

# **SQL-Alchemist-Teamprojekt**

## API-Dokumentation

Tobias Grünhagen, Philip Holzhüter, Tobias Runge

8. Juli 2015

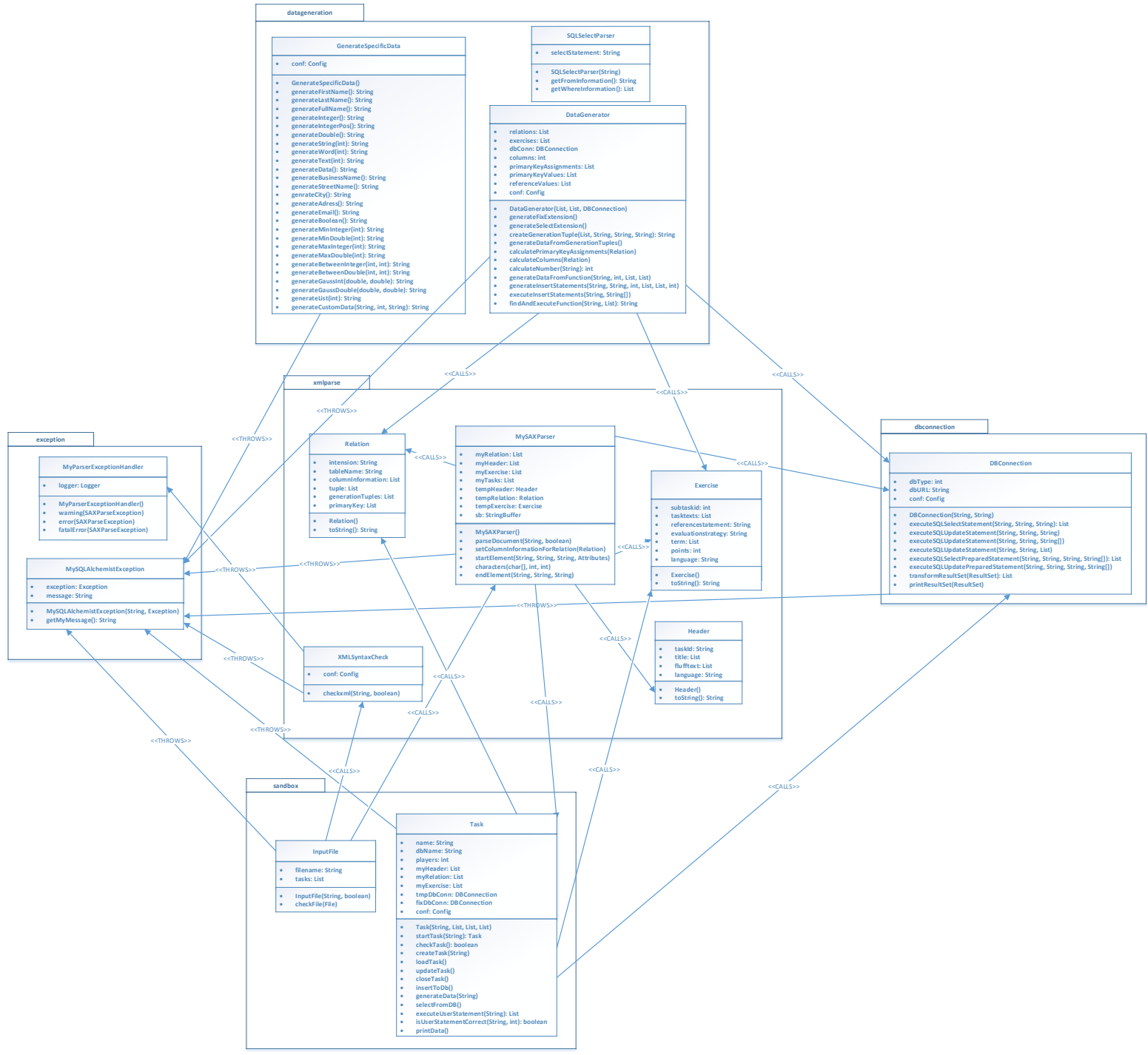
# Kapitel 1

## Allgemeines

Die API für das SQL-Alchemist-Projekt des IFIS an der Technischen Universität Braunschweig im Sommersemester 2015 dient dazu wichtige Grundfunktionen, wie das Verwalten von Aufgaben und der Datenbankanfragen, für das SQL-Lernspiel bereitzustellen.

