

# Objektorientierte Software-Entwicklung

---

Priv.-Doz Dr. Rolf Hennicker

04.10.2002



# Kapitel 4

# Objektorientierter Entwurf

## Ziele

- Aus dem statischen und dynamischen Analysemodell einen Objektentwurf entwickeln können.
- Verschiedene Alternativen zur Realisierung von Zustandsdiagrammen kennen.
- Prinzipien der Systemarchitektur verstehen.
- Graphische Benutzerschnittstellen entwickeln können.
- Die Vorgehensweise bei der Anbindung an eine Datenbank verstehen.
- Entwurfsmuster kennenlernen.

## *Ausgangspunkt*

Statisches und dynamisches Modell der objektorientierten Analyse

## *Ziel*

Modell der Systemimplementierung (beschreibt *wie* die einzelnen Aufgaben gelöst werden)

## **Wesentliche Aufgaben**

- Verfeinerung des Analysemodells durch Integration des statischen und dynamischen Modells. Führt zum *Objektentwurf*.
- Einbindung in die Systemumgebung durch den Entwurf von Benutzerschnittstellen, Datenbankschnittstellen, Netzwerk-Wrappern etc.
- Konstruktion der Systemarchitektur

## 4.1 Objektentwurf

Das statische Analysemodell (Objektmodell) wird erweitert und überarbeitet. Hierzu werden Informationen aus dem dynamischen Modell der Analyse verwendet.

### Aufgaben des Objektentwurfs

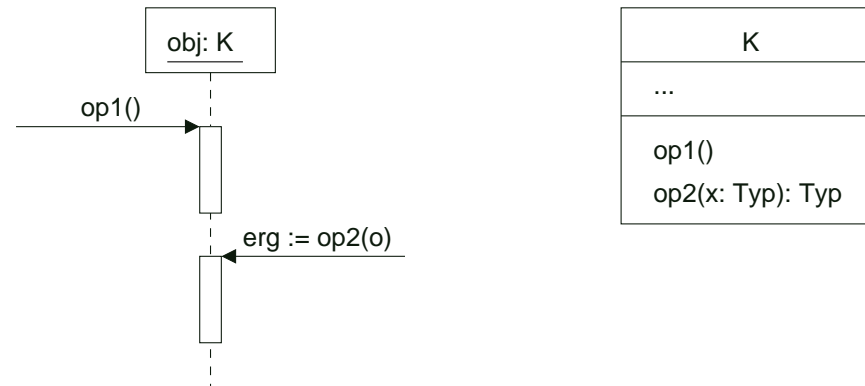
1. Operationen hinzufügen
2. Assoziationen ausrichten
3. Zugriffsrechte bestimmen
4. Mehrfachvererbung auflösen
5. Wiederverwendung von Klassen

## 4.1.1 Operationen hinzufügen

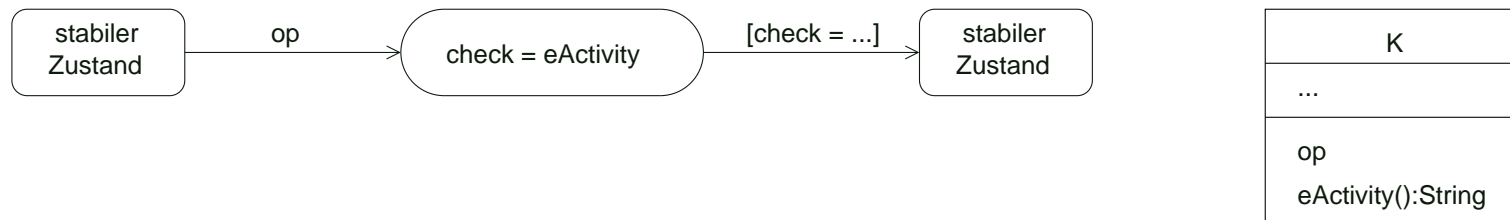
### Vorgehensweise

Sei K eine Klasse des Objektmodells.

- Führe eine Operation für jede an ein Objekt von K gesendete Nachricht ein.



- Führe eine Operation für jeden Aktivitätszustand (eines Zustands- oder Aktivitätsdiagramms) des Analysemodells ein.



*Beispiel ATM:*

ATM
geldvorrat: Real grenzen: Real
karteEin geheimzahlEin formEin betragEin geldEntnehmen beenden karteBelegEntnehmen
karteEinActivity(): String geheimzahlEinActivity(): String betragEinActivity(): String karteLesen(): String abschluss() geheimzahlUeberpruefen(): String grenzenUeberpruefen(): String

Konsortium
name: String
karteUeberpruefen(): String transaktionVerarbeiten(): String blzUeberpruefen(): String

Bank
blz: Integer name: String
karteBankUeberpruefen(): String bankTransaktionVerarbeiten(): String kartennrUeberpruefen(): String kontoAktualisieren(): String

- Algorithmen der Operationen beschreiben

*Input:* Interaktions- oder Aktivitätsdiagramme der Analyse

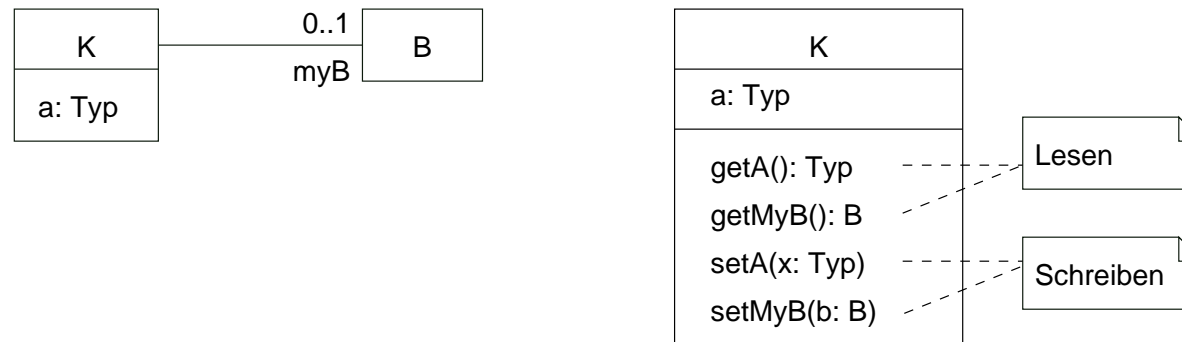
*Mögliche Darstellungen der Algorithmen:*

- Detaillierte Aktivitätsdiagramme (ggf. auch vollständige Interaktionsdiagramme)
- Pseudo-Code (z.B. basierend auf Java)

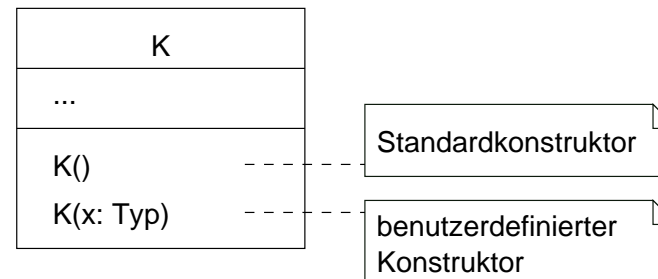
*Beachte:*

Während der Formulierung der Algorithmen wird das Objektmodell laufend überarbeitet.

- Benötigte Zugriffsoperationen für Attribute und Rollen hinzufügen (zum Lesen und/oder Schreiben)



- Konstruktoren bei nicht abstrakten Klassen hinzufügen



### Aufruf eines Konstruktors:

- im Sequenzdiagramm (ähnlich im Kollaborationsdiagramm):



- im Pseudocode: `k = new K();` //falls k ein Rollenname ist  
`K k = new K();` //falls k eine lokale Variable ist

## 4.1.2 Assoziationen ausrichten

- Analysiere in welcher/welchen Richtung(en) eine Assoziation (beim Senden von Nachrichten bzw. Operationsaufrufen) durchlaufen wird.
- Falls eine Assoziation nur in einer Richtung durchlaufen wird ("unidirektional"), markiere das entsprechende Assoziationsende mit einer Pfeilspitze. (Default ist "bidirektional")
- Füge ggf. noch fehlende Multiplizitäten und explizite Rollennamen hinzu.

*Beispiel ATM:*



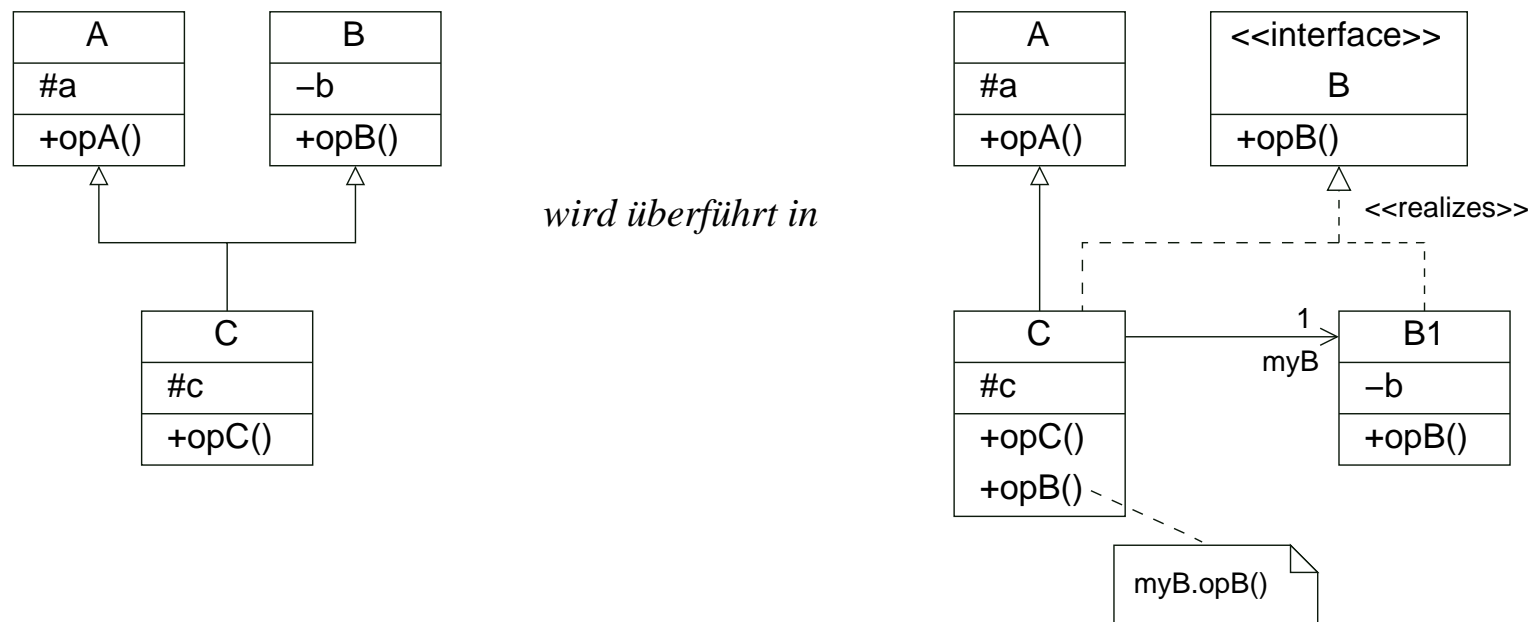
### 4.1.3 Zugriffsrechte bestimmen

- Häufig sollen nur bestimmte Merkmale der Objekte einer Klasse von außen zugreifbar sein (*"Kapselungsprinzip"*).
- Zur Zugriffskontrolle verwendet man *Sichtbarkeitsmarkierungen* für Attribute, Rollennamen und Operationen:
  - +name d.h. öffentlich zugreifbar (*"public"*)
  - -name d.h. nur innerhalb der Klasse verwendbar (*"private"*)
  - #name d.h. nur innerhalb der Klasse und in allen Subklassen verwendbar (*"protected"*)
- **Regel:** Attribute sollten nicht öffentlich zugreifbar sein!

K
-a: Typ
#b: Typ
+getA(): Typ
+setA(x: Typ)
+getB(): Typ

## 4.1.4 Mehrfachvererbung auflösen

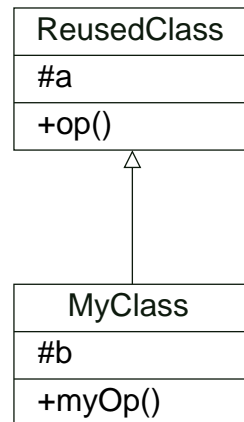
- Ist notwendig, wenn die Zielsprache keine Mehrfachvererbung für Klassen unterstützt (z.B. Java).
- Die Auflösung der Mehrfachvererbung ist möglich durch Einführung einer Schnittstelle.



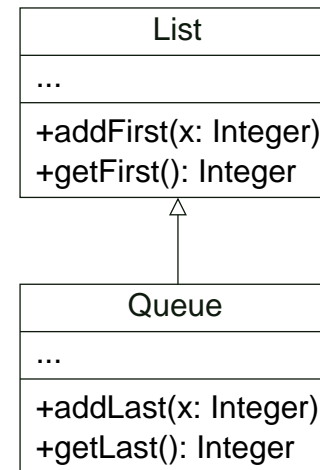
## 4.1.5 Wiederverwendung von Klassen

- Häufig ist es günstig, schon vorhandene (wohl erprobte und qualitativ hochwertige) Klassen im Entwurf wiederzuverwenden.
- Wenn eine wiederverwendete Klasse noch nicht alle gewünschten Merkmale besitzt, können diese durch *Spezialisierung* in einer Subklasse hinzugenommen werden.

*Allgemein:*



*Beispiel:*



## 4.1.6 Objektentwurf für ATM

- Es werden Algorithmen für die in Abschnitt 4.1.1 identifizierten Operationen angegeben (in Java-Pseudo-Code).
- Zusätzlich benötigte Konstruktoren und Zugriffsoperationen ("getter" und "setter") werden identifiziert.
- Das Klassendiagramm der Analyse wird entsprechend überarbeitet.
- Der Objektentwurf soll später durch eine Computersimulation für das ATM implementiert werden, bei der alle benötigten Daten (Kartenummer, BLZ, ...) vom Benutzer am Monitor eingegeben werden.
- Im Objektentwurf wird die Benutzerschnittstelle noch durch eine nicht näher spezifizierte Klasse "GUI" repräsentiert.



## Algorithmen

### 1. Operationen der Klasse ATM

Operationen, die als Ereignisse im Zustandsdiagramm vorkommen (karteEin, geheimzahlEin, ..., karteBelegEntnehmen) werden hier nicht betrachtet. Eine geeignete Behandlung wird später beim Entwurf der Benutzerschnittstelle gegeben.

```
// Konstruktor ATM
ATM() {
    geldvorrat = 100000;
    grenzen    = 250;
    gui.display("Kartennr, BLZ und gespeicherte Geheimzahl?");
}
```

```
abschluss() {
    gui.diplay("Beleg drucken");
    gui.diplay("Karte ausgeben");
    gui.diplay("Karte und Beleg entnehmen?");
}
```

```
karteEinActivity(): String {
    String check = karteLesen();

    if (check.equals("lesbar")) {
        gui.display("Geheimzahl?");
        return "Karte lesbar";
    }
    else if (check.equals("nicht lesbar")) {
        gui.display("Karte nicht lesbar");
        abschluss();
        return "Karte nicht lesbar";
    }
}

karteLesen(): String {
    aktKartennr    = gui.getKartennr();    // neues Attribut von ATM
    aktBLZ        = gui.getBLZ();        // neues Attribut von ATM
    aktGeheimzahl = gui.getGeheimzahl(); // codierte Geheimzahl, neues Attribut von ATM
    return "lesbar";
}
```

```
geheimzahlEinActivity(): String {
    Integer typedGeheimzahl = gui.getTypedGeheimzahl();
    String check = geheimzahlUeberpruefen(typedGeheimzahl);
    if (check.equals("Geheimzahl ok")) {
        // neues Attribut aktKontonr
        check = konsortium.karteUeberpruefen(aktKartennr, aktBLZ, aktKontonr);
        if (check.equals("Karte ok")) {
            gui.display("Transaktionsform?");
            return "Karte ok";
        }
        else if (check.equals("falsche BLZ")) {
            gui.display("falsche BLZ");
            abschluss();
            return "Karte nicht ok";
        }
        else if (check.equals("Karte gesperrt")) {
            gui.display("Karte gesperrt");
            abschluss();
            return "Karte nicht ok";
        }
    }
}
```

```
    else if (check.equals("Geheimzahl falsch")) {  
        gui.display("Geheimzahl?");  
        return "Geheimzahl falsch";  
    }  
}  
  
geheimzahlUeberpruefen(tgz: Integer): String {  
    if (tgz == aktGeheimzahl) return "Geheimzahl ok";  
    else return "Geheimzahl falsch";  
}
```

```
betragEinActivity(): String {
    Real betrag = gui.getBetrag();
    String check = grenzenUeberpruefen(betrag);

    if (check.equals("Grenzen ok")) {
        check = konsortium.transaktionVerarbeiten(aktBLZ, aktKontonr, betrag);
        if (check.equals("Transaktion erfolgreich")) {
            Aussentransaktion atrans =
                new Aussentransaktion("Abhebung", aktKartennr, betrag, aktBLZ, aktKontonr);
            addTransaktion(atrans); // neue Operation von Terminal
            geldvorrat = geldvorrat-betrag;
            gui.display("Geld ausgeben");
            gui.display("Geld entnehmen?");
            return "Transaktion erfolgreich";
        }
        else if (check.equals("Transaktion gescheitert")) {
            gui.display("Transaktion gescheitert");
            gui.display("Form?");
            return "Transaktion gescheitert";
        }
    }
}
```

```
else if (check.equals("Grenzen überschritten")) {  
    gui.display("Betrag?");  
    return "Grenzen überschritten";  
}  
}
```

```
grenzenUeberpruefen(b: Real): String {  
    if (b <= grenzen) return "Grenzen ok";  
    else return "Grenzen überschritten";  
}
```

## 2. Operationen der Klasse *Konsortium*

```
karteUeberpruefen(kartennr: Integer, blz: Integer, out kontonr: Integer): String {
    String check = blzUeberpruefen(blz);

    if (check.equals("BLZ richtig")) {
        check = banken[blz].karteBankUeberpruefen(kartennr, kontonr);
        if (check.equals("Karte ok")) return "Karte ok";
        else if (check.equals("Karte bei Bank gesperrt")) return "Karte gesperrt";
    }
    else if (check.equals("BLZ falsch")) return "falsche Bankleitzahl";
}

blzUeberpruefen(blz: Integer): String {
    if (banken[blz] != null) return "BLZ richtig";
    else return "BLZ falsch";
}
```

```
transaktionVerarbeiten(blz: Integer, kontonr: Integer, b: Real): String {  
    String check = banken[blz].bankTransaktionVerarbeiten(kontonr, b);  
  
    if (check.equals("Banktransaktion erfolgreich"))  
        return "Transaktion erfolgreich";  
    else if (check.equals("Banktransaktion gescheitert"))  
        return "Transaktion gescheitert";  
}
```

### 3. Operationen der Klasse Bank

```
karteBankUeberpruefen(kartennr: Integer, out kontonr: Integer): String {  
    String check = kartennrUeberpruefen(kartennr, kontonr);  
  
    if (check.equals("gueltig")) return "Karte ok";  
    else if (check.equals("gesperrt")) return "Karte bei Bank gesperrt";  
}
```

```
kartennrUeberpruefen(kartennr: Integer, out kontonr: Integer) :String {  
    // kreditkarten ist Rollename einer neuen (abgeleiteten) qualifizierten  
    // Assoziation zwischen Bank und Kreditkarte  
    if (kreditkarten[kartennr] != null) {  
        // getKonto() und getKontonr() werden als Zugriffsoperationen bei  
        // Kreditkarte bzw. Konto gebraucht  
        kontonr = kreditkarten[kartennr].getKonto().getKontonr();  
        if (! kreditkarten[kartennr].getGesperrt()) return "Karte ok";  
        else return "Karte bei Bank gesperrt";  
    }  
}
```

```
bankTransaktionVerarbeiten(kontonr: Integer, b: Real): String {  
    String check = kontoAktualisieren(kontonr, b);  
  
    if (check.equals("erfolgreich")) return "Banktransaktion erfolgreich";  
    else if (check.equals("gescheitert")) return "Banktransaktion gescheitert";  
}
```

