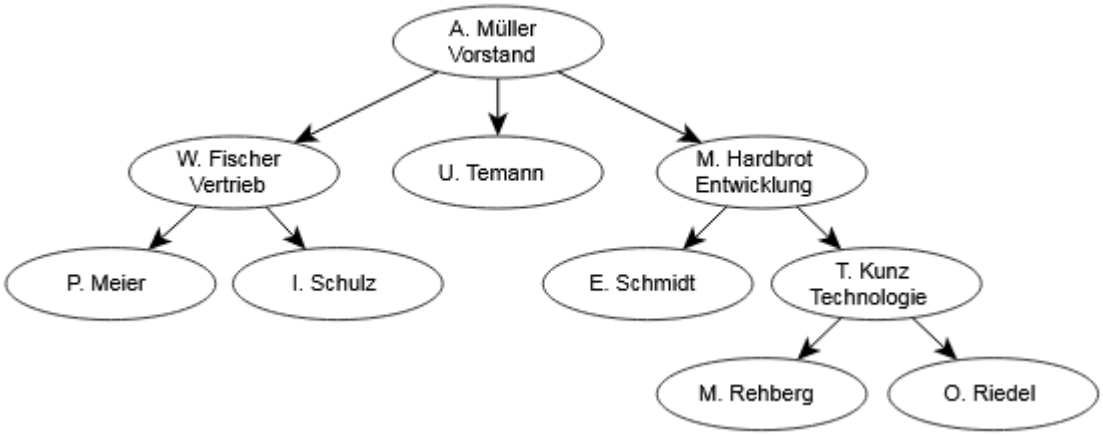
Composite Entwurfsmuster

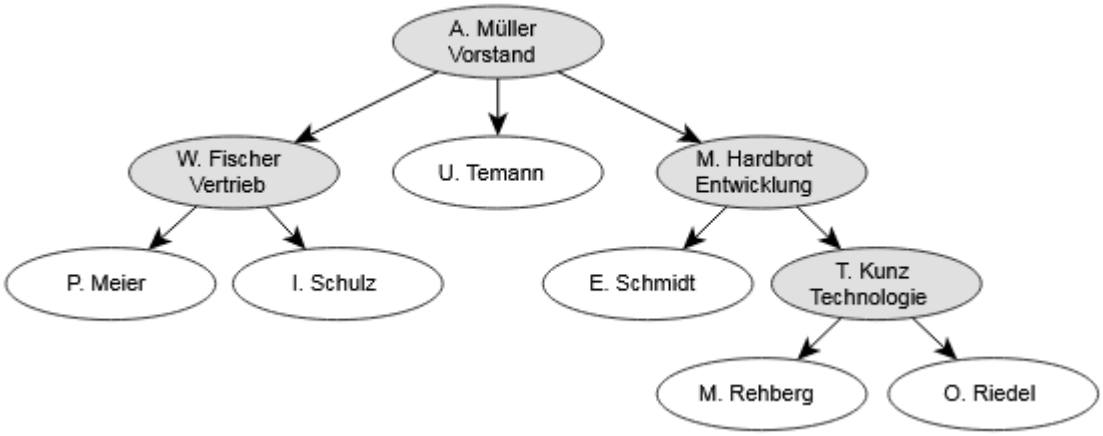
Das Composite Entwurfsmuster ermöglicht eine verschachtelte (Baum)Struktur einheitlich zu behandeln, unabhängig davon, ob es sich um ein atomares Element oder um ein Behälter für weitere Elemente handelt. Der Client kann elegant mit der Struktur arbeiten

Einführung

Wir wollen die Mitarbeiterhierarchie der Firma XY modellieren. Der Hierarchiebaum sieht folgendermaßen aus:



