten wir ein einfaches Beispiel im Kontext eine simpel gehaltenen Grafikoberfläche. Die Oberfläche [Rob22] soll wie folgt aussehen:

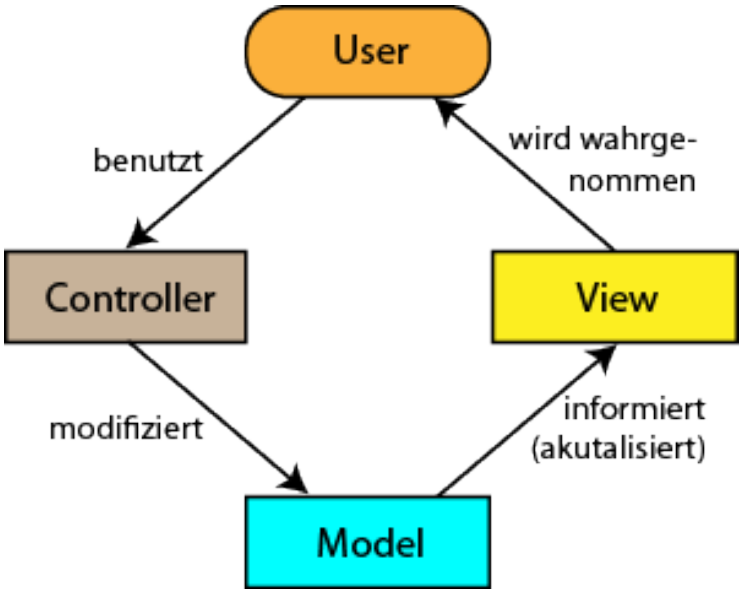


Auf der Oberfläche sind drei Darstellungen einer Zahl von 1 bis 10, die jeweils über den Slider, die numerische oder die Textdarstellung ausgewält werden kann. Alle drei sollen immer die gleiche Zahl darstellen. Ohne Inversion of Control läge es nah diese drei Darstellungen paarweise zu verbinden, indem man bei der Änderung eines immer die beiden

anderen benachrichtigt. Dies führt aber zu einer engen Verbindung und kann somit auch schwer erweitert werden. Mit jeder weiteren Darstellung dieser Zahl bekommt der Abhängigkeitengraph zwischen den Darstellungen n-mal mehr Kanten.
Ein verbreitetes 'design pattern', welches das paarweise Verbinden von Abhängigkeiten verhindert, ist das Model-View-Controller pattern.

2.3 Model-View-Controller

Die drei Hauptkomponeten des patterns teilen das Programm in seine Hauptfunktionen auf. Das Model ist für Abspeichern der Daten zuständing. Das View sorgt für die Aufarbeitung der Rohdaten des Models, sodass sie später auf der grafischen Oberfläsche richtig dargestellt werden. In dem eben genannten Beispiel wären das die drei verschiedenen Darstellung einer von Benutzer ausgewählten Zahl. Der Controller ist verantwortlich für das Senden der aufbereiteten Daten an die grafischen Oberfläche. Außerdem sorgt der Controller auch dafür, dass Änderungen auf der Oberfläche auch bei dem Model ankommen.
Der Ablauf [Ree] eines Programms kann dann vereinfacht wie folgt aussehen:



3 Dependency Injection

3.1 Allgemeines

Dependency Injection(DI) ist eine Programmiertechnik bei der ein Objekt andere benötigte Objekte zugeschickt bekommt, anstelle sie intern zu erstellen. [Pra09] Durch dieses 'Einfügen der Abhängigkeit' werden die wichtigen Teile eines Programms in ihre eigentlichen Funktionen aufgeteilt, was ganz im Sinne des objektorientiertem Programmierens ist. Ein solches Programm wird auch als 'loosly coupled program' bezeichnet, was einige Vorteile mit sich bringt. Wenn einzelne Kompeneten eines solchen Programms ausgetauscht werden, muss nur die Komponente an sich verändert werden, ohne dass man viel am restlichen Code ändern muss. Dadurch sind solche Programme nicht strikt auf eine Sprache oder Betriebsystem angewiesen. Durch diese Verhaltensweise wird sichergestellt, dass ein Programmabschnitt nicht wissen muss wie eine 'service' erstellt und nur mit seiner Abhägigkeit versehen wird.

