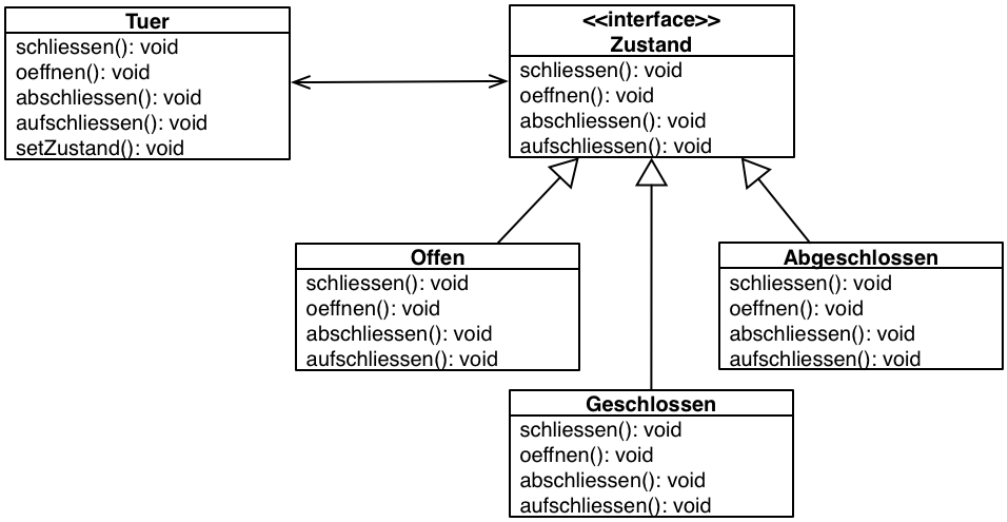
Aufgabe State Pattern: Tür (siehe [Siebler14, Kap. 6, S.75ff]):

Zustandswechsel:

1. durch bidirektionale Bez. kann der konkrete Zustand direkt den Folgezustand setzen (oder den aktuellen Zustand unverändert lassen)
2. Alternative nur unidirektionale Beziehung von Tuer zu Zustand und alle zustandsabhängigen Methoden geben den richtigen (Folge-)Zustand zurück; dann muss allerdings Tuer den Folgezustand aktiv setzen




```
public class Tuer {
    private Zustand zustand = new Offen(this);

    public void setZustand(Zustand zustand) {
        this.zustand = zustand;
    }

    public void oeffnen() {
        zustand.oeffnen();
    }
    public void schliessen() {
        zustand.schliessen();
    }
    public void abschliessen()  {
        zustand.abschliessen();
    }

    public void aufschliessen() {
        zustand.aufschliessen();
    }
```

```
public abstract class Zustand {
    public final Tuer TUER;

    Zustand(Tuer tuer) {
        this.TUER = tuer;
    }

    abstract void oeffnen();
    abstract void schliessen();
    abstract void abschliessen();
    abstract void aufschliessen();

    public String toString() {
        return this.getClass().getName();
    }
}
```

```
public class Abgeschlossen extends Zustand {

    Abgeschlossen(Tuer tuer) {
        super(tuer);
    }

    public void oeffnen() {
        System.out.println("Die Tuer ist verschlossen und kann nicht geoeffnet werden.");
    }

    public void schliessen() {
        System.out.println("Die Tuer ist bereits abgeschlossen.");
    }

    public void abschliessen()  {
        System.out.println("Die Tuer ist bereits abgeschlossen und kann kein zweites Mal
        abgeschlossen werden.");
    }

    public void aufschliessen() {
        System.out.println("Die Tuer wird aufgeschlossen.");
        TUER.setZustand(new Geschlossen(TUER));
    }
}
```

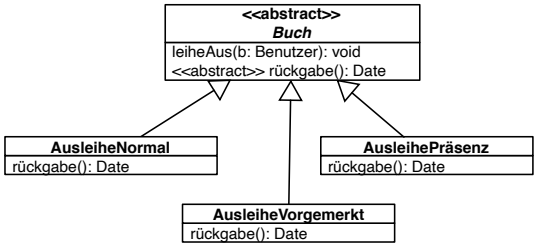
Bibliothekssystem Naive Implementierung

Implementierung ohne Pattern

```
class Buch {
     void leiheAus(Benutzer b) {....
        switch(getType()){
         case "NORMAL":
                 frist = this.rückgabeNormal();
         case "VORGEMERKT":
                 frist = this.rückgabeVorgemerkt();
         case "PRÄSENZ":
                 frist = this.rückgabePraesenz();
        }
        ....
    }
}
```


Bibliothekssystem mit Template

Lösung - Beispiel



Implementierung mit Template Pattern

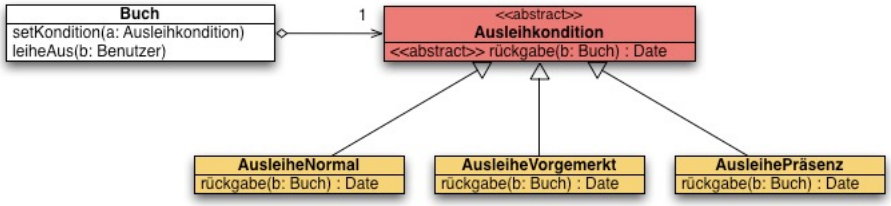
```
abstract class Buch {
        void leiheAus(Benutzer b){....
                frist = this.rückgabe();....  }

        abstract Date rückgabe();
}

class AusleiheNormal extends Buch {
        Date rückgabe(){.... //Berechnung Ausleihdatum
           }
}
```

Bibliothekssystem mit Strategie

Lösung - Beispiel



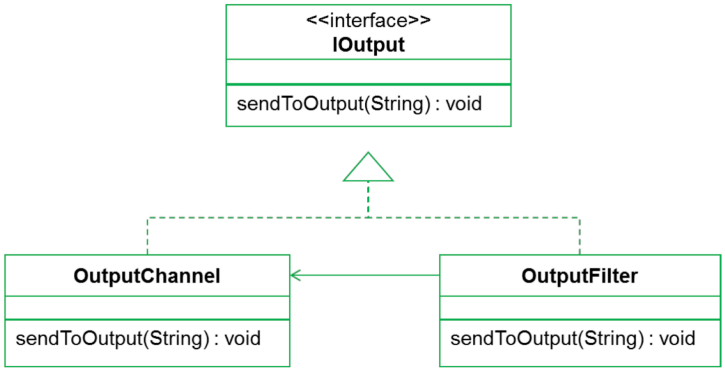
Implementierung mit Strategy-Pattern

```
class Buch {
        Ausleihkondition mKondition;

        void leiheAus(Benutzer b){....
                frist = mKondition.rückgabe(this);
                ....
        }
}
```

Aufgabe: Pattern aus Source Code
Gegeben sei folgender Java-Quellcode:
...

a) Erstellen Sie das Klassendiagramm, das den obigen Quellcode beschreibt.



b) Welches Entwurfsmuster wird durch den obigen Quellcode implementiert?
- Proxy-Pattern
- halb korrekt: Decorator (obwohl es keine zusätzliche Funktionalität gibt und keine Aggregation eines Obertyps, wie sie für Decorator charakteristisch sind).

c) Für das Entwurfsmuster liefert die GoF-Darstellung bestimmte Rollenbezeichnungen/ Klassennamen (siehe Vorlesungsfolien). Ordnen Sie die GoF-Rollennamen den obigen Klassen zu.
- IOutput - Object, Subject
- OutputChannel - RealObject, RealSubject
- OutputFilter - Proxy