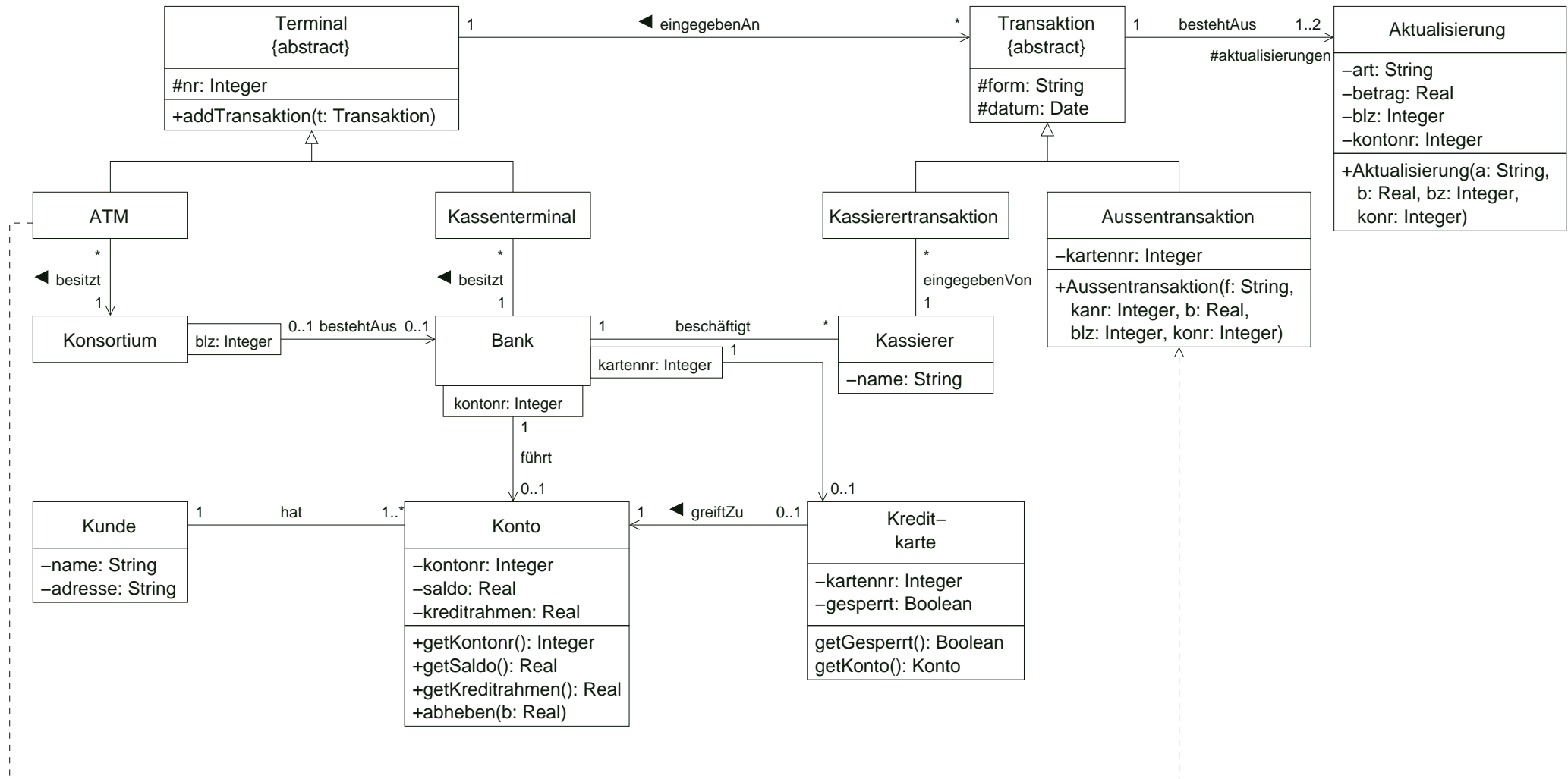
```
kontoAktualisieren(kontonr: Integer, b: Real): String {  
    Konto k = konten[kontonr];  
    if (k.getSaldo()-b >= k.getKreditrahmen()) {  
        k.abheben(b);  
        return "erfolgreich";  
    }  
    else return "gescheitert";  
}
```

## 4. *Konstruktoren für Aussentransaktion und Aktualisierung*

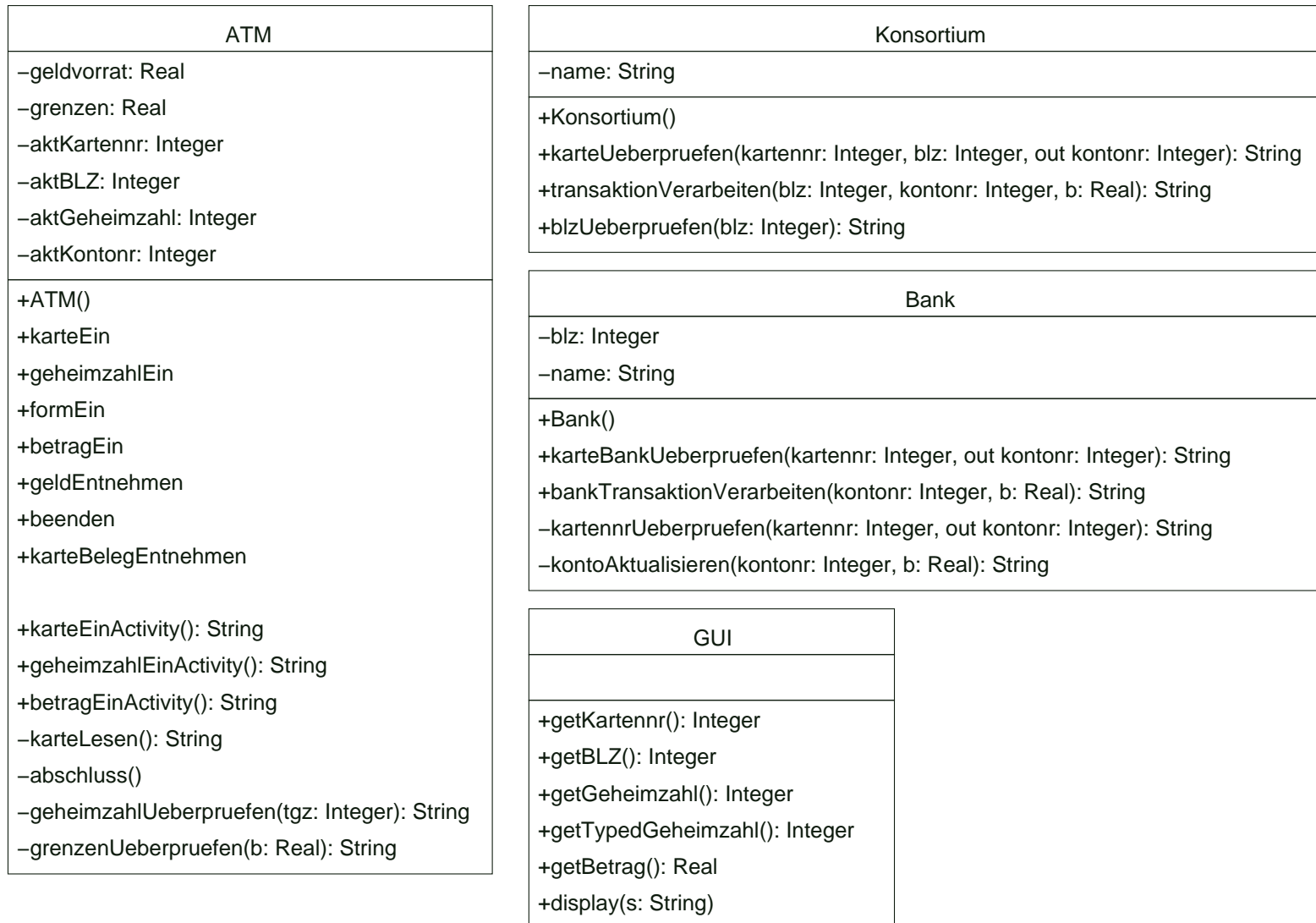
```
Aussentransaktion(f: String, kanr: Integer, b: Real, blz: Integer, konr: Integer) {  
    form = f;  
    datum = new date();  
    // Neues Attribut kartennr von Aussentransaktion.  
    // Dafür wird die Assoziation zu Kreditkarte gestrichen.  
    kartennr = kanr;  
  
    if (form.equals("Abhebung"))  
        aktualisierungen[0] = new Aktualisierung("Lastschrift", b, blz, konr);  
  
    else if (form.equals("Einzahlung"))  
        aktualisierungen[0] = new Aktualisierung("Gutschrift", b, blz, konr);  
}
```

```
Aktualisierung(a: String, b: Real, bz: Integer, konr: Integer) {  
  art = a;  
  betrag = b;  
  // blz und kontonr sind neue Attribute von Aktualisierung.  
  // Dafür wird die Assoziation zu Konto gestrichen.  
  blz = bz;  
  kontonr = konr;  
}
```

# Klassendiagramm von ATM nach dem Objektentwurf



## Überarbeitete Klassen



## Zusammenfassung von Abschnitt 4.1

- Der Objektentwurf ergibt sich aus der Integration des statischen und dynamischen Modells der Analyse (wobei die Behandlung von Zustandsdiagrammen gesondert im nächsten Abschnitt beschrieben wird).
- Im Objektentwurf werden Operationen zu den Klassen hinzugenommen.
- Während der Formulierung von Algorithmen für die Operationen wird das statische Modell laufend überarbeitet (z.B. Ausrichten von Assoziationen, Bestimmung von Zugriffsrechten).
- Weitere typische Aufgaben des Objektentwurfs betreffen die Auflösung von Mehrfachvererbung und die Wiederverwendung von Klassen.

## 4.2 Realisierung von Zustandsdiagrammen

### Gegeben

Zustandsdiagramm einer Klasse K.

### Ziel

Objektentwurf mit Algorithmen zur Realisierung des durch das Zustandsdiagramm beschriebenen Verhaltens.

*Wir unterscheiden vier Möglichkeiten:*

- Prozedurgesteuerte Realisierung
- Realisierung durch Fallunterscheidung
- Realisierung durch Zustandsobjekte
- Realisierung durch eine Zustandsmaschine

## 4.2.1 Prozedurgesteuerte Realisierung

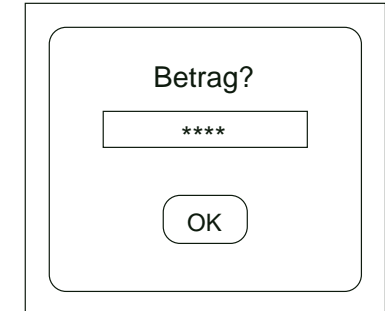
### *Idee*

Ereignisse werden durch "modale Dialoge" (erzwungene Benutzereingaben) realisiert.

### *Voraussetzung*

Objekte befinden sich an der Systemgrenze (Kontrollobjekte zur Dialogsteuerung)

### *Beispiel:*



in Pseudo-Code: `betrag = gui.get("Betrag?");`

in Java: `String betrag = JOptionPane.showInputDialog("Betrag?");`

## *Vorgehensweise*

Das gesamte Zustandsdiagramm wird überführt in eine Prozedur mit

- modalen Dialogen für die (externen) Ereignisse
- Aufrufe von Subprozeduren für die Aktivitäten
- bedingten Anweisungen für Verzweigungen
- Wiederholungsanweisungen für Zyklen des Diagramms

## *Beispiel:* Prozedurgesteuerte Implementierung des ATM

```
while(true) do {
  // mit modalen Dialogen der Form aktKartennr = gui.get("Kartennr?");
  check = KarteEinActivity();

  if (check.equals("Karte nicht lesbar"))
    gui.get("Karte und Beleg entnehmen?");

  else if (check.equals("Karte lesbar")) {
    // mit modalem Dialog typedGeheimzahl = gui.get("Geheimzahl?")
    do check = GeheimzahlEinActivity();
    while (check.equals("Geheimzahl falsch"));

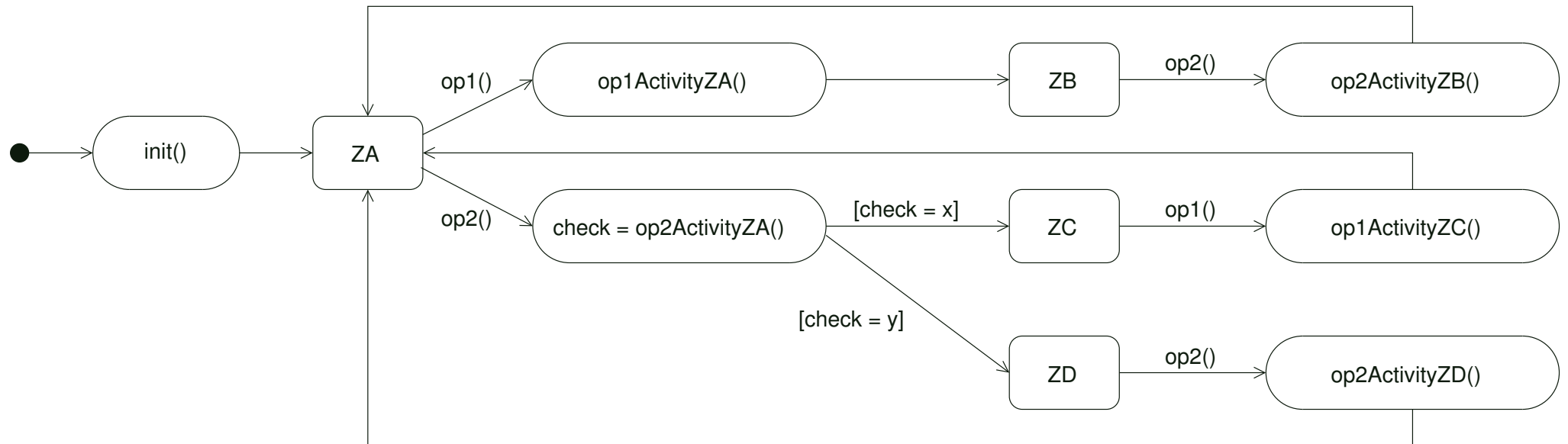
    if (check.equals("Karte nicht ok"))
      gui.get("Karte und Beleg entnehmen?");
```

```
else if (check.equals("Karte ok")) {
  do {
    gui.get("Transaktionsform?");
    // mit modalem Dialog betrag = gui.get("Betrag?");
    do check = BetragEinActivity();
    while (check.equals("Grenzen überschritten"));
  }
  while (check.equals("Transaktion gescheitert"));

  if (check.equals("Transaktion erfolgreich")) {
    gui.get("Geld entnehmen?");
    gui.get("Beenden?");
    abschluss();
    gui.get("Karte und Beleg entnehmen?");
  }
}
}
```

## 4.2.2 Realisierung durch Fallunterscheidung

Gegeben sei folgendes Zustandsdiagramm für die Objekte einer Klasse K:



## *Vorgehensweise*

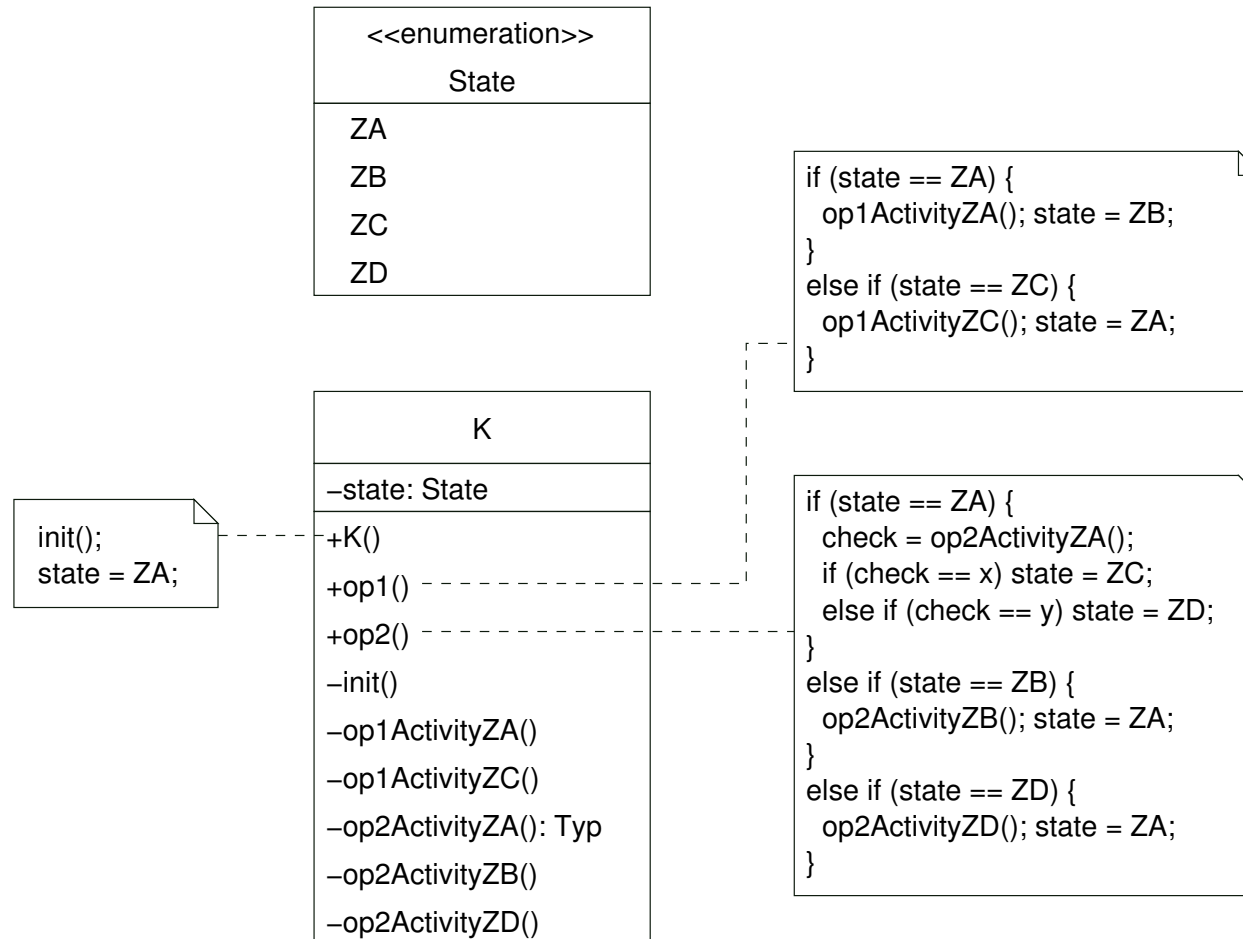
- Verwende einen Enumerationstyp zur Darstellung der (endlich vielen) stabilen Zustände.
- Führe ein explizites Zustandsattribut für die betrachtete Klasse ein.
- Realisiere die zustandsabhängigen Operationen durch Fallunterscheidung nach dem aktuellen (stabilen) Zustand.

## *Nachteil*

Schlechte Erweiterbarkeit bzgl. neuer Zustände (neue Fälle bei *jeder* zustandsabhängigen Operation hinzunehmen).

## *Vorteil*

Einfache Erweiterbarkeit bzgl. neuer Operationen.



### Bemerkung

Falls Typ = String, ersetze `check == x` durch `check.equals("x")`!

## 4.2.3 Realisierung durch Zustandsobjekte

### *Idee*

- Jedes Objekt der Klasse ist mit einem Zustandsobjekt verbunden, das den aktuellen (stabilen) Zustand des Objekts repräsentiert.
- Der Aufruf einer zustandsabhängigen Operation wird an das Zustandsobjekt delegiert.
- Das aktuelle Zustandsobjekt führt die gewünschte Aktivität aus.
- Bei Zustandsänderung wird ein neues Zustandsobjekt (der passenden Unterklasse) erzeugt und mit dem Basisobjekt verbunden.

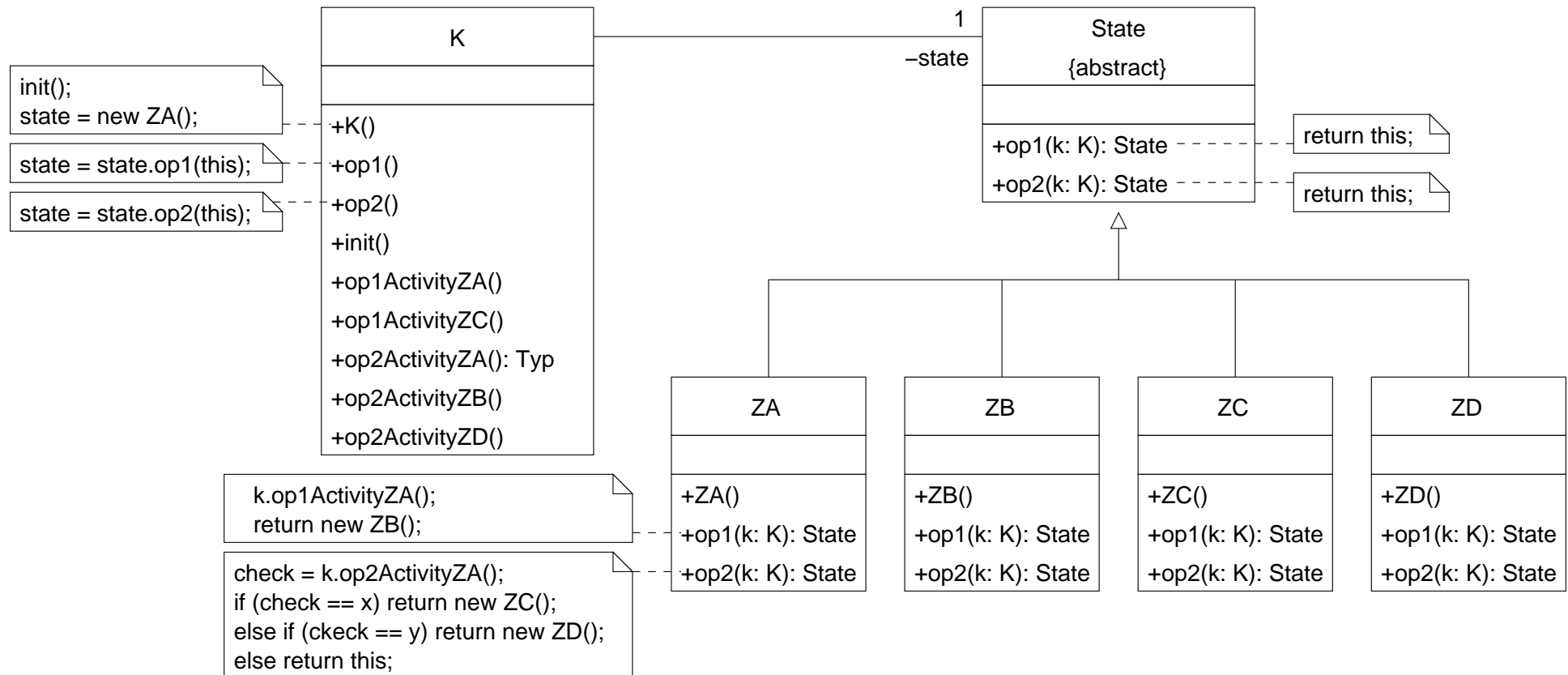
### *Vorteil*

Einfache Erweiterbarkeit bzgl. neuer Zustände.

