Framework

Vorwort

Da ich am 01.10 eine neue Stelle als Java Entwickler bei einer Bank angetreten habe und mich beruflich mit Datenbank Anwendungen und ORM (englisch object-relational mapping) beschäftige, habe ich das Projekt genutzt um mich dieser Thematik in R anzunehmen. Neben der reinen (Objektorientierten) Programmierung stand dabei die Softwarepaketierung im Fokus. Dafür habe ich das Projekt als eigenständiges Paket umgesetzt und für die weitere Distribution auf Github hochgeladen.

Die kommentierung von Code habe ich aus Übersichtsgründen weggelassen und dafür mehr Aufwand in die Benennung von (sprechenden) Variablen und grundlegender Paketarchitektur aufgebracht. Des Weiteren kann mit ?rdao::DieKlasse die Hilfe aufgerufen werden. Zeitlich war es mir leider nicht möglich, die Hilfe der einzelnen Klassen inhaltlich zu füllen.

Aus technischen Gründen musste ich während der Entwicklung von einer Access Datenbank abweichen. Alternativ habe ich daher aus dem Diamond Datensatz eine .db Datei erstellt und diese unter "rdao/db files external/Diamonds.db" auf Github abgelegt. Der originale Datensatz kann dem Paket nach dem Laden mit rdao::diamonds (siehe ?rdao::diamonds) entnommen werden.

Installation

```
# helper
installer <- function(pkg){
  new.pkg <- pkg[!(pkg %in% installed.packages()[, "Package"])]
  if (length(new.pkg))
    install.packages(new.pkg, dependencies = TRUE)
  sapply(pkg, require, character.only = TRUE)
}
packages<- c("devtools","R6","DBI")
installer(packages)
#> Loading required package: devtools
#> Loading required package: usethis
#> Loading required package: R6
#> Loading required package: DBI
#> devtools       R6      DBI
#>     TRUE     TRUE     TRUE
devtools::install_github("Chrisnice89/rdao")
#> Skipping install of 'rdao' from a github remote, the SHA1 (237bc756) has not changed since last inst
#>   Use `force = TRUE` to force installation
```

Basics

Laden des Pakets

```
library(rdao)
#> This package ist created,developed and copyrighted by Christoph Nitz.
#> Interested parties may contact <Christoph.Nitz89@gmail.com>
```

Für den Testlauf des Paketes den Pfad zur db (Gitub-Ordner: "db files external/Diamonds.db") in einer Variablen ablegen:

```
path<-"/Users/cnitz/Dev/R/rdao/db files external/Diamonds.db"
```

Um eine Klasse aus dem Paket verwenden zu können, muss eine 'factory()' Funktion aufgerufen werden. Unter einer Klasse (auch Objekttyp genannt) versteht man in der objektorientierten Programmierung ein abstraktes Modell bzw. einen Bauplan für eine Reihe von ähnlichen Objekten:

```
f <- connectionFactory()
#> <Validator> for parent class: <SqlFactory> created
#> <SqlFactory> created
class(f)
#> [1] "SqlFactory" "R6"
```

Die `connectionFactory()` Funktion instanziert intern ein Objekt der Klasse 'SqlFactory'. Sobald die factory geladen wurde, kann diese jederzeit wieder verwendet werden um ein korrespondierendes Objekt zu erstellen. Eigenschaften und Methoden der instanzierten Objekte können mit `$` aufgerufen werden:

```
#Für ein db File
builder.dbFile<-f$dbFile(path)
#> <Validator> for parent class: <Builder> created
#> <Builder> for provider: <dbFile> created

class(builder.dbFile)
#> [1] "Builder" "R6"

#Für eine MS Access Db
builder.Access<-f$msAccess("/file doesnt exisist")
#> <Validator> for parent class: <Builder> created
#> <Builder> for provider: <msAccess> created

#Für MySql / Nicht implementiert
builder.mySql<-f$mySql(database = "test_db",uid = "mySql",pwd = "password",host = "localhost",port = 543
#> <Validator> for parent class: <Builder> created
#> <Builder> for provider: <mySql> created
```

