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 28.06.2024

Lösungsvorschlag zu Übungsblatt 9

Abgabe: 08.07.2024, 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`

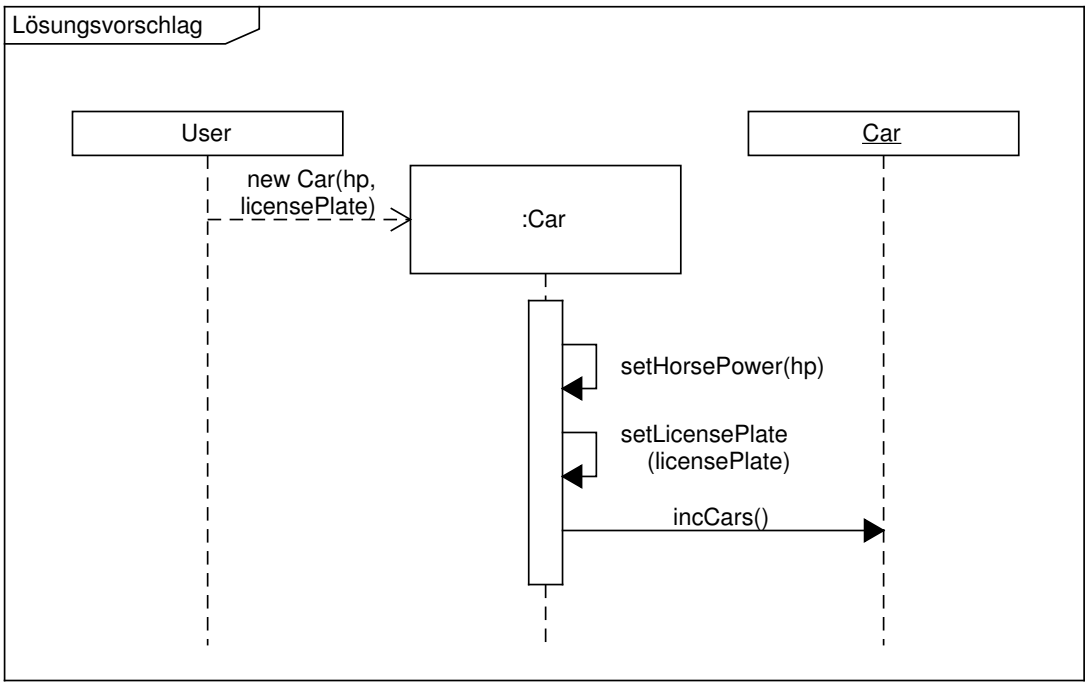
Aufgabe 33 ** *(Sequenzdiagramme modellieren, 25 Minuten)*

In den folgenden Teilaufgaben sollen Sie jeweils so detailliert wie möglich ein Sequenzdiagramm modellieren. Modellieren Sie dabei jeweils auch alle ineinander- und hintereinandergeschachtelten Operationsaufrufe. Sollten Anweisungen nicht in einem Sequenzdiagramm darstellbar sein, so ignorieren Sie diese.

a) *(*, 5 Minuten)*
Betrachten Sie den folgenden Programmcode:

```
public Car(int hp, String licensePlate) {
    setHorsePower(hp);
    setLicensePlate(licensePlate);
    Car.incCars();
}
```