3.2 Rollen bei Dependency Injection

3.2.1 Services und Clients

Ein Service ist eine umfassende Klasse, die jeden Wert, jede Funktion oder jedes Merkmal umfasst, das eine Anwendung benötigt. Ein Service ist in der Regel eine Klasse mit einem eindeutig definierten Zweck. Jede Klasse kann ein Client und ein Service sein. Ihre Bezeichnung hängt von ihrer Rolle bei dem Einsetzen einer Abhängigkeit ab. Ein Service wird immer in einen Client eingesetzt.

3.2.2 Interfaces

Ein Interface sorgt für die Abstraktion eines Services, so dass das Element, welches den Service implementiert, nicht wissen muss, wie dieser Service implementiert wird. Durch diese Abstraktion ist es möglich Abhängigkeit zu überarbeiten, ohne dabei ebenfalls den Client zu ändern.

3.2.3 Injectors

Assembler, container, provider oder auch factorys sind alles Arten von Injectors. Sie sorgen dafür, dass ein Client seine Services erhält. In einem komplexen System, wo mehrere Objekte abhängig von anderern Objekten sind und einen sogenannten *Dependency Tree* (Abhängigkeitenbaum) bilden, muss es dem Injector möglich sein, diesen Baum aufzulösen und korrekt die Abhängigkeiten einzusetzen. Dabei ist wichtig, dass zwei Objekte nicht abhängig voneinander sind, sonst entsteht eine kreisförmige Abhängigkeit, die nicht aufgelöst werden kann.

3.3 Vor- und Nachteile

3.3.1 Vorteile

Durch das Aufteilen von Abhängigkeiten und Funktionen bekommt der Code eine bessere Wiederverwendbarkeit. Die einzelnen Kompenenten können nämlich jetzt in verschiedenen Kontexten verwendet und speziell in einem Testkontext auch ausgetauscht werden. Außerdem kann der Code dadurch auch wesentlich besser gewartet werden. Da das Einsetzen von Abhängigkeiten von dem Framework übernommen wird, spart man sich auch den *boilerplate code* (gleicher, häufig vorkommender Code) der zum Erstellen einzelner Instanzen nötigt wäre.

Auch das Arbeiten im Team wird durch DI vereinfacht. Mehrere Entwickler können an voneinander abhängigem Code arbeiten, da sie nur die dahinter liegenden Interfaces kennen müssen und nicht deren speziellen Implementation.

Das Testen von Code wird dadruch vereinfacht, dass man zur Laufzeit des Programms Konfigurationen austauschen kann, so dass für jedes Testziel bestimmte Implementationen eingesetzt werden können.

3.3.2 Nachteile

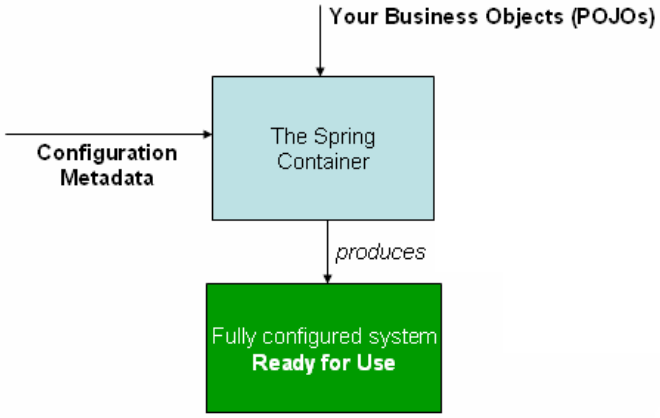
Durch DI ist das Verhalten innerhalb einer Applikation getrennt von der Erstellung der Verhaltensobjekte. Dadurch ist es für Entwickler schwer, die Funktionsweise dieser nachzuvollziehen. Für kleinere Applikationen kann das zu mehr Übersichtlichkeit führen, da die ganzen Konfigurationen ausgelagert sind, aber bei einem großen Projekt kann so viel verstreute Konfiguration schnell unübersichtlich werden, vor allen Digen bei der Fehlerbehebung oder Einarbeiten von vorhandenem Code.
Schließlich ist ein weiterer Nachteil, dass der Entwicklungsprozess abhängig ist vom Framework, welches DI implementiert. Der Programmierer hat keinen Einfluss auf die Entwicklung dieses Frameworks und dessen Lebenszeit.

