Lösungs*vorschläge* zu den Staatsexamina:
Softwaretechnologie und Datenbanken

Staatsexamen vom Herbst 2019

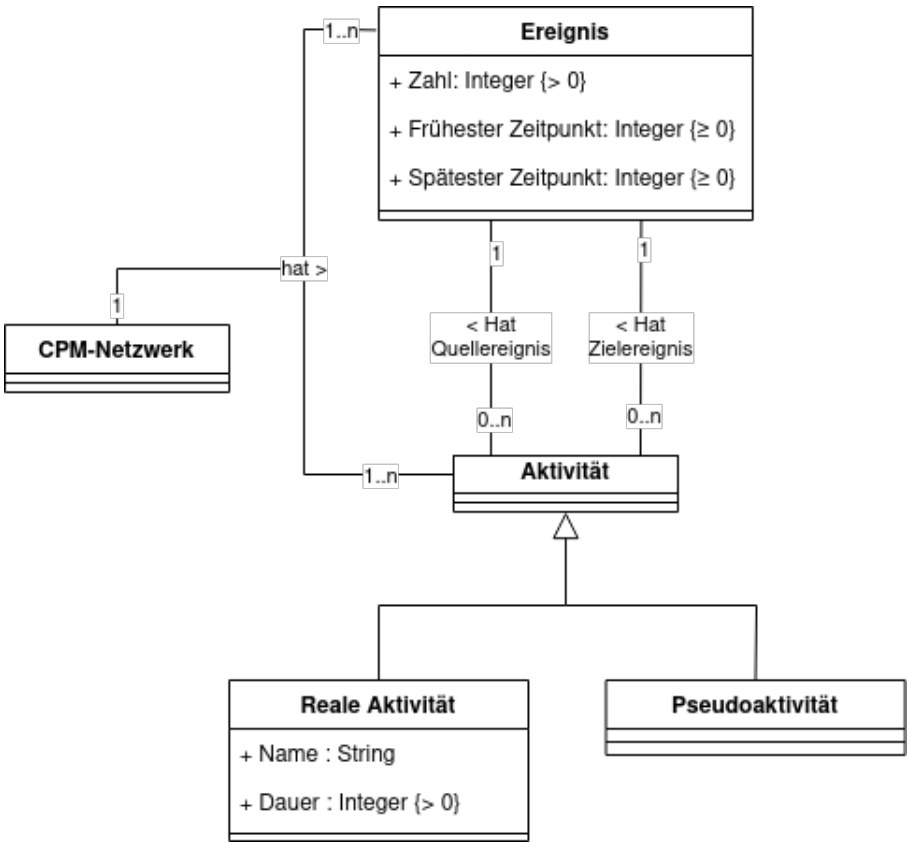
Inhaltsverzeichnis

1 Thema Nr. 1

1.1 Teilaufgabe 1: Softwaretechnologie

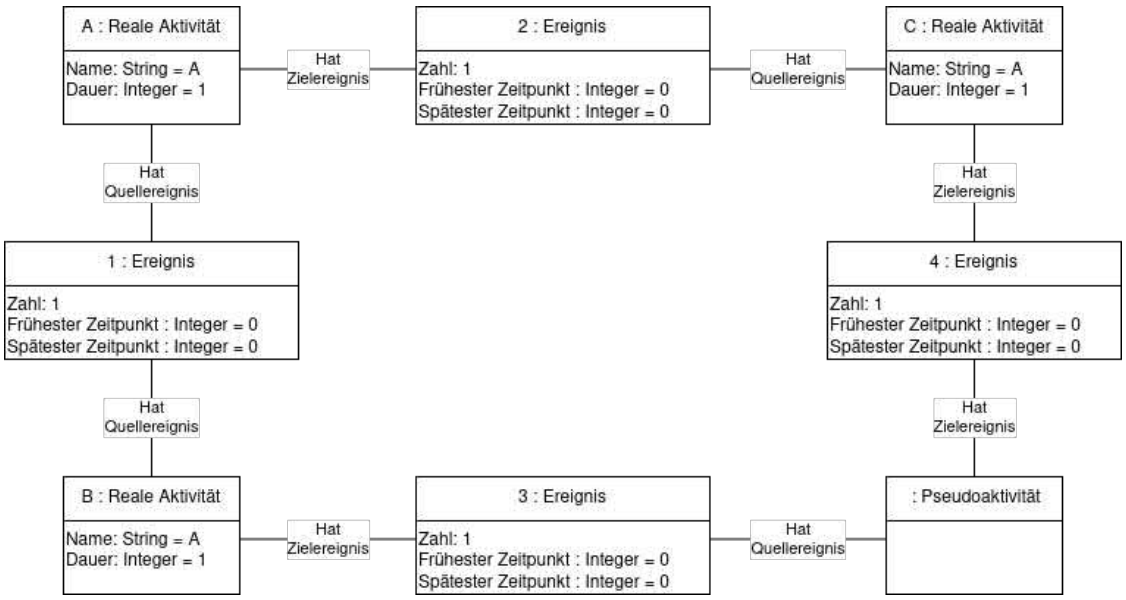
1.1.1 Aufgabe 1: UML-Klassen- und Objektdiagramme

(a) UML-Klassendiagramm:



Erklärungen zum UML-Klassendiagramm
Die Klassen *Reale Aktivität* und *Pseudoaktivität* sind hier implementiert als Unterklassen der Klasse textitAktivität. Diese besitzt keinerlei Attribute oder Methoden. So lässt sich die Beziehung, die hier zwischen Aktivitäten und Ereignissen besteht, kompakter darstellen, da andernfalls beide Unterklassen (*Reale Aktivität* und *Pseudoaktivität*) separat Beziehungen eingehen müssten mit der Klasse *Ereignis*.
Die Sichtbarkeiten waren hier streng genommen nicht gefordert, wurden der Vollständigkeit halber aber ergänzt.

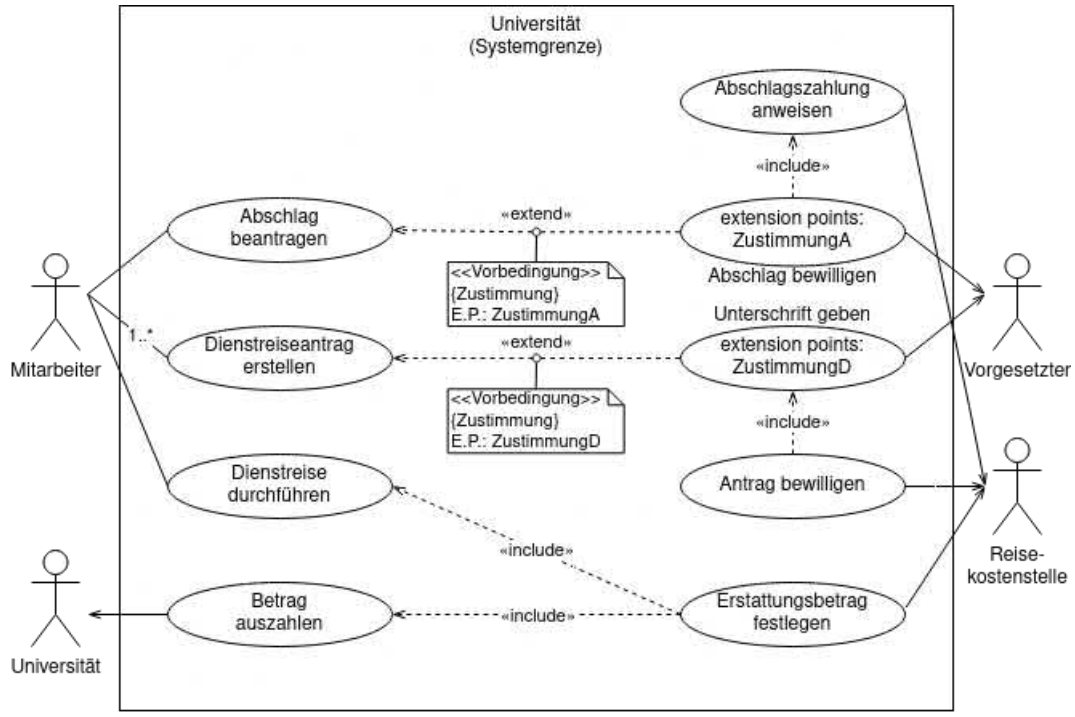
(b) UML-Objektdiagramm:



Erklärungen zum UML-Objektdiagramm
Hier zeigt sich eine konkrete Instanziierung des obigen Klassendiagramms.
Da eine Pseudoaktivität keinen Namen hat, bleibt der Platz vor dem Doppelpunkt im Bezeichner leer.

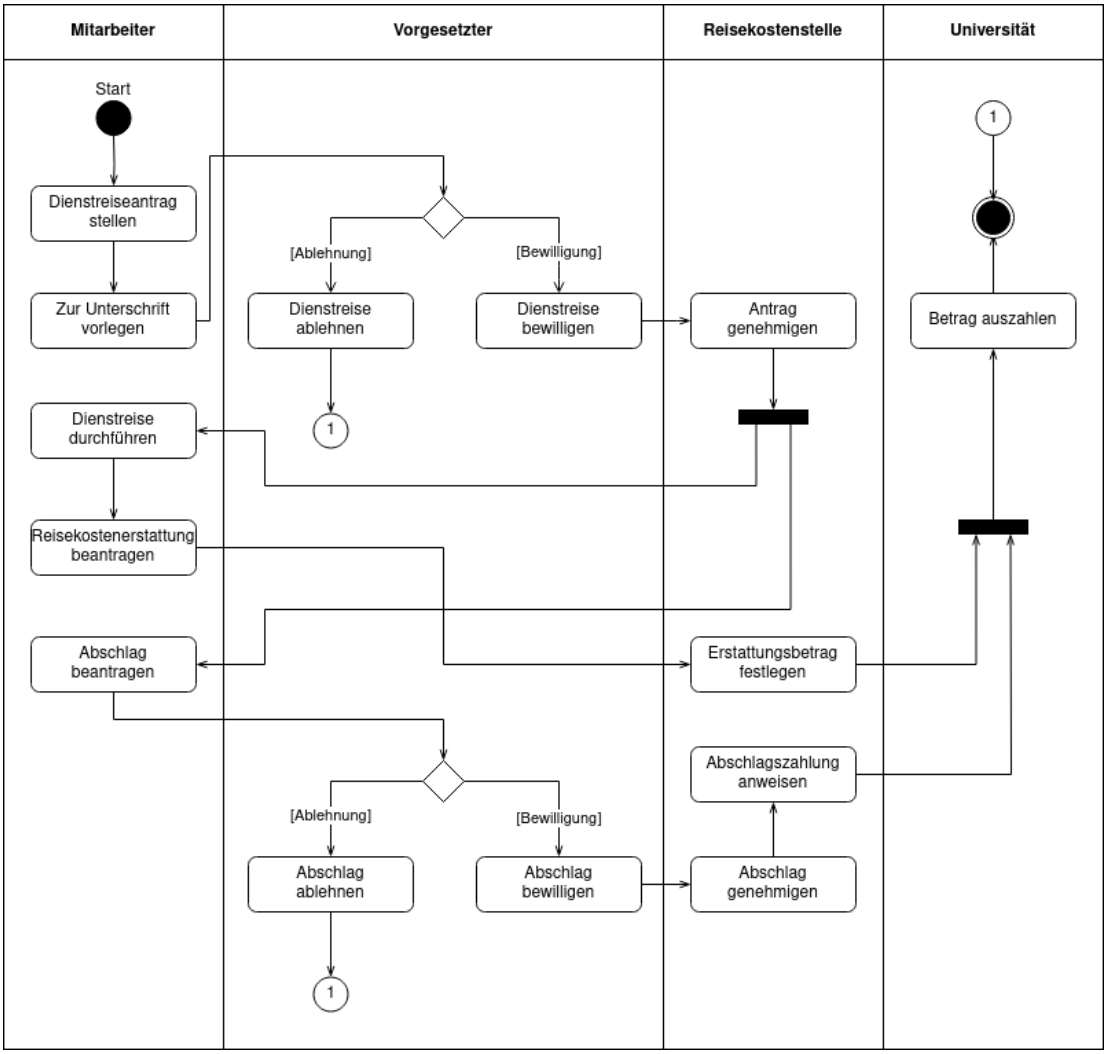
1.1.2 Aufgabe 2: UML-Anwendungsfall- und Aktivitätsdiagramme

(a) UML-Anwendungsfalldiagramm:



Erklärungen zum Anwendungsfalldiagramm
In diesem Diagramm finden sich mehrere Include-Beziehungen. Eine Include-Bezie-hung modelliert die unbedingte Einbindung der Funktionalität eines Anwendungsfalls in einen anderen. Das bedeutet konkret: jedes Mal, wenn der einbindende Anwendungsfall aufgeführt wird, muss auch der eingebundene Anwendungsfall aufgerufen werden. Damit ist die Include-Beziehung also vergleichbar mit dem Aufruf einer Unterfunktion.
In diesem Diagramm folgen viele der Ereignisse zwingend aufeinander. Bewilligt etwa der Vorgesetzte eine Dienstreise, wird dieser Antrag zwangsläufig der Reisekostenstelle zur Genehmigung vorgelegt. Und so verhält es sich mit vielen weiteren Beziehungen in diesem Diagramm.
Das Geben der Unterschrift und die Bewilligung des Reisekostenabschlags sind allerdings bedingte Anwendungsfälle, die von der Zustimmung des Vorgesetzten abhängen. Daher benötigen wir hier Extend-Beziehungen, die die Modellierung eines bedingten Anwendungsfalls ermöglichen.