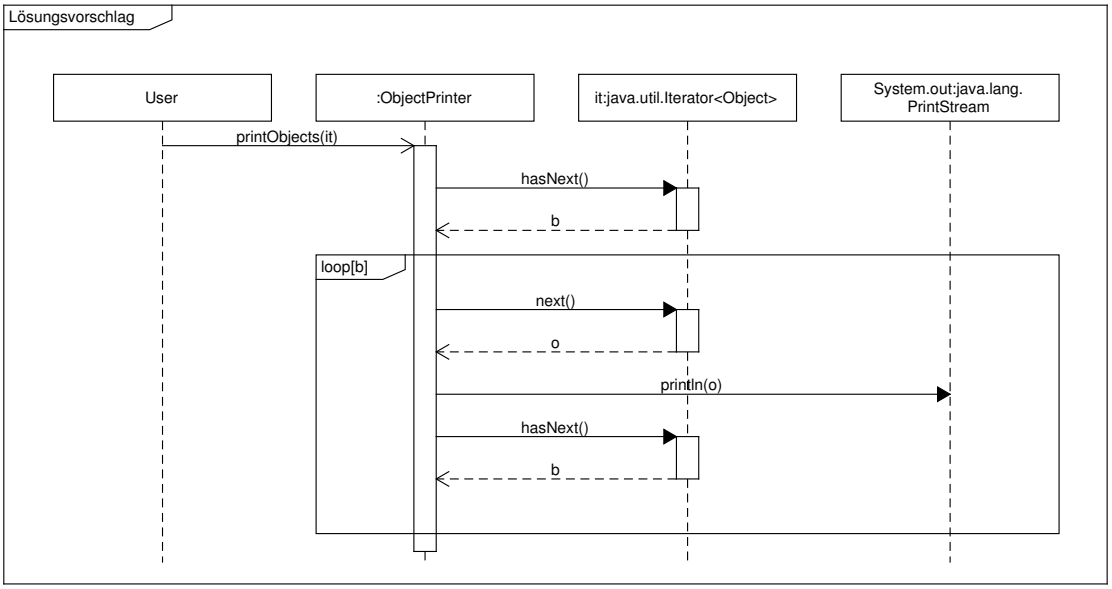
Stellen Sie Aufruf und Ausführung des Konstruktors so detailliert wie möglich als Sequenzdiagramm dar. Insbesondere ist jeder Methodenaufruf als separate Botschaft zu modellieren.



b) *(**, 10 Minuten)*
Betrachten Sie den folgenden Programmcode:

```
import java.util.Iterator;

public class ObjectPrinter {
    public void printObjects(Iterator<Object> it) {
        boolean b = it.hasNext();
        while (b) {
            Object o = it.next();
            System.out.println(o);
            b = it.hasNext();
        }
    }
}
```

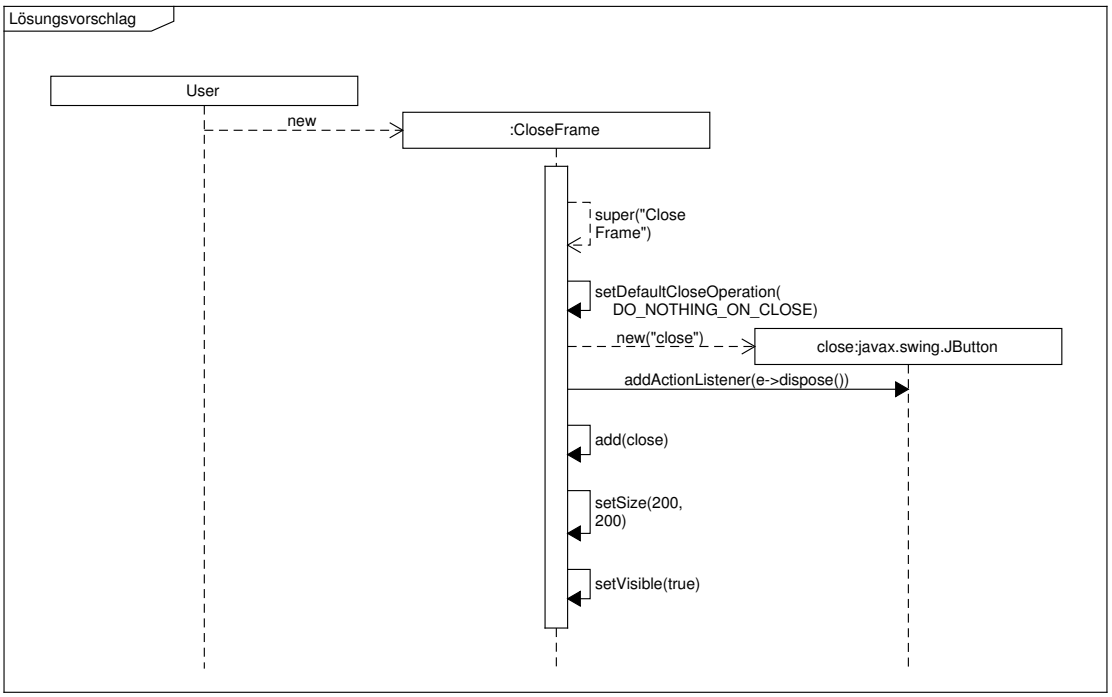
Stellen Sie Aufruf und Ausführung der Methode `printObjects` so detailliert wie möglich als Sequenzdiagramm dar. Insbesondere ist jeder Methodenaufruf als separate Botschaft zu modellieren.



