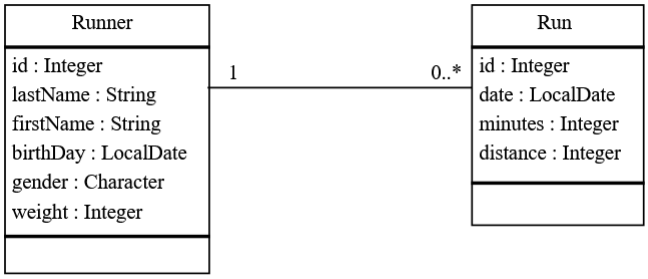
1. Gegeben ist das UML-Diagramm für eine einfache Beziehung zwischen Läufern (`Runner`) und ihren Laufleistungen (`Run`):



Dabei ist zu beachten:

- Beide Primarykeys sind autogeneriert
- Das Geschlecht wird als `W` oder `M` gespeichert.
- Das Gewicht `weight` wird in kg gespeichert.
- Die gelaufene Strecke (`distance`) wird in Meter gespeichert.
- Die Laufzeit (`minutes`) wird in Minuten gespeichert.

Lösen Sie zu diesen Vorgaben die folgenden Aufgaben:

a) Erstellen sie die Entityklassen mit bidirektionalem Mapping der 1 : n - Beziehung zwischen `Runner` und `Run` und erzeugen Sie die Datenbank mit Hilfe eines Schemaexports. Füllen Sie die Datenbank mit geeigneten Testdaten.

b) Schreiben Sie eine DAO-Methode, die alle Läufer liefert, die wenigstens eine Laufleistung haben, die mehr als `m` Meter lang ist. Sollte der übergebene Wert `m` kleiner gleich Null sein wird eine `IllegalArgumentException` geworfen.

c) Schreiben Sie eine DAO-Methode, welche die Gesamtlaufstrecke eines Läufers in einem bestimmten Zeitintervall liefert. Dieses Zeitintervall ist über zwei Parameter vom Typ `java.time.LocalDate` zu übergeben.

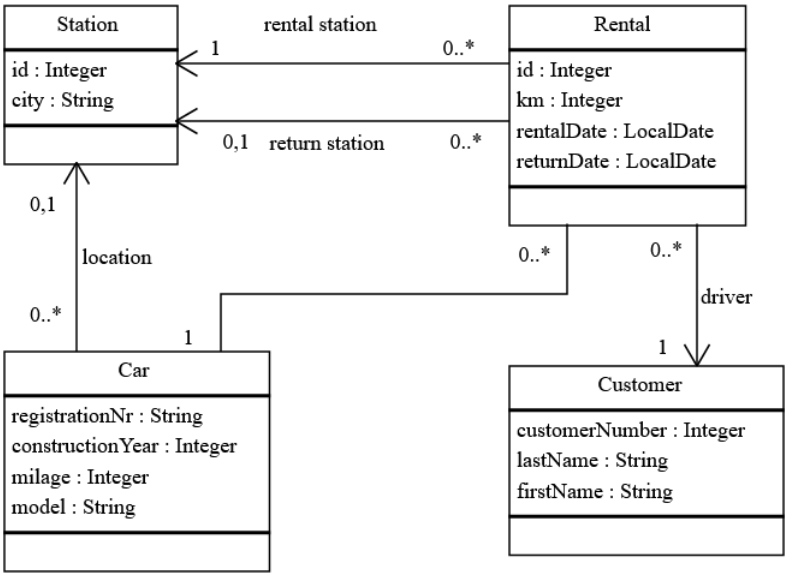
Die Implementierten Methoden sind mit einem JUnit-Test mit kommentierten Testfällen zu testen. Schreiben Sie zu diesem Zweck eine Methode `insertTestData()`, welche die folgenden Datensätze zu Beginn des Tests in die Testdatenbank einfügt:

```
Runner r1 = new Runner("Huber", "Karl", LocalDate.of(1990,6,7), 'M', 75);
Runner r2 = new Runner("Schmidt", "Eva", LocalDate.of(1997,10,26), 'W', 55);

Run l1 = new Run(LocalDate.of(2019,6,11), 55, 10350);
l1.setRunner(r1);
Run l2 = new Run(LocalDate.of(2019,6,12), 172, 42195);
l2.setRunner(r1);
Run l3 = new Run(LocalDate.of(2019,6,13), 58, 8320);
l3.setRunner(r2);
Run l4 = new Run(LocalDate.of(2019,7,14), 83, 15000);
l4.setRunner(r2);
Run l5 = new Run(LocalDate.of(2019,7,15), 115, 21000);
l5.setRunner(r2);
```

