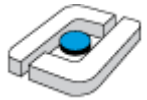
3. Annotationen und Bean Validation

- Syntax von Annotationen
- Nutzung von Annotationen
- Erstellung eigener Annotationen
- Auslesen von Annotationen
- Annotationen zur Validierung von Java Beans
- Auswertung von Bean Annotationen
- Gruppierung von Bean Annotationen
- Erstellung eigener Bean Annotationen

Einsatz von Annotationen: Weiterverarbeitung



```
// HelloWorldService.java
import javax.jws.WebMethod;
import javax.jws.WebService;
@WebService
public class HelloWorldService {
  @WebMethod
  public String helloWorld() {
    return "Hello World!";
  }
}
```

- Annotationen beginnen mit einem @ und können, z. B. von anderen Programmen zur Weiterverarbeitung genutzt werden
- hier z. B. soll die Erzeugung eines Web-Services durch die Kennzeichnung relevanter Teile durch Annotationen erfolgen
- Java besitzt seit Version 5 Annotationen, man kann selbst weitere definieren

Syntax von Annotationen

- Annotationen können Parameter haben
- ohne Parameter: `@EinfacheAnnotation`
- mit einem Parameter `@EinAnno(par="Hallo")` oder `@EinAnno("Hallo")` (bei nur einem Parameter kann der Parameternamen weggelassen werden, woher par kommt, sehen wir später)
- mit mehreren Parametern werden Wertepaare (Parametername, Wert) angegeben

  ```
  @KomplexAnno(par1="Hi", par2=42, par3={41,43})
  ```
- Annotationen können bei/vor anderen Modifiern (z.B. public, abstract) bei folgenden Elementen stehen:
  - package
  - class, interface, enum
  - Methode
  - Exemplar- und Klassenvariable
  - lokale Variablen
  - Parameter
- Man kann einschränken, wo welche Annotation erlaubt ist

Beispiel: Nutzung vordefinierter Annotation

```
public class Oben {
    public void supertolleSpezialmethode(){
        System.out.println("Ich bin oben");
    }
}
```

```
public class Unten extends Oben{
    @Override public void superTolleSpezialmethode(){
        System.out.println("Ich bin unten");
    }
    public static void main(String[] s){
        Unten u= new Unten();
        u.supertolleSpezialmethode();
    }
}
```

ohne Annotation in Java 1.4:

```
Ich bin oben
```

Compilermeldung:

```
..\..\netbeans\Annotationen\src\Unten.java:2: method does not
override a method from its superclass
    @Override public void superTolleSpezialmethode(){
1 error
```

Annotationen selbst schreiben

```
public @interface Name {
  String vor();
  String nach();
}
```

in `@interface` erlaubte Typen:
-Basisdatentypen
-String
-Aufzählungen (hier oder woanders definiert)
- (weitere) @interface
-Arrays der obigen Elemente

```
public @interface Problem {
  int id();
  String text();
  Name verantwortlich();
  enum Paket{ GUI, BUIS, INFRA, OLD;
    @Override public String toString() {
      return "das " + name() + " Paket";
    }
  };
  Paket[] betroffen() default {Paket.GUI,Paket.BUIS,
                  Paket.INFRA, Paket.OLD};
}
```

- Jede Annotation steht in ihrer eigenen Klassen-Datei
- Erben automatisch von `java.lang.annotation.Annotation`
- Ihre Methoden haben keine Parameter (ein Feld des Methodennamens wird für die Annotation festgelegt)

Mit eigenen Annotationen annotieren

```
@Problem(
  id=1,
  verantwortlich= @Name(vor="Heinz", nach="Meier"),
  text="Koordination offen",
  betroffen={Problem.Paket.GUI,Problem.Paket.BUIS}
)
public class KlasseInBearbeitung {
  String eineExemplarvariable;

  @Problem(
    id=2,
    verantwortlich= @Name(vor="Horst", nach="Meier"),
    text="Test in main schreiben"
  )
  public static void main(String[] s){
    System.out.println("Noch viel zu tun");
  }
}
```

Annotation von Annotationen

- Grundsätzlich kann es sinnvoll sein, Annotationen selbst mit Zusatzinformationen zu annotieren (Meta-Information)
- nicht gemeint ist folgendes:

```
@Override @Deprecated public void xyz(){
```

  dies ist nur eine Menge von Annotationen für ein Element
- Meta-Informationen stehen in der „Klassendefinition“ der Annotationen

Es gibt folgende Standard-Meta-Annotationen:

- `@Target` (wo darf die Annotation stehen)

```
@Target({ElementType.TYPE, ElementType.METHOD})
```

- `@Retention` („Beibehaltung“, wo soll Annotation sichtbar sein, z. B. auch noch zur Laufzeit?)

```
@Retention(RetentionPolicy.RUNTIME)
```