Diskussion:

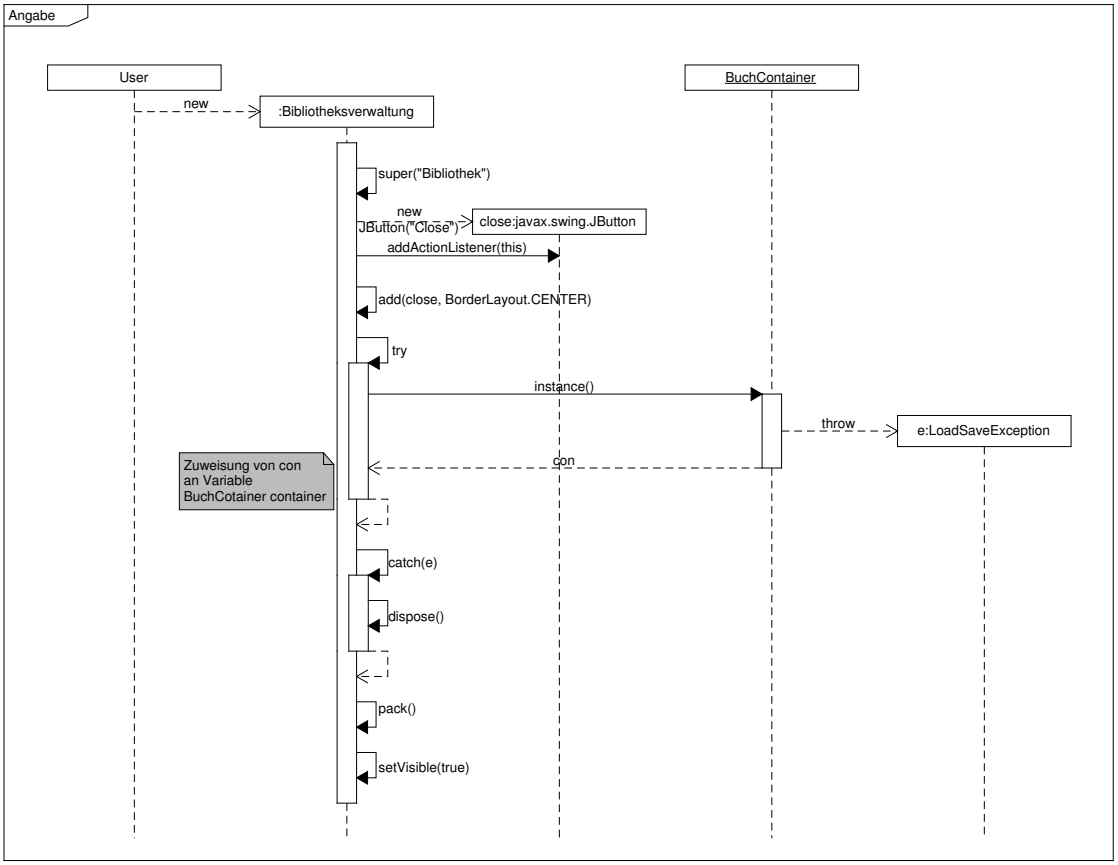
- Der Kommunikationspartner `System.out` dürfte auch ohne Klasse `PrintStream` angegeben werden.
- Die Aktionssequenz mit Rückantwort zu der Botschaft `println` kann, aber muss nicht gezeichnet werden.

