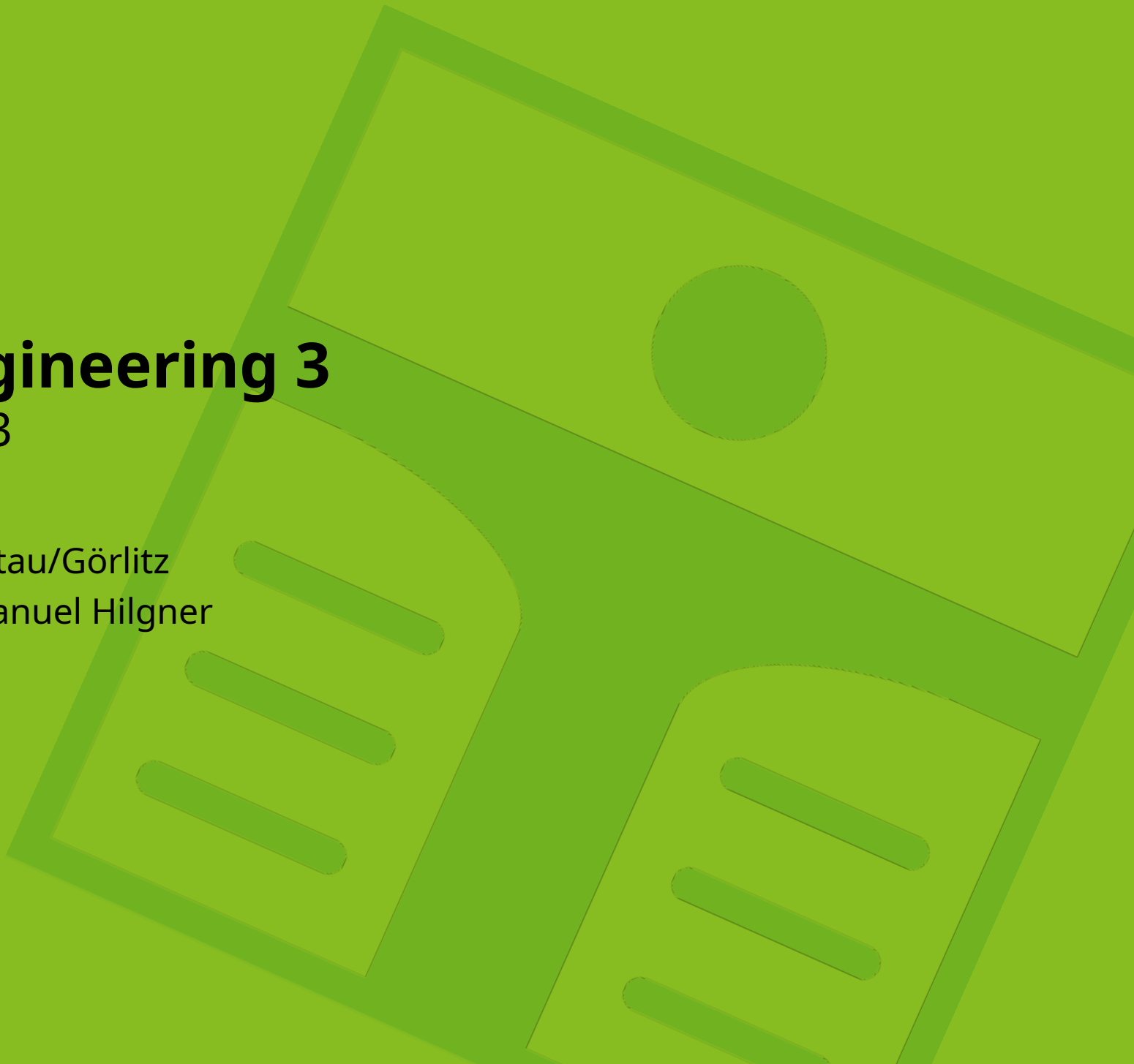
Web Engineering 3

Vorlesung 3

Hochschule Zittau/Görlitz
Christopher-Manuel Hilgner



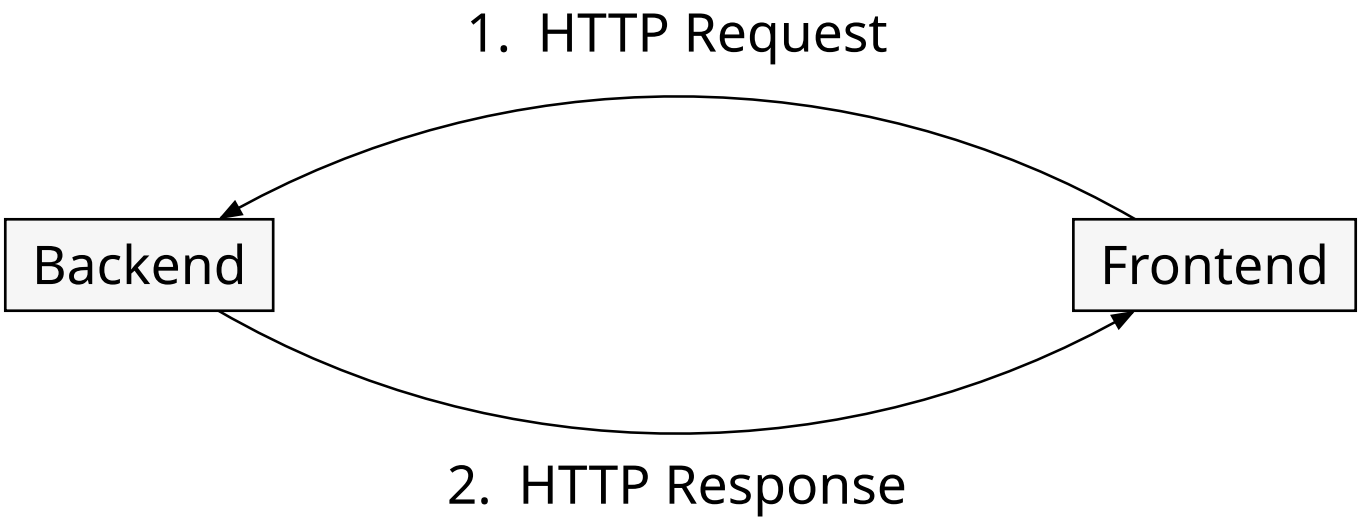
Agenda

REST Mappings

- HTTP Methoden

Wie & Warum funktioniert Spring eigentlich?

- Inversion of Control
- Dependency Injection
- Spring Beans
- Singleton- & Prototype-Pattern


1. HTTP Request
Backend
Frontend
2. HTTP Response

HTTP Request

Bestandteile

- Ein **HTTP Verb**, dass die Art der Operation definiert
- Einen **Header**, der Informationen über die Request enthält
- Einen **Pfad** zu einer Ressource
- Einen optionalen **Body**, der weitere Daten enthält

HTTP Request

Header

- `accept` Feld: Welche Ressource nimmt der Client an
- Art der Ressource über MIME Types

MIME Type:

```
1    type/subtype;parameter=value
```

- `parameter` ist optional
- **Beispiele**: `image/png`, `audio/wav`, `application/json`

HTTP Request

Pfad

- Definiert, auf welcher Ressource die Operation ausgeführt werden soll
- Erster Teil sollte die Pluralform der Ressource sein

Beispiel: `store.com/customers/223/orders/12`

Lesbarkeit der Pfade — Richtlinie 1

Gute Lesbarkeit der Pfade in der API, auch wenn man nicht mit der API vertraut ist

HTTP Response

- Datentyp angeben wenn Daten zurückgegeben werden sollen
- Content Type im Header wie bei Request
- Status Code anhängen für Informationen über Ausgang der Request

```kotlin
@GetMapping
fun getEntities() {}

@PostMapping
fun createEntity() {}

@PutMapping
fun updateEntity() {}

@DeleteMapping
fun deleteEntity() {}
```

HTTP Methoden

- HTTP definiert Verben, die Ziele von Request genauer beschreiben

Häufige Verben:

GET

POST

PUT

DELETE

Weitere Verben:

HEAD

CONNECT

OPTIONS

TRACE

PATCH

HTTP Methoden

GET

- Stellt Anfrage an Server eine Ressource zu transferieren
- Sollten immer gleiche Ergebnisse bei gleichen Parametern erziehlen
- Content sollte nie mit einer GET Request erstellt werden
