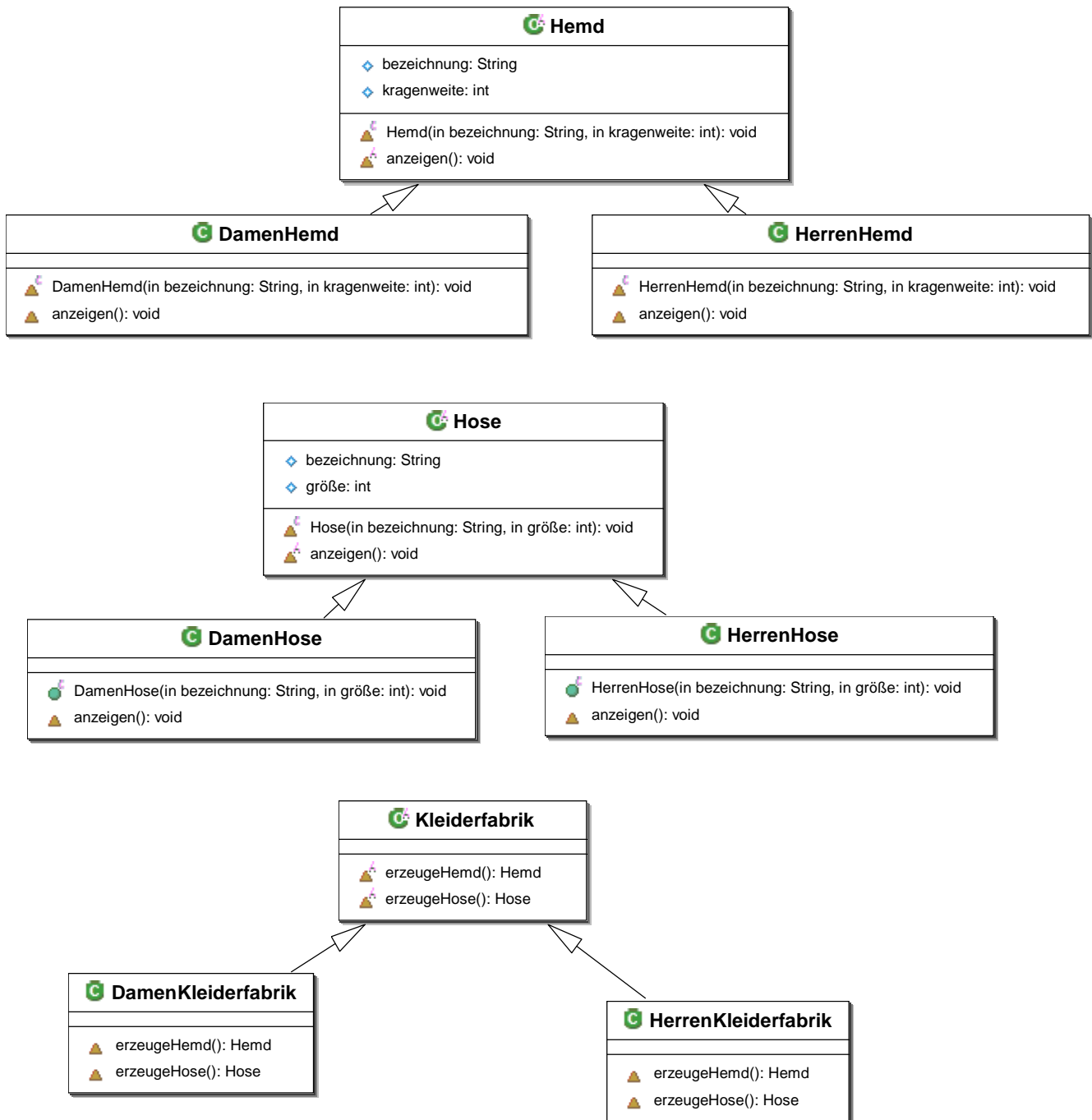


Abstrakte Fabrik (Abstract Factory)
Biete eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen.



```

public static void main(String[] args) {

    int auswahl = 0;
    Kleiderfabrik meineKleiderfabrik;

    // Konkrete Fabrik kann zur LAUFZEIT ausgewählt werden:
    if( auswahl == 1)
        meineKleiderfabrik = new HerrenKleiderfabrik();
    else
        meineKleiderfabrik = new DamenKleiderfabrik();

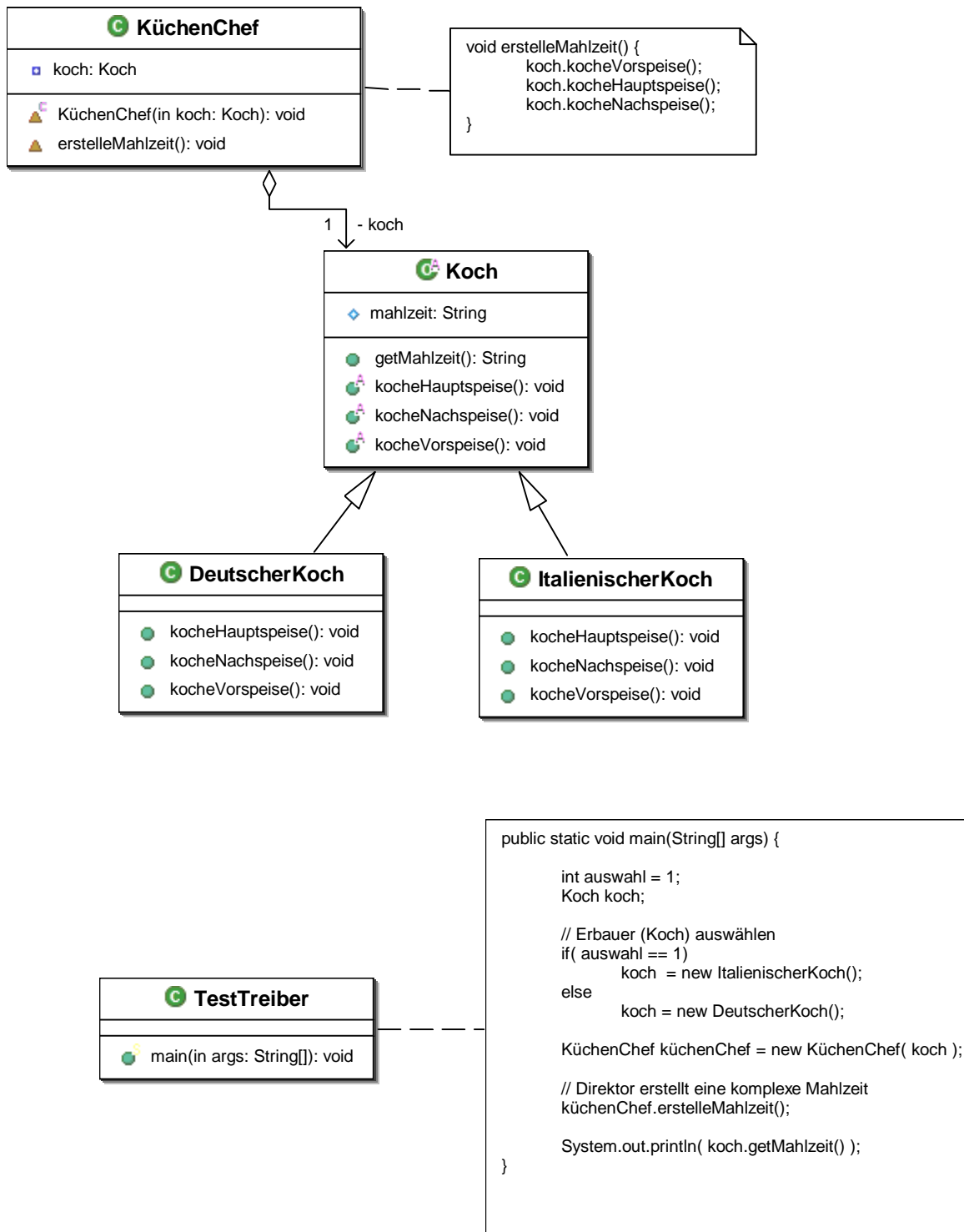
    // Konkretfabrik erstellt Instanzen von Hemd/Hose ABGELEITETER Klassen
    Hemd meinHemd = meineKleiderfabrik.erzeugeHemd();
    Hose meineHose = meineKleiderfabrik.erzeugeHose();

    meinHemd.anzeigen();
    meineHose.anzeigen();

}
  
```

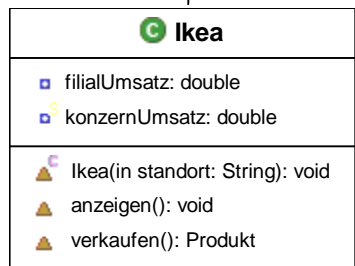
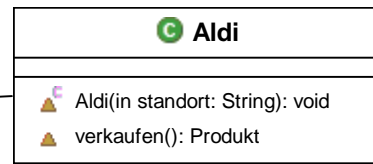
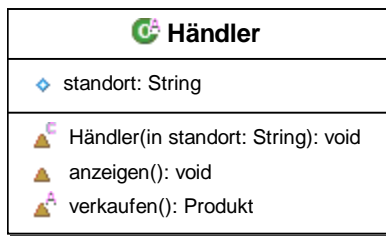
Erbauer (Builder)

Trenne die Konstruktion eines komplexen Objekts von seiner Repräsentation, so dass derselbe Konstruktionsprozeß unterschiedliche Repräsentationen erzeugen kann.

