

# Objektorientierte Software-Entwicklung

---

Prof. Dr. Rolf Hennicker

14.12.2007

# 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 Anbindung an eine relationale Datenbank vornehmen können.
- Entwurfsmuster kennenlernen.
- Prinzipien verteilter Objektsysteme kennen (wenn noch Zeit).

## *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 wird erweitert und überarbeitet. Hierzu werden Informationen aus dem dynamischen Modell der Analyse verwendet.

### Aufgaben des Objektentwurfs

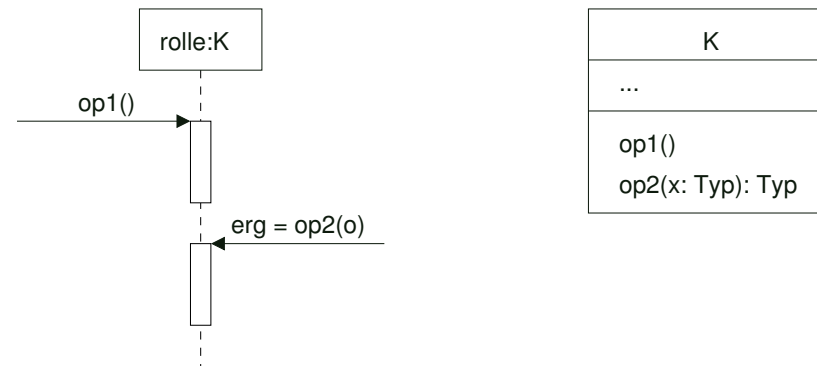
1. Operationen hinzufügen
2. Assoziationen ausrichten
3. Zugriffsrechte bestimmen
4. Mehrfachvererbung auflösen
5. Wiederverwendung von Klassen

## 4.1.1 Operationen hinzufügen

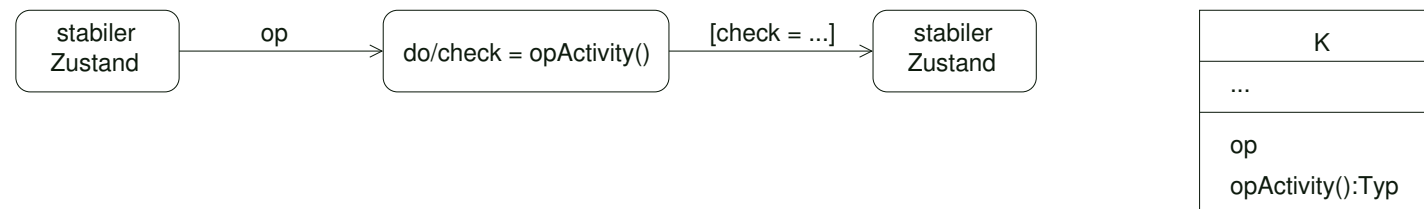
### Vorgehensweise

Sei K eine Klasse des Objektmodells.

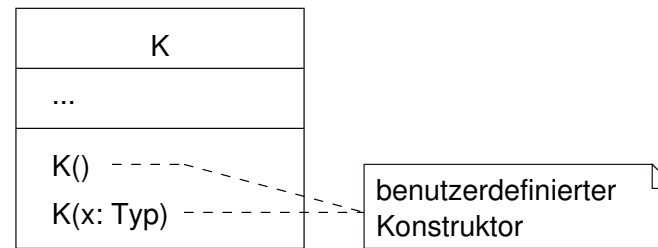
- Führe eine Operation für jede an ein Objekt von K gesendete Nachricht ein.



- Führe eine Operation für jedes Call-Event eines Zustandsdiagramms und für jede in einem Aktivitätszustand aufgerufene (lokale) Operation ein (ggf. auch für Aktionen in Aktivitätsdiagrammen).



