Kapitel 12
Objektorientiertes Modellieren

Objektorientiertes Programmieren hat immer etwas mit Modellieren zu tun. Nicht selten sind objektorientierte Programme vereinfachte Abbildungen eines Wirklichkeitsausschnitts.

Häufig lassen sich Programmierer auch einfach nur von der Realität inspirieren und verwenden Dinge der Welt als Metaphern, um die Struktur ihres Programms möglichst anschaulich und einleuchtend zu gestalten, ohne besondere Realitätsnähe anzustreben.

In diesem Kapitel sprechen wir die Phasen der objektorientierten Software-Entwicklung an und skizzieren an einem Beispiel den Weg von der objektorientierten Analyse einer Problemstellung bis zum fertigen objektorientierten Programm.

Bisher haben wir nur *einzelne* Klassen definiert. Ein vollständiges objektorientiertes Programm besteht (in der Regel) aus einem ganzen *Ensemble* von Klassen, die aufeinander abgestimmt sind und deren Objekte während des Programmlaufs interagieren. Zwischen den Objekten der Klassen gibt es Beziehungen oder Assoziationen. In den Beispielen dieses Kapitels werden die wichtigsten Typen von Assoziationen beschrieben. An dieser Stelle ein Hinweis: In der Web-Dokumentation zu diesem Buch finden Sie noch weitere Beispiele für objektorientierte Modellierung wie z.B. die Modellierung von Wegesystemen mit Graphen.

12.1 Phasen einer objektorientierten Software-Entwicklung

Die Entwicklung einer Software kann ein aufwändiger und langwieriger Prozess sein, an dem – bei größeren Projekten – viele Personen beteiligt sind. Um diesen komplexen Vorgang einigermaßen überschaubar und beherrschbar zu machen, gibt es aus dem Software-Engineering Ablaufmodelle, an denen man sich orientieren kann. Ein bekanntes Modell unterscheidet folgende Phasen:

12.1.1 Objektorientierte Analyse (OOA)

Hier geht es um die Analyse eines Wirklichkeitsausschnitts, der durch ein Software-System abgebildet werden soll. Gemeinsam mit den zukünftigen Anwendern und Fachexperten mit Spezialkenntnissen über den zu modellierenden Realitätsbereich wird das erwünschte Verhalten des Systems festgelegt und umgangssprachlich beschrieben. Klassen und Beziehungen zwischen ihnen werden herauskristallisiert und benannt – zunächst noch losgelöst von der Syntax einer konkreten Programmiersprache. Man gelangt zu einem abstrakten Modell, das man als UML-Klassendiagramm formalisieren kann, das aber noch völlig frei ist von Aspekten der technischen Realisierung.

12.1.2 Objektorientierter Entwurf (OOD)

Im Entwurf (Object Oriented Design, OOD) wird das abstrakte OOA-Modell konkretisiert und verfeinert. Erstmals werden Gesichtspunkte der technischen Realisierung und der Effizienz in die Modellierung einbezogen. Typische Fragen, die während der Entwurfsphase geklärt werden müssen, sind:

- Wahl der Programmiersprache
- Sichtbarkeit von Attributen und Methoden
- Aufteilung des Gesamtsystems auf Module, die sich in separaten Dateien befinden
- Festlegung von Vor- und Nachbedingungen für Methoden

12.1.3 Objektorientierte Programmierung (OOP)

Unter Verwendung einer objektorientierten Programmiersprache (hier: Python) wird das OOD-Modell implementiert, das heißt, es wird ein funktionstüchtiges Programm erstellt, dokumentiert und getestet.

12.2 Beispiel: Modell eines Wörterbuchs

Der Schwerpunkt dieses Buches liegt in der objektorientierten Programmierung. Wir gehen also meistens davon aus, dass die OOA- und OOD-Phase der Modellbildung abgeschlossen ist. Ohnehin sind die Programmbeispiele in diesem Buch so einfach, dass aufwändige Vorplanung verzichtbar ist. Dennoch ist es für das Erlernen von Programmiertechniken hilfreich, wenn man wenigstens in groben Zügen einige Konzepte der objektorientierten Modellentwicklung kennt. In diesem Buch werden an verschiedenen Stellen Darstellungen von OOA- und OOD-Modellen verwendet, um die Struktur von Software-Systemen zu visualisieren.

Wenden wir uns also nun einem Fallbeispiel zu, das die wesentlichen Ideen veranschaulichen soll. Es geht um die objektorientierte Modellierung eines interaktiven Wörterbuchs.

12.2.1 OOA: Entwicklung einer Klassenstruktur

Abbildung 12.1 deutet die Situation an, in der man sich befindet, wenn man nach der Übersetzung einer englischen Vokabel sucht. Man kann zwei Aspekte unterscheiden. Zum einen benötigt man ein Wörterbuch, in dem zu vielen englischen Wörtern jeweils die deutsche Bedeutung aufgeführt ist. Meist besitzt man in seinem Bücherschrank mehrere Wörterbücher für unterschiedliche Sprachen, von denen man eines auswählen muss. Aus objektorientierter Sicht handelt es sich um unterschiedliche Instanzen der Klasse Wörterbuch.

Neben dem Wörterbuch brauchen wir eine Arbeitsumgebung, in der wir das Wörterbuch benutzen. Dazu gehört etwa ein Tisch passender Größe, der ausreichend beleuchtet ist, vielleicht noch ein Bleistift für Notizen usw. An diesen Tisch kann man sich setzen, um mit dem Wörterbuch zu arbeiten. Diese Arbeitsumgebung ermöglicht also die Benutzung des Wörterbuchs. Es ist die Schnittstelle zwischen Benutzer und Wörterbuch, man spricht auch von Benutzungsoberfläche. Beachten Sie, dass die Benutzungsoberfläche auf die Bedürfnisse des Menschen, der sie verwendet, zugeschnitten sein muss.

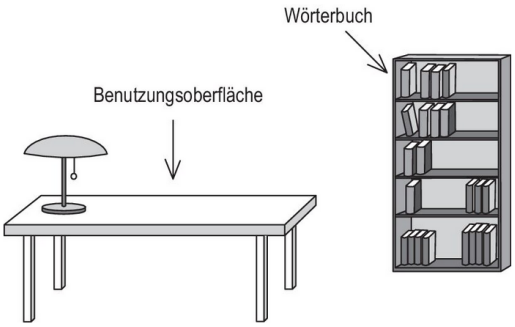

Abb. 12.1: Objekte für die Suche nach Vokabeln

Dieses Bild vom strukturellen Aufbau einer Bibliothek nehmen wir als Ausgangspunkt für die Entwicklung eines objektorientierten Programms. Wir differenzieren zwischen zwei Klassen: `Wörterbuch` und `Benutzungsoberfläche`. Damit haben wir als erstes Ergebnis unserer Analyse ein einfaches statisches Modell geschaffen (Abbildung 12.2), das wir später verfeinern werden.

Abb. 12.2: Einfaches UML-Klassendiagramm für das Modell eines Wörterbuchs

Geschäftsprozesse (use cases)

Während der objektorientierten Analyse wird festgelegt, was das Zielprodukt – die Software – alles leisten soll. Eine bewährte Methode ist die Spezifikation von Geschäftsprozessen (*use cases*). Ein Team aus Software-Entwicklern und

späteren Nutzern (Kunden, Auftraggeber) überlegt sich typische Anwendungsfälle (*use cases*) für die Software. Diese werden mit Namen belegt und in einem Geschäftsprozessdiagramm zusammengestellt. Abbildung 12.3 zeigt ein sehr einfaches Beispiel für unser Projekt.