- Caching ist möglich

Informationen in der UI Hinweis 2

Es ist zu beachten, dass wenn Ressourcen nur über URIs angefragt werden, potentiell sicherheitskritische Informationen in dieser URI landen können. Wenn es nicht möglich ist, diese Informationen in weniger kritische zu transformieren wird das Nutzen einer POST Request mit den Daten im Request Content empfohlen.

HTTP Methoden

POST

- Wird genutzt um transferierte Daten nach Server Sezifikation zu verarbeiten
- Beispiel:
  - Daten in Inputfeldern übergeben
  - Nachrichten Posten (Foren, Social Media usw.)
  - Erstellen von neuen Ressourcen
  - Daten an vorhandene Ressourcen anhängen
- Server kommuniziert mit Status Codes das Ergebnis der POST Request
- Bei Erfolg: 206 (Partial Content)
- Bei erfolgreichem Erstellen einer neuen Ressource: 201 mit Pfad zur neuen Ressource

HTTP Methoden

PUT

- Editieren von vorhandenen Ressource oder Erstellung von neuen Ressourcen
- Anfrage basiert auf mitgeschickten Daten
- Wenn Ressource nicht vorhanden ist wird sie neu erstellt
- Status Code 201 nach Erstellen neuer Ressource
- Wenn kein neuer Eintrag erstellt wurde: Status Code 200 oder 204
- Server sollte Daten in der PUT Request validieren
- Wenn Fehler in den Daten: Selbst Versuchen Daten in passendes Format zu bringen oder 409/415 zurückgeben

HTTP Methoden

DELETE

- Request an den Server Ressource zu löschen
- Entweder Entfernen von Referenzen oder komplettes Löschen der Ressource
- DELETE sollte nur bei Ressourcen erlaubt sein, die definierte Löschvorgänge besitzen
- Bei Erfolg einer der folgenden Codes:
  - **202 (Accepted)** wenn das Löschen wahrscheinlich erfolgreich sein wird, aber noch nicht durchgeführt wurde
  - **204 (No Content)** Löschen wurde ausgeführt und keine weiteren Informationen sind nötig
  - **200 (OK)** Löschen war erfolgreich und die Response enthält noch Informationen über den aktuellen Status

