Kapitel 8: Reflection

Reflection API

- Java bietet eine Reflection API.
- Diese erlaubt Zugriff auf die Typstruktur zur Laufzeit.
- Auf Basis der Typstruktur kann dynamisch Verhalten bestimmt werden.

Methoden der Klasse `Object`

`Object()`	`Object` hat einen Defaultkonstruktor…
`Object clone()`	Gibt eine (echte) Kopie des Objekts zurück
`boolean equals( Object obj)`	Gibt an, ob das Objekt einem anderen Objekt `obj` „gleicht“.
`void finalize()`	Wird vom Gargabe Collector vor dem Zerstören des Objekts aufgerufen, um Ressourcen freizugeben
`Class<?> getClass()`	**Gibt Informationen über die Klasse zurück (Reflection)**
`int hashCode()`	Gibt einen Hashcode zurück
`String toString()`	Gibt eine String-Repräsentation des Objekts zurück

Die Klasse `Class<T>`

`Constructor<?>[] getDeclaredConstructors()`	Gibt Konstruktoren der Klasse zurück
`Field[] getDeclaredFields()`	Gibt Attribute der Klasse zurück
`Method[] getDeclaredMethods()`	Gibt Methoden der Klasse zurück
`Constructor<?>[] getConstructors()`	Gibt Public-Konstruktoren der Kl. zurück
`Field[] getFields()`	Gibt Public-Attribute der Klasse zurück, einschließlich der geerbten
`Method[] getMethods()`	Gibt Public-Methoden der Klasse zurück, einschließlich der geerbten
`int getModifiers()`	Gibt Modifier der Klasse zurück
`String getName()`	Gibt den qualifizierten Namen der Klasse zurück (inkl. Packagenamen)
`String getSimpleName()`	Gibt den einfachen Namen der Klasse zurück (ohne Packagenamen)
`Package getPackage()`	Gibt das umschließende Package zurück
`Class<? super T> getSuperclass()`	Gibt die Oberkl. als `Class`–Objekt zurück

Die Klassen `Field` und `Method`

- Die Klasse `Field`

`Class<?> getDeclaringClass()`	Gibt die deklarierende Klasse zurück
`int getModifiers()`	Gibt Modifier des Attributs zurück
`Class<?> getType()`	Gibt den Typ des Attributs zurück
`String getName()`	Gibt den Namen des Attributs zurück

- Die Klasse `Method`

`Class<?> getDeclaringClass()`	Gibt die deklarierende Klasse zurück
`int getModifiers()`	Gibt Modifier der Methode zurück
`Class<?> getReturnType()`	Gibt den Rückgabetyp der Methode zurück
`String getName()`	Gibt den Namen der Methode zurück

Die Klasse `Rechteck`

```
public class Rechteck extends Flaeche {

   private int breite;
   private int hoehe;

   public Rechteck(int x, int y, int breite, int hoehe) { … }

   public void setBreite(int breite) { … }
   public int getBreite() { … }
   public void setHoehe(int hoehe) { … }
   public int getHoehe() { … }

   public void paint(Graphics g) { … }
}
```

Ausgabe

Übrigens:
Reihenfolge
ist anders…

```
Name: Rechteck

Attribute:
- int breite
- int hoehe

Konstruktoren:
- Rechteck(int, int, int, int)

Methoden:
- void setBreite(int)
- void setHoehe(int)
- int getBreite()
- int getHoehe()
- void paint(Graphics)
```

Typinfo ausgeben

```
private static void printClass(Class<?> cls) {

   System.out.println("Name: " + cls.getName());

   System.out.println("\nAttribute:");

   for (Field f : cls.getDeclaredFields()) {
      System.out.println("- "
         + f.getType().getSimpleName()
         + " " + f.getName());
   }

   …
```

Typinfo ausgeben

```
…

System.out.println("\nKonstruktoren:");

for (Constructor<?> c : cls.getDeclaredConstructors()) {
   System.out.print("- " + c.getName() + "(");
   printParameters(c.getParameterTypes());
   System.out.println(")");
}

…
```

Typinfo ausgeben

```
    …

    System.out.println("\nMethoden:");

    for (Method m : cls.getDeclaredMethods()) {
        System.out.print("- "
            + m.getReturnType().getSimpleName()
            + " " + m.getName() + "(");
        printParameters(m.getParameterTypes());
        System.out.println(")");
    }
}
```

Typinfo ausgeben

```
private static void printParameters(Class<?>[] params) {
    boolean first = true;

    for (Class<?> p : params) {
        if (first) first = false;
        else System.out.print(", ");

        System.out.print(p.getSimpleName());
    }
}
```

```
public static void main(String[] args) {
    Flaeche flaeche = new Rechteck(0, 0, 100, 30);
    printClass(flaeche.getClass());
}
```

Drei Wege zur `Class`-Instanz

- Durch `getClass()` kann auf das `Class`-Objekt einer Instanz zugegriffen werden, z.B.:

```
Flaeche flaeche = new Rechteck();
Class<?> rechteckClass = flaeche.getClass();
```

- Zugriff auf ein `Class`-Objekt ist auch über das implizit definierte (statische) Attribut `class` einer Klasse möglich, z.B.:

```
Class<?> rechteckClass = Rechteck.class;
```

- Schließlich ermöglicht `Class.forName()` den Zugriff auf ein `Class`-Objekt per Zeichenkette:

```
rechteckClass = Class.forName("Rechteck");
```

Die Klasse `Class<T>`

`Constructor<?>[] getDeclaredConstructors()`	Gibt Konstruktoren der Klasse zurück
`Field[] getDeclaredFields()`	Gibt Attribute der Klasse zurück
`Method[] getDeclaredMethods()`	Gibt Methoden der Klasse zurück
`Constructor<?>[] getConstructors()`	Gibt Public-Konstruktoren der Kl. zurück
`Field[] getFields()`	Gibt Public-Attribute der Klasse zurück, einschließlich der geerbten
`Method[] getMethods()`	Gibt Public-Methoden der Klasse zurück, einschließlich der geerbten
`int getModifiers()`	**Gibt Modifier der Klasse zurück**
`String getName()`	Gibt den qualifizierten Namen der Klasse zurück (inkl. Packagenamen)
`String getSimpleName()`	Gibt den einfachen Namen der Klasse zurück (ohne Packagenamen)
`Package getPackage()`	Gibt das umschließende Package zurück
`Class<? super T> getSuperclass()`	Gibt die Oberkl. als `Class`–Objekt zurück

Exkurs: Umgang mit Flags

```
public class Modifier {
    public static final int PUBLIC           = 0x00000001;
    public static final int PRIVATE          = 0x00000002;
    public static final int PROTECTED        = 0x00000004;
    public static final int STATIC           = 0x00000008;
    public static final int FINAL            = 0x00000010;
    public static final int SYNCHRONIZED     = 0x00000020;
    public static final int VOLATILE         = 0x00000040;
    public static final int TRANSIENT        = 0x00000080;
    public static final int NATIVE           = 0x00000100;
    public static final int INTERFACE        = 0x00000200;
    public static final int ABSTRACT         = 0x00000400;
    public static final int STRICT           = 0x00000800;
}
```

- Wie können Flags ausgelesen werden?
- Wie können Flags manipuliert werden?