c) *(**, 10 Minuten)*
Modellieren Sie durch ein Sequenzdiagramm den Aufruf und die Implementierung des Konstruktors eines sichtbaren Fensters. Das Fenster soll die Größe 200×200 und den Titel `Close Frame` haben. Es soll einen Button `close` enthalten, über den sich das Fenster schließen lässt. Das Fenster soll **nur** mit dem Button geschlossen werden können. Das Fenster soll keine weiteren Komponenten oder Funktionalitäten enthalten.



Aufgabe 34 * *(Sequenzdiagramme in Java implementieren, 16 Minuten)*

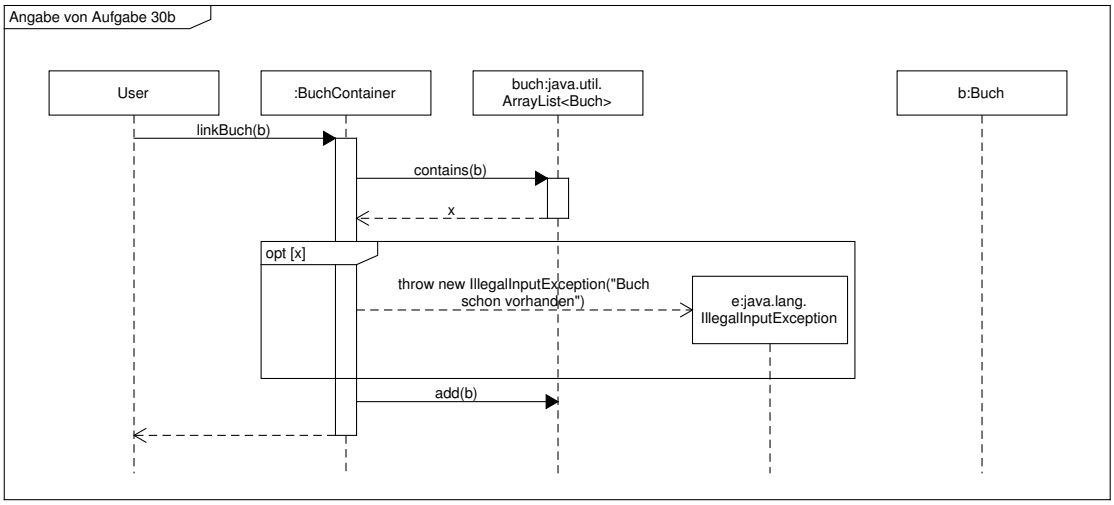
a) *(*, Alte Klausuraufgabe, 7 Minuten)*
Implementieren Sie in Java den Konstruktor einer Klasse `Bibliotheksverwaltung` gemäß dem folgenden Sequenzdiagramm.



Lösung:

```
public Bibliotheksverwaltung() {
    super("Bibliothek");
    JButton close = new JButton("Close");
    close.addActionListener(this);
    add(close, BorderLayout.CENTER);
    try {
        BuchContainer container = BuchContainer.instance();
    } catch(LoadSaveException e) {
        dispose();
    }
    pack();
    setVisible(true);
}
```

