Entwurf einer Datenbankanwendung für einen Onlineshop

Bremen, den 2. März 2017

Corinna Koch Robin Hinz Jannis Fink

Inhaltsverzeichnis

1 Informelle Beschreibung

Autor: Jannis Fink
Konzept: Corinna Koch, Robin Hinz, Jannis Fink

Wir befassen uns in unserer Ausarbeitung mit der Entwicklung einer Datenbankanwendung für einen Onlineshop. Als reale Vorlage dienten uns bestehende Shops wie z.B. *amazon.de* oder *conrad.de*. Diese Systeme bestehen aus einer Vielzahl verschiedener Datenobjekte mit teilweiser komplexen Integritätsbedingungen. Ziel des von uns erstellten Datenbankentwurfes ist es, die benötigte Programmierlogik, die auf die Datenbank aufgesetzt wird, auf ein vertretbares Minimum zu reduzieren.

Das System soll in der Lage sein, alle Daten zu speichern, die im täglichen Betrieb des Shops anfallen. Dazu gehören neben den Kundendaten auch die Daten der Angestellten und die im Lager befindlichen Artikel sowie deren Zulieferer. Ebenfalls im System gespeichert werden sollen die Bestellungen der Kunden sowie die dazu gehörende Lieferung.

Wenn das System fertiggestellt ist, soll es in der Lage sein, unter anderem die folgenden Fragen zu beantworten:

- Welche Bestellungen hat ein bestimmter Kunde getätigt?
- Welche Bestellungen dieses Kunden wurden bereits versendet?
- Mit welchem Lieferdienst/welcher Versandart wurde die Bestellung versendet?
- Welche Artikel befinden sich aktuell im Lager?
- Welche Artikel müssen nachbestellt werden?
- Welcher Lieferant kann ein bestimmtes Lager mit einem bestimmten Artikel beliefern?
- Wie viel Lagerplatz ist in einem bestimmten Lager noch vorhanden/schon belegt?

1.1 Personen

Das System muss zwei verschiedene Personen speichern. Zum Einen gibt es die Kunden, die Bestellungen tätigen können, zum anderen die Mitarbeiter. Für Kunden muss dabei über die Personenstammdaten hinaus noch eine Kontakt-Email-Adresse erfasst, für Mitarbeiter das Gehalt gespeichert werden. Für alle Personen können dabei Bankkonten hinterlegt werden, die entweder zur Bezahlung der eingekauften Artikel belastet, oder zur Überweisung des Gehaltes genutzt werden sollen.

Kunden sind darüber hinaus beliebig viele Bestellungen zugeordnet, die sie im Laufe ihrer Mitgliedschaft aufgegeben haben.

Mitarbeiter arbeiten in einem bestimmten Lagerhaus in einer fest definierten Position. Hier ist ihnen eine Menge an Bestellungen zugeordnet, die sie abzuarbeiten haben.

1.2 Artikel

Ein Artikel ist immer in einem oder mehreren Lagerhäusern vorrätig. Hierbei muss das System speichern, welcher Artikel in welchen Mengen wo vorrätig ist, damit die Mitarbeiter in der Lage sind, in ihrem Lagerhaus die für die Bestellung benötigten Artikel zusammen zu suchen und mit der Versendung der fehlenden Artikel die entsprechenden Lagerhäuser zu beauftragen.

Für jedes Lagerhaus ist neben dem aktuellen Vorrat auch stets ein Mindestvorrat gespeichert, damit das System in der Lage ist, bei zu geringen Lagerbeständen eine entsprechende Meldung auszugeben, um eine Nachbestellung anzustoßen.

1.3 Bestellung

Eine Bestellung wird von einem Kunden in Auftrag gegeben. An der Bestellung sind die Artikel in ihrer gewünschten Anzahl gespeichert. Neben den Artikeln ist einer Bestellung stets ein verantwortlicher Mitarbeiter zugeordnet, der sich um den Versand kümmert.

1.4 Anbieter

Damit die Mitarbeiter wissen, wo ein Artikel nachzubestellen ist, werden im System ebenfalls die Anbieter dieser Artikel für die verschiedenen Lagerhäuser gepflegt. So ist der Shop in der Lage, nicht mehr oder nur noch in geringen Mengen vorrätige Ware schnell und einfach nachzubestellen.

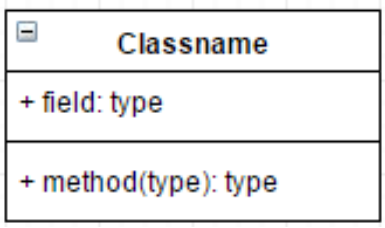
2 Konzeptionelle Beschreibung

Autor: Robin Hinz
Konzept: Corinna Koch, Robin Hinz, Jannis Fink

2.1 Unified Modeling Language Klassendiagramm

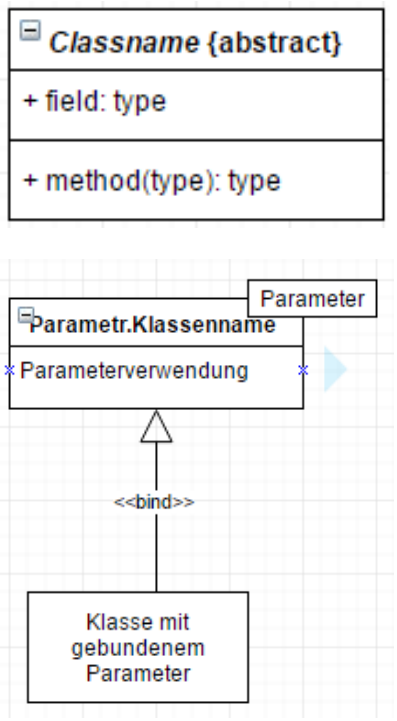
Ein UML Klassendiagramm dient zur Modellierung einer Datenbankstruktur. Dabei besteht die Datenbankstruktur aus Klassen, Schnittstellen und deren Bezeichnungen. Eine Klasse dient zur Verallgemeinerung von Objekten, dabei dient sie als abstrakter Oberbegriff für die Beschreibung der Struktur und das Verhalten der Objekte die sie gemeinsam hat. Im Klassendiagramm werden die einzelnen Klassen in Beziehungen zueinander gebracht um ein vollständiges Modell zu erhalten → 2.1 UML Klassendiagramm.

Dabei wird eine Klasse immer als ein Rechteck dargestellt. In diesem Rechteck befinden sich meist so um 2 bzw. 3 Unterteilungen. Laut UML-Spezifikation kann es auch noch weiter Unterteilungen geben. Der obere Bereich enthält dabei den Sterotyp, also das Paket zu dem die Klasse gehört und dessen Name. Der mittlere Bereich ist für die Attribute vorgesehen, also welche Eigenschaften das Objekt beinhalten soll. Und im unteren Bereich werden die Operatoren der Klasse angegeben.

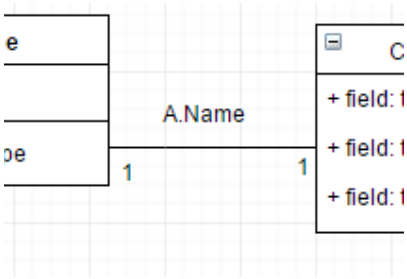


Eine Klasse kann auch abstrakt sein. Das Bedeutet das sie nicht instanziierbar ist. Um eine Klasse als abstrakt kenntlich zumachen, kann ihr Name kursiv geschrieben werden oder eben die Eigenschaft abstrakt in geschweiften Klammern mit angegeben werden.

Zudem gibt es auch noch die Parametrisierte Klasse auch Template oder Schablone genannt. Um kenntlich zumachen das es sich bei gegebener Klasse um eine Parametrisierte Klasse handelt wird in der oberen rechten Ecke an das Klassensymbol ein überlappendes Rechteck angefügt, welches die Schablonen-Parameter der Klasse enthält. Hier ist es wichtig die Funktion, die der Parameter enthält, mit anzugeben. Zudem ist es wichtig, durch eine eine gestrichelte Linie mit Pfeil, die Klasse die den Parameter bindet mit dem Template zu verbinden. Diese trägt die Bezeichnung «bind».



Eine Linie zwischen zwei Klassen nennt man Assoziation. Dabei stellt sie eine Beziehung zwischen den Klassen her, wodurch die Objekte miteinander kommunizieren können. Eine Assoziation kann auch einen Namen haben und die Multiplizität werden an ihr angegeben .

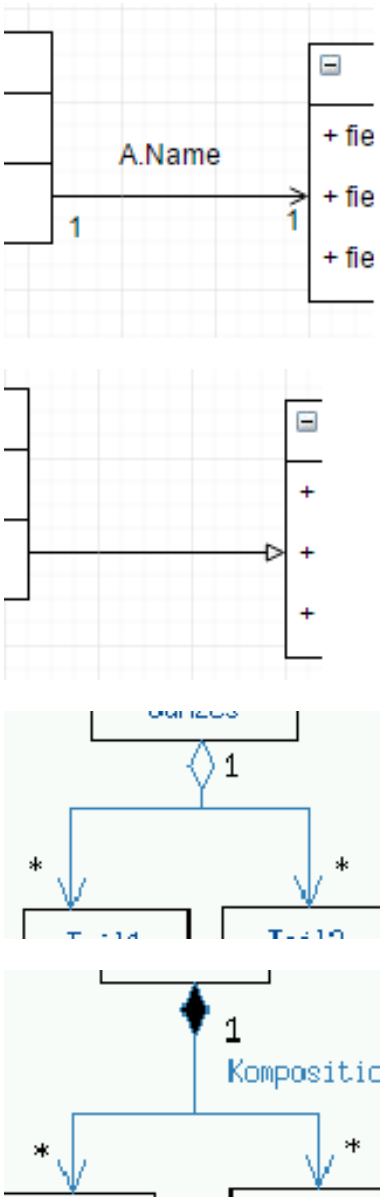


Es gibt auch gerichtete Assoziation. Hier wird mit Hilfe eines Pfeiles die Navigationsrichtung angegeben. Dabei drückt der Pfeil die Zugriffsrichtung aus.

Auch können Klassen vererben. Die Vererbung auch Generalisierung/Spezialisierung genannt, wird durch einen Pfeil mit dreieckigen Kopf gekennzeichnet.

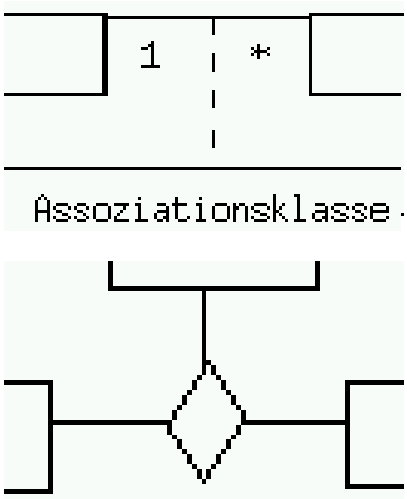
Durch eine Aggregation können auch Teile-Ganzes-Beziehung ausgedrückt werden. Dabei befindet sich eine Raute am ganzen Objekt, wo hingegen auf die Teilobjekte mit Pfeilen verwiesen wird.

Die Komposition wird gleich wie die Aggregation dargestellt, nur das hier die Raute ausgefüllt sein muss. Wie auch schon die Aggregation setzt diese Beziehung Teile zu einem Ganzen zusammen. Jedoch sind hier die Teile und das Ganze existenzabhängig.



Es kann vorkommen, dass eine Klasse von einer Assoziation zwischen zwei Klassen abhängig ist, diese Klasse wird dann als Assoziationsklasse bezeichnet. Dabei beschreibt sie die Eigenschaften, die der verbundenen Klassen nicht sinnvoll zuordenbar sind. Sie wird mit Hilfe einer gestrichelten Linie, mit der eigentlich Assoziation, verbunden. Der Name der Assoziationsklasse ist frei wählbar. Jedoch wenn die Assoziation einen Namen hat, muss die Assoziationsklasse den selben bekommen.

Als letzte Beziehung gibt es noch die mehrgliedrige Assoziation. Dabei können mehrere Klassen über ein Raute-Symbol in Beziehung treten.