- `@Documented` (keine Parameter, Annotation in Doku sichtbar?)
- `@Inherited` (keine Parameter, Annotation soll vererbt werden)

Wertebereiche der Meta-Annotationen (aus Doku)

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.METHOD})
```

CLASS Annotations are to be recorded in the class file by the compiler but need not be retained by the VM at run time.
RUNTIME Annotations are to be recorded in the class file by the compiler and retained by the VM at run time, so they may be read reflectively.
SOURCE Annotations are to be discarded by the compiler.

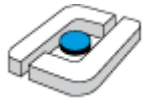
ANNOTATION_TYPE Annotation type declaration
CONSTRUCTOR Constructor declaration
FIELD Field declaration (inlcudes enum constants)
LOCAL_VARIABLE Local variable declaration
METHOD Method declaration
PACKAGE Package declaration
PARAMETER Parameter declaration
TYPE Class, interface (including annotation type), or enum declaration

Beispiel für zur Laufzeit lesbare Annotation

```
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.METHOD, ElementType.FIELD,
         ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented  // auch hier wichtig
public @interface Name {
    String vor();
    String nach();
}
```

Annotationen über Reflektion auslesen (1/2)



- Typischerweise werden Annotationen von speziellen Werkzeugen für Transformationen verwendet
- SUN hat ab Java 5 ein zusätzliches Werkzeug `apt` (annotation processing tool) für Bearbeitungen/Transformationen hinzugefügt
- Damit wir was sehen, nutzen wir Reflexion (Klassen werden als Objekte betrachtet)

```
import java.lang.annotation.Annotation;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class Reflexionstest {
  public static void main(String[] s){
    Class c = KlasseInBearbeitung.class;
    System.out.println(c);
```

Annotationen über Reflektion auslesen (2/2)

