Software Testen VU/VL – Übungsbeispiel 2

(35 Punkte)

1. Motivation

Dieses Übungsbeispiel soll als Einführung in die Testautomatisierung und zum Kennenlernen des Testing-Frameworks *JUnit* dienen. Unter Testautomatisierung versteht man den Einsatz von Testtools zur Unterstützung des Testens, wie z.B. die automatisierte Wiederholung von Tests. *JUnit* ist im Besonderen für automatisierte Modultests (Unit-Tests) geeignet.
In Teil B dieser Abgabe wollen wir Ihnen darüber hinaus einen kleinen Vorgeschmack auf die Methode der Testgetriebenen Entwicklung (test-driven development, TDD) geben. Bei dieser agilen Methode werden zunächst einige wenige Tests implementiert, welche nicht bestanden werden. Anschließend wird genau so viel Code geschrieben, damit diese Tests erfolgreich durchgeführt werden können. Anschließend wird der Code "aufgeräumt" und vereinfacht. Diese drei Schritte werden in vielen Iterationen wiederholt bis die gewünschte Funktionalität erreicht ist.

2. Aufgabenstellung

Diese Abgabe gliedert sich in zwei Teile. Teil A dient zum Kennenlernen des Testing-Frameworks *JUnit*. Teil B soll Ihnen helfen Ihre Kenntnisse anhand eines etwas umfangreicheren Beispiels zu vertiefen und gibt eine kurze Einführung in die Methode des *Test-Driven Developments*. Es müssen selbst Testfälle bestimmt und implementiert werden.

Teil A (10 Punkte)

Der erste Teil dieser Abgabe soll als Einführung in das *JUnit*-Framework dienen. Die im Anhang in Abschnitt A-1 angegebenen Testfälle sind dem Programm CalculatorFunctions entnommen, für das bereits der Testplan erstellt wurde. Ziel dieser Abgabe ist es den Umgang mit *JUnit* zu erlernen und die angeführten Testfälle zu implementieren sowie auszuführen.

a) Implementierung der Testfälle (3 Punkte): Implementieren Sie die im Anhang unter Punkt A-1 angeführten Testfälle für die Klasse `CalculatorFunctions` als JUnit-Tests in einer Klasse namens `CalculatorTest`. Benennen Sie die einzelnen Testmethoden nach dem Schema `test<name_der_getesteten_Methode><nummer_des_implementierten_Testfalls>()`. Wählen Sie sinnvolle Werte (nicht extrem große Zahlen mit sehr vielen Kommastellen), mit denen auch Java problemlos umgehen kann.

b) Finden und Dokumentieren der Fehler (4 Punkte): Finden Sie mit Hilfe dieser Tests möglichst viele Fehler in der Klasse `CalculatorFunctions` und dokumentieren Sie diese in der dafür vorgesehenen Tabelle im Abgabedokument. Versuchen Sie die Fehler so zu beschreiben, dass man sich rasch ein Bild von diesen machen kann, ohne die fehlgeschlagenen Testfälle zu kennen. Bewerten Sie die gefundenen Fehler nach den Schweregraden leicht, mittel und schwer. Leichte Fehler sind reine Anzeigefehler oder funktionale Fehler, die leicht umgangen werden können. Mittlere Fehler sind funktionale Fehler, die stören, aber mit etwas Aufwand umgangen werden können. Schwere Fehler sind funktionale, nicht umgehbare Fehler. Sie müssen die gefundenen Fehler <u>nicht</u> korrigieren!

Nr	Beschreibung des Fehlers	Schweregrad	Gefunden durch Test[1]
1			
2			
…			

Tabelle 1 - Fehlerliste

c) Testbericht (3 Punkte): Implementieren Sie die Testfälle so, dass bei deren Ausführung automatisch ein Testbericht ausgegeben wird. Hierbei können Sie sich an der Vorlage (siehe Abschnitt A-2) orientieren. Auf jeden Fall muss klar ersichtlich sein, welche Testfälle erfolgreich durchgeführt wurden bzw. welche fehlgeschlagen sind. Fügen Sie Ihren Testbericht in das Abgabedokument ein.

d) Tragen Sie bitte Ihren gesamten Arbeitsaufwand in die dafür vorgesehene Stundenliste im Abgabedokument ein.