Exkurs: Umgang mit Flags

PUBLIC	0x001
PRIVATE	0x002
PROTECTED	0x004
STATIC	0x008
FINAL	0x010
SYNCHRONIZED	0x020
VOLATILE	0x040
TRANSIENT	0x080
NATIVE	0x100
INTERFACE	0x200
ABSTRACT	0x400
STRICT	0x800

- Beispiel:

```
public class Rechteck {
   private static final int INTERVALL = 10;
   …
}
```

- Welchen Wert liefert `getModifier()`?

```
Class<?> cls = Rechteck.class;
Field f = cls.getDeclaredField("INTERVALL");
int mod = f.getModifiers();
System.out.println(mod);
```

Exkurs: Umgang mit Flags

- Flag prüfen

```
boolean privateSet =
    (mod & Modifier.PRIVATE) != 0;
```

0000 0001 10**1**0 0000 0000 00**1**0
0000 0000 00**1**0

- Flag einer Gruppe auslesen

```
int visibilityMask = 0x7;
int visibility = mod & visibilityMask;
```

0000 0001 1**010** 0000 0000 0**111**
0000 0000 0**010**

- Flag setzen

```
mod = mod | Modifier.ABSTRACT;
```

0**0**00 0001 1010 0**1**00 0000 0000
0**1**00 0001 1010

Exkurs: Umgang mit Flags

- Flag entfernen

```
mod = mod & ~Modifier.ABSTRACT;
```

0**1**00 0001 1010 1**0**11 1111 1111
0**0**00 0001 1010

- Flag ändern

```
mod = mod ^ Modifier.ABSTRACT;
```

0**0**00 0001 1010 0**1**00 0000 0000
0**1**00 0001 1010

0**1**00 0001 1010 0**1**00 0000 0000
0**0**00 0001 1010

- Flag einer Gruppe umsetzen

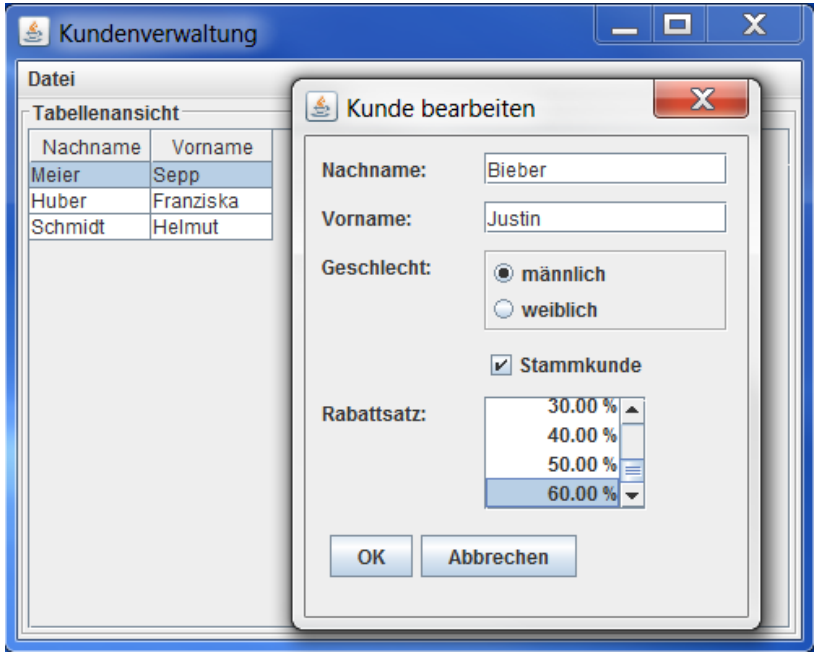
```
mod = mod & ~visibilityMask
      | Modifier.PUBLIC;
```

0000 0001 1**010** 1111 1111 1**000** 0000 0000 0**001**
0000 0001 1**001**

KundenVerwaltung v9

- Aufgabe: Änderungen von Daten sollen mit Hilfe eines Loggers auf der Konsole ausgegeben werden.

Beispiel



Konsolenausgabe nach dem Speichern

```
Nachname von 'Meier' auf 'Bieber' geändert.
Vorname von 'Sepp' auf 'Justin' geändert.
RabattSatz von '0.4' auf '0.6' geändert.
```

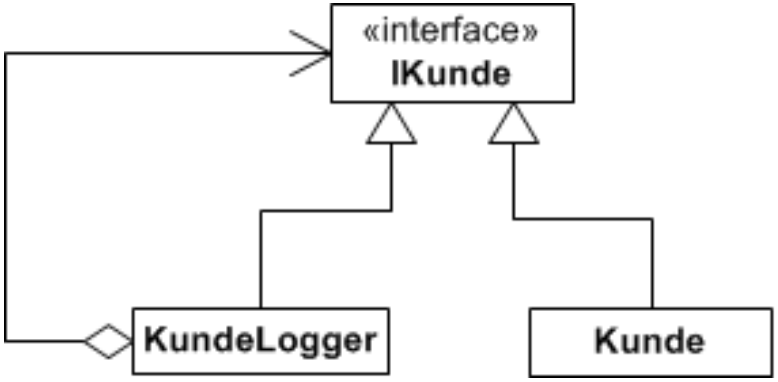
1. Ansatz

```
public void setNachname(String nachname) {

   if (!nachname.equals(getNachname())) {

      System.out.println("Nachname von '"
         + getNachname() + "' auf '"
         + nachname + "' geändert.");
   }

   this.nachname = nachname;
}
```

Bewertung

- Gut, dass die Attribute durch Getter- und Setter-Methoden gekapselt wurden.
- Aber: Separation-of-Concerns
- Also: Logging-Funktionalität in extra Klasse kapseln
- Mit: Decorator-Pattern

2. Ansatz



Schritt 1: Interface extrahieren

```
public interface IKunde {
   int getId();
   void setId(int id);
   String getNachname();
   void setNachname(String nachname);
   String getVorname();
   void setVorname(String vorname);
   Geschlecht getGeschlecht();
   void setGeschlecht(Geschlecht geschlecht);
   boolean isStammkunde();
   void setStammkunde(boolean isstammkunde);
   double getRabattSatz();
   void setRabattSatz(double rabattsatz);
}
```

```
public class Kunde implements IKunde { …
```

