Universität Augsburg
Institut für Informatik

**Übung zur Vorlesung Informatik 2** SoSe 2024
Fakultät für Angewandte Informatik
Institut für Informatik
Prof R. Lorenz, L. Petrak, J. Linne, V. Le Claire, P. Schalk 14.06.2024

# Lösungsvorschlag zu Übungsblatt 7

**Abgabe: 24.06.2024, 10:00 Uhr** (im Digicampus via VIPS: .java-Dateien für Code, .uxf für UML, .pdf für alles andere)

- Dieses Übungsblatt muss im Team abgegeben werden (Einzelabgaben sind nicht erlaubt!).
- Die **Zeitangaben** geben zur Orientierung an, wie viel Zeit für eine Aufgabe später in der Klausur vorgesehen wäre; gehen Sie davon aus, dass Sie zum jetzigen Zeitpunkt wesentlich länger brauchen und die angegebene Zeit erst nach ausreichender Übung erreichen.

* leichte Aufgabe / ** mittelschwere Aufgabe / *** schwere Aufgabe

**Allgemeine Hinweise zur Erstellung und Abgabe von UML-Diagrammen:**

- UML-Diagramme sind mit dem Editor **UMLet**[1] zu erstellen (es darf auch die webbasierte Variante **UMLetino**[2] verwendet werden). Damit die Leserichtung von Assoziationen beim PDF-Export aus UMLet erhalten bleibt, müssen Sie eine Schriftart installieren, welche die verwendeten Zeichen beinhaltet, wie z.B. `https://fonts2u.com/download/free-sans.family`. Exportieren Sie die enthaltene .zip-Datei und tragen Sie den absoluten Pfad zu den Dateien in UMLet unter *File* → *Options* → *Optional font to embedd in PDF* für alle vier Varianten ein, verwenden Sie die `Oblique`-Dateien für *italic text* und *bold+italic*.
- Die (graphischen) Syntaxvorgaben aus der Vorlesung sind einzuhalten.
- Modellieren Sie Attribute, Konstruktoren und Operationen im Stil der Vorlesung (z.B. im Hinblick auf die Benennung).
- Erfinden Sie keine Daten oder Operationen hinzu, die nicht der jeweiligen Beschreibung entnommen werden können.
- Wenn Sie eine API-Klasse bzw. -Schnittstelle als Eingabeparameter oder Rückgabetyp verwenden, reicht die Angabe als `java.package.Klasse` ohne Modellierung als eigene Klasse bzw. Schnittstelle.
- Wenn Sie Beziehungen zu API-Klassen bzw. Schnittstellen wie Vererbung oder Implementierung benötigen, modellieren Sie die Klassen bzw. Schnittstellen mit derselben Namensgebung `java.package.Klasse`.
- Operationen und Attribute von API-Klassen und Assoziationen zwischen API-Klassen werden nur dann modelliert, wenn sie für die Funktionalität der modellierten Anwendung notwendig sind oder wenn die Aufgabenstellung es explizit verlangt.
- UML-Diagramme sind **sowohl** als UMLet-Datei (`*.uxf`) **als auch** als PDF abzugeben.

[1] `https://www.umlet.com`
[2] `http://www.umletino.com`

## Aufgabe 25 ** *(Modale Dialoge, 25 Minuten)*

In dieser Aufgabe sollen Sie einen Dialog zum Editieren eines Eintrags für ein Auto implementieren. Verwenden Sie dafür die im Digicampus zur Verfügung gestellte Implementierung. Machen Sie sich insb. mit der Klasse `Car` vertraut – Sie werden einige Methoden der Klasse verwenden müssen.
*Hinweis: Sie müssen Ihren Dialog im Paket GUI implementieren, damit Sie `CarMain` ausführen können.*

a) *(**, 15 Minuten)*
Implementieren Sie einen **modalen** Dialog mit folgenden Eigenschaften:

- Der Titel des Dialogs soll `Edit Car` sein.
- An den Dialog wird sowohl das Eltern-Fenster als auch das zu editierende Auto übergeben.
- Für die beiden Eigenschaften eines Autos (`licensePlate` und `numberOfSeats` soll jeweils ein Label mit dem Namen und ein Textfeld mit dem Wert des Attributs in einem Gitter angeordnet sein.
- Es sollen zwei Buttons `Save` und `Cancel` unterhalb des Gitters sein, die nicht Teil des Gitters sind.
- Beim Klicken auf das Fensterkreuz soll nichts passieren.

b) *(***, 10 Minuten)*
Implementieren Sie die Ereignisbehandlung des Dialogs:

- Beim Klicken des `Save`-Buttons soll versucht werden, das übergebene Auto mit den eingegebenen Werten zu aktualisieren. Fangen Sie eventuell auftretende Fehler ab und informieren Sie den Benutzer über den aufgetretenen Fehler mithilfe eines passenden Hilfsdialogs. Sind die Änderungen ohne Fehler durchgeführt worden, soll der Editier-Dialog geschlossen werden. Achten Sie darauf, die Werte des `Car`-Objekts nur dann zu ändern, wenn auch wirklich alle Änderungen durchgeführt werden können!
- Beim Klicken des `Cancel`-Buttons soll der Benutzer über einen passenden Hilfsdialog gefragt werden, ob er seine Änderungen wirklich verwerfen möchte. Bestätigt der Nutzer diesen Dialog, soll der Editier-Dialog ohne Änderungen geschlossen werden.

*Hinweis: Die Werte im Dropdown-Menü des Hauptfensters aktualisieren sich erst, wenn sie einmal angeklickt werden – In Kapitel 11 lernen Sie eine Lösung für dieses Problem.*

**Lösung:**

```
package gui;

import data.Car;
import data.CarException;

import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class CarEditDialog extends JDialog implements ActionListener {
    private JTextField licenseField;
    private JTextField numberOfSeatField;
    private JButton saveButton;
    private JButton cancelButton;
    private Car car;
```

```
    public CarEditDialog(JFrame parent, Car car) {
        super(parent, "Edit Car", true);
        setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
        this.car = car;
        JPanel center = new JPanel();
        center.setLayout(new GridLayout(0, 2));
        center.add(new JLabel("License Plate:"));
        licenseField = new JTextField(car.getLicensePlate());
        center.add(licenseField);
        center.add(new JLabel("Number of Seats:"));
        numberOfSeatField = new JTextField(String.valueOf(car.getNumberOfSeats()));
        center.add(numberOfSeatField);
        add(center, BorderLayout.CENTER);
        JPanel south = new JPanel();
        saveButton = new JButton("Save changes");
        saveButton.addActionListener(this);
        cancelButton = new JButton("Cancel");
        cancelButton.addActionListener(this);
        south.add(saveButton);
        south.add(cancelButton);
        add(south, BorderLayout.SOUTH);

        pack();
        setVisible(true);
    }

    private void onSave() {
        try {
            car.setNumberOfSeats(Integer.parseInt(numberOfSeatField.getText()));
            car.setLicensePlate(licenseField.getText());
            this.dispose();
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(this, "Please enter a valid number");
        } catch (CarException e) {
            JOptionPane.showMessageDialog(this, "Number of Seats must be positive");
        }

    }

    private void onCancel() {
        if (JOptionPane.showConfirmDialog(this, "Do you want to discard your changes?")
                == JOptionPane.YES_OPTION) {
            this.dispose();
        };
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource().equals(saveButton)) {
            onSave();
        } else if (e.getSource().equals(cancelButton)) {
            onCancel();
        }
    }
}
```