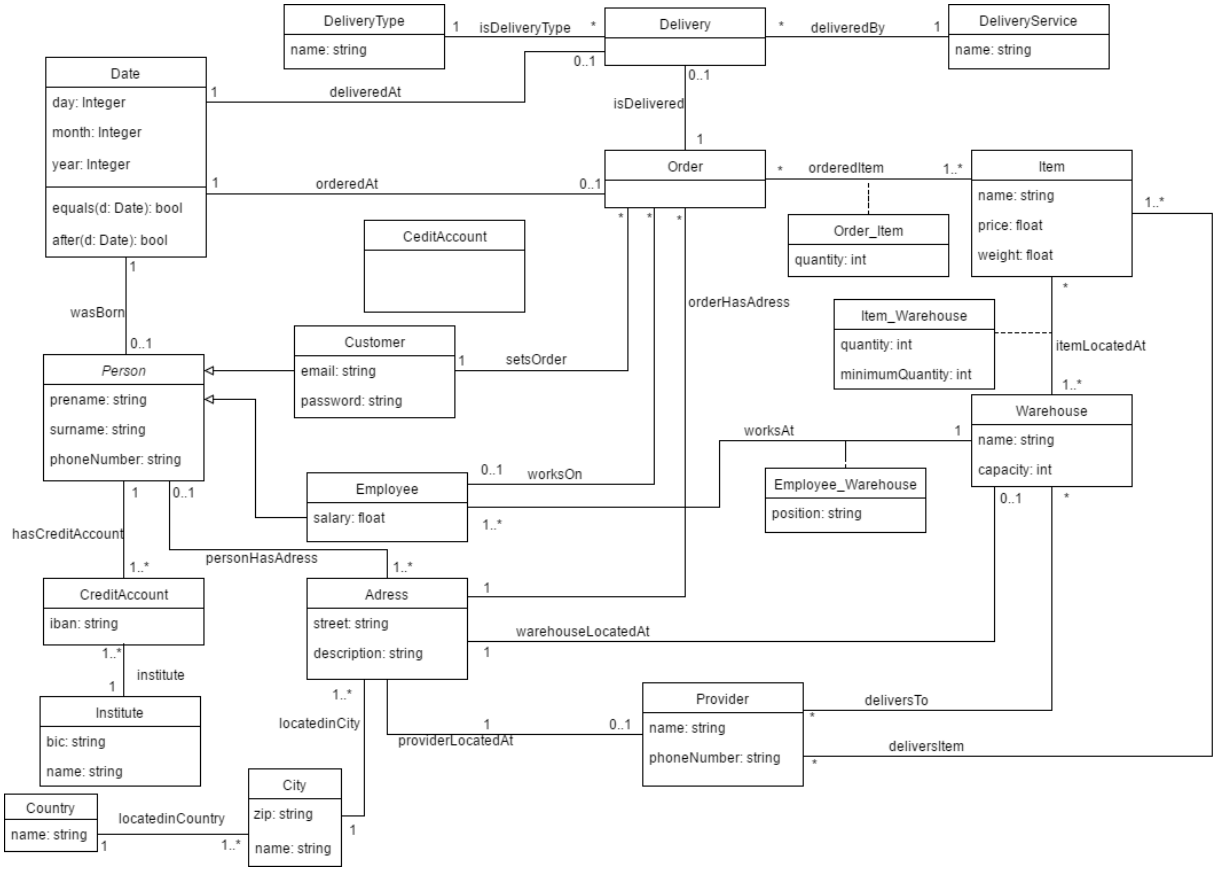


Abbildung 2.1: UML Klassendiagramm

2.2 Entity Relationship Diagramm

Ein Entity-Relationsh-Modell beschreibt mit Hilfe von Entitäten und ihren Beziehungen zueinander eine Abstraktion der realen Welt. Dabei sind Entitäten nur eine Abstraktion eines Objektes aus der realen Welt. Wie die meisten Objekte besitzt auch eine Entität Eigenschaften, die Attribute genannt werden. Verfügen Entitäten gleiche Attribute und können logisch miteinander Verknüpft werden, dann wird von einer Entitätsmenge gesprochen. Hier wird jedoch die Strukturbeschreibung als Entitätstyp bezeichnet. Ein Entitätstyp wird mit einer eindeutige Bezeichnung und mit dessen menge von Attributen beschrieben.

Wie schon erwähnt gibt es Beziehungen zwischen den Entitäten. Dadurch können sich Zusammenhänge zwischen ihnen entstehen, die optional auch eine Menge von Attributen besitzen können. Wie auch schon bei der Entität, lassen sich auch Beziehungen logisch zusammenfassen zu einer Beziehungsmenge, wobei diese dann als Beziehungstyp bezeichnet wird. Ein Beziehungstyp enthält dann die Menge der beteiligten Entitätstypen und die Menge der beteiligten Attribute → 2.2 ER-Diagramm.

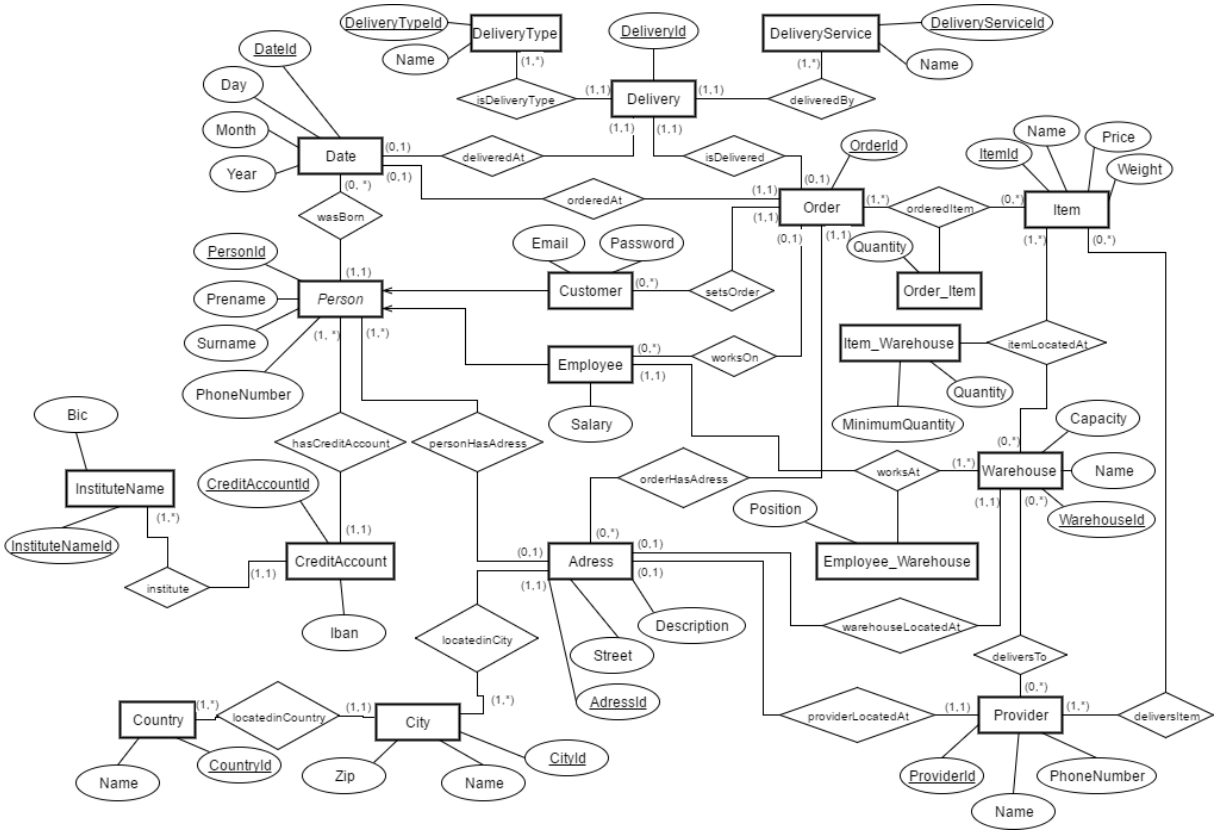
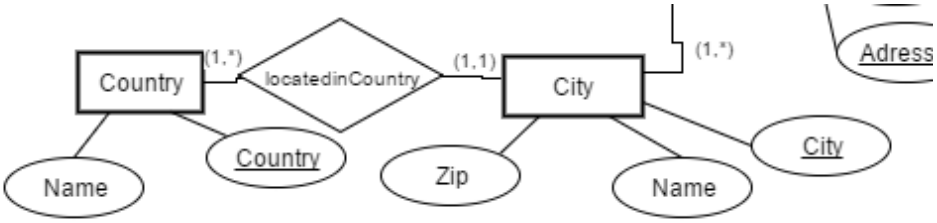


Abbildung 2.2: ER-Diagramm

Bei der Gegenüberstellung mit einem UML Klassendiagramm → 2.1 UML Klassendiagramm wirken beide doch sehr unterschiedlich, doch bei näherer Betrachtung fallen viele Gemeinsamkeiten auf. Aus diesem Grund wird auch in diesem Kapitel nicht noch einmal auf die gesamte Struktur eingegangen, sondern nur die wichtigsten unterschied gegenüber eines UML Klassendiagramm hervorge-

hoben.

Der erste große Unterschied der ins Auge sticht, wir haben es hier nicht mehr mit Klassen zu tun haben, sondern mit Entitäten. Daher ist die Darstellung im Diagramm eine andere als es noch bei einer Klasse war. Hier wird die Chen-Notation verwendet. Bei einer Entität wird der Name der Entität in einem Rechteck dargestellt und die Attribute, die die Entität ausmachen, ringsherum , mit Hilfe von Linien, mit dem Rechteck verbunden. Dabei befinden sich die Attribute in Ellipsen.



Zudem sind die Namen der Beziehungstypen in Raute vermerkt. Dabei könnte man auch die beiden Schlüsselwörter `is-a` für eine Vererbungsbeziehung oder `is-part-of` für eine Ganze-Teile-Beziehung verwenden. Analog dazu sind im UML-Klassendiagramm die Notationselemente für Aggregation/Komposition und Vererbungen in Ganze-Teile-Beziehungen grafisch dargestellt. Im E/R-Modell werden statt Kardinalitätsrestriktionen (Multiplizitäten) Klassifikation der Beziehungstypen angegeben. Diese zeigen (sowie im UML-Klassendiagramm) an, wie viele Entitätstypen bei einer Beziehung auf beiden Seiten beteiligt sind. Auch die Vergabe eines Primärschlüssels und das beinhalten an schwachen Entitäts-Mengen sind dem E/R-Modelle vorbehalten.

2.3 Allgemeine Beschreibung der Diagramme

In der Klasse `Person` befinden sich 3 Attribute, die jeweils von Datentyp String sind. Die Attribute sind `prename`, `surname` und `phoneNumber`. `Person` wird erweitert von den Klassen `Customer` und `Employee`. Dabei hat `Customer` die erweiterungs-Attribute `email` und `password` des Datentyps String und `Employee` die Attribute `salary` des Datentyp Float. Durch diese Attributs Aufteilung kann ein Arbeiter nicht gleich ein Kunde sein. Um ein Kunde zu werden müsste er sich ein Kundenkonto erstellen. Die beiden erweiterungs-Klassen haben auch noch die gemeinsame Verbindung zu der Klasse `Order`. Diese Klasse hat keinerlei Attribute, da sie ausschießlich als Verbindungsklasse dient. Dabei kann ein Kunde so viele Bestellungen aufgaben wie er will, jedoch werden dann die Bestellungen von einen bzw. keinen Arbeiter bearbeitet. Zudem hat `Order` und `Person` eine Verbindung zu `Adress`. Dabei kann eine Person beliebig viele Adressen besitzen, jedoch muss sie mindestens eine besitzen. Eine Adresse muss aber jedoch nicht unbedingt einer Person zugeordnet werden, wiederum muss sie mindestens einer Bestellung bzw. der Klasse `Order` zugewiesen werden. Um die 3. Normalform nicht zu verletzen wurde `Adress` um die Klassen `City` und `Country` erweitert. Dabei besitzt `Adress` nur noch die Attribute `street` und `description`, `City` die Attribute `zip` und `name` und `Country` nur noch `name`. Alle Attribute sind vom Typ String. Damit die Firma auch Profit macht und die Arbeiter nicht anfangen zu Streiken, sollte jeder Person mindestens ein Bankkonto zugewiesen werden, über das ein Geldfluss stattfinden kann. Dass ist auch die Verbindung von der Klasse `Person`

zu CreditAccount. Auch CreditAccount wurde unter Beachtung der 3. Normalform unterteilt und besitzt ausschließlich Attribute des Typs String. Dabei besitzt CreditAccount nur das Attribut iban und Institute die Attribute bic und name.