### *Nachteil*

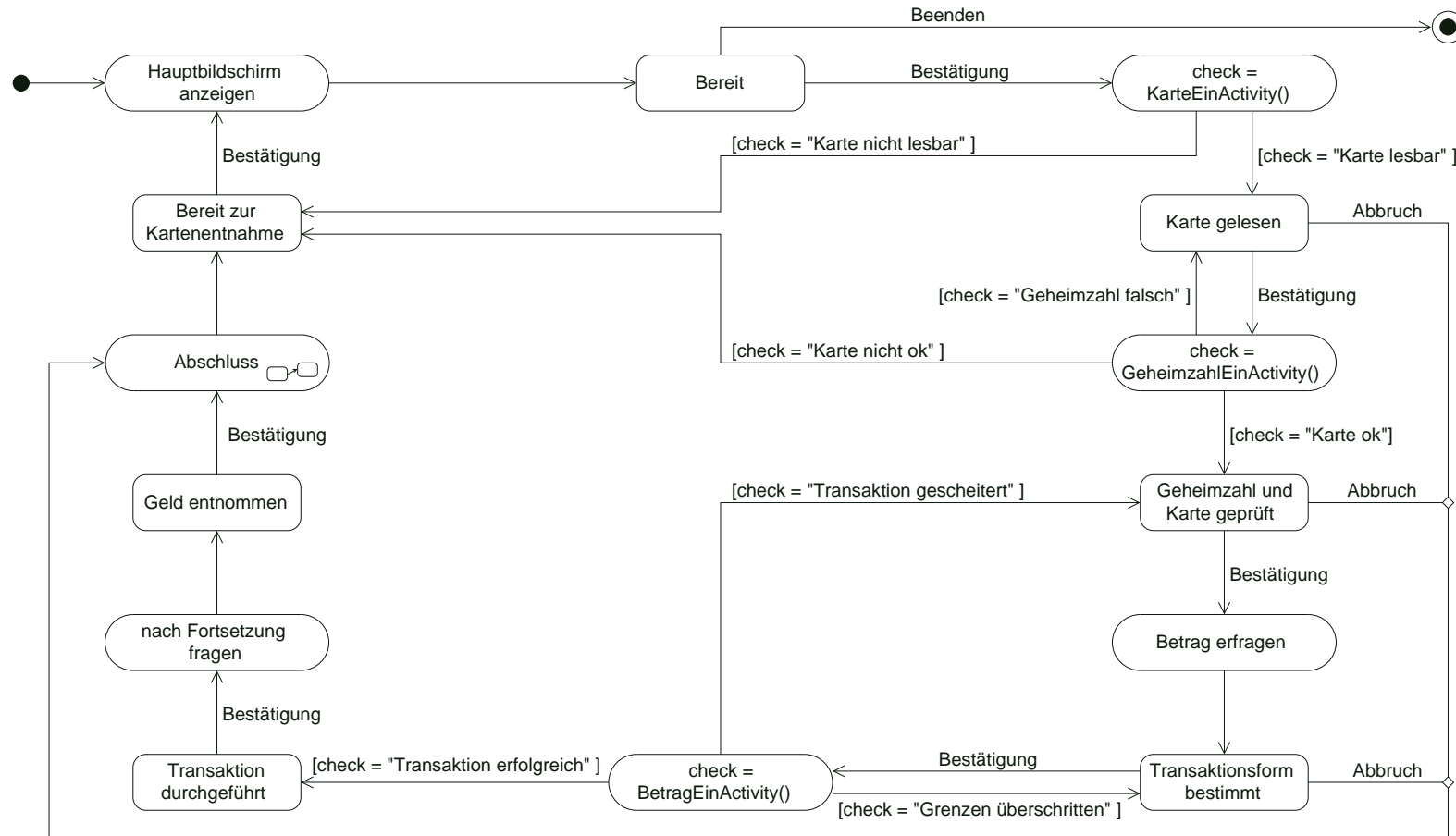
Schlechte Erweiterbarkeit bzgl. neuer Operationen.

Das Zustandsdiagramm von oben wird folgendermaßen realisiert:



Beispiel: Realisierung des ATM durch Zustandsobjekte

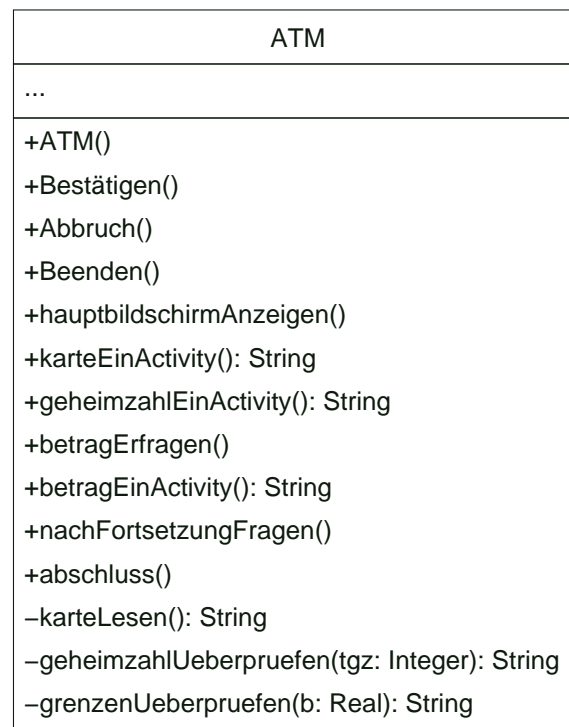
## Zustandsdiagramm für ereignisgesteuerte ATM-Simulation



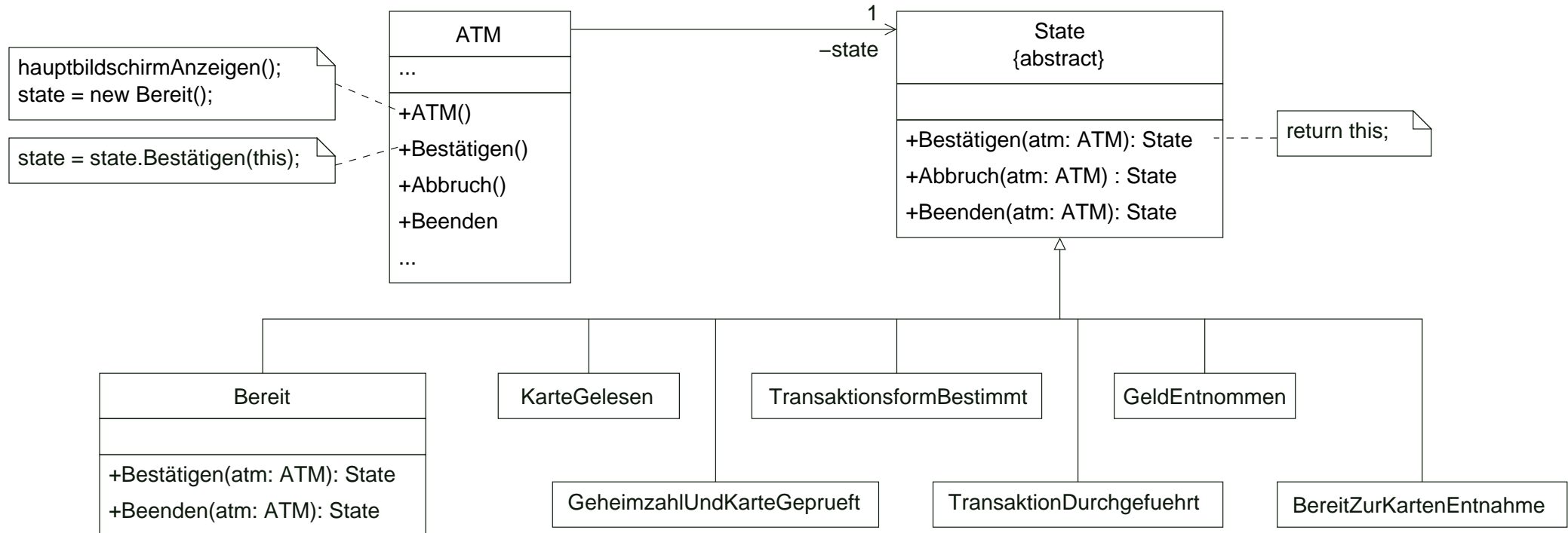
Die Ereignisse "Bestätigung", "Abbruch" und "Beenden" werden durch die Betätigung entsprechender Buttons der Benutzerschnittstelle hervorgerufen.

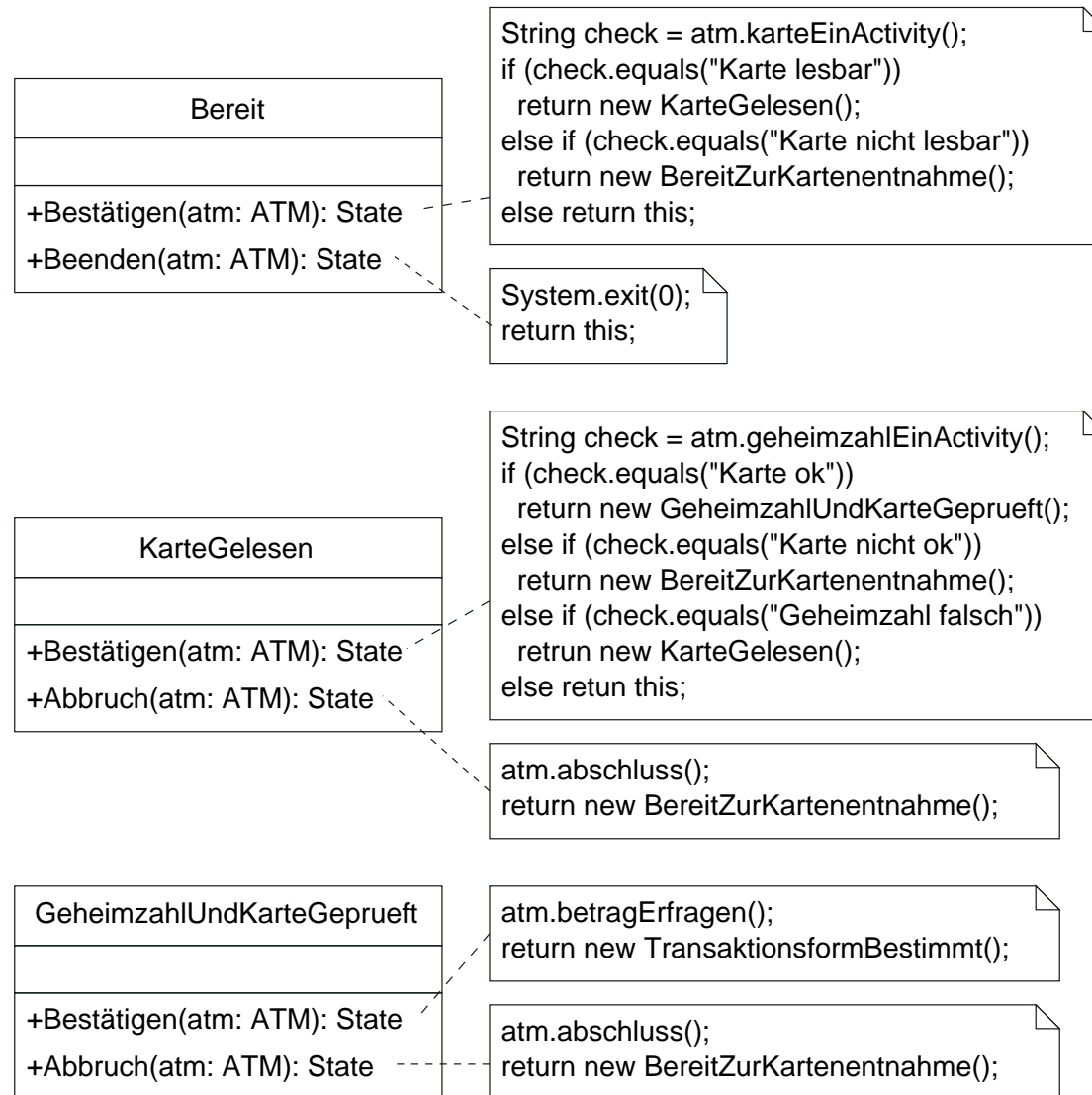
## Überarbeitete Klasse ATM

- Die Operationen "karteEin", ..., "karteBelegEntnehmen" von früher werden nicht mehr benötigt.
- Zusätzlich zu den Aktivitätszuständen werden Operationen für die Aktionszustände "Hauptbildschirm anzeigen", "Betrag erfragen" und "nach Fortsetzung fragen" eingeführt.



## Realisierung des Zustandsdiagramms





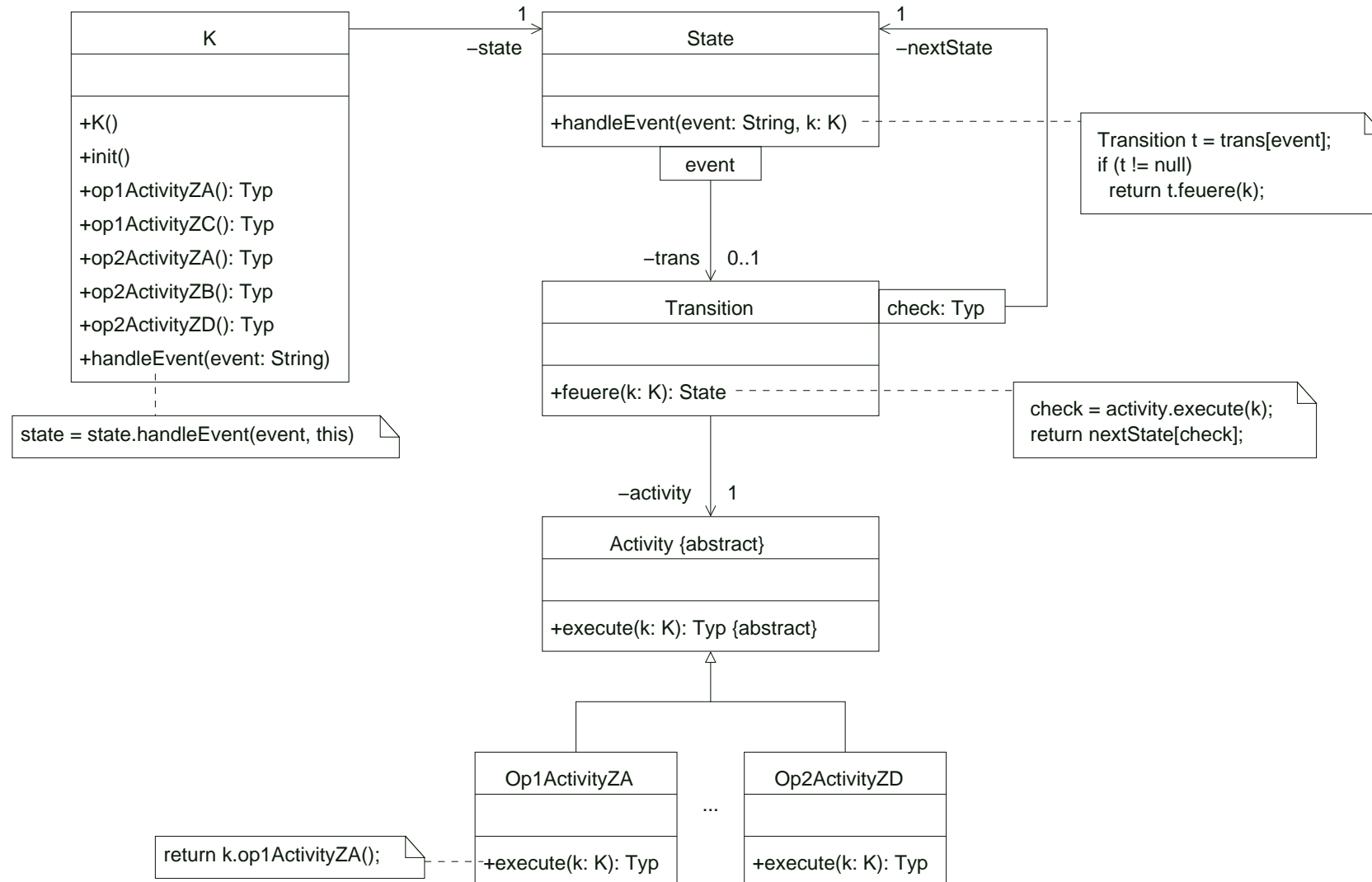
Analog werden die vier übrigen Zustandsklassen implementiert.

## 4.2.4 Realisierung durch eine Zustandsmaschine

### *Idee*

- Alle in einem Zustandsdiagramm vorkommende Größen (Zustände, Transitionen, Aktivitäten) werden durch Objekte dargestellt.
- Ereignisse werden von einer speziellen "Event-Handle"-Operation interpretiert.

Das Zustandsdiagramm von oben wird durch folgende Zustandsmaschine realisiert:



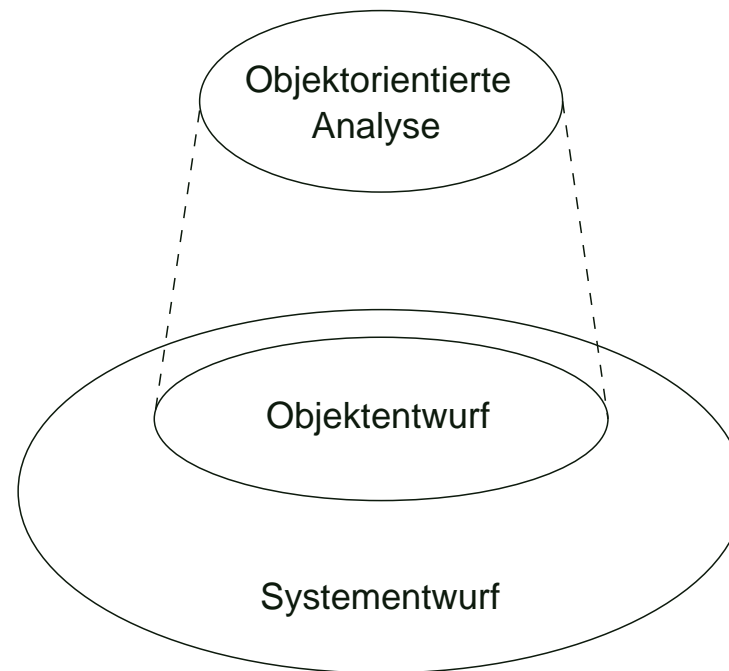
## Zusammenfassung von Abschnitt 4.2

- Zustandsdiagramme können systematisch realisiert und in einen Objektentwurf integriert werden.
- Wir unterscheiden 4 Möglichkeiten der Realisierung:
  - Prozedurgesteuert
  - Fallunterscheidung
  - Zustände als Objekte
  - Zustandsmaschine

## 4.3 Systementwurf

### Ziele

- Einbettung des Objektentwurfs in die Systemumgebung
- Festlegung der Systemarchitektur



## 4.3.1 Pakete und Subsysteme

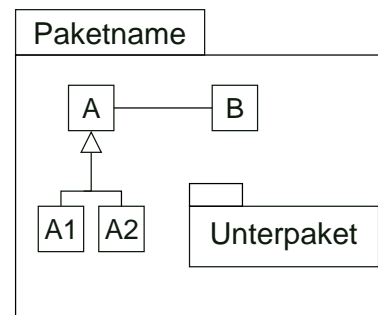
Pakete dienen zur Strukturierung von Modellen größerer Systeme. Sie fassen verschiedene Modellelemente in einer Einheit zusammen.

### Darstellung von Paketen in UML

*Paket ohne Darstellung der Inhalte:*



*Paket mit Darstellung der Inhalte:*



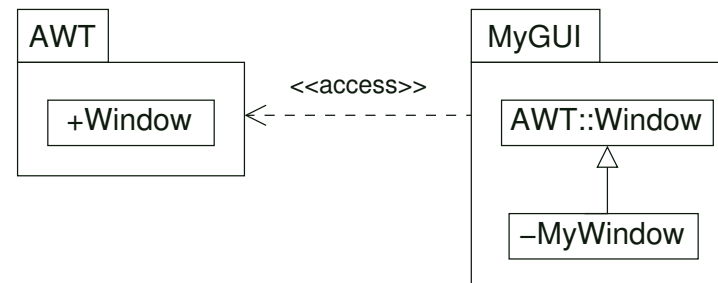
## Abhängigkeiten zwischen Paketen

### 1. Zugriffsabhängigkeit:



Elemente von Paket 1 können (gemäß ihrer Sichtbarkeit) in Paket 2 referenziert werden. Bei der Referenzierung werden Pfadangaben verwendet.