Im folgenden sind fünf Testmethoden vorgegeben. Ergänzen Sie weitere kommentierte Testfälle für die Methode `totalDistance()`.

```
// Testfall: testet DAO.longRun(25000)
// Liefert eine Liste mit der Läufer Huber Karl
@Test
void testLongRun1() {
  List<Runner> runner = dao.longRun(25000);
  assertEquals(1, runner.size());
  assertEquals("Huber", runner.get(0).getLastname());
}

// Testfall: testet DAO.longRun(10000)
// Liefert eine Liste mit zwei Läufern
@Test
void testLongRun2() {
  List<Runner> runner = dao.longRun(10000);
  assertEquals(2, runner.size());
}

// Testfall: testet DAO.longRun(50000)
// Liefert eine leere Liste
@Test
void testLongRun3() {
  List<Runner> runner = dao.longRun(50000);
  assertTrue(runner.isEmpty());
}

// Testfall: testet DAO.longRun(-10000)
// Erwartetes Ergebnis: wirft eine IllegalArgumentException
@Test
void testLongRun4() {
  assertThrows(IllegalArgumentException.class,
               () -> dao.longRun(-10000));
}

// Testfall: liefert die Gesamtstrecke des Läufers mit der ID 1 im Juli 2019
// Erwartetes Ergebnis: 36000
@Test
void testTotalDistance1() {
  Runner r = dao.findRunnerById(2);
  System.out.println(r);
  assertEquals(r.getLastname(), "Schmidt");
  int distance = dao.totalDistance(r, LocalDate.of(2019,7,1),
                                      LocalDate.of(2019,7,31));
  assertEquals(36000, distance);
}
```

2. Gegeben ist ein UML Diagramm für das vereinfachte Entitymodell einer Autovermietung:



Dabei sind die folgenden Geschäftsreglen zu beachten:

- In der Entität `Station` (Vermietstation) ist der PK autogeneriert. Der Einfachheit halber wird nur die Stadt, in der die Station steht gespeichert.
- In der Entität `Car` ist der PK das Kennzeichen, darüber hinaus sind Baujahr, Kilometerstand und das Modell (z.B. Suzuki Vitara, VW Passat) als String gespeichert. Ein Fahrzeug steht entweder an einer Station oder ist vermietet. Ein Fahrzeug, das gerade unterwegs ist (also einer noch nicht abgeschlossenen Fahrt zugeordnet ist), hat keinen Standort an einer der Vermietstationen.
- Die Entität `Rental` beschreibt einen Vermietvorgang. Sie hat einen autogenerierten PK, ein Vermietdatum und ein Rückgabedatum. Außerdem werden am Ende der Fahrt die während des Mietvorganges gefahrenen Kilometer gespeichert. Der Einfachheit wird angenommen, dass keine Reservierungen entgegengenommen werden, d.h. Fahrzeuge die an einer Station stehen sind frei und können vermietet werden. Sobald ein Fahrzeug gebucht wurde kann es erst wieder nach Abschluss der Fahrt neu vermietet werden. Ein Mieter kann ein Fahrzeug an einer Vermietstation abholen und bei einer anderen Station wieder abgeben (daher gibt es zwei Beziehungen zwischen Rental und Station). Ein noch nicht abgeschlossener Mietvorgang hat noch keine Station und noch kein Rückgabedatum zugeordnet. Rentals stellen eine Historisierung von Vorgängen dar, daher dürfen die initial getroffenen Zuordnungen zu Abfahrtsort, Mieter und Fahrzeug nicht nachträglich geändert werden.
- Ein Kunde hat eine numerische 6-stellige Kundennummer (111111 - 999999) als PK sowie einen Familiennamen und Vornamen.

