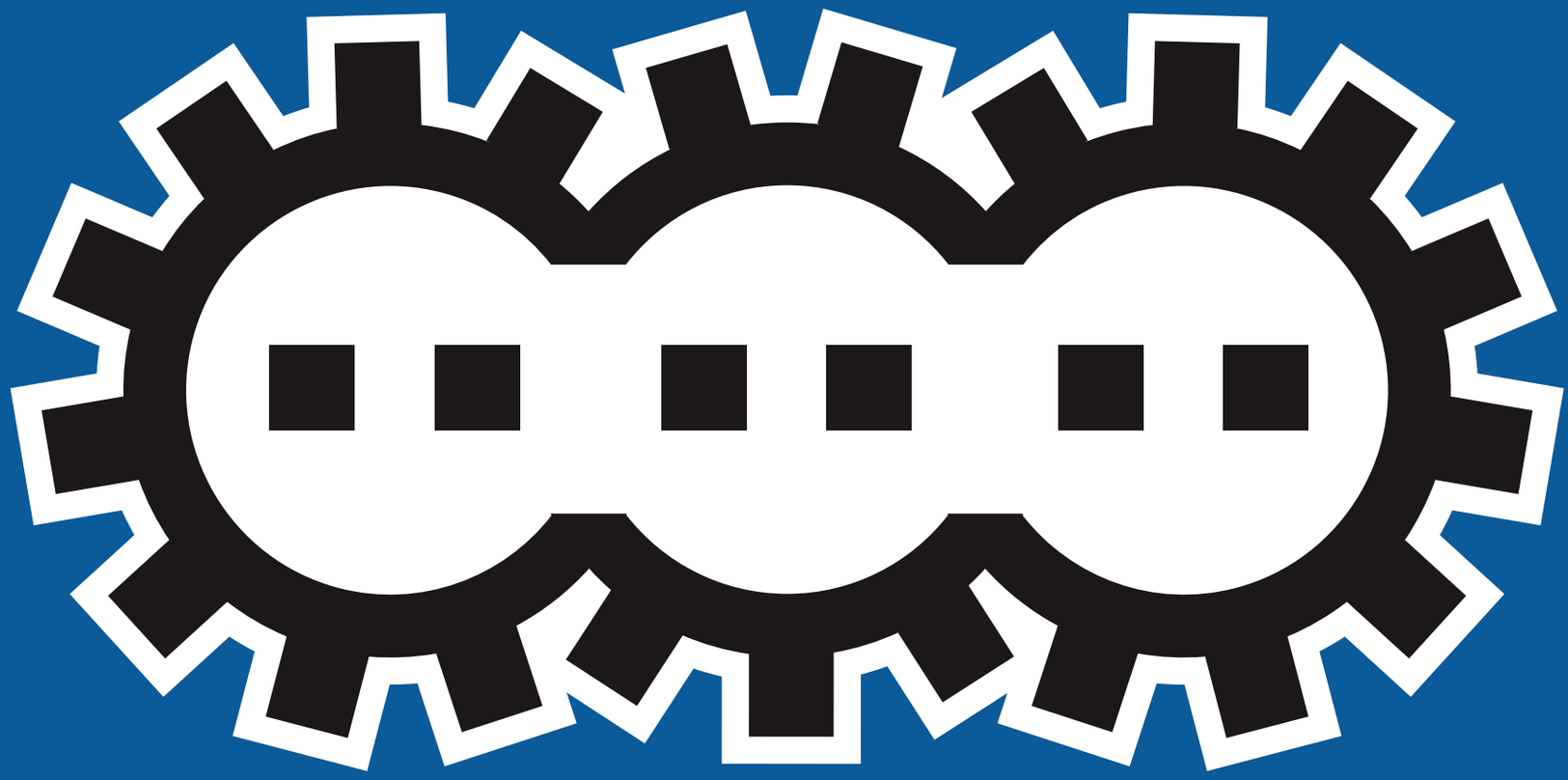
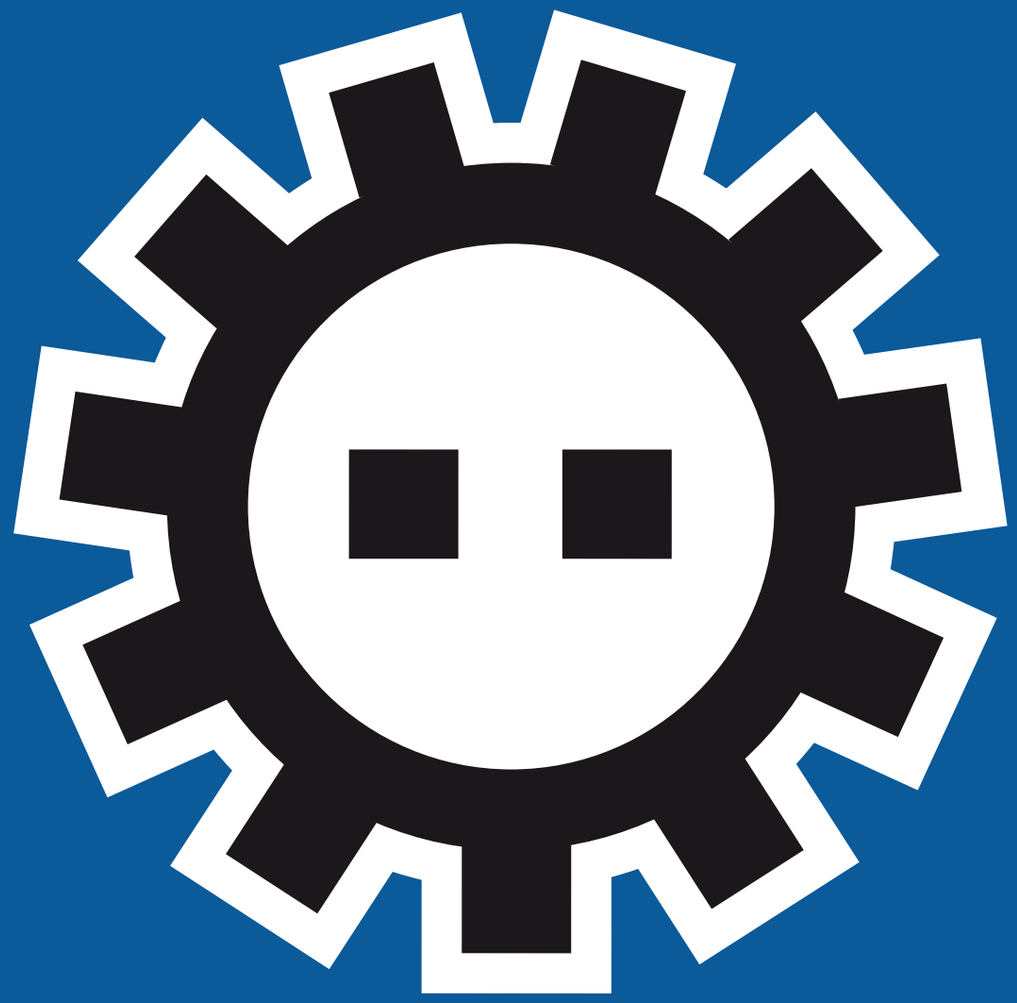