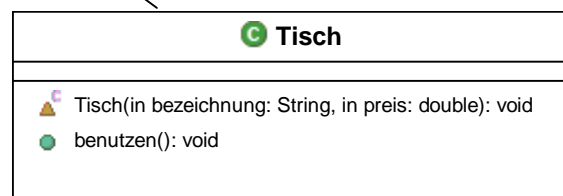
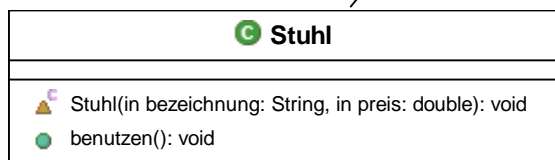
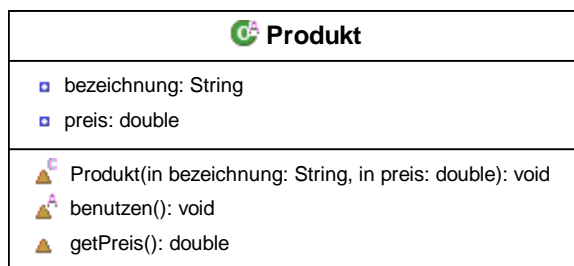


Fabrikmethode (Factory Method)

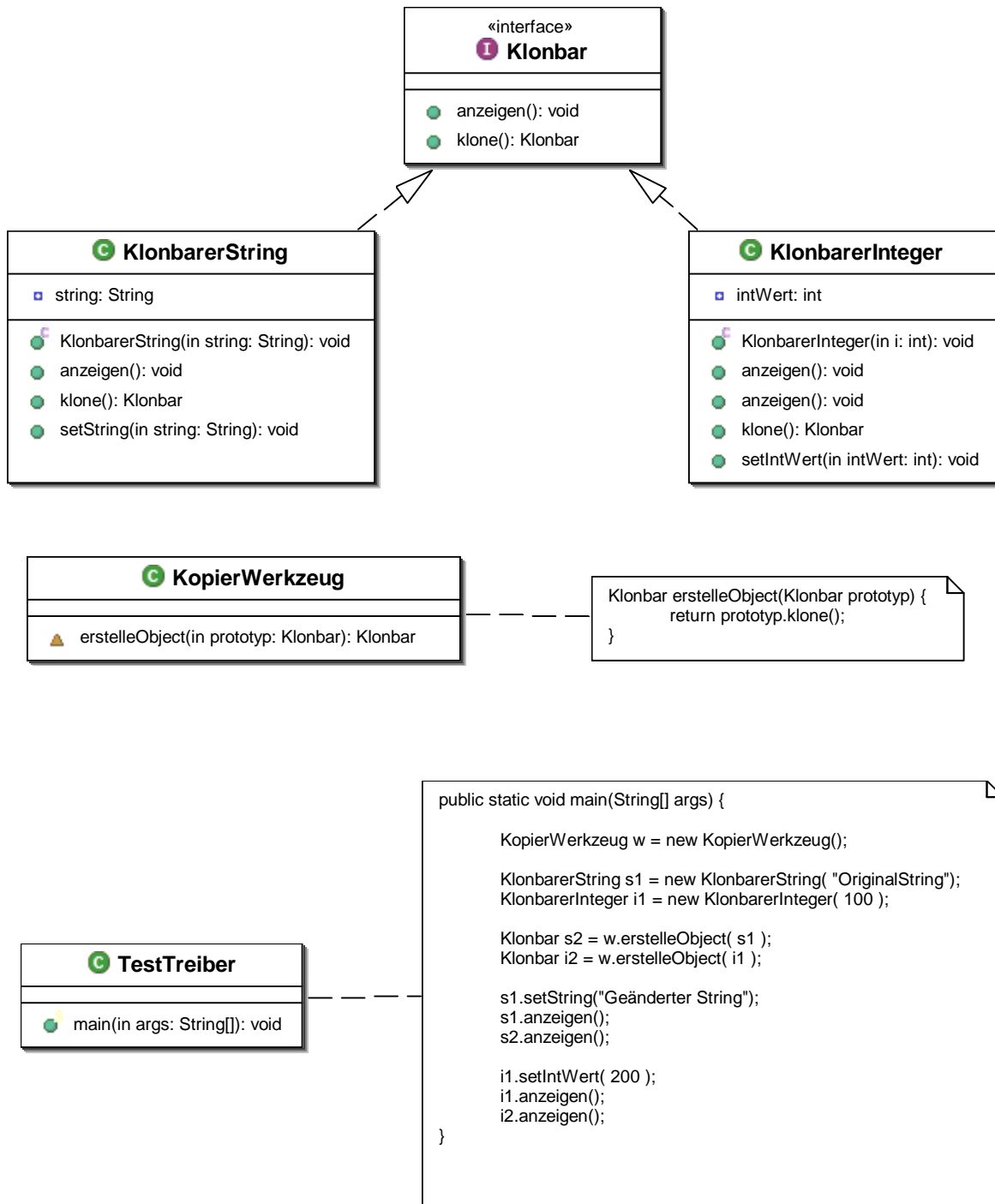
Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.



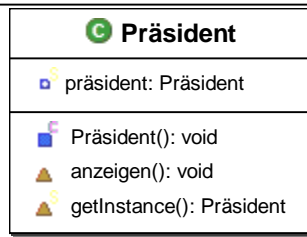
```
Produkt verkaufen() {  
    int auswahl = 0;  
    Produkt gekauftesProdukt;  
  
    // Produkt entscheidet sich erst ZUR LAUFZEIT  
    switch( auswahl )  
    {  
        case 1:  
            gekauftesProdukt = new Tisch("Tisch", 100);  
            break;  
        case 2:  
            gekauftesProdukt = new Stuhl("Stuhl", 50);  
            break;  
        default:  
            gekauftesProdukt = null;  
    }  
  
    return gekauftesProdukt;  
}
```



Prototyp (Prototype)
Bestimme die Arten zu erzeugender Objekte durch die Verwendung eines prototypischen Ex-emplars,
und erzeuge neue Objekte durch Kopieren dieses Prototypen.



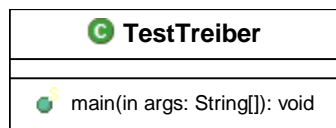
Singleton (Singleton)
Sichere ab, daß eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffs-punkt darauf bereit.



```
static Präsident getInstance() {           // statische Methode (Klassenfunktion)

    if ( prääsident == null )              // existiert bereits ein Präsident ?
    {
        prääsident = new Präsident();      // nein, dann neu anlegen.
    }

    return prääsident;                    // den einzigen Präsidenten zurückliefern.
}
```



```
public static void main(String[] args) {

    // Geht NICHT, da Konstruktor der Klasse Präsident private ist.
    // Präsident p = new Präsident();

    // Aufruf der Klassenfunktion getInstance()
    Präsident p = Präsident.getInstance(); // Präsident wird erstellt...

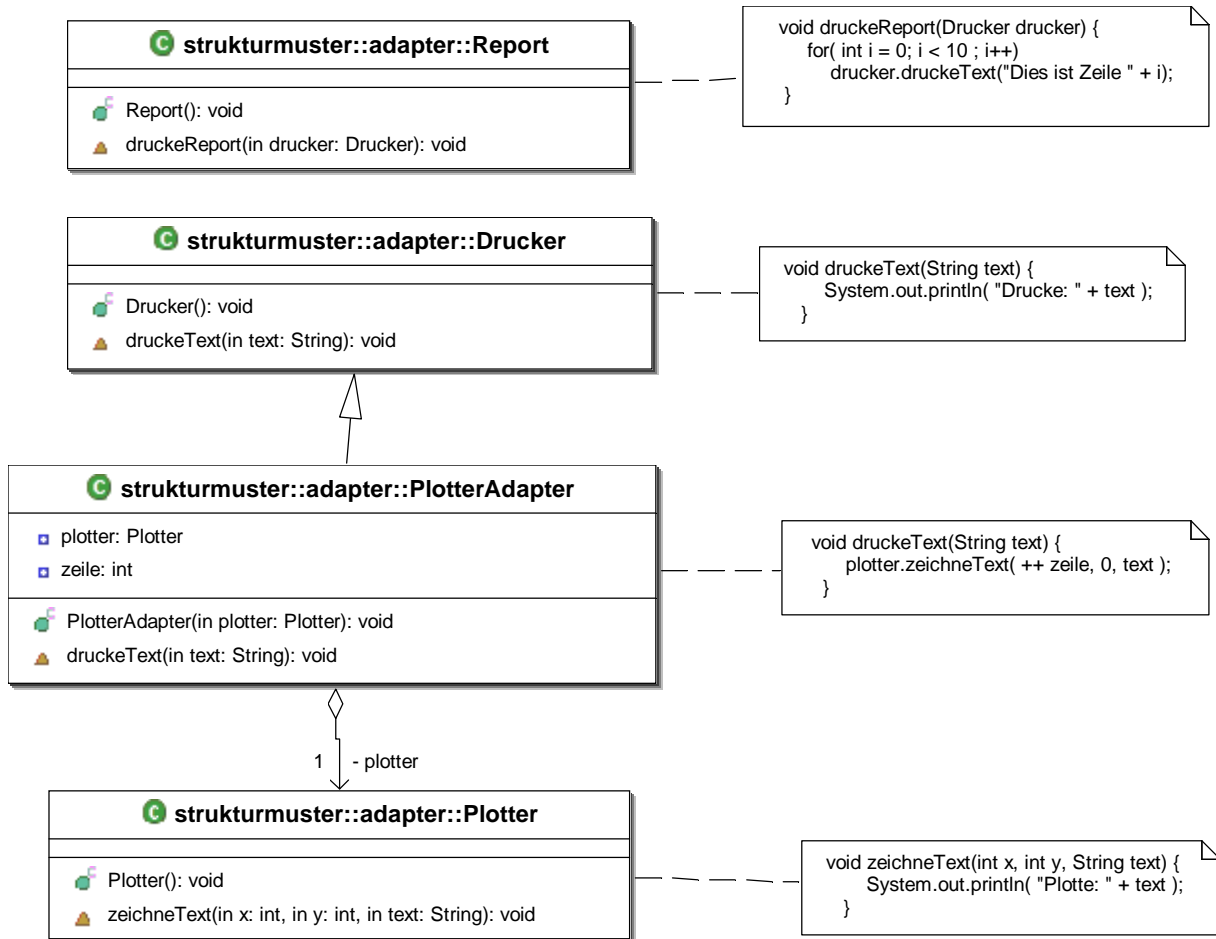
    p = Präsident.getInstance();          // vorhandener Präsident wird zurück geliefert.
    p.anzeigen();

}
```