### Beispiel:

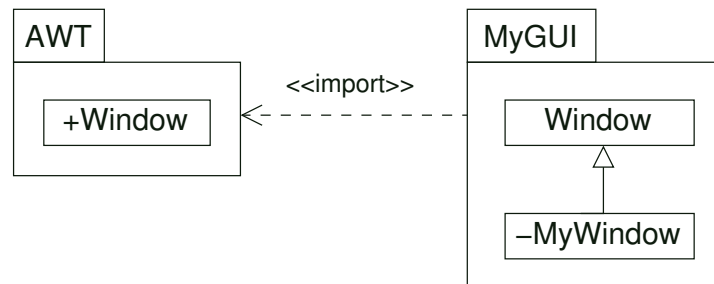


## 2. Importabhängigkeit:



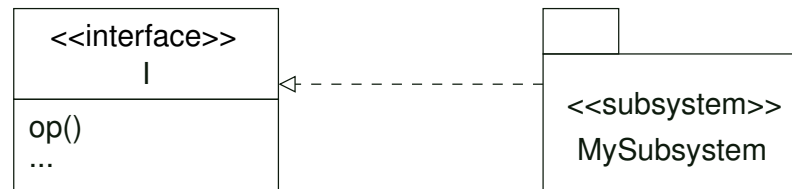
Elemente von Paket 1 können (gemäß ihrer Sichtbarkeit) direkt (d.h. ohne Pfadangaben) in Paket 2 referenziert werden. Etwaige Namenskonflikte sind vorher aufzulösen.

*Beispiel:*

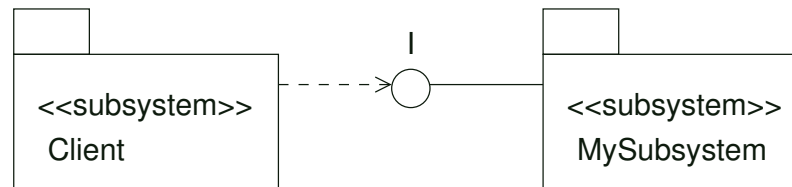


## Subsysteme

Sind spezielle Formen von Paketen, die mit Schnittstellen versehen werden können.



**Kurzform** (mit Benutzung der Schnittstelle)



## 4.3.2 Systemarchitektur

Beschreibt die Gesamtstruktur des SW-Systems durch Angabe von

- Systemkomponenten (Subsystemen)
- Beziehungen zwischen den Komponenten (ggf. mit Schnittstellen)

*Bemerkung:*

Eine grobe Systemarchitektur wird häufig schon zu Beginn der Systementwicklung angegeben.

### *Grundregeln*

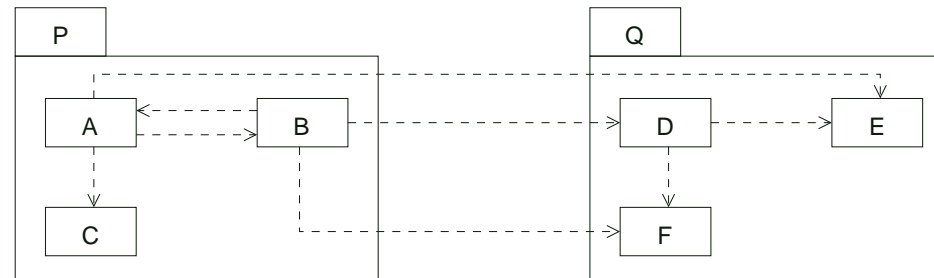
- *Hohe Kohärenz* (high cohesion)  
Zusammenfassung (logisch) zusammengehörender Teile eines Systems in einer Komponente.
- *Geringe Kopplung* (low coupling)  
Wenige Abhängigkeiten zwischen den einzelnen Komponenten.

*Vorteil:*

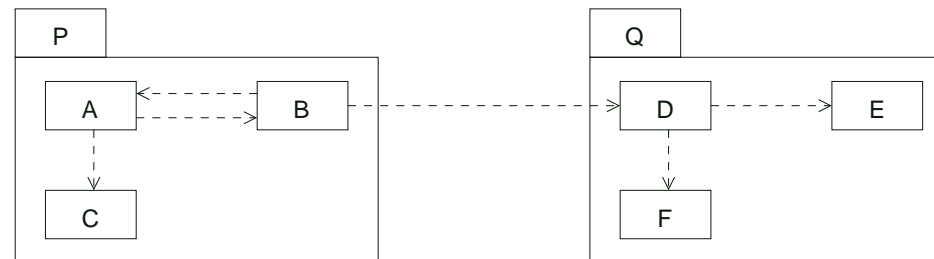
Leichte Änderbarkeit und Austauschbarkeit von einzelnen Teilen.

*Beispiel:*

## Komponenten mit hoher Kopplung



## Komponenten mit geringer Kopplung



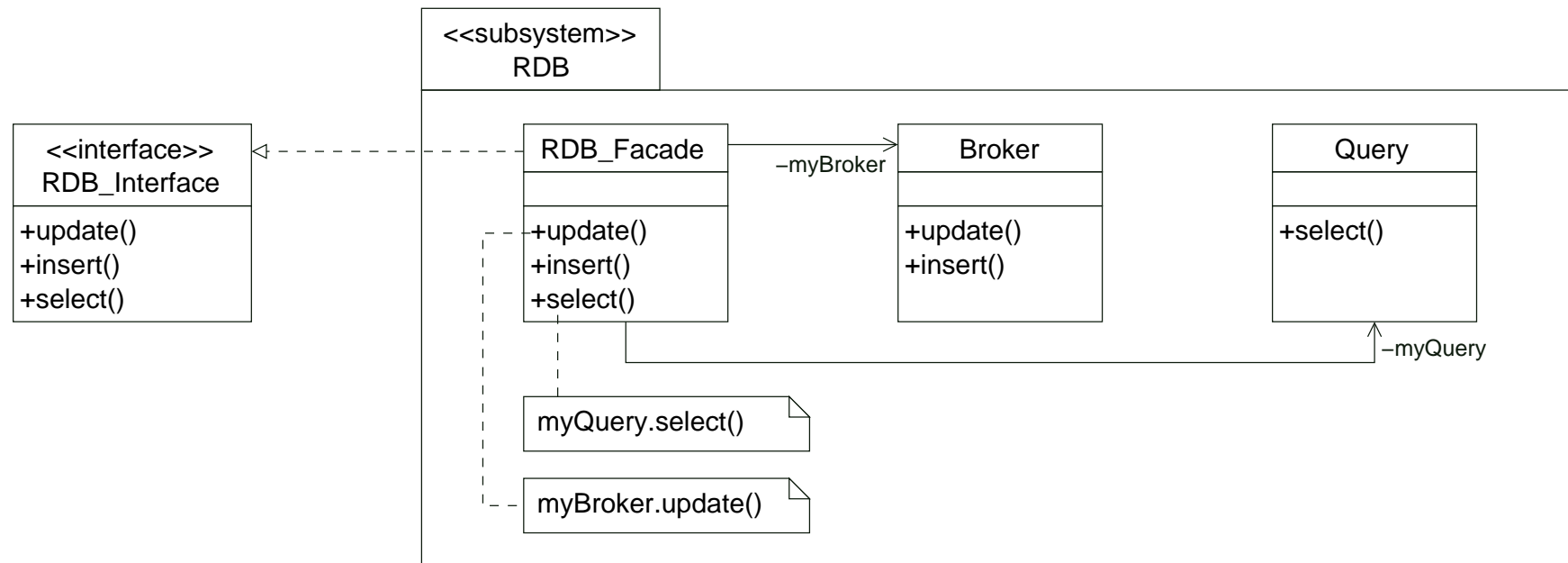
## *Beachte*

Wird an einem Teil T etwas geändert, so müssen alle anderen Teile, die eine Abhängigkeitsbeziehung hin zu T haben, auf etwaige nötige Änderungen überprüft werden.

## Fassadenklassen

- Hilfsmittel zur Erzielung geringer Kopplung.
- Fassen die Dienste verschiedener Klassen einer Komponente zusammen und delegieren Aufrufe an die "zuständigen" Objekte.
- Realisieren häufig ein Interface einer Komponente (eines Subsystems)

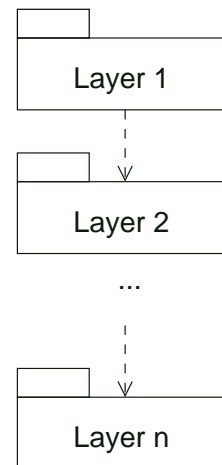
*Beispiel:*



## Schichtenarchitekturen

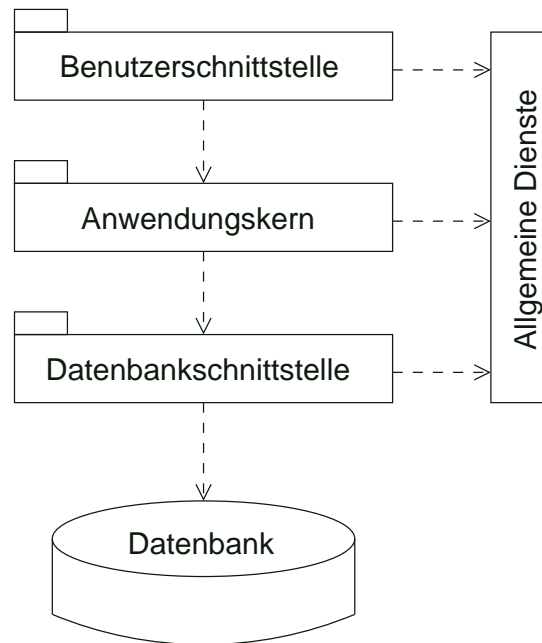
In vielen Systemen findet man *Schichtenarchitekturen*, wobei jede untere Schicht Dienste für die darüberliegende(n) Schicht(en) bereitstellt.

*z.B. OSI-Schichtenmodell für Netzwerkprotokolle, Betriebssystemschichten, ...*



- Bei "geschlossenen" Architekturen darf eine Schicht nur auf die direkt darunterliegende Schicht zugreifen; sonst spricht man von offenen "Architekturen".
- Sind verschiedene Schichten auf verschiedene Rechner verteilt, dann spricht man von Client/Server-Systemen.
- Eine Schicht kann selbst wieder aus verschiedenen Komponenten bestehen.

### 4.3.3 Drei-Schichten-Architektur für betriebliche Informationssysteme



#### *Bemerkung*

Bei Client/Server-Architekturen spricht man

- von einem "Thick-Client", wenn Benutzerschnittstelle und Anwendungskern auf demselben Rechner ausgeführt werden,
- von einem "Thin-Client", wenn Benutzerschnittstelle und Anwendungskern auf verschiedene Rechner verteilt sind.

## *Benutzerschnittstelle*

- Behandlung von Terminalereignissen (Maus-Klick, Key-Strike, ...)
- Ein-/Ausgabe von Daten
- Dialogkontrolle

## *Anwendungskern* (Fachkonzept)

- Zuständig für die Anwendungslogik (die eigentlichen Aufgaben des Problembereichs)
- Ergibt sich aus dem Objektentwurf

## *DB-Schnittstelle*

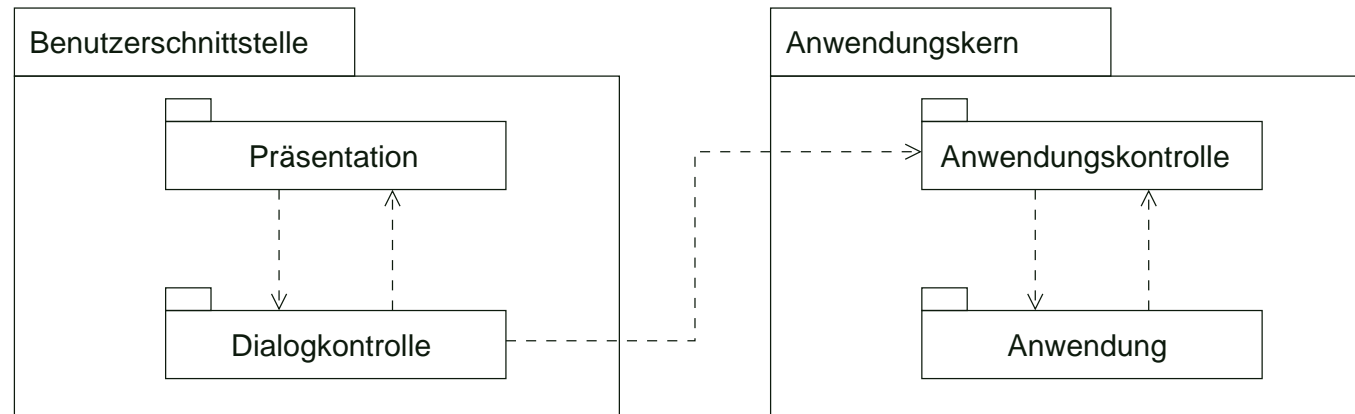
Sorgt für die Speicherung von und den Zugriff auf persistente Daten der Anwendung.

## *Allgemeine Dienste*

z.B. Kommunikationsdienste, Dateiverwaltung, Bibliotheken (APIs, GUI, DB, math. Funktionen, ...)

## Kontrollobjekte

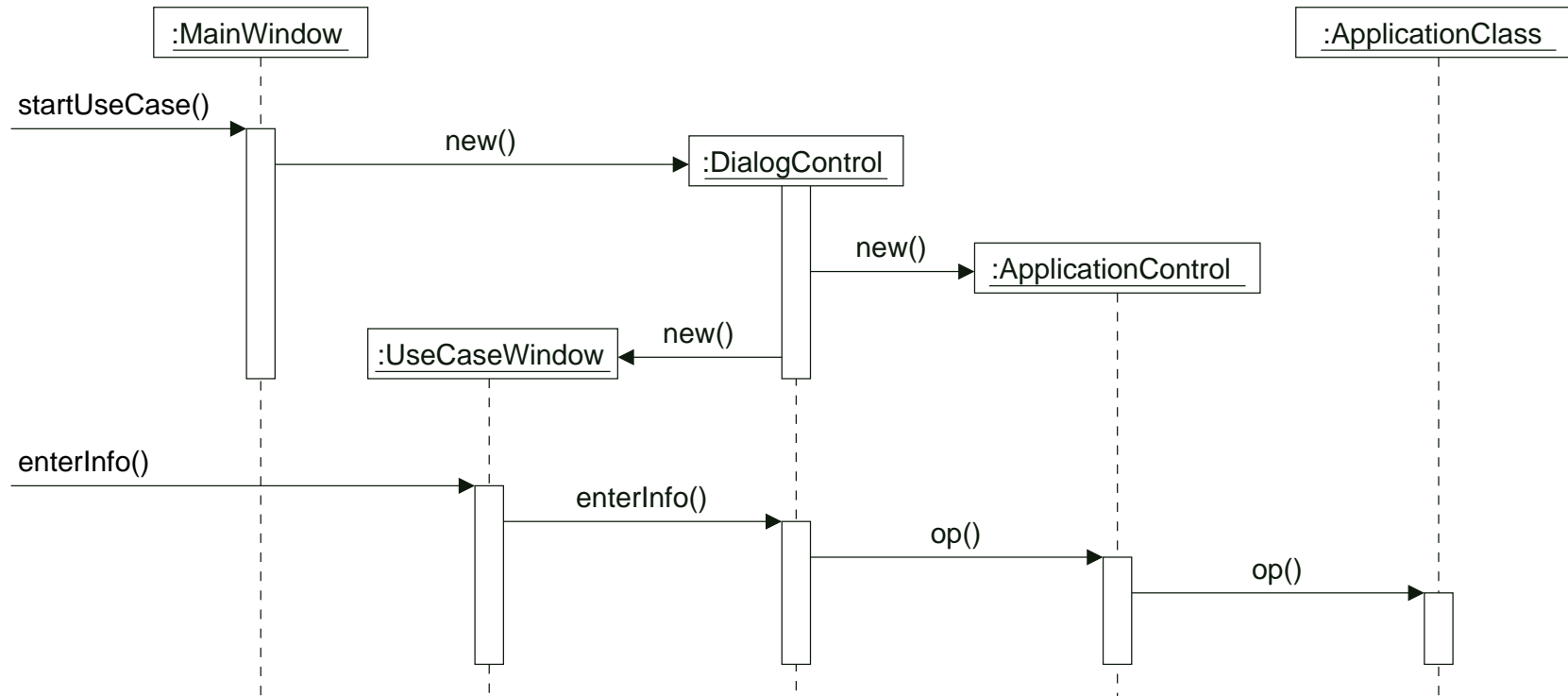
Häufig werden eigene Objekte zur Dialogsteuerung (z.B. Verwaltung mehrerer Fenster) oder zur Steuerung der Aufgaben des Anwendungskerns verwendet (z.B. ein Kontrollobjekt pro Use Case).



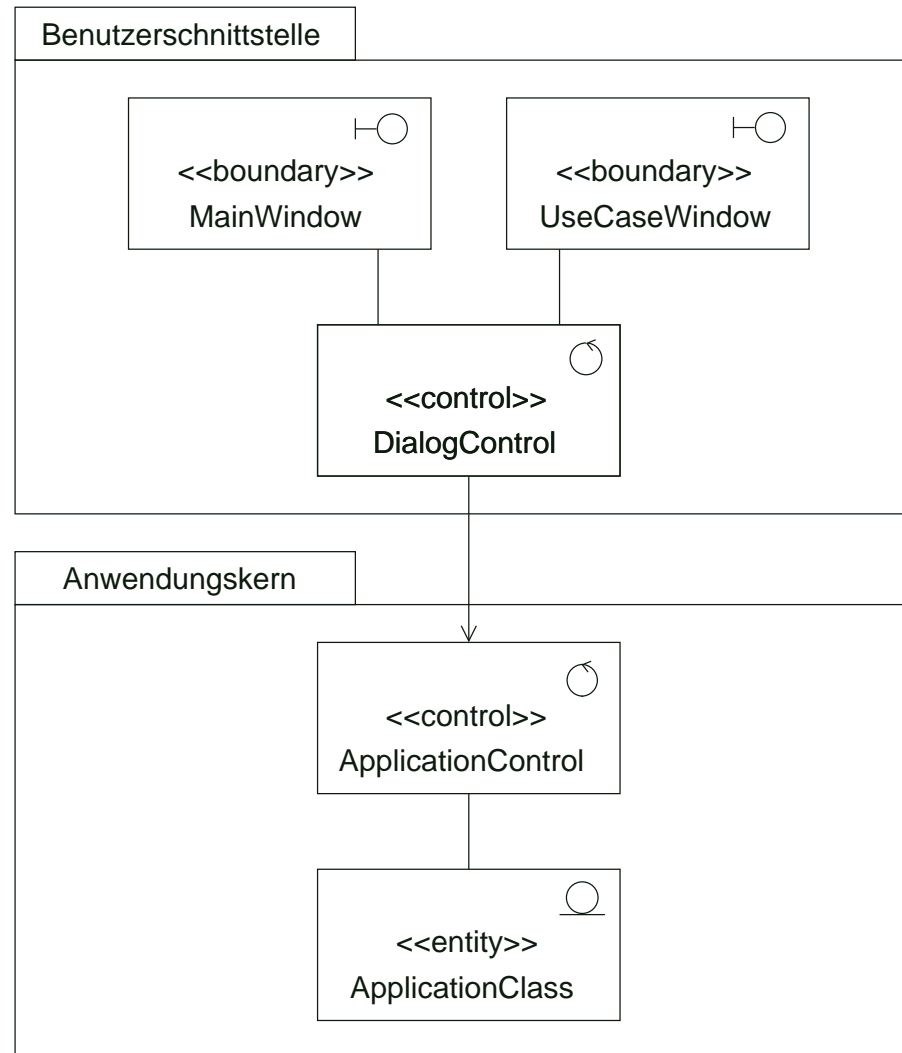
### *Bemerkung*

Kontrollobjekte haben häufig ein interessantes Verhalten, das durch Zustandsdiagramme beschrieben werden kann (z.B. ATM).

## Typisches Interaktionsmuster mit Kontrollobjekten



## Schnittstellen-, Kontroll- und Entity-Klassen



## 4.3.4 Kommunikation zwischen Benutzerschnittstelle und Anwendungskern

### Sichtbarkeitsregel

Der Anwendungskern kennt die Benutzerschnittstelle *NICHT* ("Model View Separation")!

### *Vorteil*

Änderung oder Austausch der Benutzeroberfläche hat keine Auswirkung auf den Code des Anwendungskerns.

*Beachte:* GUIs werden häufig verändert!

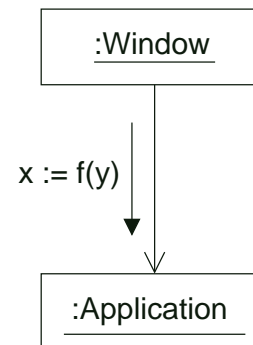
### *Problem*

Wie sollen Daten, die der Anwendungskern berechnet an die Oberfläche gelangen?

## Mögliche Lösungen

- Zu "zeigende" Daten können (manchmal) als Rückgabewert von Operationen übergeben werden.

*Beispiel:*



*Beachte:*

Dieser Ansatz funktioniert nicht, wenn das Anwendungsobjekt die Ausgabe von sich aus bewirken will (z.B. Wetterstation stellt eine Sturmwarnung fest).