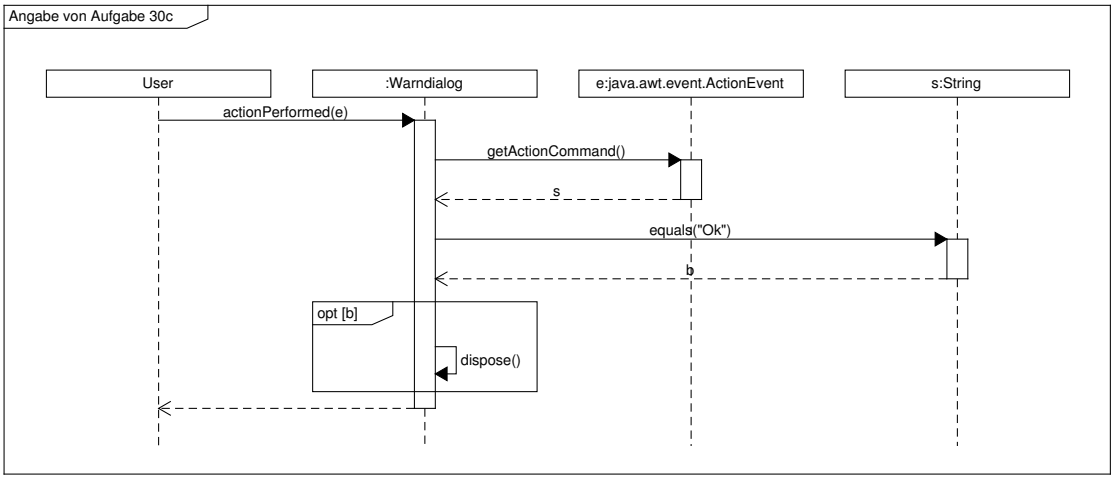
b) *(*, Alte Klausuraufgabe, 5 Minuten)*
Implementieren Sie in Java die Methode `linkBuch` einer Klasse `BuchContainer` gemäß dem folgenden Sequenzdiagramm:



Lösung:

```
public void linkBuch(Buch b) throws IllegalInputException {
    if (buch.contains(b)) {
        throw new IllegalInputException("Buch schon vorhanden");
    }
    buch.add(b);
}
```

c) *(*, 4 Minuten)*
Implementieren Sie in Java die Methode `actionPerformed` einer Fensterklasse `Warndialog` gemäß dem folgenden Sequenzdiagramm:



Lösung:

```
public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand().equals("Ok")) {
        dispose();
    }
}
```

Aufgabe 35 ** *(Container-Muster und JavaBeans, 20 Minuten)*

In dieser Aufgabe betrachten wir ein System zur Verwaltung von Reservierungen in einem Restaurant. Neben der Verwaltung der Tischnummer und dem Beginn und Ende einer Reservierung soll vor allem auch dafür gesorgt werden, dass sich eine neue Reservierung nicht mit den bisherigen Reservierungen überschneidet.
Dafür sind bereits Klassen `Reservation` und `ReservationException` vorgegeben. Diese sollen Sie nach folgenden Vorgaben so anpassen, erweitern und ergänzen, dass Reservierungen später leicht über eine Benutzeroberfläche modifiziert und erzeugt oder gelöscht werden können.
Bevor Sie beginnen, machen Sie sich zuerst mit den im Digicampus vorgegebenen Klassen `Reservation` und `ReservationException` vertraut.

a) *(**, 5 Minuten)*
Erweitern Sie die Klasse `Reservation` so, dass sie als Beans-Ereignisquelle fungieren kann. Alle Änderungen an Attributswerten sollen ein Ereignis auslösen. Dabei soll sowohl der alte als auch der neue Attributswert im Ereignis hinterlegt werden. Vergessen Sie nicht, auch eine Möglichkeit anzubieten, Abhörer hinzuzufügen und zu entfernen.