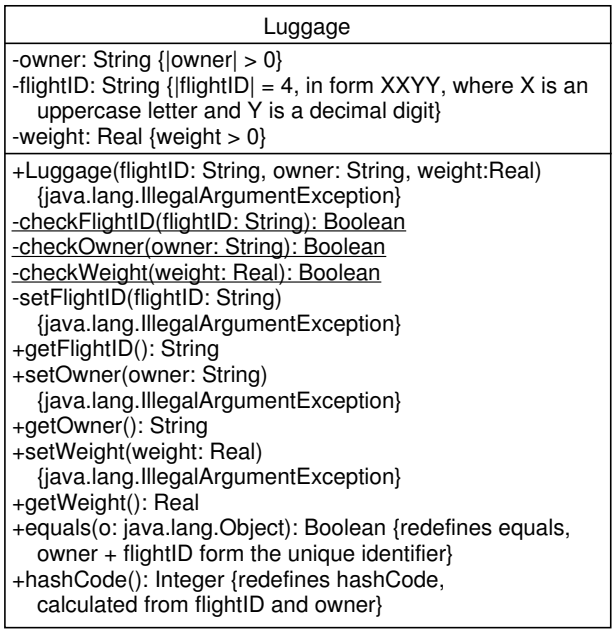
## Aufgabe 26 *(UML-Klassen erstellen, 25 Minuten)*

In dieser Aufgabe sollen Sie jeweils eine eigene UML-Datenklasse auf Basis einer Systembeschreibung modellieren. Für Ausnahmen dürfen Sie passende Java-API-Ausnahmen, in der Vorlesung eingeführte Ausnahmen oder auch eigene neue Ausnahmen mit passendem Namen verwenden.

a) *(**, 15 Minuten)*
Modellieren Sie eine Klasse `Luggage` für Gepäckstücke **mit** Standardverwaltungsoperationen nach folgender Systembeschreibung:

- Ein Gepäckstück hat eine Flug-ID, einen Besitzer und ein Gewicht.
- Der Besitzer wird durch eine nicht-leere Zeichenkette dargestellt.
- Das Gewicht ist eine positive reelle Zahl.
- Es soll eine Möglichkeit angeboten werden, `Luggage`-Objekte zu erstellen. Auftretende Ausnahmen sollen dabei weitergereicht werden.
- Die Flug-ID ist von der Form `"XXYY"`, wobei `X` für einen beliebigen Großbuchstaben und `Y` für eine beliebige Ziffer steht.
- Die Flug-ID wird beim Erstellen eines `Luggage`-Objekts gesetzt und kann danach nicht mehr verändert werden.
- Ist eine der Datenstruktur-Invarianten verletzt, soll die zugehörige set-Methode eine `IllegalArgumentException` werfen.
- Zwei `Luggage`-Objekte gelten als gleich, wenn ihre Flug-ID und ihr Besitzer übereinstimmen.
- Der Hashcode eines `Luggage`-Objekts soll aus seiner Flug-ID und dem Namen seines Besitzers berechnet werden.

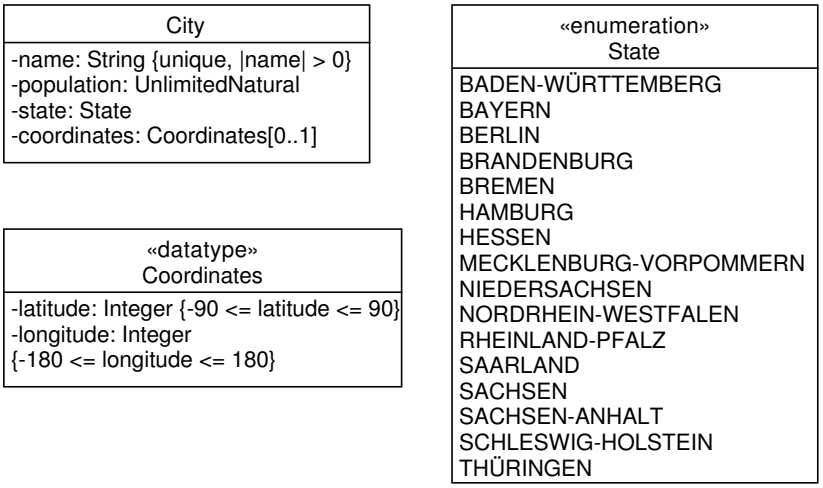
### Lösung:



b) *(**, 10 Minuten)*
Modellieren Sie folgende Systembeschreibung für ein System zur Verwaltung von Städten in Deutschland. Verwenden Sie eine Aufzählung für die deutschen Bundesländer und einen strukturierten Datentyp für die Koordinaten. Modellieren Sie **keine** Konstruktoren oder Operationen!