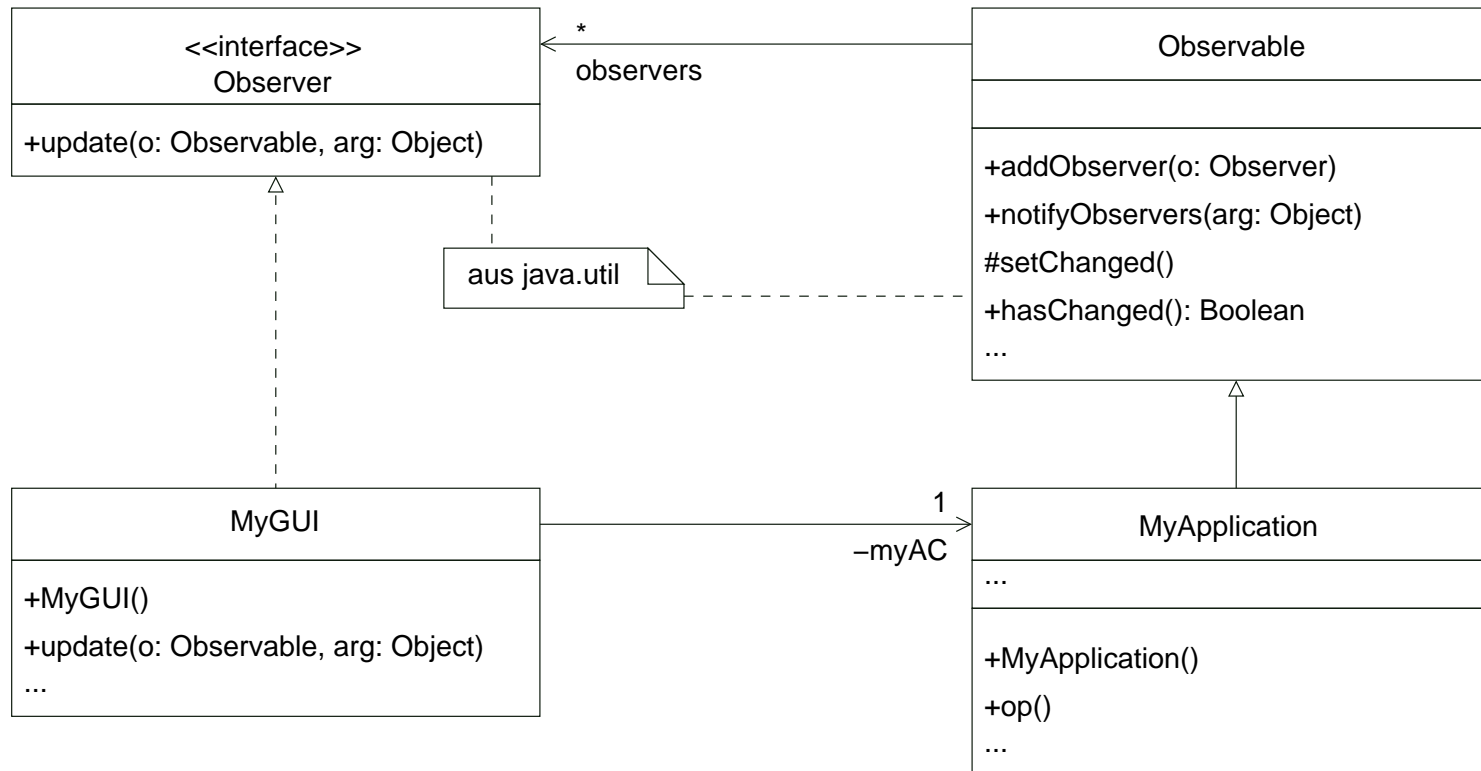
- Indirekte Kommunikation: Event-Manager oder Observer

## Indirekte Kommunikation durch Verwendung von Beobachtern

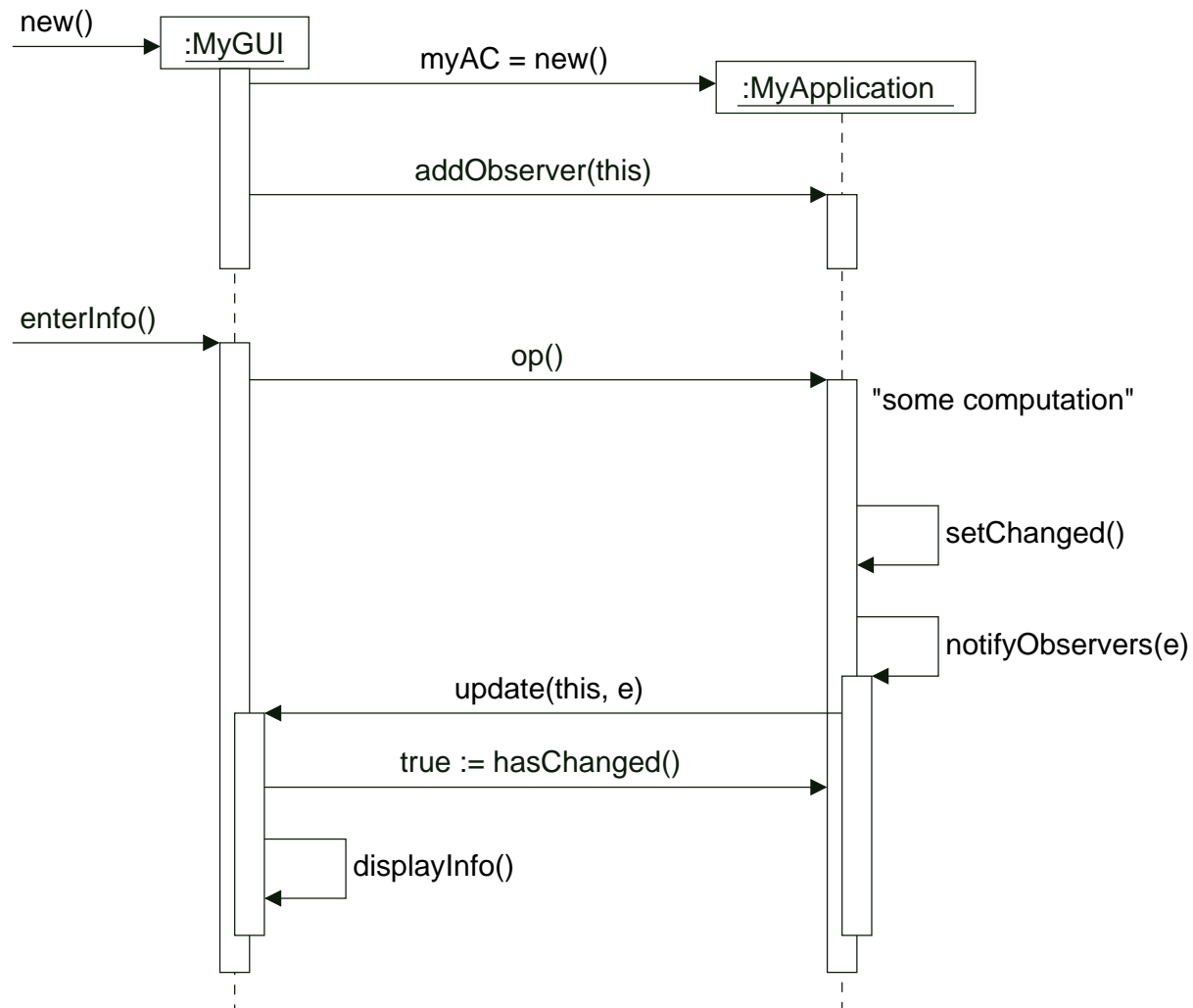
### *Idee*

- GUI-Objekte melden sich als Beobachter (Observer) beim Anwendungskern an (*addObserver*).
- Falls der Anwendungskern ein Ereignis publizieren will, benachrichtigt er alle seine Beobachter (*notifyObservers*), die entsprechend reagieren (*update*).
- Jeder konkrete Beobachter implementiert das Interface *Observer*.
- Der Anwendungskern kennt (zur Programmierzeit) nur das Observer-Interface. Konkrete Observer werden zur Laufzeit dynamisch eingebunden.

## Modell der Java-Realisierung von Beobachtern



## Typische Interaktion zwischen einem Beobachter und einem Beobachteten

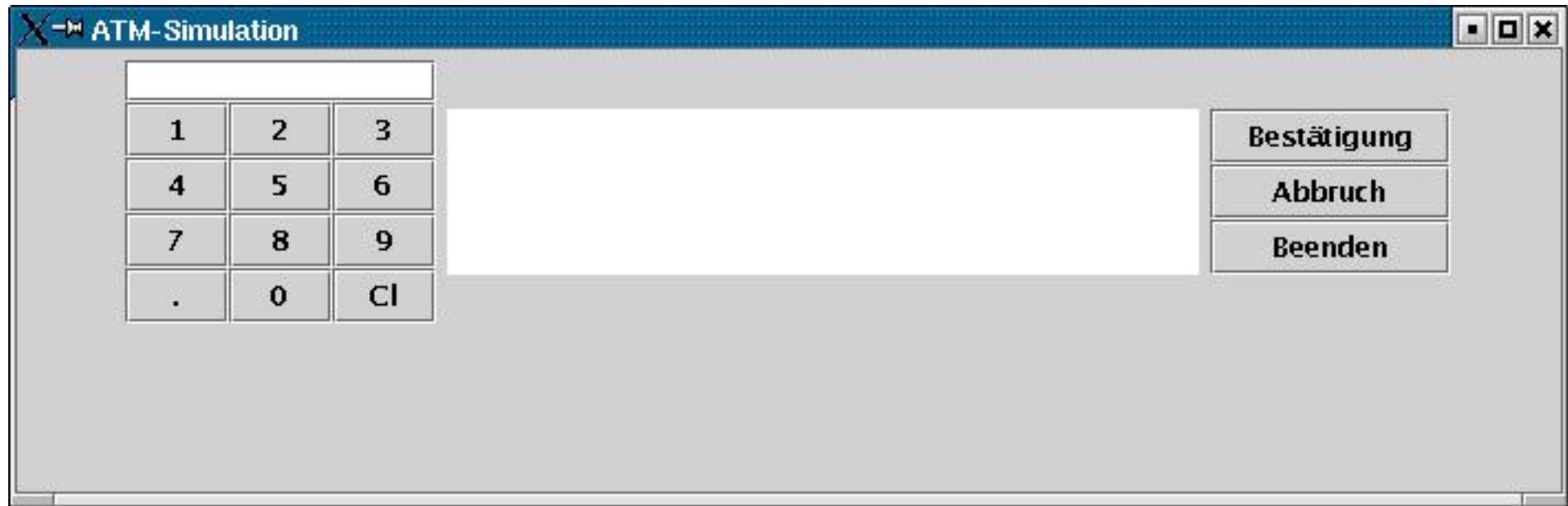


## Zusammenfassung von Abschnitt 4.3

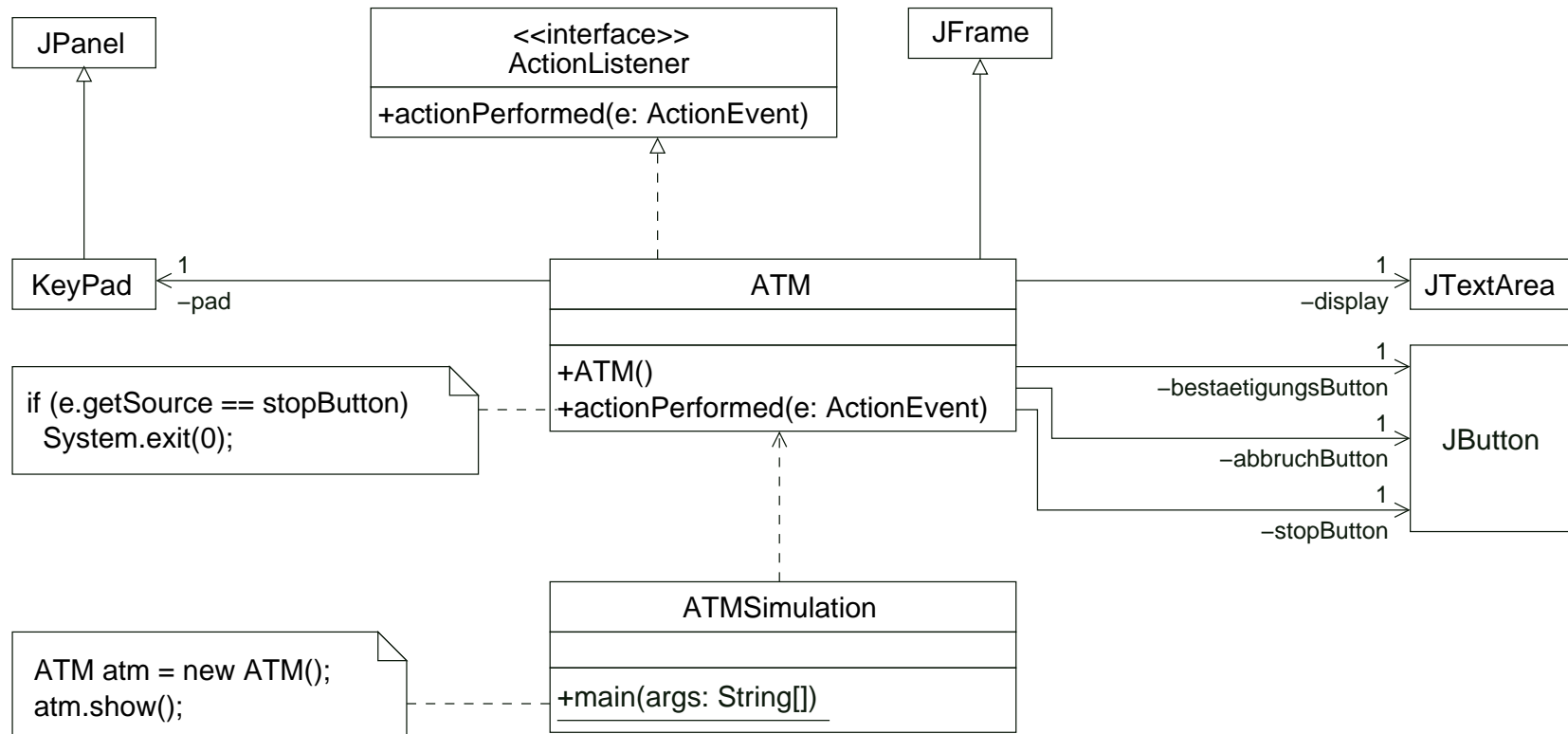
- Grundsätzliche Aufgabe des Systementwurfs ist die Festlegung der Systemarchitektur (Softwarearchitektur).
- Die Systemarchitektur beschreibt die Gesamtstruktur des Softwaresystems durch Angabe von Systemkomponenten (Subsystemen) und Beziehungen zwischen den Komponenten.
- Wichtige Grundregeln sind hohe Kohäsion und geringe Kopplung.
- Häufig werden Schichtenarchitekturen verwendet.
- Die 3-Schichten-Architektur für betriebliche Informationssysteme besteht aus den Schichten " Benutzerschnittstelle" , " Anwendungskern" und " Datenbankschnittstelle" .
- Die Sichtbarkeitsregel fordert, dass der Anwendungskern die Benutzerschnittstelle nicht kennt. (Wichtig für die leichte Austauschbarkeit der GUI!)

## 4.4 Entwurf von grafischen Benutzerschnittstellen

- GUI-Systeme ("graphical user interface") sind ereignisgesteuert: Der Benutzer löst Ereignisse aus (mit Maus, Tastatur, ...), die vom GUI-System empfangen und interpretiert werden.
- Zur Programmierung von Benutzerschnittstellen verwendet man i.a. *GUI-Toolkits*.
- Ein GUI-Toolkit stellt vorgefertigte Interaktionselemente ("widgets") zur Verfügung (z.B. Window, Button, Checkbox, ...).
- Individuelle GUI-Elemente können durch Spezialisierung gegebener Klassen definiert werden (Wiederverwendung).
- Abstrakte Toolkits erlauben die plattformunabhängige Konstruktion von GUIs. Wichtige Ausprägung: AWT ("abstract window toolkit") und Swing für Java-Programme.

*Beispiel:* Benutzerschnittstelle der ATM-Simulation

## Modell der GUI für die ATM-Simulation



```
// Vorläufige Version der ATM-Simulation zur Erstellung des GUI-Prototypen
```

```
import java.awt.*;
```

```
import java.awt.event.*;
```

```
import javax.swing.*;
```

```
class ATMSimulation {
```

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

```
        ATM atm = new ATM();
```

```
        atm.show();
```

```
    }
```

```
}
```

```
class ATM extends JFrame implements ActionListener {
```

```
    //Referenzattribute für GUI
```

```
    private Keypad pad;
```

```
    private JTextArea display;
```

```
    private JButton bestaetigungsButton;
```

```
    private JButton abbruchButton;
```

```
    private JButton stopButton;
```

```
public ATM() {  
  
    //Konstruktion der GUI  
    setSize(700, 200);  
    setTitle("ATM-Simulation");  
  
    pad = new KeyPad();  
    display = new JTextArea(5, 31);  
    bestaetigungsButton = new JButton("Bestätigung");  
    abbruchButton = new JButton("Abbruch");  
    stopButton = new JButton("Beenden");  
  
    JPanel buttonPanel = new JPanel();  
    buttonPanel.setLayout(new GridLayout(3, 1));  
    buttonPanel.add(bestaetigungsButton);  
    buttonPanel.add(abbruchButton);  
    buttonPanel.add(stopButton);  
  
    Container cont = getContentPane();  
    cont.setLayout(new FlowLayout());  
    cont.add(pad);  
}
```

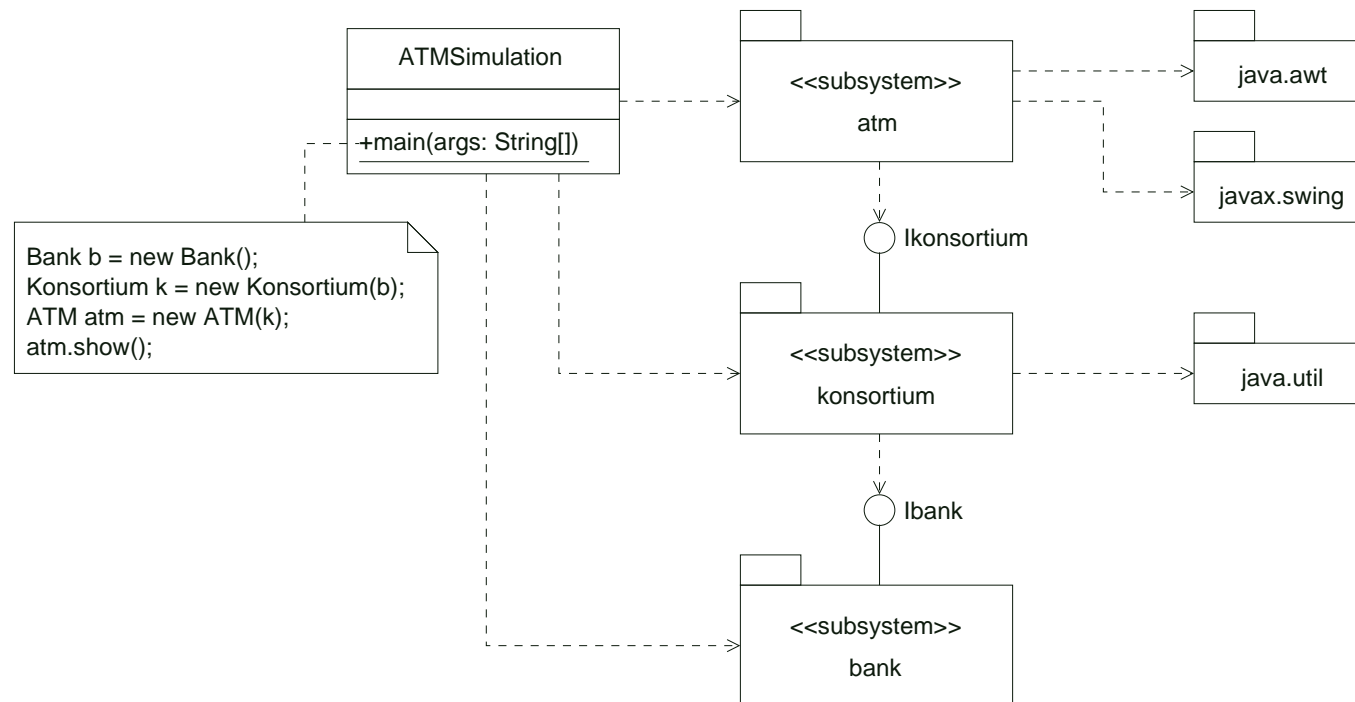
```
cont.add(display);
cont.add(buttonPanel);

//ATM als ActionListener zu allen Buttons hinzufügen
bestaetigungsButton.addActionListener(this);
abbruchButton.addActionListener(this);
stopButton.addActionListener(this);
}

public void actionPerformed(ActionEvent e) {
    if (e.getSource() == stopButton)
        System.exit(0);
}
}
```