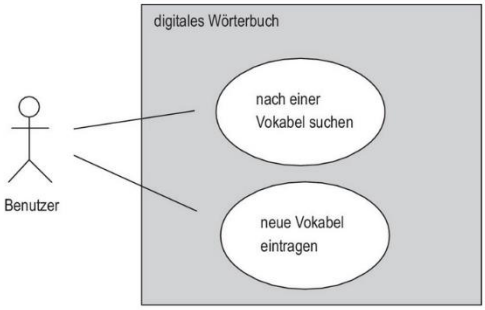

Abb. 12.3: Einfaches Geschäftsprozessdiagramm mit einem Akteur

Wir gehen von einem einzigen Akteur aus, der nach Vokabeln sucht oder neue Vokabeln in das Wörterbuch eintragen will. Bei größeren Systemen unterscheidet man häufig mehrere Benutzer, denen unterschiedliche Geschäftsprozesse zugeordnet werden. Abbildung 12.4 zeigt eine komplexere Variante für das Wörterbuch-Beispiel.

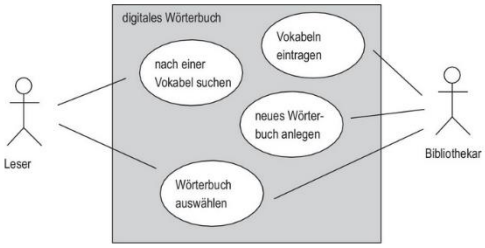

Abb. 12.4: Geschäftsprozessdiagramm mit zwei Akteuren

Jeder einzelne Geschäftsprozess wird umgangssprachlich beschrieben. Dabei orientiert man sich meist an folgendem Schema:

- Name des Geschäftsprozesses
- Ziel
- Vorbedingung: Zustand vor Beginn des Geschäftsprozesses
- Nachbedingung Erfolg: Zustand nach erfolgreicher Durchführung des Geschäftsprozesses
- Nachbedingung Fehlschlag
- Akteure
- Auslösendes Ereignis
- Beschreibung, d.h. Auflistung der einzelnen Aktionen aus Benutzersicht

Formulieren wir nun Beschreibungen der beiden Geschäftsprozesse aus Abbildung 12.2.

Geschäftsprozess: Vokabel suchen

- Ziel: Übersetzung einer Vokabel wird abgefragt.
- Vorbedingung: -
- Nachbedingung Erfolg: Eine Liste mit Übersetzungen (in Sprache 2) der eingegebenen Vokabel (in Sprache 1) wird in gut lesbarer Form auf dem Bildschirm ausgegeben. Das System ist dann wieder bereit für die Verarbeitung eines neuen Ereignisses (Auswahl einer Funktion im Menü).
- Nachbedingung Misserfolg: Ausgabe einer Fehlermeldung (»Vokabel unbekannt«)
- Akteur: Benutzer
- Auslösendes Ereignis: Auswahl der Funktion »Übersetzung« im Menü
- Beschreibung:

  Eingabe eines Wortes in Sprache 1

  Ausgabe aller gespeicherten Übersetzungen in Sprache 2

Geschäftsprozess: Neue Vokabel eintragen

- Ziel: Wörterbuch um eine Vokabel und ihre Übersetzung erweitern
- Vorbedingung: -
- Nachbedingung Erfolg: Die eingegebene Vokabel in Sprache 1 und eine Liste von Übersetzungen in Sprache 2 sind im Wörterbuch verzeichnet Falls es vor dem Geschäftsprozess bereits einen Eintrag für die Vokabel in Sprache 1 gab, wird die Liste der Übersetzungen erweitert. Das System ist bereit für die Verarbeitung eines neuen Ereignisses (Auswahl einer Funktion im Menü).
- Nachbedingung Misserfolg: -
- Akteur: Benutzer
- Auslösendes Ereignis: Auswahl der Funktion »Neues Wort eingeben« im Menü
- Beschreibung:

  Eingabe eines Wortes in Sprache 1

  Eingabe einer Übersetzung in Sprache 2

  Eingabe weiterer Übersetzungen in Sprache 2

  Beenden des Geschäftsprozesses (durch Betätigen der `Enter`-Taste)

Die Geschäftsprozesse spiegeln das gewünschte Verhalten des Systems aus Benutzersicht wider. Sie enthalten aber keine vollständige Beschreibung der Systemfunktionalität. Denn es gibt auch Operationen, die automatisch ablaufen, ohne dass ein Benutzer sie explizit anstößt. In unserem Beispiel muss der »Wortschatz« des Wörterbuchs beim Start des Programms aus einer Datei geladen werden. Da der Zustand des Wörterbuchs durch Eingabe neuer Wörter erweitert werden kann, muss der neue Zustand vor Beendigung des Programms dauerhaft gespeichert werden.

Auf der Basis der Geschäftsprozesse verfeinern wir die statische Klassenstruktur unseres Systems und geben für die beiden Klassen Attribute und Methoden an:

Klasse Wörterbuch

Objekte dieser Klasse besitzen folgende Attribute:

- Die Attribute `sprache1` und `sprache2` beschreiben die beiden Sprachen, die in dem jeweiligen Wörterbuch miteinander verknüpft sind (z.B. Deutsch und Englisch). Bei einem realen Wörterbuch findet man diese Angaben üblicherweise auf dem Buchumschlag.
- Das Attribut `vokabeln` enthält eine Abbildung von Wörtern in einer Sprache (`sprache1`) auf Wörter einer anderen Sprache (`sprache2`). Diese stellt das alphabetisch sortierte Wörterverzeichnis im Innern eines realen Wörterbuchs aus Papier dar.
- Das Attribut `pfad` enthält eine Beschreibung des Ortes, an dem die Voka-

beln auf einem Datenträger gespeichert sind, in Form einer Pfadbezeichnung. In unserem Bild entspricht das der Position im Bücherschrank.

Wir legen fest, dass Objekte der Klasse `Wörterbuch` folgende Methoden beherrschen sollen:

- Die Methode `übersetze()` liefert zu einem Wort aus der ersten Sprache die Übersetzung in Form einer Liste von Wörtern aus der zweiten Sprache. Das entspricht der Suche nach einem Wort in einem richtigen Wörterbuch. In unserem Modell jedoch sucht das Wörterbuch selbst und nicht der Leser.
- Mit Hilfe von Methode `neu()` kann in das Attribut `vokabeln` ein neues Wort nebst Übersetzung in das Wörterbuch eingetragen werden.
- Die Methode `speichere()` sorgt für das Speichern des momentanen Zustands des Vokabulars (Attribut `vokabeln`). Das entspricht in etwa dem Zurückstellen eines realen Wörterbuchs in das Regal.

Klasse Benutzungsoberfläche

Objekte dieser Klasse benötigen ein Attribut und eine Methode:

- Das Attribut `wb` enthält den Namen eines Objektes der Klasse `Wörterbuch`. Dies ist ein Objektattribut und kann individuell belegt werden. Das heißt, verschiedene Objekte der Klasse `Benutzungsoberfläche` können unterschiedliche Wörterbücher verwenden.
- Die Methode `run()` startet die Benutzungsoberfläche und überführt sie in einen Zustand, in dem sie auf Benutzereingaben wartet.

Assoziationen zwischen Klassen

Die Klassen `Benutzungsoberfläche` und `Wörterbuch` sind nicht losgelöst voneinander, sondern zwischen ihnen besteht eine Beziehung, eine Assoziation. Sie kommt dadurch zustande, dass durch das Attribut `wb` *jedem* Benutzungsoberflächenobjekt ein Objekt der Klasse `Wörterbuch` zugeordnet ist. Man kann sagen, jedes Objekt der Klasse `Benutzungsoberfläche` ***benutzt*** ein Objekt der Klasse `Wörterbuch`. Das Prädikat `benutzt` kann man als Name für diese Assoziation zwischen den beiden Klassen betrachten. Abbildung 12.5 zeigt ein UML-Klassendiagramm für unser Modell.