Methoden und Felder (properties):

```
#Methode ohne Parameterübergabe
builder.dbFile$addCredentials()
#> <Validator> for parent class: <Credentials> created
#> <Credentials> for User: <> created

#Methode mit Parameterübergabe
builder.dbFile$addCredentials(username = "Admin",password = "SesameOpen")
#> <Validator> for parent class: <Credentials> created
#> <Credentials> for User: <Admin> created

#Zugriff auf Felder ohne abschließende '()'
builder.dbFile$path
#> [1] "/Users/cnitz/Dev/R/rdao/db files external/Diamonds.db"

#Zuweisung eines Wertes zu einem Feld
builder.dbFile$path<-"new path"
builder.dbFile$path
#> [1] "new path"
builder.dbFile$path<-path

#Readonly Feld - Wert des Feldes kann nicht geändert werden #Error!
builder.dbFile$builderProvider
```

```
#> [1] "dbFile"

#builder.dbFile$builderProvider<-"msAccess"
#Fehler in builder.dbFile$builderProvider <- "msAccess" :
#kann den Wert einer festgestellten Bindung für 'builderProvider' nicht ändern
```

Methoden Verkettung:

```
class(builder.dbFile$credentials)
#> [1] "Credentials" "R6"
builder.dbFile$credentials$username
#> [1] "Admin"
```

Unter Method chaining (engl. chaining = Verkettung) wird in der Welt der objektorientierten Programmierung (OOP) eine spezielle Syntax verstanden, die das Ausführen einer Reihe von Methoden eines Objektes beschreibt.

Anwendung

Data Access Object (DAO, englisch für Datenzugriffsobjekt) ist ein Entwurfsmuster, das den Zugriff auf unterschiedliche Arten von Datenquellen (z. B. Datenbanken, Dateisystem) so kapselt, dass die angesprochene Datenquelle ausgetauscht werden kann, ohne dass der aufrufende Code geändert werden muss. Dadurch soll die eigentliche Programmlogik von technischen Details der Datenspeicherung befreit werden und flexibler einsetzbar sein. DAO ist also ein Muster für die Gestaltung von Programmierschnittstellen (APIs).

Die konkrete Implementierung der Objekte wurde mit Hilfe von R6 Klassen umgesetzt. Aus programmatischen Gründen wurde dabei gänzlich auf aktive Bindungen innerhalb der Klassen verzichtet. Des Weiteren wird im Kern für die Kommunikation zwischen Datenbank(en) und der Anwednung auf das DBI Interface zurückgegriffen.

Eigenschaften

- DAOs abstrahieren den Zugriff auf Datenbanken nicht vollständig, da sie nicht für die Transformation der Daten in die Struktur der Datenbank verantwortlich sind.
- DAOs sind jeweils für ein spezielles Speichermedium optimiert. Der Zugriff auf dieses Medium wird über das vom DAO vorgegebene bzw. zu implementierende API vorgenommen.
- DAOs minimieren den Portierungsaufwand einer Anwendung beim Wechsel des Speichermediums.

Die Methoden der `SqlFactory` instanzieren intern ein Objekt vom Typ 'Builder'. Der Erbauer (englisch builder) ist ein Entwurfsmuster (buildern pattern) aus dem Bereich der Softwareentwicklung. Es gehört zur Kategorie der Erzeugungsmuster und trennt die Konstruktion komplexer Objekte von deren Repräsentationen, wodurch dieselben Konstruktionsprozesse wiederverwendet werden können.

Der Einsatz des Erbauer-Entwurfsmusters bietet sich an, wenn

1 Zu einem komplexen Objekt unterschiedliche Repräsentationen existieren sollen

2 Die Konstruktion eines komplexen Objekts unabhängig von der Erzeugung der Bestandteile sein soll

3 Der Konstruktionsablauf einen internen Zustand erfordert, der vor einem Klienten verborgen werden soll

Erstellen einer Verbindung mit`builder$build()`

```
connection.dbFile<-builder.dbFile$build()
#> <Validator> for parent class: <SqlConnection> created
#> <SqlConnection>> for provider: <dbFile> created

class(connection.dbFile)
#> [1] "SqlConnection"          "Abstrakt SqlConnection"
#> [3] "R6"
```

```
#connection.Accees<-builder.Access$build()
#Fehler: <Builder>
#Proc: <build()>
#Beschreibung: <Ungültiger Pfad "/file doesnt exisist">

#connection.dbFile<-builder.mySql$build()
#Fehler: <Builder>
#Proc: <build()>
#Beschreibung: <Noch nicht implementiert: mySql>
```

Der Konstruktionsprozess wird an einer dedizierten Stelle (im `Builder`) gesteuert; spätere Änderungen – etwa ein Mehrphasen-Konstruktionsprozess statt einer Einphasen-Konstruktion – lassen sich ohne Änderung der Klienten realisieren. Es besteht eine enge Kopplung zwischen Produkt, konkretem Erbauer und den am Konstruktionsprozess beteiligten Klassen (hier die `SqlFactory` als Direktor).