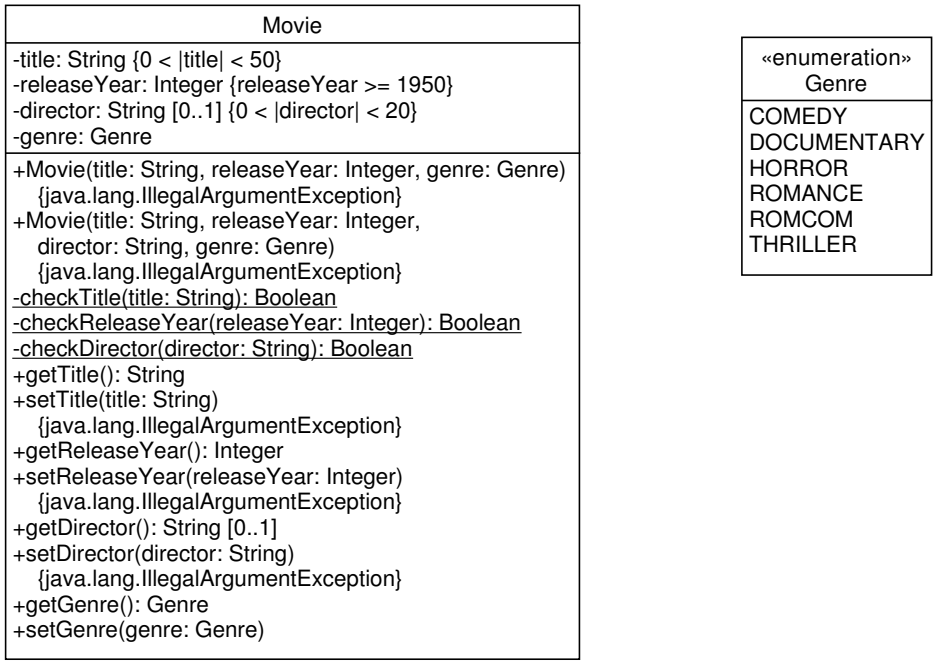
*Eine Stadt hat einen eindeutigen Namen, der durch eine nicht-leere Zeichenkette dargestellt wird. Eine Stadt hat eine positive, ganzzahlige Einwohnerzahl. Eine Stadt liegt in einem der 16 deutschen Bundesländer. Außerdem können für eine Stadt Koordinaten hinterlegt werden: Koordinaten bestehen aus dem Breitengrad, einer reellwertigen Zahl zwischen -90 und 90 (jeweils einschließlich), und einem Längengrad, einer reellwertigen Zahl zwischen -180 und 180 (jeweils einschließlich).*

**Lösung:**



## Aufgabe 27 ** *(UML-Klassen in Java implementieren, 22 Minuten)*

Betrachten Sie das folgende Klassendiagramm mit Klassen, Datentypen und Aufzählungen.




a) *(*, 3 Minuten)*
Implementieren Sie den Aufzählungstyp `Genre` in Java. Benutzen Sie dazu das Java-Sprachkonstrukt `enum`. Dessen Benutzung wurde in den Screencasts zu Kapitel 08 vorgestellt.
**Lösung:**

```
public enum Genre {
    COMEDY,
    DOCUMENTARY,
    HORROR,
    ROMANCE,
    ROMCOM,
    THRILLER
}
```

b) *(*, 19 Minuten)*
Implementieren Sie die Klasse `Movie` in Java. Implementieren Sie dabei **keine** Attribute, Konstruktoren oder Operationen, die nicht direkt dem Diagramm entnommen werden können. verwenden Sie für die Rückgabe-Methoden von optionalen Attributen die Klasse `Optional`.
**Lösung:**

```
import java.util.Optional;

public class Movie {
    private int releaseYear;
    private String title;
    private String director;
    private Genre genre;

    public Movie(int releaseYear, String title, Genre genre) {
        setReleaseYear(releaseYear);
        setTitle(title);
        setGenre(genre);
    }
```