Zuerst werden dabei eine oder mehrere Aufgaben durch XML-File oder XML-String eingelesen. Hierbei wird die Struktur validiert und bereits der Aufbau der SQL-Statements gecheckt. Beim Parsen der Aufgaben wird dann für jede Aufgabe ein neuer Task angelegt, über den die komplette Aufgabe verwaltet werden kann. Die für die Aufgabe nötigen Tabellen und Einträge werden in einer aufgabenspezifische Datenbank abgespeichert, die wieder gelöscht wird, falls sie nicht mehr benötigt wird. Anfragen an die Datenbank und auch der Zugriff auf einzelne Komponenten der Aufgabenstellung kann komplett über den Task verwaltet werden. Bereits beim Erstellen eines Tasks werden generierte Daten in die zugehörigen Tabellen eingefügt. Unterschieden wird dabei zwischen Daten, die auf festen Benutzervorgaben beruhen und Daten, die auf den Referenzstatements beruhen.

Die Programmstruktur und die Programmmzusammenhänge können in dem folgenden Klassendiagramm nachvollzogen werden. Bei Benutzung der API sollte ein InputFile-Objekt mit dem Namen des XML-Files oder dem XML-String selbst erstellt werden. Hier wird im Konstruktor automatisch die Struktur durch die XMLSyntaxCheck-Klasse validiert und mit Hilfe der MySAXParser-Klasse das Dokument eingelesen. Im MySAX-Parser werden nach dem Parsen mit Hilfe der DBConnection-Klasse die einzelnen SQL-Statements gecheckt. Dann wird für jede Aufgabe ein neues Task-Objekt erstellt, in welchem jeweils ein Header-Objekt, ein Exercise-Objekt und Relation-Objekt gespeichert werden. In jedem Task wird beim Starten der Aufgabe in einer festen Datenbank überprüft, ob der Task bereits gespielt wird. Ist dem so, wird für die Aufgabe die bereits bestehende aufgabenspezifische Datenbank benutzt, ansonsten wird eine neue Datenbank angelegt. Hierbei werden die CREATE-TABLE- und INSERT-Statements aus der Aufgabenstellung mit Hilfe von Exercise- und Relation-Objekten ausgeführt. Ebenfalls werden die bereits erwähnten Daten generiert und in die Tabellen eingefügt. Spielt am Ende der Bearbeitung einer Aufgabe kein Spieler mehr die Aufgabe, wird die aufgabenspezifische Datenbank gelöscht. So wird der Verbleib von Datenmüll und Datenleihen vermieden. In einem Task können während des Spielens SQL-Anfragen an die Datenbank gestellt werden, hierfür wird die DBConnection-Klasse verwendet.



# Kapitel 2

## Klassenerläuterungen

Im Folgenden werden die Funktionen der einzelnen Klassen erläutert.

### 2.1 InputFile.java

Diese Datei ist der Startpunkt, wenn man eine neue XML-Datei oder einen XML-String einlesen und einen neuen Task starten möchte. Der Konstruktor erhält dabei entweder als Parameter eine XML-Datei ohne die Endung “.xml“, oder den kompletten XML-String. Im ersten Fall wird die Datei auf ihre Syntax geprüft, geparkt und für jeden Task innerhalb der Datei wird ein Task-Objekt erzeugt. Im zweiten Fall wird der String auf seine Syntax geprüft, geparkt und für jeden Task innerhalb des Strings wird ein Task-Objekt erzeugt. Zum Schluss wird mit dem String ein XML-File für die weitere Bearbeitung erzeugt.

#### Klassenvariablen

String *filename*: Name der XML-Datei ohne die Dateierdung “.xml“

List *tasks*: Liste aller Tasks, die im Konstruktor erzeugt werden

Conf *conf*: Konfigurationsobjekt

#### Konstruktor

Parameter: XML-Datei ohne die Dateierdung “.xml“

oder XML-String, Wahrheitswert, ob es eine Datei oder ein String ist Checkt und parst die Datei oder den String, dabei werden Fehler geworfen, falls die Datei nicht korrekt ist. Falls alles korrekt ist, wird für jeden Task in der Datei ein Task-Objekt erzeugt und in die Liste *tasks* gespeichert. Ist der Eingabeparameter ein String wird ein XML-File mit dem String erzeugt.