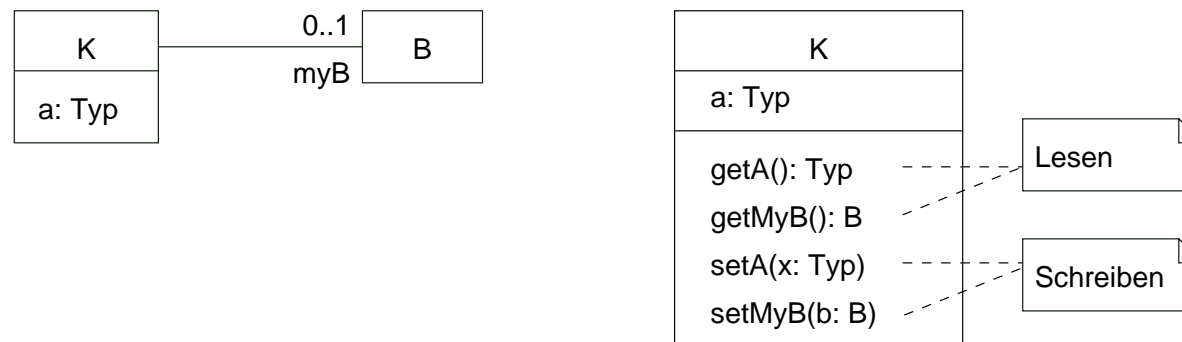
- Benutzerdefinierte Konstruktoren bei nicht abstrakten Klassen hinzufügen



Aufruf eines Konstruktors:

- im Sequenzdiagramm: *oder*
- im Pseudocode: `k = new K();` //falls k ein Rollename ist  
`K k = new K();` //falls k eine lokale Variable ist

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



*Beispiel ATM:*

ATM
geldvorrat: Real grenzen: Real
ATM()  karteEin abbruch geheimzahlEin abhebungWählen betragEin geldEntnehmen abschliessen karteBelegEntnehmen  karteneingabeAuffordernActivity() karteEinActivity(): String geheimzahlEinActivity(): String abhebungActivity() betragEinActivity(): String geldEntnehmenActivity() abschlussActivity()  karteLesen(): String geheimzahlÜberprüfen(): String grenzenÜberprüfen(): String

Konsortium
name: String
karteÜberprüfen(): String transaktionVerarbeiten(): String blzÜberprüfen(): String

Bank
blz: Integer name: String
bankKarteÜberprüfen(): String bankTransaktionVerarbeiten(): String kartennrÜberprüfen(): String kontoAktualisieren(): String

- Algorithmen der Operationen beschreiben

*Input:* Interaktions- oder (falls vorhanden) Aktivitätsdiagramme der Analyse

*Mögliche Darstellungen der Algorithmen:*

- Detaillierte Aktivitätsdiagramme (ggf. auch vollständige Interaktionsdiagramme)
- Pseudo-Code (z.B. basierend auf Java oder Verwendung einer "Action Language")

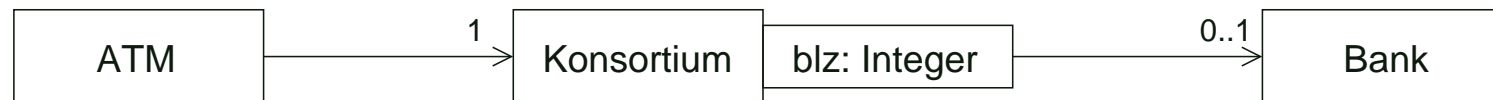
*Beachte:*

Während der Formulierung der Algorithmen wird das Objektmodell überarbeitet. Gegebenenfalls werden abgeleitete (redundante) Assoziationen zum direkteren Zugriff auf andere Objekte hinzugenommen.