Software Engineering 2

07 - Spring Framework

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen



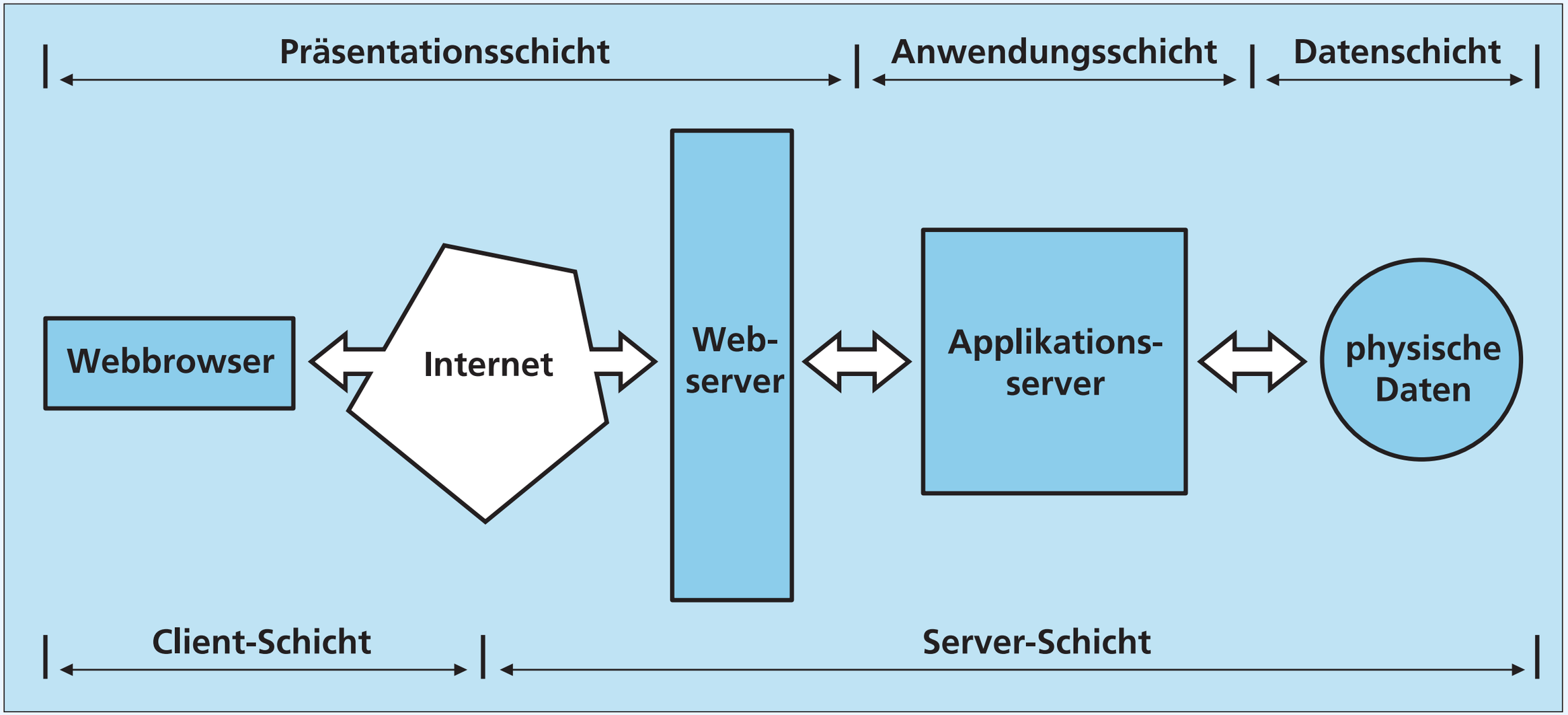
Motivation

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Voraussetzungen

- Client-Side-Techniken
  - HTML
  - CSS
  - JavaScript
- Grundkenntnisse Internet-Technologien
  - HTTP
  - URL
- Programmieren in Java
- Grundkenntnisse Software-Entwicklung
- Kenntnisse der 3-Schichten-Architektur

3-Schichten-Architektur



Java Standard Framework: JEE

J2EE Umsetzung

- EJBs (Enterprise Java Beans) sind verteilt und wiederverwendbar
  - laufen im Application Server
  - können lokal und entfernt aufgerufen werden
  - erhalten vom automatisch vom Application Server:
    - » Zugriffsschutz / Sicherheit
    - » Transaktionen
- Servlets und JSPs definieren die Web-Schnittstelle und greifen auf EJBs als Anwendung zu
- Klingt gut, aber…

komplexe EJBs

- Ein einfaches "hello world"-EJB:
  - Remote Interface (entfernter Zugriff)
  - Home Interface (Erzeugung der EJB)
  - Bean Implementierung
  - XML-Konfiguration

- Bitte schnell vergessen!

```
<ejb-jar>
  <display-name>HelloWorld</display-name>
  <enterprise-beans>
    <session>
      </description>
      <display-name>HelloWorld</display-name>
      <ejb-name>HelloWorld</ejb-name>
      <home>HelloWorldHome</home>
      <remote>HelloWorld</remote>
      <ejb-class>HelloWorldBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>HelloWorld</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

```
import java.rmi.*;
import javax.ejb.*;
public interface HelloWorld extends EJBObject {

  public String sayHello() throws
    RemoteException;

}
```

```
import java.rmi.RemoteException;
import java.ejb.CreateException;
import javax.ejb.EJBHome ;

public interface HelloWorldHome extends EJBHome {

  public HelloWorld create() throws
  CreateException, RemoteException;

}
```

```
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class HelloWorldBean implements SessionBean {

  // Methods of Remote interface
  public String sayHello() {
    return "Hello, world !";
  }

  // Methods of Home interface
  public void ejbCreate() {}

  // Methods of SessionBean interface
  protected SessionContext ctx;

  public void setSessionContext(SessionContext ctx) {
    this.ctx = ctx;
  }
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
}
```

Pojos

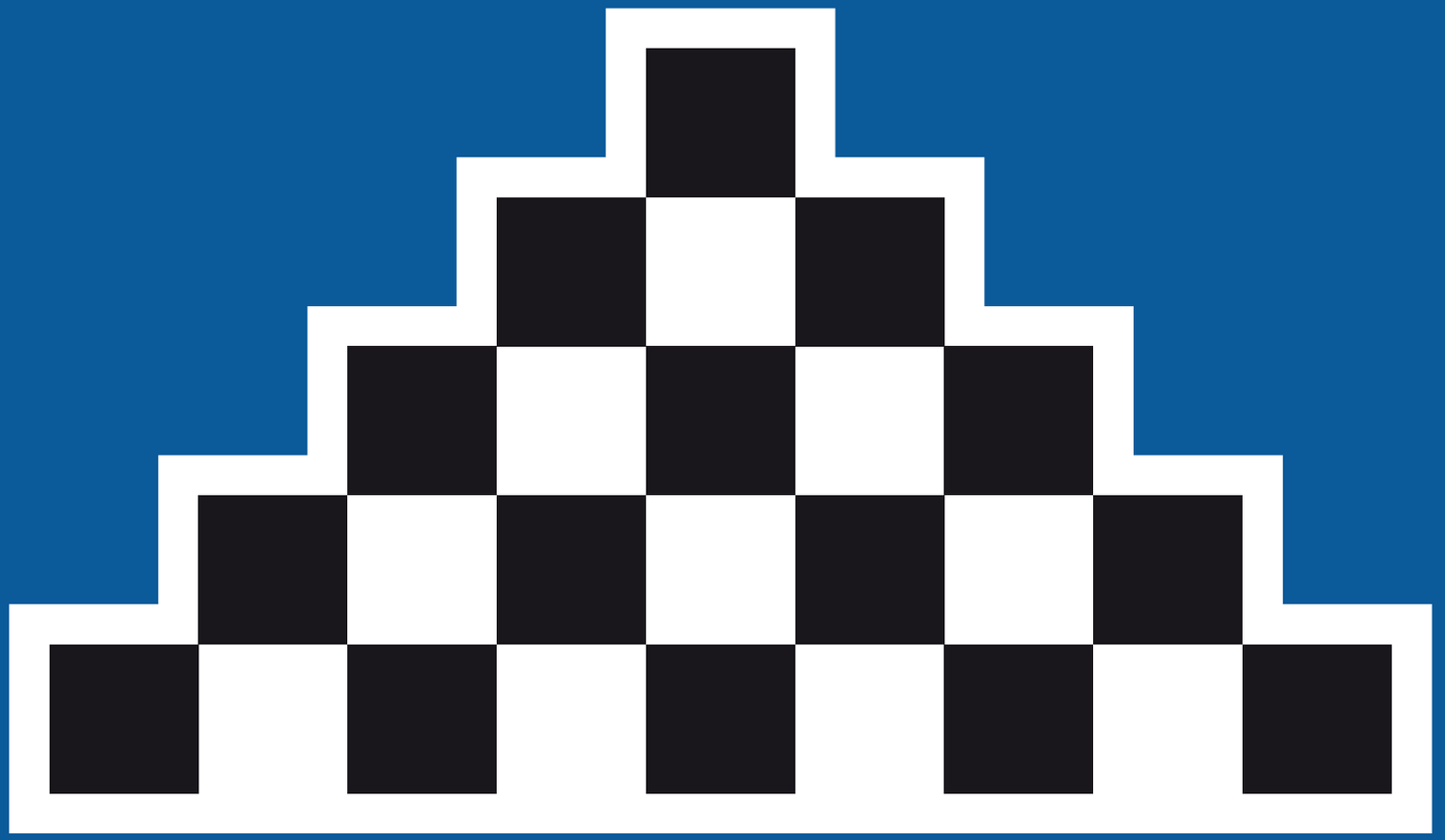
- Warum ist die Anwendung nicht aus normalen JavaBeans aufgebaut:
  - Java-Klasse nach der JavaBean-Konvention
  - eventuell noch eine Interface-Definition
- Abkehr von den EJBs → POJOs (Plain Old Java Objects)
- Keine Abhängigkeiten mehr zur Containertechnologie
  - keine Imports