Um sicherzustellen, dass unser Shop keine Kinderarbeit unterstützt und das Kinder nicht in Genuss des neuen Alien Films kommen, muss Person noch ein Geburtsdatum bekommen. Da aber auch andere Klassen ein Datum benötigen, wurde dies in eine eigene Klasse ausgelagert. Die Klasse Date wurde uns von Frank Hilken zur Verfügung gestellt und beinhaltet die Attribute day, month und year des Typs Integer und die Booleans equals und after. Dadurch hat jede Person ein festes Geburtsdatum, jedoch ist nicht am jedem Datum eine Person geboren die mit unseren Shop in Verbindung steht. Auch eine Bestellung hat ein Datum, dabei besitzt Örder"die gleich Beziehung "Date"wie schon Person zu Date.

Was wäre eine Bestellung ohne Waren? Um diese Frage zu Umgehen gibt es die Klasse Item mit dem Attributen name (String), price und weight (Float). Da eine Bestellung ein Produkt auch mehrmals enthalten kann gibt es zwischen Order und Item noch die Associations-Klasse Order_Item. Diese enthält auch nur ein Attribut quantity von den Typ Integer, da man ja nichts halbes empfangen möchte. Eine Bestellung muss aber mindestens ein Item enthalten. Da wir mit Amazon und Co. mithalten wollen, sollten unsere Lieferzeiten entsprechen kurz sein. Um dies zu erreichen sind die meisten Produkte direkt bei uns schon vorrätig und müssen nicht erste nach einer Bestellung beim Hersteller geordert werden. Daher wurden Lagerhäuser angemietet und natürlich in unseren Datenbanksystem mit eingebunden. Die Klasse trägt den Namen Warehouse und beinhaltet die Attribute name (String) und capacity (int). Dabei kann eine beliebige Anzahl eines oder mehreren Produkten, in dem selben oder unterschiedlichen Lagerhäusern liegen, da jedes Lagerhaus nur eine begrenzte Anzahl an Kapazität besitzt. Auch ein Lagerhaus muss eine Adresse besitzen und wie auch schon bei Person muss nicht jede Adresse an einen Lagerhaus vergeben sein, sonst würde es nur noch Lagerhäuser in Deutschland geben. Da in einem Lagerhaus ohne Menschen nichts los ist, benötigt es mindesten noch eine Personen die darin arbeiten. Jedoch gibt es nicht nur einen Job in der Logistik zu erledigen, daher wurde die Association-Klasse Employee_Warehouse in der Beziehung von Employee und Warehouse mit eingebunden. Employee_Warehouse enthält auch nur das Attribut position des Typs String.

Um zu wissen ob ein Produkt überhaupt noch vorrätig ist, gibt es die Associations-Klasse Item_Warehouse mit den Informationen quantity (int) und minimumQuantity (int). Das Attribut minimumQuantity ist wichtig um zu wissen wann eine Nachbestellung beim Hersteller getätigt werden muss. Auch die Hersteller/Lieferanten sind in der Datenbank mit eingebunden. Dabei enthält ihre Klasse Provider die Attribute name und phoneNumer, die jeweils vom Typ String sind. Auch diese Klasse hat eine Beziehung zu Adresse, die die gleich wie Adress zu Warehouse oder zu Person ist. Zudem hat Provider auch eine wichtige Beziehungen zu Warehouse. Denn sollte ein Produkt im Lager ausgehen, also die minimal Anzahl unterschreiten, kann es direkt bei verschieden Lieferanten bzw. Herstellern, in beliebiger Anzahl, nachbestellt werden. Da es unmöglich ist jedes Produkt immer auf Lager zuhaben, besteht auch eine direkte Beziehung von Provider zu Item. Hier kann von beliebig vielen Hersteller/Lieferanten mindestens ein Produkt bzw. beliebig viele Produkte geordert werden.

Jetzt muss nur noch die Bestellung zum Kunden gelangen. Dieses Problem löst man am besten mit einen oder mehreren Lieferservicen. Daher gibt es die Hilfsklasse Delivery die wie Order keine At-

tribute besitzt. Sie verbindet die Klassen `DeliveryService` und `DeliveryType` mit der Klasse `Order`. `DeliveryService` und `DeliveryType` haben jeweils nur ein Attribut mit dem Namen `name` und des Typs String. Dabei kann Nur ein Lieferservice eine Lieferung auf eine Art ausführen. Mit Art ist gemeint, ob es sich um eine Standart- oder Expresslieferung handelt. Es eine Bestellung getätigt muss es nicht unbedingt eine Lieferung geben, da eine Bestellung noch keiner Lieferung zugeordnet sein kann. Natürlich ist für die Garantie noch ein Lieferdatum wichtig. Daher gibt es zwischen `Delivery` und `Date` die gleiche Verbindung wie schon `Date` und `Order` sie hat.

Für das E/R-Modelle wurden zudem noch für jede Entität Primärschlüssel in Form von Id's, die den Namen der Entität tragen, vergeben. Die Association-Klassen beinhalten dabei beide Primärschlüssel der verbundenen Entitäten als Sekundärschlüssel.

Vergleiche auch [Michael Guist (2017), unbekannt (2016a), unbekannt (2016b)]

3 Integritätsbedingungen

Autor: Corinna Koch(Beschreibung und RA)
Konzept: Corinna Koch, Robin Hinz, Jannis Fink

In diesem Kapitel beschreiben wir unsere Integritätsbedingungen. Diese haben wir in der Object Constraint Language (OCL) und der Relationenalgebra (RA) definiert.

3.1 Informelle Definition und Erläuterungen

Integritätsbedingungen sind dazu da, die Konsistenz von Daten sicher zu stellen. Beim Anlegen und Ändern von Daten können inkonsistente oder ungültige Daten entstehen können, welche zu ungewollten bzw. falschen Zuständen führen können, wenn keine Integritätsbedingungen beachtet werden.

Die Konsistenz der Daten wird zum Beispiel durch Primärschlüssel (da diese eindeutig sein müssen) und Multiplizitäten sichergestellt. Außerdem gibt es noch Bedingungen, die sich aus der informellen Beschreibung ergeben und noch formal definiert werden müssen.

Wie bereits oben erwähnt haben wir für die formale Beschreibung zusätzlich zu der Beschreibung in OCL noch RA gewählt.

Der Relationenalgebra sind dabei allerdings Grenzen gesetzt.
Man kann zum Beispiel nicht überprüfen, ob ein Wert `NULL` ist. Außerdem können Strings nur auf Gleichheit überprüft werden, aber es kann nicht getestet werden, ob ein String einen bestimmten Substring beinhaltet. An dieser Stelle könnte man RA um die SQL Operation `LIKE` erweitern, wobei diese, da sie eine boolesche Operation ist, in der Formel φ von $\sigma_{\varphi}(R)$ eingesetzt werden könnte.

Eine wichtige Grenze wird in den Vorlesungsfolien wie folgend beschrieben:

“Es gibt keinen Term der Relationenalgebra, der zu jeder zweistelligen Relation R deren transitive Hülle $R*$ berechnet.“ [Gogolla (2016)].

Dadurch kann zum Beispiel die SQL Funktion GROUP BY nicht umgesetzt werden, aber eine Funktion GROUP BY^2, die immer 2 Elemente zusammenfasst.
Man könnte in RA zum Beispeil eine Operation `SUM` einfügen, die die Summe aller Werte einer Menge berechnet. $SUM(\pi_A(R))$ wäre also $\sum_{i=1}^{m} a_i$ mit $\forall a_j \in A$ und $\#A = m$. Allerdings kann man nicht wie in SQL die Werte vorher z.B. nach der ID gruppieren, da man nicht weiß, wie viele Werte zusammengefasst werden sollen. Dadurch kann die Operation `SUM` nicht immer sinnvoll eingesetzt werden bzw. es könnten falsche Werte entstehen. Sie ist also ungeeignet für RA. Aus diesem Grund konnten

wir `capacityBiggerThanItemsInside` nicht in RA umsetzen (siehe → 3.2.7 Kontext Warehouse).

Es können zwischen verschiedenen Integritätsbedingungen Abhängigkeiten bestehen.
So wird beispielsweise durch die Multiplizitäten sichergestellt, dass ein Lagerhaus mindestens einen Mitarbeiter hat.
Eine weitere Abhängigkeit besteht bei uns beim Kontext `Item_Warehouse`. Für diesen Kontext wird überprüft, ob die minimale Anzahl größer als 0 ist (Invariante `intGreaterZero`) und ob die Anzahl der gelagerten Artikel größer als die Minimalanzahl ist (Invariante `enoughItemsInStock`). Dadurch wird implizit auch geprüft, ob die Anzahl der gelagerten Artikel größer als 0 ist. D.h., diese Invariante funktioniert nur korrekt, wenn `intGreaterZero` funktioniert (siehe auch → 3.2.9 Kontext Item_Warehouse).
Eine weitere Abhängigkeit besteht zwischen `belongsToOneObject` und `adressIsCustomer-Adress` (siehe auch → 3.2.10 Kontexte Adress und Date).

3.2 Formale Definition in OCL und RA

3.2.1 attrsNotNull und stringsNotEmpty

Für jede Klasse haben wir überprüft, ob jedem Attribut ein Wert zugewiesen wird. Eine Ausnahme bildet hierbei die Klasse `Adress` mit dem Attribut `description`, welches Null sein darf.

Die dazugehörigen Invarianten heißen `attrsNotNull` und wurde nur in OCL umgesetzt, da in RA nicht auf Null geprüft werden kann.
Für die Klasse `Customer` sehen diese wie folgend aus:

```
context Customer
  inv attrsNotNull:
    (not self.email.isUndefined())
    and (not self.password.isUndefined())
    and (not self.prename.isUndefined())
    and (not self.surname.isUndefined())
    and (not self.phonenumber.isUndefined())
```

Desweiteren haben wir bei allen Attributen vom Typ String sicher gestellt, das diese keinen leeren String enthalten.

Die Invarianten heißen in OCL `stringsNotEmpty`:

```
context Customer
  inv stringsNotEmpty:
    (self.email <> '')
    and (self.password <> '')
    and (self.prename <> '')
    and (self.surname <> '')
    and (self.phonenumber <> '')
```

In RA können sie definiert werden, indem wir sicherstellen, dass wenn ein Attribut aus einem leeren String besteht, die Ergebnismenge nicht leer ist:

$$\begin{aligned}&\sigma_{Customer.email=''}(CUSTOMER)\\ \cup\ &\sigma_{Customer.password=''}(CUSTOMER)\\ \cup\ &\sigma_{Customer.prename=''}(CUSTOMER)\\ \cup\ &\sigma_{Customer.surname=''}(CUSTOMER)\\ \cup\ &\sigma_{Customer.phonenumber=''}(CUSTOMER) = \emptyset\end{aligned}$$

Alternativ könnte man auch schreiben:

$$\begin{aligned}&\sigma_{\varphi}(CUSTOMER) = \emptyset\\ \text{mit } \varphi = \ &Customer.email = ''\\ &\vee Customer.password = ''\\ &\vee Customer.prename = ''\\ &\vee Customer.surname = ''\\ &\vee Customer.phonenumber = ''\end{aligned}$$

Für die Klassen `DeliveryType`, `DeliveryService` und `Country` und die Assoziationsklasse `Employee_Warehouse` sind `attrsNotNull` und `stringsNotEmpty` die einzigen definierten Invarianten.

3.2.2 Kontexte Customer, Employee und Provider

Gemeinsame Invarianten

Im Kontext der Klassen `Customer`, `Employee` und `Provider` haben wir überprüft, ob es sich bei der angegebenen Telefonnummer um eine valide Nummer handelt. Dabei haben wir vereinfacht angenommen, das eine valide Telefonnummer ein String ist, der nur aus Ziffern und Leerzeichen besteht.

Die Invariante haben wir mit `validPhonenumber` bezeichnet:

```
1  inv validPhonenumber:
2    let number = self.phonenumber in
3    Sequence{1 .. number.size()}
4    ->forAll(i | (not number.substring(i, i).toInteger().oclIsUndefined() )
     or number.substring(i, i) = ' ')
```

Diese OCL Invariante kann nicht mit RA überprüft werden, da Strings in RA nicht näher untersucht werden können.