4 DI und IoC in Java Spring

4.1 Der Inversion of Control Container

Spring ist ein Paradebeispiel für die Verwendung von Inversion of Control und Dependency Injection. Für die Implementierung von IoC benutzt Spring einen Container der die Fuktion eines Injectors hat. Der Container sorgt für das Einsetzen der Abhängigkeiten in die sogennaten Spring Beans 4.1.1. Die Klassen org.springframework.beans und org.springframework.context sind das Herz des IoC Containers. Eine Klasse names BeanFactory stellt Basisfunktionen für die Verwaltung von Beans bereit. Das Interface ApplikationContext erweitert diese Funktionen um Eventverwaltung und ist verantwortlich für das Erstellen von Beans gemäß vorher definierter Konfigurationen. Metadaten von diesen Konfigurationen werden über XML, Java Annotations oder ganz normalen Java Code erstellt.

Eine von Spring bereitgestellte Abbildung [Sprc] soll die Funktion des IoC Containers verdeutlichen:



In der Abbildung ist zu sehen, dass der Spring Container direkt dafür verantwortlich ist, einen funktionsfähigen Code zu erstellen. Dieser Code wird zusammengestellt aus den definierten Metadaten und dem Code des Entwicklers(business logic). Metadaten müssen aber in vielen Fällen nicht explizit definiert werden, da Spring für die meisten Standartanwendungen 'default metadata' bereitstellt.

4.1.1 Spring Beans

Es ist schwierig Beans in seiner Gesamtheit darzustellen. Springs offizielle Website verweist auf folgenden Definition [Spra]:

> In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.

Beans in der Anwendung

Um Beans besser zu verstehen, ist es am besten an einem Beispiel [Tha]: zu zeigen, was man mit ihnen machen kann.

Angenommen man programmiert eine Domänenklasse, die ein Unternehmen darstellt. Diese Klasse besitzt eine Referenz auf die Adresse des Unternehmens. Die Adressklasse hat einen Straßennamen und eine Hausnummer. Das würde dann wie folgt aussehen:

```
public class Company {
    private Address address;

    public Company(Address address) {
        this.address = address;
    }

    // getter, setter and other properties
}

public class Address {
    private String street;
    private int number;

    public Address(String street, int number) {
        this.street = street;
        this.number = number;
    }

    // getters and setters
}
```

Ohne IoC müssten wir auf normalen Weg diese Klassen initiieren:

```
Adress adress = new Adress("Templergraben",55);
Company company = new Company(adress);
```

Für so ein kleines Beispiel ist dieser Ansatz vollkommen in Ordnung, aber für eine Applikation mit einer wesentlich größeren Domäne kann es bei der Programmierung ohne DI zu Schwierigkeiten kommen. Wenn wir nämlich die Domäne im Nachhinein ändern wollen, kann es zu großen *refactoring* Aufwand kommen. Außerdem kann es sein, dass die gleiche Instanz einer Klasse über die ganze Applikation verwendet werden soll. Dafür könnte jetzt der IoC Container verwendet werden. Dies sieht in Java Spring wie folgt aus:

```
@Component
public class Company {
    // this body is the same as before
}
@Configuration
@ComponentScan(basePackageClasses = Company.class)
public class Config {
    @Bean
    public Address getAddress() {
        return new Address("High Street", 1000);
    }
}
```