## 4.1.2 Assoziationen ausrichten

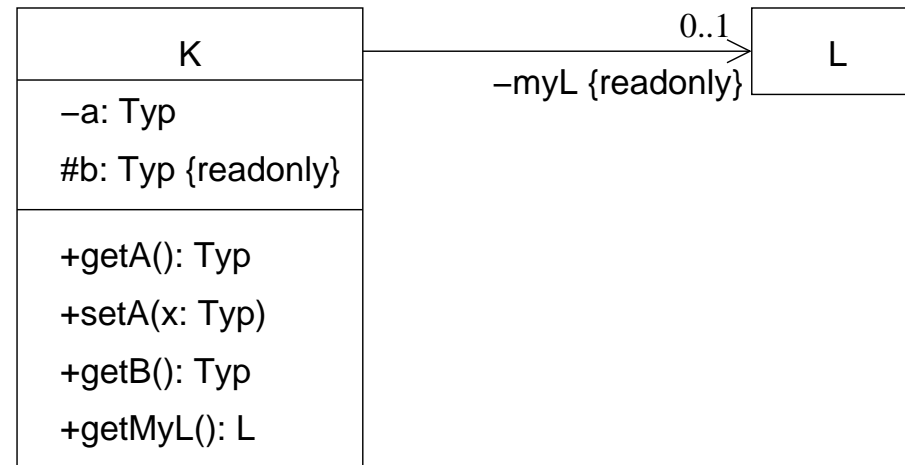
- Analysiere in welcher/welchen Richtung(en) eine Assoziation (beim Senden von Nachrichten bzw. Operationsaufrufen) durchlaufen wird.
- Falls eine Assoziation nur in einer Richtung durchlaufen wird, dann richte sie entsprechend aus.