```
    Field[] attribute= c.getDeclaredFields();
    for(Field f:attribute) System.out.println(f);
    Method[] methoden= c.getDeclaredMethods();
    for(Method m:methoden){
      System.out.println(m);
      Annotation[] annos = m.getAnnotations();
      for(Annotation an:annos){
        System.out.println(an);
        Problem p = (Problem)an;  // kritisch
        System.out.println(p.verantwortlich());
      }
    }
  }
}
```

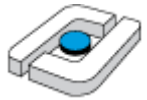
```
class KlasseInBearbeitung
java.lang.String KlasseInBearbeitung.eineExemplarvariable
public static void KlasseInBearbeitung.main(java.lang.String[])
@Problem(betroffen=[das GUI Paket, das BUIS Paket, das INFRA
Paket, das OLD Paket], id=2, verantwortlich=@Name(vor=Horst,
nach=Meier), text=Test in main schreiben)
@Name(vor=Horst, nach=Meier)
```

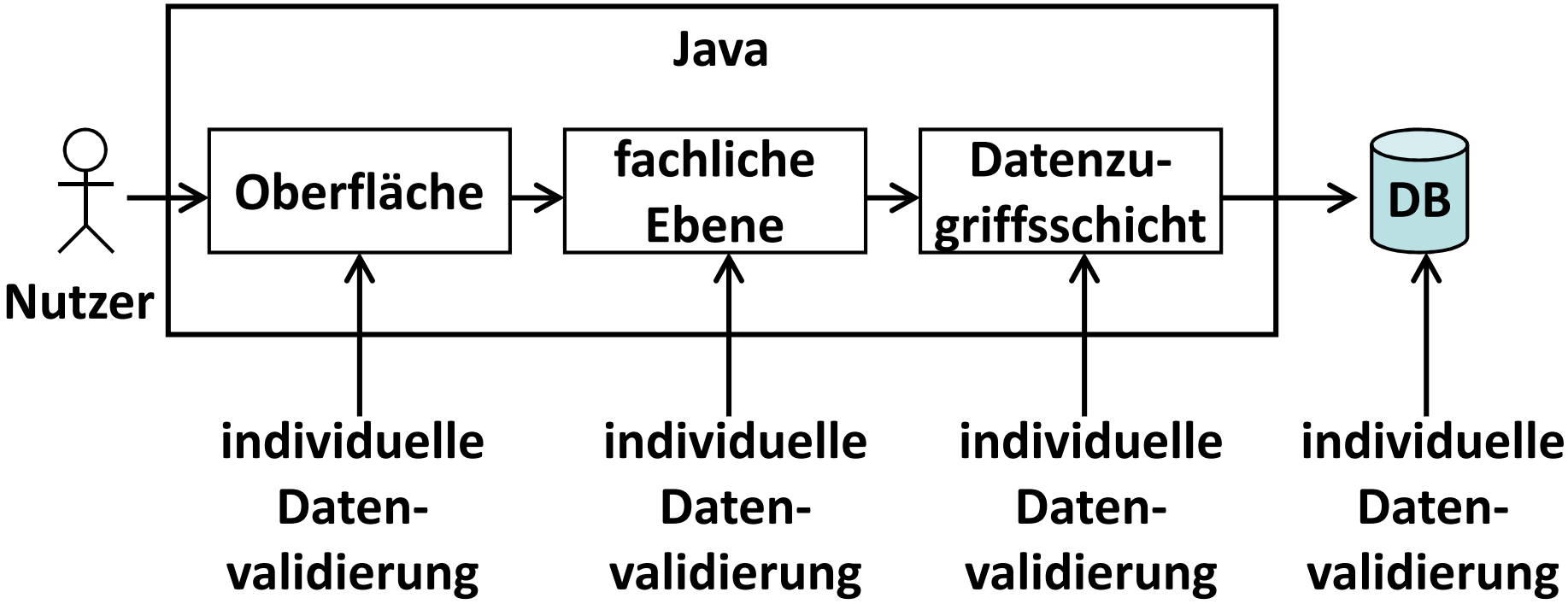
Annotationen und Java Beans

- Beans eignen sich sehr gut zur reinen Datenhaltung (vgl. Entitäten für Datenbanken)
- Beans stellen damit das Datenmodell (z. B. im Sinn MVC) dar
- generell spielt Datenqualität wichtige Rolle; haben alle Exemplarvariablen die richtigen Werte
- Prüfungen finden häufig auf verschiedenen Darstellungsebenen statt
- Prüfungen in Bean sinnvoll, Fehlerweitergabe über Hinweise oder Execeptions
- Ansatz: Dateneigenschaften (Constraints) in Annotationen festhalten
- JSR 303: Bean Validation standardisiert Interfaces von Validierungs-Frameworks, Version 1.1 ist JSR 349

http://jcp.org/en/jsr/detail?id=303 http://jcp.org/en/jsr/detail?id=349

Problem: Viele verteile Validierungsaktionen

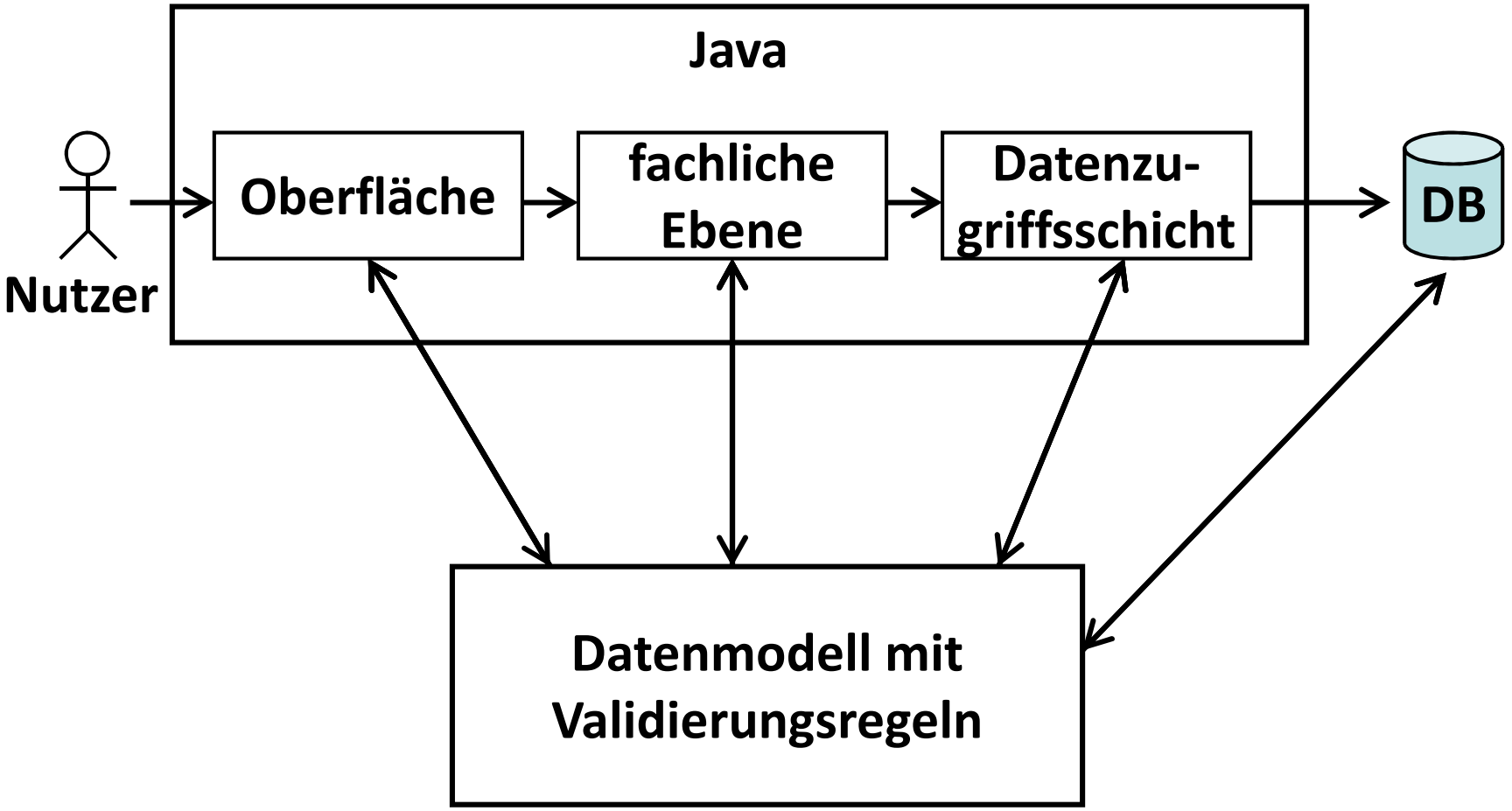




nach: [HV] Hibernate Validator, JSR 303 Reference Implementation Reference Guide (u. a. im docs-Verzeichnis der Referenz-Implementierung, \hibernate-validator-5.0.0.Alpha1-dist.zip\hibernate-validator-5.0.0.Alpha1\docs\reference\en-US\pdf)

http://sourceforge.net/projects/hibernate/files/hibernate-validator/5.0.0.Alpha1/

Ansatz: zentrale Validierung [HV]



Zentrale Idee: Annotation von Java-Elementen