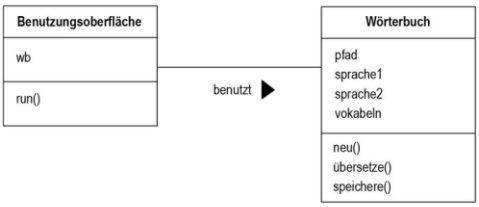

Abb. 12.5: UML-Klassendiagramm zur Modellierung eines interaktiven Wörterbuchs (OOA)

12.2.2 OOD: Entwurf einer Klassenstruktur zur Implementierung in Python

Das Ergebnis der objektorientierten Analyse ist noch sehr abstrakt. Von Einzelheiten der technischen Realisierung hat man bewusst abgesehen. In der nun folgenden Entwurfsphase (OOD) wird die Implementierung in einer Programmiersprache vorbereitet.

Bei einer industriellen Software-Entwicklung würden nun Fachexperten und Anwender das Team verlassen und das Feld den Informatikern überlassen. Wir entscheiden uns, das Programm als Stand-alone-Skript (nur eine Skriptdatei), das auf einem Einzelrechner laufen soll, zu implementieren. Die Programmiersprache soll natürlich Python sein.

Beachten Sie: Das OOA-Modell, das durch das Klassendiagramm in Abbildung 12.5 wiedergegeben wird, ist so allgemein, dass es auch auf ganz andere Weise realisiert werden könnte:

- Anstelle von Python könnte man auch eine andere objektorientierte Programmiersprache wie C++ oder Java verwenden.

- Anstelle eines Stand-alone-Programms könnte man auch ein Client-Server-System programmieren. Das Wörterbuch-Objekt würde sich dann auf einem zentralen Server und die Benutzungsoberfläche in einem separaten Client-Skript auf einem anderen Rechner befinden. Beide Systemteile könnten über das Internet kommunizieren.

Wir bleiben bei der einfachen Lösung als Stand-alone-Skript und verfeinern nun das OOA-Modell zu einem OOD-Modell. Das ist unsere Vorgehensweise:

- Wir ergänzen die vorhandenen Klassensymbole um weitere Attribute und Methoden.
- Es wird festgelegt, welche Attribute Klassenattribute sein sollen.
- Die Sichtbarkeit der Methoden und Attribute wird spezifiziert. Für UML-Klassendiagramme gibt es dafür eine spezielle Notation. Den Attribut- bzw. Methodennamen wird ein `+`, `-` oder `#` vorangestellt. Dabei bedeutet
  - `+` öffentlich (public): sichtbar für alle anderen Klassen
  - `#` geschützt (protected): sichtbar innerhalb der Klasse und aller Unterklassen (das entspricht ungefähr »schwach privat« in der Python-Redeweise)
  - `-` privat (private): sichtbar nur innerhalb der eigenen Klasse

  Für Python-Skripte brauchen wir diese Schreibweise nicht. Wir definieren die Sichtbarkeit der Attribute und Methoden durch führende Unterstriche in ihren Namen.
- Für Methoden werden Vor- und Nachbedingungen präzisiert. Diese können später als Kommentare oder Docstrings in den Programmtext übernommen werden.

Abbildung 12.6 zeigt ein Klassendiagramm der OOD-Phase zu unserem Wörterbuch-Projekt.

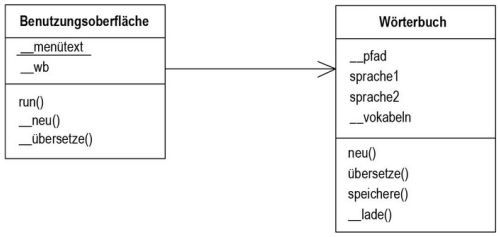

Abb. 12.6: UML-Klassendiagramm zur Modellierung eines interaktiven Wörterbuchs (OOD)

Wir haben folgende Verfeinerungen und Anpassungen an die technischen Rahmenbedingungen vorgenommen:

Klasse Wörterbuch

Die Attribute `__pfad` und `__vokabeln` sind privat. Auf sie darf (von außen) nicht direkt zugegriffen werden. Das Vokabular kann nur über die öffentlichen Methoden `neu()` und `übersetze()` erreicht werden.

Die Attribute `sprache1` und `sprache2` sind öffentlich. (Wie bereits erwähnt verzichten wir in den Beispielskripten auf private Attribute, um den Programmtext möglichst knapp zu halten.) Für die Benutzungsoberfläche ist es manchmal wichtig, die Sprachen zu kennen, auf die das Wörterbuch zugeschnitten ist. Die Methoden `neu()`, `übersetze()` und `speichere()` sind öffentlich. Dagegen ist `__lade()` privat. Diese Methode wird nur intern bei der Initialisierung eines `Wörterbuch`-Objektes – also innerhalb der Klasse – aufgerufen.

Klasse Benutzungsoberfläche

Es wurde ein neues privates Klassenattribut `menütext` eingefügt. Sein Name ist

im UML-Klassendiagramm unterstrichen. Es enthält als Text ein Menü, d.h. eine Beschreibung von Funktionen, von denen der Benutzer durch Eingabe eines Buchstabens eine auswählen kann. Dieses Attribut ist ein Klassenattribut. Das heißt also, alle Objekte der Klasse `Benutzungsoberfläche` besitzen den gleichen Menütext. Das ist deshalb sinnvoll, weil die Funktionalität, die in der Methode `run()` definiert wird, auf das Menü abgestimmt ist. Wenn zum Beispiel bei Eingabe des Buchstabens `E` das Programm beendet werden soll, so muss dies auch genau so im Menü dargestellt werden.

Das Attribut `__wb` enthält einen Namen für ein `Wörterbuch`-Objekt und bleibt privat, da ein Zugriff von außen nicht erforderlich ist.

Die Methode `run()` ist natürlich öffentlich, weil die Benutzungsoberfläche von außen gestartet werden muss. Hinzugefügt haben wir zwei private Methoden `__neu()` und `__übersetze()`, die die Dialoge mit dem Benutzer bei Eingabe einer neuen Vokabel oder der Suche nach einer Übersetzung abwickeln.

Assoziation

In UML-Klassendiagrammen, die einen Entwurf darstellen, symbolisiert man die Assoziationen häufig durch Pfeile, um die Richtung der Beziehung darzustellen. Man spricht auch von Navigation. So ist ein Objekt der Klasse `Wörterbuch` nur über ein Objekt der Klasse `Benutzungsoberfläche` zu erreichen (über das Attribut `__wb`), aber nicht umgekehrt. Bidirektionale Beziehungen werden durch Doppelpfeile dargestellt.