*Beispiel ATM:*



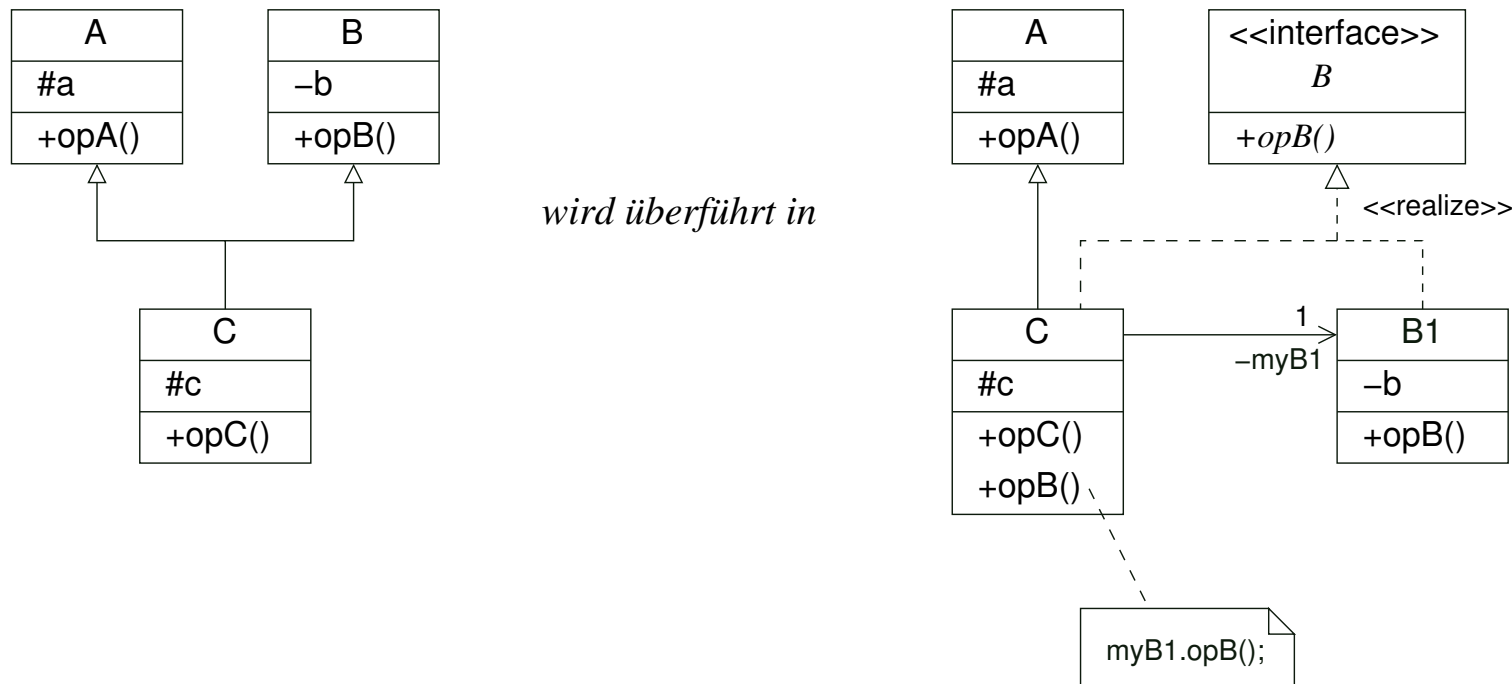
### 4.1.3 Zugriffsrechte bestimmen

Bestimme die Zugriffsrechte für Attribute, Rollennamen und Operationen (Attribute und Rollennamen sollten nicht öffentlich zugreifbar sein!)



## 4.1.4 Mehrfachvererbung auflösen

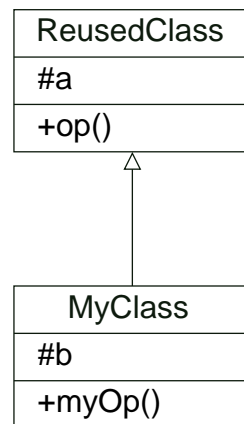
- Ist notwendig, wenn die Zielsprache keine Mehrfachvererbung für Klassen unterstützt (z.B. Java).
- Die Auflösung der Mehrfachvererbung ist möglich durch Einführung einer Schnittstelle.