### 2.2 Exercise.java

Hier finden sich alle Inhalte aus der XML-Datei, die unter *subtasks* zu finden sind. In jedem Objekt ist immer ein Subtask gespeichert.

### Klassenvariablen

int *subtaskid*: Die Nummer der Aufgabe  
List *tasktexts*: Die Aufgabenstellung der Aufgabe in Deutsch und Englisch, falls vorhanden  
String *referencestatement*: Das Lösungsstatement der Aufgabe  
String *evaluationstrategy*: Rückgabebetyp der Aufgabe (Liste oder Menge)  
String *term*: Die Terme, die für die Lösung der Aufgabe benötigt werden  
int *points*: Die Punkte für die Aufgabe bzw. der Schwierigkeitsgrad  
String *language*: Die Sprache für die Aufgabe (wird beim Parsen nicht gesetzt)

## 2.3 Header.java

Hier finden sich alle Inhalte aus der XML-Datei, die am Anfang jedes Tasks stehen.

### Klassenvariablen

String *taskId*: Die eindeutige ID der Aufgabe  
List *title*: Die Überschrift der Aufgabe  
List *flufftext*: Die allgemeine Beschreibung der Aufgabe  
String *language*: Die Sprache für die Aufgabe (wird beim Parsen nicht gesetzt)

## 2.4 Relation.java

Hier finden sich alle Inhalte aus der XML-Datei die unter *schema* zu finden sind. In jedem Objekt ist immer eine Relation gespeichert.

### Klassenvariablen

String *intension*: Ein Create-Table-Statement der Aufgabe  
String *tableName*: Der Name der Tabelle  
List *columnInformation*: Alle Spalten mit den Namen und Typen, sowie einem boolean ob der Eintrag ein Primary Key ist und, ob er eine andere Tabelle referenziert  
List *generationTuple*: Die Tuple für die Datengeneration  
List *primaryKey*: Die Primary Keys der Tabelle  
List *tuple*: Die zugehörigen Insert-Into-Statements der Aufgabe

## 2.5 Task.java

Diese Klasse enthält alle Methoden zum Verwalten von Tasks. Die Klassenvariablen sind zum einen Listen für sämtliche Inhalte des XML-Dokuments und zum Anderen die benötigten Datenbankverbindungen, für die Spielübersicht und für die einzelnen Tasks.

### Klassenvariablen

String *name*: Name des Tasks  
String *dbName*: Name der zum Task gehörenden Datenbank  
int *players*: Anzahl der Spieler dieses Tasks

List *myHeader*: Liste der Header einer XML-Datei

List *myRelation*: Liste der Inhalte des XML-Dokuments im Bereich *schema*.

List *myExercise*: Liste der Subtasks eines XML-Dokuments

DBConnection *tmpDbConn*: Datenbankverbindung zur zum Task gehörenden Datenbank

DBConnection *fixDbConn*: Datenbankverbindung zur Datenbank für die Spielübersicht

## Methoden

### **startTask(String dbType)**

Die Methode *startTask* dient im Allgemeinen dazu, einen neuen Task zu starten, bzw. ihn zu laden, falls so ein Task noch nicht existiert.

Der Parameter *dbType* gibt den Datenbanktyp an. Dabei kann es sich entweder um eine lokale Datenbank, eine InMemory-Datenbank oder eine Online-Datenbank handeln.

Zunächst wird die Methode *checkTask()* aufgerufen. Wenn so ein Task noch nicht besteht, die Methode also den Wert *false* liefert, wird die Spielerzahl dieses Tasks auf den Wert 1 gesetzt und die Methode *createTask()* aufgerufen. Falls jedoch bereits ein solcher Task besteht, so wird dieser Task aus der Datenbank zur Spielübersicht geladen, die Spielerzahl um den Wert 1 erhöht, das XML-Dokument auf Syntaxfehler geprüft und dann schließlich geparkt. Anschließend wird noch der Task aktualisiert indem *updateTask()* aufgerufen wird.