Die Verbindung

```
connection.dbFile$isConnected()
#> [1] FALSE
connection.dbFile$connect()
#> [1] TRUE
connection.dbFile$isConnected()
#> [1] TRUE
connection.dbFile$disconnect()
#> [1] TRUE
connection.dbFile$isConnected()
#> [1] FALSE
```

Die SqlConnection Klasse verwendet in seinem Nucleus das R Paket DBI. Im Gegensatz zu diesem erzeugt der Builder aber per default eine geschlossene Verbindung. Diese kann explizit per `connect()` geöffnet oder `disconnect()` geschlossen werden. Weitere Details zum Verhalten der Connection Klasse werden im nächsten Abschnitt näher erläutert. Die Methoden zum steuern der Verbindung geben jeweils einen boolean Wert als Indikator zurück.

Abfragen

```
query<-connection.dbFile$createQuery(sql = "SELECT * FROM diamonds")
#> <Validator> for parent class: <SqlCommand> created
#> <SqlCommand> :: <SELECT * FROM diamonds>
#> for provider: <dbFile> created

class(query)
#> [1] "SqlCommand"          "Abstrakt SqlCommand" "R6"

query$provider
#> [1] "dbFile"

query$sql
#> [1] "SELECT * FROM diamonds"

#query$sql<-"SELECT * FROM diamonds"
#Fehler in query$sql <- "SELECT * FROM emeralds" :
#kann den Wert einer festgestellten Bindung für 'sql' nicht ändern
```

Mit `createQuery()` instanziiert die `SqlConnection` ein SqlCommand. Die Klasse dient als CRUD (Create, Read, Update, Delete) Interface. Die aktuelle Implementierung kann SQL Statements nur als Konkatenation verarbeiten. Um SQL-Injection (dt. SQL-Einschleusung) zu unterbinden sowie die Flexibilität zu erhöhen, könnte in einem zukünftigen Release parametrisierte Abfragen implementiert werden.

Im Vergleich:

```
quality<-"Premium"

#quality<-"Premium"
query.premium<-connection.dbFile$createQuery(sql = "SELECT * FROM diamonds WHERE cut=?")
#> <Validator> for parent class: <SqlCommand> created
#> <SqlCommand> :: <SELECT * FROM diamonds WHERE cut=?>
#> for provider: <dbFile> created
#qquery.premium$addParameter(name="quality", value= quality)
query.premium$sql
#> [1] "SELECT * FROM diamonds WHERE cut=?"

query.premium<-connection.dbFile$createQuery(sql = paste("SELECT * FROM diamonds WHERE cut='",quality,"
#> <Validator> for parent class: <SqlCommand> created
#> <SqlCommand> :: <SELECT * FROM diamonds WHERE cut='Premium'>
#> for provider: <dbFile> created
query.premium$sql
#> [1] "SELECT * FROM diamonds WHERE cut='Premium'"
```

Darauf aufbauend kann das SqlCommand dann um die Möglichkeit des Prepared Statements erweitert werden. Ein Prepared Statement ist eine sogenannte vorbereitete Anweisung für ein Datenbanksystem. Im Gegensatz zu gewöhnlichen Statements enthält es noch keine Parameterwerte. Stattdessen werden dem Datenbanksystem Platzhalter übergeben. Soll ein Statement mit unterschiedlichen Parametern mehrere Male (z. B. innerhalb einer Schleife) auf dem Datenbanksystem ausgeführt werden, können Prepared Statements einen Geschwindigkeitsvorteil bringen, da das Statement schon vorübersetzt im Datenbanksystem vorliegt und nur noch mit den neuen Parametern ausgeführt werden muss.

Ergebniss einer Abfrage

Mit `fetch()` wird die Query ausgeführt und die `SqlConnection` liefert über das `SqlCommand` Interface ein Objekt der Klasse `SqlResult` zurück. Handelt es sich bei der Abfrage um eine SQL-Aktionsabfrage, bspw.ein "INSERT" oder "DELETE" wird die Query mit dem Befehl `execute()` ausgeführt und als Ergebnis wird die Anzahl der vom SQL-Statement betroffenen Datensätze zurück geliefert.

```
result<-query$fetch()
#> <Validator> for parent class: <SqlResult> created
#> <SqlCommand> :: <SELECT * FROM diamonds>
#> for provider: <dbFile> ausgeführt

result$rows()
#> [1] 53940

class(result)
#> [1] "SqlResult" "R6"
```

