Arbeitsblatt 2: DI mit dem Spring Framework

Voraussetzung
Arbeitsblatt 1 ist bearbeitet und Sie können Spring-Applikationen erfolgreich ausführen.

Ziel
Sie bauen ein kleines „Hello World“ Programm zu einer Spring-Applikation mit Dependency Injection (DI) und entsprechenden Spring Beans aus, wobei diese Spring-Applikation nur DI einsetzt und sonst auf alle Spring Features verzichtet.

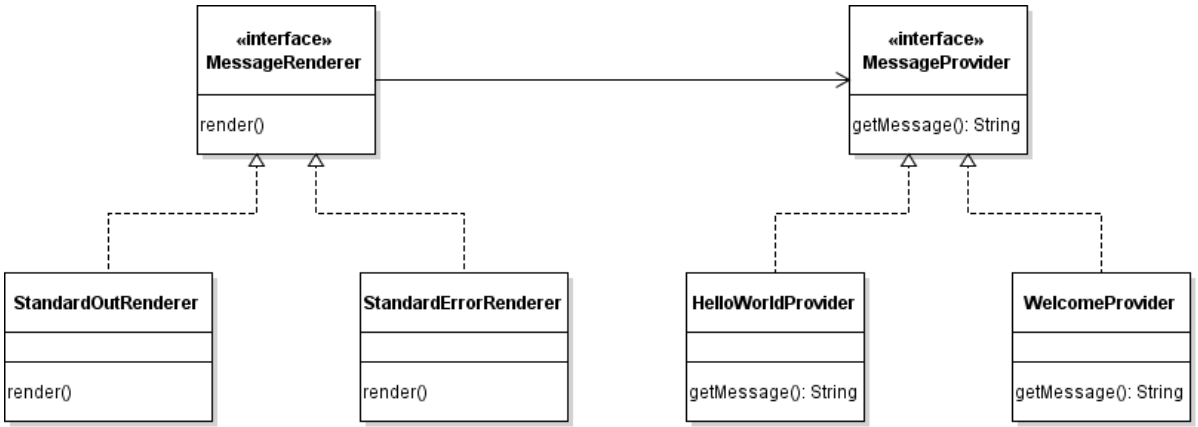
Wir gehen von folgendem Programm aus:

```
@SpringBootApplication
public class SpringIocApplication implements CommandLineRunner {

	public static void main(String[] args) {
		SpringApplication.run(SpringIocApplication.class, args);
	}

	@Override
	public void run(String... args) {
		System.out.println("Hello World");
	}

}
```

Die Applikation soll jedoch folgende Anforderungen erfüllen:

- Sie soll einen flexiblen Mechanismus vorweisen, um den Meldungserzeuger z.B. der Meldung "Hello World" einfach auswechseln zu können. Als Provider solcher Meldungen sollen Sie beispielhaft `HelloWorldProvider` (liefert „Hello World“) und `WelcomeProvider` (liefert „Herzlich willkommen“) implementieren.

- Die Darstellung der Meldung und das Verhalten, z.B. hier Ausgabe über `stdout`, sollen ebenfalls einfach ausgewechselt werden können. Auch hier sollen Sie zwei Renderer implementieren: `StandardOutRenderer` (Ausgabe auf *stdout*) und `StandardErrorRenderer` (Ausgabe auf *stderr*).



Aufgaben

1) Programmieren Sie die Spring-Applikation `SpringIocApplication` gemäss Aufgabenstellung. Programmieren Sie die entsprechenden Interfaces und POJO Klassen. Verwenden Sie die `@Primary` oder `@Qualifier` Annotation um die beiden Varianten unterscheiden zu können.

   In der Methode `run` können Sie auf dem eingesteckten Message-Renderer die Methode `render` aufrufen.

2) Schreiben Sie einen JUnit Test der prüft, ob beim Welcome-Provider die erwartete Meldung ausgegeben wird. Sie können dazu im Test ein Feld vom Typ `WelcomeProvider` definieren.

3) Übergeben Sie den beim `HelloWorldProvider` auszugebenden Text über das Property-File.

4) Erzeugen Sie die beiden Renderer-Provider über eine Java-Konfigurations-Klasse (die Annotationen auf den Renderer-Providern müssen Sie dabei wieder löschen, sonst sind die Beans mehrfach vorhanden).

5) Definieren Sie als letztes ihre Beans mit einem XML Konfigurationsfile:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

       <bean name="messageProvider" class="…">
       </bean>

       <bean name="messageRenderer" class="…">
           <property name="provider" ref="messageProvider"/>
       </bean>

</beans>
```

   und fügen Sie diese Konfiguration der Spring-Boot Applikation hinzu:

```
@SpringBootApplication
@ImportResource("classpath:app-config.xml")
public class SpringIocApplication implements CommandLineRunner {
```

   Ich nehme dabei an, dass das XML-File `app-config.xml` heisst und im `src/main/resources` Verzeichnis abgelegt ist. Um die im XML-File konfigurierte Version verwenden zu können muss eine Variable

```
@Autowired
BeanFactory context;
```

   oder

```
@Autowired
ApplicationContext context;
```

   definiert werden und dann kann in der `run`-Methode wie folgt aus dieses Bean zugegriffen werden:

```
@Override
public void run(String... args) {
   // renderer.render();
   MessageRenderer renderer = (MessageRenderer) context.getBean("messageRenderer");
   renderer.render();
}
```

   Diese Art des Zugriffs ist nur nötig falls die Konfiguration nicht eindeutig ist, d.h. falls mehrere Renderer zur Verfügung stehen. Über den Kontext kann auch auf die mit Annotationen definierten Beans zugegriffen werden.