HTTP Methoden

idempotente Methoden

- Mutiple Ausführung hat den gleichen Effekt auf dem Server
- Wichtig bei automatischen Anfragen (z.B.: Wiederholung bei Fehlschlag)
- PUT und DELETE sind automatisch idempotent
- *safe request methods* sind idempotent
- Server kann trotzdem Nebeneffekte einfügen (z.B.: Logs)
- Nebeneffekte dürfen Ergebnis nicht verändern
- Nicht idempotente Methoden sollten nicht automatisch wiederholt werden (Außer Implementation ist idempotent)

HTTP Status Codes

- Status Code gehört zu jeder HTTP Response
- Zwischen 100 und 599
- Erste Ziffer gibt Klasse der Response an
  - **1xx (Informational)**: Die Request wurde erhalten und wird verarbeitet
  - **2xx (Successful)**: Die Request wurde erfolgreich erhalten, verstanden und akzeptiert
  - **3xx (Rediraction)**: Es müssen weitere Schritte durchgeführt werden, damit die Request verarbeitet werden kann
  - **4xx (Client Error)**: Die Request enthält falschen Syntax oder kann nicht erfüllt werden
  - **5xx (Server Error)**: Der Server konnte eine eigentlich valide Request nicht erfüllen

HTTP Status Codes

Status Code	Bedeutung
200 (OK)	Standard Antwort Erfolg
201 (CREATED)	Standard Antwort für neue Ressource
204 (NO CONTENT)	Standard Antwort für Erfolg ohne Daten im Response Body
400 (BAD REQUEST)	Die Request konnte nicht verarbeitet werden
403 (FORBIDDEN)	Der Client hat keine Rechte auf diese Ressource zuzugreifen
404 (NOT FOUND)	Die Gewünschte Ressource konnte nicht gefunden werden
500 (INTERNAL SERVER ERROR)	Die generische Antwort für einen unerwarteten Fehler

Inversion of Conrol

- Kernpunkt vieler Frameworks, die das Hinzufügen und Erweitern von Funktionalitäten erlauben
- Framework erhält Kontrolle über alles, was der Nutzer geschrieben hat
- Folgendes Prinzip aus Hollywood: **“Don’t call us, we’ll call you”**

Inversion of Control

- Framework ruft vom Nutzer geschriebene Funktionen auf
- Framework stellt Hauptprogramm dar
- Koodiniert Aktivitäten der Anwendung
- Erweiterbares Skelett für eine Anwendung
- Nutzer können generische Algorithmen auf spezifische Anwendungsfälle zuschneiden

Dependency Injection

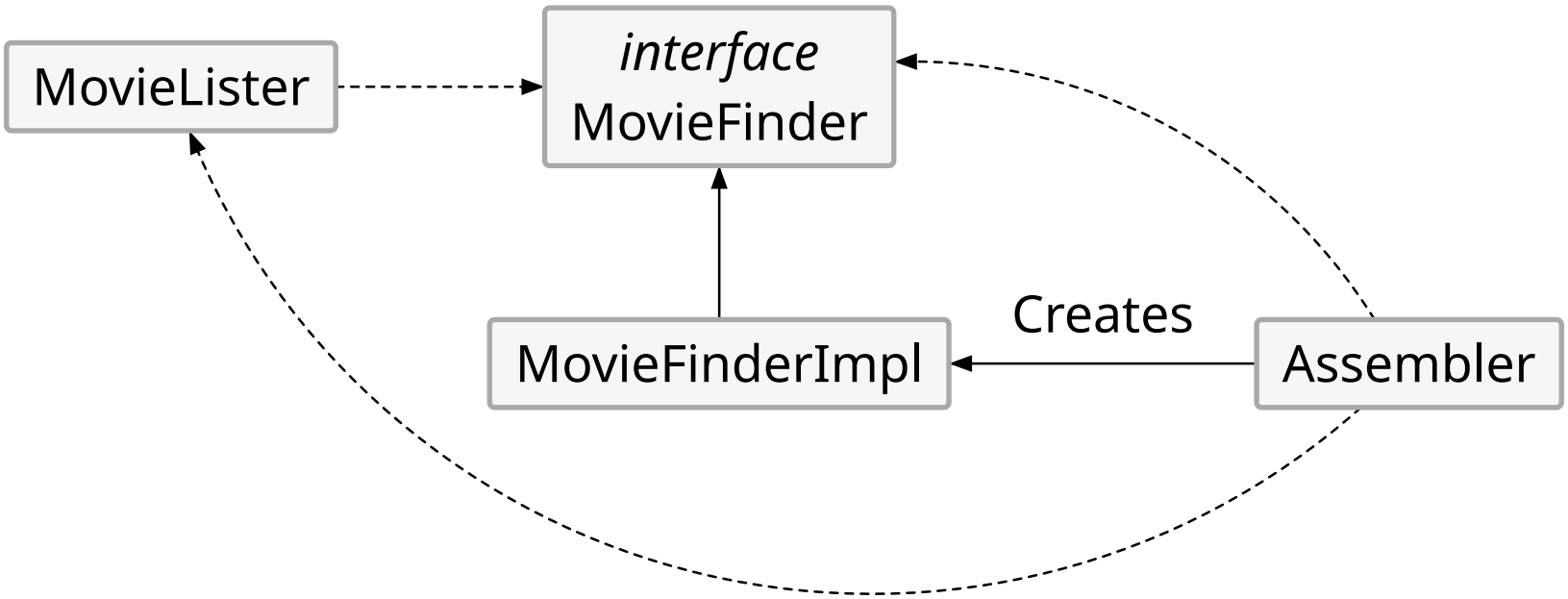
- **Ziel**: Entkoppelung von Abhängigkeiten
- Entkoppelung macht Testen einfacher und erhöht Lesbarkeit
- Klassen müssen nur noch ihre Abhängigkeiten definieren
- Abhängigkeiten werden durch Container bereitgestellt