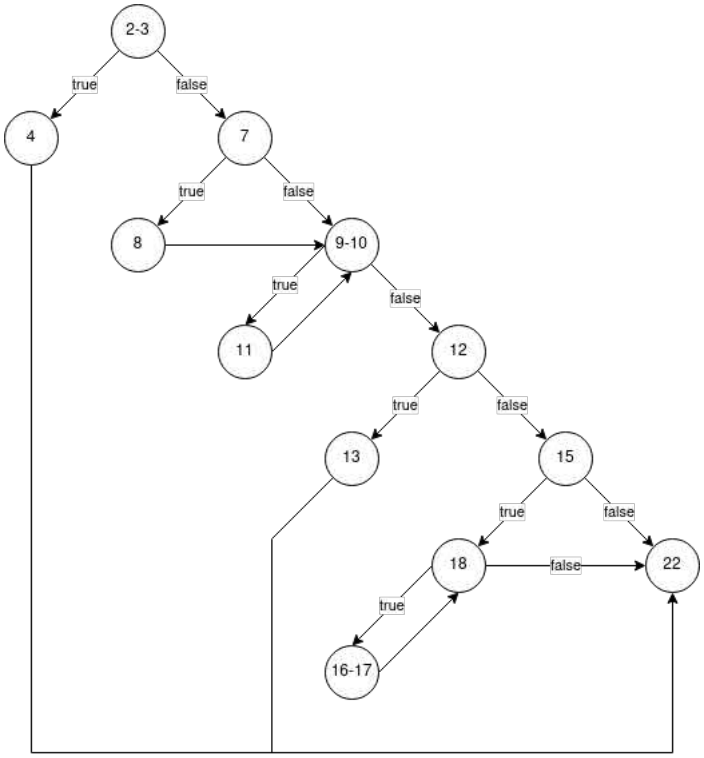
(b) UML-Aktivitätsdiagramm:



Erklärungen zum Aktivitätsdiagramm
An dem hier modellierten Szenario sind mehrere Akteure beteiligt. Daher benötigen wir mehrere Spalten, eine je Akteur. An zwei Stellen in dem Diagramm arbeiten wir mit Verzweigungsknoten, nämlich wenn aufgrund von einer Entscheidungssituation nur eine von beiden Folgeaktivitäten möglich ist. Außerdem nutzen wir Parallelisierungs- und Synchronisationsknoten, um gewisse Abläufe zu bündeln.
Grundsätzlich gibt es je Aktivitätsdiagramm nur einen Start- und einen Endpunkt. Daher nutzen wir hier Sprungmarken zum Endpunkt an Stellen, an denen der Vorgang bereits an frührerer Stelle abgebrochen werden muss.

1.1.3 Aufgabe 3: White-Box-Tests

(a)



(b) Die Knotenüberdeckung entspricht der Anweisungsüberdeckung (Englisch: *Statement Coverage*). Bei dieser Art von Testverfahren müssen alle *Anweisungen* mindestens einmal durchlaufen werden. Dies erreichen wir mit folgender Menge an Testläufen:

3 → 4 → 22
3 → 7 → 8 → 10 → 11 → 10 → 12 → 13 → 22
3 → 7 → 10 → 12 → 15 → 18 → 16 → 22

(c) Die Kantenüberdeckung entspricht der Verzweigungsüberdeckung (Englisch: *Branch Coverage*). Bei dieser Art von Testverfahren müssen alle *Zweige* mindestens einmal durchlaufen werden. Dies erreichen wir mit folgender Menge an Testläufen:

3 → 4 → 22
3 → 7 → 8 → 10 → 11 → 10 → 12 → 13
3 → 7 → 10 → 12 → 15 → 18 → 16 → 18 → 22
3 → 7 → 10 → 12 → 15 → 22

(d) Die Verzweigungsüberdeckung ist das strengere, umfangreichere Testverfahren. Sie *umfasst* die Anweisungsüberdeckung, da jede Verzweigung mindestens eine zugehörige Anweisung enthält.

1.1.4 Aufgabe 4: Entwurfsmuster

Das Entwurfsmuster Zustand ist hier ja bereits vorgegeben. Es wird eingesetzt zur Kapselung unterschiedlicher, zustandsabhänigiger Verhaltensweisen eines Objektes. Der Kontext ist hier eine Prozessverwaltung. Anfangs befinden wir uns im Zustand *Bereit*. Die Ausgaben für die Zustandsübergänge sind eigentlich nicht gefordert, hier aber implementiert aus Liebe zum Detail.

```
1  public class Prozessverwaltung {
2      Zustand aktuellerZustand ;
3
4      Zustand bereit ;
5      Zustand aktiv ;
6      Zustand suspendiert ;
7      Zustand beendet ;
8      Zustand abgebrochen ;
9
10     public Prozessverwaltung () {
11         this . bereit = new Bereit ( this ) ;
12         this . aktiv = new Aktiv ( this );
13         this . suspendiert = new Suspendiert ( this );
14         this . beendet = new Beendet ( this );
15         this . abgebrochen = new Abgebrochen ( this );
16
17         this . aktuellerZustand = bereit ;
18     }
19
20     public void setzPZustand ( Zustand z ) {
21         this . aktuellerZustand = z;
22     }
23
24     public void starten () {
           aktuellerZustand . starten () ;}
25     public void suspendieren () {
           aktuellerZustand . suspendieren () ;}
26     public void fortsetzen () {
           aktuellerZustand . fortsetzen () ;}
27     public void abbrechen () {
           aktuellerZustand . abbrechen () ;}
28     public void beenden () {
           aktuellerZustand . beenden () ;}
29
30     public Zustand getBereit () { return bereit ;}
31     public Zustand getAktiv () { return aktiv ;}
32     public Zustand getSuspendiert () { return suspendiert ;}
33     public Zustand getAbgebrochen () { return abgebrochen ;}
34     public Zustand getBeendet () { return beendet ;}
```

```
35 }
36
37 abstract class Zustand {
38     public void starten () {
39         System . out . println ( " Fehler !") ;
40     }
41     public void suspendieren () {
42         System . out . println ( " Fehler !") ;
43     }
44     public void fortsetzen () {
45         System . out . println ( " Fehler !") ;
46     }
47     public void abbrechen () {
48         System . out . println ( " Fehler !") ;
49     }
50     public void beenden () {
51         System . out . println ( " Fehler !") ;
52     }
53 }
54
55 class Bereit extends Zustand {
56     Prozessverwaltung p;
57
58     Bereit ( Prozessverwaltung p) {
59         this .p = p;
60     }
61
62     public void starten () {
63         p. setzPZustand (p. getAktiv () );
64         System . out . println ( " Wechsle in Zustand Aktiv ." );
65     }
66 }
67
68 class Aktiv extends Zustand {
69     Prozessverwaltung p;
70
71     Aktiv ( Prozessverwaltung p) {
72         this .p = p;
73     }
74
75     public void suspendieren () {
76         p. setzPZustand (p. getSuspendiert () ) ;
77         System . out . println ( " Wechsle in Zustand Suspendiert . ");
78     }
79     public void beenden () {
80         p. setzPZustand (p. getBeendet () );
81         System . out . println ( " Wechsle in Zustand Beendet .") ;
```

```
82      }
83  }
84
85  class  Suspendiert  extends  Zustand  {
86      Prozessverwaltung    p;
87
88      Suspendiert ( Prozessverwaltung     p)  {
89          this .p   = p;
90      }
91
92      public   void  fortsetzen ()    {
93          p. setzPZustand (p. getAktiv () );
94          System . out . println ( " Wechsle  in  Zustand  Aktiv ." );
95      }
96      public   void  abbrechen ()  {
97          p. setzPZustand (p. getAbgebrochen () ) ;
98          System . out . println ( " Wechsle  in  Zustand  Abgebrochen . ");
99      }
100 }
101
102 class  Beendet  extends  Zustand  {
103     Prozessverwaltung    p;
104
105     Beendet ( Prozessverwaltung   p ) {
106         this .p   = p;
107     }
108 }
109
110 class  Abgebrochen  extends  Zustand  {
111     Prozessverwaltung    p;
112
113     Abgebrochen ( Prozessverwaltung   p)  {
114         this .p   = p;
115     }
116 }
```

1.2 Teilaufgabe 2: Datenbanken

1.2.1 Aufgabe 1: Wissensfragen

1. **Schichtenarchitektur**

   Vorteil: Modularität und Flexibilität
   Eine Schichtenarchitektur ermöglicht es, das Datenbanksystem aufzuteilen in verschiedene Schichten, wobei jede Schicht bestimmte Funktionen und Verantwortlichkeiten hat. Dadurch kann jede Schicht unabhängig voneinander entwickelt, gewartet und ausgetauscht werden. Diese Modularität und Flexibilität erleichtert die Skalierbarkeit und den Wartungsaufwand und ermöglicht es, neue Funktionen oder Technologien in einer bestimmten Schicht einzuführen, ohne das gesamte System neu entwickeln zu müssen.

   Nachteil: Leistungseinbußen
   Die Verwendung einer Schichtenarchitektur kann zu einer erhöhten Komplexität und einer höheren Verarbeitungszeit führen. Jede Schicht, durch die eine Anfrage oder eine Änderung in der Datenbank gehen muss, erfordert zusätzliche Verarbeitungsschritte. Dies kann zu Leistungseinbußen führen, insbesondere wenn die Schichten ineffizient implementiert oder miteinander kommuniziert werden. Bei sehr großen und komplexen Datenbanksystemen kann dies zu Engpässen führen und die Leistung negativ beeinflussen.

2. Eine Sicht bezeichnet einen logischen Einblick in die Datenbank, die auf einer oder mehreren Tabellen basiert. Sie kann bestimmte Spalten auswählen, Zeilen filtern oder berechnete Spalten enthalten. Benutzer können dann auf sie zugreifen als ob sie eine physische Tabelle wäre, und die in ihr Daten anzeigen oder verändern.

   Ermöglicht wird dies durch zwei Dinge:

   1. Zugriffsrechte und Berechtigungen
      Das Datenbanksystem ermöglicht es den Datenbankadministratoren, den Benutzern unterschiedliche Zugriffsrechte auf Tabellen oder Spalten der Datenbank zuzuweisen. Durch die Definition von Berechtigungen kann der Administrator steuern, welche Daten die Benutzer sehen und welche Aktionen sie auf den Daten ausführen können. Dadurch können verschiedene Benutzergruppen je nach ihren Berechtigungen unterschiedliche Sichten auf die Datenbank haben.
   2. Abfragesprachen
      Das Datenbanksystem unterstützt Abfragesprachen wie SQL. Diese ermöglicht es den Benutzern die Ausführung komplexer Abfragen und eine Weiterverwendung der Anfrageergebnisse nach eigenen Anforderungen. Durch die Verwendung von Abfrageoperationen wie Projektion, Selektion, Verbund usw. können Benutzer spezifische Sichten auf die Datenbank erstellen, indem sie bestimmte Daten extrahieren, filtern oder verknüpfen.

3. Das Konzept der transitiven Hülle wird verwendet, um die transitive Beziehung zwischen Elementen zu erfassen. Es erweitert eine gegebene Relation um alle indirekten Bezie-

hungen, die sich aus den vorhandenen direkten Beziehungen ergeben. Die transitiv geschlossene Relation enthält somit alle möglichen Verbindungen zwischen den Elementen.

Transitive Hülle eines Attributes bei funktionalen Abhängigkeiten
Angenommen, wir haben eine Menge funktionaler Abhängigkeiten über Attributen in einer Relation. Die transitiv geschlossene Hülle eines Attributes ist die Menge aller Attribute, die von diesem funktional abhängig sind; entweder direkt oder indirekt über andere Attribute.

Transitive Hülle einer SQL-Abfrage
Die Transitive Hülle einer SQL-Abfrage bezieht sich auf die Erweiterung der Ergebnismenge einer Abfrage um alle indirekten Ergebnisse, die sich aus den vorhandenen direkten Ergebnissen ergeben.

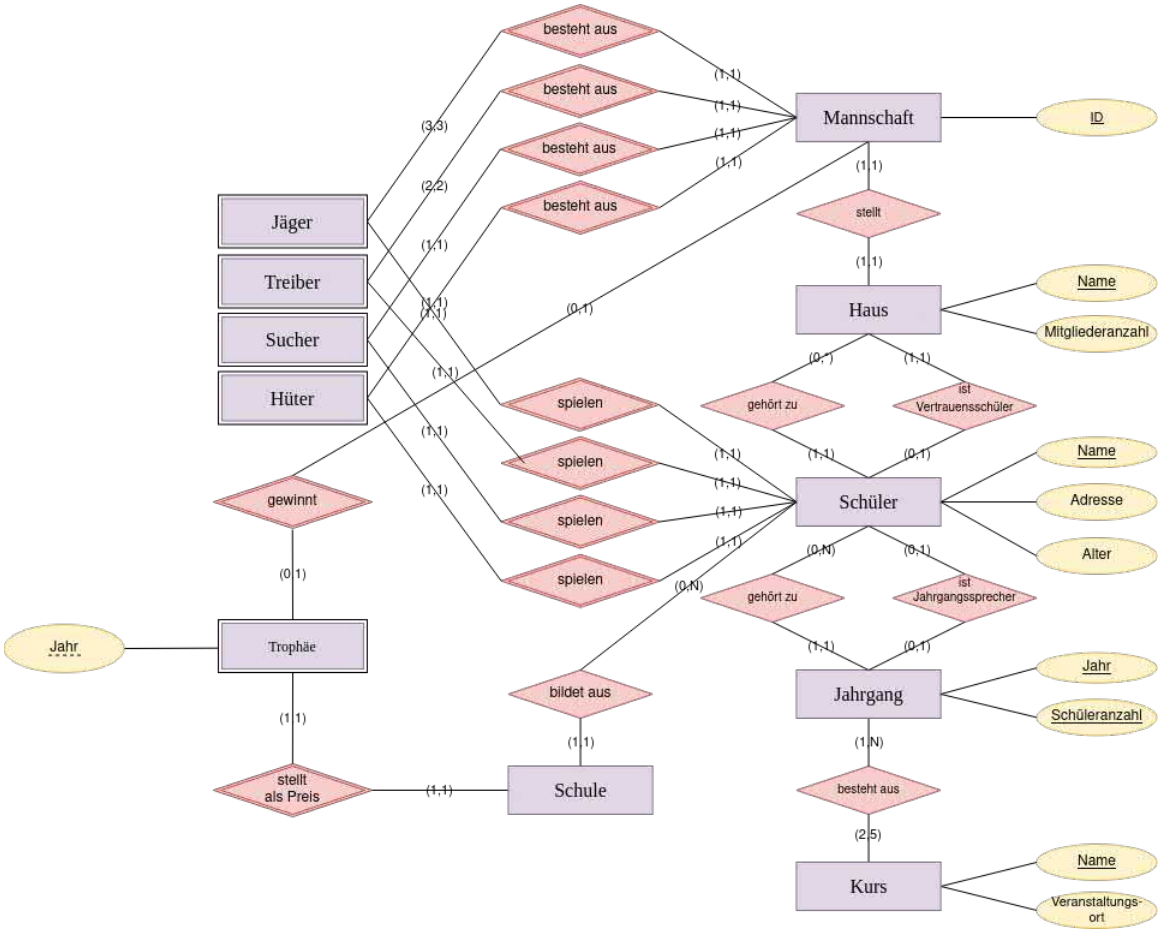
4. **B-Baum**
Der B-Baum ist eine weit verbreitete Indexstruktur, die in relationalen Datenbanken häufig verwendet wird.