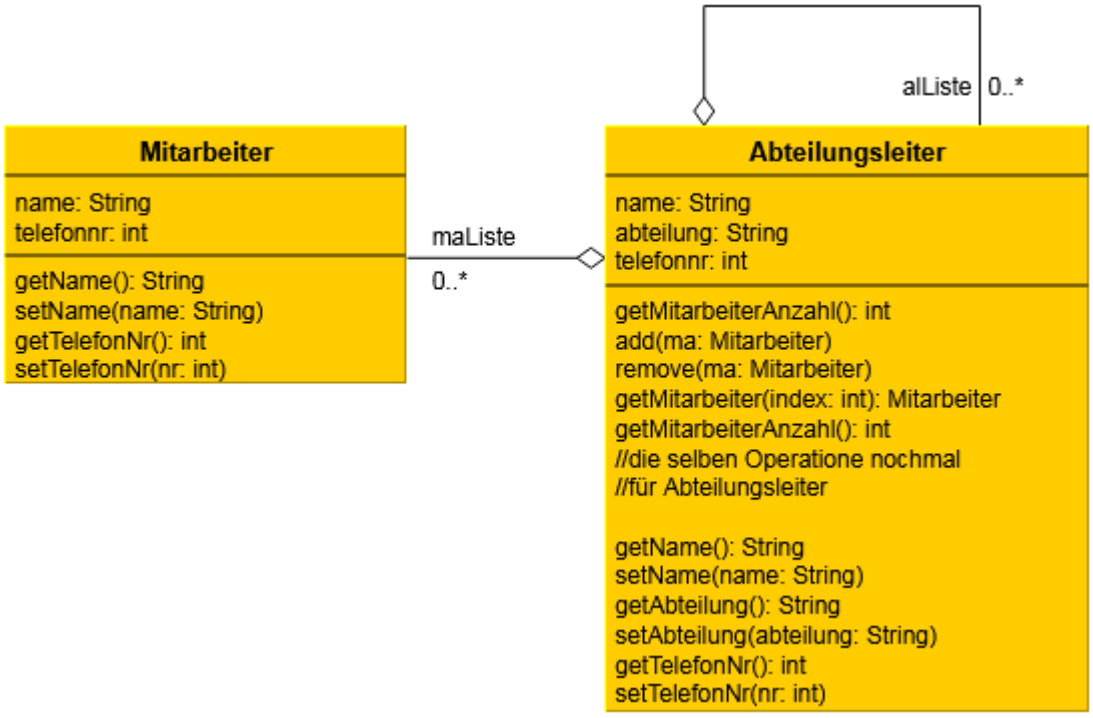
Offensichtlich gibt es "normale" Mitarbeiter und Abteilungsleiter, die andere Mitarbeiter (oder wiederum Abteilungsleiter) unter sich haben.



Welche Anforderungen werden an unser System gestellt?

- Jeder Mitarbeiter und Abteilungsleiter besitzt einen Namen und eine Telefonnummer (getName(), getTelefonnummer()). Darüber hinaus soll jeder Abteilungsleiter den Namen seiner Abteilung kennen (getAbteilung())
- Es sollen Operationen bereitgestellt werden, um Mitarbeiter in den Baum einzuordnen, zu holen oder zu entfernen (add(), getMitarbeiter(), remove()).
- Jeder Abteilungsleiter soll die Anzahl der Mitarbeiter/Abteilungsleiter in seiner Abteilung ausgeben können (getMitarbeiterAnzahl()).
- Die gesamte Hierarchie soll textuell ausgedruckt werden können (print() - dazu später mehr).

Ein erster naiver Entwurf ist folgender:



Wonach schreit dieser Entwurf förmlich? Genau, nach Vererbung. Doch bevor wir den Entwurf reparieren, wollen wir uns seine Schlechtigkeit mit all den Nachteilen in vollen Zügen anschauen.

- Codedopplung. Die Attribute für den Namen und die Telefonnummer, sowie die dazugehörigen Getter und Setter sind absolut redundant. Inkonsistenzen und aufwändige Wartbarkeit treten mit an 1 grenzender Wahrscheinlichkeit auf.