Dependency Injection

- Assembler Objekte bevölkern Felder in Klassen nach definierten Anforderungen
- Kann Implementation von einem Interface sein
- Drei Möglichkeiten der Dependency Injection

1. Contructor Injection bzw. Type 3 IoC
2. Setter Injection bzw. Type 2 IoC
3. Interface Injection bzw. Type 1 IoC

Beispiel von Dependency Injection



- `MovieLister` Klasse benötigt Implementation von `MovieFinder`
- Assembler stellt Implementation bereit und erfüllt Abhängigkeit

IoC Container

- Der IoC Container wird durch `ApplicationContext` representiert
- Instantiierung, Konfiguration und Assembling von Beans
- Intruktionen dafür werden durch Konfigurations-Metadaten übergeben
- Wege der Konfiguration:
  - Annotations
  - Konfigurations-Klasse mit Factory Methoden
  - XML-Dateien
  - Groovy Scripts

IoC Container

- Manuelle Erstellung ist nicht von Nöten
- Spring kombiniert die erstellten Klassen mit Konfigurations-Metadaten
- Nach der Initialisierung von `ApplicationContext` steht ein ausführbares, konfiguriertes System bereit

Dependency Injection in Spring

- Abhängigkeiten werden in Contructor, Factory Methoden oder Properties definiert
- In Spring: Contructor oder Setter Methoden
- Container übergibt beim Erstellen einer Bean alle Abhängigkeiten
- Bean hat keine Kontrolle über Erstellung oder Ort ihrer Abhängigkeiten

Contructor oder Setter DI

Richtlinie 3

Der Contructor sollte verpflichtende Abhängigkeiten enthalten.

Setter Methoden eignen sich gut für optionale Abhängigkeiten. `@Autowired` kann bei Settern genutzt werden, damit die Property eine verpflichtende Abhängigkeit wird. Der Constructor sollte da aber bevorzugt werden.

Contructor Injection

- Container ruft Contructor auf mit so vielen Argumenten wie Abhängigkeiten
- Jedes Argument ist eine Abhängigkeit

```kotlin
class ExampleClass(private val dependency: Dependency)
{

}
```

Setter Injection

- Leerer Contructor wird aufgerufen
- Container ruft Setter Methode in erstellten Beans auf

```kotlin
class ExampleClass {
  lateinit var dependency: Dependency
}
```


me and the boys at 2am
looking for
BEANS

Beans

Beans Definition 4

Jedes Objekt, welches Teil der Anwendung ist und von dem Spring IoC Container verwaltet wird, ist eine Bean. Eine Bean kann instantiated, assembled oder anderweitig von dem Spring IoC container gemanaged werden.

Bean Annotationen

- `@Component`: Eine generelle Angabe, die eine Klasse als Spring Bean markiert
- `@Service`: Eine Klasse, die einen Service darstellt
- `@Repository`: Eine Klasse, die ein Repository darstellt, welches mit der Persistence-Layer interagiert
- `@Controller`: Eine Klasse, die einen Controller, im Spring Model-Veiew-Controller darstellt

Bean Scopes

- Singleton
- Prototype
- Request
- Session
- Application
- WebSocket

Singleton Pattern

Übersicht

- Zeil: Eine Klasse besitzt nur eine Instanz
- Instanz soll immer genutzt werden, wenn Zugriff auf Singleton Klasse gebraucht wird
- Theoretisch reicht globale Variable
- Stoppt nicht Erstellung von mehreren Instanzen