Für den Provider sind außer `attrsNotNull`, `stringsNotEmpty` und `validPhonenumber` keine weiteren Invarianten definiert.

Invarianten Customer

Außerdem haben wir das Attribut `email` der Klasse `Customer` überprüft. Auch hier haben wir die Validierung vereinfacht, in dem wir angenommen haben, das eine korrekte Mailadresse genau einmal das Zeichen "@" enthält:

```
context Customer
  inv validMailAdress:
    let email = self.email in
    Sequence{1 .. email.size()}->exists(i | email.substring(i, i) = '@')
```

Wie oben bereits erwähnt, ist diese OCL Invariante nicht mit RA überprüfbar.

Für die Emailadresse eines Kunden muss weiterhin gelten, dass sie eindeutig ist, d.h. kein anderer Kunde darf dieselbe Mailadresse haben.

In OCL gehen wir dafür alle Instanzen der Klasse `Customer` durch und vergleichen jede Instanz mit der aktuellen Instanz, für die diese Invariante aufgerufen wird (`self`). Der Term gibt `true` zurück, wenn es sich bei der Instanz um sich selbst handelt oder die Mailadressen der Instanzen unterschiedlich sind.

```
  inv uniqueMail:
    Customer.allInstances()
    ->forAll(c | c = self or c.email <> self.email)
```

Um diese Invariante in RA zu überprüfen, werden aus dem kartesischem Produkt der Relationen $CUSTOMER \times CUSTOMER$ die Zeilen selektiert, die verschiedene Ids, aber gleiche Mailadressen haben. Die Ergebnismenge soll leer sein.

Die Umbenennung von z.B. $person\_id$ nach $person\_id1$ kann hier weggelassen werden, da sie implizit erfolgt.

$$\sigma_{person\_id1 \neq person\_id2 \wedge email1 = email2}(CUSTOMER \times CUSTOMER) = \emptyset$$

Invarianten Employee

Für die Klasse `Employee` haben wir festgelegt, das jeder Mitarbeiter nur eine Adresse haben darf (`hasOnlyOneAdress`), welche im selben Land liegen muss wie der Arbeitsplatz des Mitarbeiters, also das Lagerhaus in dem er arbeitet (`livesInSameCountryAsWarehouse`).

```
context Employee
  inv hasOnlyOneAdress:
    self.adress->size() = 1
```

```
4   inv livesInSameCountryAsWarehouse:
5     self.adress->asSequence()->first().city.country.name
6     = self.warehouse.adress.city.country.name
```

In RA überprüfen wir `hasOnlyOneAdress`, indem wir zunächst alle Einträge von Mitarbeitern in der Tabelle `person_adress` selektieren und anschließend sicherstellen, dass es keine zwei Einträge in der Ergebnisrelation gibt, die dieselbe `person_id` haben.
Wäre dies der Fall, hätte ein Mitarbeiter zwei (oder mehr) Adressen.

$$E\_PA := \sigma_{Employee.person\_id=Person\_Adress.person\_id}(EMPLOYEE \times PERSON\_ADRESS)$$

$$PERSON\_ADRESS' := \pi_{Person\_Adress.person\_id,Person\_Adress.adress\_id}(E\_PA)$$

$$\sigma_{person\_id1=person\_id2 \wedge adress\_id1 \neq adress\_id2}(PERSON\_ADRESS' \times PERSON\_ADRESS') = \emptyset$$

Um `livesInSameCountryAsWarehouse` in RA überprüfen zu können, haben wir zunächst mit dem natürlichen Verbund die Tabellen `adress`, `city` und `country` zusammengefügt und mithilfe von Projektion und Umbenennung die benötigten Attribute ausgewählt:

$$\begin{aligned} PA &:= \pi_{person\_id,person\_adress\_id}(\delta_{person\_adress\_id \leftarrow adress\_id}(PERSON\_ADRESS)) \\ EW &:= \pi_{person\_id,warehouse\_id}(EMPLOYEE\_WAREHOUSE) \\ W &:= \pi_{id,warehouse\_adress\_id}(\delta_{warehouse\_adress\_id \leftarrow adress\_id}(WAREHOUSE)) \end{aligned}$$

Für die Hilfs-Relation A erfolgt die Umbenennung impliziet.

$$\begin{aligned} ADRESS' &:= (ADRESS *_{city\_id=id} CITY) *_{country\_id=id} COUNTRY \\ A &:= \pi_{id1,id2,country1,country2}(ADRESS' \times ADRESS') \end{aligned}$$

Über den natürlichen Verbund haben wir die Relationen PA, EW und W nun zusammengefasst, PA und EW werden dabei implizit über die person_id verbunden:

$$S := (PA * EW) *_{warehouse\_id=id} W$$

Das Schema S sieht also nun wie folgend aus:
$S(person\_id, warehouse\_id, person\_adress\_id, warehouse\_adress\_id)$.

Nun kann überprüft werden, ob es Einträge mit ungleichen Adressen und ungleichen Ländern gibt. Im korrekten Fall ist die Menge dieser Einträge leer.

$$\sigma_{\varphi}(S \times A) = \emptyset \text{ mit } \varphi = S.person\_adress\_id = A.id1 \\ \wedge warehouse\_adress\_id = A.id2 \\ \wedge A.id1 \neq A.id2 \\ \wedge A.country1 \neq A.country2$$

Die letzte Invariante zum Kontext `Employee` besagt, dass das Gehalt eines Mitarbeiters größer 0 sein muss.

```
1  inv intGreaterZero:
2    self.salary > 0
```

In RA haben wir es wie folgend überprüft:

$$\sigma_{Employee.salary \leq 0}(EMPLOYEE) = \emptyset$$

3.2.3 Kontexte CreditAccount und Institute

Für die Attribute `iban` von `CreditAccount` und `bic` von `Institute` gilt das selbe wie für die Mailadresse des Kunden - sie müssen eindeutig sein:

```
1  context CreditAccount
2    inv uniqueIban:
3      CreditAccount.allInstances()
4      ->forAll(ca | ca = self or ca.iban <> self.iban)
5
6  context Institute
7    inv uniqueBic:
8      Institute.allInstances()
9      ->forAll(i | i = self or and i.bic <> self.bic)
```

Auch die Überprüfung in RA funktioniert analog:

$$\sigma_{id1 \neq id2 \wedge iban1 = iban2}(CREDITACCOUNT \times CREDITACCOUNT) = \emptyset$$
$$\sigma_{id1 \neq id2 \wedge bic1 = bic2}(INSTITUTE \times INSTITUTE) = \emptyset$$

3.2.4 Kontext Delivery

Bei der Klasse `Delivery` wollen wir sicher stellen, dass das Lieferdatum nach dem Bestellungsdatum liegt:

```
context Delivery
  inv deliveryDateAfterOrder:
    self.date.after(self.order.date)
```

In RA sieht die Überprüfung wie folgend aus:

$$\sigma_{Delivery.date \leq Order.date}(\sigma_{Delivery.id=Order.delivery\_id}(DELIVERY \times ORDER)) = \emptyset$$

Wir gehen hier vereinfacht davon aus, dass die Attribute `date` direkt mit $<, \leq, \geq, >$ verglichen werden können (eigentlich müsste man die Attribute `day`, `month` und `year` vergleichen).

3.2.5 Kontext City

Im Kontext `City` muss gesichert werden, dass Städte eindeutige Postleitzahlen haben.

Die Überprüfung in OCL erfolgt analog wie schon bei `uniqueMail` von `Customer`, `uniqueIban` von `CreditAccount` und `uniqueBic` von `Institute`.

```
context City
  inv uniqueZip:
    City.allInstances()
    ->forAll(c | c = self or c.zip <> self.zip)
```

Auch die Überprüfung in RA funktioniert analog:

$$\sigma_{id1 \neq id2 \wedge zip1 = zip2}(CITY \times CITY) = \emptyset$$

3.2.6 Kontexte Item und Order_Item

Ein Artikel muss einen Preis und ein Gewicht größer 0 haben, außerdem muss die Anzahl eines bestellten Artikels größer als 0 sein.

```
context Item
  inv intGreaterZero:
    self.price > 0
    and self.weight > 0

context Order_Item
  inv intGreaterZero:
    self.quantity > 0
```

In RA haben wir es wie folgend überprüft:

$$\sigma_{Item.price \leq 0}(ITEM)$$
$$\cup\, \sigma_{Item.weight \leq 0}(ITEM) = \emptyset$$

$$\sigma_{Order\_Item.quantity \leq 0}(ORDER\_ITEM) = \emptyset$$

3.2.7 Kontext Warehouse

Bei der Klasse `Warehouse` muss zunächst überprüft werden, ob jedes Warenhaus einen Lagerleiter hat:

```
context Warehouse
  inv hasLagerleiter:
    Employee_Warehouse.allInstances()
    ->select(ew | ew.position = 'Lagerleiter')
    ->collect(ew | ew.warehouse)
    ->includesAll(Warehouse.allInstances)
```

In RA prüfen wir, ob die Menge der Einträge in der Tabelle `warehouse` eine Teilmenge der Einträge der Tabelle `employee_warehouse` mit der Position Lagerleiter ist. Verglichen werden die Einträge über die jeweilige Lagerhaus-Id der Tabellen:

$$\pi_{id}(WAREHOUSE)$$
$$\subseteq \pi_{warehouse\_id}(\sigma_{employee\_warehouse.position='Lagerleiter'}(EMPLOYEE\_WAREHOUSE))$$

Desweiteren gilt für ein Lagerhaus, das seine Kapazität nicht unter oder gleich 0 sein darf:

```
  inv intGreaterZero:
    self.capacity > 0
```

In RA kann dies wie folgend überprüft werden:

$$\sigma_{Warehouse.capacity \leq 0}(WAREHOUSE) = \emptyset$$

Außerdem darf die Summe alle Artikel in einem Lagerhaus maximal der Kapazität des Lagerhauses entsprechen. Alle Artikel in einem Lagerhaus sind dabei die Instanzen von `Item_Warehouse`, deren Attribut `warehouse` dem zu überprüfenden `Warehouse` entspricht.

In OCL haben wir diese Invariante wie folgend umgesetzt:

```
  inv capacityBiggerThanItemsInside:
    Item_Warehouse.allInstances()
    ->select(iw | iw.warehouse = self)
    ->collect(iw | iw.quantity)->sum() <= self.capacity
```

Diese Invariante kann wie in der Einleitung beschrieben nicht in RA dargestellt werden.

3.2.8 Kontext Order

Bei der Klasse `Order` muss darauf geachtet werden, dass die Adresse der `Order` einer der Adressen des `Customer` der `Order` entspricht.

```
context Order
  inv adressIsCustomerAdress:
    self.customer.adress->includes(self.adress)
```

In RA haben wir zunächst `Order` und `Person_Adress` mit dem natürlichen Verbund zusammen gefügt und die Attribute von `person_id` selektiert, die dieselben Adressen haben. Die Menge der Attribute `customer_id` von `Order` muss eine Teilmenge des Ergebnisses sein.

$$R := (\delta_{order\_adress\_id \leftarrow adress\_id}(ORDER)) *_{customer\_id=person\_id} (\delta_{person\_adress\_id}(PERSON\_ADRESS))$$
$$S := \pi_{person\_id}(\sigma_{order\_adress\_id=person\_adress\_id}(R))$$
$$\pi_{customer\_id}(ORDER) \subseteq S$$

3.2.9 Kontext Item_Warehouse

Bei der Assoziationsklasse `Item_Warehouse` muss die minimale Anzahl, in der Artikel vorhanden sein sollen, größer 0 sein.
Außerdem muss für die Anzahl der Artikel (Attribut `quantity`) gelten, dass sie mindestens der minimalen Anzahl (Attribut `minimumQuantity`) entspricht. Da `minimumQuantity` größer als 0 sein muss, ist die Anzahl auch größer als 0. Daher wurde für `quantity` die Invariante `intGreaterZero` weggelassen.

```
context Item_Warehouse
  inv intGreaterZero:
    self.minimumQuantity > 0
  inv enoughItemsInStock:
    self.quantity >= self.minimumQuantity
```

`intGreaterZero` in RA:

$$\sigma_{Item\_Warehouse.minimumQuantity \leq 0}(ITEM\_WAREHOUSE) = \emptyset$$

Um die Invariante `enoughItemsInStock` in RA abzudecken, selektieren wir alle Vorkommnisse, bei denen die Anzahl (`quantity`) unter der minimal erlaubten Anzahl liegt (`minimumQuantity`). Die resultierende Menge muss leer sein.