Schritt 2: Logger implementieren

```
public class KundeLogger implements IKunde {

   private IKunde kunde;

   public KundeLogger(IKunde kunde) {
      this.kunde = kunde;
   }
   …
   public String getNachname() {
      return kunde.getNachname();
   }

   public void setNachname(String nachname) {

      if (!nachname.equals(getNachname())) {
         System.out.println("Nachname von '" + getNachname()
            + "' auf '" + nachname + "' geändert.");
      }

      kunde.setNachname(nachname);
   } …
}
```

Schritt 3: Logger einbinden

Deco mit Logger

```
public class KundeDbManager {
    …
    public List<IKunde> load() throws SQLException { …

        while (res.next()) {
            IKunde kunde = new Kunde();
            kunde.setId(res.getInt("id"));
            kunde.setNachname(res.getString("nachname"));
            …
            kunde = new KundeLogger(kunde);
            kunden.add(kunde);
        }
        …
        return kunden;
    }
}
```

Bewertung

- Besser, aber sollen wir einen Logger pro Entity (z.B. Kunde) machen?
- Ist das nicht auch Redundanz?
- Logging besitzt eine andere Granularität als eine konkrete Entity-Klasse!

3. Ansatz

- Generischer Logger unter Verwendung von Javas Reflection
- Nötiges Rüstzeug:
  - Dynamic Proxy
  - `Method.invoke()`

Dynamic Proxy

- Die Klasse `Proxy` ermöglicht das Konzept dynamischer Proxys.
- Sie erlaubt zur Laufzeit das Erzeugen von Klassen, welche beliebige Schnittstellen implementieren.
- Eine solcher `Proxy` muss bei seiner Instanziierung mit einem `InvocationHandler` parametrisiert werden.
- Beim Aufruf des Proxys wird dieser generisch an die Methode `invoke()` des `InvocationHandler`s weitergeleitet.
- `InvocationHandler.invoke()` hat folgende Signatur:

```
Object invoke(Object proxy, Method method, Object[] args)
                                                  throws Throwable
```

Method.invoke()

- Die Klasse `Class` hält Methoden in Form eines Arrays vom Typ `Method`.
- `Method` hat eine Methode `invoke()`, um diese generisch aufzurufen.
- Diese hat folgende Signatur:

```
Object invoke(Object obj, Object… args)
```

- Der Parameter `obj` kann `null` sein, wenn es sich um eine statische Methode handelt.

Dynamischer generischer Zugriff

- Die Klasse `Class<?>`

`T newInstance()`	Erzeugt eine Instanz

- Die Klasse `Field`

`Object get(Object obj)`	Gibt den Attributwert zurück
`void set(Object obj, Object value)`	Setzt `value` als Attributwert

- Die Klasse `Constructor<T>`

`T newInstance(Object… initargs)`	Erzeugt eine Instanz

- Die Klasse `Method`

`Object invoke(Object obj, Object… args)`	Ruft die Methode auf

EntityLogger

```
public class EntityLogger implements InvocationHandler {

   private Object entity;

   public EntityLogger(Object entity) {
      this.entity = entity;
   }
   …
   private Method getGetter(String name) {

      String getname = "get" + name;
      String isname = "is" + name;

      for (Method method : entity.getClass().getMethods()) {
         if (method.getName().equals(getname)) return method;
         if (method.getName().equals(isname)) return method;
      }

      return null;
   }
}
```

invoke()
fehlt noch

EntityLogger

```
public class EntityLogger implements InvocationHandler { …

   public Object invoke(Object proxy, Method method, Object[] args)
      throws Exception {

      if (method.getName().startsWith("set")) {
         String fname = method.getName().substring(3);
         Method getter = getGetter(fname);
         Object value = getter.invoke(entity);

         if (!value.equals(args[0])) {
            System.out.println(fname + " von '" + value
             + "' auf '" + args[0] + "' geändert.");
         }
      }

      return method.invoke(entity, args);
   } …
}
```

Logger einbinden

```
public class KundeDbManager {
   …
   public List<IKunde> load() throws SQLException { …

      while (res.next()) {
         IKunde kunde = new Kunde();
         kunde.setId(res.getInt("id"));
         kunde.setNachname(res.getString("nachname"));
         …
         InvocationHandler handler = new EntityLogger(kunde);
         ClassLoader clsloader = ClassLoader.getSystemClassLoader();
         Class<?>[] interfaces = { IKunde.class };
         Object proxy = Proxy.newProxyInstance(
                              clsloader, interfaces, handler);
         kunde = IKunde.class.cast(proxy);

         kunden.add(kunde);
      }
      …
      return kunden;
   }
}
```

Deco mit Logger →

Logger einbinden

```
public class KundeDbManager {
    …
        kunde.setId(res.getInt("id"));
        kunde.setNachname(res.getString("nachname"));
        …
        kunden.add(kunde);
      }
      …
      return kunden;
    }
}
```

Bisherige Version:

```
kunde = new KundeLogger(kunde);
```

```
InvocationHandler handler = new EntityLogger(kunde);
ClassLoader clsloader = ClassLoader.getSystemClassLoader();
Class<?>[] interfaces = { IKunde.class };
Object proxy = Proxy.newProxyInstance(
                    clsloader, interfaces, handler);
kunde = IKunde.class.cast(proxy);
```

Generischer DbManager

- Würden wir weitere Entity-Klassen persistieren wollen, hätte der DbManager immer die gleiche Struktur.
- Das wäre Redundanz?!
- Auch OR-Mapping besitzt eine andere Granularität als eine konkrete Entity-Klasse.
- Reflection für den DbManager?

Strategie

- Feldnamen definieren Namen der get-Methoden ohne das Präfix *get* und beginnend mit einem Kleinbuchstaben, außer dieser wird per Annotation *Column* explizit angegeben.
- Der Tabellenname wird durch den Klassennamen festgelegt soweit nicht durch die Annotation *Entity* anders angegeben.
- Der Primärschlüssel wird durch die Annotation *PrimaryKey* gekennzeichnet.

Annotierter `Kunde`

```
@Entity("kunden")
public class Kunde implements IKunde {
    …
    @PrimaryKey
    public int getId() {
        return id;
    }
    …
    public String getNachname() {
        return nachname;
    }
    …
    @Column(name="isstammkunde")
    public boolean isStammkunde() {
        return isStammkunde;
    }
    …
}
```

Kein expliziter Attributname nötig, da der Standardname *value* verwendet wurde.