Die Metadaten Konfiguration erfolgte jetzt über Java Annotations. Die Bean in der Config-Klasse erzeugt eine Instanz der Adressklasse. Das Besondere ist jetzt, dass über die @ComponentScan Annotation dem Container gesagt wird, dass er in der Company Klasse nach Abhängigkeiten suchen soll, die zu der in der Config-Klasse definierten Bean passen. Somit wird, wenn eine Company-Instanze erschaffen wird, auch gleichzeitig die definierte Instanz von Adress erschaffen. Somit ist der Lauf des Programms umgedreht. Überprüfen lässt sich das, indem man eine Instanz von ApplikationContext erzeugt:

```
ApplicationContext context = new AnnotationConfigApplicationContext(Confi
Company company = context.getBean("company", Company.class);
```

Lässt man sich nun *company.getAdress()* ausgeben, würde man die in der Config definierte Instanz erhalten. Bei diesem Beispiel wurde auch schon Dependency Injection angewendet. Die Instanz von Adress wurde in eine Company Instanz implizit eingesetzt.

4.2 Dependency Injection

Um Dependency Injection zu benutzen gibt es verschiedene Möglichkeiten. [Sprb]

4.2.1 Constructor Injection

Die folgende Klasse ist ein Beispiel für Construktor Injection:

```
public class SimpleMovieLister {

        private final MovieFinder movieFinder;

        public SimpleMovieLister(MovieFinder movieFinder) {
                this.movieFinder = movieFinder;
        }
}
```

Dies ist erstmal eine ganz normale Klasse, die Abhängigkeiten über den Konstruktor bekommt. Eine Möglichkeit, um diese einzusetzen, ist über eine auf XML basierende Konfigurationsdatei, die wie folgt aussehen könnte:

```
<beans>
<bean id="movielister" class="package.example.SimpleMovieLister">
        <constructor-arg ref="moviefinder"/>
</bean>

<bean id="moviefinder" class="package.example.MovieFinder"/>
</beans>
```

4.2.2 Setter Injection

Setter Injection hat einen ähnlichen Aufbau wie Constructor Injection mit dem Unterschied, dass man den Konstruktor leer lassen kann:

```
    public class ExampleBean {

        private AnotherBean beanOne;

        public void setBeanOne(AnotherBean beanOne) {
                this.beanOne = beanOne;
        }
}
```

Für das Einsetzen der Abhängigkeiten, sieht die Konfigurationsdatei wie folgt aus:

```
<bean id="exampleBean" class="examples.ExampleBean">
        <property name="beanOne">
                <ref bean="anotherExampleBean"/>
```

```
        </property>
</bean>
```

Beider dieser Möglichkeiten führen zum selben Ziel, aber ihre Verwendung muss man unterscheiden. Bei Konstruktor basierender Dependency Injection verwendet man in der Regel Abhängigkeiten, die nicht *null* sein sollen, wobei Setter-Injection optionale Abhängigkeiten erzeugt.

4.2.3 Property Injection

Bei Property Injection wird einem Attribut einer Klasse eine Abhängigkeit eingefügt. Damit Spring weiß, dass dieses Attribut eine Abhängigkeit bekommen soll, kann man das Attribut mit der Annotation @Autowired kennzeichnen. Spring kann anschließend über den IoC Container die richtige Bean in das Attribut einsetzten.

Beispiel

Es existiert eine Domäne bestehend aus einem Store der ein Verkaufs-Item enthält:

```
public class Store {
    private Item item;
    public Store(Item item) {
        this.item = item;
    }
}
```

Für die Zwecke dieses Beispiels ist die genaue Implementierung der Item Klasse unwichtig. Durch Autowiring würde die Klasse dann wie folgt aussehen:

```
public class Store{
    @Autowired
    private Item item;
}
```

Um jetzt eine Instanze von Item in Store einfügen zu können, müssen wir in einer Konfigurationsdatei festlegen, welche Instanz eingefügt wird:

```
<bean id="item" class="org.baeldung.store.ItemImpl1" />

<bean id="store" class="org.baeldung.store.Store"
autowire="byName">
</bean>
```

Item ist also in dem Beispiel nur ein Interface. Durch die Konfigurationsdatei wird spezifiziert, welche Implementation von Item in Store eingesetzt wird.