Teil B (25 Punkte)

a) Implementierung der Testklassen (10 Punkte): Entwerfen sie Testfälle zur C1-Überdeckung jeder Methode des gegebenen Fluglinienverwaltungsprogramms (außer für die Klasse Buchungssystem) und implementieren Sie diese in *JUnit*. Die vorgegebenen Testklassen, welche bereits Testfälle enthalten, müssen ergänzt werden. Die übrigen Testklassen sollen nach folgendem Schema benannt werden: <name_der_getesteten_Klasse>Test.java. D.h. die Klasse BuchungsTest.java soll alle Testmethoden enthalten, die zum Testen der Klasse Buchung.java benötigt werden, etc. Die Testmethoden sollen nach dem Schema test<name_der_getesteten_Methode><nr>() benannt werden.

Schreiben Sie außerdem eine ausführbare TestSuite AllTests.java, um alle Tests auf einmal durchführen zu können. Fertigen Sie wie bei Teil A einen Testbericht an und fügen Sie diesen in das Abgabedokument ein.

b) Finden und Dokumentieren der Fehler (10 Punkte): Finden Sie mit Hilfe der Tests möglichst viele Fehler in dem Programm. Es ist davon auszugehen, dass die Spezifikation keine Fehler enthält, suchen Sie die Fehler nur im Source-Code. Dokumentieren Sie die gefundenen Fehler in der dafür vorgesehenen Tabelle im Abgabedokument. Sie müssen die gefundenen Fehler nicht korrigieren!

Nr	Beschreibung des Fehlers	Schweregrad	Gefunden durch Test[1]
1			
2			
…			

Tabelle 2 - Fehlerliste

Unterscheiden Sie hier wiederum nach den Schweregraden leicht, mittel und schwer wie bereits in Teil A beschrieben.

c) Test-Driven Development (5 Punkte): Implementieren Sie anhand der vorgegebenen Tests (siehe GetPassagiereTest.java) die Methode getPassagiere in der Klasse Controller.

[1] Name der Test-Methode, mit welcher der Fehler gefunden wurde.

Diese soll alle Passagiere, die einen bestimmten Flug gebucht haben, in einem Objekt der Klasse *java.util.Vector* zurückgeben.

```
public Vector getPassagiere(Flug flug)
{
   …
   …
}
```

Sobald Sie eine erste Version der Methode implementiert haben, sollten Sie zur Überprüfung die vorgegebenen Tests anwenden. Die Tests zu dieser Methode sollen nicht verändert werden, implementieren Sie die Methode `getPassagiere` so, dass Sie die vorgegebenen Tests erfolgreich durchführen können.

d) Tragen Sie bitte Ihren gesamten Arbeitsaufwand in die dafür vorgesehene Stundenliste im Abgabedokument ein.

3. Vorraussetzungen

Um diese Abgabe durchführen zu können benötigen Sie Sun Java (http://java.sun.com) und das Testing-Framework *Junit* in der Version 4.0. Dieses ist unter http://www.junit.org erhältlich und enthält auch Installationsanweisungen und Tutorials. Des Weiteren empfiehlt sich die Verwendung einer IDE wie z.B. Eclipse (http://www.eclipse.org).

4. Abgabe

Ihre Abgabe sollte folgendes enthalten:

- das Package `calculator` mit dem Quellcode der von Ihnen geschriebenen Testklasse `CalculatorTest`. Achtung – das Package soll nicht verändert werden!
- das Package `buchungssystem.src` mit der von Ihnen implementierten Funktion `getPassagiere`. Achtung – das Package soll nicht verändert werden!
- das Package `buchungssystem.tests` mit sämtlichen Testklassen inkl. TestSuite.
- das Abgabedokument benannt als `<Nachname>_<Vorname>_<MatrikelNr>_<Abgabe2>.doc`

Zippen Sie alle Teile Ihrer Abgabe als `<Nachname>_<Vorname>_<MatrikelNr>_<Abgabe2>.zip` und laden Sie es in TUWeL bis spätestens **30.04.2006, 23:55** hoch. Für das Nichteinhalten der Abgabekonventionen müssen wir leider Punkte abziehen.