Adapter (Adapter)

Passe die Schnittstelle einer Klasse an eine andere von ihren Klienten erwartete Schnittstelle an.

Das Adaptermuster läßt Klassen zusammenarbeiten, die wegen inkompatibler Schnittstellen ansonsten dazu nicht in der Lage wären.



```
void druckeReport(Drucker drucker) {
    for( int i = 0; i < 10 ; i++)
        drucker.druckeText("Dies ist Zeile " + i);
}
```

```
void druckeText(String text) {
    System.out.println( "Drucke: " + text );
}
```

```
void druckeText(String text) {
    plotter.zeichneText( ++ zeile, 0, text );
}
```

```
void zeichneText(int x, int y, String text) {
    System.out.println( "Plotte: " + text );
}
```

strukturmuster::adapter::TestTreiber

- TestTreiber(): void
- main(in args: String[]): void

```
public static void main(String[] args) {
```

```
    Drucker meinDrucker = new Drucker();
    Plotter meinPlotter = new Plotter();
```

```
    Report meinReport = new Report();
```

```
    System.out.println("Drucke Report auf dem Drucker...");
    meinReport.druckeReport( meinDrucker );
```

```
    System.out.println("Drucke Report auf Plotter...");
```

```
    // Geht NICHT: Ein Plotter ist kein Drucker !
    // meinReport.druckeReport( meinPlotter );
```

```
    // Aufgabe:
```

```
    // Erstellen Sie die Klasse PlotterAdapter, so dass
```

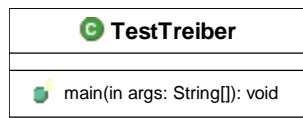
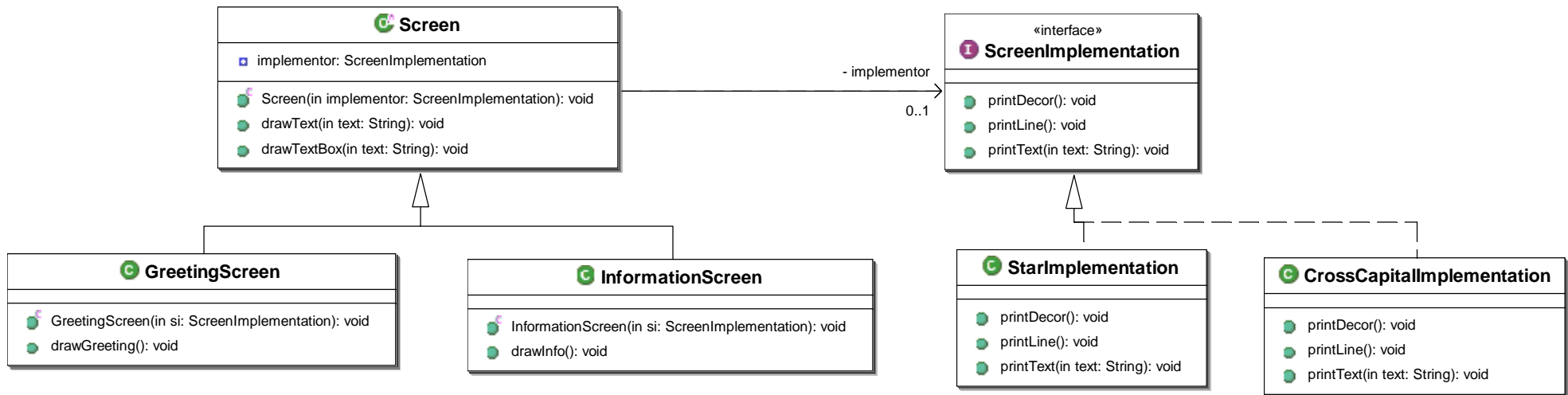
```
    // der Report auf dem Plotter ausgegeben werden kann.
```

```
    // Die Klassen Report, Drucker und Plotter dürfen NICHT verändert werden.
```

```
    // Lösung: Durch PlotterAdapter kann ein Plotter als Drucker verwendet werden
    meinReport.druckeReport( new PlotterAdapter( meinPlotter ) );
```

```
}
```

Brücke (Bridge)
Entkopple eine Abstraktion von ihrer Implementierung, so daß beide unabhängig voneinander variiert werden können.



```
public static void main(String[] args) {

    System.out.println("Creating implementations...");

    ScreenImplementation i1 = new StarImplementation();
    ScreenImplementation i2 = new CrossCapitalImplementation();

    System.out.println("Creating abstraction (screens) / implementation combinations...");

    GreetingScreen gs1 = new GreetingScreen(i1);
    GreetingScreen gs2 = new GreetingScreen(i2);
    InformationScreen is1 = new InformationScreen(i1);
    InformationScreen is2 = new InformationScreen(i2);

    System.out.println("Starting test:\n");

    gs1.drawText("\nScreen 1 (Refined Abstraction 1, Implementation 1):");
    gs1.drawGreeting();

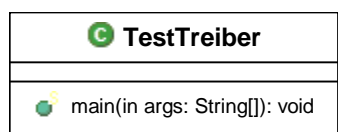
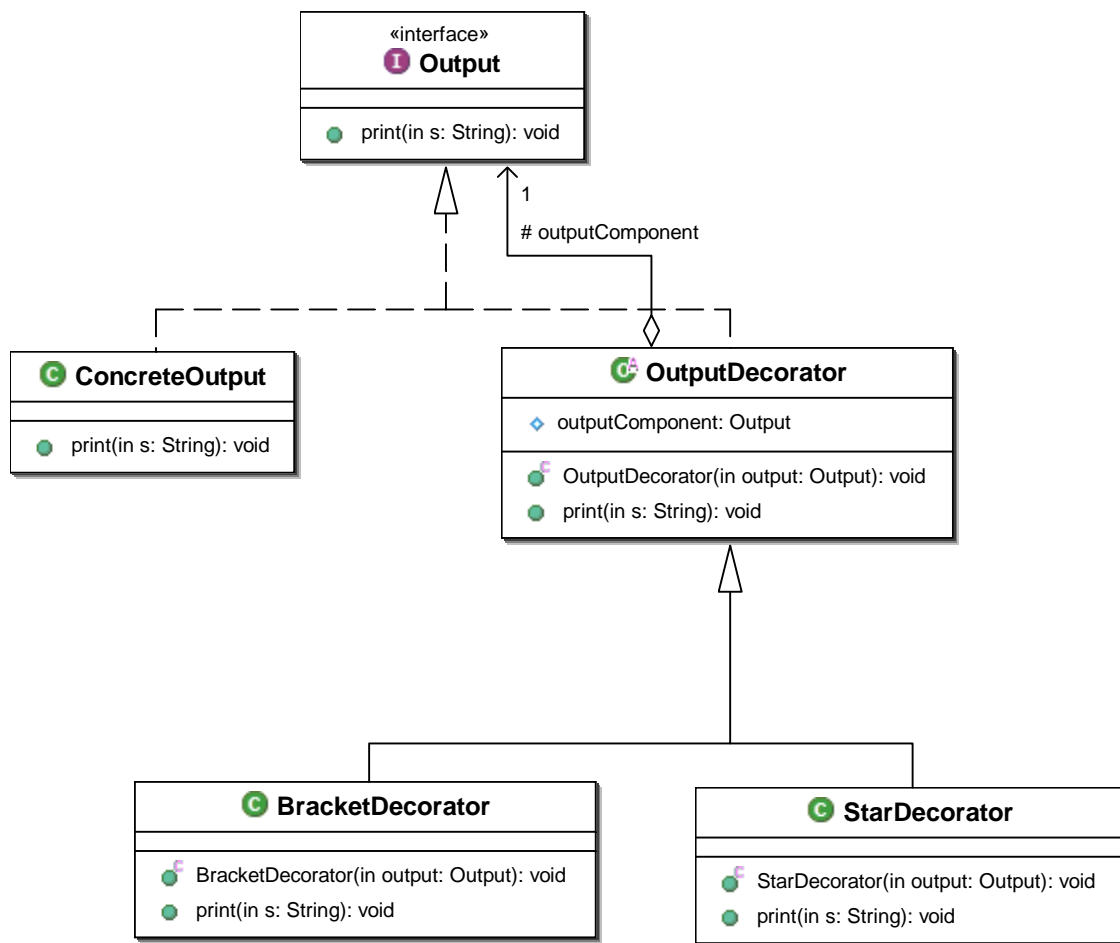
    gs2.drawText("\nScreen 2 (Refined Abstraction 1, Implementation 2):");
    gs2.drawGreeting();

    is1.drawText("\nScreen 3 (Refined Abstraction 2, Implementation 1):");
    is1.drawInfo();

    is2.drawText("\nScreen 4 (Refined Abstraction 2, Implementation 2):");
    is2.drawInfo();

}
```