```
public interface HelloWorld {

  public String sayHello();

}
```

```
public class HelloWorldBean implements HelloWorld {

  public String sayHello() {
    return "Hello, world !";
  }

}
```

Spring Framework

- Die Verantwortlichen des Spring Frameworks haben die Fehler von EJBs analysiert:
  - starke Kopplung an die Containertechnologie
  - viele Dateien, um eine Klasse zu erzeugen
- Spring bietet die gleichen Features wie J2EE
- Spring verwendet aber einen leichtgewichtigen Ansatz:
  - normale JavaBeans
  - Spring-spezifisches wird nur in der Konfigurationsdatei deklariert
  - oder mittels Konfigurationsklassen



Grundlagen

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

freies Applikationsframework für die Java-Plattform

Spring Framework

Module von Spring

Data Access
database / persistence

Web
Web MVC / JSF / JSP / Portlets

AOP
aspect oriented programming

JEE
remoting / messaging

Core
IoC-Context

Der Kern des Spring Frameworks

Inversion of Control

Motivation

- Zur Herleitung verwenden wir folgendes Beispiel:
  - Nachrichten eines Nutzers sollen aus einer Datenbank gelesen werden
- Zwei Klassen sollen dafür verwendet werden:
  - `MessageService` (laden aller Nachrichten)
  - `MessagePrinter` (Anzeigen der Nachrichten eines Nutzers)

```
public class Message {
     private String text;
     private String author;
     ...
}
```

Wie kommt `MessagePrinter` nun an die Implementierung von `MessageService`?

```
public interface MessageService {
  public Message[] findMessages();
}
```

```
public class MessageServiceImpl
        implements MessageService {
  public Message[] findMessages() {
    // Lese alle Nachrichten
    // aus einer Datenbank
  }
}
```

Naive Implementierung

```
public class MessagePrinter {

  public void outputMessages(String author) {
   MessageService messageService = new MessageServiceImpl();

    for (Message message : messageService.findMessages()) {
      if (message.getAuthor().equals(author)) {
        System.out.println(message);
      }
    }
  }
}
```

- `MessageServiceImpl` ist direkt an die Methode gekoppelt
  - alternative Implementierungen ohne Code-Änderungen nicht einsetzbar
  - schwer bis gar nicht testbar, da Datenbankabfrage nicht durch eine TestImplementierung ersetzbar

Dependency Lookup

```java
public class ServiceLocator {
  public Object getService(String serviceName) {
    if ("messageService".equals(serviceName)) {
      // Holen per JNDI
    }
  }
}
```

```java
public class MessagePrinter {
  public void outputMessages(String author) {
    ServiceLocator locator = ServiceLocator.getInstance();
    MessageService messageService =
        (MessageService)locator.getService("messageService");
    ...
```

- Übliche Vorgehensweise bei JEE
- Zusätzliche Klasse ServiceLocater notwendig
- Aufrufende Klasse muss ServiceLocater kennen
- Testbarkeit abhängig von der Implementierung des ServiceLocators

Dependency Injection (DI)

- Hollywood-Prinzip: „Don't call us, we call you“
- Klasse bekommt benötigte Komponenten von außen "injiziert"
- Klasse muss nur wissen was sie braucht, nicht wie
- DI Container garantiert, dass Abhängigkeiten gesetzt werden
- entkoppelt Klasse von der Laufzeitumgebung

```
public class MessagePrinter {

  private MessageService messageService;

  public void outputMessages(String author) {
    for (Message message : messageService.findMessages()) {
      ...
    }
  }

}
```

Wie kommt der `MessageService` in die Klasse?

Injektionsarten

```java
public class MessagePrinter {
  private MessageService messageService;

  public MessagePrinter(
          MessageService messageService) {
    this.messageService = messageService;
  }
}
```

Constructor Injection

+ Klasse immer im gültigen Zustand
- nicht kompatibel zur JavaBeans-Konvention
- zuviele Argumente bei vielen Abhängigkeiten
- für Tests müssen alle Abhängigkeiten gesetzt werden

```java
public class MessagePrinter {
  private MessageService messageService;