Anhang

A-1 Testfälle für die Klasse CalculatorFunctions

a. testAdd

#	Typ	Beschreibung Testfall	Erwartetes Ergebnis
1	NF	Addieren zweier beliebiger Zahlen	Summe der zwei Zahlen
2	NF	Addition, 1. Summand positiv und 2. Summand negativ	Summe der zwei Zahlen
3	NF	Addition, 1. Summand negativ und 2. Summand positiv	Summe der zwei Zahlen

b. testSubtract

#	Typ	Beschreibung Testfall	Erwartetes Ergebnis
1	NF	Subtraktion zweier beliebiger Kommazahlen	Differenz der zwei Zahlen
2	NF	Subtraktion, Minuend positiv und Subtrahend negativ	Differenz der zwei Zahlen
3	NF	Subtraktion, Minuend negativ und Subtrahend positiv	Differenz der zwei Zahlen

c. testMultiply

#	Typ	Beschreibung Testfall	Erwartetes Ergebnis
1	NF	Multiplikation zweier positiver Kommazahlen	Produkt der zwei Zahlen
2	NF	Multiplikation zweier beliebiger negativer Zahlen	Produkt der zwei Zahlen
3	NF	Multiplikation einer negativen Zahl mit einer positiven Zahl	Produkt der zwei Zahlen
4	NF	Multiplikation einer Zahl mit 0	0

d. testDivide

#	Typ	Beschreibung Testfall	Erwartetes Ergebnis
1	FF	Division durch 0	null
2	NF	Division zweier beliebiger Kommazahlen	Quotient der zwei Zahlen
3	NF	Division mit Dividend positiv, Divisor negativ	Quotient der zwei Zahlen
4	NF	Division mit Dividend negativ, Divisor negativ	Quotient der zwei Zahlen
5	NF	Division mit Dividend = 0, Divisor != 0	0

e. testNegate

#	Typ	Beschreibung Testfall	Erwartetes Ergebnis
1	NF	Negieren einer negativen Zahl x	positive Zahl; -x
2	NF	Negieren einer positiven Zahl x	negative Zahl; -x
3	NF	Negieren der Zahl 0	0

f. testAbsoluteValue

#	Typ	Beschreibung Testfall	Erwartetes Ergebnis
1	NF	Berechnung des Absolutwertes einer negativen Zahl x	positive Zahl; -x
2	NF	Berechnung des Absolutwertes einer positiven Zahl x	positive Zahl; x
3	NF	Berechnung des Absolutwertes der Zahl 0	0

g. testRoot

#	Typ	Beschreibung Testfall	Erwartetes Ergebnis
1	NF	Berechnung der Wurzeln einer Basis > 0 mit Exponent > 0	Wurzel der Zahl
2	FF	Berechnung der Wurzeln einer Basis > 0 mit Exponent = 0	null
3	NF	Berechnung der Wurzel einer Basis < 0 mit Exponent ganzzahlig, ungerade	Wurzel der Zahl
4	NF	Berechnung einer Wurzel einer Basis < 0 mit 1/Exponent ganzzahlig	Wurzel der Zahl
5	NF	Berechnung einer Wurzel einer Basis < 0 mit Exponent ganzzahlig und gerade	null
6	NF	Berechnung der Wurzel der Basis 0 mit Exponent < 0	null
7	NF	Berechnung der Wurzel der Basis 0 mit Exponent > 0	0

h. testPower

#	Typ	Beschreibung Testfall	Erwartetes Ergebnis
1	NF	Potenzieren einer Basis < 0 mit Exponent ganzzahlig, gerade und > 0	Potenz der Zahl; positiv
2	NF	Potenzieren einer Basis < 0 mit Exponent ganzzahlig, ungerade und > 0	Potenz der Zahl; negativ
3	NF	Potenzieren einer Basis > 0 mit Exponent < 0	Potenz der Zahl; positiv
4	NF	Potenzieren der Basis 0 mit beliebigem Exponent > 0	0
5	SF	Potenzieren einer beliebigen Basis mit Exponent = 0	1

i. testRound

#	Typ	Beschreibung Testfall	Erwartetes Ergebnis
1	NF	Abrunden einer positiven Zahl der Form x.4	x
2	NF	Aufrunden einer positiven Zahl der Form x.5	x+1
3	NF	Aufrunden einer negativen Zahl der Form -x.5	-x
4	NF	Abrunden einer negativen Zahl der Form -x.51	-x-1

j. testFactorial

#	Typ	Beschreibung Testfall	Erwartetes Ergebnis
1	SF	Berechnung der Fakultät von 0	1
2	NF	Berechnung der Fakultät einer positiven ganzen Zahl x	Fakultät der Zahl; x!
3	FF	Berechnung der Fakultät einer negativen ganzen Zahl x	null