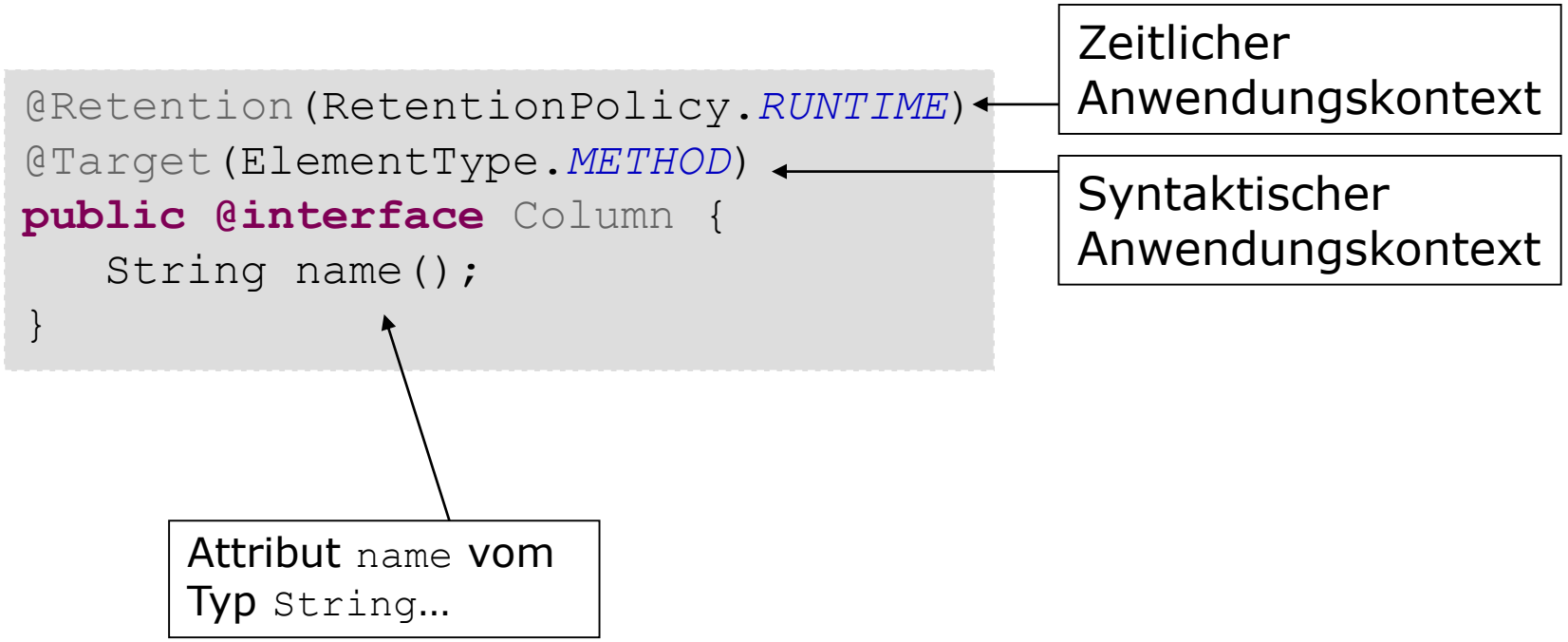
id ist Primärschlüssel

Der Feldname lautet hier *nachname*.

Annotations

- Durch Annotations können Java-Definitionen um zusätzliche Informationen angereichert werden.
- Diese Informationen können per Reflection ausgelesen werden.

Annotation `Column`



Annotation `Entity`

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Entity {
    String value() default "";
}
```

Attributname optional, da *value*

Wert optional, da Defaultwert

Annotation `PrimaryKey`

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface PrimaryKey {
}
```

Keine Attribute zur Auswertung. Stattdessen geht es um die Existenz der Annotation an sich…

Konfigurationsmöglichkeiten

- RetentionPolicy

`SOURCE`	Annotations werden vom Compiler verworfen.
`CLASS`	Annotations gehen in die class-Datei ein, aber müssen nicht zur Laufzeit verfügbar sein (default).
`RUNTIME`	Annotations stehen zur Laufzeit zur Verfügung.

- ElementType

`ANNOTATION_TYPE`	`LOCAL_VARIABLE`	`TYPE`
`CONSTRUCTOR`	`METHOD`	
`FIELD`	`PARAMETER`	

Interface `AnnotatedElement`

- Für den Zugriff auf Annotationen
- Wird u.a. implementiert von `Class`, `Field`, `Constructor` und `Method`.
- Methoden:

`<A extends Annotation> A getAnnotation( Class<A> annotationClass)`	Gibt eine bestimmte Annotation zurück
`Annotation[] getAnnotations()`	Gibt alle Annotationen zurück
`boolean isAnnotationPresent( Class<? extends Annotation> annotationClass)`	Prüft ob eine bestimmte Annotation vorhanden ist

Schritt 1: Interface extrahieren

```
public interface IDbManager<T> {

   void closeConnection();
   List<T> load() throws Exception;
   void save(T entity) throws Exception;
}
```

```
public class KundeDbManager implements IDbManager<IKunde> { …
```

```
public class DbManager<T> implements IDbManager<T> { …
```

Schritt 2: DbManager implementieren

```
public class DbManager<T> implements IDbManager<T> {

   private static class FieldInfo {
      Method getter;
      Method setter;
      String fieldName;
   }

   private static final String DBPATH = "db/kundenverwaltung.mdb";
   private Connection conn;
   private PreparedStatement updateStmt;
   private PreparedStatement selectStmt;

   private String tableName;
   private Class<? extends T> cls;

   private List<FieldInfo> fieldInfos;
   …
```

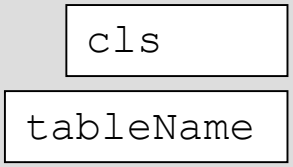
SQL-Statements zum Lesen und Schreiben

OR-Mapping

Schritt 2: DbManager implementieren

- Der Konstruktor initialisiert die Attribute…

```
…
public DbManager(Class<? extends T> cls) throws Exception {

   conn = DriverManager.getConnection("jdbc:ucanaccess://" + DBPATH);
   this.cls = cls;

   if (cls.isAnnotationPresent(Entity.class)) {
      Entity type = cls.getAnnotation(Entity.class);
      tableName = type.value();
   }

   if (tableName == null || tableName.length() == 0) {
      tableName = cls.getSimpleName();
   } …
```