Vorteil
In einem B-Baum kann ein Knoten – im Unterschied zu Binärbäumen – mehr als zwei Kind-Knoten haben. Dies ermöglicht es, mit einer variablen Anzahl an Schlüsseln (oder Datenwerten) je Knoten die Anzahl der bei einer Datensuche zu lesenden Knoten zu verringern. Die höchstens erlaubte Anzahl der Schlüssel ist abhängig von einem Parameter k, dem Verzweigungsgrad (oder Ordnung) des B-Baumes. Jeder Knoten hat im B-Baum mindestens k und höchstens $2k$ Einträge (außer der Wurzel; diese enthält $1\text{-}2k$ Einträge.

Streuwerttabelle
Eine Streuwerttabelle verwendet eine Streufunktion, um die Indexeinträge direkt in einem Array abzubilden. Jeder Schlüsselwert wird umgewandelt in einen Streuwert, der dann verwendet wird als Adresse für den Zugriff auf den entsprechenden Datensatz.

Vorteil
Gegenüber alternativen Index-Datenstrukturen wie Baumstrukturen zeichnen sich Hashtabellen üblicherweise aus durch einen konstanten Zeitaufwand bei Einfüge- bzw. Entfernen-Operationen.

1.2.2 Aufgabe 2: ER-Modellierung



Erklärungen zum ER-Diagramm

Mit der Nennung der Häusernamen (Gryffindor, etc.) wird die Aufgabenstellung eine Blendgranate. Diese sind nämlich nicht, wie man vielleicht vermuten würde, als eigenständige Entitätstypen zu definieren, sondern sind Instanzen des Entitätstyps *Haus*, das das Attribut *Name* aufweist.

1.2.3 Aufgabe 3: SQL

1.
```sql
SELECT Titel, COUNT(*) AS Anzahl
  FROM Figur
GROUP BY Titel;
```

Erklärung:
Diese Abfrage gruppiert die Datensätze aus der `Figur`-Relation nach dem *Titel*-Attribut. Dann verwendet sie die `COUNT`-Funktion, um die Anzahl der Datensätze in jeder Gruppe zu zählen. Wenn die Zeilen gelesen sind, verbleibt für jede Gruppe nur eine Zeile, die die gesamte Gruppe repräsentiert. Das Ergebnis enthält zwei Spalten: *Titel* und *Anzahl*. Dabei repräsentiert *Titel* den vorhandenen Titel und *Anzahl* die Anzahl der Vorkommen dieses Titels.

2.
```sql
SELECT f.Name
  FROM Figur f
  JOIN lebt l
    ON f.Id = l.Id
 WHERE l.Name = "Kings Landing"
```

Erklärung:
Wir Joinen die Relationen `Figur` und `lebt`, weil letztere die Information über die bewohnte Festung beinhaltet. Mt der Bedingung in der `WHERE`-Klausel behalten wir nur die Tupel mit dem Wohnort „Kings Lading“.

3.

```
1  SELECT fa.Name, COUNT(*) AS AnzCharaktere,
2         ff.AnzFestungen
3    FROM Familie fa
4    JOIN gehoert_zu gz
5      ON fa.Id = gz.Familie
6    JOIN (
7      SELECT COUNT(*) AS AnzFestungen
8        FROM Familie fa
9        JOIN besetzt be
10         ON be.Familie = fa.Id
11     GROUP BY fa.Id
12   ) AS ff
13 GROUP BY gz.id
```

Erklärung:
Um die benötigten Informationen zu versammeln, werden hier drei Relationen gejoint: `Familie`, `gehört_zu` und eine dritte, die wir uns selbst erzeugen. Sie joint `Familie` und `besetzt` für übereinstimmende Familien-IDs. Diese gruppieren wir nach der Familien-ID, und zählen die Anzahl der Zeilen je Gruppe. Ergebnis ist eine Relation, die für jede Gruppe nur die Anzahl an Festungen behält. Wir nennen sie kurz `ff` (Festungen der Familien).

Die Gesamttabelle, also das Ergebnis beider Joins, gruppieren wir nach der ID der `gehört_zu`, denn diese referenziert die Figuren aus der `Figur`-Relation. In jeder Gruppe sind also diejenigen Die Zeilen dieser Gruppen zählen wir über `COUNT(*) AS AnzCharaktere`, so dass wir zu jeder Familie schon mal die Anzahl an Charakteren nennen können.

Die Anzahl der Festungen können wir dagegen aus der gejointen Relation `ff` ergänzen. Wir führen in der Ergebnisrelation also drei Attribute auf:
Familienname, Anzahl der Charaktere, und Anzahl an Festungen.

4.

```
1   SELECT fi.Name, fi.Schwertkunst, fi.Titel
2     FROM Figur fi1, figur fi2
3     JOIN gehoert_zu gz
4       ON fi1.Id = gz.Id
5     JOIN besetzt b
6       ON gz.Familie = b.Familie
7     JOIN Festung fe
8       ON b.Festung = fe.Name
9     JOIN Familie fa
10      ON fa.Id = gz.Familie
11   WHERE fe.Name= "Westeros" AND fi1.Titel < fi2.Titel
12 GROUP BY fi1.Id
13   HAVING COUNT(*) <=5
14 ORDER BY fi.Titel ASC
```

Erklärung:
Wir joinen zunächst alle Tabellen, aus den wir Informationen benötigen, also:

- Figur
- gehoert_zu
- besetzt
- Festung
- Familie

In der `WHERE`-Klausel filtern wir dann diejenigen Zeilen, die wir benötigen, anhand der Bedingung `fe.Name = "Westeros"`. Die `SELECT`-Klausel wird wie üblich gespeist mit allen Attributen, die wir am Ende sehen möchten.

Die Zeilen 11 bis 13 sind hier ein Konstrukt, mit denen die obersten 5 behalten werden sollen, sind[1]. Dafür bilden wir in Zeile 2 das Kreuzprodukt aus der `Figur`-Relation. Da wir keine weiteren Anhaltspunkte haben, müssen wir davon ausgehen, dass der Datentyp des Attributs Titel anhand des <-Operators vergleichbar ist, zumal kein anderes Attribut in Frage kommt. In Zeile 11 behalten wir also anhand der Bedingung `fi1.Titel < fi2.Titel` die Tupel, deren Titel in `fi1` in `fi2` einen Verbundpartner hat, dessen Titel besser ist.

Beim anschließenden Gruppieren (Zeile 12) werden Tupel zusammengefasst, die dieselbe Eigenschaft haben. Wir gruppieren hier also nach der ID, und *nur* nach der ID[2], da sie ausschlaggebend ist für die Filterung, die wir nachfolgend mit der `HAVING`-Klausel durchführen. Mit der `HAVING`-Klausel behalten wir dann nur diejenigen Gruppen, die höchstens 5 Tupel aufweisen, da dies diejenigen sind, für die höchstens 4 bessere Verbundpartner gefunden wurden.

Zu guter Letzt sortieren wir in Zeile 14 aufsteigend nach dem Titel.

[1] Elegante Sprachkonstrukte wie etwa `LIMIT 5` sind leider im StEx unzulässig.
[2] Also nicht wie in der Vorlage aus dem KonzMod-Skript, in der nach *allen* Attributen gruppiert wird.

5.
```sql
DELETE FROM Figur
WHERE Lebendig = 'nein'
```

Erklärung:
Hierfür sollte die DELETE FROM-Syntax bekannt sein. Die WHERE-Bedingung definiert die zu löschenden Zeilen, in diesem Falle also alle Toten (für die das Attribut *Lebendig* entsprechend verneint werden kann).

6.
```sql
ALTER TABLE Figur
DROP COLUMN Lebendig
```

Erklärung:
Bei Kenntnis der entsprechenden Befehle selbsterklärend.

7.
```sql
CREATE TABLE Waffen (
    Name        VARCHAR (20) PRIMARY KEY ,
    Besitzer    VARCHAR (20) ,
    FOREIGN KEY ( Besitzer )    REFERENCES Figur ( Name )
    Staerke     INT  CHECK ( Staerke  >= 0  AND Staerke  <= 5)
);
```

Erklärung:
Die Herausforderung besteht hier im Auswendigkennen der Syntax der CREATE TABLE-Anweisung, und in der Auswahl der passenden Datentypen für die einzelnen Attribute.

Um das zulässige Spektrum für das Attribut Stärke zu definieren, benötigen wir außerdem eine CHECK-Klausel, die Unter- und Obergrenze festlegt.

1.2.4 Aufgabe 4: Relationale Algebra

1. **In Umgangssprache:**
   Es sollen diejenigen Figuren ausgegeben werden, die zu keiner Familie gehören.

   Oder etwas formaler:
   Wähle die Menge der Figuren f, die enthalten sind in der Tabelle Figur, zu denen es keinen Eintrag g gibt in der Tabelle gehoert_zu, der dieselbe Id hat wie f.

   In relationaler Algebra:
   Figur f $\triangleright\!\!\!\triangleleft_{f.Id = g.Id}$ gehoert_zu g

2. **In Umgangssprache:**
   Es sollen diejenigen lebendigen Figuren ausgegeben werden, die in der Festung bzw. einer der Festungen wohnen, die von der Familie Stark besetzt ist bzw. sind.

   Oder etwas formaler:
   Wähle die Menge der Figuren f, die:

   - f enthalten sind in der Tabelle Figur (f ∈ Figur),
   - in einer Festung leben (l.Id = c.Id; wobei nicht bekannt ist, aus welcher Tabelle c stammt - Danke Aufgabensteller!), die
   - den gleichen Namen hat wie die Figur (Festung(l.Festung = f.Name)),
   - und von dieser Figur besetzt wird (besetzt(f.Name = b.Festung)),
   - wobei es eine Familie b gibt, die diese Festung besetzt (b.Familie = f2.Id),
   - die mit Namen „Stark“ heißt (f2.Name = Stark).

   In relationaler Algebra:
   $\sigma_{f2.Name = 'Stark'}$(Familie f2 $\triangleright\!\!\!\triangleleft_{b.Familie = f2.ID}$ besetzt b $\triangleright\!\!\!\triangleleft_{f.Name = b.Festung}$ Figur f $\triangleright\!\!\!\triangleleft_{f.ID = l.ID \wedge f.Name = l.Festung}$ lebt l)

1.2.5 Aufgabe 5: Entwurfstheorie

1. **Berechnung der kanonischen Überdeckung**

   Definition: Kanonische Überdeckung
   Gegeben eine Menge an funtionalen Abhängigkeiten. Dann bezeichnet die kanonische Überdeckung die minimale Menge an aquivalenten funktionalten Abhängigkeiten.

   Um aus einer gegebenen Menge F von funktionalen Abhängigkeiten eine (die kanonische Überdeckung ist nicht eindeutig) kanonische Überdeckung zu finden, kann man folgenden Algorithmus verwenden:

   1. **Linksreduktion**
      Für alle $\psi \in \Psi$ ersetze $\Psi \to \Gamma$ durch $\Psi \setminus \{\psi\} \to \Gamma$, falls Γ schon bestimmt ist durch $\Psi \setminus \{\psi\} \to \Gamma$.
   2. **Rechtsreduktion**
      Für alle $\gamma \in \Gamma$ ersetze $\Psi \to \Gamma$ durch $\Psi \to \Gamma \setminus \{\gamma\}$, falls γ schon transitiv bestimmt ist durch Ψ.
   3. **Elminieren leerer Klauseln**
      Eliminiere Klauseln der Form $\Psi \to \emptyset$.
   4. **Zusammenfassen**
      Fasse Formeln $a \to b_0$, $a \to b_1 \ldots$ zusammen zu $a \to b_0 \cup b_1 \ldots$.

Zeichen	Bedeutung
Ψ (Groß-Psi)	Menge der Attribute auf der linken Seite einer FA.
ψ (Klein-Psi)	Einzelnes Attribut auf der linken Seite einer FA.
Γ (Groß-Gamma)	Menge der Attribute auf der rechten Seite einer FA.
γ (Klein-Gamma)	Einzelnes Attribut auf der rechten Seite einer FA.

Linksreduktion
Aus ZW → YQV können wir W streichen, denn über die Z → WX gelangen wir weiterhin von Z nach W. Dadurch wird nichts zerstört.

X → YZ
Z → WX
Q → XYZ
V → ZW
ZW → YQV

Rechtsreduktion
Transitiv bestimmt ist ein Attribut auf der rechten Seite (γ) dann, wenn man es auch bestimmen kann über eine andere FA. Wir betrachten also alle FAs (ψ) links mit mehr als einem Element, denn nur solche können reduziert werden.

Dabei sind folgende Reduktionen möglich:

- X → YZ kann reduziert werden um Y, denn über X → Z → YQV gelangen wir weiterhin von X nach Y.
- Q → XYZ kann reduziert werden um
  - X, denn über Q → Z → WX gelangen wir weiterhin von Q nach X.
  - Y, denn über Q → Z → YQV gelangen wir weiterhin von Q nach Y.
  - Z, denn über Q → X → YZ gelangen wir weiterhin von Q nach Z.
  - Jedoch *nicht um alle drei gleichzeitig*, sondern entweder um XY oder YZ. Denn sonst würden Abhängigkeiten zerstört werden.
- V → ZW kann reduziert werden um W, denn über V → Z → WX gelangen wir weiterhin von Z nach W.

