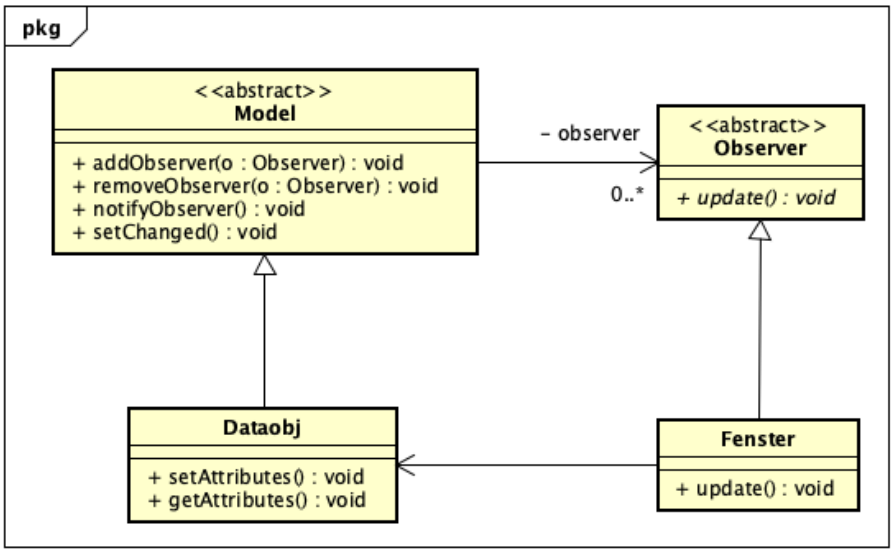
Software Engineering 1 Übungsblatt 9 Design (Pattern)

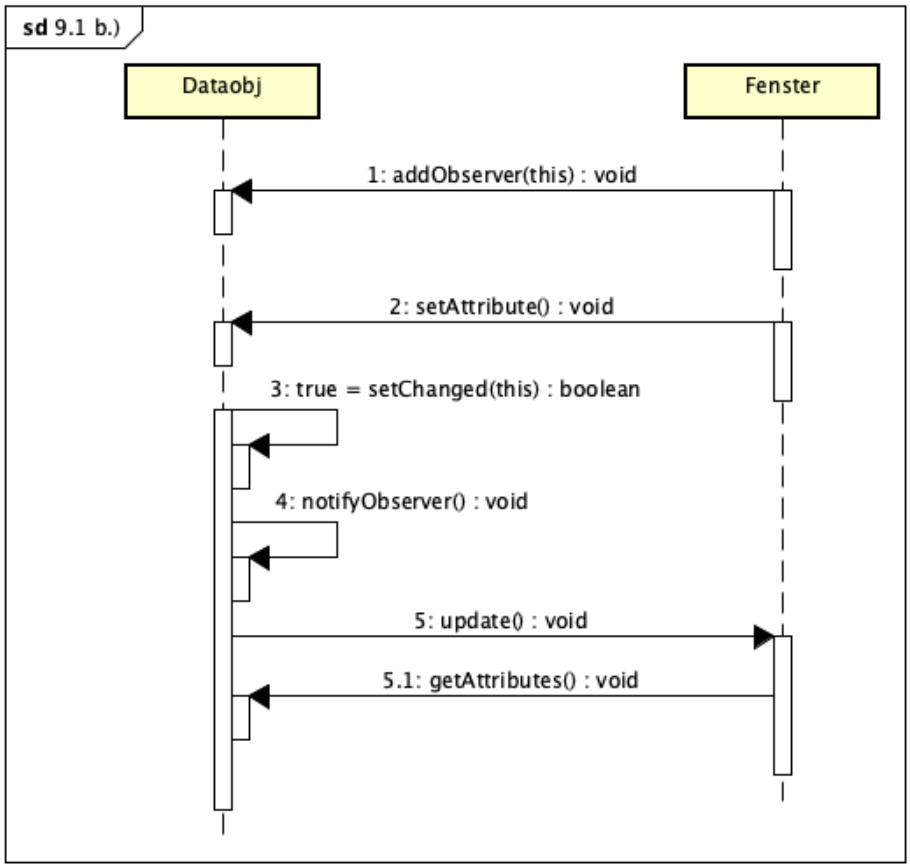
Aufgabe 9.1: Observer-Pattern – Pflichtaufgabe

Mehrere Fenster sollen dasselbe Datenobjekt anzeigen können. Sobald in einem der Fenster das Datenobjekt geändert wird, sollen in allen von der Änderung betroffenen Fenstern automatisch die geänderten Daten angezeigt werden.

a) Entwerfen Sie ein Klassendiagramm unter Verwendung des Observer-Patterns.