Dekorierer (Decorator)
Erweitere ein Objekt dynamisch um Zuständigkeiten. Dekorierer bieten eine flexible Alternative zur Unterklassenbildung, um die Funktionalität einer Klasse zu erweitern.




```
public static void main(String[] args) {  
  
    Output original = new ConcreteOutput();  
    Output bracketed= new BracketDecorator(original);  
    Output stared  = new StarDecorator(bracketed);  
  
    stared.print("<String>");  
  
    System.out.println();  
  
}
```


Fassade (Facade)

Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems.

Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Verwendung des Subsystem vereinfacht.

ReaderFassade

 `eingebenInteger(in meldung: String): int`

```
public static int eingebenInteger( String meldung ) throws IOException {
```


```
    InputStreamReader isr = new InputStreamReader( System.in );  
    BufferedReader br = new BufferedReader( isr );
```

```
    System.out.println( meldung );  
    int eingabe = Integer.parseInt( br.readLine() );
```

```
    return eingabe;
```

```
}
```

TestTreiber

 `main(in args: String[]): void`

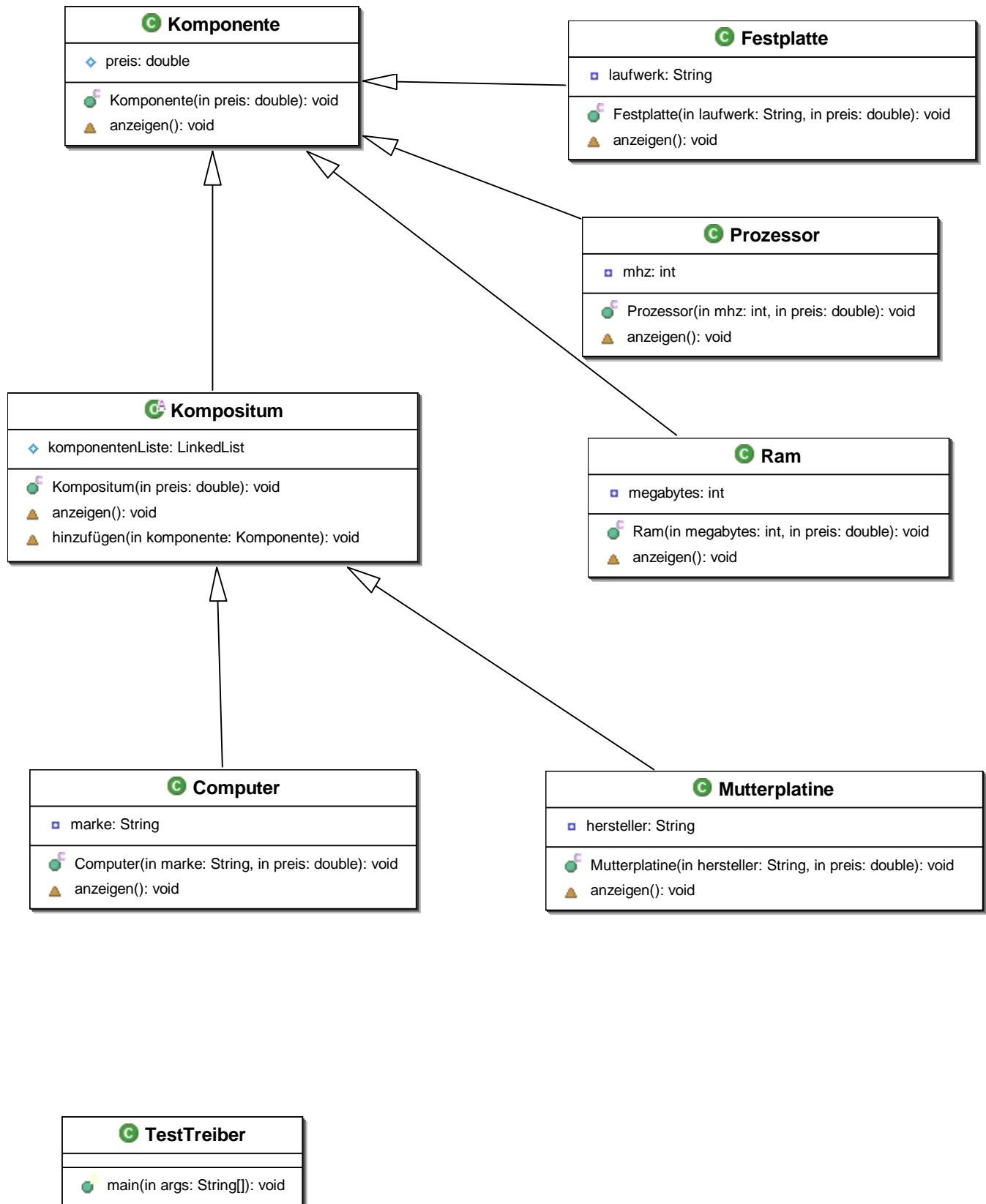
```
public static void main(String[] args) throws IOException {
```

```
    int x = ReaderFassade.eingebenInteger("x eingeben:");
```

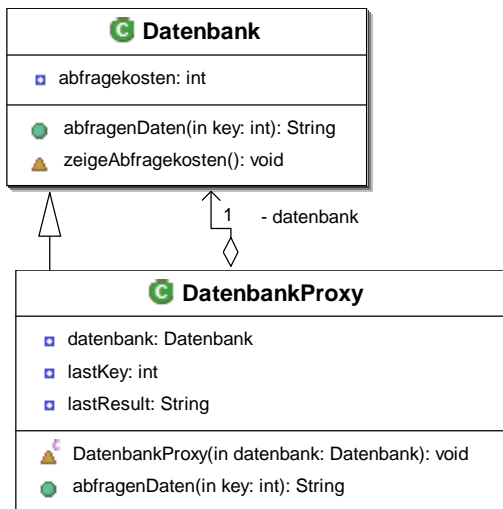
```
    System.out.println("wert von x: " + x);
```

```
}
```

Kompositum (Composite)
Füge Objekte zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu repräsentieren.
Das Kompositionsmuster ermöglicht es Klienten, einzelne Objekte sowie Kompositionen von Objekten einheitlich zu behandeln.