### **checkTask()**

Die Methode *checkTask* soll prüfen ob der Task bereits existiert.

Dazu wird auf der Datenbank zur Spielübersicht ein Prepared-Statement ausgeführt, welches alle Tasks mit dem aktuellen Namen liefert (entweder liefert es einen Eintrag oder eine leere Tabelle). Dadurch kann ein boolean-Wert zurückgegeben werden, ob der Task bereits existiert oder nicht.

### **createTask(String dbType)**

Die Methode *createTask* erstellt im Allgemeinen einen neuen Task.

Der Parameter *dbType* gibt den Datenbanktyp an (s. *startTask*). Zunächst wird in der Datenbank zur Spielübersicht ein neuer Eintrag für diesen Task angelegt. Danach wird eine Datenbank für diesen Task angelegt, auf der nach dem XML-Syntax-Check und dem Parsen des Dokuments alle Statements ausgeführt werden können (alle dafür notwendigen Tabellen werden erstellt etc.)

### **loadTask()**

Die Methode *loadTask* lädt mittels Prepared-Statement einen bereits existierenden Task und eine Datenbank, auf der später alle Statements ausgeführt werden können.

### **updateTask()**

Die Methode *updateTask* aktualisiert in der Datenbank zur Spielübersicht den Eintrag zum aktuellen Task (erhöhen bzw. erniedrigen der Spielerzahl).

### **closeTask()**

Die Methode *closeTask* löscht einen Task und verringert dabei die aktuelle Spielerzahl um den Wert 1. Falls der aktuelle Task keine Spieler mehr hat, die Spielerzahl also den Wert 0 hat, wird die zum Task gehörige Datenbank mit all ihren Inhalten des Aufgabenblattes gelöscht.

### **insertToDb()**

Die Methode *insertToDb* lädt alle Inhalte der Aufgabenblätter in die zugehörige Datenbank. Dabei wird über die Liste der Relationen iteriert und jedes Element dann über ein Prepared-Statement in die Datenbank geladen.

### **selectFromDb()**

Die Methode *selectFromDb* führt alle Reference-Statements des Aufgabenblattes aus, liefert dabei aber nicht die Ergebnisse dieser Statements zurück, sie dient primär dazu die Reference-Statements auf Ausführbarkeit zu prüfen. Dabei wird über die Liste der Exercises iteriert und jedes Element dann über ein Prepared-Statement auf der Datenbank ausgeführt.

### **executeUserStatement(String statement)**

Die Methode *executeUserStatement* dient dazu ein User-Statement auszuführen und die Ergebnisse dieses Statements zurückzuliefern. Dabei wird das Statement (String) als Parameter an die Methode übergeben und dann auf der Datenbank ausgeführt. Das Ergebnis der Abfrage wird dann zurückgeliefert.

### **isUserStatementCorrect(String statement, int subtaskid)**

Die Methode *isUserStatementCorrect* dient dazu ein User-Statement auszuführen und die Ergebnisse dieses Statements mit dem Referenzstatement zu vergleichen. Sollten sie gleich sein wird true, sonst false zurückgegeben. Dabei wird das Statement (String) und die Aufgabennummer (int) als Parameter an die Methode übergeben.

### **printData()**

Die Methode *printData* gibt alle Inhalte des Aufgabenblattes aus. Es wird für jede der drei Listen nacheinander (für die Header, Relations und Exercises) die Anzahl der Elemente und anschließend die Elemente selbst ausgegeben.

### **generateData(String generateType)**

Die Methode *generateData* generiert aus den User-GenerationTupeln oder den Referenzstatements die Insert-Statements und führt sie aus. Als Parameter wird ein String mit `userData` oder `referenceStatement` übergeben.

## Kapitel 3

# Datengenerierung

### 3.1 XML-Tags

Wir benötigen zusätzliche Informationen in der XML-Datei, damit wir Daten generieren können. Ein Beispiel zeigen wir in Listing 3.1. Innerhalb des **relation**-Tags benötigen wir ein **extensiongeneration**-Tag. In diesem wird mit dem Tag **generationtuple** die Daten für die Insert-Statements generiert. Dieser Tag kann mehrmals benutzt werden, sollte mehrere verschiedene Daten der Tabelle hinzugefügt werden. Der genaue Aufbau für die Datengenerierung folgt in Abschnitt 3.2.