Wie wir sehen sind hier zwei unterschiedliche Varianten der Reduktion möglich. Eines müssen wir aber beachten: wie eingangs in der Definition erwähnt, bezeichnet die kanonische Überdeckung die *minimale* Menge an äquivalenten funktionalen Abhängigkeiten. Also sollten wir auf Möglichkeiten achten, bestehende FAs auf der rechten Seite *möglichst vollständig* zu reduzieren, so dass (im Schritt 3) die gesamte FA eliminiert werden kann. Hier wird jedoch in keiner Variante eine rechte Seite vollständig eliminiert, daher sind beide Varianten legitim. Wir wählen Variante 1:

Variante 1:	**Variante 2**:
X → YZ	X → YZ
Z → WX	Z → WX
Q → XYZ	Q → XYZ
V → ZW	V → ZW
Z → YQV	Z → YQV

Eliminieren leerer Klauseln

Eleminiere Klauseln der Form $\Psi \rightarrow \emptyset$.

Entfällt, da im vorigen Schritt keine leere Klausel entstanden ist.

Zusammenfassen
Entfällt, da keine linken Seiten mit denselben Attributen vorliegen.

Endergebnis
Damit erhalten wir:

X → Z
Z → WX
Q → Z
V → Z
Z → YQV

2. **Funktionale Abhängigkeiten**

Lichtschwert	→ Seite der Macht
JediID	→ Name, Rasse, Lichtschwert, Seite der Macht
Name, Rasse	→ Lichtschwert, Seite der Macht
JediID, Rasse	→ Lichtschwert, Seite der Macht
Name, Rasse, Lichtschwert	→ Seite der Macht
Name, Rasse, Seite der Macht	→ Lichtschwert

3. (a) Eine Relation ist in erster Normalform (1NF), wenn sie nur atomare Attributwerte besitzt.

   (b) In Zeile 2 liegen im Tupel mit *JediID* „3“ zwei mehrwertige Attribute vor:

   - *Lichtschwert* (Rot, Blau)
   - *Seite der Macht* (Gute Seite, dunkle Seite)

   (c) Die Überführung in die erste Normalform erfordert eine Aufspaltung derjenigen Tupel, die das Kriterium der Atomarität verletzen (also die mit *JediID* 3) [3]. Mithin genügt eine Aufspaltung auf zwei Tupel anstatt vier.

   Randnotiz
   Durch diese Aufspaltung geht jedoch die Eindeutigkeit des Primärschlüsselattributs *JediID* verloren. Um die Eindeutigkeit wiederherzustellen, müssen weitere Attribute zum Primärschlüssel hinzugefügt werden. Dafür eignet sich beispielsweise *Lichtschwert*, das somit ebenfalls unterstrichen werden muss:

JediID	**Name**	**Rasse**	**Lichtschwert**	**Seite der Macht**
2	Yoda	Unbekannt	Grün	Gute Seite
3	Anakin Skywalker	Mensch	Blau	Gute Seite
3	Anakin Skywalker	Mensch	Rot	Dunkle Seite
4	Mace Windou	Mensch	Lila	Gute Seite
5	Count Doku	Mensch	Rot	Dunkle Seite
6	Ahsoka Tano	Togruta	Grün	Gute Seite
7	Yoda	Mensch	Rot	Dunkle Seite

4. (a) Eine Relation ist in zweiter Normalform (2. NF), wenn sie

   - in erster Normalform ist und
   - alle Nicht-Schlüsselattribute voll funktional abhängen von jedem Schlüsselkandidaten.

   (b) Zwecks Bestimmung der 2.NF-Konformität müssen wir die funktionalen Abhängigkeiten betrachten, die sich hier darstellen wie folgt:

[3] Freilich ist uns im Staatsexamen auch das nötige Hintergrundwissen über das Star Wars-Universum geläufig, so dass wir hier konkret wissen, dass die Farbe des Lichtschwertes und die Seite der Macht miteinander zusammenhängen, wobei ein blaues Lichtschwert die gute Seite der Macht repräsentiert, während ein rotes Lichtschwert den Bösewichtern vorbehalten ist.
Vgl. https://www.turn-on.de/article/star-wars-was-bedeuten-die-farben-der-lichtschwerter-343843

JediID	→ Name
Name	→ JediID
Lichtschwert	→ Seite der Macht
JediID, Name	→ Rasse, Lichtschwert, Seite der Macht
JediID, Rasse	→ Name, Lichtschwert, Seite der Macht
JediID, Seite der Macht	→ Name, Rasse, Lichtschwert
Name, Rasse	→ JediID, Lichtschwert, Seite der Macht
Name, Lichtschwert	→ JediID, Rasse, Seite der Macht
Name, Seite der Macht	→ JediID, Rasse, Lichtschwert

Betrachten wir beispielhaft den Schlüsselkandidaten {JediID, Rasse}. Parallel zu ihr gilt auch die FA JediID → Name. Somit bestimmt hier ein Schlüsselattribut (*JediID*) alleine ein Nichtschlüsselattribut (*Name*).

Damit ist die 2. NF verletzt.

(c) Die 2. NF ist stets automatisch erfüllt, wenn nur ein einziges Attribut einen Schlüsselkandidaten bildet, denn dann *kann* es keine Teilmengen innerhalb des Schlüsselkandidaten geben, die Nichtschlüsselattribute bestimmen. Um das zu erreichen, ändern wir den Datensatz. Wir ändern die im untersten Tupel die *JediID* von 2 zu 7, dann ist *JediID* allein ein Schlüsselkandidat.

Damit ist die 2. NF erfüllt.

5. (a) Eine Relation ist in dritter Normalform (3. NF), wenn sie

- in zweiter Normalform ist und
- kein Nicht-Schlüsselattribut transitiv abhängig ist von einem Schlüsselkandidaten.

(b) Durch die vollzogene Aufteilung des Tupels haben wir folgende Schlüsselkandidaten:

JediID, Lichtschwert	→ Name, Rasse, Seite der Macht
JediID, Seite der Macht	→ Name, Rasse, Lichtschwert

Betrachten wir den ersten Schlüsselkandidaten, so stellen wir fest, dass *Lichtschwert* allein bereits die *Seite der Macht* bestimmt.

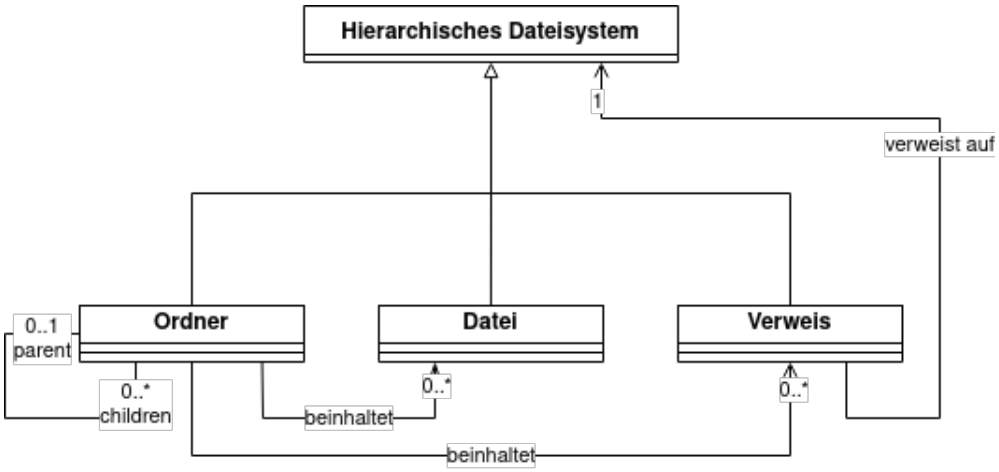
Damit ist 2. NF nicht erfüllt, und damit auch nicht die 3. NF.

2 Thema Nr. 2

2.1 Teilaufgabe 1: Softwaretechnologie

2.1.1 Aufgabe 1: Modellierung- und Muster

(a)



Erklärungen zum Klassendiagramm:
In unserer Implementierung definieren wir für den Root-Order keine eigene Klasse. Die parent-Beziehung eines Ordners zu sich selbst wird daher kardinalisiert mit 0..1, denn der Root-Ordner selbst hat keinen Elternordner.

(b)

```
1   abstract  class  FSE {
2
3       String  name ;
4       Ordner  parent ;
5       FSE  next ;
6
7       public  FSE ( Ordner p)  {
8           this . parent  = p;
9       }
10
11      //  Aufgabenteil  c)
12      FSE  delete ( FSE  o)  {
13          if  ( this . equals ( o)  {
14              kopf . delete ( kopf ) ;
15              return  next ;
16          } else  {
17              kopf  = kopf . delete (o) ;
18              next  = next . delete (o) ;
```

```
19              }
20              return ;
21          }
22      }
23
24      class   Ordner  extends  FSE  {
25          FSEList   inhalt    = new FSEList () ;
26          FSE  kopf ;
27
28          public    Ordner ()   {
29
30          }
31      }
32
33      class   FSEList    {
34          FSE  root   = ...;
35      }
36
37      class   Verweis   {
38          FSE  ziel ;
39
40          //   Aufgabenteil     c)
41          delete ( FSE   o )  {
42              if    ( this . equals ( o) )   {
43                  ziel . delete ( ziel ) ;
44                  ziel    = null ;
45                  return    next ;
46              }  else   {
47                  ziel    = ziel . delete (0) ;
48                  next   = next . delete (0) ;
49                  return    next ;
50              }
51          }
52      }
53
54      class   Datei   {
55          //   Aufgabenteil     c)
56          delete ( FSE   o )  {
57              if    ( this . equals ( o)
58                  return    next ;
59              else
60                  return    next   = next . delete ( o);
61          }
62      }
```

(c) Siehe obige Implementierung.

(d) Wenn es zyklisch ist, läuft man ggf. in eine Endlos-Rekursion. Dies könnte vermieden werden durch eine Besucht-Liste, die delete-Methode mit durchgereicht wird. Bei Azyklizität gibt es keine Probleme.

(e) Entwurfsmuster sind bewährte Lösungsschablonen für wiederkehrende Entwurfsprobleme in der Software-Entwicklung. Sie bilden eine wiederverwendbare Vorlage zur Lösung von Problemlösung, die in einem bestimmten Zusammenhang einsetzbar sind.

Drei Beispiele:

1. **Abstrakte Fabrik**
   Die Abstrakte Fabrik ist Erzeugungsmuster. Es definiert eine Schnittstelle zur Erzeugung einer Familie von Objekten. Dabei werden die konkreten Klassen der zu instanziierenden Objekte nicht näher festgelegt. Verwendet wird die Abstrakte Fabrik, wenn
   - ein System unabhängig von der Art der Erzeugung seiner Produkte arbeiten soll,
   - ein System mit einer oder mehreren Produktfamilien konfiguriert werden soll,
   - eine Gruppe von Produkten erzeugt und gemeinsam genutzt werden soll oder
   - in einer Klassenbibliothek die Schnittstellen von Produkten ohne deren Implementierung bereitgestellt werden sollen.

   Eine typische Anwendung ist die Erstellung einer grafischen Benutzeroberfläche mit unterschiedlichen Oberflächenmotiven.
2. **Strategie**
   Die Strategie ist ein Verhaltensmuster. Es definiert eine Familie austauschbarer Algorithmen. Strategie-Objekte werden ähnlich wie Klassenbibliotheken verwendet. Im Gegensatz dazu handelt es sich jedoch nicht um externe Programmteile, sondern um integrale Bestandteile des eigentlichen Programms, die deshalb als eigene Objekte definiert wurden, damit sie durch andere Algorithmen ausgetauscht werden können. Die Verwendung von Strategien bietet sich an, wenn
   - viele verwandte Klassen sich nur in ihrem Verhalten unterscheiden.
   - unterschiedliche (austauschbare) Varianten eines Algorithmus benötigt werden.
   - verschiedene Verhaltensweisen innerhalb einer Klasse fest integriert sind (meist über Mehrfachverzweigungen), aber die verwendeten Algorithmen wiederverwendet werden sollen bzw. die Klasse flexibler gestaltet werden soll.
3. **Kompositum**
   Das Kompositum-Muster ist ein Entwurfsmuster aus der Kategorie der Strukturmuster, das verwendet wird, um hierarchische Strukturen von Objekten zu erstellen und diese einheitlich zu behandeln. Es ermöglicht, dass sowohl einzelne Objekte als auch zusammengesetzte Objekte einheitlich behandelt werden können.

Für die Teilaufgabem a und b haben wir zurückgegriffen auf das Muster *Kompositum*.

2.1.2 Aufgabe 2: Assertions

(a) Das Programm erhält zwei zweidimensionale Reihungen A und B sowie einen Wert m, und liefert eine zweidimensionale Reihung der Größe m.
Es befüllt die einzelnen Felder der Rückgabe-Reihung mit der Summe der jeweiligen Produkte aus den Zeilenwerten von A und den Spaltenwerte von B, die der Reihe nach durchlaufen werden.

(b)

```
1    assert   m >= 0;
2    assert   m <   A. lenth ;
3    assert   m <   B. length ;
4
5    // Alternativ   waeren auch folgende   Zusicherungen   denkbar
         und sinnvoll :
6    assert   A  !=  null ;
7    assert   B  !=  null ;
```

(c) Wenn die Bedingung, die die Assertion sicherstellen soll, nicht erfüllt wird, wird ein Assertion Error geworfen. Da Assertions aktiviert werden müssen, damit sie ausgelöst werden können, hängt das Programmverhalten ab vom jeweiligen PC, und ist damit inkonsistent.
Weiter mpssen Assertions abgefangen und weitergegeben werden, sonst können sie zu ungewollten Programmabbrüchen führen.

(d) Hoffmann S.96 Bei der Berechnung von mathematischen Funktionen können Kontrakte ein sinnvolles Werkzeug sein, um vor Ausführung des Programms Fehler zu finden, und die Möglichkeit einer Berechnung sicherzustellen. Beispielsweise kann man sicherstellen, dass eine Divisionsfunktion keine Eingaben mit 0-Werten erhält.