$$\sigma_{Item\_Warehouse.quantity<Item\_Warehouse.minimumQuantity}(ITEM\_WAREHOUSE) = \emptyset$$

3.2.10 Kontexte Adress und Date

Für die Klassen `Adress` und `Date` mussten wir sicher stellen, das jede Instanz der Klassen nur einer anderen Klasse zugeordnet ist. Würden nämlich beispielsweise das Attribut `adress_id` einer Person und eines Lagerhauses auf die selbe Adresse verweisen, würde bei einer Adressänderung durch die Person (zum Beispiel wegen eines Umzuges) auch die Lagerhausadresse geändert werden. Um dies zu verhindern, haben wir mithilfe der disjunktiven Normalform folgende Invarianten erstellt:

```
1  context Adress
2    inv belongsToOneObject:
3      (self.person.isUndefined() and self.warehouse.isUndefined() and (not
       self.provider.isUndefined()))
4      or (self.person.isUndefined() and (not self.warehouse.isUndefined())
       and self.provider.isUndefined())
5      or ((not self.person.isUndefined()) and self.warehouse.isUndefined()
       and self.provider.isUndefined())
6
7  context Date
8    inv belongsToOneObject:
9      (self.person.isUndefined() and self.delivery.isUndefined() and (not
       self.order.isUndefined()))
10     or (self.person.isUndefined() and (not self.delivery.isUndefined()) and
        self.order.isUndefined())
11     or ((not self.person.isUndefined()) and self.delivery.isUndefined() and
        self.order.isUndefined())
```

Eine Adresse könnte auch einer Bestellung zugeordnet sein. Dies kann aber in `belongsToOneObject` außer Acht gelassen werden, da durch die Multiplizitäten sicher gestellt ist, dass die Bestellung auch einem Kunden zugeordnet ist. Durch die Invariante `adressIsCustomerAdress` wiederum ist sichergestellt, dass die der Bestellung zugeordnete Adresse zu den Adressen des Kunden gehört.

Bei RA überprüfen wir, ob die `adress_id` bzw. `date_id` Paare der betreffenden Klassen alle verschieden sind.

$$\sigma_{\varphi}(PERSON \times (DELIVERY \times ORDER)) = \emptyset$$
$$\text{mit } \varphi = Person.was\_born\_date\_id = Delivery.delivery\_date\_id$$
$$\vee\ Person.was\_born\_date\_id = Order.date\_id$$
$$\vee\ Delivery.delivery\_date\_id = Order.date\_id$$

$$\sigma_{\varphi}(PERSON\_ADRESS \times (WAREHOUSE \times PROVIDER)) = \emptyset$$
$$\text{mit } \varphi = Person\_Adress.adress\_id = Warehouse.adress\_id$$
$$\vee\ Person\_Adress.adress\_id = Provider.date\_id$$
$$\vee\ Warehouse.adress\_id = Provider.adress\_id$$

4 Systemzustände

Autor: Robin Hinz
Konzept: Corinna Koch, Robin Hinz, Jannis Fink

Systemzustände dienen dazu das Datenbankmodel zu testen und zu schauen ob es überhaupt zu realisieren ist. Dafür werden Zustände erzeugt, die Invarianten und modelleigene Bedingungen erfüllen. Es werden jedoch auch Zustände erzeugt, die nicht erwünscht sind, da man mit diesen die Wirksamkeit der Invarianten und Multiplizitäten zeigen kann.

In dieser Hausarbeit wurde dafür das in der Vorlesung bereitgestellte Tool USE verwendet.

Um ein Zustand zu instanziieren, müssen zuerst alle Klassen des Modells erstellt werden. Da fast jede Klasse Attribute besitzt, müssen diesen noch Werte zugewiesen werden. Dabei ist zu beachten um welchen Datentyp es sich bei den jeweiligen Attribut handelt. Anschließend können noch Beziehungen zwischen den Attributen und Klassen hergestellt werden.

Dazu gibt es folgendes Beispiel, in welchem alle Invarianten und modelleigenen Bedingungen erfüllt sind.

```
-- Warehouses
!create house1 : Warehouse
!set house1.name := 'Lagerhaus 1'
!set house1.capacity := 1000
!create adressHouse1 : Adress
!set adressHouse1.street := 'Lagerstrasse 1'
!set adressHouse1.description := 'Lagerhaus 1'
!insert (adressHouse1, bremen) into locatedInCity
!insert (house1, adressHouse1) into warehouseLocatedAt
```

Interessanter ist jedoch ob unsere Invarianten und Multiplizitäten ihre Funktion als diese erfüllen. Dafür wurden weitere Zustände erzeugt, um diese auf ihre Korrektheit zu testen.

Als erstes wurde getestet, ob erkannt wird, wenn alle Attribute `NULL` sind. Dazu wurden alle Attribute aus den Klasse gelöscht. Der anschließende Test war erfolgreich, da für jeder Klasse `attrsNotNull` mit `FAILED` als gescheitert bestätigt wurde. Nur für das Attribut `description` wurde kein gescheitert ausgegeben.

Da `description` auch `NULL` bzw. leer sein darf, wurde dies in einem weiteren Zustand überprüft.

```
-- adress.description := empty string
!create adressMax1 : Adress
!set adressMax1.street := 'Musterstrasse 1'
!set adressMax1.description := ''
```

```
-- adress.description := undefined
!create adressMax2 : Adress
!set adressMax2.street := 'Bibliothekstrasse 1'
```

Der darauf folgende Test wurde mit ok bestätigt.

Auch dürfen die Strings, bis auf `description`, nicht leer sein. Daher ließen wir als nächstes alle Attribute leer.

```
!set max.prename := ''
!set max.surname := ''
!set max.phonenumber := ''
```

Die Bedingung `stringsNotEmpty` wurde mit `FAILED` in diesem Fall korrekt erfüllt.

Eine Lieferungsdatum sollte auch nicht vor einer Bestelldatum liegen. Dies haben wir in einem weiteren Zustand überprüft.

```
!set deliveryDate.day := 6
!set deliveryDate.month := 12
!set deliveryDate.year := 2015
!create orderDate : Date
!set orderDate.day := 3
!set orderDate.month := 12
!set orderDate.year := 2016
```

Anschließend wurde überprüft, ob unsere Angestellten auch nur eine Adresse haben dürfen.

```
!create adressDieter : Adress
!set adressDieter.street := 'Dieterstrasse 3'
!set adressDieter.description := 'Dieters Haus'
!insert (adressDieter, bremen) into locatedInCity
!create adressDieter2 : Adress
!set adressDieter2.street := 'Dieterstrasse 5'
!set adressDieter2.description := 'Dieters 2. Haus'
!insert (adressDieter2, bremen) into locatedInCity
```

Nachdem dieser Test auch erfolgreich war, widmeten wir uns dem Lagerhaus. Hier sollte die Anzahl an Artikeln nicht die minimale Anzahl unterschreiten.

```
!set beer_house1.quantity := 50
!set beer_house1.minimumQuantity := 100
```

Jedoch sollte eine Gleichheit als funktionierender Zustand erkannt werden.

```
!set beer_house1.quantity := 100
!set beer_house1.minimumQuantity := 100
```

Anschließend musste noch überprüft werden ob für die Menge an Produkten genügen Kapazität im Lager ist, also ob die Kapazität größer als die Menge ist.

```
!create house1 : Warehouse
!set house1.name := 'Lagerhaus 1'
!set house1.capacity := 100
!create beer_house1 : Item_Warehouse between (beer, house1)
!set beer_house1.quantity := 200
!set beer_house1.minimumQuantity := 100
```

Auch dieser Vergleich sollte bei Gleichheit funktionieren.

```
!create house1 : Warehouse
!set house1.name := 'Lagerhaus 1'
!set house1.capacity := 200
!create beer_house1 : Item_Warehouse between (beer, house1)
!set beer_house1.quantity := 200
!set beer_house1.minimumQuantity := 100
```

In diesen Zusammenhang machen auch negative Zahlen keinen Sinn. Daher wurde ein Zustand mit negativen Zahlen erstellt.

```
!set beer_house1.quantity := -200
!set beer_house1.minimumQuantity := -100
```

Auch würden Null-Einträge keinen Sinn machen.

```
!set house1.capacity := 0
```

Was wäre ein Lagerhaus ohne Lagerleiter. Damit dies nicht passiert erstellten wir dazu einen weiteren Zustand, aus dem wir den Lagerleiter entfernten.

```
!create dieter_house1 : Employee_Warehouse between (dieter, house1)
!set dieter_house1.position := 'Lagerarbeiter'
```

Auch sollten die Lagerarbeiter und das Lager im selben Land sein, ansonsten wäre das Pendeln doch zu zeitaufwendig.

```
!create amsterdam : City
!set amsterdam.zip := '8575'
!set amsterdam.name := 'Amsterdam'
!create netherlands : Country
!set netherlands.name := 'Netherlands'
!insert (amsterdam, netherlands) into locatedInCountry
!insert (adressDieter, amsterdam) into locatedInCity
!insert (adressHouse1, bremen) into locatedInCity
!insert (bremen, germany) into locatedInCountry
```

Anschließend wurde geprüft ob sich das Geburtsdatum vom Bestelldatum unterscheidet, da ein Datum immer nur einem Objekt zugeordnet sein darf.

```
!set dateMax.day := 1
!set dateMax.month := 1
!set dateMax.year := 1990
!insert (orderMaxBeer, dateMax) into orderedAt
```

Auch sollte das Geburtsdatum nicht gleich des Lieferdatums sein, da ein Datum immer nur einem Objekt zugeordnet sein darf.

```
!set dateMax.day := 1
!set dateMax.month := 1
!set dateMax.year := 1990
!create deliveryDate : Date
!set deliveryDate.day := 1
!set deliveryDate.month := 1
!set deliveryDate.year := 1990
```

Unser Modell sollte nur funktionieren, wenn eine Adresse genau eine Zuordungen hat.

```
!create adressMax1 : Adress
!set adressMax1.street := 'Musterstrasse 1'
!set adressMax1.description := 'Zuhause'

!create house1 : Warehouse
!set house1.name := 'Lagerhaus 1'
!set house1.capacity := 1000

!insert (house1, adressMax1) into warehouseLocatedAt
```

Unsere Kunden und Mitarbeiter sollten auch telefonisch erreichbar sein. Damit dies gelingt sollten die Telefonnummern stimmen. Damit keine ungültige Eingabe gemacht wird, wurde dies überprüft.

```
!set max.phonenumber := '0421 ABC'
!set dieter.phonenumber := '0421 987bdhd321'
```

Auch die Mail-Adressen sollten stimmen.

```
!set max.email := 'maxmuster.de'
```

Zudem sollte es nicht zwei identische geben.

```
!set max.email := 'max@muster.de'
!set lisa.email := 'max@muster.de'
```

Es ist auch immer sehr ärgerlich, wenn die Bestellung nicht beim Kunden ankommt. Dazu muss die Adresse einer Bestellung eine der Adressen von der Person sein, die die Bestellung angelegt hat.

```
!create adressMax1 : Adress
!set adressMax1.street := 'Musterstrasse 1'
!set adressMax1.description := 'Zuhause'
!create wrongAdress : Adress
```

```
!set wrongAdress.street := 'Falscher Weg 1'
!set wrongAdress.description := 'Falsche Adresse'
```

Auch sollte es dafür nicht zwei unterschiedliche Städte mit gleicher Postleitzahl geben.

```
!create bremen : City
!set bremen.zip := '28359'
!set bremen.name := 'Bremen'
!create hamburg : City
!set hamburg.zip := '28359'
!set hamburg.name := 'Hamburg'
```

Wichtig ist zudem, dass das Geld auf dem richtigen Konto landet. Dazu darf es nicht zwei Institutes mit gleichem BIC geben.

```
!create musterBank : Institute
!set musterBank.bic := 'ABCDEF987'
!set musterBank.name := 'Muster Bank'
!create dieterBank : Institute
!set dieterBank.bic := 'ABCDEF987'
!set dieterBank.name := 'Dieter Bank'
```

Es sollten auch keine zwei Banken mit gleicher IBAN existieren dürfen.

```
!create creditMax : CreditAccount
!set creditMax.iban := 'DE1234567890'
!create creditDieter : CreditAccount
!set creditDieter.iban := 'DE1234567890'
```

5 Realisierung als relationales Datenbankschema

Autor: Jannis Fink
Konzept: Corinna Koch, Robin Hinz, Jannis Fink

In diesem Kapitel beschreiben wir, wie wir vorgegangen sind, um mithilfe des bereits entworfenen UML und ER Modells der Daten eine Datenbank zu erstellen.