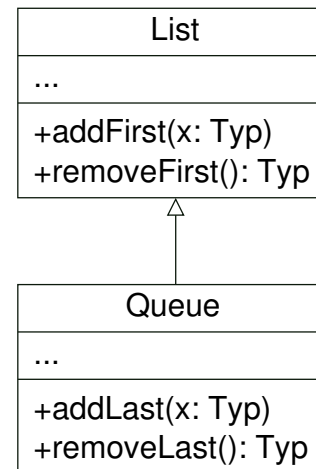
## 4.1.5 Wiederverwendung von Klassen

- 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:*



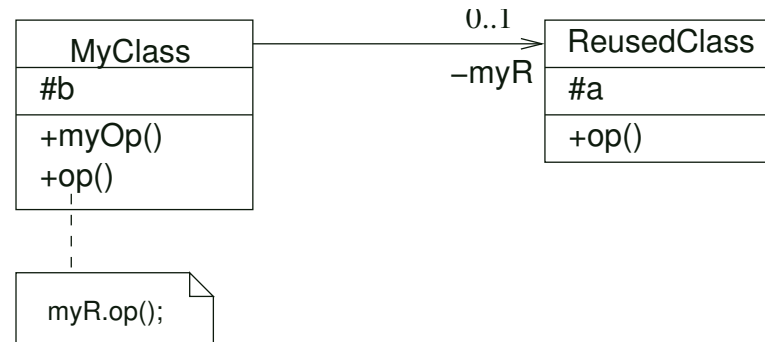
*Beispiel:*



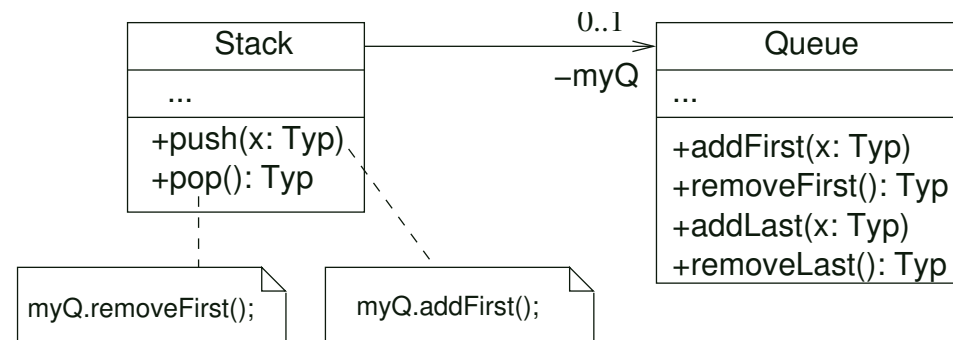
**Beachte:** Wenn die wiederverwendete Klasse auch Operationen anbietet, die die spezielle Klasse nicht benötigt, ist eventuell das Kapselungsprinzip verletzt. In diesem Fall ist Wiederverwendung durch *Delegation* vorzuziehen.

- Wiederverwendung durch *Delegation*:

*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).
- Die Algorithmen werden durch Verfeinerung aus den Aktivitätsdiagrammen in Kapitel 3.4 hergeleitet.
- Zusätzlich benötigte Konstruktoren und Zugriffsoperationen ("getter" und "setter") werden identifiziert.
- Das Klassendiagramm der Analyse wird entsprechend überarbeitet.

## Algorithmen

### 1. Operationen der Klasse ATM

Operationen, die als Ereignisse im Zustandsdiagramm vorkommen (karteEin, ..., karteBelegEntnehmen) werden hier nicht betrachtet.

Eine geeignete Behandlung wird später bei der Realisierung des Zustandsdiagramms gegeben.

```
// Konstruktor ATM
ATM() {
    geldvorrat = 100000;
    grenzen    = 250;
    karteneingabeAuffordernActivity();
}
```

```
karteneingabeAuffordernActivity() {
    display("Karte eingeben?");
}
```

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

    if (check.equals("lesbar")) {
        display("Geheimzahl?");
        return "Karte lesbar";
    }
    else if (check.equals("nicht lesbar")) {
        display("Karte nicht lesbar!");
        abschlussActivity();
        return "Karte nicht lesbar";
    }
    else return "Error";
}
```

```
geheimzahlEinActivity(): String {
    Integer typedGeheimzahl = getTypedGeheimzahl();
    String check = geheimzahlUeberpruefen(typedGeheimzahl);

    if (check.equals("Geheimzahl ok")) {
        // neues Attribut aktKontonr
        check = konsortium.karteUeberpruefen(aktKartennr, aktBLZ, aktKontonr);
        if (check.equals("Karte ok")) {
            display("Transaktionsform?");
            return "Karte ok";
        }
        else if (check.equals("falsche BLZ")) {
            display("falsche BLZ!");
            abschlussActivity();
            return "Karte nicht ok";
        }
    }
}
```

```
else if (check.equals("Karte gesperrt")) {
    display("Karte gesperrt!");
    abschlussActivity();
    return "Karte nicht ok";
}
else return "Error";
}

else if (check.equals("falsche Geheimzahl")) {
    display("falsche Geheimzahl!");
    display("Geheimzahl?");
    return "Geheimzahl falsch";
}
else return "Error";
}

abhebungActivity() {
    display("Betrag?");
}
```

```
betragEinActivity(): String {
    Real betrag = 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;
            display("Geld ausgeben");
            display("Geld entnehmen?");
            return "Transaktion erfolgreich";
        }
        else if (check.equals("Transaktion gescheitert")) {
            display("Transaktion gescheitert!");
            display("Transaktionsform?");
            return "Transaktion gescheitert";
        }
        else return "Error";
    }
}
```

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

geldEntnehmenActivity() {
    display("Fortsetzung?");
}

abschlussActivity() {
    display("Beleg drucken");
    display("Karte ausgeben");
    display("Karte und Beleg entnehmen?");
}
```