12.2.3 OOP: Implementierung der Klassenstruktur

Das folgende Python-Skript implementiert das Modell, das wir im letzten Abschnitt entworfen haben. Es besteht aus zwei Klassendefinitionen und einem Hauptprogramm, in dem Objekte der beiden Klassen verwendet werden. Sämtliche Namen für Klassen, Methoden und Attribute, die im OOD-Modell festgelegt worden sind, werden genauso übernommen. Übrigens sieht Python das gesamte Skript ebenfalls als Objekt an – nämlich als eine Instanz der Klasse `__main__`.

```
import pickle
```



```
class Wörterbuch:
    def __init__(self, pfad, sprache1, sprache2):
        self.__pfad = pfad                                              #1
        try:
            datei = open(self.__pfad, 'rb')
            self.__vokabeln = pickle.load(datei)
            datei.close()
        except:
            self.__vokabeln={}
        self.sprache1 = sprache1
        self.sprache2 = sprache2

    def speichere(self):                                                #2
        datei = open(self.__pfad, 'wb')
        pickle.dump(self.__vokabeln, datei)
        datei.close()

    def neu(self, wort1, wort2):                                        #3
        if wort1 not in self.__vokabeln.keys():
            self.__vokabeln[wort1] = [wort2]
        else:
            self.__vokabeln[wort1] += [wort2]

    def übersetze(self, wort):                                          #4
        if wort in self.__vokabeln.keys():
            return self.__vokabeln[wort]
        else:
            return ['Wort unbekannt']

class Benutzungsoberfläche:                                             #5
    __menütext = '''Bitte wählen Sie!
    (N)eues Wort eingeben
    (Ü)bersetzung
    (E)nde
    '''

    def __init__ (self, wörterbuch):                                    #6
        self.__wb = wörterbuch

    def run(self):                                                      #7
        wahl = ' '
        while wahl not in 'Ee':
            print(self.__menütext)
            wahl = input('Ihre Wahl: ')
            if wahl in 'nN':
                self.__neu()
            elif wahl in 'uU':
                self.__übersetze()
        print('Danke für die Verwendung des Wörterbuchs!')
```



```
        self.__wb.speichere()

    def __neu(self):                                                    #8
        wort = input(self.__wb.sprache1 + ": ")
        übersetzung = input(self.__wb.sprache2 + ": ")
        while übersetzung != '':
            self.__wb.neu(wort, übersetzung)
            übersetzung = input(self.__wb.sprache2 + ': ')
        print()

    def __übersetze(self):                                              #9
        wort1 = input(self.__wb.sprache1 + ': ')
        print(self.__wb.sprache2 + ': ', end=' ')
        for übersetzung in self.__wb.übersetze(wort1):
            print (übersetzung, end='')
        print()
        print()

# Hauptprogramm
if __name__ == '__main__':
    pfad = 'englisch.wb'     # Pfad der Wörterbuch-Datei
    w = Wörterbuch(pfad, 'Deutsch','Englisch')
    menü = Benutzungsoberfläche(w)
    menü.run()
```

Erläuterung:

#1: `__pfad` und `__vokabeln` sindstarkprivate Objektattribute. Vor allem das (unter Umständen sehr umfangreiche) Vokabular des Wörterbuchs, das auf einem Datenträger gespeichert wird, ist besonders schützenswert. Das Attribut `__vokabeln` ist ein Dictionary. Es enthält Einträge der Form `wort:[wort1,wort2, ...]`. Das heißt, jedem Wort aus Sprache 1 wird eine Liste von Übersetzungen (Wörter aus Sprache 2) zugeordnet. Das Dictionary wird bei der Initialisierung geladen. Falls noch keine Datei unter dem angegebenen Pfad `self.__pfad` existiert, wird ein leeres Dictionary `{}` angelegt. Voraussetzung ist allerdings, dass das Verzeichnis tatsächlich existiert. Sonst gibt es später beim Abspeichern (Methode `speichere()`) einen Laufzeitfehler.

Die Attribute `sprache1` und `sprache2` sind öffentlich. Auf sie kann z.B. ein Objekt der Klasse `Benutzungsoberfläche` zugreifen. An dieser Stelle wird gegen das Geheimnisprinzip verstoßen, um den Programmtext kurz zu halten. Anderenfalls müssten für diese beiden Attribute Zugriffsroutinen `getSprache1()` und `getSprache2()` geschrieben werden.

#2: Abspeichern des Dictionarys `self.__vokabeln` unter dem Pfad `self.__pfad`.

#3: Wenn das Wörterbuch noch keinen Eintrag zum Wort `wort1` enthält, wird ein neues Paar `wort1:[wort2]` angelegt. Ansonsten wird die Liste von Übersetzungen zum Schlüssel `wort1` um `wort2` verlängert.

#4: Diese öffentliche Methode liefert zum übergebenen Parameter `wort` eine Liste der Übersetzungen. Falls `wort` in der Schlüsselliste des Dictionarys nicht vorkommt, wird eine entsprechende Fehlermeldung zurückgegeben.

#5: Die lange Zeichenkette `menütext` ist ein Klassenattribut (Klassenvariable).

#6: Das einzige Objektattribut `self.wb` bezeichnet ein Objekt der Klasse `Wörterbuch`.

#7: Die (einzige) öffentliche Methode `run()` verarbeitet in einer Schleife Eingaben des Benutzers. Wenn z.B. der Buchstabe `n` oder `N` eingegeben wird, wird die private Methode `__neu()` aufgerufen. Die Schleife wird erst verlassen, wenn `e` oder `E` eingegeben worden ist. Dann wird der aktuelle Zustand des Dictionarys `self.__vokabeln` gespeichert.

#8: Private Methode zur Eingabe einer neuen Vokabel. Hier wird (unter Verletzung des Geheimnisprinzips) auf die Attribute `wb.sprache1` und `wb.sprache2` zugegriffen. Zuerst wird ein Wort aus der ersten Sprache abgefragt und dann so lange Übersetzungen – also Wörter aus der zweiten Sprache – abgefragt, bis der Benutzer kein weiteres Wort eingibt, sondern nur auf `Enter` drückt. Dabei wird jedes Mal die Methode `wb.neu()` aufgerufen, um die neue Übersetzung in das Vokabular des Wörterbuchs einzutragen.

#9: Diese private Methode fragt zunächst nach einem Wort in der ersten Sprache, holt dann aus dem Wörterbuch eine Liste von Übersetzungen und gibt dann die in der Liste enthaltenen Wörter aus der zweiten Sprache aus.

Beispiel für einen Programmlauf:

```
Bitte wählen Sie!
    (N)eues Wort eingeben
    (Ü)bersetzung
    (E)nde

Ihre Wahl: n
```

```
Deutsch: Hilfe
Englisch: aid
Englisch: help
Englisch:

Bitte wählen Sie!
    (N)eues Wort eingeben
    (Ü)bersetzung
    (E)nde

Ihre Wahl: u
Deutsch: Hilfe
Englisch:  aid help

Bitte wählen Sie!
    (N)eues Wort eingeben
    (Ü)bersetzung
    (E)nde

Ihre Wahl: e
Danke für die Verwendung des Wörterbuchs!
```

12.3 Assoziationen zwischen Klassen

12.3.1 Reflexive Assoziationen

Eine reflexive Assoziation besteht zwischen Objekten derselben Klasse. Ein Beispiel hierfür ist ein Stammbaum. Eine Person modellieren wir durch eine Klasse namens `Stammbaum` mit folgenden Attributen:

- Das Attribut `name` enthält eine Zeichenkette mit dem Namen der Person.
- Die Attribute `vater` und `mutter` enthalten entweder Objekte der Klasse `Stammbaum`, die Vater und Mutter repräsentieren, oder sie enthalten das Objekt `None`, falls z.B. Vater bzw. Mutter nicht bekannt ist oder nicht angegeben werden soll.

Ein Objekt der Klasse `Stammbaum` ist also mit null bis höchstens zwei Objekten derselben Klasse assoziiert. Diese Zahlenangaben bezeichnet man als Kardinalitäten der Beziehung. Abbildung 12.7 zeigt das zugehörige Klassendiagramm. Die Zahlen an der Linie stellen die Kardinalitäten der Beziehung dar.

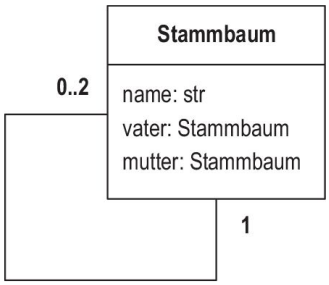

Abb. 12.7: Klassendiagramm mit reflexiver Assoziation

Im folgenden Python-Skript wird zunächst die Klasse `Stammbaum` definiert. Dann erzeugen wir – unten bei den Großeltern beginnend – eine Objektstruktur, die den Stammbaum von Jenny über insgesamt drei Generationen wiedergibt. Jedes Objekt der Klasse `Stammbaum` enthält maximal zwei Namen für Objekte der gleichen Klasse als Attribut.