Mit `data` kann direkt auf die Daten zugegriffen werden:

```
#(Achtung Feld Zugriff)
head(result$data)
#>   carat       cut color clarity depth table price    x    y    z
#> 1  0.23     Ideal     E     SI2  61.5    55   326 3.95 3.98 2.43
```

```
#> 2  0.21   Premium     E     SI1  59.8    61   326 3.89 3.84 2.31
#> 3  0.23      Good     E     VS1  56.9    65   327 4.05 4.07 2.31
#> 4  0.29   Premium     I     VS2  62.4    58   334 4.20 4.23 2.63
#> 5  0.31      Good     J     SI2  63.3    58   335 4.34 4.35 2.75
#> 6  0.24 Very Good     J    VVS2  62.8    57   336 3.94 3.96 2.48

#result$data<-NULL
#Fehler in result$data <- NULL :
#kann den Wert einer festgestellten Bindung für 'data' nicht ändern
```

Verhalten der Verbindung bei einer Abfrage

```
connection.dbFile$isConnected()
#> [1] FALSE

result.premium<-query.premium$fetch(disconnect = TRUE)
#> <Validator> for parent class: <SqlResult> created
#> <SqlCommand> :: <SELECT * FROM diamonds WHERE cut='Premium'>
#> for provider: <dbFile> ausgeführt

connection.dbFile$isConnected()
#> [1] FALSE

head(result.premium$data)
#>   carat     cut color clarity depth table price    x    y    z
#> 1  0.21 Premium     E     SI1  59.8    61   326 3.89 3.84 2.31
#> 2  0.29 Premium     I     VS2  62.4    58   334 4.20 4.23 2.63
#> 3  0.22 Premium     F     SI1  60.4    61   342 3.88 3.84 2.33
#> 4  0.20 Premium     E     SI2  60.2    62   345 3.79 3.75 2.27
#> 5  0.32 Premium     E      I1  60.9    58   345 4.38 4.42 2.68
#> 6  0.24 Premium     I     VS1  62.5    57   355 3.97 3.94 2.47
```

Die Verbindung wird nur für einen Task geöffnet und anschließend sofort wieder geschlossen. Der optionale Steuerungsparameter `disconnect` ist per default auf TRUE gesetzt.

Testlauf

Objektorientierte Programmiersprachen (OOP) kapseln Daten und Verhalten in Objekten, hingegen legen relationale Datenbanken Daten in Tabellen ab. Die beiden Paradigmen sind grundlegend verschieden. So kapseln Objekte ihren Zustand und ihr Verhalten hinter einer Schnittstelle und haben eine eindeutige Identität.

Für den Testlauf wird der diamonds Datensatz verwendet und beispielhaft eine einfache (fiktive) Geschäftslogik bestehend aus einer Tabelle sowie nur lesenden Aufgaben angenommen.

Im Testlauf soll exemplarisch dargestellt werden, welche Möglichkeiten R im Bereich der Objektorientierten Programmierung bietet und wie eine größere Datenbank Anwendung strukturiert nach den Grundsätzen der Objektorientierung implementiert werden könnte.

Geschäftslogik

```
business<-diamondsFactory(connection.dbFile)
#> <Validator> for parent class: <Diamonds interface> created
#> <Validator> for parent class: <Diamonds interface> created
#> <Validator> for parent class: <HashMap> created
```

```
#> <Validator> for parent class: <HashMap> created

class(business)
#> [1] "Diamonds interface"     "Abstrakt businessLogic"
#> [3] "R6"

diamonds.best<-business$loadBestQuality()
#> <Validator> for parent class: <SqlCommand> created
#> <SqlCommand> :: <SELECT * FROM diamonds WHERE cut = 'Very Good'>
#> for provider: <dbFile> created
#> <Validator> for parent class: <SqlResult> created
#> <SqlCommand> :: <SELECT * FROM diamonds WHERE cut = 'Very Good'>
#> for provider: <dbFile> ausgeführt

head(diamonds.best$data)
#>   carat       cut color clarity depth table price    x    y    z
#> 1  0.24 Very Good     J    VVS2  62.8    57   336 3.94 3.96 2.48
#> 2  0.24 Very Good     I    VVS1  62.3    57   336 3.95 3.98 2.47
#> 3  0.26 Very Good     H     SI1  61.9    55   337 4.07 4.11 2.53
#> 4  0.23 Very Good     H     VS1  59.4    61   338 4.00 4.05 2.39
#> 5  0.30 Very Good     J     SI1  62.7    59   351 4.21 4.27 2.66
#> 6  0.23 Very Good     E     VS2  63.8    55   352 3.85 3.92 2.48

diamonds.selected<-business$loadColumns("carat","color")
#> <Validator> for parent class: <SqlCommand> created
#> <SqlCommand> :: <SELECT carat FROM diamonds>
#> for provider: <dbFile> created
#> <Validator> for parent class: <SqlResult> created
#> <SqlCommand> :: <SELECT carat FROM diamonds>
#> for provider: <dbFile> ausgeführt

head(diamonds.selected$data)
#>   carat
#> 1  0.23
#> 2  0.21
#> 3  0.23
#> 4  0.29
#> 5  0.31
#> 6  0.24
```