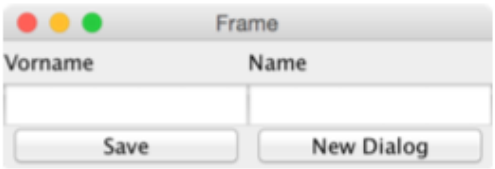


b) Zeichnen Sie ein Sequenzdiagramm, dass den Ablauf der Methodenaufrufe beim Registrieren und Benachrichtigen der Objekte darstellt.



c) Implementieren Sie eine Klasse `Person` mit den Attributen `name`, `vorname`.

d) Implementieren Sie eine Klasse `PersonWindow` als Beobachter (Observer) für Instanzen der Klasse `Person`. Innerhalb der Klasse `PersonWindow` sollen alle Eigenschaften der `Person`-Instanz in Textfeldern dargestellt und editiert werden.



Wenn der „`New Dialog`"-Button gedrückt wird, soll ein neues Fenster erzeugt werden. Wenn der „`Save`"-Button gedrückt wird, sollen die Werte aus den Textfeldern in die `Person`-Instanz übernommen werden. Anschließend sollen alle die `Person`-Instanz beobachtenden Fenster über die Änderungen informiert und die geänderten Daten angezeigt werden.

e) Schreiben Sie eine Treiber-Klasse (`main`-Klasse) mit der Sie mehrere Instanzen der Klasse `PersonWindow` erzeugen können, die alle dasselbe `Person`-Objekt anzeigen und bei Änderungen aktualisieren.

```java
public class Person implements Model{

    private String name;
    private String surname;
    private HashSet<Observer> observers = new HashSet<Observer>();
    @SuppressWarnings("unused")
    private boolean isChanged;

    public Person() {

    }

    public Person(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }

    public String getName() {
        return this.name;
    }
    public void setName(String name) {
        this.name = name;
        notifyObserver();
    }

    public String getSurName() {
        return this.surname;
    }
    public void setSurName(String surname) {
        this.surname = surname;
        notifyObserver();
    }

    @Override
    public void addObserver(Observer o) {
        this.observers.add(o);
    }

    @Override
    public void remoceObserver(Observer o) {
        this.observers.remove(o);
    }

    @Override
    public void notifyObserver() {
        for(Observer o : this.observers) {
            o.update();
        }
    }

    @Override
    public void setChanged() {
        this.isChanged = true;
    }
}
```

```java
public interface Model {

    public void addObserver(Observer o );

    public void remoceObserver(Observer o);

    public void notifyObserver();

    public void setChanged();
}
```

```java
public interface Observer {
    public void update();
}
```

```java
public class PersonWindow  implements Observer{
    String testFielName;
    Person p;

    public PersonWindow() {

    }

    public void setPerson(Person p) {
        this.p = p;
    }

    public void update() {
        this.testFielName = p.getName();
        System.out.println(this + ": " + p.getName());
    }

}
```