Skript:

```
class Stammbaum:
    def __init__(self, name, vater, mutter):
        self.name = name                          # String
        self.vater = vater                                          #1
        self.mutter = mutter

    def __str__(self):                                              #2
        s = ''
        if self.name:
            s += self.name + '\n'
            if self.mutter:                                         #3
                s += 'Mutter von ' + self.name + \
                     ': ' + str(self.mutter)
            if self.vater:
```



```
                s += 'Vater von '+ self.name + \
                     ': ' + str(self.vater)
        return s

sarah = Stammbaum('Sarah', None, None)                              #4
willy = Stammbaum('Willy', None, None)
marianne = Stammbaum('Marianne', None, None)
anton = Stammbaum('Anton', None, None)
marlene = Stammbaum('Marlene', willy, sarah)
werner = Stammbaum('Werner', anton, marianne)
jenny = Stammbaum('Jenny', werner, marlene)
print(jenny)                                                        #5
```

Programmlauf:

```
Jenny
Mutter von Jenny: Marlene
Mutter von Marlene: Sarah
Vater von Marlene: Willy
Vater von Jenny: Werner
Mutter von Werner: Marianne
Vater von Werner: Anton
```

Erläuterung:

#1: Die Attribute `self.mutter` und `self.vater` bezeichnen Objekte der Klasse `Stammbaum` oder `None`, während `self.name` einen String enthält.

#2: In dieser Methode wird eine String-Repräsentation des `Stammbaum`-Objektes generiert. Zum Beispiel bei einer `print()`-Anweisung (**#5**) wird diese Methode aufgerufen.

#3: Es wird ein Text aus mehreren Zeilen berechnet, der folgendermaßen aufgebaut ist:

- Name der Person
- String-Repräsentation des Stammbaums der Mutter, sofern die Mutter angegeben ist, d.h. das Attribut `self.mutter` nicht `None` ist. Hier findet also ein rekursiver Aufruf der `__str__()`-Methode statt. Der Stammbaum der Mutter beginnt wieder mit einem Namen, nämlich dem Namen der Mutter.

- String-Repräsentation des Stammbaums des Vaters, sofern der Vater angegeben ist.

#4: Zuerst werden Objekte für die Großeltern von Jenny instanziiert. Deren Eltern sind nicht bekannt, deshalb wird `None` als zweites und drittes Argument übergeben. Die Großeltern-Objekte werden zur Erzeugung der Eltern-Objekte benötigt und diese tauchen zum Schluss bei der Erzeugung des Objektes `jenny` als Argumente auf. Abbildung 12.8 zeigt die resultierende Objektstruktur.

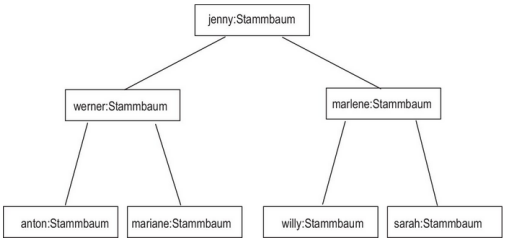

Abb. 12.8: UML-Objektdiagramm von Jennys Stammbaum

12.3.2 Aggregation

Ein spezieller und gleichzeitig häufig vorkommender Typ einer Assoziation zwischen Objekten zweier Klassen ist die *Aggregation*. Es handelt sich dabei um die Beziehung zwischen einem Ganzen (Aggregat) und Teilen, aus denen es zusammengesetzt ist. Manchmal spricht man auch von einer »hat«-Beziehung oder »besteht aus«-Beziehung. Fast alle Objekte des Alltags sind Aggregate. Ein Auto z.B. besteht aus vier Objekten der Klasse Rad, einem Motor, einer Karosserie und anderen Teilen. In UML-Klassendiagrammen stellt man eine Aggregation durch eine Linie mit einer ungefüllten Raute an der Aggregat-Klasse dar (Abbildung 12.9).

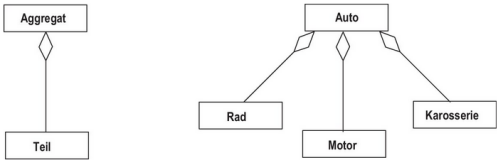

Abb. 12.9: UML-Klassendiagramme mit Aggregationen

12.4 Beispiel: Management eines Musicals

Als komplexes Beispiel einer objektorientierten Modellierung entwickeln wir ein System zur Verwaltung des Kartenverkaufs für ein Musical.

12.4.1 OOA

In der OOA gehen wir von folgenden vereinfachenden Annahmen aus:

- Das Musical wird nur in *einem* Saal gespielt.
- Es gibt einen einheitlichen Eintrittspreis.
- Das System wird von zwei Akteuren benutzt, einem Manager, der neue Vorstellungen einrichten kann, und einem Kartenverkäufer, der Eintrittskarten buchen kann.

Die Aufgaben, die das Programm lösen soll (*use cases*), werden durch das Geschäftsprozessdiagramm in Abbildung 12.10 beschrieben.

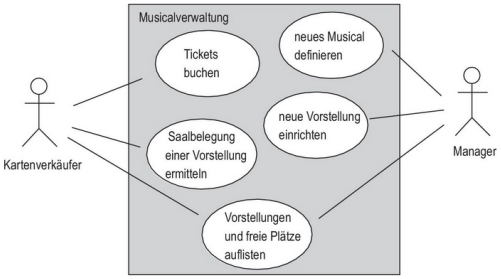

Abb. 12.10: Geschäftsprozessdiagramm für eine Musicalverwaltung

Kern des Systems ist ein Modell eines Musicals. Da sind natürlich viele Varianten denkbar. Abbildung 12.11 gibt eine mögliche Lösung als UML-Klassendiagramm wieder. Das Modell orientiert sich an den use cases. Alles wird so einfach wie möglich gehalten, aber es wird darauf geachtet, dass die geforderten Geschäftsprozesse mit diesem Modell realisiert werden können.

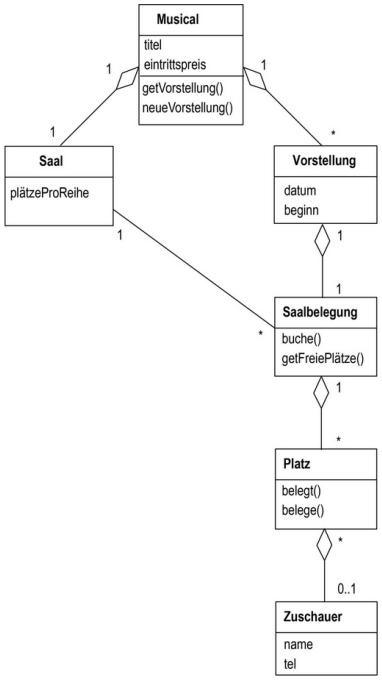

Abb. 12.11: UML-Klassendiagramm für das Modell eines Musicals (OOA)

Einem Objekt der Klasse `Musical` sind genau ein `Saal`-Objekt und beliebig viele Objekte der Klasse `Vorstellung` zugeordnet (Kardinalität = `*`). Eine Vorstellung wiederum enthält genau eine Saalbelegung (Kardinalität = 1).