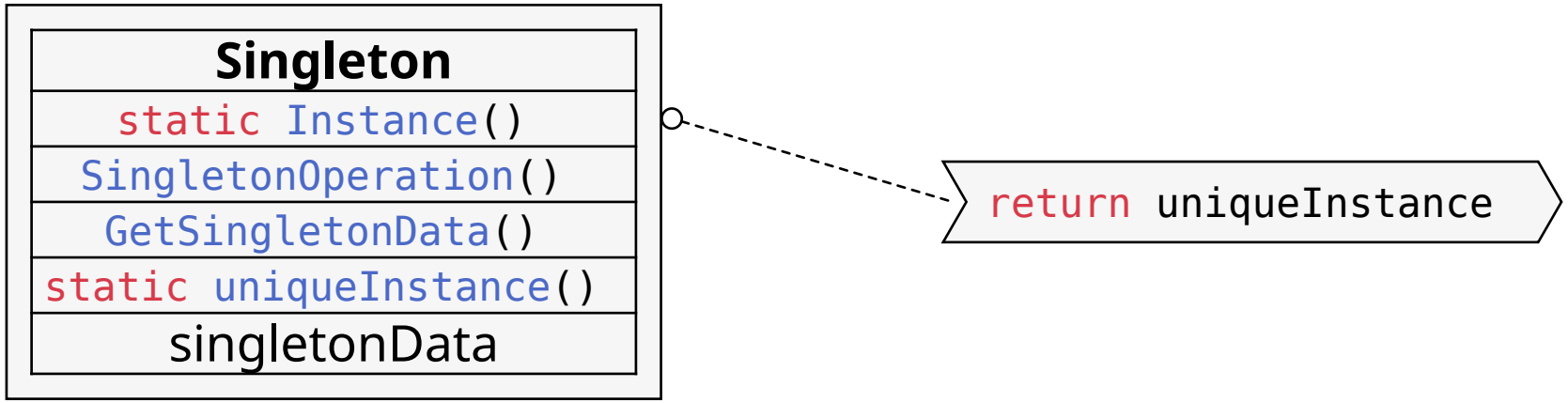
Singelton Pattern

Übersicht

- Singleton-Klasse muss selbst ihre einzige Instanz verwalten
- Erstellung läuft über Klasse und Erstellung mehrerer Instanzen wird blockiert
- Singleton-Klasse stellt Zugang zur Instanz bereit
- Zugriff sollte nur über die `Instance` Methode möglich sein
- `Instance` erstellt Instanz, wenn noch keine vorhanden ist

Singleton Pattern

Struktur



Singleton Pattern

Wann sollte es genutzt werden?

- Wenn nur eine Instanz einer Klasse existieren soll
- Instanz sollte von einem definierten Punkt erreichbar sein
- Einzelne Klasse sollte durch Subklassen erweiterbar sein
- Client sollte erweiterte Instanz nutzen können ohne Code zu modifizieren

Singleton Pattern - Vorteile

1. **Kontrollierter Zugriff auf eine einzige Instanz**: Da die Singleton-Klasse ihre eigene Instanz verwaltet, kann sie genau festlegen, wie auf sie zugegriffen wird und wer auf sie zugreifen kann.
2. **Reduzierter Namespace**: Verbesserung zu globalen Variablen. Der Namespace wird dabei nicht mit globalen Variablen unnötig belegt.
3. **Erlaubt das Anpassen von Operationen und Repräsentation**: Es erlaubt einfaches Wechseln von erlaubten Klasseninstanzen. Die gleiche Logik, die nur eine Instanz erlaubt, kann auf eine beliebige Anzahl erweitert werden. Nur die Zugriffsfunktion muss sich ändern.
4. **Höhere Flexibilität als Klassenoperationen**: Statische Klassenoperationen können die gleiche Funktionalität wie ein Singleton enthalten. Allerdings erlaubt diese Methodik nur schwer das Erstellen von mehreren Instanzen. Außerdem sind diese Operationen nicht `virtual`, also können sie nicht überschrieben werden.