A-2 Beispiel für einen Testbericht

```
Durchführen der CalculatorFunctions-Tests am - Fri Mar 17 15:03:12 CEST 2006
--------------------------------------------------------------------------
.testAdd1                     Passed
.testAdd2                     Passed
.testAdd3                     Passed
.testSubtract1                Passed
.testSubtract2                Passed
.testSubtract3                Passed
.testMultiply1                Passed
.testMultiply2                Passed
.testMultiply3                Passed
.testMultiply4                Passed
.testDivide1                  Passed
.testDivide2                  Passed
.testDivide3                  Passed
.testDivide4                  Passed
```

```
.testDivide5                 Passed
.testNegate1                 Passed
.testNegate2                 Passed
.testNegate3                 Passed
.testExample                 F
.testAbsoluteValue1          Passed
.testAbsoluteValue2          Passed
.testAbsoluteValue3          Passed
.testRoot1                   Passed
.testRoot2                   Passed
.testRoot3                   Passed
.testRoot4                   Passed
.testRoot5                   Passed
.testRoot6                   Passed
.testRoot7                   Passed
.testPower1                  Passed
.testPower2                  Passed
.testPower3                  Passed
.testPower4                  Passed
.testPower5                  Passed
.testRound1                  Passed
.testRound2                  Passed
.testRound3                  Passed
.testRound4                  Passed
.testFactorial1              Passed
.testFactorial2              Passed
.testFactorial3              Passed

Time: 0,03

There were x failures
```

A-3 Programmdokumentation für das Buchungssystem

Der folgende Abschnitt soll Ihnen einen kurzen Überlick über das in Teil B dieser Abgabe verwendete Fluglinienverwaltungsprogramm geben.

a. Kurzbeschreibung

Beim vorliegenden Programm handelt es sich um ein vereinfachtes Fluglinienverwaltungsprogramm. Das Programm hat folgende Features:

1 - Flugzeug eintragen
2 - Flug eintragen
3 - Kundendaten eintragen
4 - Flug buchen
5 - Fluginformationen anzeigen

Das Programm wird über die Tastatur gesteuert. Falls dem Programm beim Aufruf der Pfad einer korrekt formatierten Eingabedatei als Parameter übergeben wird, wird diese bei Programmstart eingelesen. Dadurch haben Sie im Programm immer einige Datensätze.

Anmerkung: Für die Durchführung von Beispiel 2 ist es nicht erforderlich, das Programm selbst auszuführen und damit zu arbeiten. Wir empfehlen dies jedoch, da Sie so leichter eine Übersicht über die Programmfunktionen bekommen können. Beachten Sie dabei bitte, dass das Programm fehlerhaft ist und daher einige Funktionen möglicherweise nicht korrekt ausgeführt werden können.

Eine beispielhafte Eingabedatei ist in der Angabe enthalten. Sie können diese Datei nach Belieben