In der Saalbelegung sind – angepasst an die Beschaffenheit des Saals – den Sesseln in den verschiedenen Sitzreihen Objekte der Klasse `Platz` zugeordnet. Die Klasse `Saalbelegung` besitzt also eine Assoziation zur Klasse `Saal`. Bei der Initialisierung eines neuen Objektes der Klasse `Saalbelegung` muss der Aufbau des Saales aus Sitzreihen mit jeweils einer bestimmten Anzahl von Plätzen ins Kalkül gezogen werden. Andererseits besteht eine Saalbelegung aus im Prinzip beliebig vielen Platz-Objekten (Aggregation mit den Kardinalitäten 1 und `*`).

Ein Platz kann von höchstens einem Zuschauer belegt werden (Kardinalität 0..1). Manchmal bleiben Plätze auch leer. Andererseits kommt es häufig vor, dass ein Zuschauer mehrere Plätze gebucht hat. Deshalb trägt die Aggregation an der Seite der Klasse `Platz` die Kardinalität `*`.

Die Benutzungsoberflächen für die beiden Akteure Manager und Kartenverkäufer werden durch eigene Klassen realisiert, für die wir keine UML-Diagramme angegeben haben.

12.4.2 OOD

Es ist sinnvoll, das System auf drei Skripte zu verteilen:

- Ein Skript mit dem Dateinamen `musical.py` (Modulname `musical`) enthält die Klassen des OO-Modells des Musicals gemäß Abbildung 12.11. Dieses Skript ist kein lauffähiges Programm, sondern ein Modul, das importiert werden kann. Um den Import besonders einfach zu machen, speichern wir alle Programmdateien im selben Verzeichnis ab.
- Ein Skript mit dem Dateinamen `musicalmanager.py` importiert die Klassen aus dem Modul `musical` und enthält eine Klasse `Manager`, in der die Benutzungsoberfläche für den Akteur Manager aus dem Geschäftsprozessdiagramm definiert ist. Das Skript ist ein lauffähiges Programm, das heißt, es wird ein Objekt der Klasse `Manager` instanziiert und »gestartet«.

- Ein ebenfalls lauffähiges Skript mit dem Dateinamen `musicaltickets.py` importiert ebenfalls die Klassen aus dem Modul `musical` und enthält eine Klasse `Kartenverkauf`, mit der Benutzungsoberfläche für den Akteur Kartenverkäufer.

12.4.3 OOP

Skript zum OO-Modell eines Musicals

```
# musical.py  - Modell eines Musicals
import pickle

class Musical:
    def __init__(self, titel, eintrittspreis, saal):
        self.titel = titel
        self.eintrittspreis = eintrittspreis
        self.saal = saal
        self.vorstellungen = []  # Liste von Vorstellungen

    def getVorstellung(self, datum):
        '''Rückgabe eines Vorstellungsobjektes mit
        passendem Datum, falls vorhanden, sonst None'''
        for vorstellung in self.vorstellungen:
          if vorstellung.datum == datum: return vorstellung
        # Nach dem return bricht die Ausführung der Funktion ab

    def neueVorstellung(self, vorstellung):
        '''Objekt vorstellung wird in Liste eingefügt'''
        self.vorstellungen += [vorstellung]

    def __str__(self):                                              #1
        beschreibung = '\n' + self.titel + '\n' + \
                       len(self.titel)*'=' + '\n'
        for vorstellung in self.vorstellungen:
            beschreibung += str(vorstellung) + '\n'
        return beschreibung

class Vorstellung:
    def __init__(self, datum, beginn, saal ):
        self.datum = datum
        self.beginn = beginn
        self.saalbelegung = Saalbelegung(saal)
        self.saal = saal

    def __str__(self):                                              #1
```



```
        beschreibung = self.datum + '\n' + \
        str(self.saalbelegung.getFreiePlätze()) + \
        ' freie Plätze\n'
        return beschreibung

class Saalbelegung:
    '''pflegt Liste von Listen mit Platz-Objekten'''
    def __init__(self, saal):
        self.belegung = []
        self.saal = saal
        for i in range(len(saal.plätzeProReihe)):                   #2
            reihe = []
            for j in range(saal.plätzeProReihe[i]):
                platz = Platz()
                reihe += [platz]
            self.belegung += [reihe]

    def buche(self, reihe, platz, zuschauer):                       #3
        '''weist dem Platz platz in Reihe reihe einen
        Zuschauer zu'''
        if not self.belegung[reihe][platz].belegt():
           self.belegung[reihe][platz].belege(zuschauer)
           return 'Platz gebucht'
        else: return 'Platz schon belegt'

    def getFreiePlätze(self):
        '''liefert Anzahl der freien Plätze'''
        frei = 0
        for reihe in self.belegung:
            for platz in reihe:
                if not platz.belegt(): frei += 1
        return frei

    def __str__(self):                                              #1
        beschreibung = 'Saalbelegung\n'
        beschreibung += '   Platz: 1  2  3  4  5  6  7  8  9  10 '
        beschreibung += '11 12 13 14\n'
        nr = 1                              # Reihennummer
        for reihe in self.belegung:
            beschreibung += 'Reihe ' + format(nr, '2d') + ': '
            for platz in reihe:
                beschreibung += str(platz)
            nr += 1
            beschreibung += '\n'         # neue Zeile
        return beschreibung

class Zuschauer:
    def __init__(self, name, tel):
        self.name, self.tel = name, tel
```



```
class Platz:
    def __init__(self):
        self.zuschauer = None

    def belegt(self):
        if self.zuschauer: return True
        else: return False

    def belege(self, zuschauer):
        self.zuschauer = zuschauer

    def __str__(self):                                              #1
        if self.belegt():
            return self.zuschauer.name[:2] + ' '
            # vom Zuschauernamen nur die ersten beiden Zeichen
        else:
            return '-- '       # freier Platz

class Saal:
    def __init__(self, liste):
        self.plätzeProReihe = liste
```

Erläuterung:

#1: Für einige Klassen wird die Standardfunktion `str()` überladen. Die Methode `__str__()` gibt eine druckbare Beschreibung des jeweiligen Objektes zurück. So liefert z.B. die Methode `Platz.__str__()` eine Kurzbeschreibung einer Platzbelegung aus drei Zeichen, und zwar entweder den String `'-- '`, falls der Platz noch frei ist, oder die Anfangsbuchstaben des Zuschauers, der diesen Platz gebucht hat. Dieser String kann für die Darstellung einer kompletten Übersicht der Platzbelegung eines Saals verwendet werden.

#2: Das Attribut `belegung` ist eine Liste von Sitzreihen des Saals. Jede Reihe ist wiederum eine Liste von Plätzen. Bei der Initialisierung eines Objektes der Klasse `Saalbelegung` »erkundigt sich« das Objekt beim zugehörigen Saal-Objekt, wie viele Reihen es gibt (`len(saal.plätzeProReihe)`) und wie viel Plätze jede Reihe enthält. Das Saal-Attribut `plätzeProReihe` ist eine Liste von Zahlen, wobei `plätzeProReihe[i]` die Anzahl der Plätze in Reihe `i` ist.

#3: Zu beachten ist hier, dass die Nummerierung realer Reihen und Plätze in einem Vorstellungsraum bei eins beginnt. Intern verwenden wir jedoch Listen, und die Indexe für Listenelemente beginnen bei null. Das heißt, Platz `1` in Reihe `1` wird durch `self.belegung[0][0]` repräsentiert. Deshalb muss von den als Argument übergebenen Reihen- und Platznummern eins abgezogen werden.