Schritt 2: DbManager implementieren

fieldInfos

```
fieldInfos = new ArrayList<FieldInfo>();
FieldInfo keyfield = null;

Class<?> interfce = cls.getInterfaces()[0];

for (Method method : interfce.getDeclaredMethods()) {

   String methodname = method.getName();
   String prefix;

   if (methodname.startsWith("get")) prefix = "get";
   else if (methodname.startsWith("is")) prefix = "is";
   else continue;

   FieldInfo fi = new FieldInfo();
   fi.getter = method;

   String corename = methodname.substring(prefix.length());
   String settername = "set" + corename;
   fi.setter = interfce.getMethod(settername, method.getReturnType());
```

Schritt 2: DbManager implementieren

```
    Method implmethod = cls.getMethod(methodname, method.getParameterTypes());

    if (implmethod.isAnnotationPresent(Column.class)) {
        Column col = implmethod.getAnnotation(Column.class);
        fi.fieldName = col.name();
    }
    else {
        fi.fieldName = corename.substring(0, 1).toLowerCase() + corename.substring(1);
    }

    if (implmethod.isAnnotationPresent(PrimaryKey.class)) {
        keyfield = fi;
    }
    else {
        fieldInfos.add(fi);
    }
}
```

`fieldInfos`

Schritt 2: DbManager implementieren

```
    StringBuffer cmd = new StringBuffer();
    cmd.append("update " + tableName + " set ");

    boolean first = true;
    for (FieldInfo fi : fieldInfos) {
       if (first) first = false;
       else cmd.append(", ");

       cmd.append(fi.fieldName + " = ? ");
    }

    cmd.append("where " + keyfield.fieldName + " = ?");

    updateStmt = conn.prepareStatement(cmd.toString());

    selectStmt = conn.prepareStatement("select * from " + tableName);

    fieldInfos.add(keyfield);
}
```

updateStmt

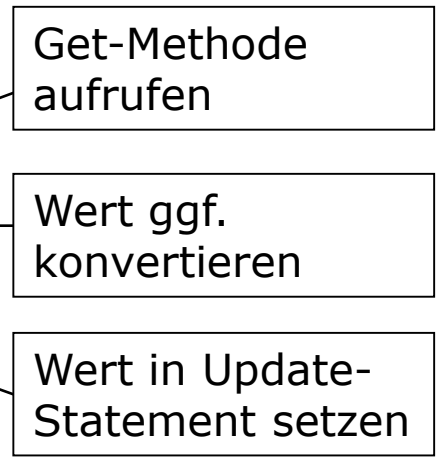
selectStmt

fieldInfos

Schritt 2: DbManager implementieren

- Objekte -> RDB…

```
public void save(T entity) throws Exception {

   for (int i = 0; i < fieldInfos.size(); i++) {
      FieldInfo fi = fieldInfos.get(i);
      Object value = fi.getter.invoke(entity);
      value = convert(value);
      updateStmt.setObject(i + 1, value);
   }

   updateStmt.executeUpdate();
}
```



Schritt 2: DbManager implementieren

```
private static Object convert(Object value) throws Exception {

    if (value instanceof Enum) {
        Enum<?> e = (Enum<?>)value;
        return e.ordinal();
    }
    else {
        return value;
    }
}
```

Objekte werden ggf. für die Speicherung in der DB umgewandelt.

Schritt 2: DbManager implementieren

- RDB -> Object… (zunächst die Konvertierungsfunktion)

```
private static Object convert(Class<?> type, Object value)
                                                throws Exception {
   if (type.isEnum()) {
      Integer index = (Integer)value;
      Method method = type.getDeclaredMethod("values");
      Object arr = method.invoke(null);
      Object eval = Array.get(arr, index);
      return eval;
   }
   else {
      return value;
   }
}
```

Objekte werden ggf. für die Anwendung umgewandelt.

Schr

Wert aus `ResultSet` lesen

Wert ggf. konvertieren

Set-Methode aufrufen

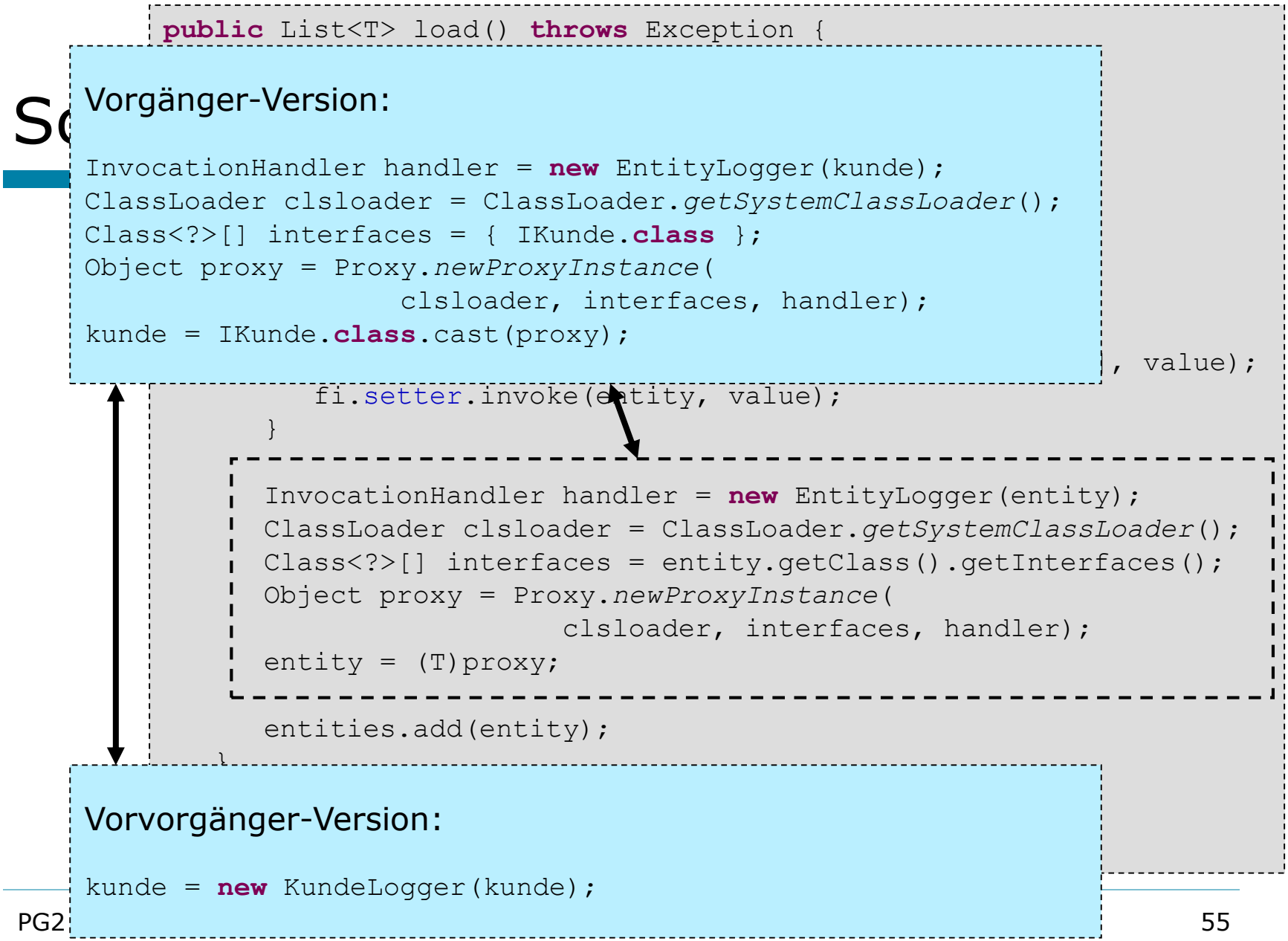
```
public List<T> load() throws Exception {

   ResultSet res = selectStmt.executeQuery();
   List<T> entities = new ArrayList<T>();

   while (res.next()) {
      T entity = cls.newInstance();

      for (FieldInfo fi : fieldInfos) {
         Object value = res.getObject(fi.fieldName);
         value = convert(fi.setter.getParameterTypes()[0], value);
         fi.setter.invoke(entity, value);
      }

      InvocationHandler handler = new EntityLogger(entity);
      ClassLoader clsloader = ClassLoader.getSystemClassLoader();
      Class<?>[] interfaces = entity.getClass().getInterfaces();
      Object proxy = Proxy.newProxyInstance(
                        clsloader, interfaces, handler);
      entity = (T)proxy;

      entities.add(entity);
   }

   return entities;
}
```