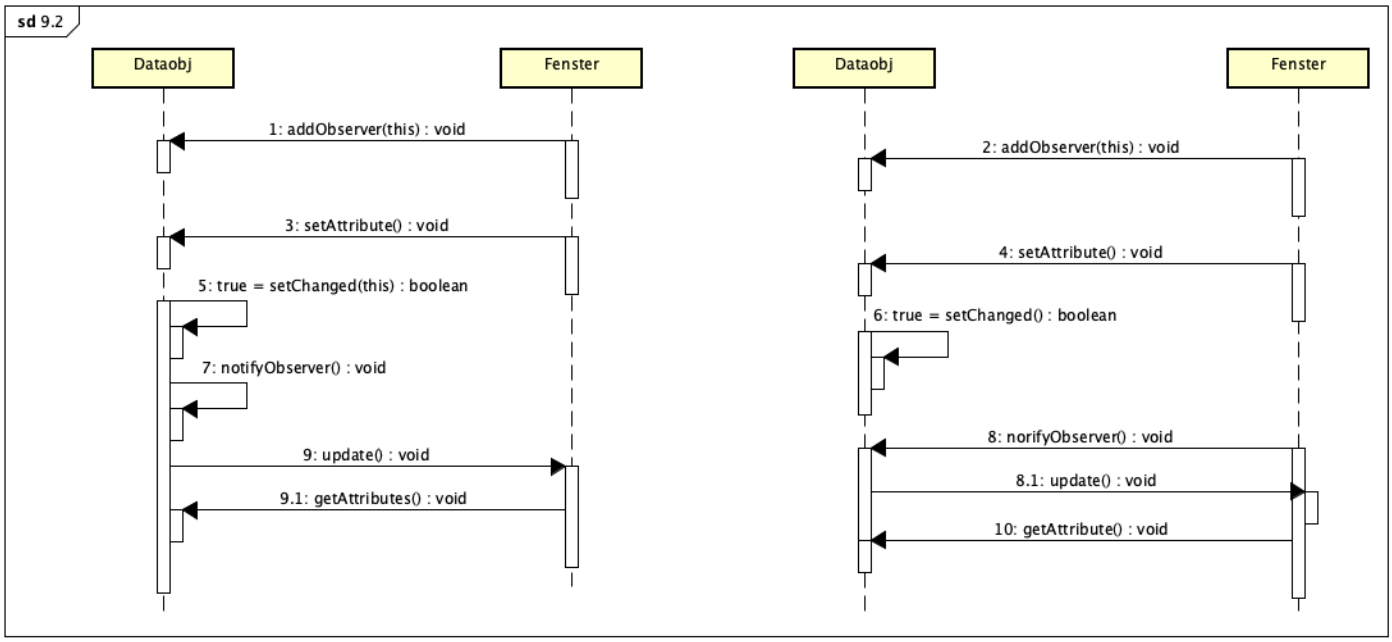
```java
public class main {

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        Person p = new Person("Deti", "Lushaj");

        PersonWindow f1 = new PersonWindow();
        PersonWindow f2 = new PersonWindow();
        PersonWindow f3 = new PersonWindow();


        f1.setPerson(p);
        f2.setPerson(p);
        f3.setPerson(p);

        p.addObserver(f1);
        p.addObserver(f2);
        p.addObserver(f3);

        p.setName("Detijon");

        boolean change = true;
        do {
            System.out.println("Would you like to Change the name again? (y)");

            if(input.nextLine().equals("y")) {
                System.out.println("Please enter the new name: ");
                p.setName(input.nextLine());
            } else {
                change = false;
            }


        } while (change);

    }
}
```

Aufgabe 9.2: Observer-Pattern – Auslösen von `notifyObservers()`

Im Observer-Pattern ruft die Methode `notifyObservers()` nach Änderungen der Model-Daten für alle Observer deren `update()`–Methode auf. Es existieren zwei alternative Stellen im Kontrollfluss, um `notifyObservers()` aufzurufen:

1. Das beobachtete Datenobjekt (ConcreteModel) ruft nach einer Änderung `notifyObservers()` auf.
2. Der Beobachter (ConcreteObserver) ruft nach einer Änderung `notifyObservers()` auf.

Zeichnen Sie jeweils ein Sequenzdiagramm, das die Abfolge der Methodenaufrufe des Benachrichtigungsmechanismus darstellt. Welche Vor- und Nachteile haben jeweils die beiden Alternativen?



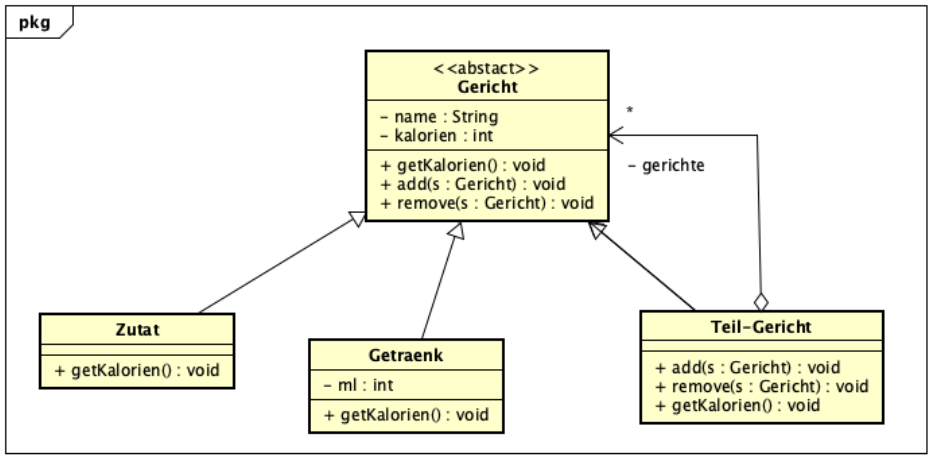
1. Variante	**2. Variante**
Vorteile • Benachrichtigungen können nicht vergessen werden	**Vorteile** • Benachrichtigungen können zusammengefasst werden
Nachteile • Es können sehr viele Benachrichtigungen entstehen	**Nachteile** • Benachrichtigungen müssen manuell aufgerufen werden und darf dementsprechend nicht vergessen werden