## 4.5 Realisierung der ATM-Simulation

### 1. Systemarchitektur



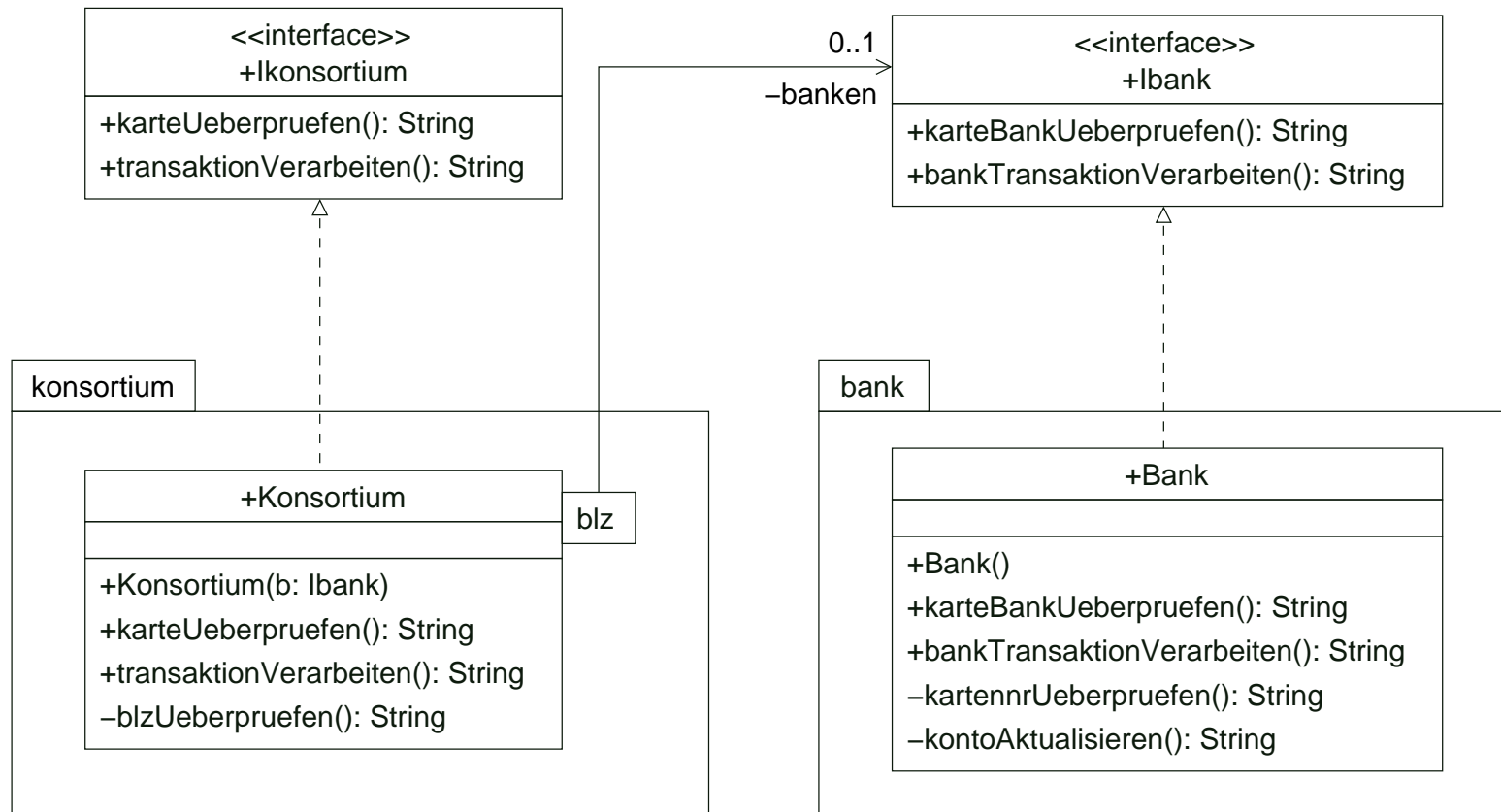
*Bemerkung:*

”atm” bildet die Benutzerschnittstelle, ”konsortium” und ”bank” bilden den Anwendungskern.

## *Vereinfachungen (im folgenden):*

- Beim Lesen der Kreditkarte wird keine Kartenummer und keine BLZ eingegeben, sondern nur die auf der Karte gespeicherte Geheimzahl gelesen.
- Konsortium und Bank werden durch vereinfachte Implementierungen realisiert. Die formalen Parameter ihrer Operationen werden weggelassen.
- Das ATM speichert keine Transaktionen (im Gegensatz zum Entwurf).

## 2. Die Subsysteme "konsortium" und "bank"



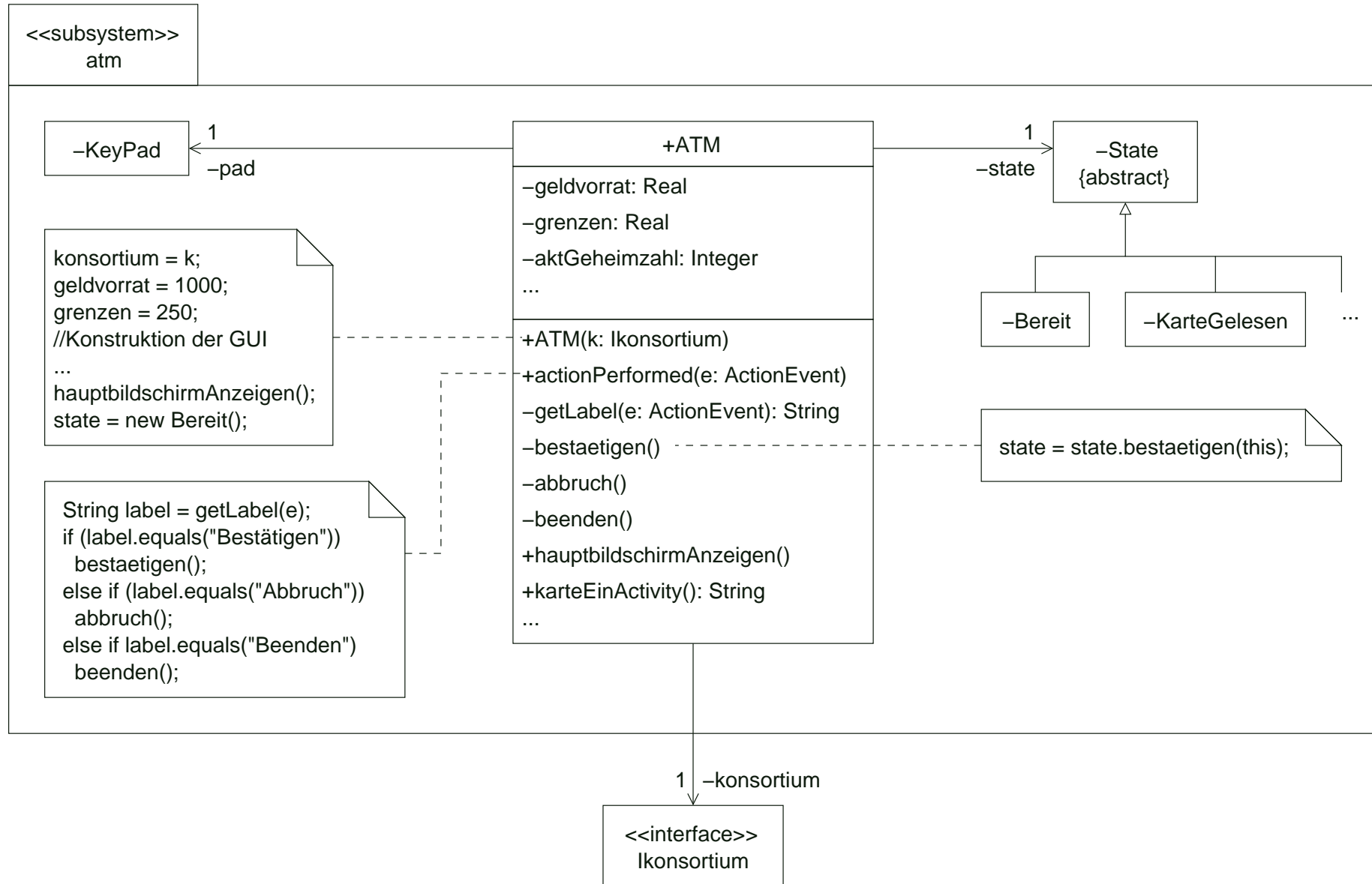
### 3. *Das Subsystem "atm"*

Basiert auf

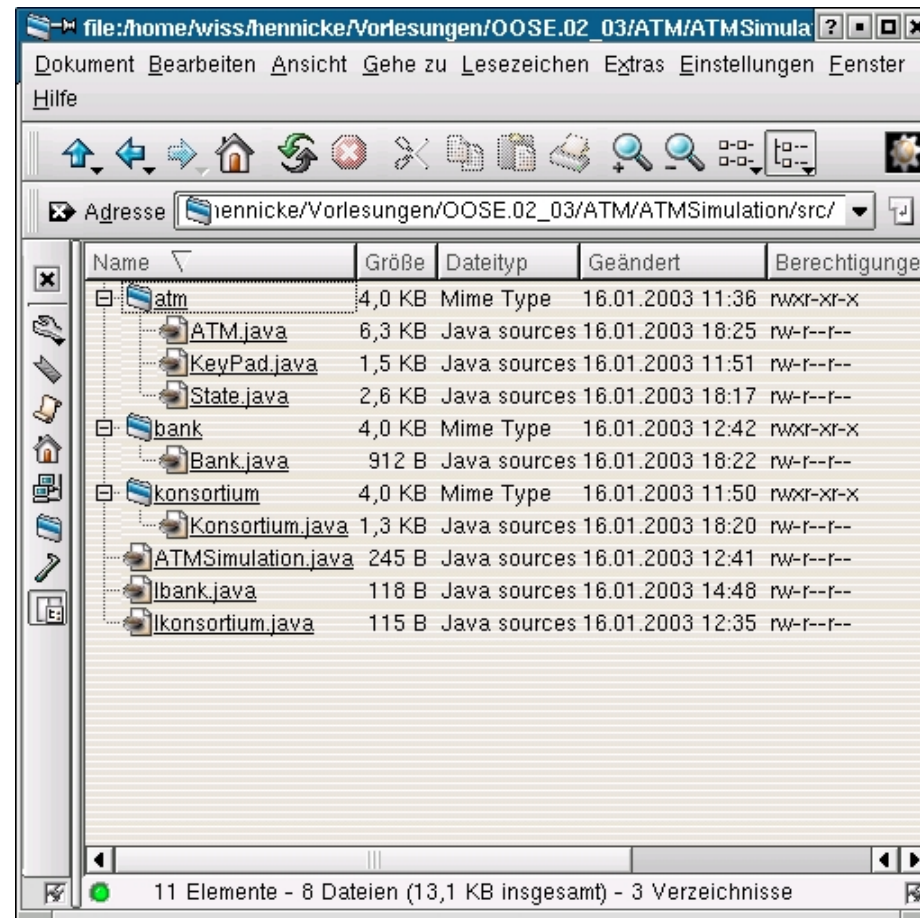
- a) Realisierung des Zustandsdiagramms der ereignisbasierten ATM-Simulation (vgl. Abschnitt 4.2.3).
- b) Benutzeroberfläche der ATM-Simulation (vgl. Abschnitt 4.4)
- c) Pseudo-Code der nicht zustandsabhängigen Operationen des ATM (vgl. Abschnitt 4.1.6)

*Beachte:*

Bei der Integration von a) und b) muss der Eventhandling-Mechanismus von AWT berücksichtigt werden, indem eine geeignete Implementierung von "actionPerformed" gegeben wird.



## 4. Java-Implementierung der ATM-Simulation



### **Beachte:**

Die Verzeichnisstruktur ist konform zur Systemarchitektur.

## Klasse ATM

```
package atm;

import Ikonsortium;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/* Implementierung der Klasse ATM mit integrierter GUI */

public class ATM extends JFrame implements ActionListener {

    //ATM-Attribute
    private double geldvorrat;
    private double grenzen;
    private int aktGeheimzahl;
    private int aktKartennr; //wird in der Simulation nicht verwendet
    private int aktBLZ;      //wird in der Simulation nicht verwendet
    private int aktKontonr; //wird in der Simulation nicht verwendet
```

```
//Referenzattribut fuer Konsortium
private Ikonsortium konsortium;

//Referenzattribute für GUI
private KeyPad pad;
private JTextArea display;
private JButton bestaetigungsButton;
private JButton abbruchButton;
private JButton stopButton;

//Referenzattribut für Zustandsobjekt
private State state;

public ATM(Ikonsortium k) {

    //Konsortium initialisieren
    konsortium = k;

    //ATM-Attribute initialisieren
    geldvorrat = 1000;
    grenzen = 250;
```

```
//Konstruktion der GUI
setSize(700, 200);
setTitle("ATM-Simulation");

pad = new KeyPad();
display = new JTextArea(5, 31);
bestaetigungsButton = new JButton("Bestätigung");
abbruchButton = new JButton("Abbruch");
stopButton = new JButton("Beenden");

JPanel buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(3, 1));
buttonPanel.add(bestaetigungsButton);
buttonPanel.add(abbruchButton);
buttonPanel.add(stopButton);

Container cont = getContentPane();
cont.setLayout(new FlowLayout());
cont.add(pad);
cont.add(display);
cont.add(buttonPanel);
```

```
//ATM als ActionListener zu allen Buttons hinzufügen
bestaetigungsButton.addActionListener(this);
abbruchButton.addActionListener(this);
stopButton.addActionListener(this);

//Hauptbildschirm anzeigen
hauptbildschirmAnzeigen();

//Zustandsobjekt initialisieren
state = new Bereit();
}

public void actionPerformed(ActionEvent e) {
    String label = getLabel(e);
    if (label.equals("Bestätigen"))
        bestaetigen();
    else if (label.equals("Abbruch"))
        abbruch();
    else if (label.equals("Beenden"))
        beenden();
}
```

```
private String getLabel(ActionEvent e) {  
    JButton source = (JButton)e.getSource();  
    return source.getText();  
}
```

```
private void bestaetigen() {  
    //Delegation an das Zustandsobjekt  
    state = state.bestaetigen(this);  
}
```

```
private void abbruch() {  
    //Delegation an das Zustandsobjekt  
    state = state.abbruch(this);  
}
```

```
private void beenden() {  
    //Delegation an das Zustandsobjekt  
    state = state.beenden(this);  
}
```

```
//Öffentliche Operationen für die Aktivitäts- und Aktionszustände
```

```
public void hauptbildschirmAnzeigen() {
    display.setText("<<Hauptbildschirm: Aufforderung zur Karteneingabe>>");
    display.append("\nAuf Karte gespeicherte Geheimzahl? (Eingabe bestätigen)");
}

public String karteEinActivity() {
    String check = karteLesen();
    if (check.equals("lesbar")) {
        display.setText("Karte lesbar!");
        display.append("\nGeheimzahl? (Eingabe bestätigen)");
        return "Karte lesbar";
    }
    else if (check.equals("nicht lesbar")) {
        display.setText("Karte nicht lesbar!");
        abschluss();
        return "Karte nicht lesbar";
    }
    else return "Error";
}
```

```
public String geheimzahlEinActivity() {
    int typedGeheimzahl = (int)pad.getValue();
    pad.clear();
    String check = geheimzahlUeberpruefen(typedGeheimzahl);
    if (check.equals("Geheimzahl ok")) {
        check = konsortium.karteUeberpruefen();
        if (check.equals("Karte ok")) {
            display.setText("Karte ok!");
            display.append("\nTransaktionsform? (Bestätigung = Abhebung)");
            return "Karte ok";
        }
        else if (check.equals("falsche BLZ")) {
            display.setText("Karte nicht ok: falsche BLZ!");
            abschluss();
            return "Karte nicht ok";
        }
        else if (check.equals("Karte gesperrt")) {
            display.setText("Karte nicht ok: Karte gesperrt!");
            abschluss();
            return "Karte nicht ok";
        }
    }
}
```

```
        else return "Error";
    }
    else if (check.equals("Geheimzahl falsch")) {
        display.setText("Geheimzahl falsch!");
        display.append("\nGeheimzahl? (Eingabe bestätigen)");
        return "Geheimzahl falsch";
    }
    else return "Error";
}

public void betragErfragen() {
    display.setText("Betrag? (Eingabe bestätigen)");
}

public String betragEinActivity() {
    double betrag = pad.getValue();
    pad.clear();
    String check = grenzenUeberpruefen(betrag);
    if (check.equals("Grenzen ok")) {
        check = konsortium.transaktionVerarbeiten();
    }
}
```

```
if (check.equals("Transaktion erfolgreich")) {
    geldvorrat = geldvorrat - betrag;
    display.setText("Transaktion erfolgreich!");
    display.append("\nGeld wird ausgegeben");
    display.append("\nGeld entnehmen? (Bestätigung)");
    return "Transaktion erfolgreich";
}
else if (check.equals("Transaktion gescheitert")) {
    display.setText("Transaktion gescheitert!");
    display.append("\nTransaktionsform? (Bestätigung = Abhebung)");
    return "Transaktion gescheitert";
}
else return "Error";
}
else if (check.equals("Grenzen überschritten")) {
    display.setText("Grenzen überschritten!");
    display.append("\nBetrag? (Eingabe bestätigen)");
    return "Grenzen überschritten";
}
else return "Error";
}
```

```
public void nachFortsetzungFragen() {
    display.setText("Fortsetzung? (Bestätigung = Nein)");
}

public void abschluss() {
    pad.clear();
    display.append("\nBeleg wird gedruckt");
    display.append("\nKarte wird ausgegeben");
    display.append("\nKarte und Beleg entnehmen? (Bestätigung)");
}

//private Operationen
private String karteLesen() {
    aktGeheimzahl = (int)pad.getValue();
    pad.clear();
    return "lesbar";
}
```

```
private String geheimzahlUeberpruefen(int tgz) {
    if (tgz == aktGeheimzahl) return "Geheimzahl ok";
    else return "Geheimzahl falsch";
}

private String grenzenUeberpruefen(double b) {
    if (b <= grenzen) return "Grenzen ok";
    else return "Grenzen überschritten";
}
}
```

## Klasse State

```
package atm;

/* Das folgende Programm realisiert das Zustandsdiagramm der ATM-Simulation
   durch Zustandsobjekte (vgl. auch das State-Pattern) */

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

abstract class State {
    public State bestaetigen(ATM atm) {
        return this;
    }
    public State abbruch(ATM atm) {
        return this;
    }
    public State beenden(ATM atm) {
        return this;
    }
}
```

```
class Bereit extends State {  
  
    public State bestaetigen(ATM atm) {  
        String check = atm.karteEinActivity();  
        if (check.equals("Karte lesbar"))  
            return new KarteGelesen();  
        else if (check.equals("Karte nicht lesbar"))  
            return new BereitZurKartenentnahme();  
        else return this;  
    }  
  
    public State beenden(ATM atm) {  
        System.exit(0);  
        return this;  
    }  
}
```

```
class KarteGelesen extends State {  
  
    public State bestaetigen(ATM atm) {  
        String check = atm.geheimzahlEinActivity();  
        if (check.equals("Karte ok"))  
            return new GeheimzahlUndKarteGeprueft();  
        else if (check.equals("Karte nicht ok"))  
            return new BereitZurKartenentnahme();  
        else if (check.equals("Geheimzahl falsch"))  
            return new KarteGelesen(); //oder einfacher: return this;  
        else return this;  
    }  
  
    public State abbruch(ATM atm) {  
        atm.abschluss();  
        return new BereitZurKartenentnahme();  
    }  
}
```

```
class GeheimzahlUndKarteGeprueft extends State {  
  
    public State bestaetigen(ATM atm) {  
        atm.betragErfragen();  
        return new TransaktionsformBestimmt();  
    }  
  
    public State abbruch(ATM atm) {  
        atm.abschluss();  
        return new BereitZurKartenentnahme();  
    }  
}
```

```
class TransaktionsformBestimmt extends State {

    public State bestaetigen(ATM atm) {
        String check = atm.betragEinActivity();
        if (check.equals("Transaktion erfolgreich"))
            return new TransaktionDurchgefuehrt();
        else if (check.equals("Transaktion gescheitert"))
            return new GeheimzahlUndKarteGeprueft();
        else if (check.equals("Grenzen überschritten"))
            return new TransaktionsformBestimmt(); //oder einfacher: return this;
        else return this;
    }

    public State abbruch(ATM atm) {
        atm.abschluss();
        return new BereitZurKartenentnahme();
    }
}
```

```
class TransaktionDurchgefuehrt extends State {
    public State bestaetigen(ATM atm) {
        atm.nachFortsetzungFragen();
        return new GeldEntnommen();
    }
}
```

```
class GeldEntnommen extends State {
    public State bestaetigen(ATM atm) {
        atm.abschluss();
        return new BereitZurKartenentnahme();
    }
}
```

```
class BereitZurKartenentnahme extends State {
    public State bestaetigen(ATM atm) {
        atm.hauptbildschirmAnzeigen();
        return new Bereit();
    }
}
```

## Interface Ikonsortium

```
public interface Ikonsortium {  
    public String karteUeberpruefen();  
    public String transaktionVerarbeiten();  
}
```

## Klasse Konsortium

```
package konsortium;  
  
import Ikonsortium;  
import Ibank;  
import java.util.Hashtable;  
  
public class Konsortium implements Ikonsortium {  
  
    private Hashtable banken = new Hashtable();  
  
    //Bei der Konstruktion des Konsortiums wird genau eine Bank eingefügt  
    public Konsortium(Ibank b) {  
        banken.put(new Integer(101), b);  
    }  
}
```

```
public String karteUeberpruefen() {
    String check = blzUeberpruefen();
    if (check.equals("BLZ richtig")) {
        Ibank b = (Ibank)banken.get(new Integer(101));
        check = b.karteBankUeberpruefen();
        if (check.equals("Karte ok"))
            return "Karte ok";
        else if (check.equals("Karte bei Bank gesperrt"))
            return "Karte gesperrt";
        else return "Error";
    }
    else if (check.equals("BLZ falsch"))
        return "falsche Bankleitzahl";
    else return "Error";
}
```

```
public String transaktionVerarbeiten() {
    Ibank b = (Ibank)banken.get(new Integer(101));
    String check = b.bankTransaktionVerarbeiten();
    if (check.equals("Banktransaktion erfolgreich"))
        return "Transaktion erfolgreich";
    else if (check.equals("Banktransaktion gescheitert"))
        return "Transaktion gescheitert";
    else return "Error";
}

//Dummy-Implementierung
private String blzUeberpruefen() {
    return "BLZ richtig";
//    return "BLZ falsch";
}
}
```

## Interface Ibank

```
public interface Ibank {  
    public String karteBankUeberpruefen();  
    public String bankTransaktionVerarbeiten();  
}
```

## Klasse Bank

```
package bank;  
  
import Ibank;  
  
public class Bank implements Ibank {  
  
    public String karteBankUeberpruefen() {  
        String check = kartennrUeberpruefen();  
        if (check.equals("gültig")) return "Karte ok";  
        else if (check.equals("gesperrt")) return "Karte bei Bank gesperrt";  
        else return "Error";  
    }  
}
```

```
public String bankTransaktionVerarbeiten() {
    String check = kontoAktualisieren();
    if (check.equals("erfolgreich")) return "Banktransaktion erfolgreich";
    else if (check.equals("gescheitert")) return "Banktransaktion gescheitert";
    else return "Error";
}

/* Die folgenden Operationen haben lediglich Dummy-Implementierungen
zum Testen der ATM Simulation */
private String kartennrUeberpruefen() {
    return "gültig";
//    return "gesperrt";
}
private String kontoAktualisieren() {
    return "erfolgreich";
//    return "gescheitert";
}
}
```

## Zusammenfassung von Abschnitt 4.5

- Die Systemarchitektur der ATM-Simulation basiert auf 3 Subsystemen (atm, konsortium und bank), die über Schnittstellen miteinander verbunden sind.
- Für die Klassen Konsortium und Bank werden z.T. vereinfachte Implementierungen verwendet.
- Das Subsystem "atm" beinhaltet die GUI und die Realisierung des Zustandsdiagramms der ATM-Simulation.