Listing 3.1: XML-Datei für die Meta-informationen

```
<relation>
  <intension>
    CREATE TABLE Effects(id int not null primary key,
      ename varchar(100),
      description varchar(1000))
  </intension>
  <extension>
    <tuple>
      INSERT INTO Effects(id, ename) VALUES (1, 'Dizzyness')
    </tuple>
  </extension>
  <extensiongeneration>
    <generationtuple>
      10;none;max,int,20;random,effect,80;random,string,200
    </generationtuple>
    <generationtuple>weitere</generationtuple>
  </extensiongeneration>
</relation>
```

### 3.2 Aufbau für die Datengenerierung

Im Folgenden wird erläutert, wie Daten generiert werden können. Der Allgemeine Aufbau ist (keine Leerzeichen vor oder nach Kommata/Semikolons):



`Anzahl;Ref;Spalte1;Spalte2;Spalte3;...`

1. Anzahl: Anzahl besteht aus einer Zahl oder `span,Anfang,Ende`.
2. Ref: Mit Ref kann angegeben werden, dass sich die generierten Daten auf Werte aus einer anderen Tabelle beziehen (referenzieren). Hierbei wird unterschieden zwischen drei Möglichkeiten:
  - `none`, was besagt, dass es keine Referenzierung gibt
  - `refAll`, was besagt, dass für alle Werte der referenzierten Spalte genau Anzahl Reihen generiert werden
  - `refRandom`, was besagt, dass für einen zufälligen Wert der referenzierten Spalte genau Anzahl Reihen generiert werden

Der Aufbau von Ref ist der Folgende:

`RefType,referenzierter Tabellename,referenzierter Spaltenname`

3. Spalte: Hier werden die Daten für eine Spalte generiert und alle Parameter werden durch ein Komma getrennt. Also ist der Aufbau einer Spalte:

`Funktionsname,Para1,Para2,...`

### **Funktionsname einer Spalte (mit den dazugehörigen Parametern):**

- random: zufälliger Eintrag
  - `int, int+, double, string(optional mit Länge), firstname, lastname, fullname, date, business, street, adress, email, custom(name, opt. random, opt. Default)`
    - \* für den Parameter name nach custom gibts folgende Möglichkeiten: `alias, chemical element, colour, comic title, effect, gender, genre, land, occupation, programming language, publisher, size`

die Angaben in Klammern sind weitere Parameter Beispiel: `random,string,5` oder `random,firstname` oder `random,colour,80,weiß`

- min: mindestens größer als
  - `int(min), double(min)`
- max: maximal so groß
  - `int(max), double(max)`
- between: zwischen zwei Werten
  - `int(min, max), double(min, max)`
- gauss: gaussche Normalverteilung
  - `int(Mittelwert, Standardabweichung), double(Mittelwert, Standardabweichung)`

- list: liste von Zahlen mit Parameter Start
- fix: selbst ausgedachter Wert: Wert mit " also fix,'ausgedacht'

weitere Beispiele: min,int,5 oder gauss,int,5,4

Sollen für eine Tabelle Daten generiert werden, die eine andere Tabelle referenziert, muss bei der entsprechenden Spalte angegeben werden, auf welche Spalte der referenzierten Tabelle sich diese Spalte bezieht. Wie in Listing 3.2 ersichtlich wird für die Spalte **ref**, **Tabellenname**, **Spalte** angegeben, also z.B. **ref,powders,id**.

Listing 3.2: weitere XML-Datei für die Meta-informationen

```
<relation>
  <intension>
    CREATE TABLE p2e(pid int not null, eid int not null,
      strength int, PRIMARY KEY (pid, eid),
      FOREIGN KEY (pid) REFERENCES Powders(id),
      FOREIGN KEY (eid) REFERENCES Effects(id))
  </intension>
  <extension>
    <tuple>
      INSERT INTO p2e VALUES (1, 1, 4)
    </tuple>
  </extension>
  <extensiongeneration>
    <generationtuple>
      5;none;ref,powders,id;ref,effects,id;max,int,200
    </generationtuple>
  </extensiongeneration>
</relation>
```