4.2.4 Konfigurationsmethoden

Wie bereits an den Beispielen zu sehen ist, gibt es verschiedene Methoden um eine Bean zu definieren und diese in eine andere Einzusetzen. Die drei Hauptmethoden sind dabei die XML-Konfigurationsdatei, das Autowiring und das Definieren von Beans über Java Code mit der Configuration Annotation.
Definiert man jetzt mehrere Beans des gleichen Typen, und versucht dann so eine Bean über DI zu erstellen, dann ist Spring verwirrt. Es muss expliziet definiert werden, welche Bean eingefügt werden soll und dafür gibt es drei Möglichkeiten für unterschiedliche Situationen:

@Primary Annotation

Mit der Primary Annatation wird angegeben, welche der beiden Beans eher erstellt und eingesetzt werden soll [baea]:

```
    @Configuration
public class Config {

    @Bean
    public Employee JohnEmployee() {
        return new Employee("John");
    }

    @Bean
    @Primary
    public Employee TonyEmployee() {
        return new Employee("Tony");
    }
}
```

Falls diese Bean nicht existiert wird die andere genommen. Somit weiß Spring immer, welche Bean zu nehmen ist

@Qualifier Annotation

Bei der Qualifier Annotation kann expliziet angegeben werden, welche Bean wo eingesetzt wird [baeb]:

```
    @Component
@Qualifier("fooFormatter")
public class FooFormatter implements Formatter {}

@Component
@Qualifier("barFormatter")
public class BarFormatter implements Formatter {}

public class FooService {
    @Autowired
    @Qualifier("fooFormatter")
    private Formatter formatter;
}
```

Wenn beides, Qualifier und Primary angegeben sind, dann hat der Qualifier immer den Vorrang. Der Primary hingegen definiert nämlich nur einen Standart.

4.2.5 @Profile Annotation

Die @Profile Annotation wird in einem anderen Kontext verwendet. Um diese zu benutzen brauchen man ebenfalls zwei Beans des gleichen Typen. Der Unterschied ist, dass man ein 'profile' für einen bestimmten Applikationskontext erschafft und somit je nach Kontext andere Beans einfügt. Ein Beispiel dafür ist das Austauschen von Beans, die für eine Datenbankanbindung verantwortlich sind in einem Testkontext, um Produktionsdaten beim Testen nicht zu beschädigen. Dies könnte wie folgt aussehen [Par]:
Zuerst werden die Beans mit einem 'profile' betitelt:

```
@Component
@Profile("dev")
public class DevDatasourceConfig implements DatasourceConfig {
    @Override
    public void setup() {
        System.out.println("Setting up datasource for DEV environment. ")
    }
}

@Component
@Profile("production")
public class ProductionDatasourceConfig implements DatasourceConfig {
    @Override
    public void setup() {
       System.out.println("Setting up datasource for PRODUCTION environme
```

```
    }
}
```

Dies könnte man auch wieder über XML machen:

```
<beans profile="dev">
    <bean id="devDatasourceConfig"
    class="org.baeldung.profiles.DevDatasourceConfig" />
</beans>
```

Als nächstes muss man den aktiven Kontext einstellen. Es gibt viele verschiedene Möglichkeiten dafür, aber in einem Testkontext bietet es sich an, in der jeweiligen application.properties Datei (die propertie-Datei vom Production-Code oder Test-Code) die folgede Zeile hinzuzufügen:

```
spring.profiles.active=dev
```

bezieungsweise:

```
spring.profiles.active=prod
```

Nun wird aus einem Integrationstest die richtige Bean verwendet wenn diese eingesetzt wird:

```
public class SpringProfilesWithMavenPropertiesIntegrationTest {
    @Autowired
    DatasourceConfig datasourceConfig;

    public void setupDatasource() {
        datasourceConfig.setup();
    }
}
```