5.1 Designentscheidungen

Bei der Realisierung als relationales Datenbankschema haben wir uns am ER Modell (→ 2.2 ER-Diagramm) orientiert. In diesem Modell sind die meisten Attribute der Klassen schon aufgeführt. Lediglich die Fremdschlüssel sowie die Vererbungsbeziehungen müssen noch ergänzt werden. Außerdem müssen die nach SQL übertragbaren Integritätsbedingungen hinzugefügt werden.

5.1.1 Namenskonventionen

Bei der Schreibweise haben wir uns dafür entschieden, statt der im ER Modell verwendeten *CamelCase* Schreibweise der Attribute und Entitäten alle Namen klein zu schreiben und die Wortbestandteile durch Unterstriche zu trennen. So wird z.B. aus `CreditAccount` im ER Modell `credit_account` in der Datenbank.

Um das Arbeiten mit der Datenbank auch in Zukunft so angenehm wie möglich zu gestalten, haben wir uns dafür entschieden, dass alle Tabellen einen einheitlichen Primärschlüssel haben. Dieser Schlüssel heißt in allen Tabellen `id`. Möchte man nun in der Tabelle `person` auf einen Eintrag in der Tabelle `date` verweisen, so geschieht das über eine Spalte `date_id`, die entsprechend als Fremdschlüssel definiert ist. Eine Ausnahme von dieser Konvention stellen Vererbungshierarchien dar, bei denen in den Subklassen der auf das Elternobjekt verweisende Fremdschlüssel gleichzeitig Primärschlüssel ist.

Bei der Benennung der Fremdschlüssel kann vom Schema `tabellenname_id` abgewichen werden. Auf das Geburtsdatum könnte zum Beispiel auch mit dem Spaltennamen `birth_date_id` verwiesen werden. Die Zieltabelle sollte aber trotzdem immer klar erkennbar sein.

5.1.2 Normalformen

Unsere Datenbank ist nach der dritten Normalform normalisiert. Dadurch wollen wir dafür sorgen, dass Redundanzen im System vermieden werden. Um nach der dritten Normalform normalisiert zu sein, muss eine Datenbank auch nach der ersten und zweiten Form normalisiert sein. Um der ersten Normalform zu entsprechen, müssen alle Attribute einen atomaren Wertebereich haben, so dürfen zum Beispiel Vor- und Nachname der in der Datenbank gespeicherten Personen nicht zusammen in einem Feld stehen.

Nach der zweiten Normalform müssen alle Nichtschlüsselattribute immer von allen Primärschlüsseln der Tabelle abhängen. Dadurch soll erreicht werden, dass unterschiedliche Sachverhalte auch in unterschiedlichen Tabellen organisiert werden.

Wenn eine Datenbank sich auch in der dritten Normalform befindet, dann muss sie zusätzlich zu beiden vorherigen Bedingungen zusätzlich die Bedingung erfüllen, dass kein Nichtschlüsselattribut von einem Schlüsselattribut transitiv abhängt. Diese Bedingung soll ebenfalls dafür sorgen, Redundanzen in der Datenbank zu minimieren. Man könnte zum Beispiel die Tabellen `item` und `warehouse` so zusammenfassen, dass in der Tabelle `item` zusätzlich die Spalten `warehouse_name` und `warehouse_capacity` definiert sind. Da ein Lagerhaus mit dem gleichen Namen auch immer die gleiche Kapazität aufweist (da es sich um das gleiche Lagerhaus handelt), würde die Spalte `warehouse_name` von der Spalte `warehouse_capacity` abhängen (und umgekehrt). Ändert man bei zwei Artikeln aus dem gleichen Lagerhaus nur die Kapazität bei einem Artikel, hat man einen inkonsistenten Zustand in der Datenbank. In diesem Beispiel ist die dritte Normalform verletzt.

5.2 Integritätsbedingungen

Autor: Corinna Koch
Konzept: Corinna Koch, Jannis Fink

Einige der Integritätsbedingungen aus Kapitel → 3 Integritätsbedingungen können auch nach SQL übersetzt werden.

Diese werden immer, wenn man Daten in die Tabelle einfügt oder ändert, überprüft, sodass keine inkonsistenten Zustände entstehen können.

Zunächst haben wir die Invarianten `attrsNotNull` hinzugefügt, dies geschieht einfach mit den Keywords `NOT NULL`. Außerdem haben wir vier Attribute, die immer einzigartig sein müssen. Dies sind die Postleitzahl in `city`, die IBAN in `credit_account`, die BIC in `institute` und die Mailadresse in `customer`. Um sicherzustellen, dass diese Attribute eindeutig sind, kann das Keyword `UNIQUE` genutzt werden. Das Attribut `iban` wird jetzt also wie folgend definiert:

```
1     iban TEXT NOT NULL UNIQUE,
```

Für die Invariante `stringsNotEmpty` wurde ein gleichnamiger Constraint für die jeweiligen Attribute angelegt. Dieses sieht für die Tabelle `city` wie folgend aus:

```
    CONSTRAINT stringsNotEmpty CHECK (zip <> '' AND name <> '')
```

Außerdem haben wir für alle Zahlenwerte, die nicht unter 0 fallen dürfen, die Constraint `intGreaterZero` hinzugefügt (analog zu der entsprechenden Invariante in OCL und RA). Für die Tabelle `item` haben wir zum Beispiel folgenden Constraint eingefügt:

```
    CONSTRAINT intGreaterZero CHECK (price > 0 AND weight > 0)
```

Bei der Tabelle `item_warehouse` haben wir zusätzlich zu `intGreaterZero` auch `enoughItemsInStock` aus OCL und RA übernommen:

```
    CONSTRAINT intGreaterZero CHECK (minimum_quantity > 0),
    CONSTRAINT enoughItemsInStock CHECK (quantity > minimum_quantity)
```

Die anderen Integritätsbedingungen müssen auf Anwendungsebene getestet werden, da ein CHECK CONSTRAINT nur auf die aktuell hinzugefügte Zeile zugreifen kann.

5.3 Date/Adress

Im ersten Schritt haben wir die Klassen erstellt, die im weiteren Entwurf der Datenbank von vielen Tabellen gemeinsam genutzt werden würden. Dabei gibt es zwar in vielen RDBMS den Datentyp `datetime`, `date` oder `time`, da dieser Datentyp allerdings nicht in UML bzw. ER vertreten ist, haben wir für Datumswerte eine eigene Relation erstellt:

```
CREATE TABLE "date" (
    id INT,
    day INT NOT NULL,
    month INT NOT NULL,
    year INT NOT NULL,
    PRIMARY KEY (id)
);
```

Um transitive Abhängigkeiten nach der dritten Normalform zu vermeiden, wurden für die Adresse drei Tabellen erstellt. In der Tabelle `country` werden die Länder gespeichert, in der Tabelle `city` die Städte. Die Einträge dieser Tabelle verweisen mittels Fremdschlüssel auf das Land, in dem sie sich befinden. Die Tabelle `adress` wiederum speichert neben der genauen Adresse noch die Stadt, in der sie sich befindet, mittels Fremdschlüssel.

```
CREATE TABLE country (
    id INT,
    name TEXT NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT stringsNotEmpty CHECK (name <> '')
);

CREATE TABLE city (
```

```
    id INT,
    country_id INT,
    zip TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (country_id) REFERENCES country(id),
    CONSTRAINT stringsNotEmpty CHECK (zip <> '' AND name <> '')
);

CREATE TABLE adress (
    id INT,
    city_id INT,
    street TEXT NOT NULL,
    description TEXT NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (city_id) REFERENCES city(id),
    CONSTRAINT stringsNotEmpty CHECK (street <> '')
);
```

Beide Tabellen speichern einfache Attribute und haben keine Fremdschlüssel zu anderen Tabellen. Da es sich bei `date` um ein reserviertes Schlüsselwort in PostgreSQL (und den meisten anderen RDBMS) handelt, wurde es mit Anführungszeichen versehen, damit es zu keinem Syntaxfehler kommt.

5.4 Person

Die Tabelle `Person` verfügt über die zwei Unterklassen `Customer` und `Employee`, die einen Kunden bzw. Angestellten modellieren. Durch diese Art der Modellierung kann ein Kunde nicht gleichzeitig auch ein Angestellter sein, was in unserem Modell durchaus gewünscht ist.

Realisiert wird diese Vererbungshierarchie auf Datenbankebene durch die drei Tabellen `person`, `customer` und `employee`. Die Tabelle für Personen sieht dabei folgendermaßen aus:

```
CREATE TABLE person (
    id INT,
    was_born_date_id INT,
    prename TEXT NOT NULL,
    surname TEXT NOT NULL,
    phonenumber TEXT NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (was_born_date_id) REFERENCES "date"(id),
    CONSTRAINT stringsNotEmpty CHECK (prename <> '' AND surname <> '' AND
   phonenumber <> '')
);
```

Die Tabellen der Angestellten und Kunden sehen so aus:

```
CREATE TABLE customer (
    person_id INT,
```

```
    email TEXT NOT NULL UNIQUE,
    password TEXT NOT NULL,
    PRIMARY KEY (person_id),
    FOREIGN KEY (person_id) REFERENCES person(id)
    FOREIGN KEY (person_id) REFERENCES person(id),
    CONSTRAINT stringsNotEmpty CHECK (email <> '' AND password <> '')
);

CREATE TABLE employee (
    person_id INT,
    salary FLOAT NOT NULL,
    PRIMARY KEY (person_id),
    FOREIGN KEY (person_id) REFERENCES person(id),
    CONSTRAINT intGreaterZero CHECK (salary > 0)
);
```

Beide Tabellen speichern in der Spalte `person_id` den Primärschlüssel der Person ab, zu der sie gehören. Dieser Fremdschlüssel bildet in beiden Tabellen gleichzeitig den Primärschlüssel, da es nur einen Angestellten bzw. Kunden zu einer bestimmten Person geben kann.

5.4.1 Adresse

Personen besitzen mindestens eine, maximal jedoch beliebig viele Adressen. Die Adressen der Person werden in folgender Verknüpfungstabelle gespeichert:

```
CREATE TABLE person_adress (
    person_id INT,
    adress_id INT,
    PRIMARY KEY (person_id, adress_id),
    FOREIGN KEY (person_id) REFERENCES person(id),
    FOREIGN KEY (adress_id) REFERENCES adress(id)
);
```

5.5 CreditAccount

Einer Person sind einer oder mehrere Bankkonten zugeordnet. Ein Bankkonto besteht dabei nur aus der IBAN. Die BIC sowie der Name der Bank werden in einer separaten Tabelle gespeichert. Auf diese wird dann mittels Fremdschlüssel verwiesen.

```
CREATE TABLE institute (
    id INT,
    bic TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL,
    PRIMARY KEY (id)
    PRIMARY KEY (id),
    CONSTRAINT stringsNotEmpty CHECK (bic <> '' AND name <> '')
```

```
);

CREATE TABLE credit_account (
    id INT,
    iban TEXT NOT NULL UNIQUE,
    PRIMARY KEY (id),
    FOREIGN KEY (institute_id) REFERENCES institute(id)
    FOREIGN KEY (institute_id) REFERENCES institute(id),
    CONSTRAINT stringsNotEmpty CHECK (iban <> '')
);
```

Damit ein, bzw. mehrere Bankkonten den Personen zugeordnet werden können, wird eine Verknüpfungstabelle benötigt. In dieser Tabelle wird in zwei Spalten die Kombination von Person und Bankkonto gespeichert, die zusammen gehören. Zwar dürfen nach unserem Datenmodell Mitarbeiter nur eines, Kunden aber mehrere Konten besitzen, dies haben wir aber in unserer Datenbank nicht modelliert. Eine entsprechende Überprüfung muss das vorgeschaltete Verwaltungsprogramm übernehmen. Die Tabelle ist wie folgt aufgebaut:

```
CREATE TABLE person_credit_account (
    person_id INT,
    credit_account_id INT,
    PRIMARY KEY (person_id, credit_account_id),
    FOREIGN KEY (person_id) REFERENCES person(id),
    FOREIGN KEY (credit_account_id) REFERENCES credit_account(id)
);
```

5.6 Item/Warehouse

Die Klassen `Item` bzw. `Warehouse` speichern die Artikel des Onlineshops bzw die Warenhäuser. Sie sind einfache Datentypen, die einige wenige Metadaten speichern:

```
CREATE TABLE item (
    id INT,
    name TEXT NOT NULL,
    price decimal NOT NULL,
    weight decimal NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT stringsNotEmpty CHECK (name <> ''),
    CONSTRAINT intGreaterZero CHECK (price > 0 AND weight > 0)
);
```

```
CREATE TABLE warehouse (
    id INT,
    name TEXT NOT NULL,
    capacity INT NOT NULL,
    adress_id INT NOT NULL,
    PRIMARY KEY (id),
```

```
    FOREIGN KEY (adress_id) REFERENCES adress(id),
    CONSTRAINT stringsNotEmpty CHECK (name <> ''),
    CONSTRAINT intGreaterZero CHECK (capacity > 0)
);
```

Über einen Artikel wird neben seinem Namen nur sein Preis und Gewicht gespeichert.