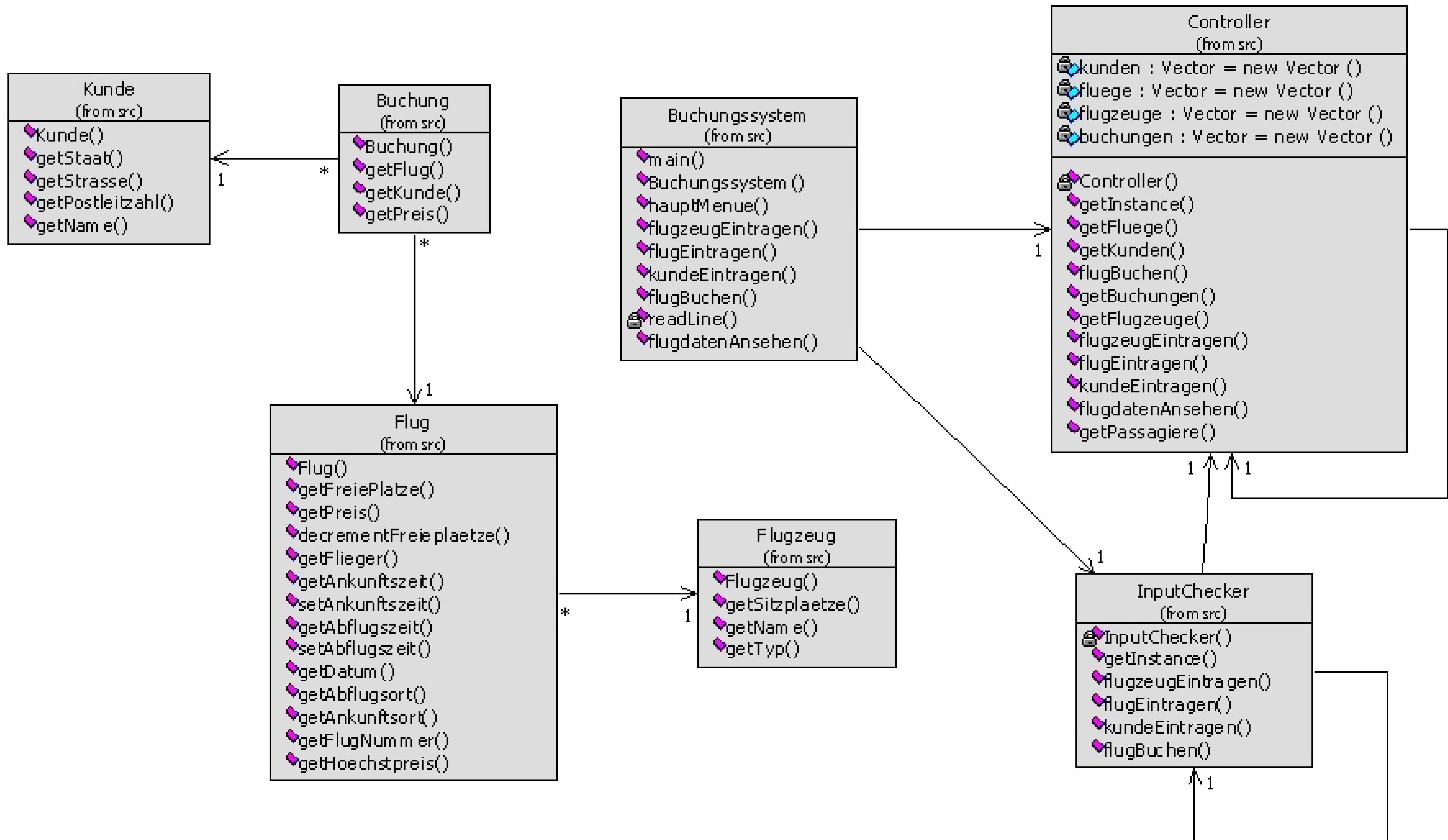
erweitern.

Die Eingabedatei hat folgendes Format:

1 → Flugzeugtyp → Flugzeugname → Sitzplaetze
2 → Datum → Abflugszeit → Ankunftszeit → Höchtspreis → Abflugsort → Ankunftsort → Flugnummer → Flugzeugname
3 → Name → Postleitzahl → Strasse → Staat
4 → Flugnummer → Abflugsort → Ankunftsort → Datum → Abflugszeit → Ankunftszeit → Kundenname
5 → Flugnummer

Der erste Parameter gibt an, welche Methode ausgeführt werden soll (siehe oben). Die restlichen Parameter werden der jeweiligen Methode übergeben. Die einzelnen Parameter sind durch Tabulatorzeichen getrennt; das Ende einer Zeile zeigt den Anfang eines neuen Methodenaufrufs an.

b. Klassendiagramm



c. Dokumentation der Klassen

Klasse Buchung.java

Konstruktor:

`public Buchung(Kunde k, Flug f, double Preis)`	Initialisiert ein Buchungs-Objekt

Methoden:

`public Flug getFlug()`	Gibt den zur Buchung gehörenden Flug zurück
`public Kunde getKunde()`	Gibt den zur Buchung gehörenden Kunden zurück
`public double getPreis()`	Gibt den zur Buchung gehörenden Preis zurück

Klasse Buchungssystem.java

Konstruktor:

`public Buchungssystem(String[] args)`	Liest das File, dessen Pfad übergeben wurde, Zeile für Zeile ein, und ruft die entsprechenden Methoden des `InputChecker` auf.