  public void setMessageService(
          MessageService messageService) {
    this.messageService = messageService;
  }
}
```

Setter Injection

+ kompatibel zur JavaBeans-Konvention
+ für Tests können einzelne Abhängigkeiten gesetzt werden
- Klasse kann in ungültigem Zustand sein
- mehr Code notwendig

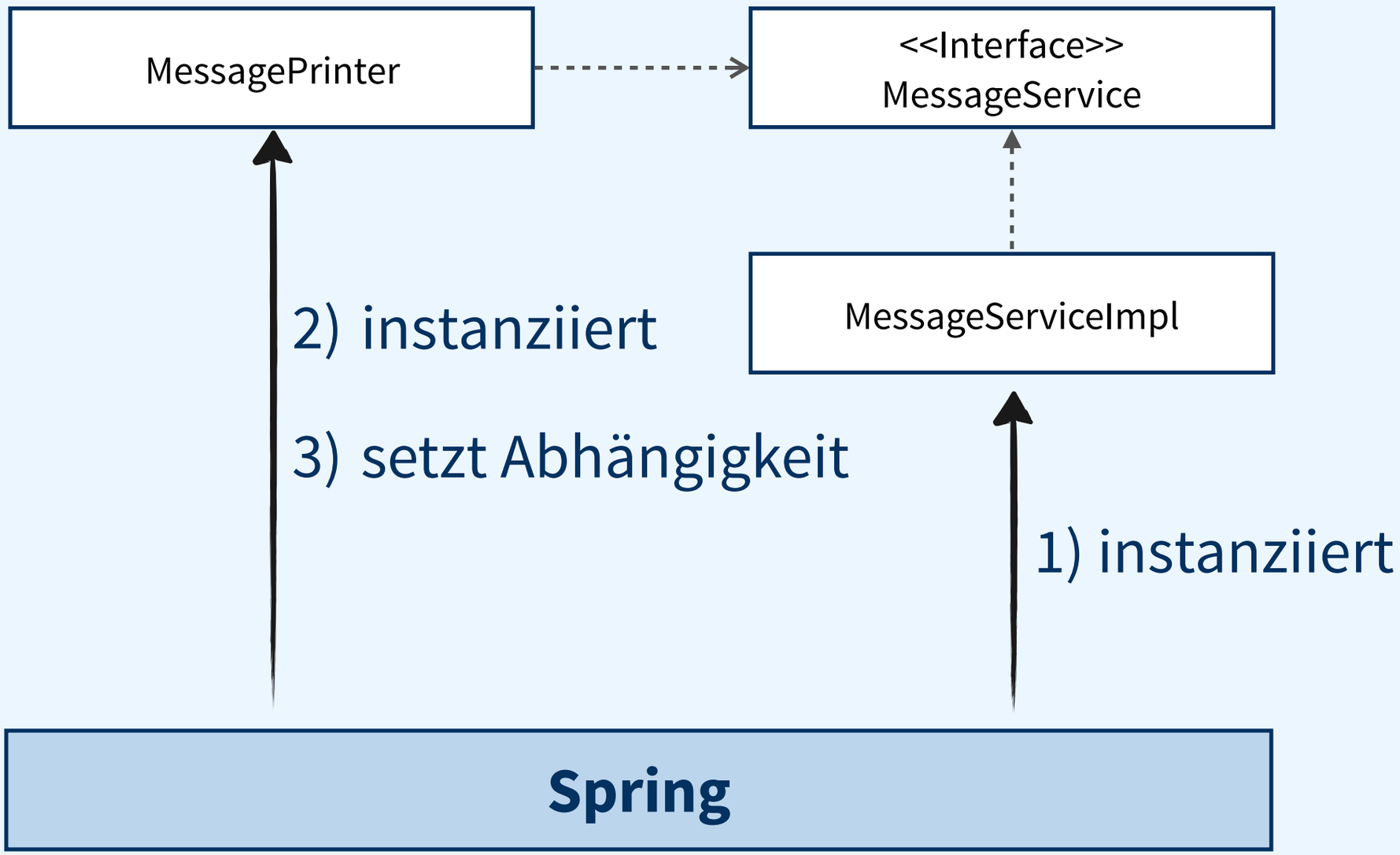
Umsetzung mit Spring

```java
@Component
public class MessagePrinter {

  @Autowired
  MessageService messageService;

  public void setMessageService(
                MessageService messageService) {
    this.messageService = messageService;
  }

  public void outputMessages(String author) {
   for (Message message : messageService.findMessages()) {
      if (message.getAuthor().equals(author)) {
        System.out.println(message);
      }
    }
  }
}
```

```java
@Component
public class MessageServiceImpl
        implements MessageService {
  public Message[] findMessages() {
    // Lese alle Nachrichten
    // aus einer Datenbank
  }
}
```

- Beans werden mit `@Component` annotiert, sodass Spring sie zur Laufzeit findet und instanziert
- Abhängigkeiten werden mittels `@Autowired` automatisch verknüpft

Umsetzung mit Spring



Testbarkeit

- Durch die Entkopplung von MessagePrinter und MessageServiceImpl erhöhen wir die Testbarkeit
  - Testimplementierungen können adhoc erzeugt werden (Mocks)
  - Spring wird an dieser Stelle nicht benötigt

```
public class MockMessageService implements MessageService {
  public Message[] findMessages() {
    return new Message[] {
        new Message("Testnachricht", "Stefan"),
        new Message("kurze Nachricht", "Frank")};
  }
}
```

```
public class MessagePrinterTest {
  public void testOutputMessages() {
    MessagePrinter printer = new MessagePrinter();
    printer.setMessageService(new MockMessageService());
    lister.outputMessages("Frank");
 }
}
```

Spring Context

- Bleibt noch die Frage, wie Spring die Beans instanziiert?
  - als Teil der Web-Anwendung (später in der Vorlesung)
  - kann aber auch programmatisch geschehen:

```java
@SpringBootApplication
public class MessageApp {

  public static void main(String[] args) {
    ApplicationContext ctx = SpringApplication.run(MessageApp.class, args);