2.1.3 Aufgabe 3: Softwarearchitektur- und Agilität

(a) Allgemein kann man die Qualität von Software anhand verschiedener Merkmale bestimmen. Dirk Hoffmann nennt in seinem Buch *Software-Qualität* beispielsweise folgende Qualitätsmerkmale von Software:

- Funktionalität
- Laufzeit
- Zuverlässigkeit
- Benutzbarkeit
- Übertragbarkeit
- Wartbarkeit
- Transparenz
- Testbarkeit

Die ersten vier davon sind nach außen sichtbar und orientieren sich am Anwender bzw. Kunden. Letztere vier dagegen repräsentieren die „inneren Werte" von Software. Sie beziehen sich somit auf die Beschaffenheit der Software-Architektur.

Drei davon werden im Folgenden besprochen:

1. **Transparenz**
   Transparenz bezieht sich auf die interne Umsetzung der Programmfunktionalität, insbesondere deren Ordentlichkeit und Verständlichkeit.
   Ein negatives Beispiel wäre eine Softwarearchitektur, in der die Funktionalität nicht im implementierten Code ersichtlich wird. Die gewünschten Funktionen sind zwar vorhanden, es lässt sich aber nicht oder nur schwer erkennen, wie der Code sie umsetzt.
2. **Wartbarkeit**
   Wartbarkeit bezeichnet die Möglichkeit, Software auch nach dessen Inbetriebnahme korrigieren, modifizieren und erweitern zu können.
   Ein negatives Beispiel wäre so genannter „Spaghetti-Code", der verworrene Kontrollstrukturen aufweist, und dadurch im Vergleich zu klar strukturiertem Code eine deutlich schlechtere Wartbarkeit aufweist.
3. **Übertragbarkeit**
   Übertragbarkeit (auch „Portierbarkeit") bedeutet, wie einfach eine bestehende Software in eine andere Umgebung übertragen werden kann.
   Ein negatives Beispiel wäre eine Software-Architektur, die perfekt auf eine bestimmte Umgebung zugeschnitten ist, etwa indem sie sie speziellen Fähigkeiten der Hardware bestmöglich ausnutzt. Damit wird aber die Übertragbarkeit schwierig, weil die Hardware einer anderen Umgebung komplett anders ausfallen kann.

(b) **Information Hiding**
Information Hiding (dt. auch „Geheimnisprinzip" oder „Datenkapselung") bezeichnet in der objektorientierten Softwareentwicklung ein Modularisierungsprinzip, das besagt, dass die Art und Weise, wie ein Modul seine Aufgaben erfüllt, im Innern des Moduls vor dem Zugriff von außen verborgen werden soll. Über das Modul sollen nach außen nur die Dinge bekannt sein, die als Modulschnittstelle definiert werden (Abstraktion von der internen Realisierung). Der direkte Zugriff auf die interne Datenstruktur wird unter-

bunden und erfolgt stattdessen über definierte Schnittstellen (Black-Box-Modell).
In Java wird Information Hiding auf Klassenebene in erster Linie erreicht durch die Verwendung von Zugriffsmodifikatoren (`public`, `private` und `protected`) und Getter- und Setter-Methoden.
Unter Umständen kann Information Hiding führen zu Geschwindigkeitseinbußen durch den Aufruf von Zugriffsfunktionen. Der direkte Zugriff auf die Datenelemente wäre schneller. Daneben erfordert es zusätzlichen Programmieraufwand für die Erstellung von Zugriffsfunktionen.

(c) **Refactoring**
Refactoring (dt. auch „Refaktorierung“) bezeichnet in der Software-Entwicklung die Strukturverbesserung von Quelltexten unter Beibehaltung des beobachtbaren Programmverhaltens. Dabei sollen Lesbarkeit, Verständlichkeit, Wartbarkeit und Erweiterbarkeit verbessert werden, mit dem Ziel, den jeweiligen Aufwand für Fehleranalyse und funktionale Erweiterungen deutlich zu senken.

(d) **Scrum - Die Kernidee**
Scrum (englisch für „Gedränge“) bezechnet ein Vorgehensmodell der Agilen Software-Entwicklung, die das Ziel verfolgt, rasch eine funktionsfähige Software-Lösung auszuliefern, um den Kundenwunsch schnell zu erfüllen. Zentral ist dabei eine enge Zusammenarbeit mit dem Kunden, die eine flexible Handhabung von Anforderungen und eine unmittelbare Rückmeldung des Kunden in Bezug auf gelieferte Software-Komponenten ermöglichen soll.
Scrum besteht aus einer Sammlung von Prozeduren, Rollen und Methoden zum Zwecke einer erfolgreichen Projektdurchführung.

1) Die Rollen von Scrum
Das Konzept von Scrum kennt drei Rollen:

1. **Product Owner**
   Der Product Owner ist verantwortlich für die Eigenschaften und den wirtschaftlichen Erfolg des Produkts. Er gestaltet das Produkt mit dem Ziel der Nutzenmaximierung (bspw. Unternehmensumsatz). Er erstellt, priorisiert und erläutert die zu entwickelnden Produkteigenschaften, und er urteilt darüber, welche Eigenschaften am Ende eines Sprints fertiggestellt wurden.
   Als Produktverantwortlicher hält er regelmäßig Rücksprache mit den Stakeholdern, um deren Anforderungen zu verstehen und mit denen anderer Stakeholder abzuwägen.
2. **Entwickler**
   Die Entwickler sind verantwortlich für die Realisierung der Produktfunktionalitäten in der vom Product Owner gewünschten Reihenfolge. Zudem tragen sie die Verantwortung für die Einhaltung der vereinbarten Qualitätsstandards. Das Scrum-Team ist dabei selbstorganisiert.
3. **Scrum Master**
   Der Scrum Master ist für die korrekte Durchführung des Scrum-Ansatzes verantwortlich. Dazu arbeitet er mit dem Entwicklungsteam zusammen, gehört aber selbst nicht dazu.

Er führt die Scrum-Regeln ein, überprüft deren Einhaltung, und kümmert sich um die Behebung von Störungen im Team. Er moderiert die Sprint-Retrospektive, und oft auch das Sprint Planning und Backlog Refinement.

Zusammen bilden sie das „Scrum-Team". Es tritt mit den Beteiligten in Kontakt, den so genannten „Stakeholdern". Fortschritt und Zwischenergebnisse sind für alle Stakeholder transparent. Stakeholder dürfen bei den meisten Ereignissen zuhören.

2) Wesentliche Prozessschritte
Das Prozessmodell umfasst drei wesentliche Phasen:

1. **Pregame**
   In einem *Pregame* erfolgt die Festlegung der wesentlichen Produkteigenschaften und der grundlegenden Architektur sowie die Projektplanung. Alle Eigenschaften und gewünschten Features werden gemeinsam mit dem Kunden gesammelt und priorisiert in ein Produkt-Backlog (= Auflistung an Anforderungen). In diesem Backlog werden auch geänderte Anforderungen gesammelt.
2. **Sprint**
   Die eigentliche Entwicklungsarbeit wird in einem *Sprint* durchgeführt.
   Vor einem Sprint werden die wichtigsten und machbaren Features ausgewählt aus dem priorisierten Produkt-Backlog, und übernommen in das Sprint-Backlog. Wichtig ist dabei, dass die Auswahl und Planung nur so viele Funktionen umfasst wie das Team im nächsten Sprint auch tatsächlich umsetzen kann. Das Sprint-Backlog beinhaltet also einen machbaren Auszug aus dem Produkt-Backlog.
3. **Postgame**
   Die abschließende Phase – das *Postgame* – umfasst die Bereitstellung und Auslieferung neuer Funktionalität in Form von (neuen) Releases.

3) Artefakte
Scrum umfasst drei Artefakte:

1. **Produkt-Backlog**
   Das Produkt-Backlog ist eine geordnete Auflistung der Anforderungen an das Produkt. Es wird ständig weiterentwickelt. Alle Arbeit, die das Entwicklungsteam erledigt, muss ihren Ursprung im Produkt-Backlog haben. Der Product Owner ist verantwortlich für die Pflege des Produkt-Backlogs, er verantwortet die Reihenfolge bzw. Priorisierung der Einträge.
   Zu Beginn eines Projektes enthält das Produkt-Backlog die bekannten und am besten verstandenen Anforderungen. Die Priorisierung der Eintragungen erfolgt unter wirtschaftlichen Gesichtspunkten.
   Die Verwaltung des Backlog („Backlog Refinement") ist ein fortlaufender Prozess seitens des Product Owners und des Entwicklungsteams. Er umfasst:
   - Ordnen der Einträge
   - Löschen von Einträgen, die nicht mehr wichtig sind
   - Hinzufügen von neuen Einträgen
   - Detaillieren von Einträgen
   - Zusammenfassen von Einträgen
   - Schätzen von Einträgen
   - Planung von Releases

2. **Sprint-Backlog**
   Das Sprint-Backlog ist der aktuelle Plan der für einen Sprint zu erledigenden Aufgaben. Es umfasst die Produkt-Backlog-Einträge, die für den Sprint ausgewählt wurden, und die dafür nötigen Aufgaben (z. B. Entwicklung, Test, Dokumentation). Es wird laufend aktualisiert zwecks Übersicht über den aktuellen Bearbeitungsstand. Um es für alle sichtbar zu machen, wird häufig ein Taskboard[4] verwendet. Aufgaben daraus werden allerdings nicht vom Scrum Master vergeben; vielmehr überlegen die Teammitglieder selbst, welche Aufgaben Priorität haben und zu ihren Kenntnissen und Erfahrungen passen.
3. **Product-Inkrement**
   Das Inkrement ist die Summe aller Einträge im Produkt-Backlog, die während des aktuellen und allen vorangegangenen Sprints fertiggestellt wurden.
   Am Ende eines Sprints muss das neue Inkrement in einem nutzbaren Zustand sein und der Definition of Done entsprechen.

Die Rolle von Ist- und Soll-Architekturen in agilen Entwicklungskontexten wie Scrum

In einem agilen Entwicklungskontext wie Scrum liegt der Fokus auf der schnellen Bereitstellung funktionsfähiger Software in kurzen Iterationen. Das Konzept der Ist- und Soll-Architektur kann in diesem Kontext verstanden werden wie folgt:

Ist-Architektur
Sie repräsentiert die aktuelle Architektur und technische Umgebung des Produkts zu Beginn des Entwicklungsprozesses. In Scrum wird die Ist-Architektur oft verwendet als Ausgangspunkt, um die Entwicklung zu starten. Sie kann unvollkommen oder unvollständig sein und wird iterativ verbessert und erweitert, wenn das Produkt wächst.

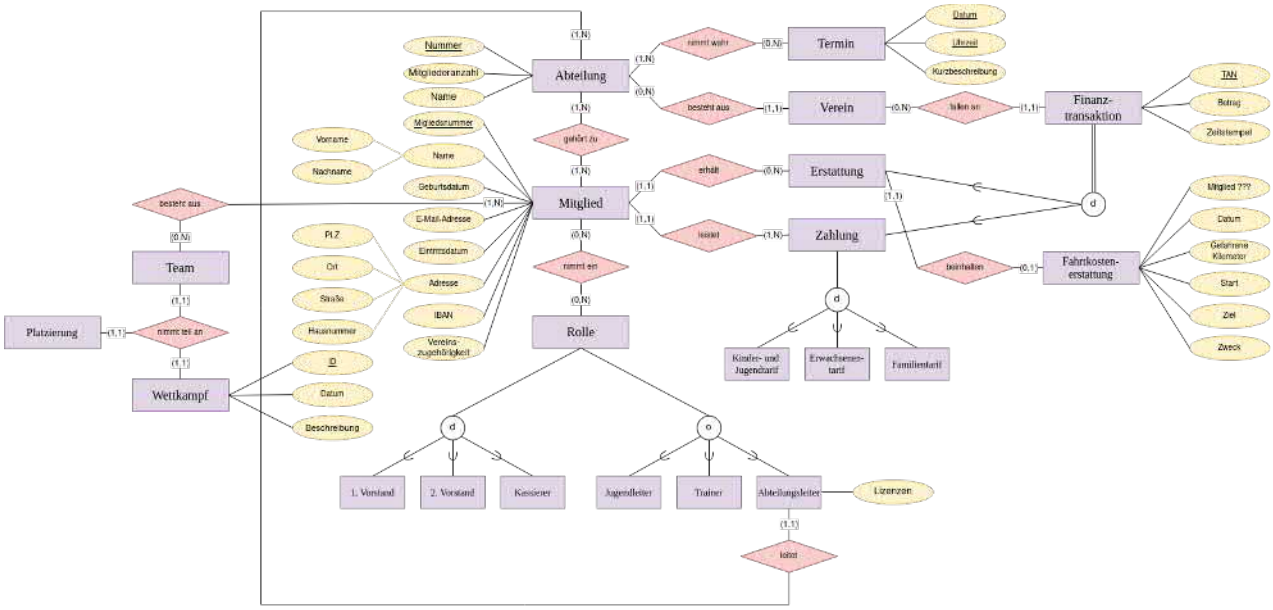
Soll-Architektur
Sie ist das angestrebte Architekturziel, das mit dem Entwicklungsprozesses angestrebt wird. Sie enthält die geplanten Verbesserungen, die während der Sprints implementiert werden, um die technische Qualität und die Architektur des Produkts zu optimieren. Die Soll-Architektur entwickelt sich im Laufe der Zeit, basierend auf den Anforderungen und den Erkenntnissen aus den Sprints.

In agilen Methoden wie Scrum wird die Architekturarbeit kontinuierlich durchgeführt und angepasst, um sicherzustellen, dass das Produkt den aktuellen und zukünftigen Anforderungen gerecht wird. Dieser iterative Ansatz ermöglicht es, schnell auf Veränderungen zu reagieren und die Architektur des Produkts schrittweise zu verbessern, anstatt alles auf einmal zu planen und zu entwickeln.

[4] Ein Taskboard ist ein Werkzeug, das von Einzelpersonen oder Teams verwendet wird, um Aufgaben – die so genannten „Tasks“ – zu visualisieren. Eine Aufgabe wird dabei auf einer Karte notiert, die im Laufe der Realisierung über das Taskboard wandert – bspw. von „zu erledigen“, über „in Bearbeitung“zu „erledigt“.
In Scrum ist das Taskboard ein zentrales Instrument, um das sich die Entwickler beim Daily Scrum versammeln und den Workflow im Sprint dokumentieren. Zur besseren Orientierung werden in Scrum auf dem Taskboard sowohl das Sprint-Ziel als auch die verschiedenen User Storys dargestellt, die im Zuge des Sprint Plannings vereinbart wurden.