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

```
geheimzahlUeberpruefen(tgz: Integer): String {
    if (tgz == aktGeheimzahl) return "Geheimzahl ok";
    else return "falsche Geheimzahl";
}
```

```
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].bankKarteUeberpruefen(kartennr, kontonr);  
        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";  
}
```

```
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";
    else return "Error";
}

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

### 3. Operationen der Klasse *Bank*

```
bankKarteUeberpruefen(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";  
    else return "Error";  
}
```

```
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";  
    else return "Error";  
}
```

```
kartennrUeberpruefen(kartennr: Integer, out kontonr: Integer) :String {
    // kreditkarten ist Rollenname 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 "gueltig";
        else return "gesperrt";
    }
}
```

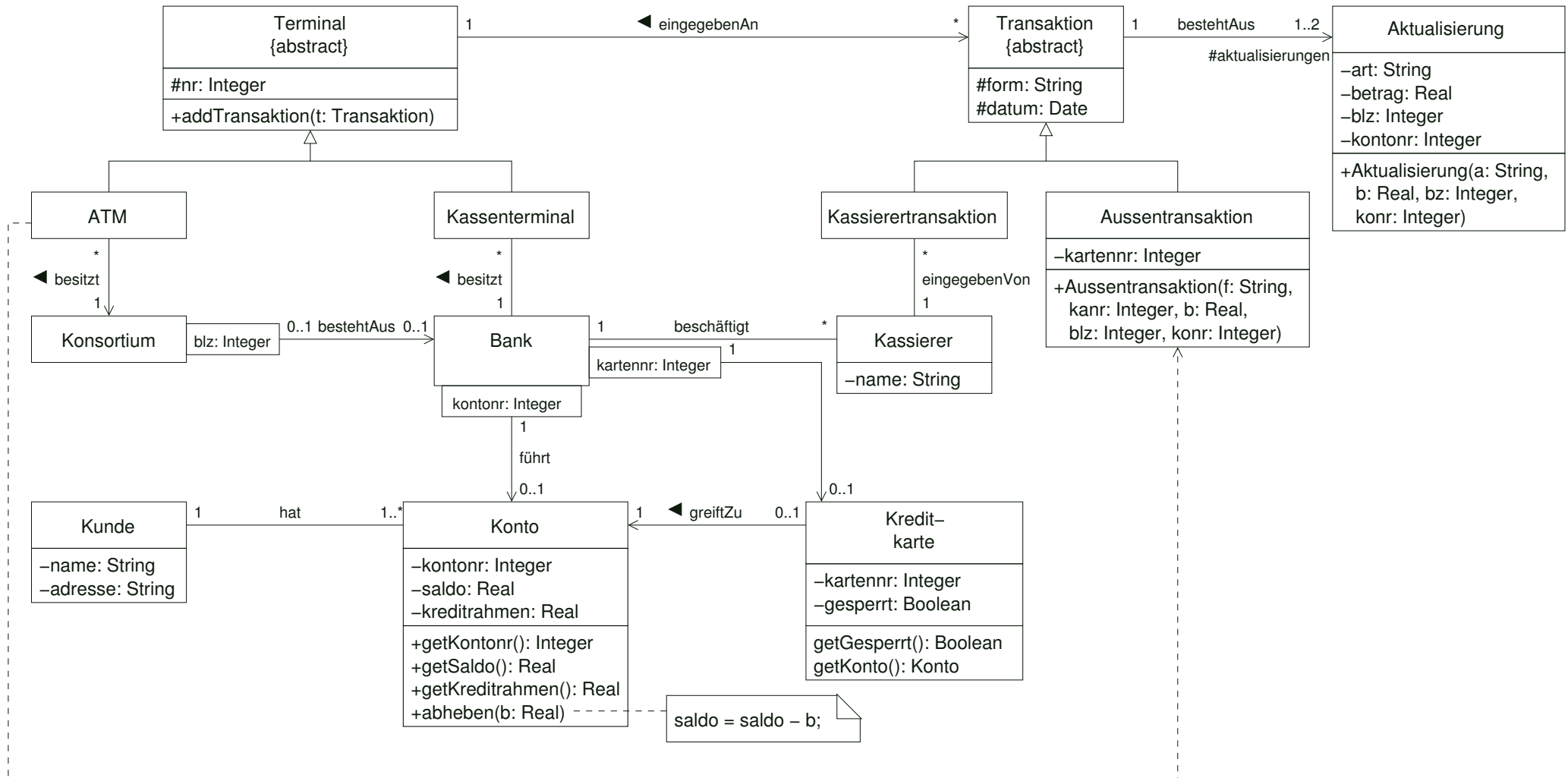
```
kontoAktualisieren(kontonr: Integer, b: Real): String {
    Konto k = konten[kontonr];
    if (k.getSaldo()-b >= k.getKreditrahmen()) {
        k.abheben(b);
        return "erfolgreich";
    }
    else return "gescheitert";
}
```

## 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

ATM
-geldvorrat: Real -grenzen: Real -aktKartennr: Integer -aktBLZ: Integer -aktGeheimzahl: Integer -aktKontonr: Integer
+ATM() +karteEin +abbruch +geheimzahlEin +abhebungWaehlen +betragEin +geldEntnehmen +abschliessen +karteBelegEntnehmen  -karteneingabeAuffordernActivity() -karteEinActivity(): String -geheimzahlEinActivity(): String -abhebungActivity() -betragEinActivity(): String -geldEntnehmenActivity() -abschlussActivity()  -karteLesen(): String -geheimzahlUeberpruefen(tgz: Integer): String -grenzenUeberpruefen(b: Real): String