5 Todo-Liste

5.1 Grundlage des Beispiels

Die Idee hinter der Applikation ist eine Todo-Liste[1]. Die Applikation soll im Browser laufen und im Hintergrund soll eine Datenbank angebunden werden. Damit Spring in das Projekt eingebunden ist, wird Maven Managment System verwendet. Abgesehen von Spring Boot werden außerdem noch die H2 Datenbank, die Lombok Biliothek und Spring Erweiterungen eingebunden, die für eine Webanbindung und Testbarkeit des Projekts sorgen. H2 ist eine Datenbank, die zur Laufzeit des Programms erstellt wird und Lombok sorgt für automatisch generierten 'boiler code' wie getter,setter und constructor.

5.2 Aufbau

5.2.1 H2-Datenbank

Für die Datenbank muss die Datei application.properties angepasste werden:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

Die Datenbank ist als in-memory Datenbank definiert und existiert nur zur Laufzeit. Es handelt sich bei H2 um eine SQL Datenbank, aber alle Tabellen für die Datenbank werden von Spring automatisch durch Entity Klassen erstellt.

[1] https://git.fh-aachen.de/yu8931s/seminar

5.2.2 Main Methode

Durch die Standartkonfiguration von Spring kann die main-Methode zum Starten des Programms kompakt gehalten werden:

```
@SpringBootApplication
public class SeminarApplication {
        public static void main(String[] args) {
            SpringApplication.run(SeminarApplication.class, args);
        }

}
```

Die Annotation SpringBootApplication fasst einige Konfigurationen mit Standartwerten zusammen. Zu aller erst wird Spring Boots auto-configuration aktiviert. Dadurch wird ein gewisser Applikationskontext definiert basierend auf den Bibliotheken, die in das Projekt importiert wurden. In diesem Beispiel ist das ein Kontext für eine Webapp und für die H2-Datenbank. Als nächstes wird der ComponentScan aktiviert, was in der Applikation Dependency Injection ermöglicht. Zu letzt kriegt die App eine Configuration Annotation, die dafür sorgt, dass man in dem Kontext neue Beans hinzufügen kann und diese über Konfigurationsdatein definieren kann.

5.2.3 Data Objects

Das Data-Object der Todo-Liste beinhaltet die wichtigen Informationen, die an das Frontend geschickt werden und von dort verändert wieder ans Repository kommen:

```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class TodoEntity {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;
        private String task;
        private boolean completed;
}
```

Die Annotations Data, NoArgsConstructor und AllArgsConstructor sind von Lombok und sorgen für den richtigen 'boiler code', um die Daten von dem Objekt zu erhalten und zu verändern. Das wichtige an der Klasse ist die Entity Annotation. Die Datenobjekte müssen damit gekennzeichnet sein, da für das Repository Spring Data Jpa verwendet wird.

5.2.4 Repository

Durch die Hilfe von Spring Data Jpa kann das Repository ebenfalls sehr kompakt gehalten werden:

```
@Repository
public interface TodoRepository extends JpaRepository<TodoEntity, Long> {
}
```

Das Repository ist vorerst nur ein Interface. Es ist definiert, dass dieses Repository TodoEnitys persistent hält und die Id von TodoEntity ein Long ist. Mit der Repository Annotation erlaubt man dem Komponenten-Scan die Klasse zu finden und in späteren Codeabschnitten eine Implementation von dem Repsoitory zu verwenden.

Die Service Annotation macht die Klasse ebenfalls zu einer Komponente und darf somit eine TodoRepository Bean von dem Framework bekommen.