    MessagePrinter messagePrinter = ctx.getBean(MessagePrinter.class);
    messagePrinter.outputMessages("Stefan");
  }
}
```

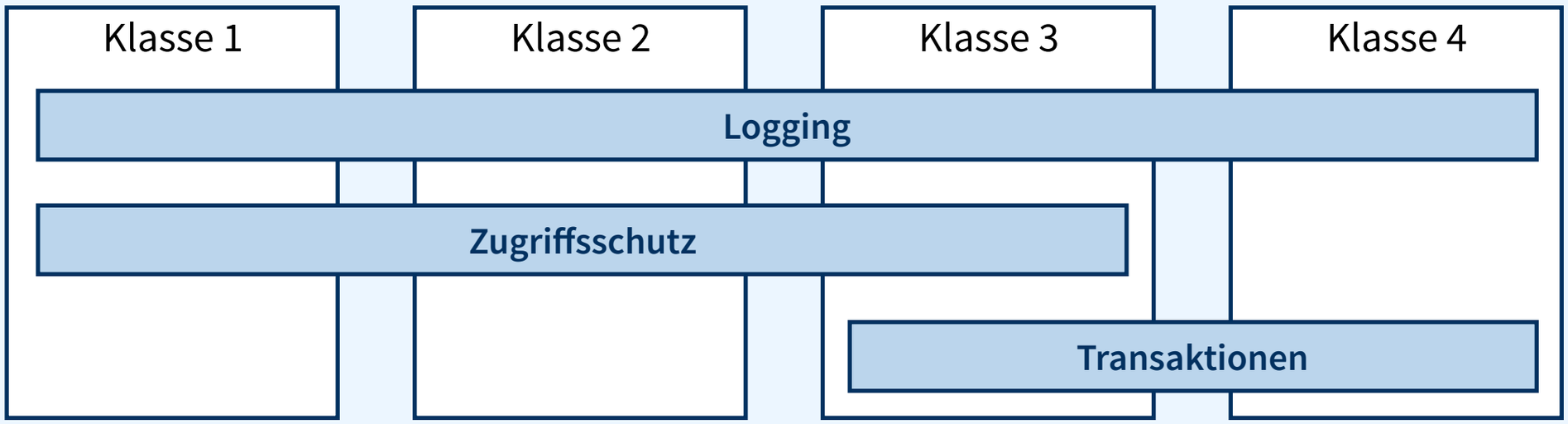
Programmieren mit Aspekten

Spring AOP

AOP Definition

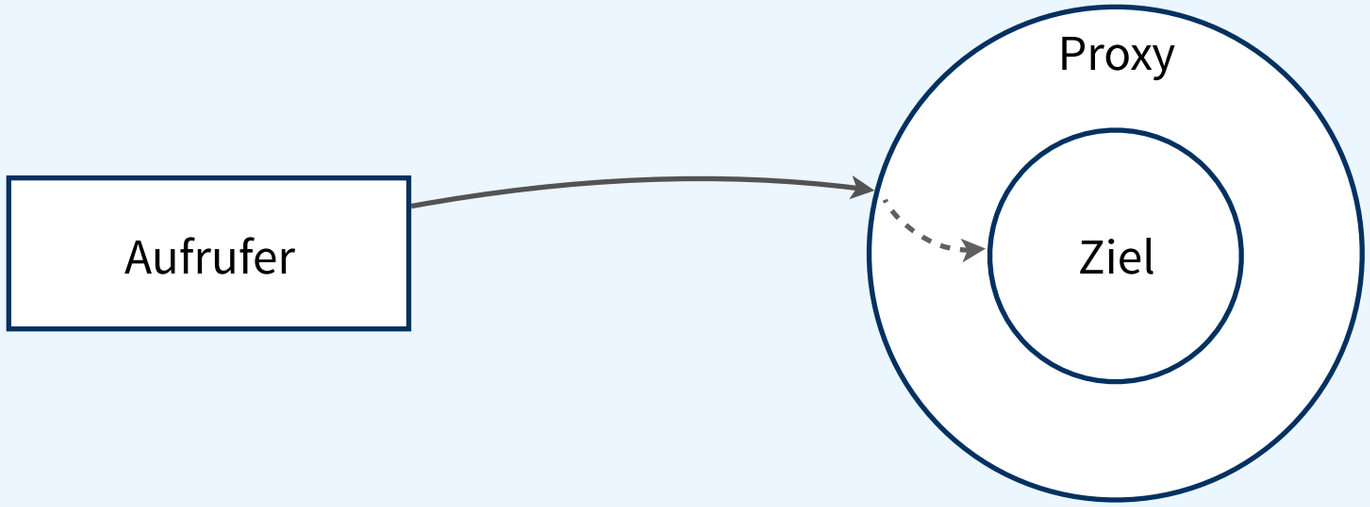
- Aspect Oriented Programming erlaubt es, querschnittliche Funktionen zu implementieren
- querschnittliche Funktionen → Anforderungen an eine Anwendung, die nicht in jeder Methode redundant programmiert werden sollen, z.B.:
  - Logging → der Aufruf einer Methode soll geloggt werden
  - Transaktionen → am Anfang einer Methode soll eine Transaktion eröffnet und am Ende geschlossen oder zurückgerollt werden
  - Sicherheit → eine Methode darf nur von angemeldeten Nutzern aufgerufen werden

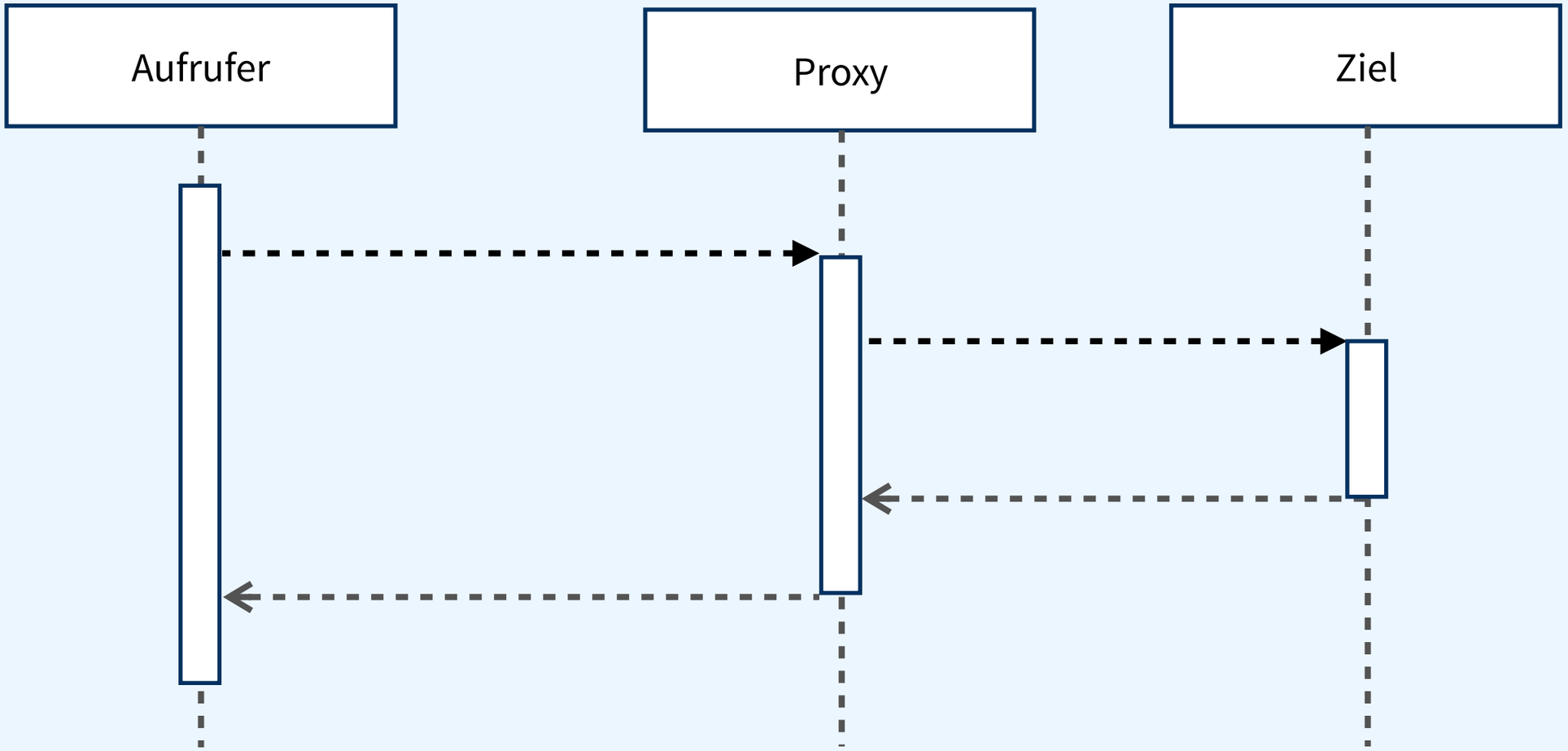
Orthogonale Aspekte



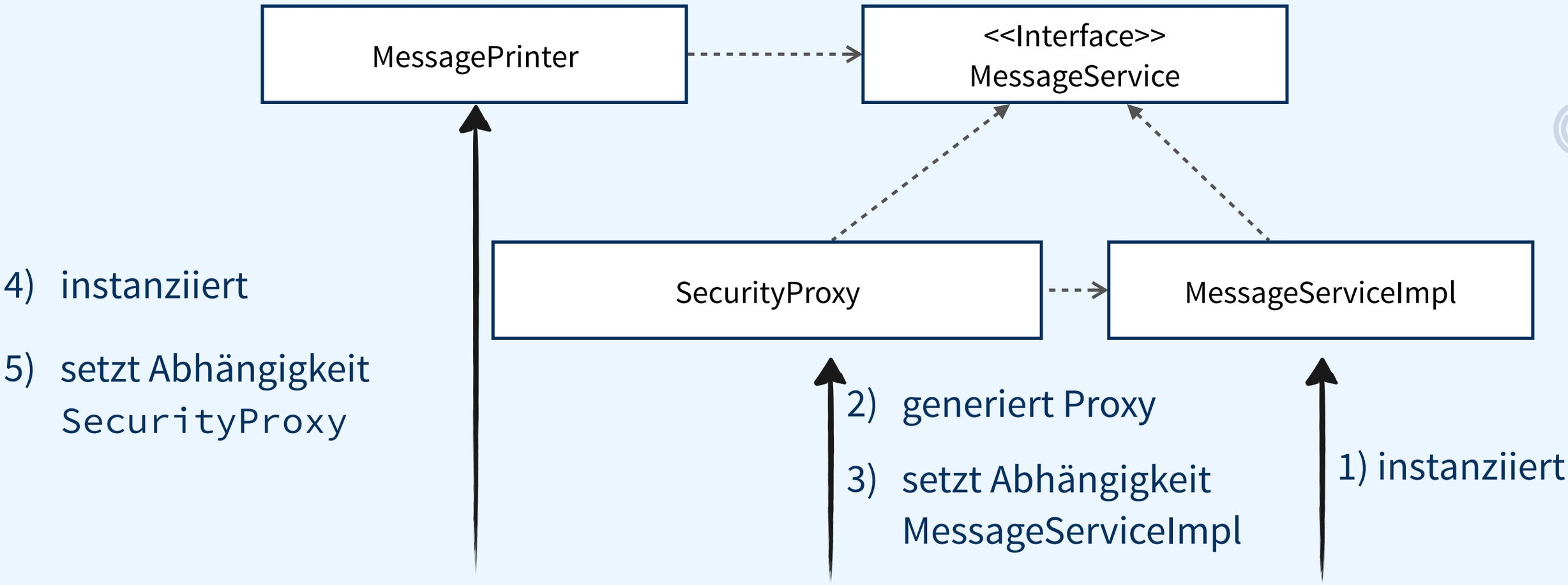
- Die Objektorientierung bietet aber nur zwei (beschränkte) Möglichkeiten, um Aspekte umzusetzen:
  - Vererbung
  - Delegation
- besser: deklarative Definition von Aspekten
  → Konfiguration mit Spring, nicht in den Anwendungsklassen

AOP mittels Proxy Klasse





AOP Beispiel

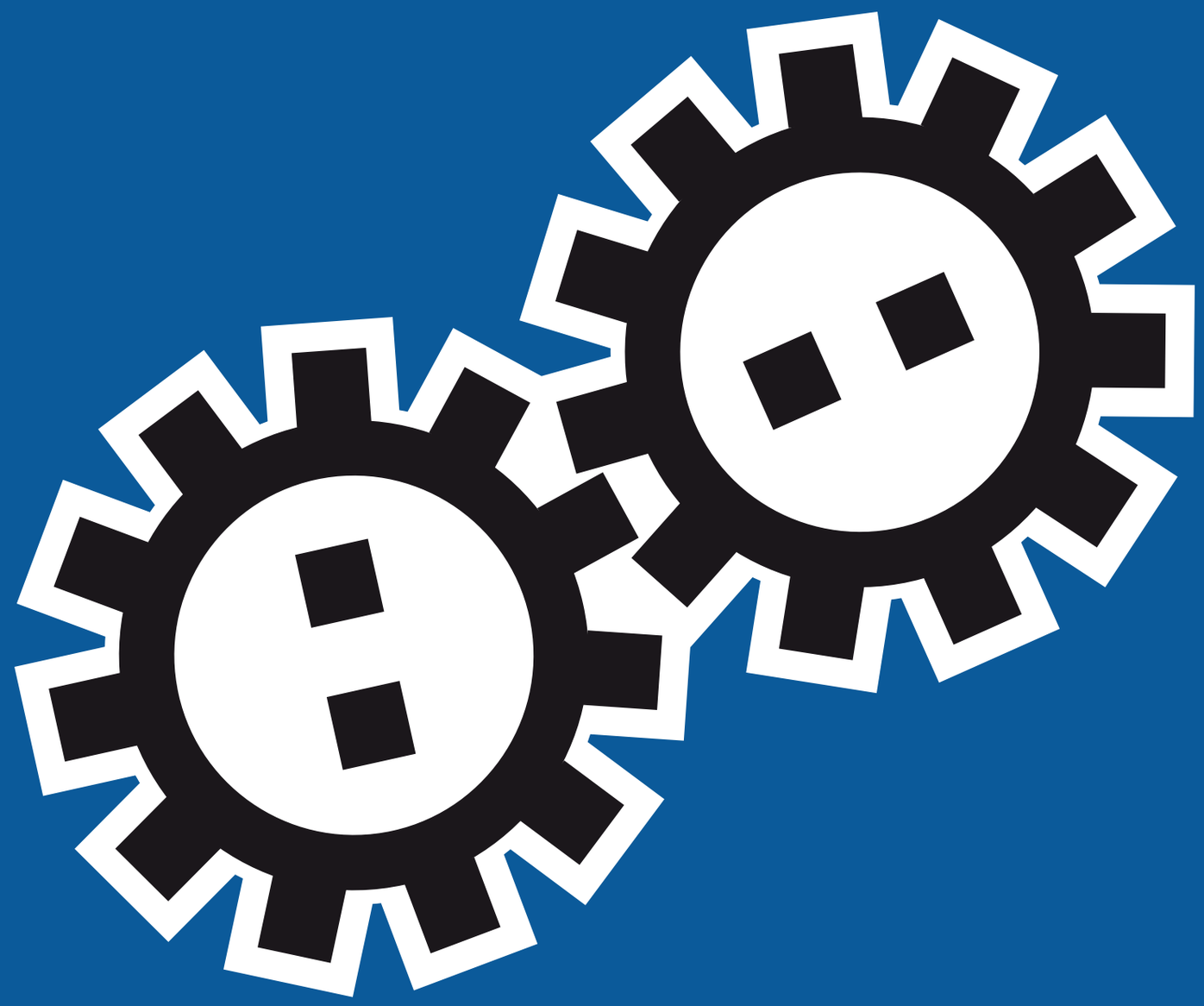


Spring

- `MessagePrinter` verwendet jetzt `SecurityProxy` als Implementierung von `MessageService`
- `SecurityProxy` kann Aufrufe abfangen oder weiter an `MessageServiceImpl` delegieren

AOP Gefahren

- Aspekte sind in den Klassen »unsichtbar« und werden erst zur Laufzeit »eingewebt«
  - mangelhafte Unterstützung in den IDEs
  - evtl. unterschiedliches Laufzeitverhalten
  - Debuggen wird erschwert
- Spring erlaubt deshalb die Definition von Aspekten mittels Java-Annotations
  - Aspekte sind wieder näher am Code
  - sehen wir später bei der Definition des Zugriffschutzes



Techniken

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Konvention für wiederverwendbare Klassen

Exkurs: JavaBeans

JavaBeans

- Wiederverwendbare Softwarekomponenten in Java
- Java-Klassen zur Kapselung von mehreren Objekten in einem Objekt
- JavaBeans müssen sich an Vorgaben halten:
  - private Instanzvariablen (auch Properties oder Eigenschaften genannt)
  - Einen Konstruktor ohne Parameter
  - Zugriff auf Variablen mit öffentlichen Get- und Set-Methoden

Properties

- Die JavaBean ist ein Behälter für andere Objekte (Properties)
- Properties werden als private Instanzvariablen umgesetzt

User.java