2.2 Teilaufgabe 2: Datenbanken

2.2.1 Aufgabe 1: ER-Modellierung

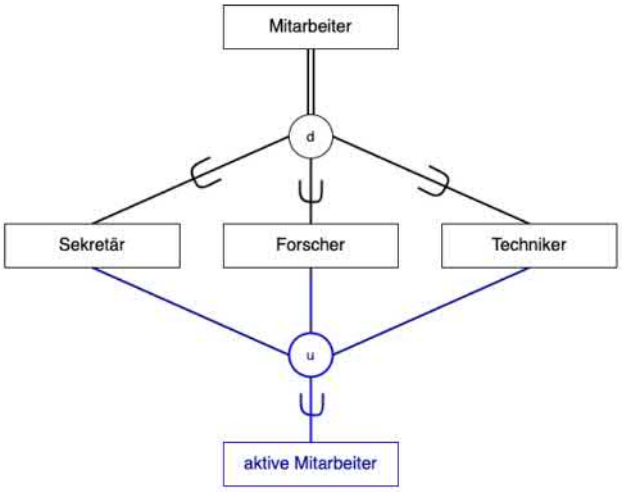


2.2.2 Aufgabe 2: ER-Modellierung

Hinweis
In dieser Aufgabenstellung befinden sich zwei Fehler:

1. Das Teilmengensymbol ist an der falschen Stelle angebracht und müsste sich eigentlich auf der Seite der drei Unterklassen befinden (siehe Lösung).
2. Der Begriff der „Entität" ist falsch. Richtig wäre „Entitätstyp".

Die Menge aller Mitarbeiter setzt sich Diagramm zusammen aus Sekretären, Forschern und Technikern. Jeder Mitarbeiter muss einer dieser Stellenbeschreibungen zugeordnet sein (totale Teilnahme) und kann nicht zugleich auf zwei Stellen sitzen, also beispielsweise nicht zugleich Sekretär als auch Techniker sein (disjunkte Unterklassen). Bei der Erweiterung des Diagramms müssen wir also einen Entitätstyp ergänzen, der die Teilmenge der aktiven Mitarbeiter darstellt. Diese Teilmenge setzt sich zusammen aus den aktiven Mitarbeitern aus den drei disjunkten Teilmengen *Sekretär*, *Forscher* und *Techniker*.
Für die Modellierung wählen wir das Konstrukt der **partiellen Kategorie**. Eine Kategorie ist definiert als eine Teilmenge der Vereinigungsmenge von mehreren Entitätsmengen. Die Vereinigungsmenge, hier bestehend aus allen Sekretären, Forschern und Technikern, enthält sowohl aktive als auch inaktive Mitarbeiter. Für eine totale Kategorie gilt die totale Teilnahme an der Vereinigungsmenge. Die partielle Kategorie enthält davon diejenigen, die aktiv sind. Deshalb wählen wir eine **partielle** Kategorie.



2.2.3 Aufgabe 3: Relationale Algebra

(a)

P	Q	S	A	B	C
10	a	5	10	b	6
10	a	5	10	b	5
25	a	6	25	c	3

$$R_1 \underset{R_1.P=R_2.A}{\rhd\lhd} R_2$$

(b)

P	Q	S	A	B	C
15	b	8	10	b	6
15	b	8	10	b	5
NULL	NULL	NULL	25	c	3

$$R_1 \underset{R_1.Q=R_2.B}{\rhd\lhd} R_2$$

Dem Mathe-Paket fehlt hier leider ein Symbol für einen Right-Outer-Join. Dieser schließt alle Tupel der rechten Seite mit ein, unabhängig davon ob sie in der linken Tabelle einen passenden Verbundpartner finden oder nicht. Dadurch ergeben sich die NULL-Einträge, da dieser rechte Eintrag keinen linken Verbundparter gefunden hat.

(c) Die Division zweier Relationen R und S kann ausgedrückt werden mittels

1. Projektion (π),
2. Mengendifferenz ($-$) und
3. Kartesischem Produkt ($\times$):

$$R \div S = \pi_{R\text{-}S}(R) - \pi_{R\text{-}S}((\pi_{R\text{-}S}(R) \times S) - R)$$

Hierbei bildet die Relation R den Dividend und die Relation S den Divisor.
Die Ergebnisrelation, also der Quotient aus R und S, enthält nur die Attribute aus R, die nicht enthalten sind in S. Die erste Projektion, $\pi_{R\text{-}S}(R)$, beseitigt daher zunächst diejenigen **Attribute** aus dem Dividend R, die im Divisor S enthalten sind.
Der Ausdruck $(\pi_{R\text{-}S}(R) \times S) - R$ berechnet eine temporäre Relation, die **Tupel** von R enthalten soll, die nicht in der Ergebnismenge enthalten sein dürfen.
Schließlich wird die Ergebnismenge der Division berechnet, indem die Tupel von R subtrahiert werden, die in der temporären Relation gefunden wurden.

Eine genaue Erläuterung, wie die temporäre Relation gebildet wird, sprengt eigentlich den Rahmen dieser mit 3 Punkten bewerteten Aufgabe. Der Vollständigkeit und der Nachvollziehbarkeit halber soll sie an dieser Stelle aber dennoch nicht ausbleiben:

Zunächst nehmen wir den Dividend R und beseitigen die Attribute des Divisors S. Von dieser Projektion bilden wir das Kreuzprodukt mit dem Divisor ($\pi_{R\text{-}S}(R) \times S$), so dass wir alle möglichen Kombinationen von Tupeln aus R und S erhalten. Warum dies geschieht,

wird besser nachvollziehbar, wenn wir uns vor Augen führen, was bei der Division eigentlich geschieht:

Dividiert man zwei Relationen R durch S, so erhält man als Ergebnis eine Relation, die vom Dividend R, aus dem die Attribute aus S wegprojiziert wurden, nur noch diejenigen Tupel behält, die in der Ursprungsversion von R verknüpft sind *mit allen Tupeln, die vorkommen in S*. Zwecks Veranschaulichung betrachten wir folgendes Beispiel:

R:

Vater	Mutter	Kind	Alter
Hans	Helga	Harald	5
Hans	Helga	Maria	4
Hans	Ursula	Sabine	2
Martin	Melanie	Gertrud	7
Martin	Melanie	Maria	4
Martin	Melanie	Sabine	2
Peter	Christina	Robert	9

S:

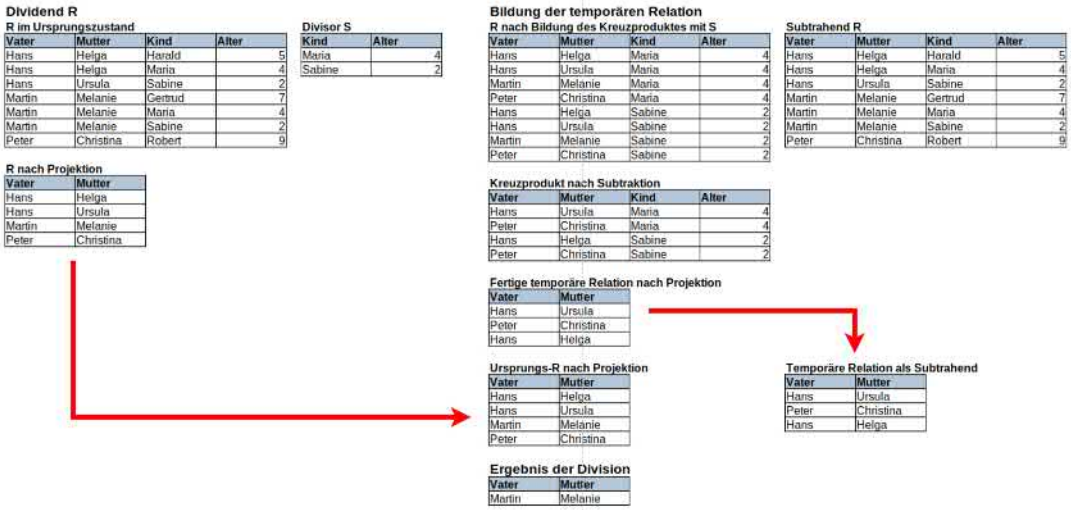
Kind	Alter
Maria	4
Sabine	2

R÷S

Vater	Mutter
Martin	Melanie

Dividiert man also R durch S, so erhält man als Ergebnis eine Relation, die nur noch diejenigen Ehepaare enthält, die sowohl eine Tochter Maria im Alter von 4 Jahren haben als auch eine Tochter Sabine im Alter von 2 Jahren.
Nun haben wir also unser ursprüngliches R durch Projektion gekürzt um die Attribute aus S, und anschließend multipliziert mit S ($\pi_{R-S}(R) \times S$). Der Punkt bei der Bildung dieses Kreuzproduktes ist, dass wir mit der Bildung aller möglichen Kombinationen von R und S die nötige „Rohmasse“ bilden, aus der wir uns die gewünschte „Austechform“ bilden können, die wir im allerletzten Schritt auf das projizierte Eingangs-R drauflegen.
Zur Bildung dieser Ausstechform subtrahieren wir zunächst einmal die Tupel aus R ($(\pi_{R-S}(R) \times S) - R$), woraufhin wir wieder die Attribute aus S wegprojizieren (π_{S}).
Subtrahieren wir nämlich nun von unserem ursprünglichen Dividend R (reduziert um die Attribute des Divisors S) diese Ausstechform, verbleibt das gewünschte Plätzchen, beziehungsweise das Ergebnis der Division: es verbleiben letztlich nur die Tupel aus R, die verknüpft waren mit allen Tupeln, die enthalten sind in S.

Was im Einzelnen mit den Relationen passiert, versucht die mit folgende Übersicht zu veranschaulichen:



Anmerkung: Im Unterschied zu SQL-Abfragen in Datenbanken handelt es sich bei den Relationen in der Relationalen Algebra um Mengen, für die gilt, dass Duplikate unzulässig sind. Projizieren wir also eine Relation, schlägt sich dies, wie hier zu sehen, potenziell nieder in der Anzahl an in der Relation enthaltenen Tupel.

2.2.4 Aufgabe 4: Normalisierung

(a) **Bestimmung der Schlüsselkandidaten**

> **Definition: Schlüsselkandidat**
> Ein Schlüsselkandidat ist ein minimaler Superschlüssel. Ein Superschlüssel wiederum ist Attribut oder eine Attributkombination, von der alle anderen Attribute der Relation funktional abhängig sind. Keine echte Teilmenge der Attribute eines Schlüsselkandidaten bestimmt vollständig die Werte aller anderen Attribute der Relation.

Um die Schlüsselkandidaten von R zu ermitteln, müssen wir die funktionalen Abhängigkeiten herausfinden, die die rechte Seite überdecken. Dafür benötigen wir den **Attributhüllenalgorithmus**. Er ermittelt die „Attributhülle", mit der man angeben kann, welche anderen Attribute einer Relation durch die gegebenen Attribute bestimmt werden können. Die beiden Attribute F und E finden wir nur auf der linken Seite, sie können also nicht durch Ableiten gewonnen werden. Daher müssen sie unabdingbar enthalten sein in den möglichen Schlüsselkandidaten. Wir verwenden sie deshalb als Eingabewert für den Attributhüllenalgorithmus. Sollte sich herausstellen, dass die nur-linksseitigen Attribute nicht ausreichen, um alle Attribute abzudecken, müssen *beid*seitige Attribute ergänzt werden. Nur-rechtsseitige Attribute kommen jedoch nicht in Frage, da sie keine Ableitungen ermöglichen.

Die Reihenfolge der Auswahl der funktionalen Abhängigkeiten im Algorithmus muss sich übrigens nicht richten nach der Reihenfolge ihrer Auflistung in der Aufgabenstellung. Häufig ist es sinnvoller, diese Reihenfolge sinnvoll zu wählen. Dabei wählen wir der Reihe nach am besten solche FAs, deren linke Seite bereits enthalten ist in der Ergebnismenge:

Ergebnismenge *Erg*	**Betrachtete FA**	**Ereignis**
{F,E}	-	Initialisierung der Ergebnismenge *Erg* mit den Attribut-Eingabewerten F und E.
{F,E} ∪ {B}	F → B	F links ist bereits enthalten in *Erg*. Wir ergänzen B rechts zu *Erg*.
{F,E,B} ∪ {A}	B → A	B links ist bereits enthalten in *Erg*. Wir ergänzen A rechts zu *Erg*.
{F,E,B,A} ∪ {D}	A → D	A links ist bereits enthalten in *Erg*. Wir ergänzen D rechts zu *Erg*.
{F,E,B,A,D} ∪ {C}	AB → C	A und B links sind bereits enthalten in *Erg*. Wir ergänzen C rechts zu *Erg*.
{F,E,B,A,D,C} ∪ {B}	DE → B	D und E links sind bereits enthalten in *Erg*. Allerdings ist B rechts auch bereits enthalten in *Erg*. Also passiert nichts.
{F,E,B,A,D,C}	-	Vollständige Attributhülle ist erreicht.

Wir können also feststellen, dass die transitive Hülle der beiden Attribute F und E alle Attribute des Relationenschemas abdeckt. Da F und E nur auf der linken Seite der FAs

zu finden sind, können sie nicht transitiv erreicht werden. Beide sind also für einen Superschlüssel unentbehrlich. Und da wir ausgehend von ihnen alle Attribute erreichen können, bilden sie gleichzeitig einen Superschlüssel. Gleichzeitig ist er minimal, da es keine Teilmenge gibt, die zur Überdeckung ausreicht. Er ist also ein Schlüsselkandidat - und gleichzeitig auch der einzig mögliche, weil er alle nur-linksseitigen Attribute umfasst.