- Fallunterscheidungen und enge Kopplung. Sowohl der Client als auch der Abteilungsleiter muss in seinem Code ständig zwischen Mitarbeitern und Abteilungsleitern unterscheiden. Dies erzeugt eine Menge an schwer wartbaren Code und verhindert Allgemeingültigkeit. Client und Abteilungsleiter sind hart an Implementierungen gekoppelt. Damit widersprechen wir einem wichtigem OO-Entwurfsprinzip:

> *Stütze dich auf Abstraktion, nie auf konkrete Klassen.*

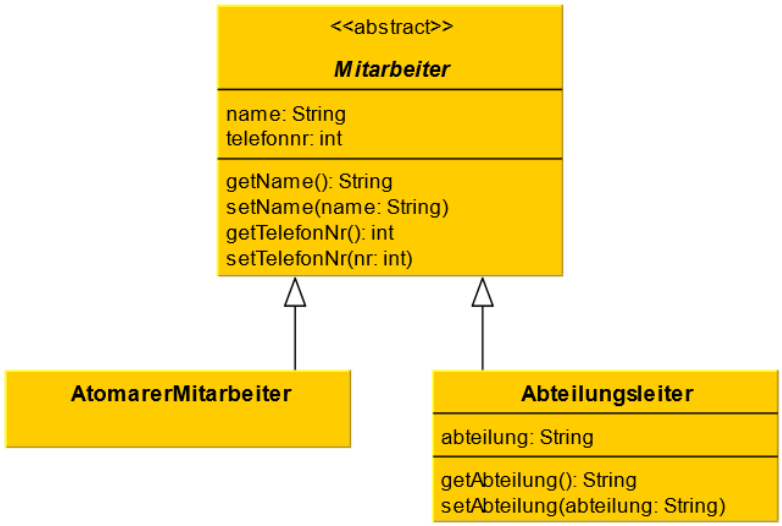
Das sei an einem Beispiel verdeutlicht. Wie würde der Client über den Baum wandern (traversieren), um die Gesamthierarchie auszugeben? Er müsste fortwährend zwischen Mitarbeitern und Abteilungsleitern differenzieren und sie getrennt behandeln:

Schlechte Ausgabe der Hierarchie:

```
private void print(Abteilungsleiter leiter) {
    for(int i= 0; i < leiter.getMitarbeiterAnzahl(), i++){
        Mitarbeiter ma = leiter.getMitarbeiter(i);
        System.out.println(ma.getName()+ ". Abteilung: "+ma.getAbteilung());
    }

    for(int i= 0; i < leiter.getAbteilungsleiterAnzahl(), i++){
        Abteilungsleiter al = leiter.getAbteilungsleiter(i);
        System.out.println(al.getName()+" ist Leiter von "+al.getAbteilung());
        print(al);//Rekursiver Aufruf
    }
}
```

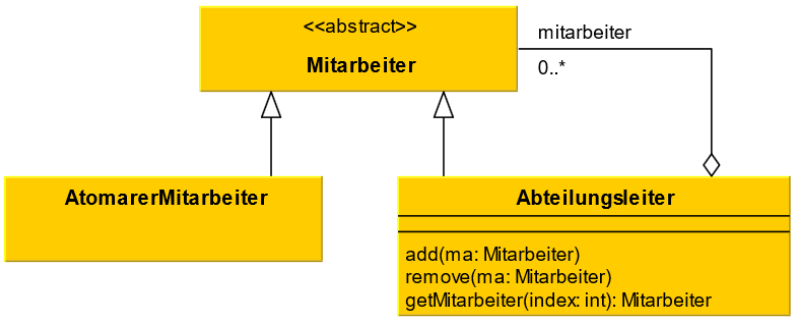
Extrem fehleranfällig, unschön und unflexibel. Jede Operation über der gesamten Hierarchie wird zu einer Qual.