```
public class User {

  private String username;
  private String password;
  private String fullname;
  private String email;
  private List<User> followUsers = new ArrayList<User>();
}
```

Konstruktor

- Klasse muss einen Konstruktor ohne Parameter besitzen
  - Einfache Erzeugung der Bean ohne zusätzliche Abhängigkeiten
    `new User();`
  - Kann weitere optionale Konstruktoren besitzen

User.java

```java
public class User {

  private String username;
  private String password;
  private String fullname;
  private String email;
  private List<User> followUsers = new ArrayList<User>();

  public User() {
  }

  public User(String username, String password) {
    this.username = username;
    this.password = password;
  }
}
```

Getter und Setter

- Properties werden durch Getter-Methoden (oder Accessors) zum Lesen bereitgestellt
- Properties werden durch Setter-Methoden (oder Mutators) modifiziert
- Die Konvention zur Benennung der Methoden erlaubt einfachen Zugriff auf die Properties, z.B. aus Frameworks wie JSP oder Thymeleaf

Namenskonvention für eine Eigenschaft mit dem Namen "propertyName":

```
private <type> propertyName;

public <type> getPropertyName() {
  return propertyName;
}

public void setPropertyName(<type> propertyName) {
  this. propertyName = propertyName;
}
```

Getter und Setter

Beispiel für eine User-JavaBean (Auszug):

User.java

