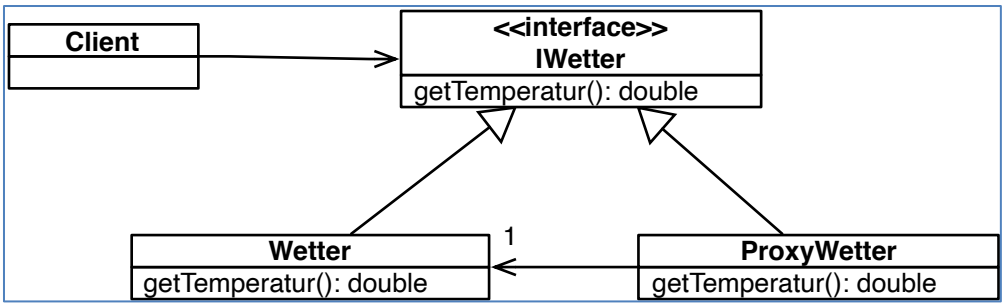
Aufgabe 3.1: Proxy Pattern Wetterstation (siehe [Balzert11, Kap. 8.7, S.84/85]):




```
public interface IWetter
{
public double getTemperatur();
}
```

```
1  public class ProxyWetter implements IWetter {
2      Wetter wetterstation;
3      public ProxyWetter() {
4          wetterstation = new Wetter();
5      }
6
7      public double getTemperatur() {
8          double tempInFahrenheit = wetterstation.getTemperatur();
9          return (tempInFahrenheit - 32 ) * 5 / 9;
10     }
11 }
```

```
public class Wetter implements IWetter
{
    public Wetter() {
    }

    public double getTemperatur() {
        return (Math.floor(Math.random() * 101));
    }
}
```

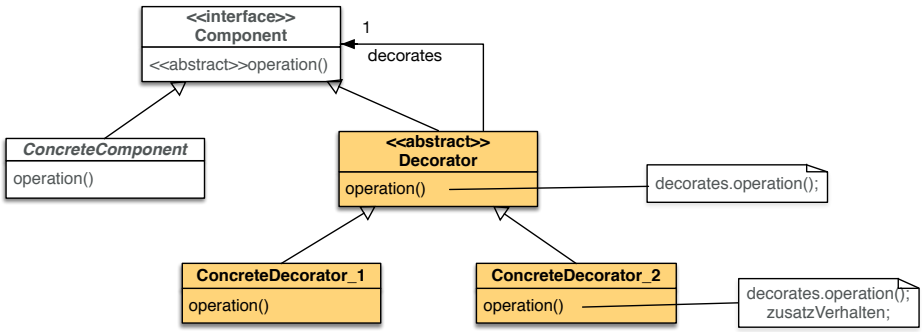
```
public class ClientWetter {
    public static void main(String[] args)  {
        IWetter wetterstation = new ProxyWetter();
        System.out.println("Aktuelle Temperatur 1: " + wetterstation.getTemperatur());
        System.out.println("Aktuelle Temperatur 2: " + wetterstation.getTemperatur());
    }
}
```

Aufgabe 2: Proxy Pattern vs. Decorator Pattern

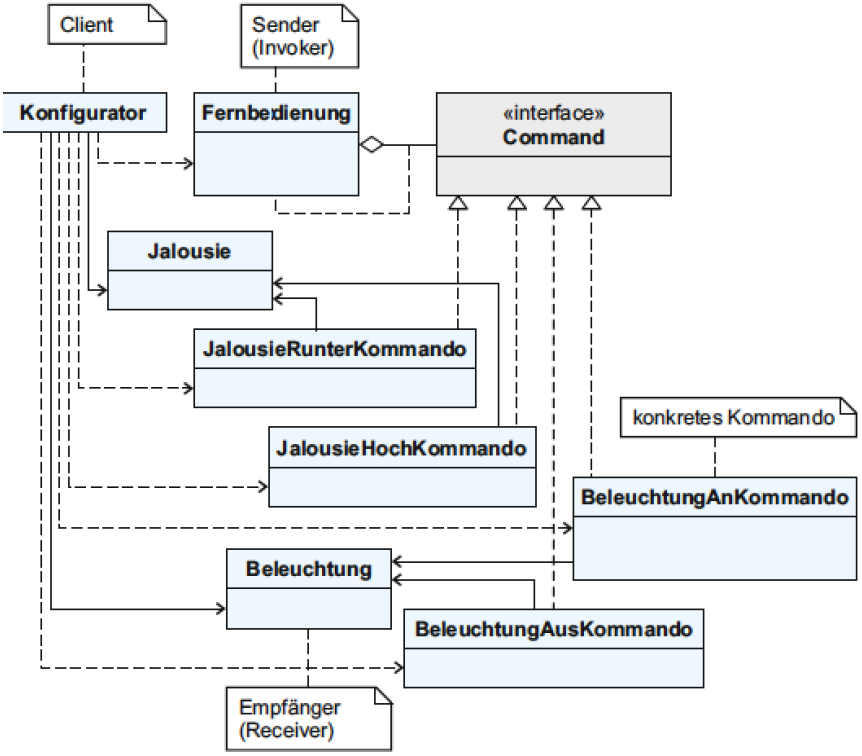
Das Proxy Pattern und das Decorator Pattern erweitern jeweils Objekte um Zuständigkeiten. Welche Gemeinsamkeiten und welche Unterschiede besitzen die beiden Pattern?