Der obige Entwurf ist also schlecht. Wie könnte eine bessere Modellierung aussehen?



Zunächst führen wir Abstraktion in Form der Superklasse Mitarbeiter ein, von der sowohl normale Mitarbeiter als auch Abteilungsleiter erben. In diesem Atemzug überarbeiten wir unser Vokabular. Fortan ist jeder Mitarbeiter (ob nun normaler Mitarbeiter oder Abteilungsleiter) auch ein Mitarbeiter - so wie in der Wirklichkeit. Dies zeigt, wie uns die Realität bei der OO-Modellierung helfen kann. Mitarbeiter, die keine Abteilung leiten, werden als atomare Mitarbeiter bezeichnet.

Wenden wir uns nun dem Abteilungsleiter zu. Muss er wirklich zwischen ihm unterstellten atomaren Mitarbeitern und Abteilungsleitern unterscheiden? Würde man nicht viel mehr Flexibilität gewinnen, wenn er sich stattdessen auf die Abstraktion Mitarbeiter stützt? Auf diese Weise kann er atomare Mitarbeiter und Abteilungsleiter gleichbehandeln. Es ist ihm gleich, ob ihm unterstellte Mitarbeiter selbst Abteilungsleiter sind oder nur normale Mitarbeiter.



Dies ist besonders bei den Verwaltungsmethoden interessant und vereinfacht die Implementierung enorm, da die Aufrufe einfach an die Liste delegiert werden können.

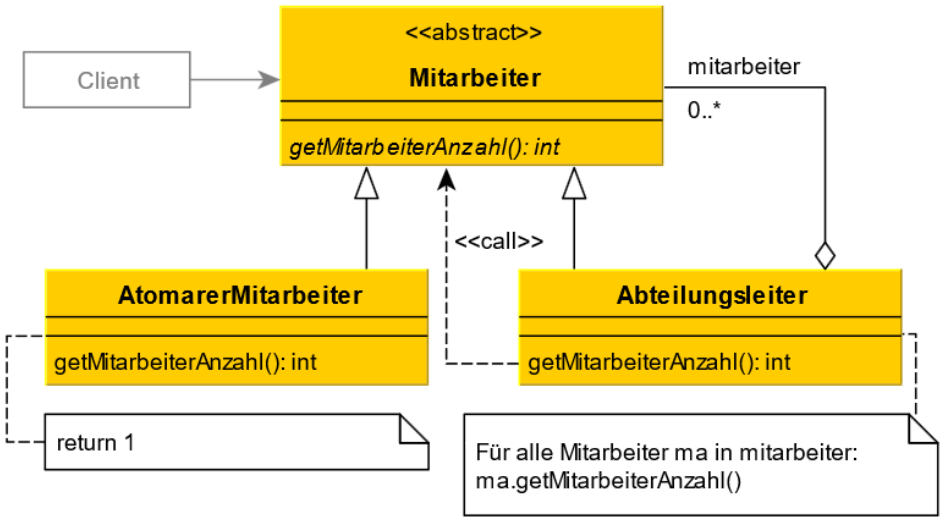
Dank Abstraktion und Polymorphie kann die Abteilungsleiterklasse alle Arten von Mitarbeitern sehr einfach vorhalten und verwalten:

```
class Abteilungsleiter extends Mitarbeiter{

    private List<Mitarbeiter> mitarbeiter = new ArrayList<Mitarbeiter>();

    public void add(Mitarbeiter ma){
        mitarbeiter.add(ma);
    }

    public void remove(Mitarbeiter ma){
        mitarbeiter.remove(ma);
    }

    public Mitarbeiter getMitarbeiter(int index){
        return mitarbeiter.get(index);
    }
}
```

Weiterhin wird auch der Clientcode wesentlich einfacher, denn in diesem muss ebenso keine Unterscheidung mehr getroffen werden (solange die Mitarbeiterschnittstelle ausreicht). Dieser Punkt wird im folgendem noch deutlicher.