Innerhalb der Businesslogik werden die relationen Zusammenhänge der zugrundeligenden Datenbank implementiert. Die Businesslogik kann um komplexe SQL-Abfragen über meherere Tabellen sowie weiterverarbeitende, Clientseitige Dunkelverabreitung erweitert werden. Beispiel: Kreditdaten kommen aus der Datenbank in der Businesslogik an, fließen in der Businesslogik in ein Prognosemodell und werden dann erst ans Frontend geliefert.

Errorhandling

```
diamonds.lowbudget<-business$loadQuality("Poor","Fair", "Good")
#> Warning: <Diamonds interface>
#> Proc: <loadQuality()>
#> Beschreibung: <Nur folgende Qualitäten können abgefragt werden: Fair;Good;Ideal;Premium;Very Good>
```

```
diamonds.lowbudget<-business$loadQuality("Fair", "Good")
#> <Validator> for parent class: <SqlCommand> created
#> <SqlCommand> :: <SELECT * FROM diamonds WHERE cut IN ("Fair")>
#> for provider: <dbFile> created
#> <Validator> for parent class: <SqlResult> created
#> <SqlCommand> :: <SELECT * FROM diamonds WHERE cut IN ("Fair")>
#> for provider: <dbFile> ausgeführt

head(diamonds.lowbudget$data)
#>   carat  cut color clarity depth table price    x    y    z
#> 1  0.22 Fair     E     VS2  65.1    61   337 3.87 3.78 2.49
#> 2  0.86 Fair     E     SI2  55.1    69  2757 6.45 6.33 3.52
#> 3  0.96 Fair     F     SI2  66.3    62  2759 6.27 5.95 4.07
#> 4  0.70 Fair     F     VS2  64.5    57  2762 5.57 5.53 3.58
#> 5  0.70 Fair     F     VS2  65.3    55  2762 5.63 5.58 3.66
#> 6  0.91 Fair     H     SI2  64.4    57  2763 6.11 6.09 3.93
```

Das Errorhandling wird ebenfalls in der Businesslogik abgebildet.

Objektrelationale Abbildung

Des Weiteren kann mit Hilfe von Objekt-orientierter Programmierung und dem `rdao` framework eine Objektrelationale Abbildung dagestellt werden. Eine Objektrelationale Abbildung (englisch object-relational mapping, ORM) ist eine Technik der Softwareentwicklung, mit der ein in einer objektorientierten Programmiersprache geschriebenes Anwendungsprogramm seine Objekte in einer relationalen Datenbank ablegen kann. Dem Programm erscheint die Datenbank dann als objektorientierte Datenbank, was die Programmierung erleichtert.

Im einfachsten Fall werden Klassen auf Tabellen abgebildet, jedes Objekt entspricht einer Tabellenzeile und für jedes Attribut wird eine Tabellenspalte reserviert. Die Identität eines Objekts entspricht dem Primärschlüssel der Tabelle:

```
diamonds<-business$loadDiamonds()
#> <Validator> for parent class: <SqlCommand> created
#> <SqlCommand> :: <SELECT * FROM diamonds>
#> for provider: <dbFile> created
#> <Validator> for parent class: <SqlResult> created
#> <SqlCommand> :: <SELECT * FROM diamonds>
#> for provider: <dbFile> ausgeführt

class(diamonds)
#> [1] "list"

class(diamonds[[1]])
#> [1] "Diamond" "R6"
```

Fazit

Mit dem DBI- und dem R6 Paket existieren in R mächtige Werkzeuge zum entwickeln von Objektorientierten Datenbanklösungen.

Die Anwendung kann als Blaupause für weitere Implementierungen dienen und vermittelt einen ersten Eindruck wie eine Objektorientierte Programmstruktur in R aussehen könnte.

Aufgrund der Komplexität des Themas und der im Rahmen einer Projektarbeit bemessenen Zeit verbleiben viele offene Fragen:

Wie könnten tatsächliche Interface implementiert werden? Wie schreibt die Anwendung das gemappte Objekt (effizient) zurück in die Datenbank? Wie könnte das SqlResult noch weiter gekapselt werden?