- Gemeinsamkeiten:
    - Realisierung der Pattern ähneln sich
        - eigentliches Objekt und Proxy/Decorator implementieren das gleiche Interface
        - Proxy/Decorator referenziert ein anderes Objekt, ohne zu wissen, ob es sich dabei um ein weiteres Proxy/Decorator oder um das gewünschte Objekt handelt
        - bei beiden Delegation von Aufruf an Original-Objekt, das Original-Objekt bleibt dabei unverändert
- Unterschiede:
    - unterschiedliche **Ziele** der Pattern:
        - Proxy kontrolliert den Zugang zum eigentlichen Objekt, schränkt den Zugang ggf. ein; mehr Implementierungseigenschaft
        - Decorator erweitert ein Objekt um Funktionalität, ohne das Objekt selbst zu verändern; Decorator erlaubt Kette von Referenzen; dynamisch beliebig viele Objekte; mehr konzeptionelle Eigenschaft

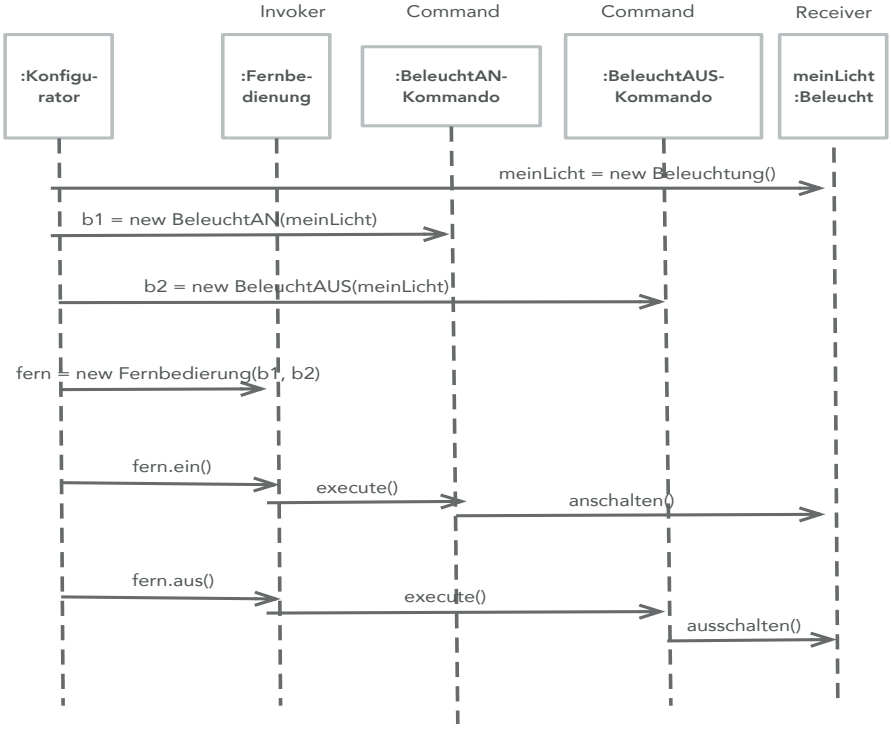
Decorator:



Aufgabe 3 Command Pattern Fernbedienung (siehe [Balzert11, Kap. 8.6, S.76ff]):