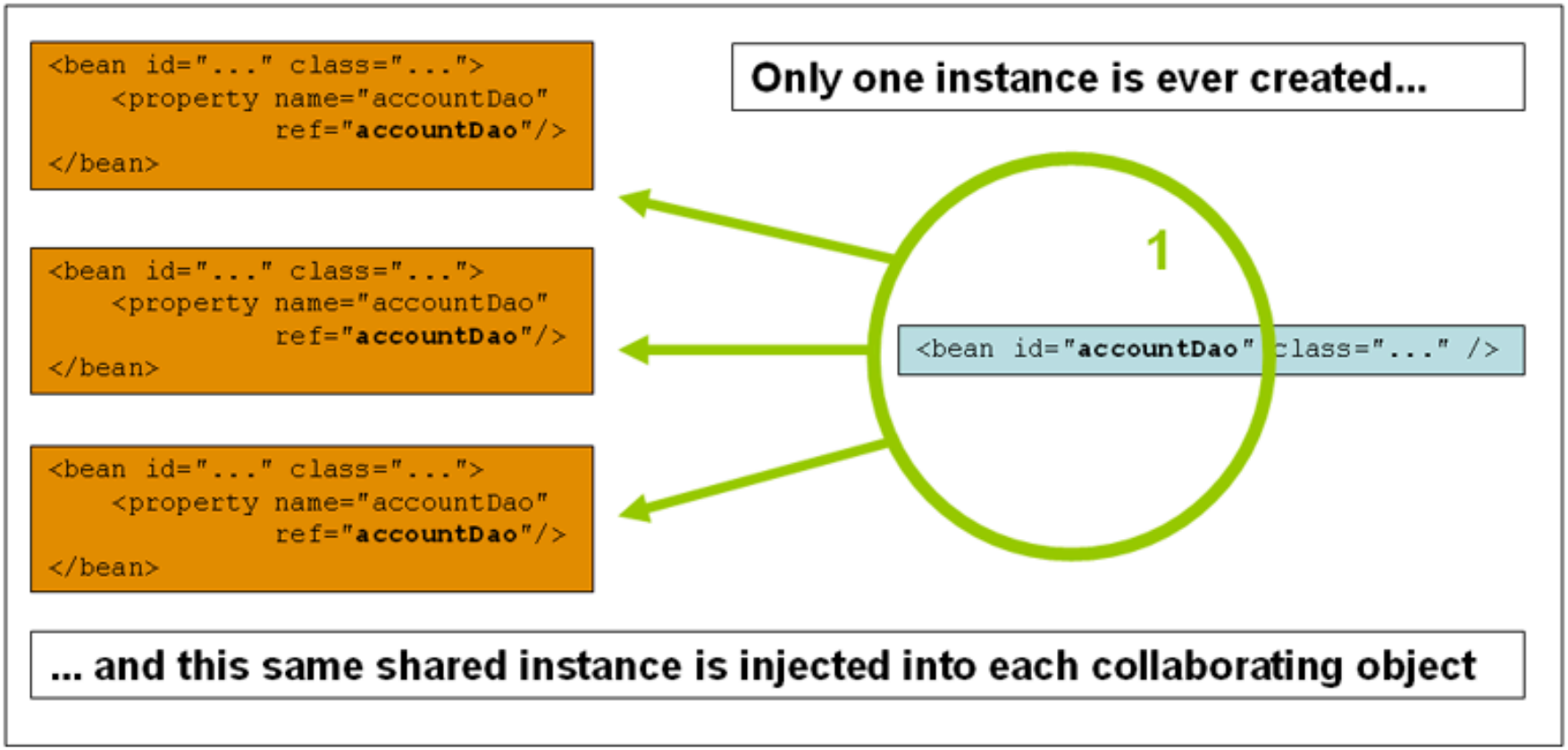
Bean Scopes - Singleton

- Eine Instanz einer Bean wird in gesamter Anwendung genutzt
- Wird in Chache aus Singleton-Beans gespeichert
- Jede Anfrage oder Referent auf diese Bean wird auf Cache verwiesen
- Singleton ist Standard für Beans

XML

```xml
<bean
  id="accountService"
  class="com.something.DefaultAccountService"
/>
```

Bean Scopes - Singleton



Bean Scopes - Singleton

Einsatz von Singleton Beans Richtlinie 5

Singleton Beans sollten für stateless Beans eingesetzt werden.

Prototype Pattern

Übersicht

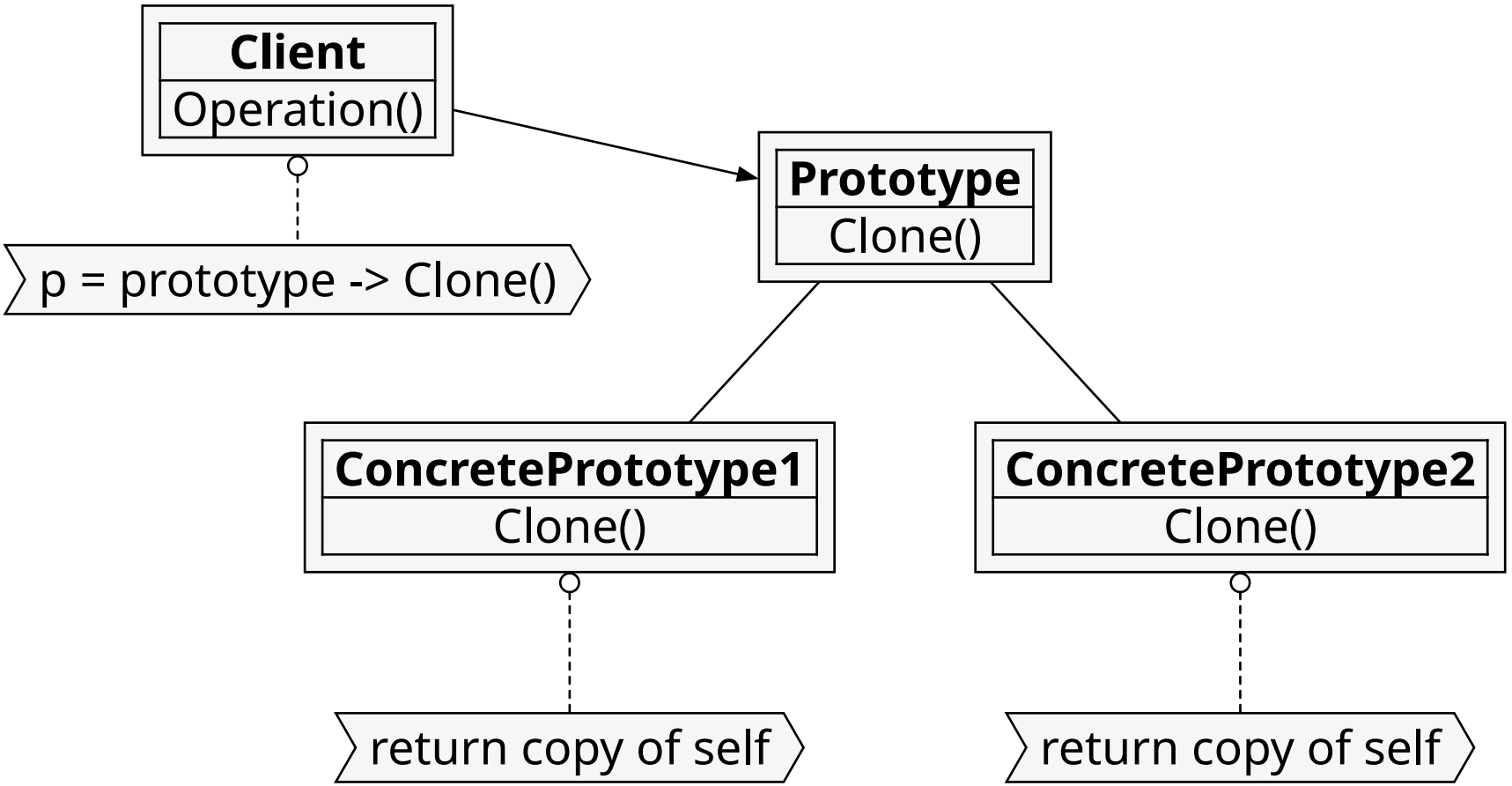
- **Ziel**: Erstellung von neuen Kopien, basierend auf einer existierenden Instanz, dem Prototypen