5.2.5 Service

Der Service stellt die Verbindung zwischen dem Repository und dem Controller her. Es muss also jetzt eine Instanz von dem TodoRepository angebunden werden. Durch Dependency Injection ist es möglich das Repository durch @autowire in den Service einzufügen:

```
@Service
public class TodoService {
    @Autowired
    private TodoRepository todoRepository;

    //Service Methoden zum bearbeiten der Daten
}
```

Da keine zusätzlichen Konfigurationen angegeben sind, erstellt Spring eine Standartimplementation eines Repository. Dadurch ist es möglich Methoden aus dem Repository wie save, findAll oder findById aufzurufen.

5.2.6 Controller

Der Controller ist die Anbindung an das Frontend ist ist fürs Routehandling verantwortlich. In dieser Klasse ist nun eine Instanz von dem Service nötig. Da im Service das Repository mit Autowire eingefügt wird, muss der Service selber auch mit Autowire erstellt werden:

```
@Controller
@RequestMapping("/todos")
public class TodoController {
    @Autowired
    private TodoService todoService;

    //weitere Routing Methoden
}
```

Mit der Controller Annotation wird auch der Controller sichtbar für den Komponentenscan und kann somit Autowire verwenden.

5.3 Zusammenfassung

In der Applikation wurde ein linearer 'Dependency Tree' aufgebaut und die Abhängigkeit ist wie folgt:

Controller → Service → *Repository*

Die Abhängigkeiten werden durch Autowire eingefügt, wobei viel durch automatsiche Codegenerierung geschiet. Das ist der große Vorteil den Spring bietet.

6 Fazit

Spring bietet ein umfassendes Framework, welches auf Dependency Injection und Inversion of Control basiert. Während Inversion of Control eine im Hintergrund liegende Technologie bei Spring ist, die nicht expliziet konfiguriert werden muss und benutzt werden muss, spielt Dependency Injection einer der wichtigsten Rollen in Spring. Es gibt viele verschieden Möglichkeiten, DI in Spring zu verwenden, die dem Developer freien Raum bei dem Aufbau des Codes lassen.

Die Dependency Injection in Spring hat viele Vorteile, wie erhöhte Testbarkeit, Wartbarkeit und das Reduzieren von 'boilerplate code'. Es gibt aber auch Nachteile die es mit sich zieht. Da Spring viele Konfigurationen automatisch erstellt, verliert ein Entwickler den Überblick darüber, was in der Applikation schon konfiguriert wurde und was nicht. Außerdem kann sowas auch Fehlerquellen verursachen, die durch Annotationen verursacht werden.

Zusammengefasst kann man aber behaupten, dass das Programmieren mit Java durch Spring vereinfacht wird und das sowohl die Idee als auch die Umsetzung von Dependency Injection in Spring eine Bereicherung ist.

A Literaturverzeichnis

[baea] baeldung. Spring @primary annotation. *https://www.baeldung.com/spring-primary*. 26.12.23.

[baeb] baeldung. The spring @qualifier annotation. *https://www.baeldung.com/spring-qualifier-annotation*. 26.12.23.

[Par] Eugen Paraschiv. Spring profiles. *https://www.baeldung.com/spring-profiles*. 26.12.23.

[Pra09] Dhananjay Prasanna. *Dependency Injection: Design patterns using Spring and Guice*. Simon and Schuster, 2009.

[Ree] Trygve M. H. Reenskaug. The model-view-controller (mvc) – its past and present. *http://home.ifi.uio.no/trygver/2003/javazone-jaoo/MVC_pattern.pdf*. 25.12.23.

[Rob22] Martin P. Robillard. *Introduction to Software Design in Java*. Springer, 2022.

[Spra] Spring. Bean overview. *https://docs.spring.io/spring-framework/reference/core/bean/introduction.html*. Letzter Besuch: 15.12.23.

[Sprb] Spring. Dependency injection. *https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html*. Letzter Besuch: 16.12.23.

[Sprc] Spring. Spring container overview. *https://docs.spring.io/spring-framework/reference/core/beans/basics.html*. Letzter Besuch: 15.12.23.

[Tha] Nguyen Nam Thai. What is a spring bean? *https://www.baeldung.com/spring-bean*. Letzter Besuch: 15.12.23.