## 4.6 Anbindung an eine Datenbank

### *Ziel*

Speicherung von *persistenten* Objekten, d.h. von Objekten des Anwendungskerns, die dauerhaft benötigt werden (z.B. Kunden, Konten, Flüge, Bücher, ...).

### *Möglichkeiten*

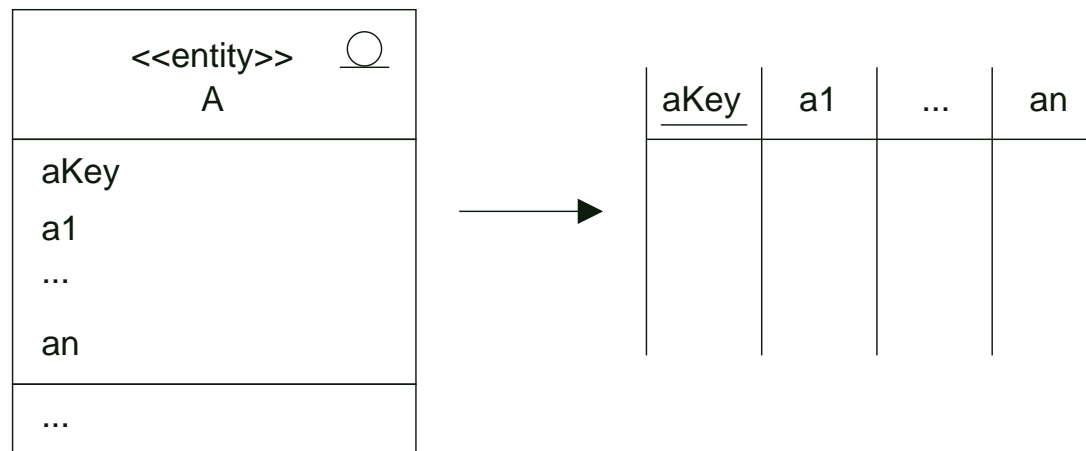
- *Objektorientierte Datenbanken:*  
Unterstützen Assoziationen und Vererbung. Standard ODL (object definition language) für Schemadeklarationen.
- *Relationale Datenbanken:*  
Unterstützen Assoziationen und Vererbung *nicht*. Deshalb wird eine explizite Schnittstelle zwischen dem Anwendungskern und dem relationalen Datenbanksystem benötigt.
- *Objektrelationale Datenbanken:*  
Verkapseln eine relationale Datenbank mit einer objektorientierten Hülle.

## 4.6.1 Abbildung eines Objektmodells auf Tabellen

### *Voraussetzung*

Für jede Entity-Klasse A ist ein Primärschlüsselattribut aKey eingeführt.

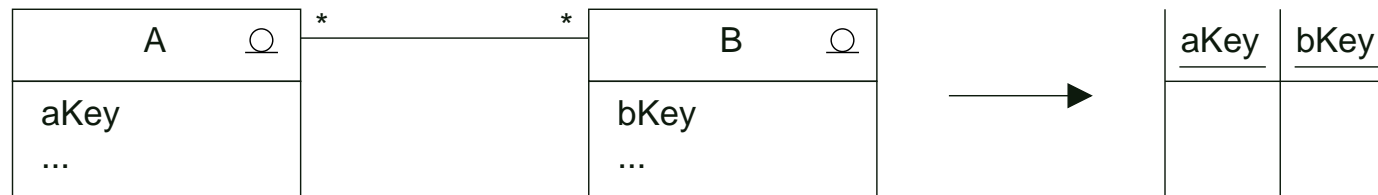
### Abbildung von Klassen



## Abbildung von Assoziationen

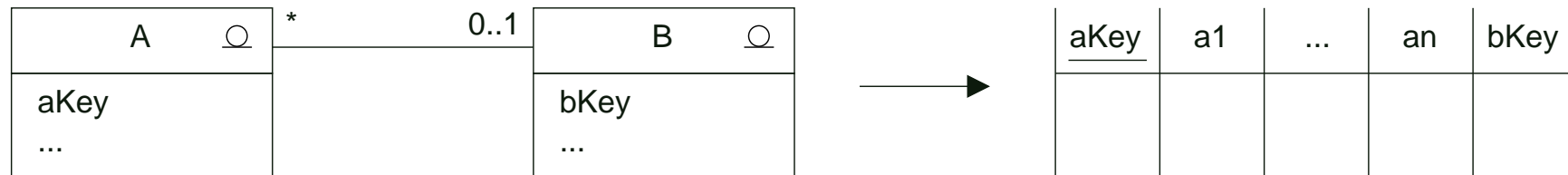
### Multiplizität \* - \*

Verwendung einer eigenen Tabelle mit den Primärschlüsseln von A und B.



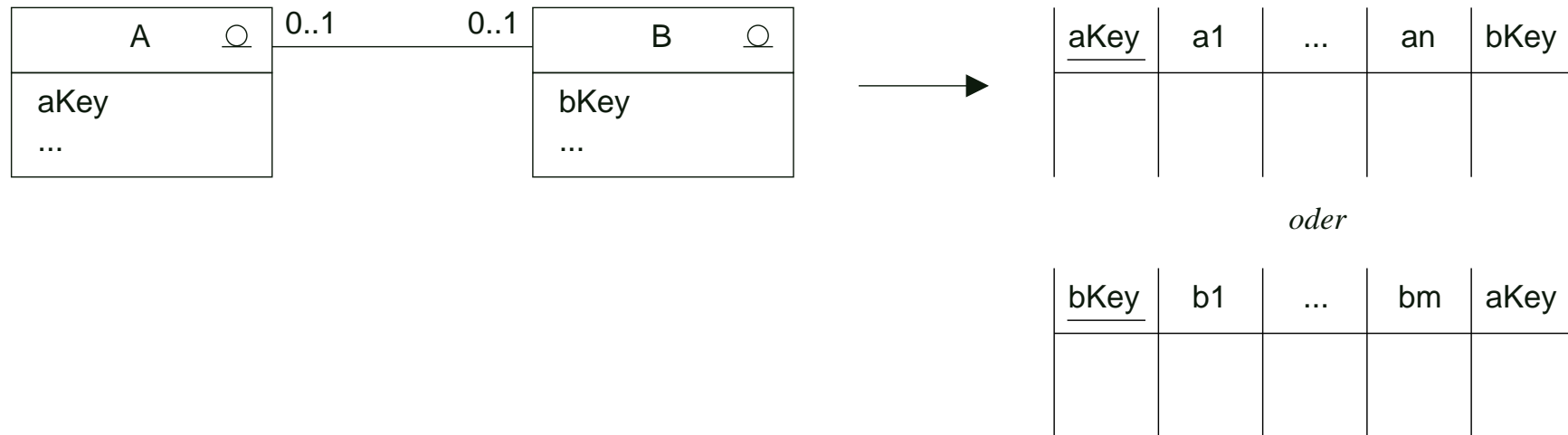
### Multiplizität \* - 0..1

Primärschlüssel von B als Fremdschlüssel in die Tabelle von A aufnehmen.




## Multiplizität 0..1 - 0..1

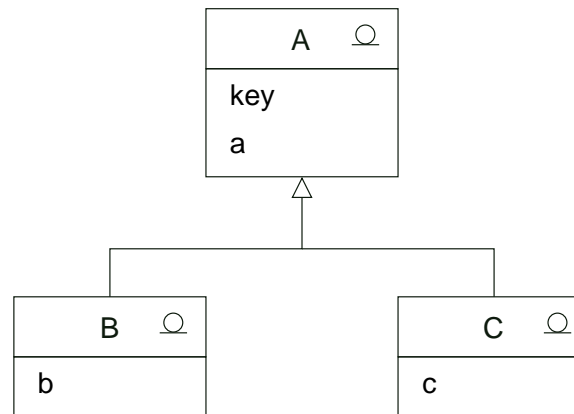
Primärschlüssel von B in die Tabelle von A oder  
Primärschlüssel von A in die Tabelle von B aufnehmen.



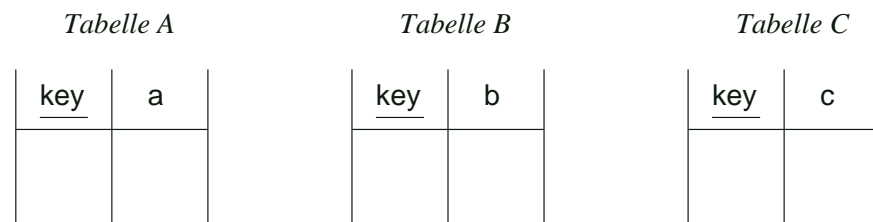
## Bemerkung

Bei  ist der Primärschlüssel von B als Fremdschlüssel zur Tabelle von A hinzuzunehmen.

## Abbildung von Vererbung



### *Variante I: Je eine Tabelle pro Klasse*



### *Nachteil:*

Bei Anfragen und Manipulationen, die Objekte einer Unterklasse betreffen, müssen ggf. Einträge in mehreren Tabellen berücksichtigt werden (z.B. bei Auswahl aller Attributwerte aller B-Objekte).

## Variante II: Tabellen von Unterklassen enthalten geerbte Attribute

<i>Tabelle A</i>		<i>Tabelle B</i>			<i>Tabelle C</i>		
<u>key</u>	a	<u>key</u>	a	b	<u>key</u>	a	c

Falls A abstrakt ist, genügt eine Tabelle pro Unterklasse (Tabelle A entfällt).

### *Nachteil:*

Bei Änderungen an der Form der Oberklasse, müssen auch die Tabellen der Unterklassen verändert werden.

## Variante III: Eine Tabelle für alle Ober- und Unterklassen

<i>Tabelle ABC</i>				
<u>key</u>	a	b	c	Typ

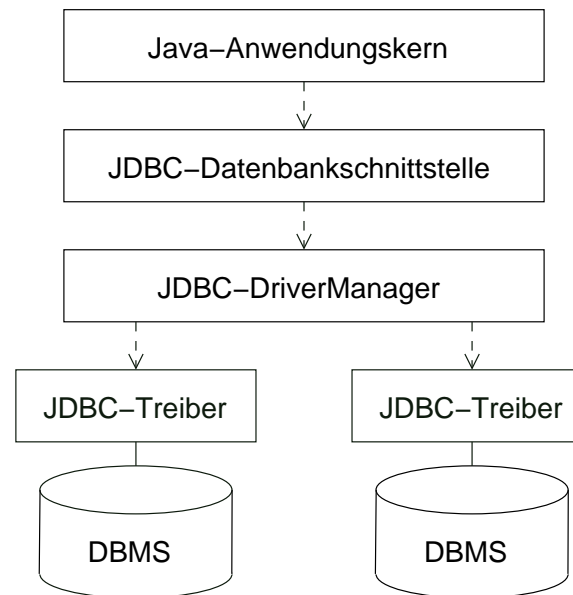
### *Nachteil:*

In Spalten, die für Objekte einer Unterklasse nicht relevant sind, müssen Nullwerte eingetragen werden.

## 4.6.2 Datenbankanbindung mit der JDBC

- JDBC (Java Database Connectivity) bietet eine SQL-Schnittstelle für Java-Programme.
- Die JDBC ist unabhängig von einem konkreten Datenbanksystem. Zum Zugriff auf ein konkretes Datenbanksystem muss ein entsprechender Treiber geladen werden.

### Schichten einer JDBC-Anwendung





- "Driver", "Connection", "Statement" und "ResultSet" sind Schnittstellen, die von den Klassen eines geladenen Treiberpakets implementiert werden.
- Der DriverManager registriert Treiber für bestimmte DB-Systeme. Der Aufruf der Operation "getConnection" stellt (mittels eines für die gegebene URL passenden Treibers) eine Verbindung zu einem DB-System her.
- Mittels eines Connection-Objekts kann ein Statement-Objekt erzeugt werden (Operation "createStatement").
- Mittels eines Statement-Objekts kann z.B. eine Anfrage an die DB gestellt werden (Operation "executeQuery").
- Die Ergebnistabelle der Anfrage wird in einem ResultSet-Objekt gespeichert.
- Die Tabelle eines ResultSet-Objekts kann mit den Operationen "next" und "getXXX" durchlaufen werden.

*Beispiel:*

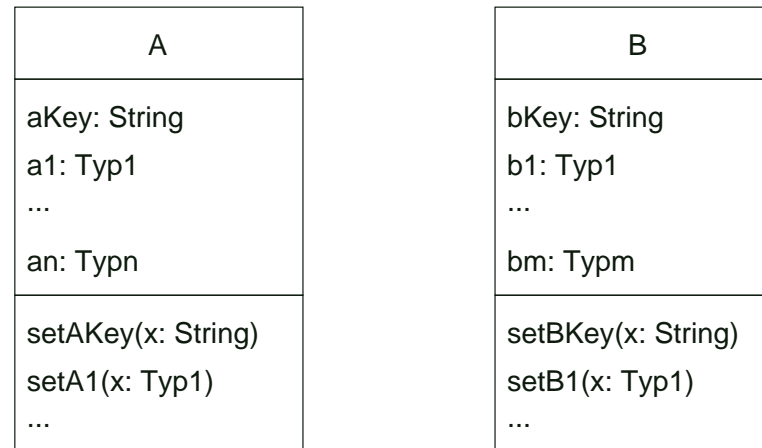
```
try {
    Class.forName("imaginary.sql.iMysqlDriver");// Treiber auch von außen setzbar
    String url = "jdbc:mysql://localhost/myDB";
    Connection con = DriverManager.getConnection(url);
    Statement stmt = con.createStatement();
    ResultSet rs    = stmt.executeQuery("SELECT * FROM test");

    while (rs.next()) {
        // rs.getXXX(y); Zugriff auf die Spalte mit der Nummer y und dem Typ XXX
    }
    con.close();
}
catch (Exception e) {
    e.printStackTrace();
}
```

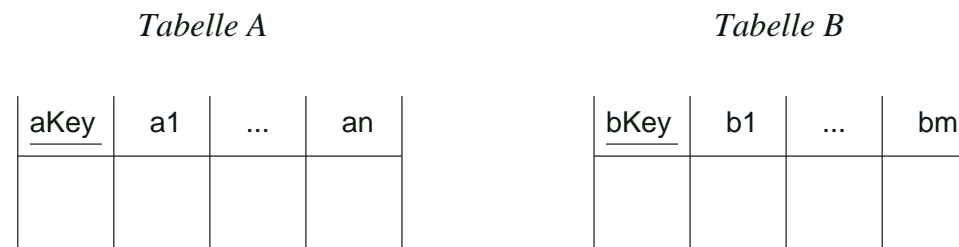
### 4.6.3 Materialisierung von Objekten

#### Gegeben

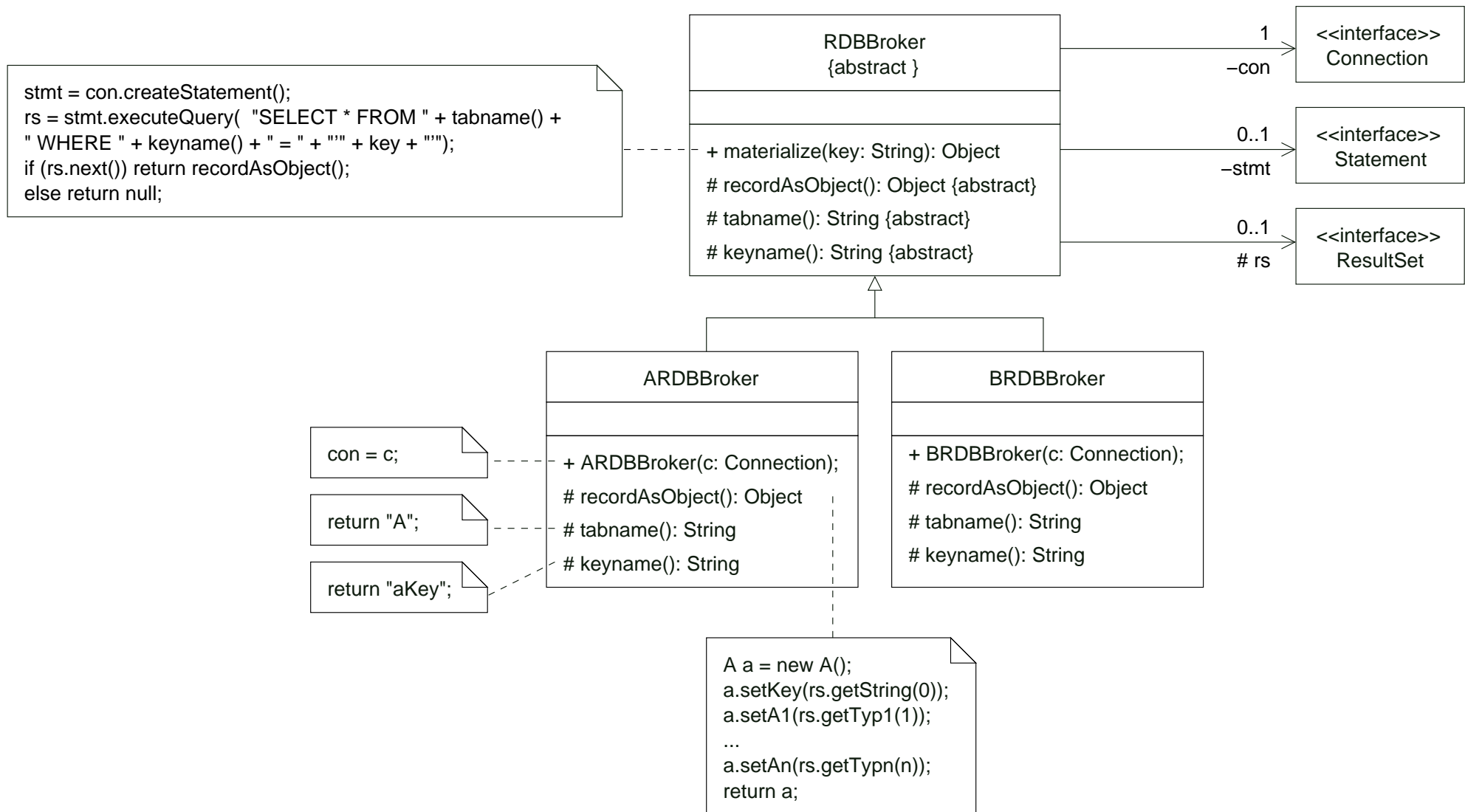
*Klassen im Anwendungskern:*



*Tabellen in der relationalen Datenbank:*



## Materialisierung mit der JDBC



*Beispiel:*

```
try {
    String url = "jdbc:RDB-Typ//Rechner:Port/Datenbank";
    Connection con = DriverManager.getConnection(url);
    ARDBBroker ardb = new ARDBBroker(con);
    BRDBBroker brdb = new BRDBBroker(con);
    A a = (A) ardb.materialize("xyz");
    B b = (B) brdb.materialize("uvw");
}
catch (Exception e) {
    e.printStackTrace();
}
```

## Zusammenfassung von Abschnitt 4.6

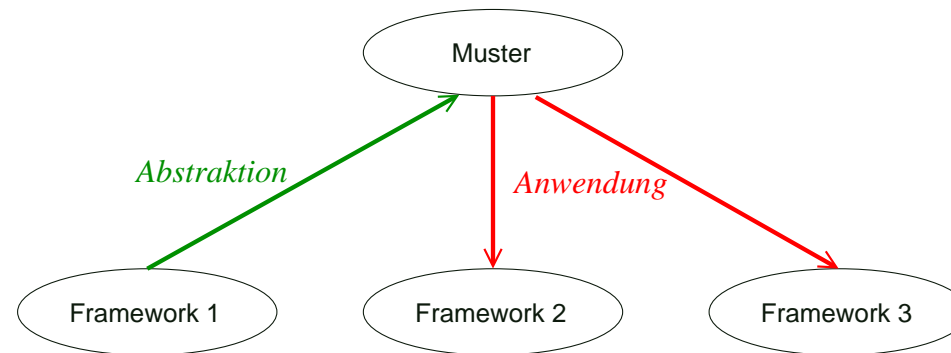
- Zur Speicherung persistenter Objekte muss der Anwendungskern an eine Datenbank angeschlossen werden.
- Dazu können objektorientierte, relationale oder objektrelationale Datenbanken verwendet werden.
- Bei der Verwendung einer relationalen Datenbank muss zunächst das Objektmodell auf Tabellen abgebildet werden. (Insbesondere müssen Vererbungshierarchien geeignet abgebildet werden!)
- Die JDBC bietet eine plattformunabhängige Schnittstelle zur Anbindung von Java-Programmen an relationale Datenbanken.