(b) Zur Bestimmung der Menge an FAs benötigen wir den **Synthesealgorithmus**:

1. Bestimmung der kanonischen Überdeckung durch **Reduktion** der FAs.
   Diese besteht wiederum aus:
   a) Linksreduktion
   b) Rechtsreduktion
   c) Eliminieren leerer Klauseln
   d) Zusammenfassen
2. Erzeugen der neuen Relationenschemata aus der kanonischen Überdeckung.
3. Falls nötig die Hinzunahme einer Relation, die nur den Ursprungsschlüssel enthält.
4. Entfernen der Schemata, die enthalten sind in einem anderen Schema.

Schritt 1: Bestimmung der kanonischen Überdeckung

> **Definition: Kanonische Überdeckung**
> Gegeben eine Menge an funtionalen Abhängigkeiten. Dann bezeichnet die kanonische Überdeckung die minimale Menge an aquivalenten funktionalten Abhängigkeiten.

Zur Bestimmung der kanonischen Überdeckung (F_C) befolgen wir die oben genannten Schritte:

Linksreduktion

> Für alle $\psi \in \Psi$ ersetze $\Psi \rightarrow \Gamma$ durch $\Psi \setminus \{\psi\} \rightarrow \Gamma$,
> falls Γ bereits bestimmt ist durch $\Psi \setminus \{\psi\} \rightarrow \Gamma$.

Wir betrachten also alle FAs (ψ) links mit mehr als einem Element, denn nur solche können reduziert werden:

- AB → C kann reduziert werden um A, denn über B → A gelangen wir weiterhin von B nach A.

~~A~~B → C
A → D
F → B
DE → B
B → A

Rechtsreduktion

> Für alle $\gamma \in \Gamma$ ersetze $\Psi \rightarrow \Gamma$ durch $\Psi \rightarrow \Gamma \setminus \{\gamma\}$,
> falls γ bereits transitiv bestimmt ist durch Ψ.

Transitiv bestimmt ist ein Attribut auf der rechten Seite (γ) dann, wenn man es auch bestimmen kann über eine andere FA. Solche liegen hier nicht vor, daher entfällt dieser

Schritt.

Eliminieren leerer Klauseln

Eliminiere Klauseln der Form $\Psi \rightarrow \emptyset$.

Entfällt, da im vorigen Schritt keine leere Klausel entstanden ist.

Zusammenfassen

Fasse Formeln der Form $a \rightarrow b_0$, $a \rightarrow b_1 \ldots$ zusammen zu $a \rightarrow b_0 \cup b_1 \ldots$

Klauseln mit gleicher linker Seite werden zusammengefasst. Dementsprechend können wir B → C und B → A zusammenfassen zu B → CA.

Damit haben wir nun also folgende FAs:

B → CA
A → D
F → B
DE → B

(c) Ein Relationstyp befindet sich genau dann in der dritten Normalform (3NF), wenn

- er sich in der zweiten Normalform (2NF) befindet und
- kein Nichtschlüsselattribut transitiv von einem Kandidatenschlüssel abhängt.

Mithilfe der Links- und Rechtsreduktion im zuvor ausgeführten Synthesealgorithmus haben wir diese transitiven Abhängigkeiten eliminiert, und damit dafür gesorgt, dass FAs 3NF-konform sind. Zur Überführung des Relationenschemas in die dritte Normalform sind dann aber noch drei weitere Schritte erforderlich:

Schritt 2: Erzeugen neuer Relationenschemata

Aus allen $\psi \rightarrow \Gamma$ wird $R(\psi, \Gamma)$.

Wir erzeugen also folgende Relationenschemata:

R(B,C,A)
R(A,D)
R(F,B)
R(D,E,B)

Schritt 3: Ggf. Hinzufügen einer Relation

Hier muss ein Schema erzeugt werden zur Verknüpfung der Teilschemata, die bei der Erzeugung der kanonischen Überdeckung entstanden sind. Falls ein Schlüssel aus R bereits enthalten ist in einer FA in F_C, erübrigt sich dieser Schritt.
Im Aufgabenteil a (siehe 1) hatten wir den Schlüsselkandidaten EF ermittelt.Dieser ist nicht enthalten in F_C, muss also auch noch ergänzt werden:

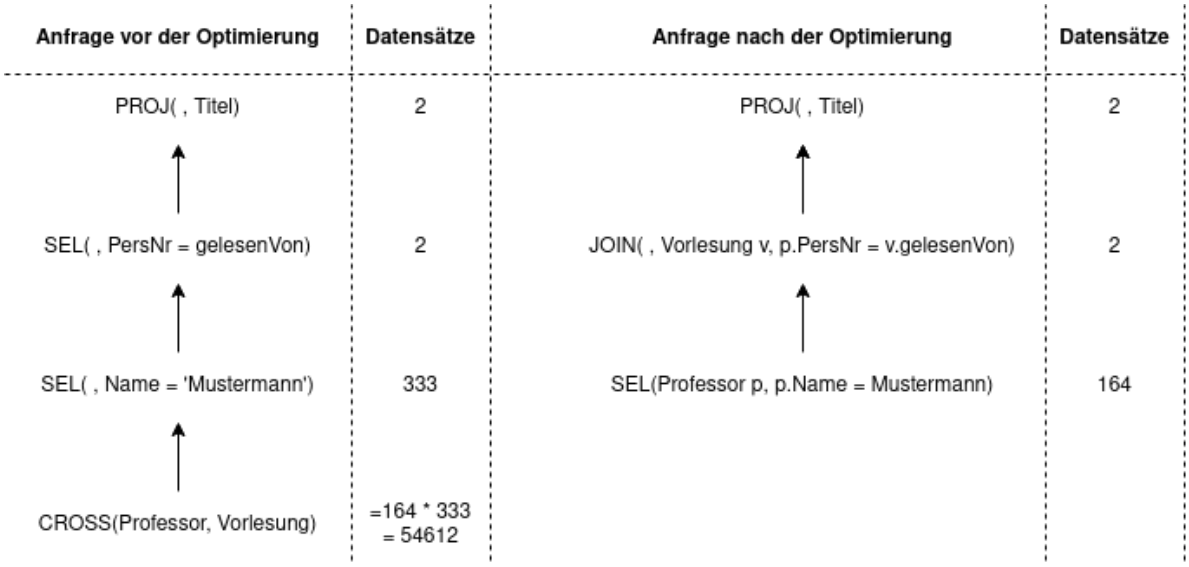
R(E,F)

Schritt 4: Entfernen überflüssiger Schemata

Überflüssig sind Schemata, wenn sie vollständig enthalten sind in einem anderen. Das ist hier nicht der Fall, also ist hier nichts zu tun. Damit haben wir folgendes Endergebnis:

R(E,F)
R(B,C,A)
R(A,D)
R(F,B)
R(D,E,B)

2.2.5 Aufgabe 5: Anfrageoptimierung



Erklärung
Wir ziehen in der optimierten Variante die Selektion vor, da durch diese die Professor-Tabelle gestutzt wird auf diejenigen Tupel, die den Mustermann-Professor enthalten. Dadurch wird das gewalte Kreuzprodukt der unoptimierten Variante vermieden.
Die zweite Selektion erfolgt dann gleich mit dem Join. Das ergibt keine Berechnungsbeschleunigung, es macht lediglich die Schreibweise kompakter, da die Join-Operation Kreuzprodukt und Selektion zugleich ist.
Warum bleiben in der unoptimierten Variante nach der ersten Selektion 333 Datensätze übrig? Weil die Vorlesung-Tabelle 333 Datensätze enthält, von denen nun jeder mit einem Mustermann-Professor-Tupel der Professor-Tabelle verbunden ist.

2.2.6 Aufgabe 6: Wissensfragen

(a) Der **Equi-Join** ist eine Verbundoperation, der im Selektionsprädikat nur den Vergleichsoperator „=“ zulässt.

Der **Natural Join** ist ein Equi-Join mit der zusätzlichen Eigenschaft, dass für ihn kein Selektionsprädikat ausdrücklich formuliert werden muss. Stattdessen wird dieses automatisch ausgeführt und ist implizit definiert wie folgt:

- Die Werte aller Attribute beider Relationen, die denselben Namen haben, werden auf Gleichheit geprüft. Zugelassen ist also nur das Vergleichsprädikat „=“ (was auch erklärt, warum der Natural-Join ein spezieller Equi-Join ist). Im Unterschied zum Equi-Join ist beim Natural Join aber keine Auswahl der zu vergleichenden Attribute möglich. Stattdessen werden stets *alle* gleichnamigen Attribute verglichen.
- Mehrere solcher Gleichheitsbedingungen werden verundet. Sie müssen also alle wahr sein für eine erfolgreiche Prüfung.
- Abschließend werden namensgleiche Attribute der zweiten Relationen wegprojiziert, so dass jeder Attributname nur einmal in der Ergebnismenge auftritt. Diese Projektion verursacht keinen Verlust von Informationen, da aufgrund der Gleichheitsbedingung in den namensgleichen Attributen die gleichen Werte stehen.

(b) Der **Theta-Join** ist eine Verbundoperation auf Relationen, der alle Vergleichsoperatoren zulässt. Konkret wird beim Theta-Join erst einmal das Kreuzprodukt zwischen den beiden Relationen gebildet. Anschließend wird auf dieses Kreuzprodukt eine Selektion anhand des Selektionsprädikats angewendet.

(c) **Unionkompatibilität** bezeichnet die Voraussetzung, dass die beteiligten Relationen in SQL kompatible Schemata aufweisen müssen, um bestimmte Operatoren anwenden zu dürfen. Kompatibilität ist für zwei Relationen gegeben, wenn folgende Bedingungen erfüllt sind:

- Gleiche Anzahl an Attributen.
- Die Attribute sind angeordnet in der gleichen Reihenfolge.
- Die Attribute haben gleiche oder kompatible Datentypen (bspw. Ganzzahlen (INT) und Gleitkommazahlen (FLOAT)).

Drei SQL-Operatoren, die Unionkompatibilität voraussetzen, sind:

1. **Union**
   Liefert die Vereinigungsmenge von zwei SELECT-Abfragen.
2. **Intersect**
   Liefert die Schnittmenge der Ergebnisse von zwei SELECT-Abfragen.
3. **Except**
   Liefert die Differenzmenge der Ergebnissen von zwei SELECT-Abfragen.

Diese Operatoren setzen Unionkompatibilität voraus, da sie verschiedene Relationen kombinieren oder vergleichen. Und das ist nur möglich, wenn oben genannte Bedingungen erfüllt sind.

(d) **Backward Recovery** und **Forward Recovery** sind zwei Strategien zur Fehlerbehandlung in Computersystemen.
Backward Recovery bezieht sich auf die Wiederherstellung eines Systems nach dem Auftreten eines Fehlers. Dabei wird der Systemzustand zurückgesetzt auf einen vorherigen konsistenten, funktionsfähigen Zustand, bevor der Fehler aufgetreten ist. Dies geschieht durch die Verwendung von Sicherungskopien oder Protokollen, die den vorherigen Zustand des Systems speichern.
Forward Recovery bedeutet dagegen die Behandlung eines Fehlers während der Ausführung eines Systems, bei dem das System überführt wird in einen *alternativen* funktionsfähigen Zustand. Dabei versucht man, den Fehler zu isolieren oder zu umgehen, so dass das System weiterarbeiten kann mit möglichst minimalen Beeinträchtigungen.

(e) Das **Zwei-Phasen-Freigabe-Protokoll** ist ein Protokoll zur Koordination von Transaktionen in Datenbankverwaltungssystemen, das Datenkonsistenz und Integrität sicherstellen soll. Dies erreicht es dadurch, dass es die atomare Ausführung mehrerer Transaktionen erzwingt.

Das Protokoll sieht zwei Phasen vor:

1. **Expansionsphase**
   In dieser Phase erhält jede Transaktion eine eindeutige Transaktions-ID (TID) und führt ihre Operationen aus. Wenn eine Transaktion eine Ressource (bspw. eine Datenbanktabelle) anfordert für Lese- oder Schreibzugriffe, wird die Ressource blockiert von anderen Transaktionen.
   Wenn eine Transaktion ihre Arbeit beendet hat und alle Ressourcen freigibt, tritt sie ein in die Schrumpfungsphase.
2. **Schrumpfungsphase**
   In dieser Phase gibt jede Transaktion die von ihr verwendeten Ressourcen schrittweise frei. Dies geschieht in umgekehrter Reihenfolge der Expansionsphase. Eine Transaktion gibt eine Ressource frei, nachdem sie sichergestellt hat, dass sie diese nicht mehr benötigt.
   Nachdem eine Transaktion alle ihre Ressourcen freigegeben hat, wird sie beendet.

Auf diese Weise versucht das Zwei-Phasen-Freigabeprotokoll sicherzustellen, dass keine inkonsistenten Zustände oder Verklemmungen auftreten. Wenn eine Transaktion fehlschlägt (bspw. aufgrund eines Systemabsturzes oder eines Benutzerabbruchs), können die anderen Transaktionen weiterhin ordnungsgemäß arbeiten, da die Ressourcen freigegeben werden.
Wenn eine Transaktion erfolgreich alle ihre Ressourcen freigegeben hat, ist die Transaktion atomar abgeschlossen und hat keine Auswirkungen auf andere Transaktionen.

(f) **Partial/Global Undo/Redo** sind Konzepte, die verwendet werden im Zusammenhang mit der Wiederherstellung von Transaktionen in Datenbankverwaltungssystemen.

- **Partial Undo (Unvollständiges Zurücksetzen)**
  Partial Undo bedeutet das teilweise Rückgängigmachen von Änderungen, die durch eine Transaktion vorgenommen wurden. Wenn eine Transaktion fehlerhaft ist oder abgebrochen wird, können nur bestimmte Teile der Transaktion rückgängig gemacht werden, während andere Teile, die bereits erfolgreich waren, beibehalten

werden. Dies ermöglicht es, den Datenbestand teilweise in einen konsistenten Zustand zurückzusetzen.