```
import java.io.Serializable;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.Size;
public class Student implements Serializable {
  @Min(100000) @Max(999999)
  private int matnr;
  @Size(max=8)
  private String name;
  public int getMatnr() {return matnr;}
  public void setMatnr(int matnr) {this.matnr = matnr;}
  public String getName() {return name;}
  public void setName(String name) {this.name = name;}
  @Override public String toString(){
    return name+" ("+matnr+")";
  }
}
```

Frage: Wann wie validieren?

1. Idee: Jedes Mal, wenn Objekt bearbeitet wird, Regeln prüfen, bei Verstoß Exception
- Vorteil: immer konsistente Daten
- Nachteil: unflexibel, Daten beim Anlegen und Ändern evtl. nicht immer konsistent

2. Idee: Überprüfung zu selbst gewählten Zeitpunkten starten, bei Verstoß Exception
- Vorteil: flexibel einsetzbar, performanter als 1.
- Nachteil: Exception ist harter Fehler, zerstört Ablauf

3. Idee: Überprüfung zu selbst gewählten Zeitpunkten starten, Ergebnis gibt Auskunft über Fehler
- Vorteil: flexibel, Nutzer kann Schweregrad für Fehler individuell festlegen

Rahmenwerk für Validierungsbeispiele

```
public class Main<T> { // T ist zu analysierende Klasse
  public int analyse(T o, Class... cl) {
    System.out.println("Analyse von " + o);
    ValidatorFactory factory =
                Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();
    Set<ConstraintViolation<T>> cv = validator.validate(o,cl);
    for (ConstraintViolation<T> c : cv)
      System.out.println(" :: " + c.getMessage());
    return cv.size();
  }
  ... // jeweils main-Methode mit Beispiel
}
```

- Anmerkung: Exception-Lösung leicht realisierbar