## 4.7 Entwurfsmuster

### 4.7.1 Grundlagen

#### *Grundidee*

Dasselbe (bewährte) Lösungsmuster kann für Probleme, die einander ähnlich sind, wiederverwendet werden.



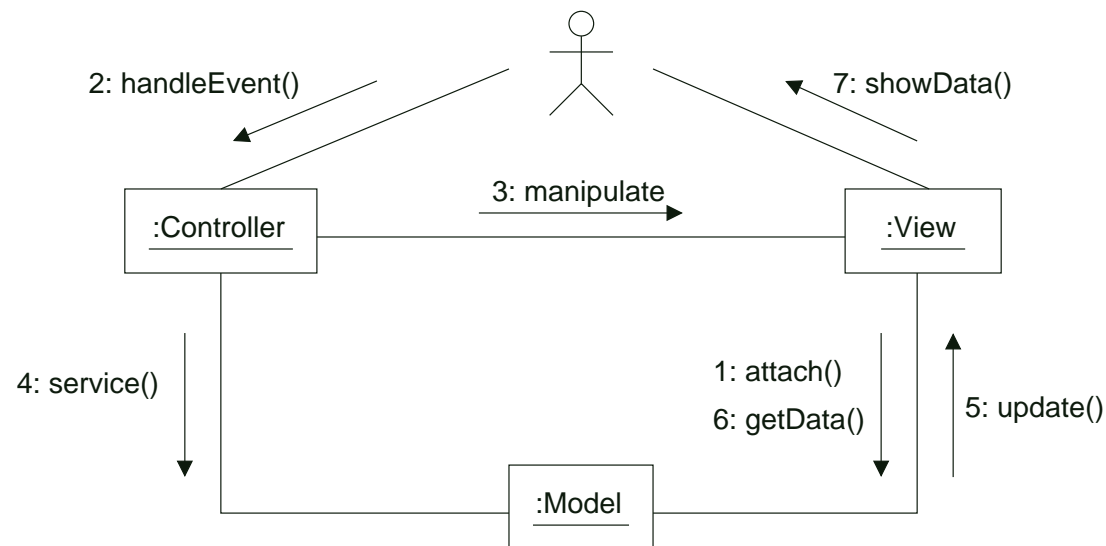
#### *Vorteile*

- Wiederverwendung von Lösungsprinzipien
- abstrakte Dokumentation von Entwürfen
- gemeinsames Vokabular zur (schnellen) Verständigung unter Entwicklern

## Geschichte

- 1977 Alexander: Architekturmuster für Gebäude und Städtebau
- 1980 Smalltalk's MVC-Prinzip (Model View Controller)
- Seit 1990 Objektorientierte Muster
- 1995 Design Pattern Katalog von Gamma, Helm, Johnson, Vlissides (GoF "Gang of Four")

### MVC-Architektur (vereinfacht)



## Wesentliche Elemente eines Entwurfsmusters

- Name des Musters
- Beschreibung der Problemklasse, bei der das Muster anwendbar ist
- Beschreibung eines Anwendungsbeispiels
- Beschreibung der Lösung (Struktur, Verantwortlichkeiten, ...)
- Nutzen/Kosten-Analyse

## 4.7.2 Design-Pattern Katalog (Gamma et al.)

### *Beschreibungsform für Design Pattern*

- **Pattern Name and Classification**

The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary.

- **Intent**

A short statement that answers the following question: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

- **Also Known As**

Other well-known names for the pattern, if any.

- **Motivation**

A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract descriptions of the pattern that follows.

- **Applicability**

What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

- **Structure**

A graphical representation of the classes in the pattern using a notation based on the Object Modelling Technique (OMT). We also use interaction diagrams to illustrate sequences of requests and collaborations between objects.

- **Participants**

The classes and/or objects participating in the design pattern and their responsibilities.

- **Collaborations**

How the participants collaborate to carry out their responsibilities.

- **Consequences**

How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspects of system structure does it let you vary independently?

- **Implementation**

What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

- **Sample Code**

Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

- **Known Uses**

Examples of the pattern found in real systems. We include at least two examples from different domains.

- **Related Patterns**

What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

## *Klassifikation von Design Pattern*

- **Creational Patterns** (befassen sich mit der Erzeugung von Objekten)
  - **Abstract Factory** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
  - **Builder** Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - **Factory Method** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
  - **Prototype** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
  - **Singleton** Ensure a class only has one instance, and provide a global point of access to it.
- **Structural Patterns** (befassen sich mit der Komposition von Klassen oder Objekten)
  - **Adapter** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - **Bridge** Decouple an abstraction from its implementation so that the two can vary independently.
  - **Composite** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

- **Decorator** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **Facade** Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **Flyweight** Use sharing to support large numbers of fine-grained objects efficiently.
- **Proxy** Provide a surrogate or placeholder for another object to control access to it.
- **Behavioral Patterns** (befassen sich mit der Interaktion von Objekten und der Verteilung von Verantwortlichkeiten)
  - **Chain of Responsibility** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
  - **Command** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
  - **Interpreter** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
  - **Iterator** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

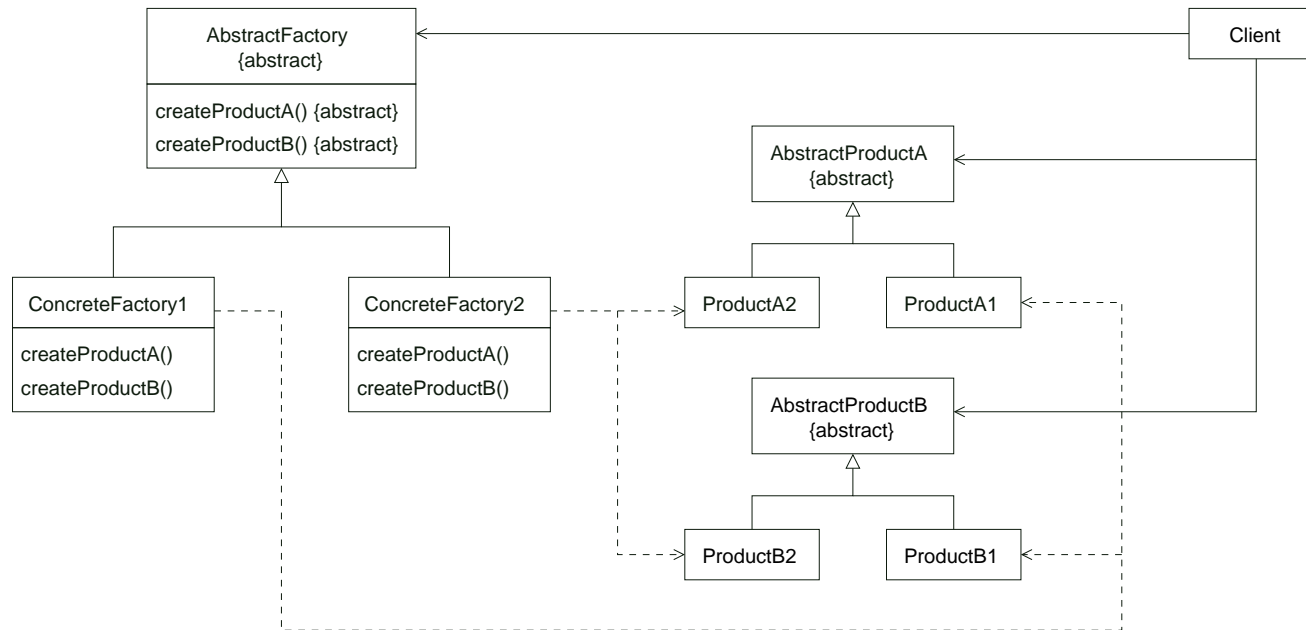
- **Mediator** Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- **Memento** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- **Observer** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **State** Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.
- **Strategy** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- **Template Method** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- **Visitor** Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the element on which it operates.

## Beispiel 1: Abstract Factory (Creational Pattern)

### Zweck

Stellt eine Schnittstelle zum Erzeugen mehrerer zusammengehöriger oder verwandter Objekte zur Verfügung, ohne dass dazu die konkreten Objektklassen benötigt werden.

### Struktur



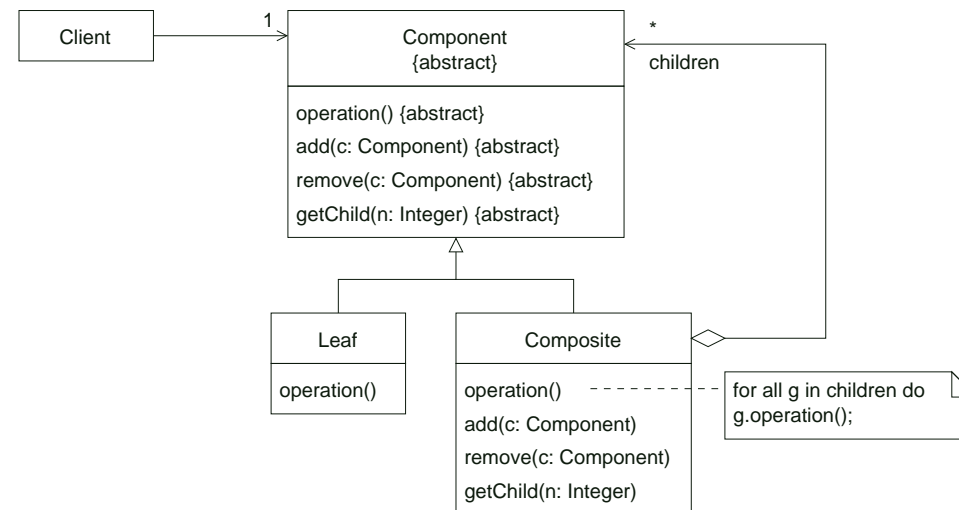
*Anwendungsbeispiel: Toolkit in AWT*

## Beispiel 2: Composite (Structural Pattern)

### Zweck

Anordnung von einzelnen Objekten in Baumstrukturen um "Teil-Ganzes"-Hierarchien darzustellen. Das Composite-Pattern ermöglicht es dem Klienten, sowohl einzelne als auch zusammengesetzte Objekte einheitlich zu behandeln.

### Struktur



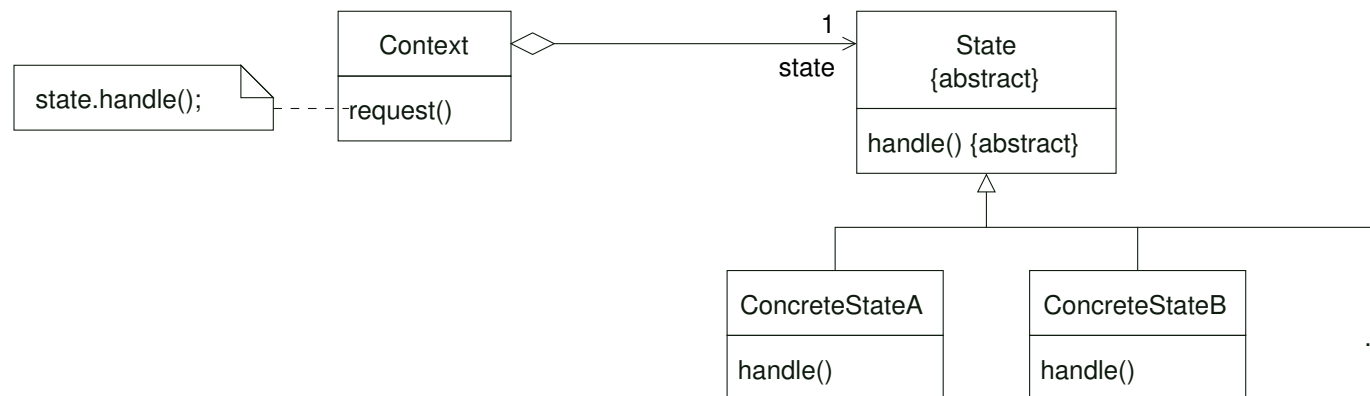
*Anwendungsbeispiel:* Komponenten in AWT/Swing; geometrische Figuren

## Beispiel 3: State (Behavioral Pattern)

### Zweck

Ein Objekt soll sein Verhalten in Abhängigkeit von seinem internen Zustand ändern können.

### Struktur



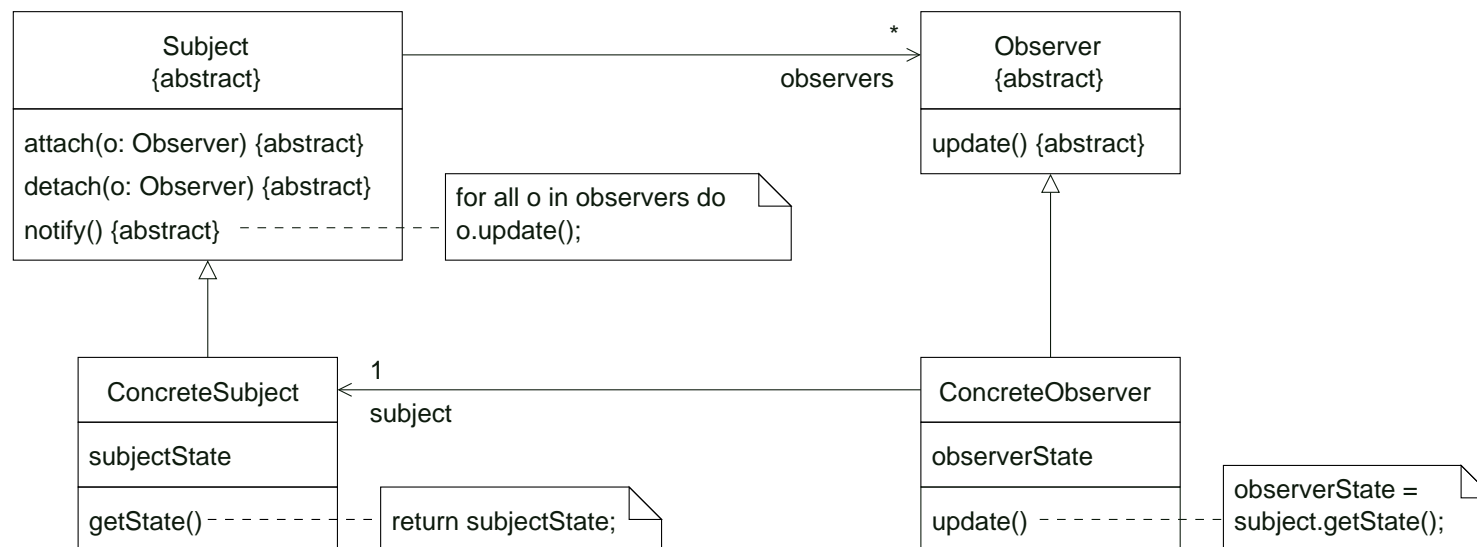
*Anwendungsbeispiel:* Realisierung von Zustandsdiagrammen durch Zustandsobjekte

## Beispiel 4: Observer (Behavioral Pattern)

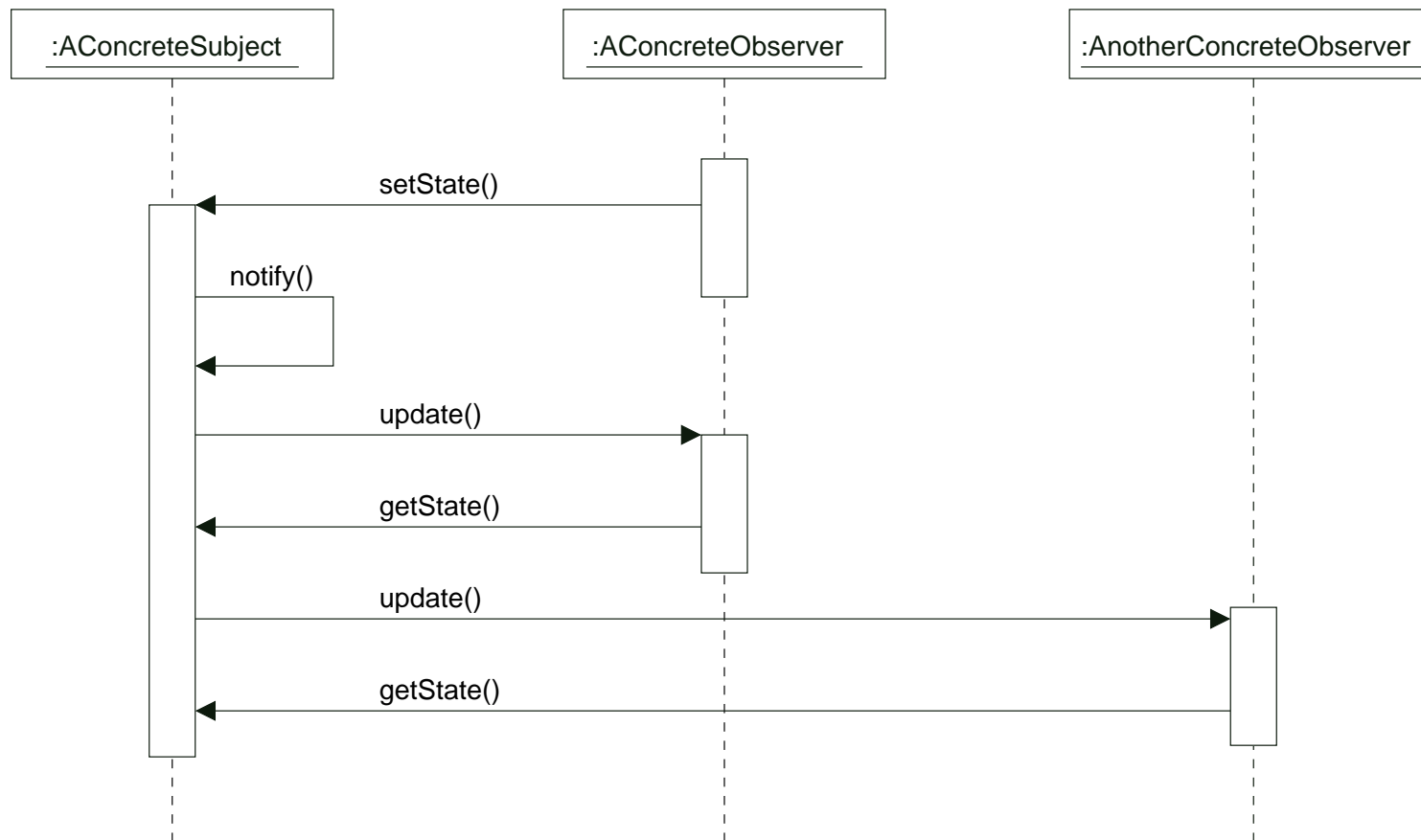
### Zweck

Definiert eine 1:\*-Beziehung zwischen Objekten, so dass, wenn ein Objekt seinen Zustand ändert, alle davon abhängigen Objekte über diese Zustandsänderung informiert werden.

### Struktur



## Interaktionen



*Anwendungsbeispiel:* Indirekte Kommunikation zwischen Anwendungskern und GUI

## Zusammenfassung von Abschnitt 4.7

- Ein Entwurfsmuster liefert eine generische Beschreibung einer bewährten Lösung für eine wiederkehrende Problemklasse.
- Die Beschreibung eines Entwurfsmuster erfolgt in einer strukturierten Form, die aus verschiedenen Elementen besteht (Name des Musters, Zweck, Anwendbarkeit, Lösungsstruktur, ...)
- Im Design-Pattern Katalog von Gamma et al. werden 23 objektorientierte Pattern beschrieben.
- Diese Pattern sind nach den Gesichtspunkten "Creational" (z.B. Abstract Factory), "Structural" (z.B. Composite) und "Behavioral" (z.B. State, Observer) klassifiziert.