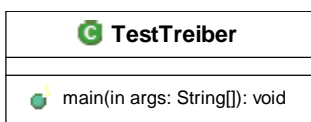
Proxy (Proxy)
Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts.



```
public String abfragenDaten(int key) {
    String result;

    // Proxy verhindert Zugriff auf vertrauliche Daten.
    if( key <= 100 )
    {
        System.out.print("Kein Zugriff.");
        result = "<nichts>";
    }
    // Proxy erspart zeitaufwändigen Datenbankzugriff, wenn Daten im Cache vorhanden
    else if( key == lastKey )
    {
        System.out.print("Daten aus dem Cache abgerufen.");
        result = lastResult;
    }
    else // ansonsten Abfrage an die Datenbank weiterleiten
    {
        result = datenbank.abfragenDaten( key );
        lastKey = key;
        lastResult= result;
    }

    return result;
}
```



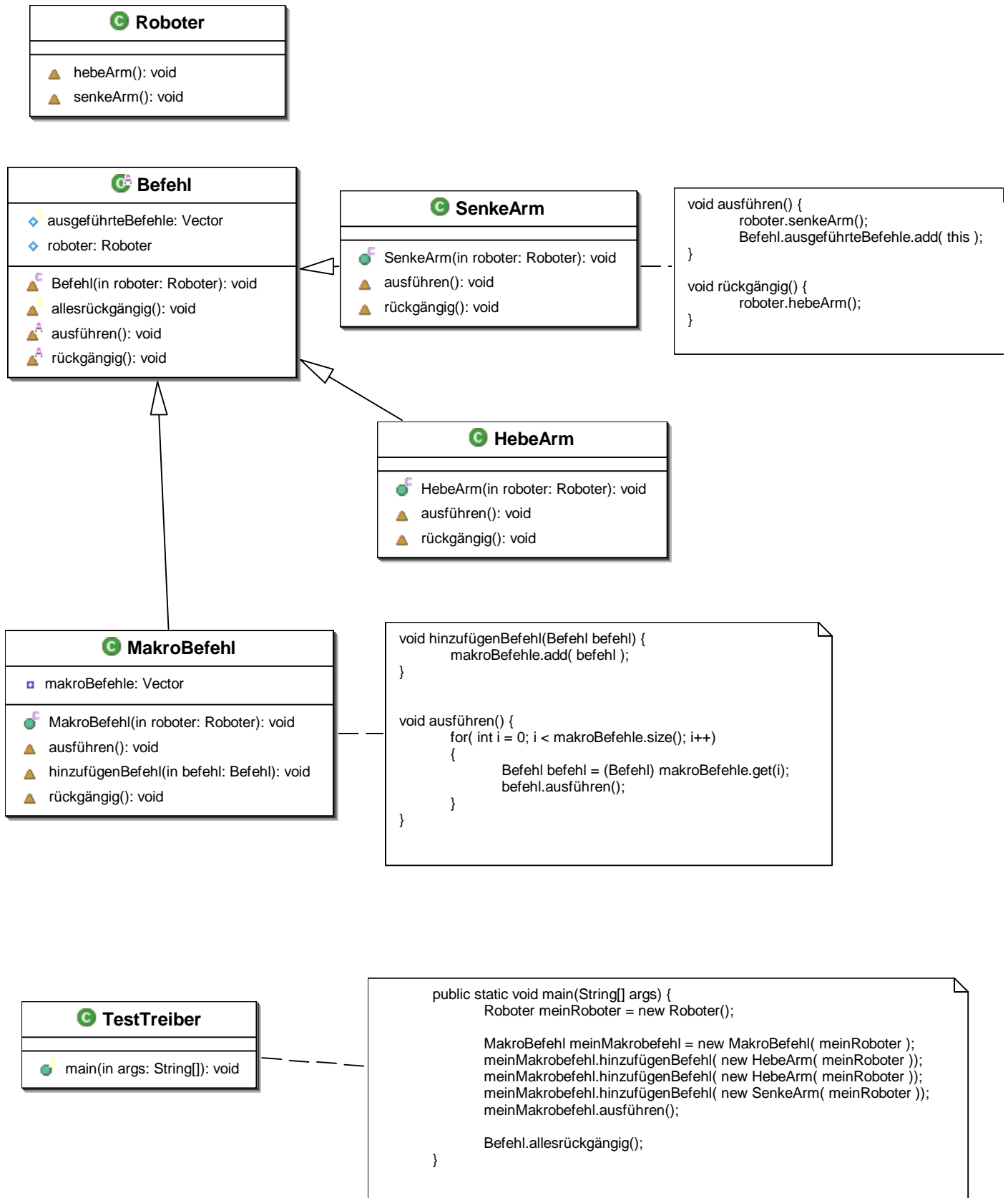
```
public static void main(String[] args) {
    Datenbank      meineDatenbank = new Datenbank();
    DatenbankProxy meinDatenbankProxy = new DatenbankProxy( meineDatenbank );
    String         record;

    // Datenbankabfrage an den PROXY richten
    record = meinDatenbankProxy.abfragenDaten( 500 );
    System.out.println("Ergebnis: " + record );

    // Daten werden bei zweiten Aufruf aus dem Cache geliefert:
    record = meinDatenbankProxy.abfragenDaten( 500 );
    System.out.println("Ergebnis: " + record );

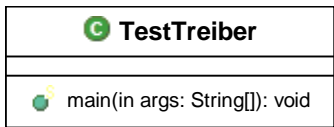
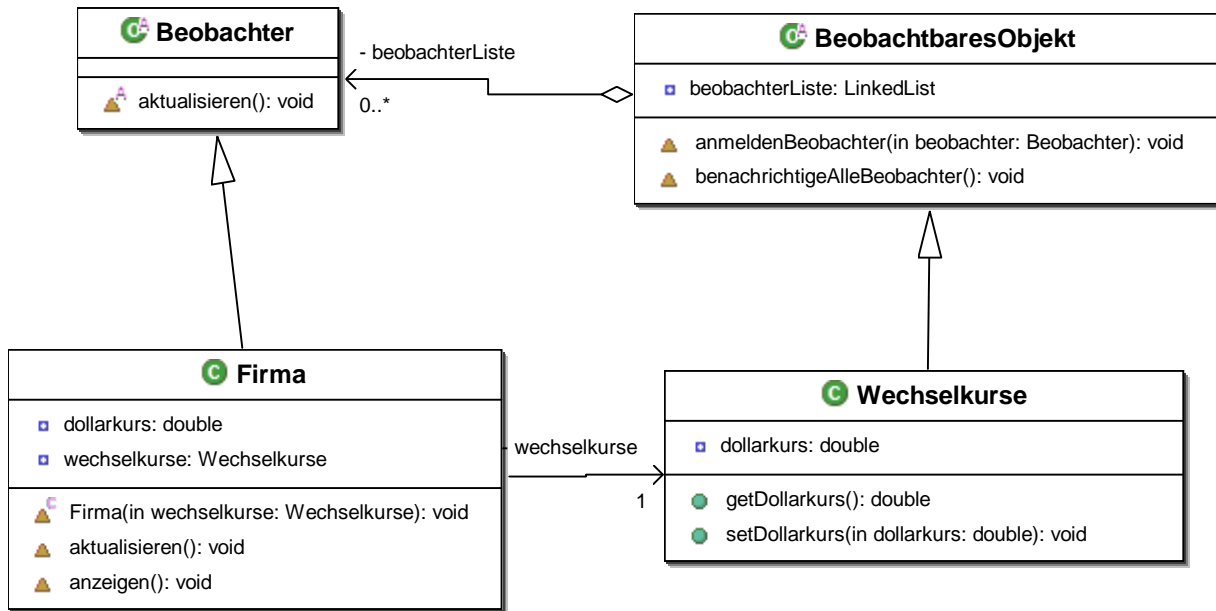
    // Proxy schützt vertrauliche Daten:
    record = meinDatenbankProxy.abfragenDaten( 0 );
    System.out.println("Ergebnis: " + record );
}
```

Befehl (Command)
 Kapsle einen Befehl als ein Objekt. Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrisieren, Operationen in eine Schlange zu stellen, ein Logbuch zu fuhren und Ope-rationen rückgängig zu machen.



Beobachter (Observer)

Definiere eine 1-zu-n-Abhängigkeit zwischen Objekten, so daß die Änderung des Zustands eines Objekts dazu führt, daß alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.



```
public static void main(String[] args) {
```

```
    Wechselkurse meineWechselkurse = new Wechselkurse();
    meineWechselkurse.setDollarkurs( 1.10 );
```

```
    Firma meineFirma = new Firma( meineWechselkurse );
    Firma deineFirma = new Firma( meineWechselkurse );
```

```
    meineWechselkurse.anmeldenBeobachter( meineFirma );
    // meineWechselkurse.anmeldenBeobachter( deineFirma );
```