Das Warenhaus speichert neben seinem Namen nur einen Fremdschlüssel, der auf die Adresse verweist, direkt.

5.6.1 ItemWarehouse

In dieser Assoziationsklasse wird gespeichert, welcher Artikel in welchem Lagerhaus vorrätig ist. Neben dieser Information wird zusätzlich noch gespeichert, wie viele Artikel aktuell vorhanden sind und wie viele Artikel mindestens vorhanden sein sollten:

```
CREATE TABLE item_warehouse (
    item_id INT,
    warehouse_id INT,
    quantity INT NOT NULL,
    minimum_quantity INT NOT NULL,
    PRIMARY KEY (item_id, warehouse_id),
    FOREIGN KEY (item_id) REFERENCES item(id),
    FOREIGN KEY (warehouse_id) REFERENCES warehouse(id),
    CONSTRAINT intGreaterZero CHECK (minimum_quantity > 0),
    CONSTRAINT enoughItemsInStock CHECK (quantity > minimum_quantity)
);
```

5.6.2 EmployeeWarehouse

Eine weitere Assoziationsklasse wird benötigt, in der die Mitarbeiter der einzelnen Lager gespeichert werden. In dieser Klasse wird neben dem Lager und dem Mitarbeiter auch die Position jedes einzelnen Mitarbeiters gespeichert:

```
CREATE TABLE employee_warehouse (
    employee_id INT,
    warehouse_id INT,
    position TEXT NOT NULL,
    PRIMARY KEY (employee_id, warehouse_id),
    FOREIGN KEY (employee_id) REFERENCES employee(person_id),
    FOREIGN KEY (warehouse_id) REFERENCES warehouse(id)
);
```

5.7 Provider

Um Auskunft darüber erhalten zu können, bei welchem Lieferanten die einzelnen Artikel im Lager abgespeichert werden können, sollen auch diese Informationen im System verwaltet werden. Dazu wird zunächst eine Klasse `Provider` angelegt:

```
CREATE TABLE provider (
    id INT,
    adress_id INT,
    name TEXT NOT NULL,
    phone_number TEXT NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (adress_id) REFERENCES adress(id),
    CONSTRAINT stringsNotEmpty CHECK (name <> '' AND phonenumber <> '')
);
```

In dieser Klasse werden der Name, die Telefonnummer und die Adresse jedes einzelnen Lieferanten gespeichert.

Eine wichtige Information ist die, welcher Lieferant in der Lage ist, welche Artikel zu liefern. Dazu wird eine Verknüpfungsklasse angelegt. In dieser werden die einzelnen Lieferanten mit den Artikeln verknüpft, die sie liefern können.

```
CREATE TABLE item_provider (
    item_id INT,
    provider_id INT,
    PRIMARY KEY (item_id, provider_id),
    FOREIGN KEY (item_id) REFERENCES item(id),
    FOREIGN KEY (provider_id) REFERENCES provider(id),
    CONSTRAINT stringsNotEmpty CHECK (position <> '')
);
```

Um jetzt auch noch erfahren zu können, welcher Lieferant welches Lagerhaus beliefern kann, wird eine weitere Verknüpfungsklasse benötigt. Dieses Mal werden die Lieferanten mit den Lagerhäusern verknüpft, die sie beliefern können.

```
CREATE TABLE provider_warehouse (
    provider_id INT,
    warehouse_id INT,
    PRIMARY KEY (provider_id, warehouse_id),
    FOREIGN KEY (provider_id) REFERENCES provider(id),
    FOREIGN KEY (warehouse_id) REFERENCES warehouse(id)
);
```

5.8 Order

Der wichtigste Teil des Onlineshops sind die Bestellungen der Kunden. Diese werden in der Tabelle `order` verwaltet. Eine Bestellung hat dabei eine eindeutige Nummer und verweist auf die verschie-

denen Bestandteile der Bestellung. Neben dieser eindeutigen Nummer wird der Kunde, das Bestelldatum, der verantwortliche Mitarbeiter und die Lieferung (dazu später mehr) direkt mittels Fremdschlüssel am Objekt gespeichert:

```
CREATE TABLE "order" (
    id INT,
    date_id INT,
    customer_id INT,
    employee_id INT,
    adress_id INT,
    delivery_id INT,
    PRIMARY KEY (id),
    FOREIGN KEY (date_id) REFERENCES "date"(id),
    FOREIGN KEY (customer_id) REFERENCES customer(person_id),
    FOREIGN KEY (employee_id) REFERENCES employee(person_id),
    FOREIGN KEY (adress_id) REFERENCES adress(id),
    FOREIGN KEY (delivery_id) REFERENCES delivery(id)
);
```

Die Artikel, die zur Bestellung gehören, werden mit einer Assoziationsklasse mit der Bestellung verknüpft. Die Assoziationsklasse speichert auch die gewünschte Menge der bestellten Artikel:

```
CREATE TABLE order_item (
    order_id INT,
    item_id INT,
    quantity INT NOT NULL,
    PRIMARY KEY (order_id, item_id),
    FOREIGN KEY (order_id) REFERENCES "order"(id),
    FOREIGN KEY (item_id) REFERENCES item(id),
    CONSTRAINT intGreaterZero CHECK (quantity > 0)
);
```

5.9 Delivery

Den letzten Teil des Systems bilden die Versandinformationen. Zu jeder Bestellung gibt es einen Eintrag in der Tabelle `delivery`, die Verweise auf den Lieferdienst und die Lieferart (Standard, Premium, ...) speichert. Das Versanddatum (`delivery_date_id`) der Tabelle `delivery` ist initial nicht gefüllt.

```
CREATE TABLE delivery (
    id INT,
    delivery_service_id INT,
    delivery_type_id INT,
    delivery_date_id INT,
    PRIMARY KEY (id),
    FOREIGN KEY (delivery_service_id) REFERENCES delivery_service(id),
    FOREIGN KEY (delivery_type_id) REFERENCES delivery_type(id),
    FOREIGN KEY (delivery_date_id) REFERENCES "date"(id),
```

```
    CONSTRAINT stringsNotEmpty CHECK (name <> '')
);
```

```
CREATE TABLE delivery_service (
    id INT,
    name TEXT NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT stringsNotEmpty CHECK (name <> '')
);
```

```
CREATE TABLE delivery_type (
    id INT,
    name TEXT NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT stringsNotEmpty CHECK (name <> '')
);
```

6 Sichten

Autor: Jannis Fink
Konzept: Jannis Fink

In diesem Kapitel wird beschrieben, welche Views wir erstellt haben, um das Arbeiten mit den im letzten Kapitel erstellten Tabellen zu vereinfachen.

6.1 Namenskonventionen

Wie auch bei den Tabellen und Attributen der Datenbank befolgen wir auch bei der Benennung der Views und ihrer Spalten bestimmmten Namenskonventionen. Der Name einer View wird ebenfalls in der Form `kleinbuchstaben_mit_unterstrichen` formuliert und sollte die enthaltenen Daten möglichst gut beschreiben. Um später immer erkennen zu können, dass bei einer Abfrage ein View und nicht etwa eine Tabelle benutzt wurde, soll der Name immer auf `_v` enden.

6.2 Personen

Das Arbeiten mit Kunden und Angestellten ist durch den benötigten `INNER JOIN` bei jeder Abfrage nicht wirklich übersichtlich. Deswegen wird ein View erstellt, der die Attribute der Person mit dem dazu gehörigen Angestellten bzw. Kunden zusammenführt. Diese Views werden `customer_v` für den Kunden bzw. `employee_v` für den Mitarbeiter genannt und sehen folgendermaßen aus:

```
1 CREATE OR REPLACE VIEW customer_v AS
2 SELECT * FROM person INNER JOIN customer ON person.id = customer.person_id;
3
4 CREATE OR REPLACE VIEW employee_v AS
5 SELECT * FROM person INNER JOIN employee ON person.id = employee.person_id;
```

6.3 Bestellungen

6.3.1 Offene Bestellungen

Das Arbeiten mit Bestellungen soll ebenfalls vereinfacht werden. Im normalen Arbeitsalltag sind für die Mitarbeiter nur die offenen Bestellungen interessant. Aus diesem Grund soll eine View erstellt wer-

den, die nur die offenen Bestellungen zurück gibt. Sie soll `open_order_v` heißen.

```
CREATE OR REPLACE VIEW open_order_v AS
SELECT * FROM "order" WHERE delivery_id IS NULL;
```

6.3.2 Bestellungen pro Kunde

Für spätere Analysen kann es wichtig sein zu wissen, welcher Kunde wie viele Bestellungen tätigt. Zu diesem Zweck wird ein View erstellt, welches die Bestellungen der Kunden zählt und diese nach Jahr gruppiert zurück gibt. Um das Ergebnisset möglichst klein zu halten, wir für den Kunden nur seine ID ausgegeben. Möchte man wissen, welcher Kunde zu einer bestimmten ID gehört, ist eine weitere Abfrage an die Datenbank nötig.

```
CREATE OR REPLACE VIEW orders_by_customer_and_year_v AS
SELECT
    c.person_id AS customer_id,
    d.year,
    count(o.id) AS number_deliveries
FROM "order" o INNER JOIN customer_v c ON o.customer_id = c.person_id INNER
    JOIN date d on o.date_id = d.id
GROUP BY d.year, c.person_id;
```

Dieses View gibt drei Spalten zurück. In der ersten Spalte steht die Kundennummer, die durch das AS `customer_id` nach `customer_id` umbenannt wird. In der Spalte danach wird das Jahr, das betrachtet wird, zurückgegeben. Durch das GROUP BY Statement am Ende des Views werden die Ergebnisse nach der Kundennummer und dem Jahr der Bestellung gruppiert. `count(o.id)` zählt dann die Anzahl der Bestellungen in den einzelnen Gruppen.

6.4 Artikel

6.4.1 Artikel, die nachbestellt werden müssen

Eine wichtige Information im täglichen Betrieb ist die, welche Artikel in welchem Lager nachbestellt werden müssen. Hierfür wird ebenfalls ein View erstellt, der diese Information übersichtlich darstellt. Um das zu erreichen werden die Tabellen `warehouse`, `item` und `item_warehouse` mittels `INNER JOIN` mithilfe ihrer Primärschlüssel zusammengefügt. Dieses Ergebnis wird dann mittels `WHERE` auf die Zeilen eingeschränkt, in denen die Minimalmenge unterhalb des aktuellen Lagerbestandes liegt. Von den verbleibenden Zeilen werden dann die wichtigen Informationen zurückgegeben. Dazu gehören die ID des Artikels und des Lagerhauses, deren Namen und eine Auskunft über die Minimalmenge, den aktuellen Lagerbestand sowie die Kapazitiät des Lagerhauses.

```
CREATE OR REPLACE VIEW items_to_be_reordered_v AS
SELECT
    w.id AS warehouse_id,
    w.name AS warehouse_name,
```

```
    i.id AS item_id,
    i.name AS item_name,
    iw.quantity,
    iw.minimum_quantity,
    w.capacity AS warehouse_capacity
FROM item_warehouse iw INNER JOIN warehouse w ON iw.warehouse_id = w.id
   INNER JOIN item i ON iw.item_id = i.id
WHERE iw.quantity < iw.minimum_quantity
```

6.5 Geschäftszahlen

Für die Geschäftsleitung ist es auch interessant zu wissen, wie gut das Geschäft läuft. Zu diesem Zweck soll das System eine nach Jahren sortierte Auflistung der Umsätze liefern.

```
CREATE OR REPLACE VIEW sales_by_year_v AS
SELECT d.year, sum(i.price * oi.quantity) AS sales
FROM order_item oi INNER JOIN item i ON oi.item_id = i.id INNER JOIN "order
   " o ON o.id = oi.order_id INNER JOIN date d on d.id = o.date_id
GROUP BY d.year
```

Das größte Problem bei diesem View stellt die Tatsache dar, dass die benötigten Informationen über viele Tabellen verteilt sind. In der Tabelle `order_item` ist die verkaufte Menge, in der Tabelle `item` der Preis und in der Tabelle `order` der Zeitpunkt des Verkaufs gespeichert. Ein weiterer `INNER JOIN` ist nötig, da wir das Datum nicht im dafür vorgesehen Spaltentyp, sondern in einer Extra-Tabelle speichern.