Im nächsten Schritt wollen wir die Methode *getMitarbeiterAnzahl()* implementieren und zeigen, wie elegant diese realisiert werden kann. Beginnen wir mit *getMitarbeiterAnzahl().* Wir erweitern die Mitarbeiterschnittstelle um diese Methode und definieren

getMitarbeiterAnzahl() abstrakt und zwingen folglich die Unterklassen zur Implementierung der Methode.



Was ist das gewünschte Verhalten? Ruft man *getMitarbeiterAnzahl()* auf einem atomaren Mitarbeiter auf, so gibt dieser 1 zurück. Klar, er hat ja niemanden unter sich. Auf einem Ab-

teilungsleiter aufgerufen, soll die Methode die Anzahl all seiner Mitarbeiter (plus 1 für ihn selbst) ausgeben. Dies schliesst untergeordnete Abteilungsleiter und dessen Mitarbeiter mit ein. Was wäre da einfacher als über die Mitarbeiterliste zu iterieren und auf jedem Mitarbeiter *getMitarbeiterAnzahl()* aufzurufen und die Ergebnisse zu kumulieren?

Hier spielen wir den grossen Vorteil der Abstraktion aus. Der Abteilungsleiter kennt nur die Schnittstelle Mitarbeiter und ruft *getMitarbeiterAnzahl()* auf. Dank **Polymorphie** liefern atomare Mitarbeiter 1 und Abteilungsleiter rufen ihrerseits rekursiv *getMitarbeiterAnzahl()* auf ihren Mitarbeitern auf.

Die Implementierung von getMitarbeiterAnzahl():

```
abstract class Mitarbeiter{
    public abstract int getMitarbeiterAnzahl();
}

class AtomarerMitarbeiter extends Mitarbeiter{
    public int getMitarbeiterAnzahl() {
        return 1;
    }
}

class Abteilungsleiter extends Mitarbeiter{

    private List<Mitarbeiter> mitarbeiter = new ArrayList<Mitarbeiter>();

    public int getMitarbeiterAnzahl() {
        int summe = 1; //1 für sich selbst
        for (Mitarbeiter ma : mitarbeiter) {
            summe += ma.getMitarbeiterAnzahl();
        }
        return summe;
    }

    //Verwaltungsmethoden...
}
```

Was haben wir damit gewonnen? Eine sehr elegante und flexible Implementierung, die sich überaus einfach durch den Client bedienen lässt. Er muss nicht mehr manuell durch die Baumstruktur wandern, sondern überlässt dies der Baumstruktur selbst.

Der Client benötigt nur einen Aufruf. Wie die Mitarbeiteranzahl intern errechnet wird, interessiert ihn nicht:

```
public static void main(String[] args) {
    Mitarbeiter vorstand = new Abteilungsleiter();
    //Mitarbeiterhierarchie unter vorstand aufbauen (add())

    //Mitarbeiterhierarchie nutzen
    System.out.println(vorstand.getMitarbeiterAnzahl());
}
```

Sehr schön. Nun zur letzten Anforderung - dem textuellen Ausdrucken der gesamten Hierarchie. Die Realisierung dieser *print()-Methode* gestaltet sich analog zu *getMitarbeiterAnzahl()*. Die allgemeine Mitarbeiterschnittstelle wird um die abstrakte Methode *print()* erweitert. Jeder Mitarbeiter liefert beim Aufruf seinen Namen, Telefonnummer und Abteilung. Abteilungsleiter müssen natürlich zusätzlich den *print()-Aufruf* an ihre unterstellten Mitarbeiter delegieren.