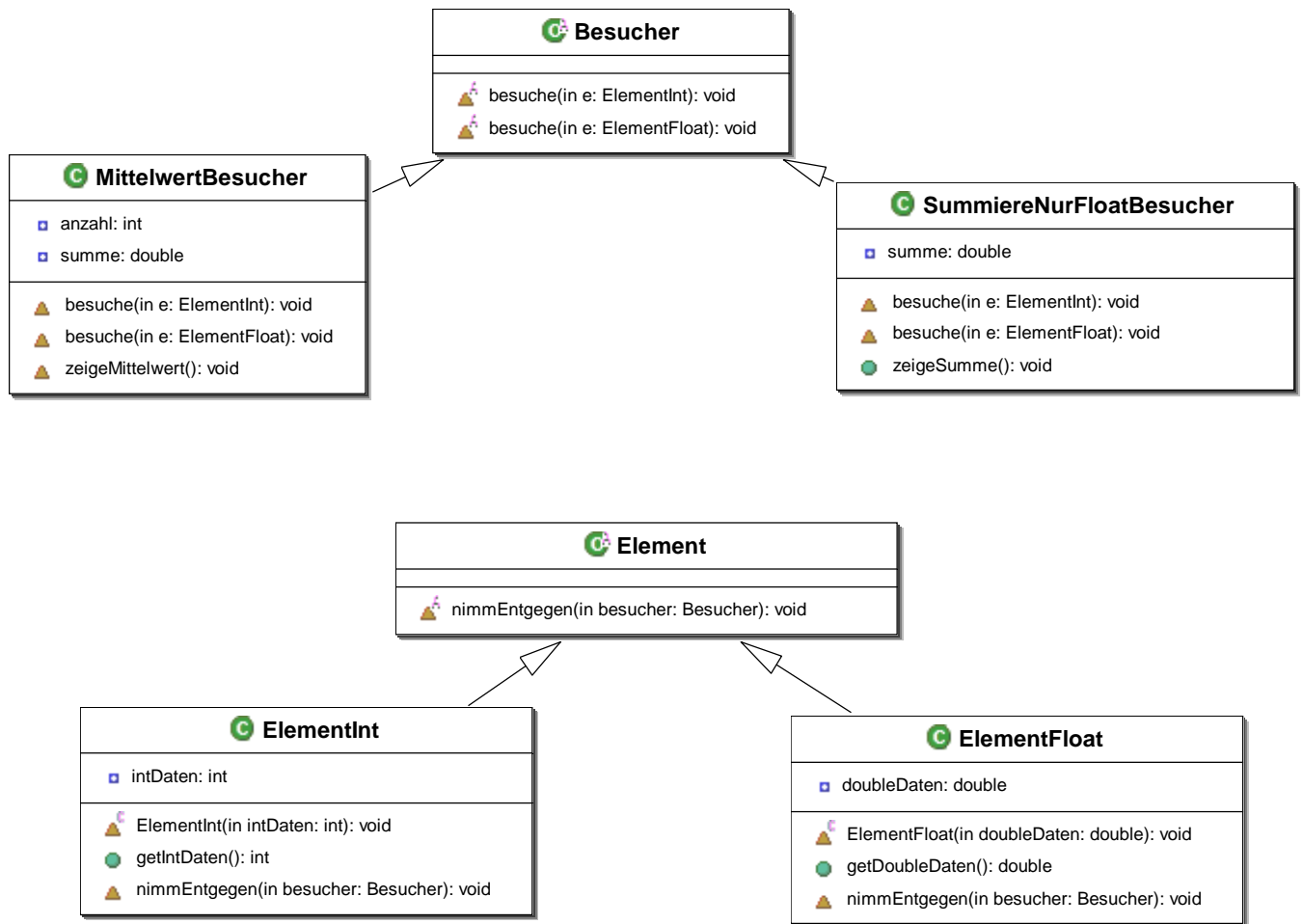
```
    // Alle angemeldeten Beobachter werden automatisch aktualisiert
    meineWechselkurse.setDollarkurs( 1.25 );
```

```
    meineFirma.anzeigen();
    deineFirma.anzeigen();
```

```
}
```

Besucher (Visitor)

Kapsle eine auf den Elementen einer Objektstruktur auszuführende Operation als ein Objekt. Das Besuchermuster ermöglicht es Ihnen, eine neue Operation zu definieren, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.



TestTreiber

`main(in args: String[]): void`

```
public static void main(String[] args) {
    int i;
    int MAX_ELEMENTE = 5;

    Element[] elementContainer = new Element[MAX_ELEMENTE];

    elementContainer[0] = new ElementFloat (1.0f);
    elementContainer[1] = new ElementInt (2);
    elementContainer[2] = new ElementFloat (3.0f);
    elementContainer[3] = new ElementInt (4);
    elementContainer[4] = new ElementFloat (5.0f);

    SummiereNurFloatBesucher nurFloatBesucher = new SummiereNurFloatBesucher();


    // Berechnung der Summe aller floats durch den Floatbesucher
    for (i=0; i<MAX_ELEMENTE; i++)
    {
        elementContainer[i].nimmEntgegen(nurFloatBesucher);
    }
    nurFloatBesucher.zeigeSumme();

    // Berechnung des Mittelwerts durch den Mittelwertbesucher
    MittelwertBesucher mittelwertBesucher = new MittelwertBesucher();
    for (i=0; i<MAX_ELEMENTE; i++)
    {
        elementContainer[i].nimmEntgegen(mittelwertBesucher);
    }
    mittelwertBesucher.zeigeMittelwert();
}
```

Iterator (Iterator)

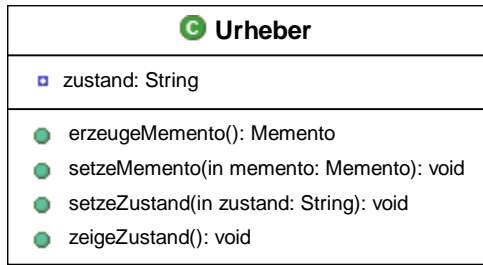
Biete eine Möglichkeit, um auf die Elemente eines zusammengesetzten Objekts sequentiell zugreifen zu können, ohne die zugrundeliegende Repräsentation offenzulegen.

TestTreiber

 main(in args: String[]): void

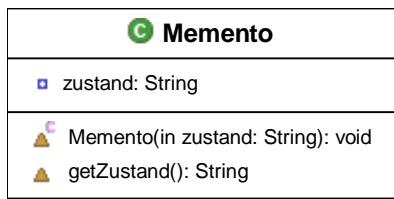
```
public static void main(String[] args) {  
  
    // Daten in einem vector speichern  
    //Vector daten = new Vector();  
  
    // Daten in einer verketteten Liste speichern  
    LinkedList daten = new LinkedList();  
  
    daten.add( new Integer(10) );  
    daten.add( new Integer(100) );  
    daten.add( new Integer(200) );  
    daten.add( new Integer(50) );  
  
    // Ein Iterator bietet containerunabhängigen Zugriff auf die Daten  
    Iterator it = daten.iterator();  
  
    while( it.hasNext() )  
    {  
        System.out.println( it.next() );  
    }  
}
```

Memento (Memento)
Erfasse und externalisiere den internen Zustand eines Objekts, so daß das Objekt später in diesen Zustand zurückversetzt werden kann.

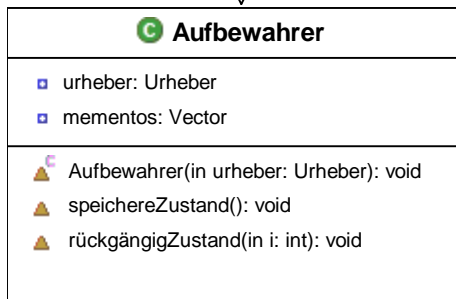


```
public Memento erzeugeMemento(){
    return new Memento( zustand );
}

public void setzeMemento(Memento memento) {
    zustand = memento.getZustand();
}
```