```
if(cv.size()>0) throw new IllegalArgumentException(...)
```

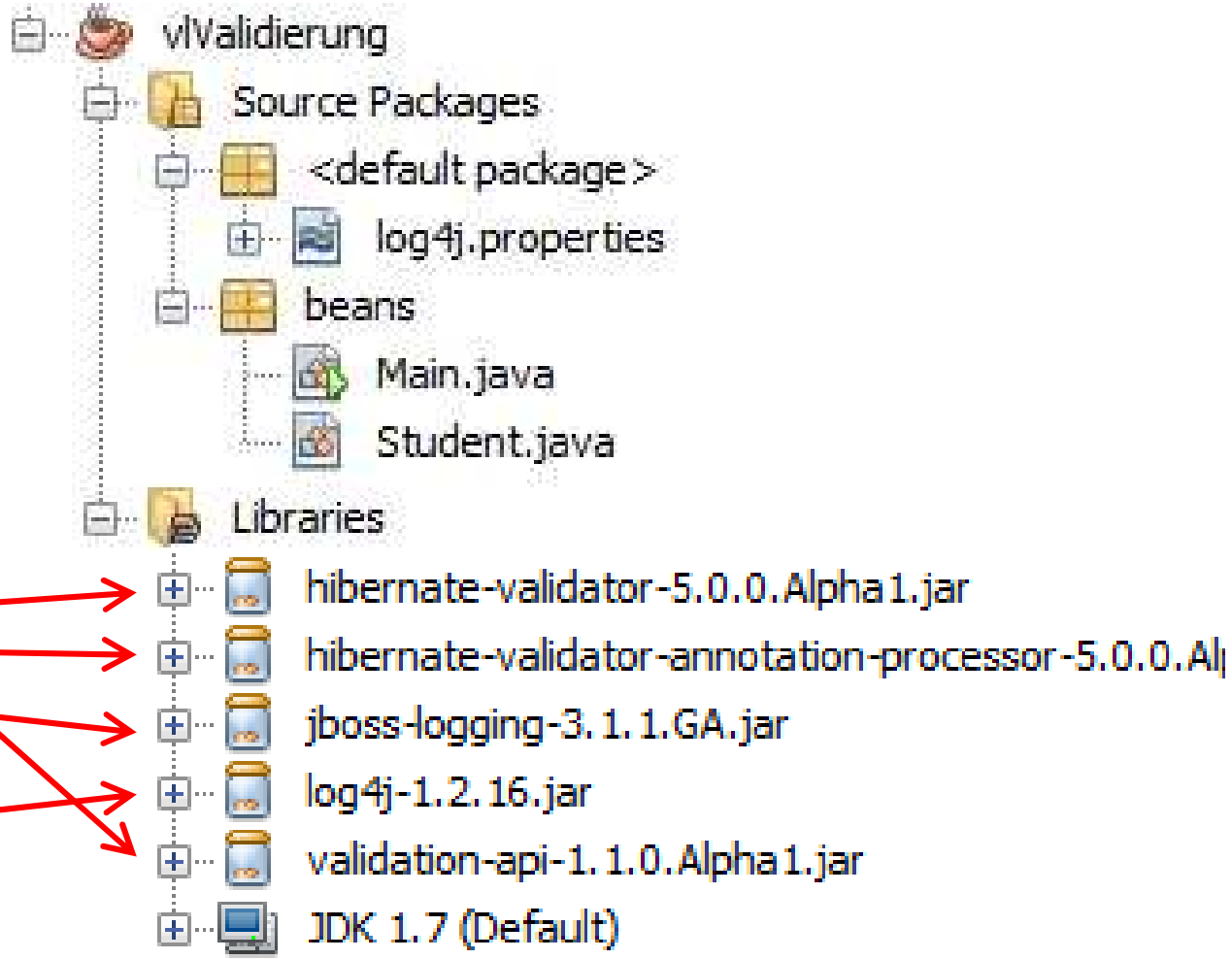
Validierungen durchführen

```
public static void mainStudent(String[] args) {
   Main<Student> m = new Main<>();
   Student s = new Student();
   m.analyse(s);
   s.setMatnr(424242);
   s.setName("Anschie");
   m.analyse(s);
   s.setMatnr(43);
   s.setName("123456789");
   m.analyse(s);
}
```

```
Analyse von null (0)
 :: muss grössergleich 100000 sein
Analyse von Anschie (424242)
Analyse von 123456789 (43)
 :: muss grössergleich 100000 sein
 :: muss zwischen 0 und 8 liegen
```

Projektstruktur (1/2)

Einbindung einer Bean-Validation-Implementierung (Referenzimplementierung Hibernate)

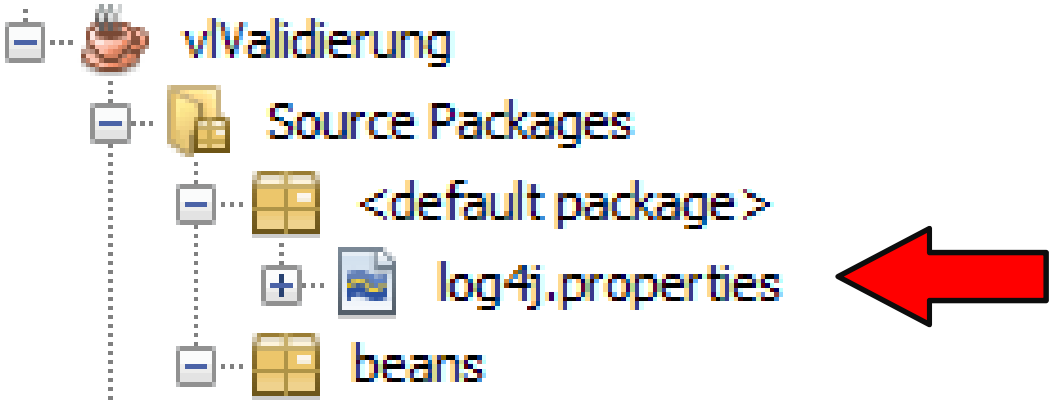


aus dist

aus dist\lib\required

aus diat\lib\optional

Projektstruktur (2/2)



- minimale Konfiguration des Loggers

log4j.properties

Source | History