Schr

```
public List<T> load() throws Exception {

   ResultSet res = selectStmt.executeQuery();
   List<T> entities = new ArrayList<T>();

   while (res.next()) {
      T entity = cls.newInstance();
      for (FieldInfo fi : fieldInfos) {
         Object value = res.getObject(fi.fieldName);
         value = convert(fi.setter.getParameterTypes()[0], value);
         fi.setter.invoke(entity, value);
      }
      InvocationHandler handler = new EntityLogger(entity);
      ClassLoader clsloader = ClassLoader.getSystemClassLoader();
```

Bisherige Version:

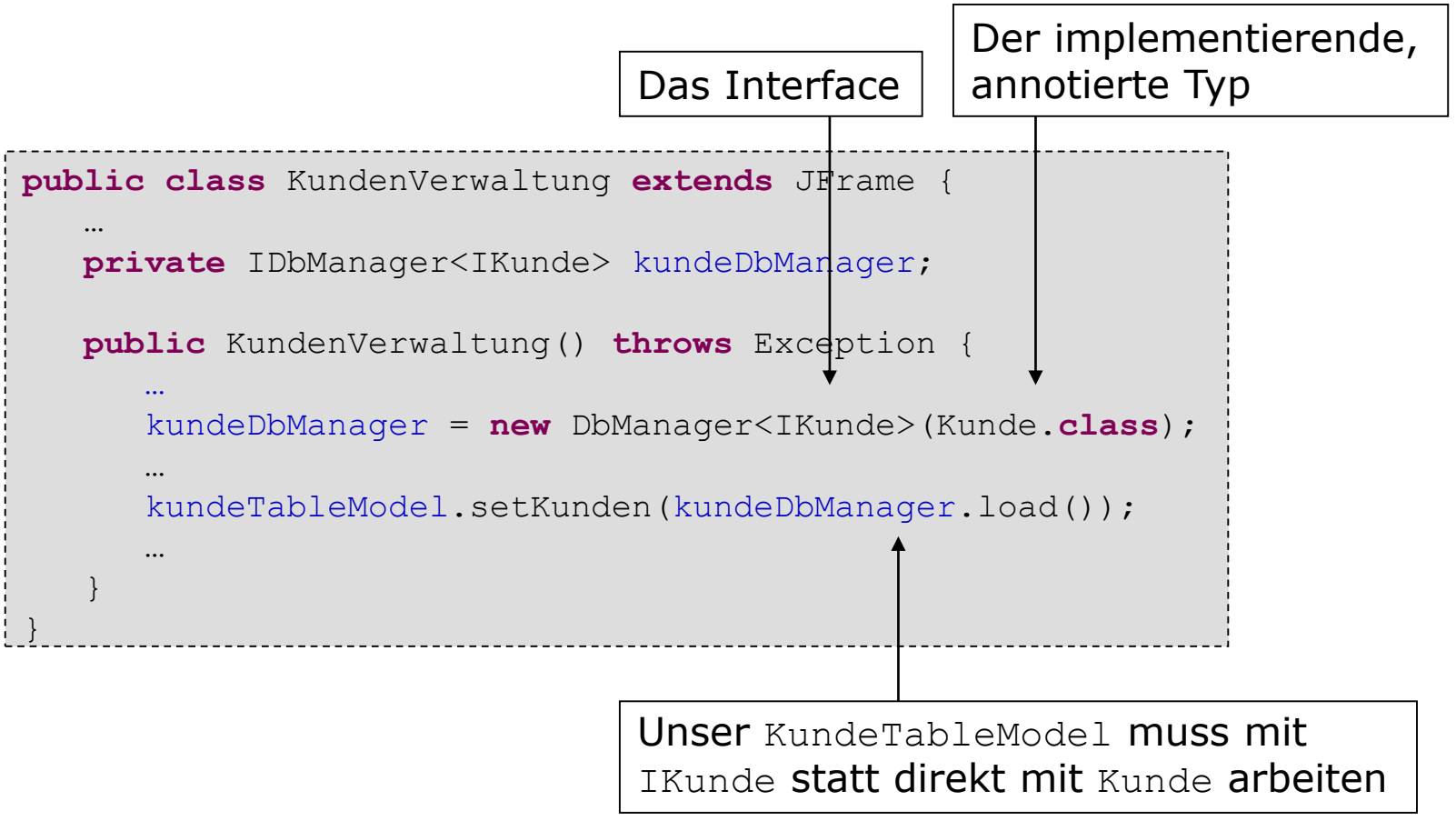
```
kunde.setId(res.getInt("id"));
kunde.setNachname(res.getString("nachname"));
kunde.setVorname(res.getString("vorname"));
kunde.setGeschlecht(Geschlecht.values()[res.getInt("geschlecht")]);
kunde.setStammkunde(res.getBoolean("isstammkunde"));
kunde.setRabattSatz(res.getDouble("rabattsatz"));
```


public List<T> load() throws Exception {
Vorgänger-Version:
InvocationHandler handler = new EntityLogger(kunde);
ClassLoader clsloader = ClassLoader.getSystemClassLoader();
Class<?>[] interfaces = { IKunde.class };
Object proxy = Proxy.newProxyInstance(
clsloader, interfaces, handler);
kunde = IKunde.class.cast(proxy);
, value);
fi.setter.invoke(entity, value);
}
InvocationHandler handler = new EntityLogger(entity);
ClassLoader clsloader = ClassLoader.getSystemClassLoader();
Class<?>[] interfaces = entity.getClass().getInterfaces();
Object proxy = Proxy.newProxyInstance(
clsloader, interfaces, handler);
entity = (T)proxy;
entities.add(entity);
Vorvorgänger-Version:
kunde = new KundeLogger(kunde);