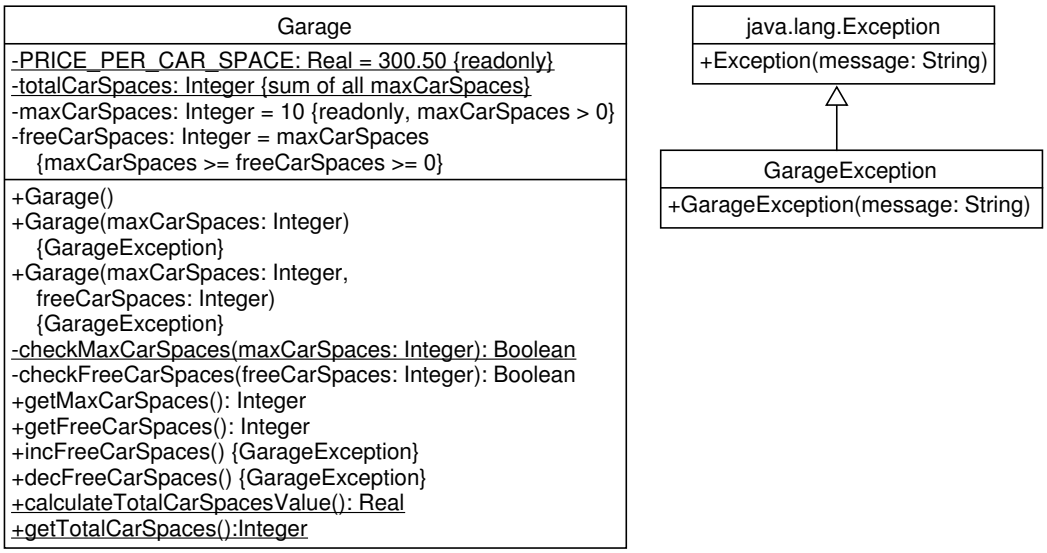
```
    public Movie(int releaseYear, String title, String director, Genre genre) {
        this(releaseYear, title, genre);
        setDirector(director);
    }

    private static boolean checkReleaseYear(int releaseYear) {
        return releaseYear >= 1950;
    }

    private static boolean checkTitle(String title) {
        return !title.isEmpty() && title.length() < 50;
    }

    private static boolean checkDirector(String director) {
        if (director == null)
            return true;
        return !director.isEmpty() && director.length() < 20;
    }

    public int getReleaseYear() {
        return releaseYear;
    }

    public void setReleaseYear(int releaseYear) {
        if (!checkReleaseYear(releaseYear))
            throw new IllegalArgumentException("Release Year must be at least 1950");
        this.releaseYear = releaseYear;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        if (!checkTitle(title))
            throw new IllegalArgumentException("Title must not be empty and less than 50
            ↪  characters long");
        this.title = title;
    }

    public Optional<String> getDirector() {
        return Optional.ofNullable(director);
    }

    public void setDirector(String director) {
        if (!checkDirector(director))
            throw new IllegalArgumentException("Director must be null or a non-empty string
            ↪  of length less than 20");
        this.director = director;
    }

    public Genre getGenre() {
        return genre;
    }

    public void setGenre(Genre genre) {
        this.genre = genre;
    }
}
```

## Aufgabe 28 ** *(Eigene Klasse modellieren und implementieren, 35 Minuten)*

In dieser Aufgabe sollen Sie eine eigene Klasse zuerst modellieren und dann implementieren. Verwenden Sie dazu folgende Systembeschreibung zur Verwaltung von Garagen:

*Eine Garage hat eine Anzahl von Stellplätzen und eine Anzahl von freien Stellplätzen. Eine Garage hat mindestens einen Stellplatz. Die Anzahl an Stellplätzen einer Garage kann nicht mehr verändert werden. Es sind nie mehr Stellplätze frei als die Garage Stellplätze hat. Es kann auch keine negative Anzahl an Stellplätzen frei sein. Standardmäßig haben neue Garagen 10 Stellplätze, es können aber auch größere oder kleinere Garagen aufgenommen werden oder auch Garagen, die bereits Stellplätze vergeben haben. Es soll Möglichkeiten geben, die Anzahl an freien Stellplätzen um eins zu erhöhen bzw. zu verringern. Der Preis eines Stellplatzes ist immer 300,50 Euro. Es soll außerdem eine Möglichkeit geben, die Gesamtanzahl der Stellplätze aller Garagen zu bestimmen und den Gesamtwert der Stellplätze aller Garagen zu berechnen.*

a) *(**, 14 Minuten)*
Modellieren Sie eine Klasse Garage, die alle Elemente der Systembeschreibung beinhaltet. Modellieren Sie außerdem eine eigene **geprüfte** Ausnahme für die Behandlung von eventuell auftretenden Fehlern. Für diese soll eine Fehlernachricht gesetzt werden können.
**Lösung:**




b) *(**, 21 Minuten)*
Implementieren Sie die beiden Klassen, die Sie in der vorherigen Teilaufgabe modelliert haben.

*Hinweis: Achten Sie insbesondere darauf, dass die Anzahl an insgesamt vorhandenen Stellplätzen auch in den Fehlerfällen der Konstruktoren korrekt bleibt.*

**Lösung:**
**GarageException.java:**

```
public class GarageException extends Exception {
    public GarageException(String message) {
        super(message);
    }
}
```

**Garage.java:**

```
public class Garage {
    private static final double PRICE_PER_CAR_SPACE = 300.50;
    private static int totalCarSpaces = 0;
    private final int maxCarSpaces;
    private int freeCarSpaces;

    public Garage() {
        maxCarSpaces = 10;
        totalCarSpaces += maxCarSpaces;
        initFreeCarSpaces();
    }

    public Garage(int maxCarSpaces) throws GarageException {
        if (!checkMaxCarSpaces(maxCarSpaces))
            throw new GarageException("maxCarSpaces must be > 0");
        this.maxCarSpaces = maxCarSpaces;
        totalCarSpaces += maxCarSpaces;
        initFreeCarSpaces();

    }

    public Garage(int maxCarSpaces, int freeCarSpaces) throws GarageException {
        this(maxCarSpaces);
        if (!checkFreeCarSpaces(freeCarSpaces)) {
            totalCarSpaces -= maxCarSpaces;
            throw new GarageException("freeCarSpaces must be >= 0 and <= maxCarSpaces");
        }
        this.freeCarSpaces = freeCarSpaces;
    }

    public static boolean checkMaxCarSpaces(int maxCarSpaces) {
        return maxCarSpaces > 0;
    }

    private boolean checkFreeCarSpaces(int freeCarSpaces) {
        return freeCarSpaces >= 0 && freeCarSpaces <= maxCarSpaces;
    }

    private void initFreeCarSpaces() {
        freeCarSpaces = maxCarSpaces;
    }

    public void incFreeCarSpaces() throws GarageException {
        if (!checkFreeCarSpaces(freeCarSpaces + 1))
            throw new GarageException("Can't increment freeCarSpaces: All car spaces are
            ↪  already free");
        freeCarSpaces++;
    }

    public void decFreeCarSpaces() throws GarageException {
        if (!checkFreeCarSpaces(freeCarSpaces - 1))
            throw new GarageException("Can't decrement freeCarSpaces: All car spaces are
            ↪  already taken");
        freeCarSpaces--;
    }

    public static double calculateTotalCarSpacesValue() {
        return totalCarSpaces * PRICE_PER_CAR_SPACE;
    }

    public static int getTotalCarSpaces() {
        return totalCarSpaces;
    }
}
```

c)

Testen Sie Ihre Implementierung mit der im Digicampus zur Verfügung gestellten Programmklasse `GarageMain.java`.