Methoden:

`public static void main(String[] args)`	Erzeugt ein Buchungssystem-Objekt und ruft das Hauptmenü auf
`public void hauptMenue()`	Gibt das Hauptmenü aus, liest die Eingabe des Benutzers ein und ruft die entsprechenden Methoden auf
`public void flugzeugEintragen()`	Liest die notwendigen Daten zum Eintragen eines Flugzeugs ein und gibt diese an den `InputChecker` weiter
`public void flugEintragen()`	Liest die notwendigen Daten zum Eintragen eines Flugs ein und gibt diese an den `InputChecker` weiter
`public void kundeEintragen()`	Liest die notwendigen Daten zum Eintragen eines Kunden ein und gibt diese an den `InputChecker` weiter

`public void **flugBuchen**()`	Liest die notwendigen Daten zum Buchen eines Flugs ein und gibt diese an den `InputChecker` weiter
`private String **readLine**()`	Liest eine Zeile von System.in und gibt diese zurück
`public void **flugdatenAnsehen**()`	Liest eine Flugnummer ein und gibt die Daten des dazugehörigen Flugs aus

Klasse Controller.java

Diese Klasse verwaltet Kunden, Flüge, Flugzeuge und Buchungen. Es darf nur ein Objekt (Instanz) der Klasse Controller erzeugt werden. Um dies zu gewährleisten, ist diese Klasse nach dem Singleton-Pattern implementiert.

Konstruktor:

`private **Controller**()`	Privater Konstruktor; gemäß dem Singleton-Pattern

Methoden:

`public static Controller **getInstance**()`	Gibt immer dieselbe Instanz der Klasse `Controller` zurück bzw. erzeugt eine Instanz falls keine existiert (Singleton-Pattern)
`public Vector **getFluege**()`	Gibt den Vector mit den Flügen zurück
`public Vector **getKunden**()`	Gibt den Vector mit den Kunden zurück
`public void **flugBuchen**(Kunde k, Flug f)`	Bucht den Flug f für den Kunden k, die freien Plätze für diesen Flug werden um 1 vermindert, das Buchungsobjekt zum Vector hinzugefügt
`public Vector **getBuchungen**()`	Gibt den Vector mit den Buchungen zurück
`public Vector **getFlugzeuge**()`	Gibt den Vector mit den Flugzeugen zurück
`public void **flugzeugEintragen**(String typ, String name, int sitzplaetze)`	Erzeugt ein Flugzeug-Objekt und fügt es in den Flugzeug-Vector ein
`public void **flugEintragen**(Flugzeug f, String datum, String abflugszeit, String ankunftszeit, int hoechstpreis, String abflugsort, String ankunftsort, String flugnr)`	Erzeugt ein Flug-Objekt und fügt es in den Flüge-Vector ein

`public void **kundeEintragen**(String name, String plz, String strasse, String staat)`

	Erzeugt ein Kunden-Objekt und fügt es in den Kunden-Vector ein
`public String` **`flugdatenAnsehen`**`(String flugnummer)`	Gibt die Daten des zur Flugnummer gehörenden Flugs zurück. Existiert der Flug nicht wird eine Fehlermeldung zurückgegeben.
`public Vector` **`getPassagiere`**`(Flug f)`	Gibt alle Passagiere des Flugs f als in einem Vector zurück.

Klasse Flug.java

Konstruktor:

`public` **`Flug`**`(Flugzeug f, String datum, String abflugszeit, String ankunftszeit, int hoechstpreis, String abflugsort, String ankunftsort, String flugnummer)` Initialisiert ein Flug-Objekt

Methoden:

`public int` **`getFreiePlatze`**`()`	Gibt die Anzahl der freien Plätze zurück
`public double` **`getPreis`**`()`	Gibt den aktuellen Preis des Fluges zurück. Der Preis eines Fluges berechnet sich folgendermaßen: Die ersten 30% der Plätze kosten 30% des Höchstpreises, die nächsten 40% kosten 70% des Höchstpreises und die restlichen Plätze 100% des Höchstpreises. Bei einer Sitzplatzkapazität von 100 Plätzen und einem Höchstpreis von 1000€ zahlt man für die ersten 30 Tickets je 300€, für die nächsten 40 Tickets je 700€und für die restlichen 30 Plätze je 1000€
`public void` **`decrementFreieplaetze`**`()`	Vermindert die Anzahl der freien Plätze um eins
`public Flugzeug` **`getFlieger`**`()`	Gibt das Flugzeug des Fluges zurück
`public String` **`getAnkunftszeit`**`()`	Gibt die Ankunftszeit des Flugs zurück
`public void` **`setAnkunftszeit`**`(String ankunftszeit)`	Setzt die Ankunftszeit des Fluges auf `ankunftszeit`
`public String` **`getAbflugszeit`**`()`	Gibt die Abflugszeit des Flugs zurück
`public void` **`setAbflugszeit`**`(String abflugszeit)`	Setzt die Abflugszeit des Fluges auf `abflugszeit`
`public String` **`getDatum`**`()`	Gibt das Datum des Flugs zurück
`public String` **`getAbflugsort`**`()`	Gibt den Abflugsort des Flugs zurück
`public String` **`getAnkunftsort`**`()`	Gibt den Ankunftsort des Flugs zurück

`public String` **`getFlugNummer`**`()`	Gibt die Flugnummer des Flugs zurück
`public int` **`getHoechstpreis`**`()`	Gibt den Höchstpreis für diesen Flug zurück

Klasse Flugzeug.java

Konstruktor:

`public` **`Flugzeug`**`(String name, int sitzplaetze, String typ)`	Initialisiert ein Flugzeug-Objekt

Methoden:

`public int` **`getSitzplaetze`**`()`	Gibt die Anzahl der Sitzplätze zurück
`public String` **`getName`**`()`	Gibt den Namen des Flugzeugs zurück
`public String` **`getTyp`**`()`	Gibt den Typ des Flugzeugs zurück

Klasse InputChecker.java

Konstruktor:

`private` **`InputChecker`**`()`	Privater Konstruktor; gemäß dem Singleton-Pattern

Methoden:

`public static InputChecker` **`getInstance`**`()`	Gibt immer dieselbe Instanz der Klasse `InputChecker` zurück bzw. erzeugt eine Instanz falls keine existiert (Singleton-Pattern)

`public void` **`flugzeugEintragen`**`(String typ, String name, String sitzplaetze)`

Überprüft die übergebenen Daten. Sind diese korrekt wird der Aufruf an den `Controller` weitergegeben. Folgende Kriterien sind erforderlich: Sitzplätze: ganze Zahl > 0, Typ und Name: ungleich null

`public void` **`flugEintragen`**`(String datum, String abflugszeit, String ankunftszeit, String hoechstpreis, String abflugsort, String ankunftsort, String flugNummer, String flugzeugname)`

Überprüft die übergebenen Daten. Sind diese korrekt wird der Aufruf an den `Controller` weitergegeben. Folgende Kriterien sind erforderlich: Höchstpreis: ganze Zahl > 0, alle anderen Parameter: ungleich null

`public void` **`kundeEintragen`**`(String name, String plz, String strasse, String staat)`

Überprüft die übergebenen Daten. Sind diese korrekt wird der Aufruf an den `Controller` weitergegeben. Folgende Kriterien sind erforderlich: alle Parameter müssen ungleich null sein, der Name muss auch ungleich dem leeren String (““) sein

`public void` **`flugBuchen`**`(String flugnummer, String abflugsort, String ankunftsort, String datum, String abflugszeit, String ankunftszeit, String kundenname)`

Überprüft die übergebenen Daten. Sind diese korrekt wird der Aufruf an den `Controller` weitergegeben. Folgende Kriterien sind erforderlich: der zur übergebenen Flugnummer gehörende Flug muss existieren und darf nicht ausgebucht sein, der Kunde muss eingetragen sein, alle übrigen Parameter: ungleich null

Klasse Kunde.java

Konstruktor:

`public` **`Kunde`**`(String name, String plz, String strasse, String staat)`	Initialisiert ein Kunden-Objekt

Methoden:

`public String` **`getStaat`**`()`	Gibt den Staat des Kunden zurück
`public String` **`getStrasse`**`()`	Gibt die Straße des Kunden zurück
`public String` **`getPostleitzahl`**`()`	Gibt die Postleitzahl des Kunden zurück
`public String` **`getName`**`()`	Gibt den Namen des Kunden zurück