Schritt 3: DbManager einbinden

```
public class KundenVerwaltung extends JFrame {
    …
    private IDbManager<IKunde> kundeDbManager;

    public KundenVerwaltung() throws Exception {
        …
        kundeDbManager = new DbManager<IKunde>(Kunde.class);
        …
        kundeTableModel.setKunden(kundeDbManager.load());
        …
    }
}
```

Schritt 3: DbManager einbinden



Schritt 3: DbManager einbinden

```
public class KundenVerwaltung extends JFrame {
    …
    private IDbManager<IKunde> kundeDbManager;

    public KundenVerwaltung() throws Exception {
      …
      kundeDbManager = new DbManager<IKunde>(Kunde.class);
      kundeTableModel.setKunden(kundeDbManager.load());
      …
    }
}
```

Bisherige Version:

```
kundeDbManager = new KundeDbManager();
```

Schr

```
public List<T> load() throws Exception {

   ResultSet res = selectStmt.executeQuery();
   List<T> entities = new ArrayList<T>();

   while (res.next()) {
      T entity = cls.newInstance();

      for (FieldInfo fi : fieldInfos) {
         Object value = res.getObject(fi.fieldName);
         value = convert(fi.setter.getParameterTypes()[0], value);
         fi.setter.invoke(entity, value);
      }

      InvocationHandler handler = new EntityLogger(entity);
      ClassLoader clsloader = ClassLoader.getSystemClassLoader();
      Class<?>[] interfaces = entity.getClass().getInterfaces();
      Object proxy = Proxy.newProxyInstance(
                        clsloader, interfaces, handler);
      entity = (T)proxy;

      entities.add(entity);
   }

   return entities;
}
```

Passt das so?

Schr

```
public List<T> load() throws Exception {

   ResultSet res = selectStmt.executeQuery();
   List<T> entities = new ArrayList<T>();

   while (res.next()) {
      T entity = cls.newInstance();

      for (FieldInfo fi : fieldInfos) {
         Object value = res.getObject(fi.fieldName);
         value = convert(fi.setter.getParameterTypes()[0], value);
         fi.setter.invoke(entity, value);
      }

      InvocationHandler handler = new EntityLogger(entity);
      ClassLoader clsloader = ClassLoader.getSystemClassLoader();
      Class<?>[] interfaces = entity.getClass().getInterfaces();
      Object proxy = Proxy.newProxyInstance(
                        clsloader, interfaces, handler);
      entity = (T)proxy;

      entities.add(entity);
   }

   return entities;
}
```

Passt nicht rein…

Bewertung

- Gut: Der DbManager kann nun mit beliebigen Entity-Klassen umgehen.
- Aber: Er ist eng an den Logger gekoppelt. Der Logger ist nur *irgendein* Decorator…
- Separation-of-Concerns…
- Lösung: Außerhalb des DbManagers dekorieren!

Schritt 2, Version 2

```
public List<T> load() throws Exception {

   ResultSet res = selectStmt.executeQuery();
   List<T> entities = new ArrayList<T>();

   while (res.next()) {
      T entity = cls.newInstance();

      for (FieldInfo fi : fieldInfos) {
         Object value = res.getObject(fi.fieldName);
         value = convert(fi.setter.getParameterTypes()[0], value);
         fi.setter.invoke(entity, value);
      }

      entities.add(entity);
   }

   return entities;
}
```

Besser…

Schritt 3, Version 2

```
public KundenVerwaltung() throws Exception {
   …
   kundeDbManager = new DbManager<IKunde>(Kunde.class);
   List<IKunde> kunden = kundeDbManager.load();

   for (int i = 0; i < kunden.size(); i++) {
      IKunde entity = kunden.get(i);

      InvocationHandler handler = new EntityLogger(entity);
      ClassLoader clsloader = ClassLoader.getSystemClassLoader();
      Class<?>[] interfaces = entity.getClass().getInterfaces();
      Object proxy = Proxy.newProxyInstance(
                        clsloader, interfaces, handler);
      entity = (IKunde)proxy;

      kunden.set(i, entity);
   }

   kundeTableModel.setKunden(kunden);
   …
}
```

Schritt 3, Version 2

Passt nicht rein…

```
public KundenVerwaltung() throws Exception {
   …
   kundeDbManager = new DbManager<IKunde>(Kunde.class);
   List<IKunde> kunden = kundeDbManager.load();
   for (int i = 0; i < kunden.size(); i++) {
      IKunde entity = kunden.get(i);

      InvocationHandler handler = new EntityLogger(entity);
      ClassLoader clsloader = ClassLoader.getSystemClassLoader();
      Class<?>[] interfaces = entity.getClass().getInterfaces();
      Object proxy = Proxy.newProxyInstance(
                        clsloader, interfaces, handler);
      entity = (IKunde)proxy;

      kunden.set(i, entity);
   }
   kundeTableModel.setKunden(kunden);
   …
}
```

Bewertung

- Gut: Der DbManager ist jetzt entkoppelt vom Logger.
- Aber: Das Umwickeln mit Loggern passt auch nicht so recht in KundenVerwaltung…
- Separation-of-Concerns…
- Lösung: LoggerWrapper

LoggerWrapper

```
public class LoggerWrapper<T> implements IDbManager<T> {

   private IDbManager<T> base;

   public LoggerWrapper(IDbManager<T> base) {
      this.base = base;
   }

   public void closeConnection() {
      base.closeConnection();
   }

   public void save(T entity) throws Exception {
      base.save(entity);
   }
   …
}
```