Lösung:

```
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class Reservation {
    private int tableNumber;
    private LocalDateTime start;
    private LocalDateTime end;
    private PropertyChangeSupport changes = new PropertyChangeSupport(this);


    public void setTableNumber(int tableNumber) {
        int oldValue = this.tableNumber;
        this.tableNumber = tableNumber;
        changes.firePropertyChange("tableNumber", oldValue, tableNumber);
    }


    public void setStart(LocalDateTime start) throws ReservationException {
        if (!checkStartEnd())
            throw new ReservationException("Start must be before end");
        LocalDateTime oldValue = this.start;
        this.start = start;
        changes.firePropertyChange("start", oldValue, start);
    }


    public void setEnd(LocalDateTime end) throws ReservationException {
        if (!checkStartEnd())
            throw new ReservationException("Start must be before end");
        LocalDateTime oldValue = this.end;
        this.end = end;
        changes.firePropertyChange("end", oldValue, end);
    }


    public void addPropertyChangeListener(PropertyChangeListener l) {
        changes.addPropertyChangeListener(l);
    }

    public void removePropertyChangeListener(PropertyChangeListener l) {
        changes.removePropertyChangeListener(l);
    }
}
```

Alle anderen Methoden bleiben unverändert.

b) *(**, 15 Minuten)*
Implementieren Sie eine Containerklasse für die Reservierungen, die

- das Singelton-Muster erfüllt,
- die `Iterable`-Schnittstelle für Reservierungen implementiert,
- Möglichkeiten bereitstellt, um Reservierungen hinzuzufügen oder zu löschen,
- als Beans-Ereignisquelle fungiert, wenn eine Reservierung hinzugefügt oder gelöscht wird, und
- sicherstellt, das keine überschneidenden Reservierungen im Container gespeichert werden (werfen Sie hier im Fehlerfall eine Instanz der gegebenen geprüften Ausnahmeklasse `ReservationException`).

Vergessen Sie nicht, auch eine Möglichkeit anzubieten, Abhörer hinzuzufügen und zu entfernen.
Lösung:

```
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.util.ArrayList;
import java.util.Iterator;

public class ReservationContainer implements Iterable<Reservation> {
    private ArrayList<Reservation> reservations;
    private static ReservationContainer unique = null;
    private PropertyChangeSupport changes = new PropertyChangeSupport(this);

    private ReservationContainer() {
        reservations = new ArrayList<>();
    }

    public static ReservationContainer getInstance() {
        if (unique == null)
            unique = new ReservationContainer();
        return unique;
    }

    @Override
    public Iterator<Reservation> iterator() {
        return reservations.iterator();
    }

    public void linkReservation(Reservation reservation) throws ReservationException {
        for (Reservation r : reservations) {
            if (r.collidesWith(reservation))
                throw new ReservationException("Reservations cannot overlap");
        }
        reservations.add(reservation);
        changes.firePropertyChange("reservations", null, reservation);
    }

    public void unlinkReservation(Reservation reservation) {
        if (reservations.contains(reservation)) {
            reservations.remove(reservation);
            changes.firePropertyChange("reservations", reservation, null);
        }
    }

    public void addPropertyChangeListener(PropertyChangeListener l) {
        changes.addPropertyChangeListener(l);
    }

    public void removePropertyChangeListener(PropertyChangeListener l) {
        changes.removePropertyChangeListener(l);
    }
}
```

c) *(**, 5 Minuten)*
Verwenden Sie die Klassen `ReservationMain` und `ReservationChangeListener`, um ihre Implementierung zu testen. Sie finden beide Klassen im Digicampus. Es sollte folgende Ausgabe entstehen:

```
 Reservations cannot overlap
Table 2, from Montag, 1. Juli, 19:00 to Montag, 1. Juli, 21:00
Reservation has been changed!
Source: Table 3, from Montag, 1. Juli, 19:00 to Montag, 1. Juli, 21:00
Property name: tableNumber
Old value: 2
New value: 3
```

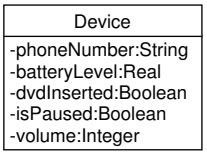
Erweitern Sie die Methode `propertyChange` der Klasse `ReservationChangeListener` so, dass sie auch auf die Ereignisse, die von Ihrer Containerklasse geworfen werden, reagiert. Es soll jeweils auf der Kommandozeile ausgegeben werden, ob eine Reservierung erstellt oder gelöscht wurde, und die betroffene Reservierung auf der Kommandozeile ausgegeben werden.