Lösen Sie zu diesem Geschäftsmodell die folgenden Aufgaben:

(a) Definieren Sie aus dem UML-Diagramm und den Geschäftsregeln ein angemessenes Entity-Modell, wobei die n : 1 - Beziehungen vorzugsweise unidirektional (also mit Hilfe von `@ManyToOne` - Annotations) gemappt werden sollen. Lediglich ein Fahrzeug soll auch seine Fahrten kennen (diese Beziehung ist also bidirektional zu mappen).

(b) Schreiben Sie eine Datenbankzugriffsklasse, die wenigstens die folgenden Methoden enthalten:

```
// Speichert eine neue Station in der Datenbank
// liefert true bei Erfolg, false bei einem Fehler
public boolean insertStation(Station station)

// Speichert ein neues Fahrzeug in der Datenbank
// liefert true bei Erfolg, false bei einem Fehler
public boolean inserCar(Car car)

// Speichert einen neuen Mieter in der Datenbank
// liefert true bei Erfolg, false bei einem Fehler
public boolean insertCustomer(Customer c)


// Speichert einen neuen Mietvorgang in der Datenbank
// liefert true bei Erfolg, false bei einem Fehler
public boolean insertRental(Rental rental)

// Liefert eine Liste aller Fahrzeuge, die in der Station st
// verfuegbar sind.
public List<Car> findCarsByStation(Station station)

// Liefert eine Liste aller Mietvorgänge, die der Customer c
// getätigt hat.
public List<Fahrt> findRentalsByCustomer(Customer c))

// Schließt den Mietvorgang r mit dem Rueckgabedatum returnDate
// bei der Station returnStation ab, wobei km die gefahrenen Kilometer
// sind. Beachte dass auch die Daten im Mietfahrzeug aktualisiert
// werden müssen.
// liefert true bei Erfolg, false bei einem Fehler
// public boolean returnCar(Rental r, Station returnStation,
//                          LocalDate returnDate, Integer km)
```

(c) Schreiben Sie eine Test, welche Ihre erzeugten Tabellen mit Testdaten initialisiert und die oben beschriebenen Methoden geeignet testet. Dokumentieren Sie ihre Testfälle. Dabei sind wenigsten folgende Testdaten zu verwenden:

- wenigstens 3 Stationen
- wenigstens 10 Fahrzeuge
- wenigstens 3 Mieter
- wenigstens 5 Fahrten, davon 3 abgeschlossene und 2 nicht abgeschlossene

3. Gegeben ist ein SQL-Skript (`kurssystem.sql`), das eine einfache Datenbank für die Verwaltung eines EDV-Kurssystems generiert. Es besteht aus folgenden Tabellen:

   - Tabelle `dozent`
   - Tabelle `kurs`
   - Tabelle `kunde`
   - Tabelle `kurs_kunde`
   - Tabelle `kurstyp`

   Lösen Sie die folgenden Teilaufgaben:

   (a) Erzeugen Sie aus dem SQL-Skript eine PostgerSQL-Datenbank (es kann auch die Datenbank aus Labor 07 verwendet werden) und daraus mit Reverse-Engineering JPA-Entitäten. Ergänzen Sie die Entitäten mit sinnvollen Constraints unter Verwendung des Java-Validation-Frameworks.

   (b) Entwickeln Sie zu jeder Entität eine eigene Datenbankzugriffsklasse (Repository) mit entsprechenden Methoden.

   (c) Entwickeln Sie eine Serviceklasse als Singleton, die das gegebene Interface `IKurssystemService` implementiert. Nur diese Serviceklasse soll auf die Repositories zugreifen. Implementieren Sie die Fehlerklasse `KursDBException` als unchecked Exception.

   (d) Entwickeln Sie zur Serviceklasse eine JUnit5-Testklasse und testen Sie jede Methode sinnvoll. Dokumentieren Sie den Zweck und das erwartete Ergebnis jeder Testmethode.