Skript für die Benutzungsoberfläche Management

```
# musicalmanager.py  - Verwaltung eines Musicals
import pickle
from musical import *

class Manager:
    __menütext = '''
    Musical-Manager
    ------------------
    (n)eue Vorstellung
    (Ü)berblick Vorstellungen
    (E)nde
     '''

    def __init__(self, datei):                                      #1
        self.__datei = datei
        self.__lade_musical()
        self.__run()

    def __run(self):                                                #2
        ''' Menü und Verarbeitung von Auswahlen'''
        print(self.__menütext)
        wahl = '--'
        while wahl not in ['e', 'E']:
          wahl = input('Auswahl: ')
          if wahl in ['n', 'N']:
            self.__neueVorstellung()
          elif wahl in ['Ü', 'ü']:
            print(str(self.__musical))
          print(self.__menütext)
        print("Danke für die Benutzung von Musical-Manager")
        self.__speichern()

    def __neueVorstellung(self):
        '''Dialog zum Einrichten einer neuen Vorstellung'''
        datum = input('Termin: ')
        beginn = input('Beginn der Vorstellung: ')
        vorstellung = Vorstellung(datum, beginn, self.__musical.saal)
        self.__musical.neueVorstellung(vorstellung)

    def __neuesMusical(self):                                       #3
        '''Dialog zur Definition eines neuen Musicals'''
```

```
        titel = input('Titel: ')
        eintrittspreis = float(input('Eintrittspreis: '))
        anzahl_reihen = int(input('Anzahl Sitzreihen: '))
        liste = []
        for i in range(anzahl_reihen):
          sitze = int(input('Sitze in Reihe ' + str(i) + ': '))
          liste.append(sitze)
        saal = Saal(liste)
        self.__musical = Musical(titel, eintrittspreis, saal)

    def __lade_musical(self):
        '''Musical-Objekt wird geladen, falls Datei vorhanden
        sonst muss neues Musical definiert werden'''
        try:
            f = open(self.__datei, 'rb')  # lesen im Binärmodus
            self.__musical = pickle.load(f)
            f.close()
            print('\n              W I L L K O M M E N')
            print('beim Management-System für das Musical',
                  self.__musical.titel)
        except:
            print('Kein Musical gespeichert.')
            print('Richten Sie neues Musical ein.')
            self.__neuesMusical()

    def __speichern(self):
        '''Musical-Objekt speichern'''
        f = open(self.__datei, 'wb') # schreiben im Binärmodus
        pickle.dump(self.__musical, f)
        f.close()

m = Manager('daten/hairspray.txt')
```

Erläuterung:

#1: Bei der Initialisierung eines Manager-Objektes wird dem Musical eine (feste) Datei zugeordnet, in der die Daten des Musicals gespeichert werden. Zunächst wird versucht, ein vorhandenes Musical-Objekt aus dieser Datei zu laden. Wenn die Datei nicht existiert, wird durch Aufruf der Methode `__neuesMusical()` ein Dialog zur Einrichtung eines neuen Musicals eingeleitet. Schließlich wird die Methode `__run()` aufgerufen und der Dialog gestartet. Hinweis: Wenn nach erstmaliger Benutzung des Programms ein neues Musical eingerichtet werden soll, muss zuvor die Datei mit dem alten Musical-Objekt (im Beispiel `daten/hairspray.txt`) gelöscht werden.

#2: In der Schleife der `__run()`-Methode wird immer wieder das Menü auf den Bildschirm gebracht, eine Auswahl abgewartet und dann die ausgewählte Funktion ausgeführt. Bei Eingabe von `'e'` oder `'E'` wird die Schleife verlassen, das Musical-Objekt in seinem neuen Zustand abgespeichert und das Programm beendet.

#3: Im Dialog zum Einrichten eines neuen Musicals werden zunächst Titel und Eintrittspreis festgelegt. Dann wird Reihe für Reihe die Bestuhlung des Saals in eingegeben.

Beispieldialog Management:

```
              W I L L K O M M E N
beim Management-System für das Musical Hairspray

    Musical-Manager
    ---------------
    (n)eue Vorstellung
    (Ü)berblick Vorstellungen
    (E)nde

Auswahl: n
Termin: 1.10.2023
Beginn der Vorstellung: 20.00 Uhr

    Musical-Manager
    ---------------
    (n)eue Vorstellung
    (Ü)berblick Vorstellungen
    (E)nde

Auswahl: u

Hairspray
=========
1.10.2023
58 freie Plätze

2.10.2023
58 freie Plätze

    Musical-Manager
    ------------------
    (n)eue Vorstellung
    (Ü)berblick Vorstellungen
```



```
    (E)nde

Auswahl: e
Danke für die Benutzung von Musical-Manager
```

Skript für den Kartenverkauf

```
# musicaltickets.py - Kartenverkauf
import pickle
from musical import *

class Kartenverkauf:
    __menütext = '''
    Musical-Ticketservice
    ---------------------
    (B)uchung
    (Ü)berblick Vorstellungen
    (E)nde
    '''

    def __init__(self, datei):
        self.__datei = datei
        self.__lade_musical()
        self.__run()

    def __lade_musical(self):
        ''' versucht, aus Datei Musical zu laden'''
        try:
            f = open(self.__datei, 'rb')
            self.__musical = pickle.load(f)
            f.close()
            print('\n              W I L L K O M M E N')
            print('beim Buchungssystem für das Musical',
                  self.__musical.titel)
        except:
            print('Kein Musical gespeichert. ')

    def __run(self):
        ''' Menü und Verarbeitung von Auswahlen'''
        print(self.__menütext)
        wahl = '-'
        while wahl not in 'eE':
            wahl = input('Auswahl: ')
            if wahl in 'bB': self.__buchen()
            elif wahl in 'üÜ': print(self.__musical)
            print(self.__menütext)
        print('Danke für die Benutzung von Musical-Ticketservice')
```



```
        self.__speichern()

    def __buchen(self):                                          #1
        '''Dialog zum Buchen mehrerer Plätze '''
        datum = input('Datum der Vorstellung: ')
        vorstellung = self.__musical.getVorstellung(datum)
        if not vorstellung:                                      #2
            print('An diesem Tag gibt es keine Vorstellung')
        else:
          print(vorstellung.saalbelegung)
          name = input('Name: ')
          tel = input('Telefonnummer: ')
          zuschauer = Zuschauer(name, tel)
          reihe = 'x'                                            #3
          while reihe != '':
            reihe = input('Reihe: ')
            if reihe != '':                                      #4
                platz = input('Platz: ')
                print(vorstellung.saalbelegung.buche(
                  int(reihe)-1, int(platz)-1, zuschauer))

    def __speichern(self):
        f = open(self.__datei, 'wb')
        pickle.dump(self.__musical, f)
        f.close()

kasse1 = Kartenverkauf('daten/hairspray.txt')
```

Erläuterung:

#1: Der aufwändigste Dialog betrifft das Buchen von Plätzen einer Vorstellung. Nach Eingabe eines Datums wird zunächst ein Objekt der Klasse `Vorstellung` gesucht, dessen Attribut `datum` den gleichen Wert wie das eingegebene Datum (ein String) besitzt.

#2: Falls der Aufruf `self.__musical.getVorstellung(datum)` den Wert `None` zurückgibt, erfolgt eine Fehlermeldung.

#3: Ansonsten werden zunächst Name und Telefonnummer abgefragt und in der `while`-Schleife so lange neue Plätze gebucht, bis der Benutzer bei der Frage nach der gewünschten Reihe nichts mehr eingibt, sondern nur Enter drückt. Im Einzelnen funktioniert die Schleife so: Zu Beginn wird die Variable `reihe` mit dem willkürlichen Wert `'x'` belegt, damit die Schleifenbedingung (folgende Zeile) erfüllt ist und die Schleife ausgeführt wird. Im Schleifeninneren bittet das Pro-

gramm um Eingabe einer Reihennummer. Wenn der eingegebene Wert ein leerer String '' ist (es wurde nur Enter gedrückt), wird die Schleife sofort abgebrochen (#4). Ansonsten fragt das System nach einer Platznummer und bucht einen Platz durch Aufruf von

```
vorstellung.saalbelegung.buche(int(reihe)-1, int(platz)-1, zuschauer)
```

Weil diese Methode eine Meldung zurückgibt, die die Buchung entweder bestätigt oder nicht bestätigt, kann der Methodenaufruf in einen `print()`-Aufruf eingebaut werden, damit die Meldung direkt auf dem Bildschirm ausgegeben wird. In der Parameterliste werden die Strings `reihe` und `platz` in ganze Zahlen umgewandelt.