Mitarbeiter deklariert die abstrakte *print()*-Methode:

Atomare Mitarbeiter geben in *print()* ihren Namen und ihre Telefonnummer aus:

Abteilungsleiter geben in *print()* ihren Namen, ihre Abteilung und ihre Telefonnummer aus. Weiterhin delegieren sie den *print()*-Befehl an ihre Mitarbeiter:

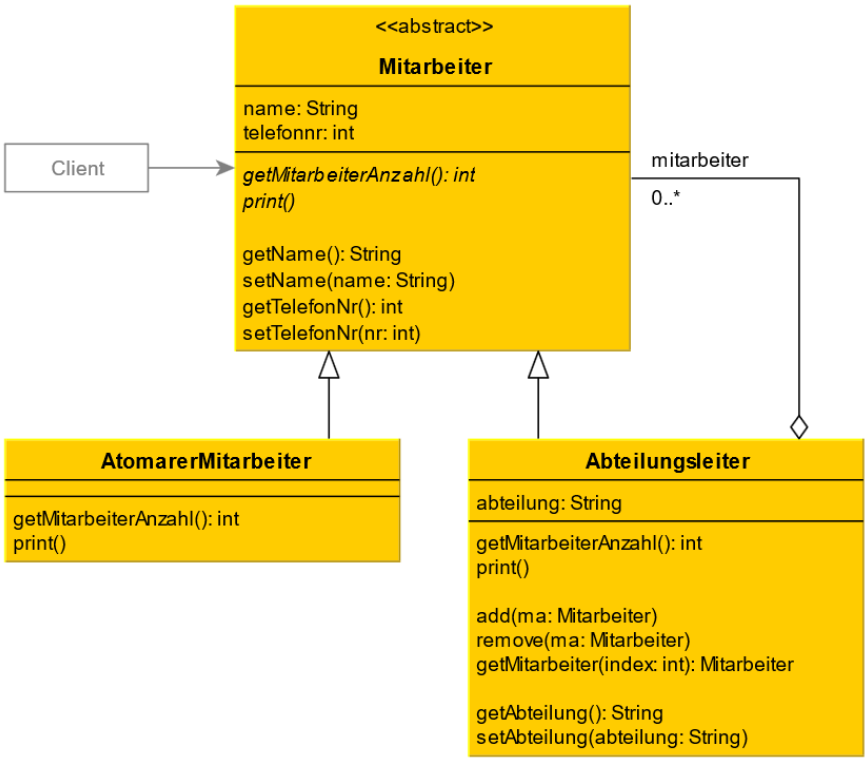
Der Client kann nun denkbar einfach das Ausdrucken der Firmenhierarchie anstossen:

Ausgabe:

```
Abteilungsleiter A. Müller (Vorstand). Tel: 4
      Abteilungsleiter W. Fischer (Vertrieb). Tel: 1
            P. Meier. Tel: 123
            I. Schulz. Tel: 112
      U. Temann. Tel: 442
      Abteilungsleiter M. Hardbrot (Entwicklung). Tel: 3
            M. Rehberg. Tel: 323
            Abteilungsleiter T. Kunz (Technologie). Tel: 2
                  M. Rehberg. Tel: 223
                  O. Riedel. Tel: 212
```

Herzlichen Glückwunsch, wir haben so eben das Composite Design Pattern realisiert! Atomare Mitarbeiter und aggregierende Abteilungsleiter sind unter einer gemeinsamen Schnittstelle zusammengefasst, sodass sie durch den Client einheitlich behandelt werden können.

Das Klassendiagramm sieht wie folgt aus:



Aufträge

- Erstellen Sie folgende Klassen
  - Mitarbeiter (abstract)
  - AtomarerMitarbeiter abgeleitet von Mitarbeiter
  - Abteilungsleiter abgeleitet von Mitarbeiter
  - Testclient
- Bauen Sie die Hierarchie gemäss Folie 2

Quellen:

https://www.tutorialspoint.com/design_pattern/

https://www.philipphauer.de/study/se/design-pattern.php