Wann sollte es genutzt werden?

- System muss unabhängig davon sein, wie seine Produkte erstellt, zusammengebaut und repräsentiert werden
- Wenn Klassen instanziiert werden sollen, die zur Runtime spezifiziert wurden, zum Beispiel durch `dynamic loading`.
- Um zu vermeiden, dass man eine Hierarchie an Factories baut, die die Klassenhierarchie kopiert.
- Wenn Klassen nur eine begrenzte Anzahl an State besitzen können.

Prototype Pattern

Struktur



Prototype Pattern

Struktur

- **Prototype**: Deklariert ein Interface, damit es kopiert werden kann.
- **ConcretePrototype**: Implementiert eine Operation, damit es sich selbst kopieren kann.
- **Client**: Erstellt ein neues Objekt, indem der Prototype gefragt wird, ob er eine neue Kopie von sich selbst erstellen kann.

Prototype Pattern - Vorteile

1. **Hinzufügen und Entfernen von Produkten zur Laufzeit**: Durch das Registrieren einer Prototyp-Instanz beim Client kann eine Produktklasse in das System integriert werden. Das kann, anders als bei anderen Patterns, zur Laufzeit geschehen.
2. **Spezifizierung von neuen Objekten durch das Ändern von Werten**: Dynamische Systeme erlauben das Definieren von neuem Verhalten durch Objektkomposition, zum Beispiel durch das Anpassen von Variablen. Neue Arten von Objekten werden somit durch das Instanziieren von existierenden Klassen als Prototypen von Client-Objekten erzeugt. Der Client kann somit neue Möglichkeiten erlangen, indem er Aufgaben an Prototypen auslagert.

Prototype Pattern - Vorteile

3. **Spezifizierung von neuen Objekten durch das Ändern von Struktur**: Bauen von Objekten durch Teile und Unterteile. Komplexe Strukturen können so vom Nutzer erzeugt werden. Das Prototype-Pattern unterstützt solche Ansätze durch `Deep Copy`, die im Parent-Teil implementiert sein muss.
4. **Reduziertes Subclassing**: Anders als Factories erlaubt das Prototype-Pattern das Erstellen von neuen Objekten ohne spezifische Factory-Methoden. Dadurch wird keine `Creator` Klassenhierarchie benötigt.
5. **Dynamische Konfiguration einer Anwendung durch Klassen**: Wenn die Umgebung es erlaubt, können dynamisch geladene Klassen durch das Prototype-Pattern instanziiert werden. Der Constructor der Klasse kann nicht in dieser Umgebung genutzt werden. Deshalb wird eine Instanz automatisch beim Laden erstellt und mit einem Prototype-Manager registriert. Über den Prototype-Manager können diese Klassen dann abgefragt werden.

Bean Scopes - Prototype

- Eine neue Instanz wird bei jeder Anfrage erstellt
- Anfrage kann durch Injection in eine andere Bean oder durch Funktionsaufruf geschehen