- **Partial Redo (Unvollständiges Wiederholen)**
  bezieht sich auf die teilweise Wiederholung von Änderungen, die durch eine Transaktion vorgenommen wurden. Wenn eine Transaktion fehlerhaft ist oder abgebrochen wird und einige ihrer Änderungen bereits auf Datenebene wirksam waren, können nur die noch ausstehenden oder unvollständigen Änderungen erneut durchgeführt werden. Dies stellt sicher, dass die Datenbank vollständig aktualisiert wird und keine inkonsistenten Zustände aufweist.
- **Global Undo (Vollständiges Zurücksetzen)**
  Global Undo bezeichnet das Rückgängigmachen von Änderungen auf Systemebene, die durch eine oder mehrere Transaktionen bewirkt wurden. Dies kann erforderlich sein, um Fehler oder Inkonsistenzen zu korrigieren, die durch fehlerhafte oder unvollständige Transaktionen verursacht wurden. Dadurch soll das System in einen vorherigen konsistenten Zustand zurückgesetzt werden.
- **Global Redo (Vollständiges Wiederholen)**
  Global Redo bezeichnet die Wiederholung von Änderungen auf Systemebene, die durch eine oder mehrere Transaktionen bewirkt wurden und möglicherweise rückgängig gemacht wurden. Dadurch soll das System auf den aktuellen Stand und sichergestellt werden, dass alle durchgeführten Änderungen berücksichtigt werden.

Welche Rolle spielen diese Konzepte mit Blick auf das ACID-Prinzip (Atomicity, Consistency, Isolation, Durability)?
Partial Undo und Partial Redo machen sich Atomarität zunutze, da sie es ermöglichen, Transaktionen als atomare Einheiten zu behandeln und Änderungen nur in dem Ausmaß rückgängig zu machen, dass Konsistenz sichergestellt wird.
Global Undo und Global Redo unterstützen Konsistenz, indem darauf abzielen, inkonsistente Zustände zu korrigieren.

(g) Das **WAL-Prinzip** ist eine Methode zur Sicherung von Datenintegrität und -konsistenz in Datenbanksystemen. Es basiert auf der Idee, dass alle Änderungen an einer Datenbank gespeichert werden in einem Protokoll vor dem Schreiben der tatsächlichen Daten auf dem Datenträger.
Das Prinzip besteht aus folgenden Schritten:

1. Vor dem Schreiben von Daten auf den Datenträger wird zuerst ein Eintrag mit den Änderungen im Transaktions-Log (WAL-Log) erstellt. Der Eintrag enthält Informationen über die geplanten Änderungen, wie beispielsweise die zu aktualisierenden Datensätze und die Art der Änderungen (Einfügungen, Löschungen oder Aktualisierungen).
2. Nachdem der Eintrag im Transaktions-Log gespeichert wurde, werden die entsprechenden Änderungen an den eigentlichen Daten vorgenommen.
3. Erst nachdem die Daten erfolgreich auf dem Datenträger gespeichert wurden, gilt die Transaktion als abgeschlossen.

Das WAL-Prinzip bietet mehrere Vorteile:

- **Möglichkeit zur Wiederherstellung**
  Im Falle eines Systemabsturzes können die Änderungen im Transaktions-Log verwendet werden, um das System in einen konsistenten Zustand zurückzusetzen. Durch das Nachverfolgen der Änderungen im Log können die Datenbankwiederherstellungsdienste die fehlgeschlagenen oder unvollständigen Transaktionen erkennen und rückgängig machen oder erneut ausführen.
- **Datenintegrität**
  Das WAL-Prinzip gewährleistet, dass Datenänderungen immer protokolliert werden, bevor sie auf den Datenträger geschrieben werden. Dadurch wird sichergestellt, dass das Protokoll immer einen konsistenten Zustand der Datenbank widerspiegelt. Im Falle eines Systemabsturzes können die Datenbankwiederherstellungsdienste das Transaktionsprotokoll verwenden, um inkonsistente Zustände zu erkennen und zu korrigieren.
- **Leistungsfähigkeit**
  Durch das Sammeln von Datenänderungen im Transaktionsprotokoll können Schreibvorgänge auf dem Datenträger effizienter gestaltet werden. Das Schreiben der Datenbankänderungen auf den Datenträger kann in Batches oder in einer optimierten Reihenfolge erfolgen, was zu einer besseren Schreibleistung führt.

(h) Ein Datenbankindex ist eine Datenstruktur, die in Datenbankverwaltungssystemen verwendet wird, um den Zugriff auf Daten zu beschleunigen. Er funktioniert ähnlich wie das Register eines Buches, das das Auffinden von Informationen erleichtert, indem es eine Verknüpfung herstellt zwischen den Schlüsselwerten und den Positionen der tatsächlichen Daten.
Insofern besteht ein Index aus einer geordneten Liste von Schlüsselwerten und deren zugehörigen Zeigern auf die Speicherpositionen der Daten. Beim Abfragen von Daten kann das System den Index verwenden, um schnell die Datensätze zu finden, die den angegebenen Suchkriterien entsprechen, anstatt die gesamte Datenbank durchsuchen zu müssen.
Zwei häufige Arten von Datenbankindizes sind:

- **B-Baum-Index**
  Der B-Baum organisiert die Indexeinträge in einem balancierten Baum. Dadurch ist ein effizienter Datenzugriff möglich. Er ist in der Lage, sowohl nach Primärschlüsseln als auch nach sekundären Schlüsseln zu suchen und unterstützt Bereichsanfragen.
- **Streuindex**
  Der Streuindex verwendet eine Streufunktion, um die Indexeinträge direkt in einer Streureihung abzubilden. Jeder Schlüsselwert wird in einen Streuwert umgewandelt, der dann als Adresse für den Zugriff auf den entsprechenden Datensatz verwendet wird. Streuindizes sind effizient für den direkten Zugriff auf bestimmte Schlüsselwerte, aber sie unterstützen keine Bereichsanfragen und sind empfindlich gegenüber Kollisionen, wenn zwei oder mehr Schlüssel denselben Streuwert erzeugen.

2.2.7 Aufgabe 7: SQL

(a)

```
1  SELECT f.Fahrername, f.Fahrer_ID,
2         COUNT(*) AS Disqualifikationen
3    FROM Fahrer f
4    JOIN Rennteilnahme rt
5      ON f.Fahrer_ID = r.Fahrer_ID
6   WHERE r.Jahr >= 2005
7     AND r.Jahr <= 2017
8     AND rt.disqualifiziert = "ja"
9  GROUP BY f.Fahrer_ID
10 ORDER BY Disqualifikationen DESC
```

Erklärung:
Zunächst joinen wir die Tabellen, die die benötigten Informationen enthalten, also:

- `Fahrer` und
- `Rennteilnahme`

In der Gesamttabelle stehen nun zu jeder `Fahrer_ID` mehrere Einträge, die durch den Join mit der Tabelle `Rennteilnahme` alle Teilnahmen auflisten, die zu diesem Fahrer gehören. Von diesen interessieren uns aber nur die disqualifizierten Rennteilnahmen im Zeitraum von 2005 bis 2017, also filtern wir in der `WHERE`-Klausel nach den entsprechenden Bedingungen.
In Zeile 9 gruppieren wir nach der `Fahrer_ID`, um die ihr zugeordneten Einträge in Zeile 1 mit `COUNT(*)` zählen zu können, die ja bereits nur noch disqualifizierte Einträge enthalten. Zuletzt sortieren wir in Zeile 10 noch die Disqualifikationen absteigend mit `ORDER BY`

(b)

```
1  SELECT DISTINCT Land
2    FROM Strecke
3    JOIN (
4      SELECT s.land
5        FROM Strecke s
6        JOIN Rennteilnahme rt
7          ON s.Strecken_ID = rt.Strecken_ID
8    GROUP BY s.land, rt.disqualifiziert
9      HAVING rt.disqualifiziert = "Ja"
10         AND COUNT(*) >= 1
11         ) AS RoteListe
12       ON Strecke.Land <> RoteListe.Land
```

Erklärung:
Wir möchten diejenigen Länder wissen, in denen noch nie ein Fahrer disqualifiziert wurde. Dafür besorgen wir uns eine „Rote Liste“ mit Ländern, in denen bereits ein Fahrer disqualifiziert wurde, und subtrahieren sie von der Liste *aller* Länder aus der Tabelle `Strecke`. Die Rote Liste erstellen wir, indem wir zunächst die benötigten Tabellen `Strecke` und `Rennteilnahme` joinen. Wir gruppieren das Ergebnis nach Ländern und Disqualifikationen und filtern diese Gruppierung mit der `HAVING`-Klausel um diejenigen Einträge, in den mindestens eine Disqualifikation vorlag.

In der äußeren Anfrage nehmen wir also die Tabelle `Strecke` und joinen sie mit der Roten Liste anhand Ungleichheit des Attributs `Land`, wodurch die Länder der Roten Liste herausgefiltert werden. Wir wählen *DISTINCT Land*, da uns jedes Land nur einmalig interessiert.

(c)

```
1  SELECT f. Fahrer_ID ,   f. Fahrername ,  f. Nation ,
2         f. Rennstall
3    FROM Fahrer  f1 ,
4    JOIN  Rennteilnahme  rt1 ,  Rennteilnahme  rt2 ,
5      ON f. Fahrer_ID   = rt1 . Fahrer_ID
6     AND f . Fahrer_ID   = rt2 . Fahrer_ID
7    JOIN  Strecke  s
8      ON rt1 . Strecken_ID   = s . Strecken_ID
9   WHERE s. Streckenname  = " Abu Dhabi "
10    AND rt1 . Jahr   = 2011
11    AND rt1 . Gesamtzeit   > rt2 . Gesamtzeit
12 GROUP BY f. Fahrer_ID
13   HAVING COUNT (*) <= 5
14 ORDER BY rt1 . Gesamtzeit   ASC
```

Erklärung:
Wir joinen zunächst alle Tabellen, aus den wir Informationen benötigen:

- Fahrer
- Rennteilnahme
- Strecke

Die SELECT-Klausel wird wie üblich gespeist mit allen Attributen, die wir am Ende sehen möchten und filtern in der WHERE-Klausel anhand der Vorgaben s.Streckenname = ”Abu Dhabi” AND rt1.Jahr = 2011 der Aufgabenstellung.
Wie in Aufgabe 1.2.3.4 müssen wir uns hier eines Konstruktes zur Nachahmung einer Top-N-Anfrage bedienen. Anders als in Aufgabe 1.2.3.4 befindet sich hier aber die Informationen mit den gesuchten Informationen für die Anfrage (Fahrer) und das Vergleichsattribut (die *Gesamtzeit* in Rennteilnahme) in zwei unterschiedlichen Relationen. Sie muss also doppelt gejoint werden. Die Zeilen 11 bis 13 bewerkstelligen dann, dass die obersten 5 Werte behalten werden sollen. In Zeile 13 behalten wir also anhand der Bedingung rt1.Gesamtzeit > rt2.Gesamtzeit die Einträge mit Fahrern, die in Rennteilnahme einen Verbundpartner haben, dessen Platzierung besser ist.
Beim anschließenden Gruppieren werden Tupel zusammengefasst, die dieselbe Eigenschaft haben. Wir gruppieren hier also nach der Fahrer_ID. Wohlgemerkt: *nur* nach der ID (also nicht wie in der Vorlage aus dem KonzMod-Skript, in der nach *allen* Attributen gruppiert wird), da sie ausschlaggebend ist für die Filterung, die wir anschließend mit der HAVING-Klausel durchführen. Mit der HAVING-Klausel behalten wir dann nur diejenigen Gruppen, die höchstens 5 Tupel aufweisen, da dies diejenigen sind, für die höchstens 4 bessere Verbundpartner gefunden wurden.
Zu guter Letzt sortieren wir in Zeile 16 aufsteigend nach der Gesamtzeit, so dass die Ausgabe nach der Platzierung aufsteigend erfolgt.

(d)

```
1  // Spalte "Gehalt" zur Tabelle "Fahrer" hinzufuegen:
2  ALTER TABLE Fahrer
3  ADD Gehalt DECIMAL(10,2);
4
5  // Trigger erstellen, um das Gehalt um 10 Prozent anzuheben:
6  CREATE OR REPLACE TRIGGER Gehaltserhoehung
7  AFTER UPDATE OF Rennstall ON Fahrer
8  FOR EACH ROW
9  WHEN(NEW.Rennstall <> OLD.Rennstall)
10 BEGIN
11 UPDATE Fahrer
12     SET Gehalt = Gehalt * 1.1
13   WHEREFahrer_ID = :NEW.Fahrer_ID;
14 END;
```

Erklärung:
Der obige Code führt zwei Schritte aus:

1. Zeilen 2 bis 3 ergänzen die Spalte `Gehalt` zur Tabelle `Fahrer`.
   Das Datenfeld `Gehalt` sei hier definiert als Dezimalzahl mit 10 Stellen insgesamt und zwei Stellen nach dem Dezimalpunkt.
2. Zeilen 6 bis 14 erstellen einen Trigger namens `Gehaltserhoehung`.
   Zeile 7 definiert, dass der Trigger ausgelöst wird nach einem Update-Vorgang auf der Spalte `Rennstall` in der Tabelle `Fahrer`. Er prüft, ob sich der Wert der Spalte `Rennstall` geändert hat (mit `NEW.Rennstall <> OLD.Rennstall`).
   Trifft das zu, wird, die Änderung ausgelöst. Diese wird definiert im Block von Zeile 10 bis 14 (zwischen `BEGIN` und `END`): das Gehalt des betreffenden Fahrers erhöht um 10 Prozent.

Die Selektion „`WHERE Fahrer_ID = :NEW.Fahrer_ID`“ in Zeile 13 erklärt sich wie folgt:

`:NEW.Fahrer_ID` ist ein Platzhalter, der in einem Trigger verwendet wird.
Das Präfix `:NEW` bezieht sich dabei auf den neuen Wert (nach dem `UPDATE`) der Zeile, die den Trigger ausgelöst hat.
`Fahrer_ID` ist der Name der Spalte in der Tabelle `Fahrer`.

Mit dieser Bedingung wird sichergestellt, dass das Update nur auf denjenigen Fahrer angewendet wird, der den Rennstall gewechselt hat. Es vergleicht den Wert der Fahrer_-ID des neuen Datensatzes (`NEW.Fahrer_ID`) mit dem Fahrer_ID-Wert in der Tabelle. Wenn sie übereinstimmen, wird das Update durchgeführt auf das Gehalt für den Fahrer mit dieser spezifischen Fahrer_ID.