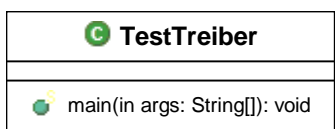


1..* - mementos



```
void speichereZustand() {
    mementos.add( urheber.erzeugeMemento() );
}

void rückgängigZustand(int i) {
    urheber.setzeMemento( (Memento) mementos.get(i) );
}
```



```
public static void main(String[] args) {

    Urheber urheber = new Urheber();
    Aufbewahrer aufbewahrer = new Aufbewahrer( urheber );

    urheber.setzeZustand("Zustand0");
    urheber.zeigeZustand();
    aufbewahrer.speichereZustand();

    urheber.setzeZustand("Zustand1");
    urheber.zeigeZustand();
    aufbewahrer.speichereZustand();

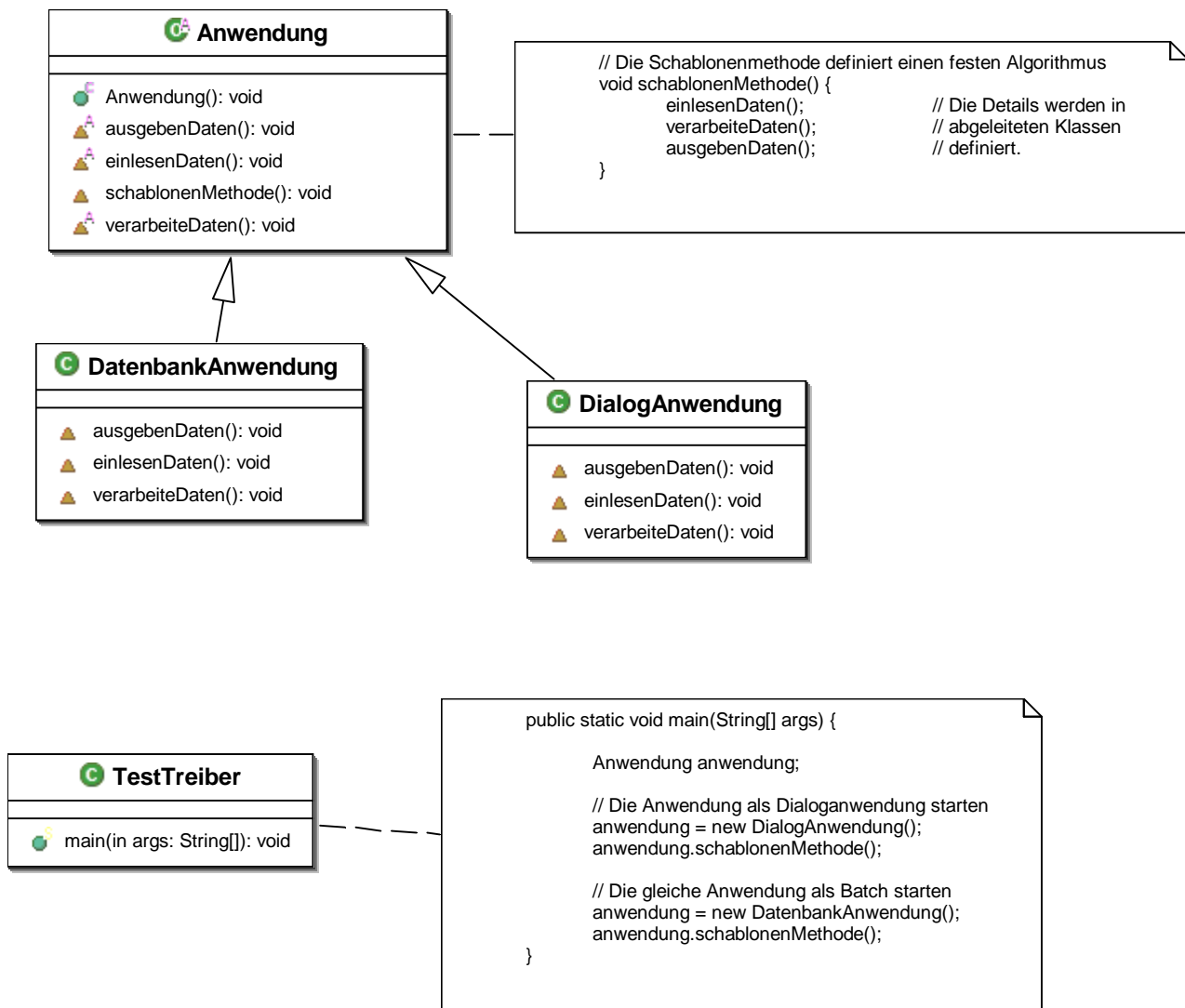
    aufbewahrer.rückgängigZustand(0);
    urheber.zeigeZustand();

    aufbewahrer.rückgängigZustand(1);
    urheber.zeigeZustand();

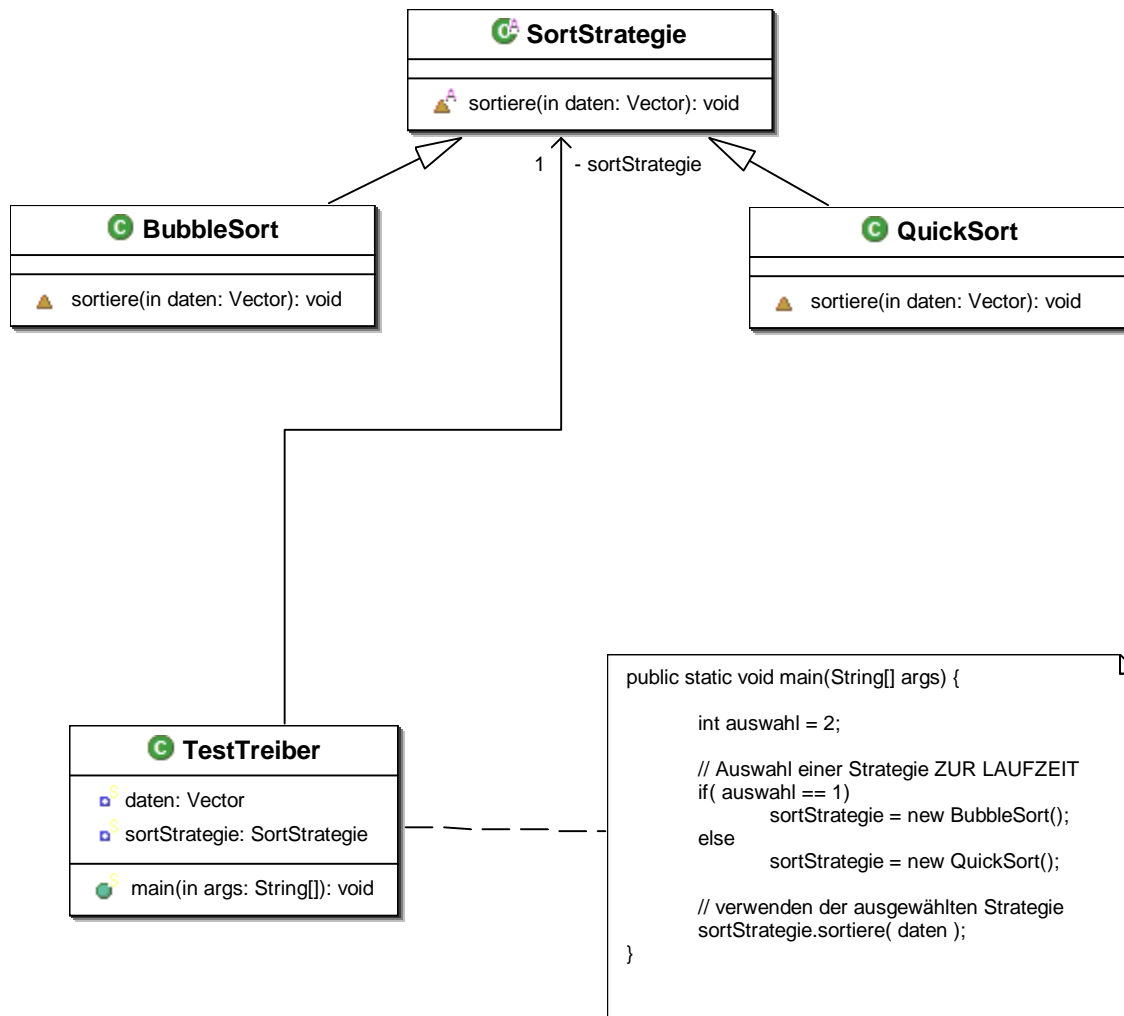
}
```


Schablonenmethode (Template Method)

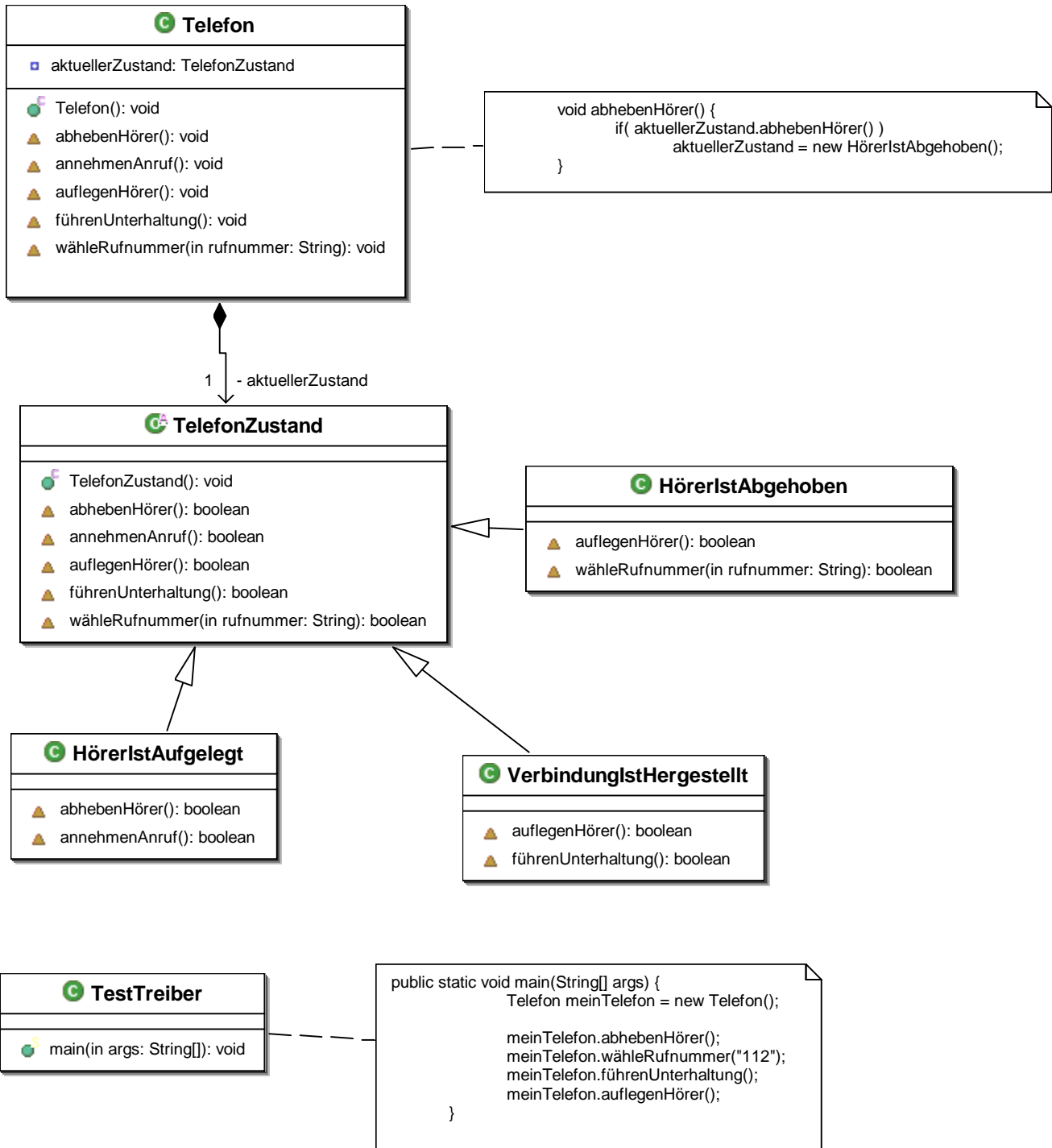
Definiere das Skelett eines Algorithmus in einer Operation und delegiere einzelne Schritte an Unterklassen.
Die Verwendung einer Schablonenmethode ermöglicht es Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne seine Struktur zu verändern.