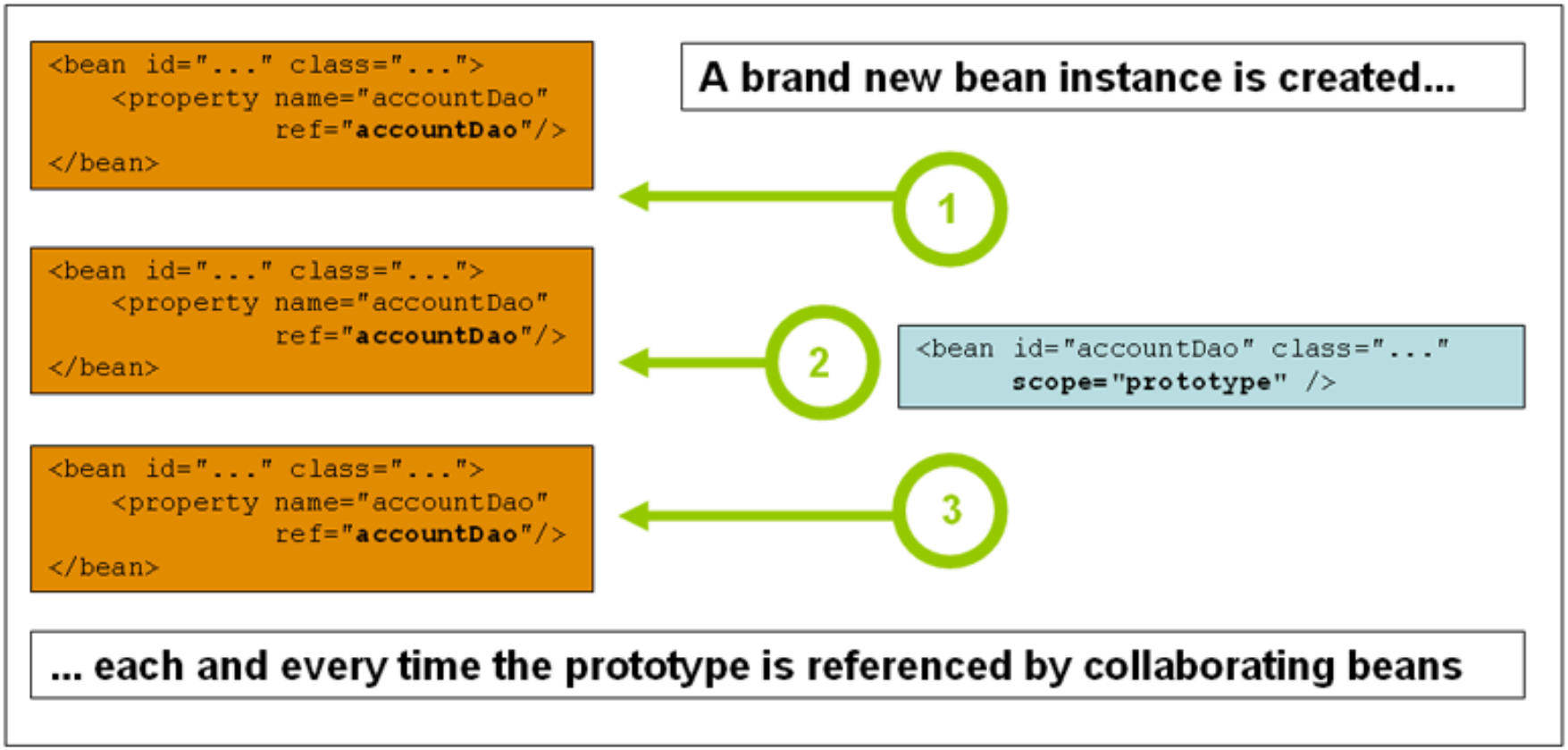
```kotlin
getBean()
```

```xml
<bean
  id="accountService"
  class="com.something.DefaultAccountService"
  scope="prototype"
/>
```

Bean Scopes - Prototype

- Spring verwaltet nicht kompletten Lebenszyklus
- Löschen muss durch Nutzer geschehen
- Selbst geschriebener Bean Post-Processor kann genutzt werden um Ressourcen freizugeben

Bean Scopes - Prototype



Bean Scopes - Request

- Eine einzelne Instanz für jede HTTP Anfrage
- Bean existiert nur so lange, wie die HTTP Anfrage beantwortet wird
- Andere Beans vom gleichen Typ sehen Änderungen nicht
- Sobald die Anfrage bearbeitet wurde, wird die Bean entfernt

XML

```xml
<bean
  id="loginAction"
  class="com.something.LoginAction"
  scope="request"
/>
```

Bean Scopes - Request

```kotlin
@RequestScope
@Component
class LoginAction {
  // ...
}
```

Bean Scopes - Session

- Eine einzelne Instanz für jede HTTP Session
- Bean wird auf Session gescoped
- State der Bean kann nur geändert werden, wenn die Session aktiv ist
- Andere Beans vom gleichen Typ sehen Änderungen nicht
- Beim Ende der Session wird die Bean entfernt

XML

```xml
<bean
  id="userPreferences"
  class="com.something.UserPreferences"
  scope="session"
/>
```

Bean Scopes - Session

Kotlin

```kotlin
@SessionScope
@Component
class UserPreferences {
  // ...
}
```

Bean Scopes - Application

- Eine Bean für die gesamte Web Anwendung
- Bean wird auf `ServletContext` gescoped
- Bean wird als Attribut von `ServletContext` gespeichert
- Unterschiede zu Singleton Beans:
  - Es existiert eine Bean pro `ServletContext`
  - Es wird exposed als Attribut von `ServletContext`

XML

```xml
<bean
  id="appPreferences"
  class="com.something.AppPreferences"
  scope="application"
/>
```

Bean Scopes - Application

```kotlin
@ApplicationScope
@Component
class AppPreferences {
  // ...
}
```


Mio
Mensa im Osten
Studentenwerk Dresden