Die View gibt anschließend das Jahr in einer und die Gesamtverkäufe über alle Artikel aufsummiert in einer zweiten Spalte aus. Hierzu wird mittels GROUP BY nach dem Jahr gruppiert und dann mit der Funktion `sum` die einzelnen Umsätze addiert. Der einzelne Umsatz errechnet sich dabei aus dem Produkt von der verkauften Menge und dem Einzelpreis.

In unserer Berechnung lassen wir die Tatsache außen vor, dass sich der Preis eines Artikels mit der Zeit sehr wahrscheinlich ändert. Eine Preisänderung hätte auch verfälschte Geschäftszahlen zur Folge.

7 Standardabfragen

Autor: Jannis Fink(SQL, Beschreibung), Corinna Koch(RA)
Konzept: Corinna Koch, Robin Hinz, Jannis Fink

In diesem Kapitel beschreiben wir einige konkrete Probleme und wie wir diese mit dem Datenbanksystem lösen.

Die Abfragen haben wir in SQL und RA formuliert.

7.1 Artikel

7.1.1 Artikelsuche

Möchten wir wissen, in welchen Lagerhäusern ein bestimmter Artikel zu einer gewissen Menge vorrätig ist, im Beispiel der Artikel mit dem Namen `beer` und der Menge `10`, ginge das so:

```
SELECT * FROM warehouse WHERE id IN (
    SELECT warehouse_id FROM item_warehouse WHERE item_id IN (
        SELECT id FROM item WHERE name='beer'
    ) AND quantity >= 10
)
```

Wie auch bei den erstellten Views schon häufiger beobachtet wurde, liegen die benötigten Informationen in verschiedenen Tabellen. Ein Subselect ist nötig, um vom Namen des Artikels auf dessen ID zu kommen. Um dann in der Verknüpfungstabelle alle Warenhäuser bzw. die IDs derer zu finden, die den Artikel in der gewünschten Menge vorrätig haben, ist ein weiterer Subselect möglich. Jetzt haben wir alle IDs der Warenhäuser, in denen der Artikel vorrrätig ist, und können diese alle ausgeben.

In RA selektieren wir zunächst die `id` aller Artikel mit dem Namen `beer`.

$$A := \pi_{id}(\sigma_{name='beer'}(ITEM))$$

In der Tabelle `item_warehouse` selektieren wir die `item_id` und `warhouse_id` aller Vorkommnisse, die mindestens die Menge `10` haben.

$$B := \pi_{item\_id,warehouse\_id}(\sigma_{quantity \geq 10}(ITEM\_WAREHOUSE))$$

Nun selektieren wir mit Hilfe des kartesischen Produktes die IDs der Lagerhäuser, in denen `beer` mindestens die Menge `10` hat.

$$C := \pi_{B.warehouse\_id}(\sigma_{A.id=B.item\_id}(A \times B))$$

Nun müssen wir nur noch die Lagerhäuser selbst selektieren. Dabei wählen wir mit der Projektion genau die Attribute von `Warehouse` aus, die in SQL mit `SELECT *` gewählt wird (ohne diese Projektion hätte die Ergebnisrelation auch das Attribut `warehouse_id`).

$$\pi_{id,name,capacity,adress\_id}(\sigma_{Warehouse.id=C.warehouse\_id}(WAREHOUSE \times C))$$

7.2 Kunde

7.2.1 Bestellte Artikel

Um zu erfragen, welche eindeutigen Artikel schon von einem bestimmten Kunden bestellt wurden, kann man folgendermaßen vorgehen:

```
SELECT DISTINCT * FROM item WHERE id IN (
    SELECT item_id FROM order_item WHERE order_id IN (
        SELECT id FROM "order" WHERE customer_id = 1
    )
)
```

In dieser Anfrage werden durch das Schlüsselwort `DISTINCT` gefundene, eindeutige Artikel, die mehrfach bestellt wurden nur einmal zurückgegeben. Um die bestellten Artikel zu ermitteln, werden mehrere Subselects benötigt. Einer ermittelt alle Bestellungen, die ein bestimmter Kunde getätigt hat. In unserem Beispiel ist das der Kunde mit der ID `1`. Für diese Bestellungen werden dann mithilfe der Tabelle `order_item` die IDs der Artikel ermittelt, die dieser Kunde bestellt hat.

In RA selektieren wir zunächst die IDs der Bestellungen, deren Kunde die ID `1` hat.

$$A := \pi_{id}(\sigma_{customer\_id=1}(ORDER))$$

Als nächstes selektieren wir die IDs der Artikel, welche in den Bestellungen geordert wurden.

$$B := \pi_{item\_id}(\sigma_{Order\_Item.order\_id=A.id}(ORDER\_ITEM \times A))$$

Zum Schluss können die Artikeldaten selektiert werden. Wie schon oben beschrieben nutzen wir die Projektion, um nur die Attribute von `Item` auszuwählen.

$$\pi_{id,name,price,weight}(\sigma_{B.item\_id=Item.id}(B \times ITEM))$$

DISTINCT wird in RA implizit angewandt, da in RA alle Operationen auf Mengen durchgeführt werden.

7.3 Mitarbeiter

7.3.1 Offene Bestellungen

Es könnte ebenfalls interessant sein zu wissen, welcher Mitarbeiter welche Bestellungen bearbeitet hat, die noch nicht versendet wurden. Diese Information könnte genutzt werden, um eine TODO Liste für die Mitarbeiter zu erstellen.

```
1 SELECT
2     o.id AS order_id,
3     e.person_id AS employee_id,
4     (e.prename || ' ' || e.surname) AS employee_name,
5     d.year,
6     d.month,
7     d.day
8 FROM open_order_v o INNER JOIN employee_v e ON e.person_id = o.employee_id
     INNER JOIN date d ON d.id = o.date_id
9 WHERE d.year = 2016
```

Mit dieser Anfrage werden die offenen Bestellungen aus dem Jahr 2016 berechnet und den jeweiligen Verantwortlichen zugeordnet. Um die Bestellung und den verantwortlichen Angestellten zu finden, werden zunächst ihre IDs ausgegeben. Anschließend wird zur Anzeige im Programm der Name bestehend aus Vor- und Nachnamen direkt zusammengebaut und as `employee_name` ausgegeben. Im Anschluss wird noch das ursprüngliche Bestelldatum ausgegeben.

Bei dieser Abfrage bedienen wir uns der im letzten Kapitel erstellten Views. So wird `open_order_v` benutzt, um die Bestellungen direkt auf die offenen Bestellungen einzugrenzen. Statt eines `JOIN`, um die Tabellen `person` und `employee` zusammenzuführen und so Zugriff auf den Namen des Angestellten zu haben, wurde der vorher erstellte View `employee_v` benutzt.

Für die Übersetzung nach RA nehmen wir vereinfacht an, dass wir die View `open_order_v` nutzen können. Da diese Werte auf `NULL` überprüft, können wir sie nicht nach RA übersetzen.
Desweiteren verzichten wir in RA auf die Zusammenfassung von `prename` und `surname`.

Die View `employee_v` kann in RA über den natürlichen Verbund definiert werden:

$$EMPLOYEE\_V := PERSON * EMPLOYEE$$

Die Abfrage kann nun nach RA übersetzt werden. Dafür haben wir zunächst mit dem natürlichen Verbund die Tabellen zusammengefasst. Um die Selektion zu vereinfachen haben wir dabei die nötigen Attribute direkt umbenannt.

$$OPEN\_ORDER := (OPEN\_ORDER\_V * (\delta_{date\_id \leftarrow id}(DATE))) \\ * (\delta_{employee\_id \leftarrow person\_id}(EMPLOYEE\_V))$$

Nun können wir die gewünschten Attribute selektieren:

$$\pi_{order\_id,employee\_id,prename,surname,year,month,date}(\sigma_{year=2016}(OPEN\_ORDER))$$

7.4 Lagerhäuser

7.4.1 Vorrätige Artikel

Unser System erlaubt es ebenfalls, die vorrätigen Artikel in einem bestimmten Lagerhaus abzufragen. Um feststellen zu können, welche Artikel hier am ehesten nachbestellt werden müssen, kann hier die Differenz zur Mindestmenge berechnet und aufsteigend nach dieser sortiert werden. So kann der Mitarbeiter schnell sehen, welche Artikel demnächst nachbestellt werden müssen.

```
1 SELECT
2     w.id AS warehouse_id,
3     w.name AS warehouse_name,
4     i.id AS item_id,
5     i.name AS item_name,
6     (iw.quantity - iw.minimum_quantity) AS save_items
7 FROM warehouse w INNER JOIN item_warehouse iw ON iw.warehouse_id = w.id
     INNER JOIN item i ON iw.item_id = i.id
8 WHERE w.id = 1
9 ORDER BY save_items ASC
```

Bei dieser Anfrage werden zunächst die benötigten Tabellen zusammengejoint. Die benötigten Informationen liegen in den Tabellen `warehouse`, `item` und `item_warehouse`. Das Lagerhaus und der Artikel werden mit ihrem jeweiligen Namen und der ID ausgegeben, sodass eine weitere Suche dieser Datenobjekte in der Datenbank leicht möglich ist. Darüber hinaus wird eine Spalte `save_items` berechnet, in der die Differenz zwischen dem aktuellen Lagerbestand und dem Mindestbestand angezeigt wird. Nach dieser Spalte wird dann absteigend sortiert, sodass der Mitarbeiter in der Lage ist, die am dringendsten benötigten Artikel nachzubestellen. Um die Auswahl auf ein bestimmtes Lagerhaus einzuschränken, wird das Ergebnis auf ein bestimmtes Lagerhaus eingeschränkt. In unserem Beispiel ist das das Lagerhaus mit der ID `1`.

In RA kann das Sortieren und die Berechnung der Differenz zur Mindestmenge nicht umgesetzt werden.

Deshalb haben wir sowohl die Anzahl als auch die Mindestmenge selektiert. Die in SQL selektierten Spalten können in RA durch den natürlichen Verbund ausgewählt werden.

$$WAREHOUSE' := \delta_{warehouse\_id \leftarrow id, warehouse\_name \leftarrow name}(\pi_{id,name}(WAREHOUSE))$$
$$ITEM' := \delta_{item\_id \leftarrow id, item\_name \leftarrow name}(\pi_{id,name}(ITEM))$$
$$WAREHOUSE' * (ITEM\_WAREHOUSE * ITEM')$$

Literatur

[Gog16] Martin Gogolla. *Vorlesungsfolien Datenbanksysteme, Kapitel 3: Relationale Anfragesprachen, Seite 4*. 2016. URL: http://www.db.informatik.uni-bremen.de/teaching/courses/ws2016_dbs/kap3.pdf (besucht am 01. 03. 2017).

[Mic] Hochschule Darmstadt Michael Guist. *Klassendiagramm*. URL: https://www.fbi.h-da.de/labore/case/uml/klassendiagramm.html (besucht am 01. 03. 2017).

[unb16a] unbekannt. *Entity-Relationship-Modell*. 2016. URL: https://de.wikipedia.org/wiki/Entity-Relationship-Modell (besucht am 01. 03. 2017).

[unb16b] unbekannt. *Klassendiagramm*. 2016. URL: https://de.wikipedia.org/wiki/Klassendiagramm (besucht am 01. 03. 2017).