```
1  log4j.rootLogger=WARN, console
2  log4j.appender.console=org.apache.log4j.ConsoleAppender
3  log4j.appender.console.layout=org.apache.log4j.PatternLayout
4  log4j.appender.console.layout.conversionPattern=%5p [%t] (%F:%L) - %m%n
```

Eigene Fehlermeldungen (1/2)

```
public class Student2 {

    @Min(value=100000, message="sechs Stellen")
    @Max(value=999999, message="sechs Stellen")
    private int matnr;
    @NotNull(message="Ohne Name is nich")
    private String name;

   // get-, set- und toString wie vorher
```

Eigene Fehlermeldungen (2/2)

```
public static void mainStudent2(String[] args) {
   Main<Student2> m = new Main<>();
   Student2 s = new Student2();
   m.analyse(s);
   s.setMatnr(424242);
   s.setName("Anschie");
   m.analyse(s);
   s.setMatnr(43);
   s.setName("123456789");
   m.analyse(s);
 }
```

```
Analyse von null (0)
 :: Ohne Name is nich
 :: sechs Stellen
Analyse von Anschie (424242)
Analyse von 123456789 (43)
 :: sechs Stellen
```

Analyse assoziierter Objekte (1/3)

```
public class Modul {
  @Min(100) @Max(999)
  private int modnr;
  @NotNull @Size(max=30)
  private String titel;
  @Min(1) @Max(6)
  private int semester;
  @Min(1) @Max(30)
  private int cp;
  private boolean abgeschlossen;
  @Digits(integer=1,fraction=1)
  private double note;

  public Modul(){}  // get -und set- wie üblich

  public Modul(int modnr, String titel, int semester, int cp,
               double note) { //... wie üblich}

  @Override
  public String toString() {
    return "Note:"+note+" "+titel+"("+modnr+")";
  }
```

Analyse assoziierter Objekte (2/3)

```
public class Student3 { // Konstruktor, get, set wie üblich
  @Min(value=100000, message="sechs Stellen")
  @Max(value=999999, message="sechs Stellen")
  private int matnr;
  @NotNull(message="Ohne Name is nich")
  private String name;
  @Valid @Size(max=28)
  private List<Modul> abgeschlossen = new ArrayList<Modul>();

  public void bestanden(Modul m){
    abgeschlossen.add(m);
  }

  @Override
  public String toString(){
    StringBuffer erg= new StringBuffer(name+" ("+matnr
                                          +")  Zeugnis:\n");
    for(Modul m:abgeschlossen)
      erg.append("  "+m.toString()+"\n");
    return erg.toString();
  }
```

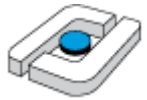
Analyse assoziierter Objekte (3/3)

```
public static void mainStudent3(String[] args) {
   Main<Student3> ms = new Main<>();
   Main<Modul> mm = new Main<>();
   Modul bah = new Modul(42, "C-Techniken", 0, 31, 5.5);
   mm.analyse(bah);
   Student3 s = new Student3();
   s.setMatnr(4242);
   s.setName("Anschie");
   s.bestanden(bah);
   ms.analyse(s);
 }
```

```
Analyse von Note:5.5 C-Techniken(42)
 :: muss kleinergleich 30 sein
 :: muss grössergleich 100 sein
 :: muss grössergleich 1 sein
Analyse von Anschie (4242)  Zeugnis:
  Note:5.5 C-Techniken(42)

 :: muss grössergleich 100 sein
 :: muss kleinergleich 30 sein
 :: sechs Stellen
 :: muss grössergleich 1 sein
```

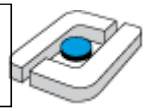
Einschub: genauere Validierungsmöglichkeiten



```
public static void mainValidierungen(String[] args) {
  Modul bah = new Modul(42, "C-Techniken", 0, 31, 5.0);
  ValidatorFactory ft = Validation.buildDefaultValidatorFactory();
  Validator validator = ft.getValidator();
  Set<ConstraintViolation<Modul>> cv =
          validator.validateProperty(bah, "titel");
  System.out.println("titel: " + cv.size());
  cv = validator.validateProperty(bah, "cp");
  System.out.println("cp: " + cv.size());
  cv = validator.validateValue(Modul.class, "titel", null);
  System.out.println("titel als null: " + cv.size());
}
```