Aufgabe 9.3: Kompositum-Pattern

Annahme: Der Kaloriengehalt eines Speisegerichts (Vor-/Nachspeise, Hauptgang, Getränke etc.) errechnet sich aus der Summe der Kaloriengehalte sämtlicher Zutaten und Getränke.
Entwickeln Sie unter Anwendung des **Kompositum-Patterns** ein Klassenmodell, mit dem man die Zusammensetzung eines (Speise-)Gerichts abbilden kann.

- Ein (Speise-)Gericht (z.B. Pizza Vier Jahreszeiten mit Salat und Vanille-Eis mit Sahne als Dessert) besteht aus einzelnen Zutaten (z.B. Tomaten, Zwiebeln, Käse) und Getränken (z.B. O-Saft), jeweils mit Namen und Kalorien, oder aus einem Teilgericht (z.B. Salat, Dessert, Teig).
- Ein Teilgericht (z.B. Teig) besteht aus einzelnen Zutaten (z.B. Mehl, Salz, Pfeffer) und kann wiederum Teilgerichte enthalten (z.B. Spezialwürzmischung).
- Für jede einzelne Zutat bzw. Getränk kann der Kaloriengehalt angegeben werden.
  - Bei einer Zutat ist der Kaloriengehalt als Wert gegeben.
  - Bei einem Getränk errechnet sich der Kaloriengehalt aus den Kalorien pro 100ml mal der Getränkegröße (also mal 3 bei einem 300ml Glas).
  - Der Kaloriengehalt eines Teilgerichts errechnet sich aus der Summe der Kalorien der enthaltenen Zutaten, Getränke und Teilgerichte.

a) Erstellen Sie ein UML-Klassenmodell mit seinen erforderlichen Beziehungen.
b) Spezifizieren Sie die erforderlichen Methoden, um ein Gesamtgericht zusammenzustellen und den Gesamtkaloriengehalt zu bestimmen.
c) Implementieren Sie das Klassendiagramm in Java und schreiben Sie ein Hauptprogramm, das Ihre Klassen testet!



Entweder Methoden in Gericht leer oder nur in Teilgericht

- AddGericht in Gericht leer
  - Vorteil: es muss nicht zwischen atomaren Klassen und Containern, also Teilgericht unterschieden werden
  - Nachteil: atomaren Klassen haben diese Methoden, auch wenn die nichts tun
- Nur in Teilgericht
  - Vorteil: dann haben die atomaren Klassen nicht diese Methoden
  - Nachteil: es muss zwischen den beiden unterschieden werden

Bei add() und remove() Änderungen mit extra Attribut nicht sinnvoll, weil nicht alle Änderungen berücksichtigt werden von der Hierarchie, hier bezüglich Kalorien addieren bzw. subtrahieren. Oben in der Hierarchie berücksichtigen nicht von unten hinzugefügten Sachen
→ extra Methode, wo durch alle Kinder iteriert wird notwendig wie in getKalorien()

Nicht Teilgericht Beziehung mit 1 wäre Decorator-Pattern => keine Hierarchie / Container, sondern Kette. Deswegen immer Stern

```java
public class Gericht {

    private String name;
    private int cal;

    public Gericht() {
    }
    public Gericht(String n, int c) {
        this.name = n;
        this.cal = c;
    }

    public void addGericht(Gericht g) {}

    public void removeGericht(Gericht g) {}

    public String getName() {
        return name;
    }
    public int getCalorien() {
        return cal;
    }

}
```

```java
public class Teilgericht extends Gericht {

    private HashSet<Gericht> gerichte = new HashSet<>();

    public Teilgericht() {
    }

    @Override
    public void addGericht(Gericht g) {
        this.gerichte.add(g);
    }

    @Override
    public void removeGericht(Gericht g) {
        this.gerichte.remove(g);
    }

    @Override
    public int getCalorien() {
        int gescal = 0;
        for(Gericht g : gerichte) {
            gescal += g.getCalorien();
        }
        return gescal;
    }

}
```

```java
public class Zutat extends Gericht {

    public Zutat(String s, int c) {
        super(s,c);
    }
}
```

```java
public class Getraenk extends Gericht{

    public Getraenk(String s, int c) {
        super(s,c);
    }

}
```