Sequenzdiagramm




```
class Jalousie { //Empfänger (Receiver)

    public void herunterfahren() {
        System.out.println("Jalousie wird heruntergefahren");
    }
    public void hochfahren() {
        System.out.println("Jalousie wird hochgefahren");
    }
}
```

```
//Schnittstelle Command
public interface Command {
    public void execute( );
}
```

```
//Konkretes Kommando
class JalousieHochKommando implements Command {
    private Jalousie meineJalousie;
    public JalousieHochKommando(Jalousie meineJalousie) {
        this.meineJalousie = meineJalousie;
    }
    public void execute( ) {
        meineJalousie.hochfahren( );
    }
}
```

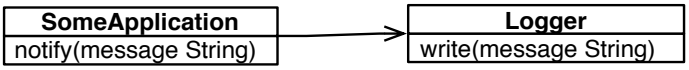
```
//Aufrufer des Kommandos (Invoker)
class Fernbedienung {
    private Command einKommando, ausKommando;
    public Fernbedienung(Command einKommando, Command ausKommando) {
        this.einKommando = einKommando;
        this.ausKommando = ausKommando;
    }
    void ein( ) {
        einKommando.execute( );
    }
    void aus( ) {
        ausKommando.execute( );
    }
}
```

```
//Client
public class Konfigurator {
    public static void main(String[] args) {
        Beleuchtung meinLicht = new Beleuchtung( );

        //meinLicht wird dem BeleuchtungAnKommando zugeordnet
        BeleuchtungAnKommando meinLichtAn = new BeleuchtungAnKommando(meinLicht);
        BeleuchtungAusKommando meinLichtAus = new BeleuchtungAusKommando(meinLicht);

        //Zuordnung von meinLicht zu der Fernbedienung
        Fernbedienung meineFernbedienungKnopf1 = new Fernbedienung(meinLichtAn,meinLichtAus);
        meineFernbedienungKnopf1.ein( );
        meineFernbedienungKnopf1.aus( );

        Jalousie meineJalousie = new Jalousie( );
        JalousieHochKommando meineJalousieHoch = new JalousieHochKommando(meineJalousie);
        JalousieRunterKommando meineJalousieRunter = new JalousieRunterKommando(meineJalousie);
        Fernbedienung meineFernbedienungKnopf2 = new Fernbedienung(meineJalousieHoch,
        meineJalousieRunter);
        meineFernbedienungKnopf2.ein( );
        meineFernbedienungKnopf2.aus( );

        //Ohne Kommando-Muster (die Schnittstelle pro Gerät muss bekannt sein)
        meinLicht.anschalten();
        meinLicht.ausschalten();
        meineJalousie.herunterfahren();
        meineJalousie.hochfahren();
    }
}
```

Aufgabe: Dependency Injection (DI)

Traditionell waren Objekte selber für Abhängigkeiten (Erzeugung von Objekten und Verwaltung der Referenzen) verantwortlich. Durch DI wird die Steuerung der Abhängigkeiten umgedreht (Inversion of Control-Prinzip). Inversion of Control ist Prinzip und DI die konkrete Methode.

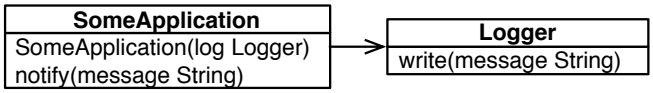
(a) SomeApplication ist von Logger abhängig.



(b)

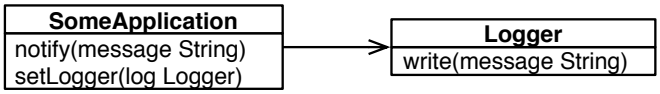
Constructor Injection: hält Referenz permanent und nicht änderbar

```
class SomeApplication{
    Logger log = null;

    public SomeApplication(Logger concreteLogger){
        this.log = concreteLogger;
    }
    public void notify(String message){
        log.write(message);
    }
}
```



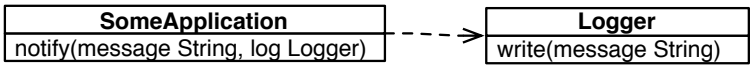
Setter Injection: hält Referenz permanent und zur Laufzeit änderbar

```
class SomeApplication{
    Logger log = null;

    public void notify(String message){
        log.write(message);
    }

    public void setLogger(Logger concreteLogger){
        this.log = concreteLogger;
    }
}
```



Method Injection: hält die Referenz nicht

```
class SomeApplication{

    public void notify(Logger log, String message){
        log.write(message);
    }

}
```



(c) Durch Einführung eines Interface für Logger