```
titel: 0
cp: 1
titel als null: 1
```

Annotationen im Standard 1.0



Annotation	Kurzerklärung/Beispiel
@AssertFalse	Element muss false sein
@AssertTrue	Element muss true sein
@DecimalMax	z. B. @DecimalMax("42"), auch BigInteger
@DecimalMin	analog, weiterhin auch BigDecimal, String, double
@Digits(integer,fraction)	Stellen vor und nach dem Komma, BigDecimal
@Future	Datum (Date, Calendar) in der Zukunft
@Length(min,max)	String-Länge zwischen min und max einschließlich
@Max	@Max(45) Maximalwert
@Min	Minimalwert
@NotNull	Element nicht null
@Null	Element muss nul sein
@Past	Datum in der Vergangenheit
@Pattern(regex,flag)	String muss regulären Ausdruck erfüllen
@Size(min,max)	zwischen min und max einschl. (auch Collection)
@Valid	für rekursive Validierung

Flexibilisierung der Validierung

- Bis jetzt: Jede Validierungsregel wird immer vollständig bei `validate` geprüft
- Häufig: Nicht alle Regeln müssen immer eingehalten werden
- Beispiel: Eingabeseite mit n-Feldern
  - wenn erster Wert eingeben wird, sollen/dürfen restliche Felder nicht validiert werden
  - ähnlich bei Wizzards zur Abfrage von Daten
- Ansatz: Regeln können Gruppen zugeordnet werden; man kann angeben welche Gruppen validiert werden sollen
- bisher: alles in Default-Gruppe, genauer `javax.validation.groups.Default`

Beschreibung von Gruppen

- Da Java (bisher) der Grundregel folgt „keine neuen Schlüsselwörter“, können einfach Interfaces oder Klassen als Gruppenbezeichner genutzt werden

```
public interface AbschlussChecks {
}

public interface AnlegenChecks {
}
```

- Die Nutzung als Gruppe sollte im Kommentar zum Interface stehen

Nutzung von Gruppen (1/2)

```
public class Modul2 { // Konstruktor, get, set wie üblich
  @Min(value=100, groups=AnlegenChecks.class)
  @Max(value=999, groups=AnlegenChecks.class)
  private int modnr;
  @NotNull @Size(max=30)
  private String titel;
  @Min(value=1, groups=AnlegenChecks.class)
  @Max(value=6, groups=AnlegenChecks.class)
  private int semester;
  @Min(value=1,groups={AbschlussChecks.class,AnlegenChecks.class})
  @Max(value=30,
       groups={AbschlussChecks.class,AnlegenChecks.class})
  private int cp;
  private boolean abgeschlossen;
  @Digits(integer=1,fraction=1,groups=AbschlussChecks.class)
  private double note;
```

Nutzung von Gruppen (2/2)

```
public static void mainModul2(String[] args) {
  Main<Modul2> mm = new Main<Modul2>();
  Modul2 bah = new Modul2(42, "C-Techniken", 0, 31, 5.55);
  mm.analyse(bah);
  mm.analyse(bah, AbschlussChecks.class);
  mm.analyse(bah, AnlegenChecks.class);
  mm.analyse(bah, AbschlussChecks.class, AnlegenChecks.class);
  bah.setTitel(null);
  mm.analyse(bah);
 }
```

```
Analyse von Note:5.55 C-Techniken(42)
Analyse von Note:5.55 C-Techniken(42)
 :: numerischer Wert außerhalb erlaubten Wertebereic
 :: muss kleinergleich 30 sein
Analyse von Note:5.55 C-Techniken(42)
 :: muss kleinergleich 30 sein
 :: muss grössergleich 100 sein
 :: muss grössergleich 1 sein
Analyse von Note:5.55 C-Techniken(42)
 :: numerischer Wert außerhalb erlaubten Wertebereic
 :: muss kleinergleich 30 sein
 :: muss grössergleich 100 sein
 :: muss grössergleich 1 sein
Analyse von Note:5.55 null(42)
 :: kann nicht null sein
```

Erstellung eigener Validierungsannotationen

- Standardisierte Annotationen prüfen sehr gut einfache Constraints
- konkrete Wertebereiche, z. B. für Noten, so nicht definierbar
- (komplexe) Zusammenhänge zwischen Exemplarvariablen nicht darstellbar (vgl. Constraints und Trigger für Datenbanken)
- eigene Annotationen auf Exemplarvariablen, aber auch z. B. Klassen anwendbar
- genereller Ansatz: eigene Annotation schreiben, dazu eine dort genutzte Validierungsklasse angeben

Beispiel - eigene Annotation

```
@Target({ElementType.METHOD, ElementType.FIELD,
         ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = ModulCheckValidator.class)
@Documented
public @interface ModulCheck {
  String message() default "Moduleintrag kaputt";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
  boolean value() default true;
}
```

- message, groups, payLoad müssen in dieser Form angegeben werden
- weitere Attribute können beliebig ergänzt werden

Beispiel: Validierungsklasse (gleich genauer)