```
public class User {
  private String username;
  private String password;
  …

  public String getUsername() {
    return username;
  }

  public void setUsername(final String username) {
    this.username = username;
  }

  public String getPassword() {
    return password;
  }

  public void setPassword(final String password) {
    this.password = password;
  }
  …
}
```

Verarbeitungsmethoden

- Weitere Methoden (außer `get...` und `set...`) können hinzugefügt werden
- Für eine User-JavaBean könnten das z.B. sein:
  - `addFollower(User otherUser);`
  - `validateEmail();`
- Werden hier nicht weiter spezifiziert, regulärer Java-Code

verteilte Anwendungen mit Spring

Spring Remoting

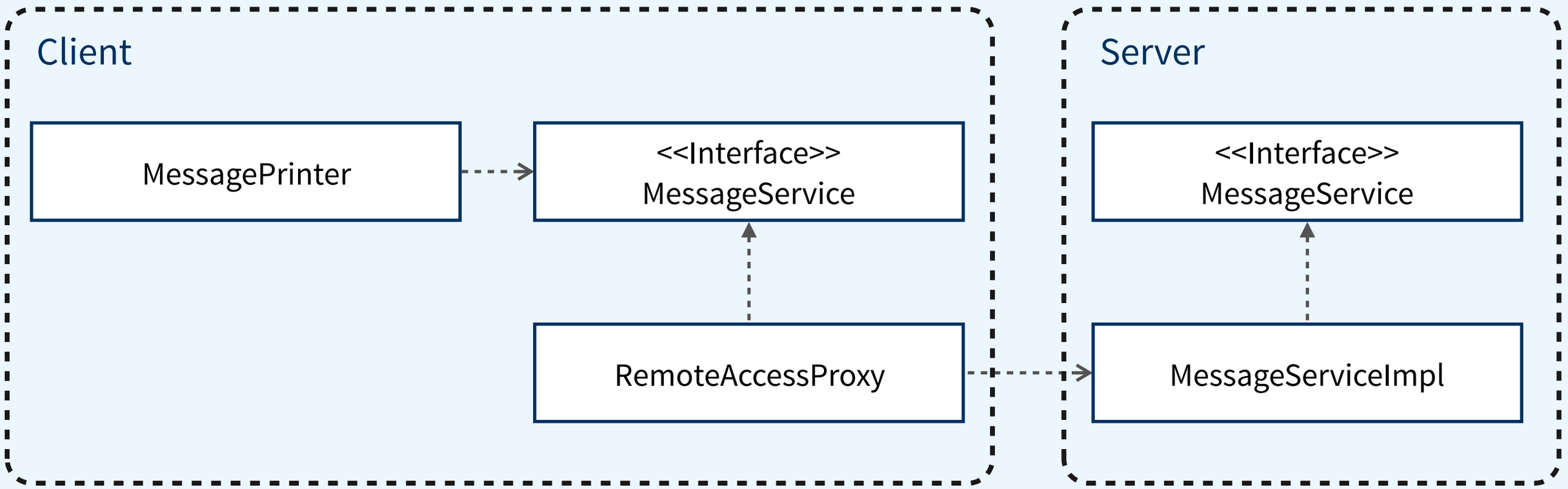
Spring Remoting

- Mittels Dependency Injection bekommen Beans Implementierungen der Abhängigkeiten
  - Spring kann zur Laufzeit Proxies erstellen (z.B. Security, Logging)
- Spring bietet aber auch Proxies, um auf entfernte Implementierungen transparent zuzugreifen
- Unterstützte Technologien:
  - RMI (Java Remote Method Invocation)
  - RMI over http
  - Web-Services
  - REST (Representational State Transfer)

Spring Remoting

- `MessagePrinter` verwendet nur das Interface MessageService
- Spring erzeugt zur Laufzeit einen Proxy, der …
  - das Interface `MessageService` implementiert
  - die Methoden transparent in Netzwerkaufrufe umsetzt

➡ `MessagePrinter` ruft `MessageServiceImpl` scheinbar lokal auf



Web-Anwendungen automatisiert bauen

Maven Build Tool

Automation

- Erstellung von Web-Anwendungen → wiederholte Ausführung von Aufgaben
  - z.B.: Erstellung einer ZIP-Datei
- manuelle Ausführung hat viele Nachteile:
  - unangenehme Aufgabe
  - fehlerträchtig
  - nur eingeschränkt reproduzierbar
  - schwierig bei Team-Arbeit
- Software-Qualität leidet!

Automation: Ziele

- Der Erstellungsprozess sollte unabhängig von der IDE sein
- Der Erstellungsprozess sollte in die IDE integriert sein
- Dokumentationserstellung sollte Teil der Automation sein:
  - Projektdokumentation
  - Dokumentation der Compiler-Fehler und -Warnungen
  - Test-Reports
- Der Erstellungsprozess sollte reproduzierbar auf allen Entwicklungsmaschinen gleich sein

Automation: Aufgaben

Aufgabe	Beschreibung
Validation	Überprüfung, ob Quelldokumente korrekt sind, z.B. für XML-Dokumente
Source Generation	Erzeugung von Quelldateien aus Metadaten, z.B. aus Annotationen
Compilation	Kompilierung der Java-Quelldateien
Test Execution	Ausführung der Unit-Tests
Report Generation	Erzeugung von Reports, z.B. die Ergebnisse der Test oder die Testabdeckung
Documentation Generation	Erzeugung von Dokumentation, z.B. von JavaDocs oder der Benutzerdokumentation
Packaging	Erstellung eines Programmpakets, z.B. eines WAR für eine Web-Anwendung

Automation: Werkzeuge

- Automation existiert für viele Programmiersprachen und Plattformen
- Umfang und Konzept variieren aber erheblich
  - Unix make (für C/C++ Projekte)
  - Apache Ant für Java
  - NAnt für .net
  - Apache Maven
  - Rake für Ruby
- Shellskripte und Batchdateien sind auch verbreitet
  - sind aber meist nicht plattformunabhängig

Was ist Maven?

- Maven bietet einen vordefinierten Automationszyklus
  - siehe Tabelle Automation: Aufgaben
- Durch die Verwendung einer definierten Verzeichnisstruktur wird kaum Konfigurationsaufwand benötigt
  - convention over configuration
- Viele Plugins verfügbar, die den Automationszyklus erweitern
  - embedded web container
- Maven verwaltet auch die Abhängigkeiten von Projekten
  - transitive dependency management

Maven Verzeichnisstruktur

```
projekt/
+-src
| +-main
| | +-java
| | | +-klasse.java
| | +-resources
| | | +-konfiguration.xml
| +-test
|   +-java
|     +-testklasse.java
+-target
| +-classes
| +-site
| +-...
+-pom.xml
```

- src/ → Alle Quelldateien
  - src/main/java/ → Java Quellcode
  - src/main/resources/ → Konfiguration
- target/ → Alle generierten Dateien
  - target/classes/ → kompilierte Klassen
  - target/site/ → Dokumentation
  - target/projekt.war → Web-Anwendung
- pom.xml → Maven Konfigurationsdatei

pom.xml

- Trotz Konventionen kann (und muss) in Maven vieles konfiguriert werden
  - Projektname (groupId/artifactId/version)
  - zu erzeugende Dokumentationen
  - weitere Automationsplugins
  - Abhängigkeiten
- Komplette Übersicht unter: http://maven.apache.org/pom.html
- weitere Infos im Praktikum
- minimales Beispiel →

```xml
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0</version>

  <packaging>war</packaging>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
    </dependency>
  </dependencies>