Strategie (Strategy)
Definiere eine Familie von Algorithmen, kapsle jeden einzelnen und mache sie austausch-bar.
Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von ihn nutzenden Klienten zu variieren.



Zustand (State)
Ermöglicht es einem Objekt, sein Verhalten zu ändern, wenn sein interner Zustand sich ändert.
Es wird so aussehen, als ob das Objekt seine Klasse gewechselt hat.



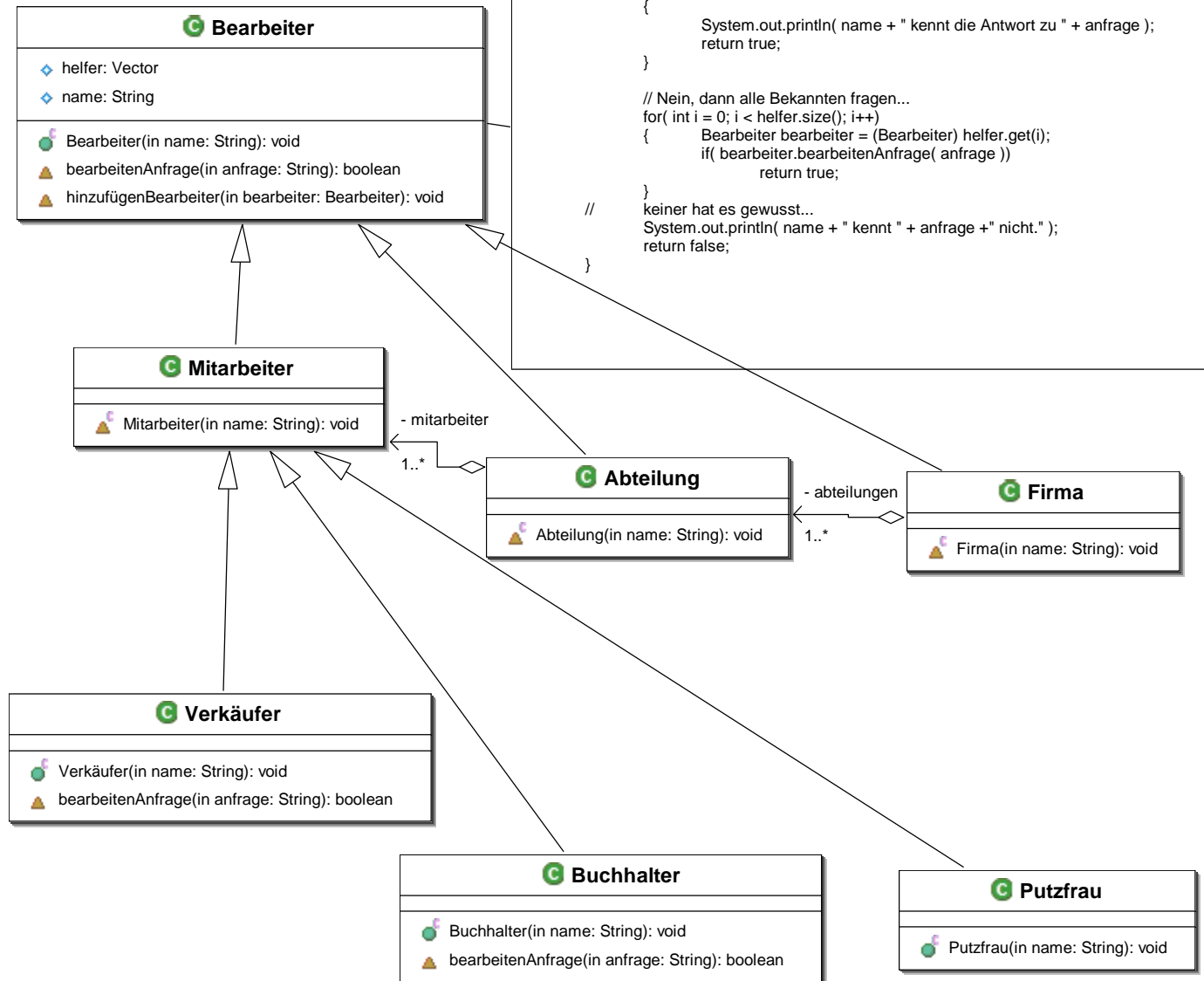
Zuständigkeitskette (Chain of Responsibility)
 Vermeide die Kopplung des Auslösers einer Anfrage an seinen Empfänger, indem mehr als ein Objekt die Möglichkeit erhält, die Anfrage zu erledigen. Verkette die empfangenden Objekte, und leite die Anfrage an der Kette entlang, bis ein Objekt sie erledigt.

```
boolean bearbeitenAnfrage(String anfrage) {

    // Weiss der Bearbeiter die Antwort selbst?
    if( anfrage.equals("Testfrage"))
    {
        System.out.println( name + " kennt die Antwort zu " + anfrage );
        return true;
    }

    // Nein, dann alle Bekannten fragen...
    for( int i = 0; i < helfer.size(); i++)
    {
        Bearbeiter bearbeiter = (Bearbeiter) helfer.get(i);
        if( bearbeiter.bearbeitenAnfrage( anfrage ))
            return true;
    }

    // keiner hat es gewusst...
    System.out.println( name + " kennt " + anfrage + " nicht." );
    return false;
}
```



```
public static void main(String[] args) {

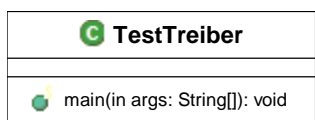
    // Firma anlegen
    Firma meineFirma = new Firma("ABC GmbH");

    // Vertrieb zur Firma hinzufügen
    Abteilung vertrieb = new Abteilung("Vertrieb");
    vertrieb.hinzufügenBearbeiter( new Putzfrau("Putzfrau Schulz"));
    vertrieb.hinzufügenBearbeiter(new Verkäufer("Verkäufer Müller"));
    meineFirma.hinzufügenBearbeiter( vertrieb );

    // Buchhaltungsabteilung zur Firma hinzufügen
    Abteilung buchhaltung = new Abteilung("Buchhaltung");
    buchhaltung.hinzufügenBearbeiter( new Putzfrau("Putzfrau Schmidt"));
    buchhaltung.hinzufügenBearbeiter(new Buchhalter("Buchhalter Meier"));
    meineFirma.hinzufügenBearbeiter( buchhaltung );

    meineFirma.bearbeitenAnfrage("Kontostand");
    meineFirma.bearbeitenAnfrage("Preis");
    meineFirma.bearbeitenAnfrage("Sinn des Lebens");

}
```




SmartString

- ▣ beobachter: Vector
- ▣ original: SmartString
- ▣ wert: String

- anzeigen(): void
- holenWert(): String
- setzenWert(in w: String): void
- setzenWert(in o: SmartString): void

TestTreiber

 main(in args: String[]): void

```
public static void main(String[] args) {  
    SmartString obj1 = new SmartString();  
    SmartString obj2 = new SmartString();  
    SmartString obj3 = new SmartString();  
  
    obj1.setzenWert("Wert1");  
    obj2.setzenWert( obj1 );  
    obj3.setzenWert( obj1 );  
  
    obj1.anzeigen();  
    obj2.anzeigen();  
    obj3.anzeigen();  
  
    obj2.setzenWert("neu");  
  
    obj1.anzeigen();  
    obj2.anzeigen();  
    obj3.anzeigen();  
  
}
```