```
public class ModulCheckValidator implements
                    ConstraintValidator<ModulCheck,Double>{
  private boolean auch5;
  private double[] ok={1.0,1.3,1.7,2.0,2.3,2.7,3.0,3.3,3.7,4.0};

  public void initialize(ModulCheck a) {
    auch5=a.value();
  }

  public boolean isValid(Double t,
        ConstraintValidatorContext cvc) {
    System.out.println("in isValid: "
        +cvc.getDefaultConstraintMessageTemplate());
    boolean ergebnis=false;
    for(double d:ok)
      if(d==t)
        return true;
    return ergebnis || (auch5 && t==5.0);
  }
}
```

Beispiel: Nutzung eigener Annotation (1/2)

```
public class Modul3 {// Kostruktor, get, set wie üblich
  @Min(value=100, groups=AnlegenChecks.class)
  @Max(value=999, groups=AnlegenChecks.class)
  private int modnr;
  @NotNull @Size(max=30)
  private String titel;
  @Min(value=1, groups=AnlegenChecks.class)
  @Max(value=6, groups=AnlegenChecks.class)
  private int semester;
  @Min(value=1,groups={AbschlussChecks.class,AnlegenChecks.class})
  @Max(value=30,groups={AbschlussChecks.class,
                        AnlegenChecks.class})
  private int cp;
  private boolean abgeschlossen;
  @Digits(integer=1,fraction=1,groups=AbschlussChecks.class)
  @ModulCheck(false)
  private double note;
```

Beispiel: Nutzung eigener Annotation (2/2)

```
public static void main(String[] args) {
   Main<Modul3> mm = new Main<>();
   Modul3 bah = new Modul3(42, "C-Techniken", 0, 31, 4.3);
   mm.analyse(bah);
   bah.setNote(5.0);
   mm.analyse(bah);
   bah.setNote(2.3);
   mm.analyse(bah);
 }
```

```
Analyse von Note:4.3 C-Techniken(42)
in isValid: Moduleintrag kaputt
 :: Moduleintrag kaputt
Analyse von Note:5.0 C-Techniken(42)
in isValid: Moduleintrag kaputt
 :: Moduleintrag kaputt
Analyse von Note:2.3 C-Techniken(42)
in isValid: Moduleintrag kaputt
```

Validierungsklasse genauer

- muss Interface `ConstraintValidator` realisieren
- Interface generisch mit zwei Klassen <K1,K2>
- K1 muss Name der Annotation sein
- K2 Typ der Exemplarvariable, die annotiert werden kann
- sollen verschiedene K2-Typen möglich sein, muss es jeweils eine individuelle Realisierung geben, sonst zur Laufzeit
  ```
  javax.validation.UnexpectedTypeException: No validator
  could be found for type: java.lang.Integer
  ```
- In `initalize`-Methode kann auf Werte der Attribute der Annotation zugegriffen werden
- mit `isValid` muss true oder false berechnet werden
- `ConstraintValidatorContext` kann auch zur Erzeugung von Execeptions genutzt werden

Objektvalidierung

- statt einzelner Exemplarvariablen will man oft Zusammenhänge evaluieren (CONSTRAINT -> TRIGGER)
- Ansatz: Klasse annotieren

```
@Objektregel(groups = Objektregelcheck.class)
public class Voodoo {
```

- Annotation (auch) für Klassen zulassen

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = {ObjektRegelValidator.class})
@Documented
public @interface Objektregel {
```

- Validierer (wie bekannt) schreiben

```
public class ObjektRegelValidator implements
          ConstraintValidator<Objektregel, Voodoo> {
  public boolean isValid(Voodoo t,
                         ConstraintValidatorContext cvc) {
```

Weitere Möglichkeiten bei Bean Validation

- Die Klasse `ConstraintViolation` bietet einige Methoden die Verletzung genauer zu analysieren
- Statt immer mehrere Annotationen zu nutzen, können diese zu einer neuen Annotation zusammengefasst werden
- Mit der Annotation `@GroupSequence` kann die Reihenfolge der Validierungen der Gruppen bestimmt werden
- Annotation mit `@GroupSequence` erlaubt auch Default-Validierung für Klasse zu ersetzen
- Fehlermeldungen können (leicht) über Property-Dateien internationalisiert werden
- Validierungsregeln auch in XML formulierbar
- Validierungsregeln könnten alternativ bei get- und set-Methoden stehen
- Einschränkung: Keine Validierung von Klassenvariablen

Weitergehende Validierungsansätze

- Zusammenfassen von Constraints

```
@Constraint(validatedBy={})
@Min(10000)
@Max(99999)
public @interface GueltigeMatrikelnummer { ...
```

- Constraints für Parameter und Rückgabewerte (-> Design by Contract/Programming)

```
public @NotNull Set<Movie> findMoviesByTitle(
        @NotNull @Size(min = 3) String title) {...
```

  - Anmerkung: Vor- und Nachbedingungen wie in der Programmiersprache Eiffel (Bertrand Meier)
  - Benötigt weitere Arten von Validierungsklassen

- z.B.:http://musingsofaprogrammingaddict.blogspot.com/2011/01/method-validation-with-hibernate.html