Dialogbeispiel für den Kartenverkauf:

```
            W I L L K O M M E N
beim Buchungssystem für das Musical Hairspray

    Musical-Ticketservice
    ---------------------
    (B)uchung
    (Ü)berblick Vorstellungen
    (E)nde

Auswahl: b
Datum der Vorstellung: 1.10.2023
Saalbelegung

   Platz: 1  2  3  4  5  6  7  8  9  10 11 12 13 14
Reihe  1: -- -- -- -- -- -- -- -- -- --
Reihe  2: -- -- -- -- -- -- -- -- -- --
Reihe  3: -- -- -- -- -- -- -- -- -- -- --
Reihe  4: -- -- -- -- -- -- -- -- -- -- --
Reihe  5: -- -- -- -- -- -- -- -- -- -- -- -- --

Name: Meyer
Telefonnummer: 0234 68867
Reihe: 1
Platz: 6
Platz gebucht
Reihe: 1
Platz: 7
Platz gebucht
```



```
Reihe:

    Musical-Ticketservice
    ---------------------
    (B)uchung
    (Ü)berblick Vorstellungen
    (E)nde

Auswahl: ü

Hairspray
=========
1.10.2023
56 freie Plätze

2.10.2023
58 freie Plätze

    Musical-Ticketservice
    ---------------------
    (B)uchung
    (Ü)berblick Vorstellungen
    (E)nde

Auswahl: b
Datum der Vorstellung: 1.1.2023
Saalbelegung

   Platz: 1  2  3  4  5  6  7  8  9  10 11 12 13 14
Reihe  1: -- -- -- -- -- Me Me -- -- --
Reihe  2: -- -- -- -- -- -- -- -- -- --
Reihe  3: -- -- -- -- -- -- -- -- -- -- --
Reihe  4: -- -- -- -- -- -- -- -- -- -- --
Reihe  5: -- -- -- -- -- -- -- -- -- -- -- -- --
```

12.5 Aufgaben

Aufgabe 1

Erweitern Sie die Klasse `Kartenverkauf` um eine private Methode, die eine Eintrittskarte druckt (d.h. auf dem Bildschirm ausgibt).

Auf der Eintrittskarte sollen folgende Angaben vermerkt sein:

Titel des Musicals, Datum und Beginn der Vorstellung, Reihe und Platznummer, Name des Zuschauers, der die Karte bestellt hat, und Eintrittspreis.

Die Methode soll im Rahmen der Buchung eines Platzes aufgerufen werden.

Aufgabe 2

Erweitern Sie die Klasse `Vorstellung` im Modul `musical` um eine Methode namens `getZuschauer()`, die eine Liste aller Zuschauer (Objekte der Klasse `Zuschauer`) der jeweiligen Vorstellung liefert.

Aufgabe 3

Erweitern Sie die Klasse `Musical` (Abschnitt 12.4) um eine Methode, die eine Vorstellung storniert.

Kopfzeile der Methodendefinition:

```
def storniere(self, datum):
```

Vorbedingung: Das Argument `datum` ist eine Zeichenkette.

Nachbedingung: Falls in der Liste `self.vorstellungen` ein Objekt `v` der Klasse `Vorstellung` existiert, dessen Attribut `v.datum` mit dem Argument `datum` übereinstimmt, wird dieses Element `v` aus der Liste `self.vorstellungen` entfernt. Zurückgegeben wird ein String mit einer Auflistung aller Zuschauer dieser gelöschten Vorstellung samt deren Telefonnummern, so dass diese über den Ausfall der Vorstellung informiert werden können.

Hinweis: Verwenden Sie die Methode `getZuschauer()` der Klasse `Vorstellung` aus Aufgabe 2.

Aufgabe 4

Für einen Radiosender soll ein System entwickelt werden, das Musikcharts (»Hitparade«, »EinsLive-Hörercharts« etc.) verwaltet. Die Redaktion des Senders stellt am Ende einer Woche 20 Musiktitel zur Wahl. Dabei handelt es sich um die 15 am häufigsten gewählten Titel der letzten Woche plus fünf Neuvorstellungen, die von der Redaktion ausgewählt werden. Die Hörer haben dann bei einem Voting die Möglichkeit, von diesen Titeln einen auszuwählen, der ihnen am besten gefällt.

1. Erstellen Sie für das System ein Geschäftsprozessdiagramm.
2. Entwickeln Sie ein objektorientiertes Modell für die Musikcharts und stellen Sie ein UML-Klassendiagramm auf (nur die Musikcharts ohne Benutzungsoberflächen).

12.6 Lösungen

Die Erweiterungen sind in den folgenden Listings fett gedruckt.

Lösung 1

Skript:

```
class Kartenverkauf(object):
...
def __druckeKarte(self, vorstellung, reihe, platz, name):
print('----------------------------------------')
print('Eintrittskarte')
print(self.__musical.titel)
print('Name:', name)
print('Reihe', reihe, 'Platz', platz)
print('am', vorstellung.datum, end=' ')
print('Beginn', vorstellung.beginn)
print('----------------------------------------'
```

Beispielausgabe:

```
----------------------------------------
Eintrittskarte
Hairspray
Name: Steffen, Sascha
Reihe 2 Platz 4
am 1.10.2023 Beginn 20 Uhr
```

```
-----------------------------------------
```

Lösung 2

Erweiterung der Klasse `Vorstellung` im Modul `musical`:

```
class Vorstellung(object):
...
    def getZuschauer(self):
'''liefert Liste mit Zuschauern der Vorstellung '''
liste = []
for reihe in self.saalbelegung.belegung:
for platz in reihe:
if platz.belegt():
liste += [platz.zuschauer]
return liste
```

Lösung 3

Erweiterung der Klasse `Musical` im Modul `musical`:

```
class Musical(object):
...
def storniere(self, datum):
vorstellung = self.getVorstellung(datum)
if not vorstellung:
return 'Keine Vorstellung an diesem Tag'
else:
zuschauerliste = vorstellung.getZuschauer()
text = 'Liste aller Zuschauer der Vorstellung am '
text += datum + '\n\n'
for zuschauer in zuschauerliste:
text += zuschauer.name+' '+zuschauer.tel+'\n'
self.vorstellungen.remove(vorstellung) #löschen
return text
...
```

Erweiterung der Klasse `Manager` im Skript `musicalmanager.py`:

```
class Manager:
    __menütext = '''
    Musical-Manager
    ------------------
    (n)eue Vorstellung
    (Ü)berblick Vorstellungen
    (s)torniere Vorstellung
    (E)nde
    '''
...

    def __run(self):
...
        elif wahl in ['Ü', 'ü']:
            print(str(self.__musical))
        elif wahl in ['S', 's']: self.__storniere()
...
def __storniere(self):
    datum = input('Datum: ')
    text = self.__musical.storniere(datum)
    print(text)
...
```

Beispieldialog:

```
    Musical-Manager
    ------------------
    (n)eue Vorstellung
    (Ü)berblick Vorstellungen
    (s)torniere Vorstellung
    (E)nde

Auswahl: s
Datum: 1.10.2023
Liste aller Zuschauer der Vorstellung am 1.10.2023

Meyer, Sabine  02331 89664
Steffen, Sascha  0201 344560
```

Lösung 4

a)