Lösung:

```
public void propertyChange(PropertyChangeEvent evt) {
    if (evt.getSource().getClass().equals(Reservation.class)) {
        System.out.println("Reservation has been changed!");
        System.out.println("Source:\t" + evt.getSource());
        System.out.println("Property name:\t" + evt.getPropertyName());
        System.out.println("Old value:\t" + evt.getOldValue());
        System.out.println("New value:\t" + evt.getNewValue());
    }
    if (evt.getSource().equals(ReservationContainer.getInstance())) {
        if (evt.getOldValue() == null) {
            System.out.println("New Reservation created:");
            System.out.println(evt.getNewValue());
        } else {
            System.out.println("Reservation deleted:");
            System.out.println(evt.getOldValue());
        }
    }
}
```

Aufgabe 36 ** *(SOLID-Prinzipien, 18 Minuten)*

a) *(**, 4 Minuten)*
Erfüllt das folgende UML-Klassendiagramm das *Single Responsibility Principle*? Begründen Sie Ihre Antwort.



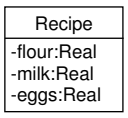
Nein, denn die Klasse `Device` enthält Attribute, die aus zwei unterschiedlichen Verantwortungsbereichen stammen: Die Attribute `phoneNumber` und `batteryLevel` stammen aus dem Verantwortlichkeitsbereich `Handy`, die Attribute `dvdInserted` und `isPaused` sowie `volume` aus dem Verantwortlichkeitsbereich `DVD-Spieler`.

b) *(**, 4 Minuten)*
Erfüllt der folgende Java-Programmcode das *Liskov Substitution Principle*? Begründen Sie Ihre Antwort.

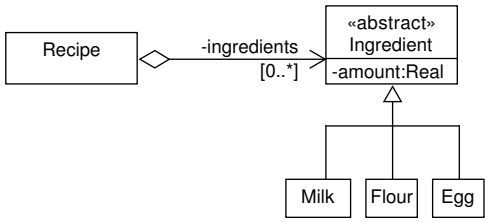
```
public class Car {
        private double tankLevel;

        public double refuel(double amount) {
                tanklevel += amount;
        }
}

public class ElectricCar extends Car {
        @Override
        public double refuel(double amount) {
                throws new IllegalOperationException("Electric cars can't be refueled");
        }
}
```

Nein, denn die Unterklasse `ElectricCar` überschreibt die Methode `refuel` der Klasse `Car` und fügt der Methode eine geworfene Ausnahme, die abgefangen werden muss, hinzu. Dadurch könnten Objekte des Typs `ElectricCar` nicht für beliebige Objekte des Typs `Car` zur Laufzeit des Programms eingesetzt werden.

c) *(**, 6 Minuten)*
Welches SOLID-Prinzip verletzt die folgende Modellierung? Begründen Sie Ihre Antwort und schlagen Sie eine Verbesserung vor.



Das Open-Closed-Prinzip wird verletzt, denn die Klasse `Recipe` ist nicht abgeschlossen bezüglich der Zutaten. Besser ist eine Modellierung wie folgt:



d) *(**, 4 Minuten)*
Erklären Sie, inwiefern die Implementierung der Ereignisbehandlung in der Java-API eine Umsetzung des *Dependency Inversion Principle* ist.

Damit ein Objekt A auf ein Ereignis reagieren kann, muss das ereignisauslösende Objekt B dieses Objekt A (das ereignisempfangende Objekt) nicht kennen, sondern nur wissen, dass es eine bestimmte Methode für die Reaktion auf das Ereignis hat. Diese Methode, um auf bestimmte Ereignisse zu reagieren, wird durch eine abstrakte Schnittstelle vorgegeben; die Schnittstelle wird vom Low-Level-Modul (Objekt A) implementiert; dem High-Level-Modul (Objekt B) ist nur die Abstraktion der Schnittstelle bekannt (DIP).