Wird dekoriert…

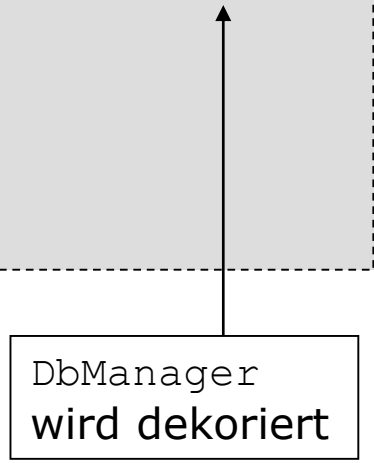
LoggerWrapper

```
public class LoggerWrapper<T> implements IDbManager<T> { …

   public List<T> load() throws Exception {

      List<T> entities = base.load();

      for (int i = 0; i < entities.size(); i++) {
         T entity = entities.get(i);

         InvocationHandler handler = new EntityLogger(entity);
         ClassLoader clsloader = ClassLoader.getSystemClassLoader();
         Class<?>[] interfaces = entity.getClass().getInterfaces();
         Object proxy = Proxy.newProxyInstance(
                              clsloader, interfaces, handler);
         entity = (T)proxy;

         entities.set(i, entity);
      }

      return entities;
   }
}
```

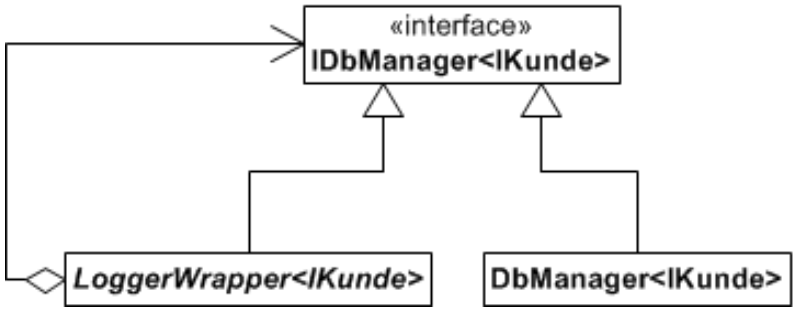
Liste von `EntityLogger`n

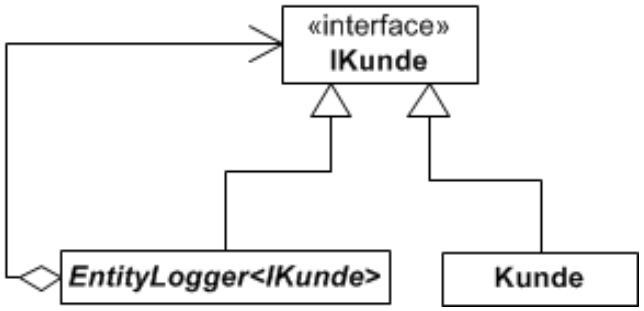
LoggerWrapper einbinden

```
public KundenVerwaltung() throws Exception {
    …
    kundeDbManager = new DbManager<IKunde>(Kunde.class);
    kundeDbManager = new LoggerWrapper<IKunde>(kundeDbManager);

    List<IKunde> kunden = kundeDbManager.load();

    kundeTableModel.setKunden(kunden);
    …
}
```



Architektur

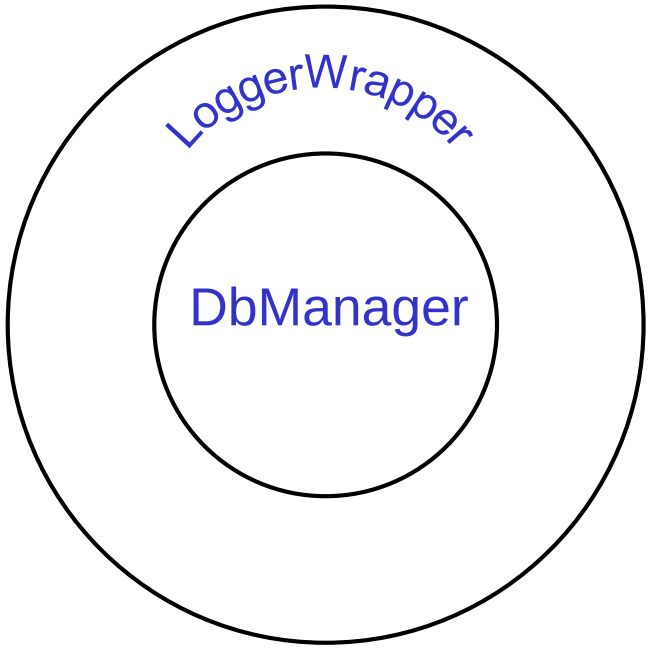


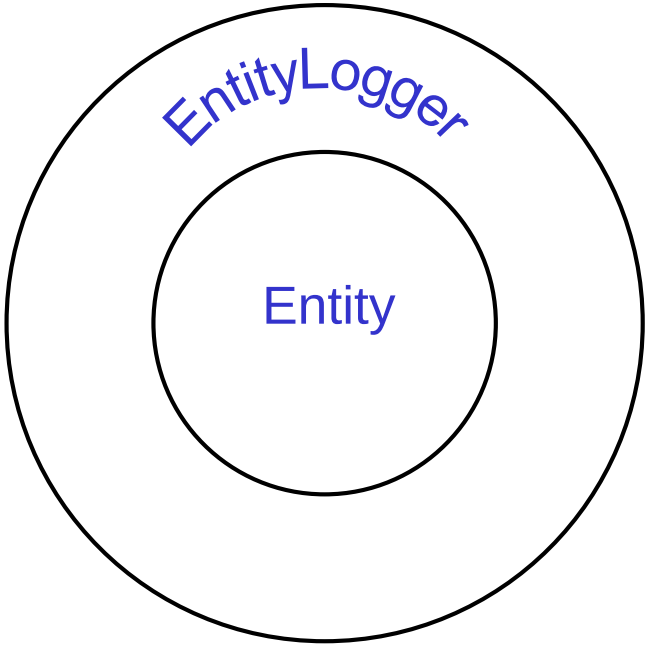


- `DbManager` wird durch `LoggerWrapper` dekoriert
- `LoggerWrapper` ist an `IDbManager` gebunden

- `Kunde` wird durch `EntityLogger` dekoriert
- `EntityLogger` ist eigentlich unabhängig von `IKunde`

Architektur





- Die Kommunikation zum `DbManager` verläuft über den `LoggerWrapper`
- Die Kommunikation zur `Entity` verläuft über den `EntityLogger`

Aspektorientierte Programmierung

- Wir haben Objekte um den Aspekt *Logging* erweitert.
- Im Allgemeinen könnten wir Objekte um mehrere *Aspekte* erweitern wollen.
- Wir wollen dabei die Kommunikation mit einem Objekt überwachen und ggf. auch manipulieren (z.b. Autorisierungs-Aspekt).