</project>
```

Maven verwenden

- Java-Programm von http://maven.apache.org/ herunterladen
  - mvn compile (Java Quellen von src/main/java nach target/classes kompilieren)
  - mvn package (WAR-Paket aus kompilierten Klassen und Web-Dateien erstellen)
  - mvn jetty:run (Paket im integrierten Web-Container ausführen)
- …oder als Plugin in Eclipse installieren (m2e)
  - im Praktikum

Zusammenfassung

Spring Motivation

- Klassische Entwicklung von Enterprise-Anwendungen ist komplex
- Spring vereinfacht Entwicklung durch Rückführung auf Java-Klassen ohne Abhängigkeiten zu fremden Bibliotheken
  - POJO (Plain Old Java Objects) bzw. JavaBeans
- Spring wird nur in der Konfigurationsklasse (`MessageApp`) verwendet
  - die eigentlichen Anwendungsklassen verwenden nur Annotationen, sonst keine Abhängigkeiten/Code von Spring

Dependency Injection

- Spring ist für Verdrahtung und Konfiguration von Komponenten (Beans) zuständig
- Vorteile:
  - lose Kopplung zwischen Komponenten
  - kein Code für das Nachschlagen in den Komponenten
  - Rekonfiguration ohne Änderungen des Codes möglich
  - bessere Testbarkeit durch Testimplementierungen (Mocks)

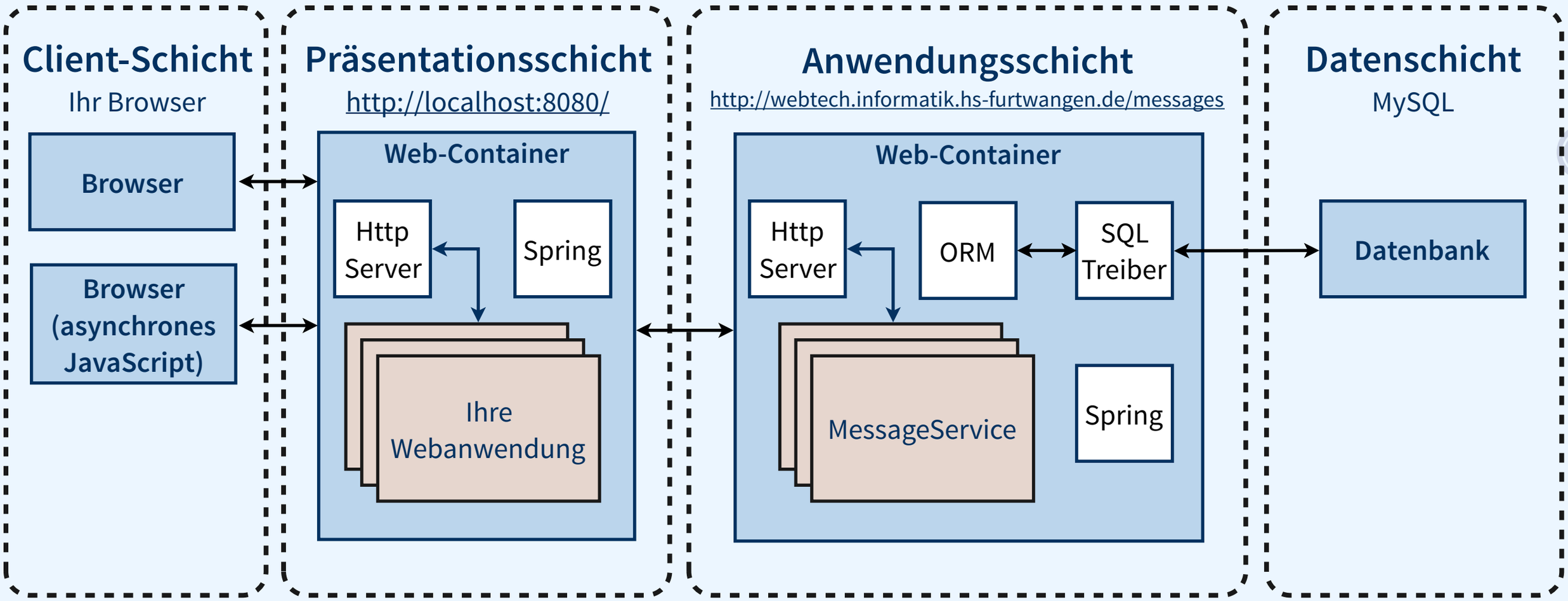
Spring Framework

- Basierend auf der Dependency Injection und den Proxy-Objekten bietet Spring viele Funktionen, die konfiguriert werden können:
  - Logging
  - Transaktionen
  - Persistenz
  - Sicherheit
  - Remoting
  - Spring Web MVC (später)

Architektur der Web-Anwendung in dieser Veranstaltung

Übersicht Architektur

Architektur



- Anwendungsschicht und Datenschicht stehen zentral bereit
- Alle Studierenden arbeiten auf denselben Daten
- Zugriff auf die Anwendungsschicht über das Interface `MessageService` und Spring Remoting

MessageService

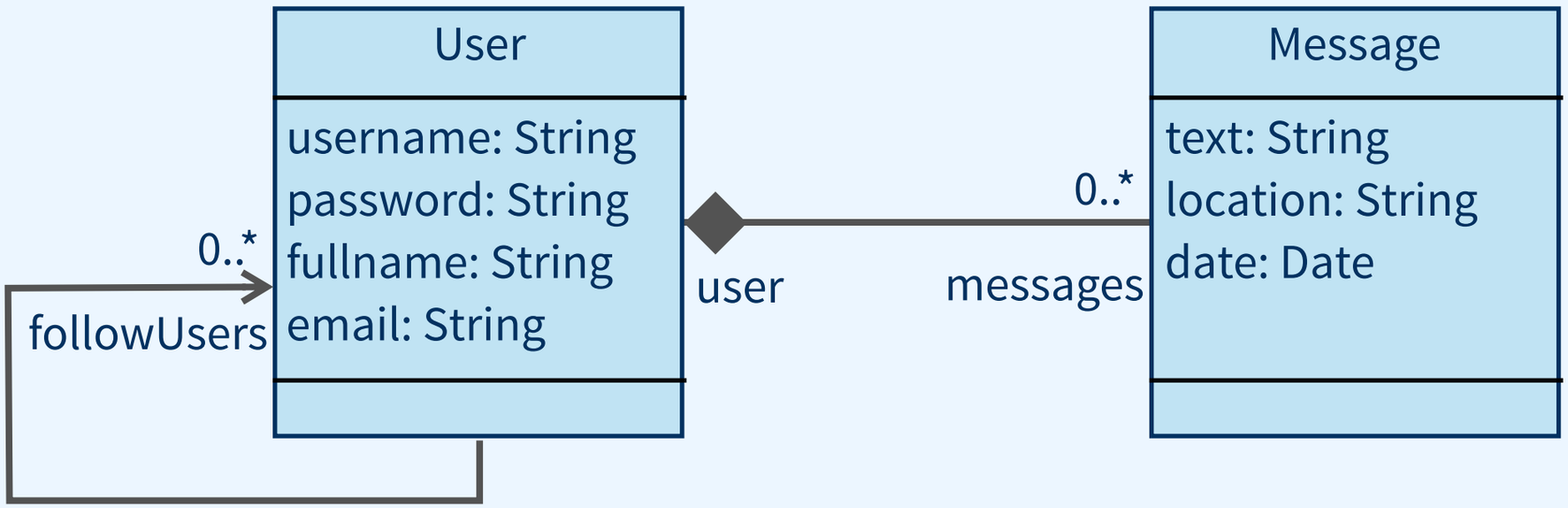
<<Interface>>
MessageService

findAllMessages(): List<Message>
findAllMessages(user: User): List<Message>
findLatestMessages(date: Date): List<Message>
findLatestMessages(user: User, date: Date): List<Message>
findMessageById(id: int): Message
saveMessage(message: Message)
deleteMessage(message: Message)

findAllUsers(): List<User>
findUserById(id: int): User
findUserByUsername(username: String): User
createUser(user: User)
saveUser(user: User)
deleteUser(user: User)

- Finder-Methoden (find…) sind öffentlich
- Schreibender Zugriff erst nach Anmeldung (spätere Vorlesung)
  - Zugriff würde jetzt noch fehlschlagen

Datenobjekte



- `User` und `Message` sind Datenobjekte, die vom Service zurückgegeben werden
- Sie können sich alle Klassen und Interfaces im Archiv "messages-model.jar" unter Maven-Dependencies anschauen!

Danke