Konsortium
-name: String
+karteUeberpruefen(kartennr: Integer, blz: Integer, out kontonr: Integer): String +transaktionVerarbeiten(blz: Integer, kontonr: Integer, b: Real): String +blzUeberpruefen(blz: Integer): String

Bank
-blz: Integer
-name: String
+bankKarteUeberpruefen(kartennr: Integer, out kontonr: Integer): String +bankTransaktionVerarbeiten(kontonr: Integer, b: Real): String -kartennrUeberpruefen(kartennr: Integer, out kontonr: Integer): String -kontoAktualisieren(kontonr: Integer, b: Real): String

## 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.
- Die Algorithmen von (nicht-trivialen) Operationen werden beschrieben durch
  - (möglichst vollständige) Aktivitätsdiagramme oder durch
  - Pseudo-Code, der durch Verfeinerung aus Aktivitätsdiagrammen hergeleitet ist.
- Während der Formulierung von Algorithmen für die Operationen wird das statische Modell laufend überarbeitet (u.a. Ausrichten von Assoziationen, Einführung von abgeleiteten Assoziationen).
- 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

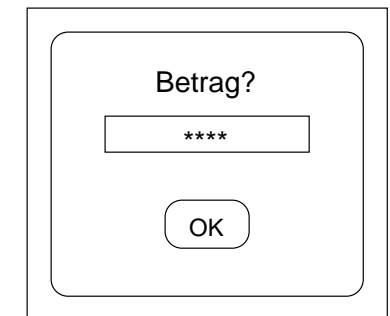
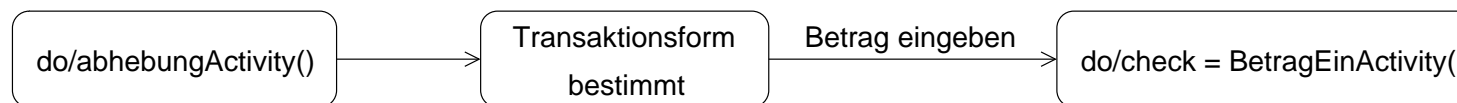
### *Idee*

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

### *Voraussetzung*

Objekte befinden sich an der Systemgrenze (Kontrollobjekte zur Dialogsteuerung)

### *Beispiel:*



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

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

## *Vorgehensweise*

Das gesamte Zustandsdiagramm wird überführt in eine Prozedur mit

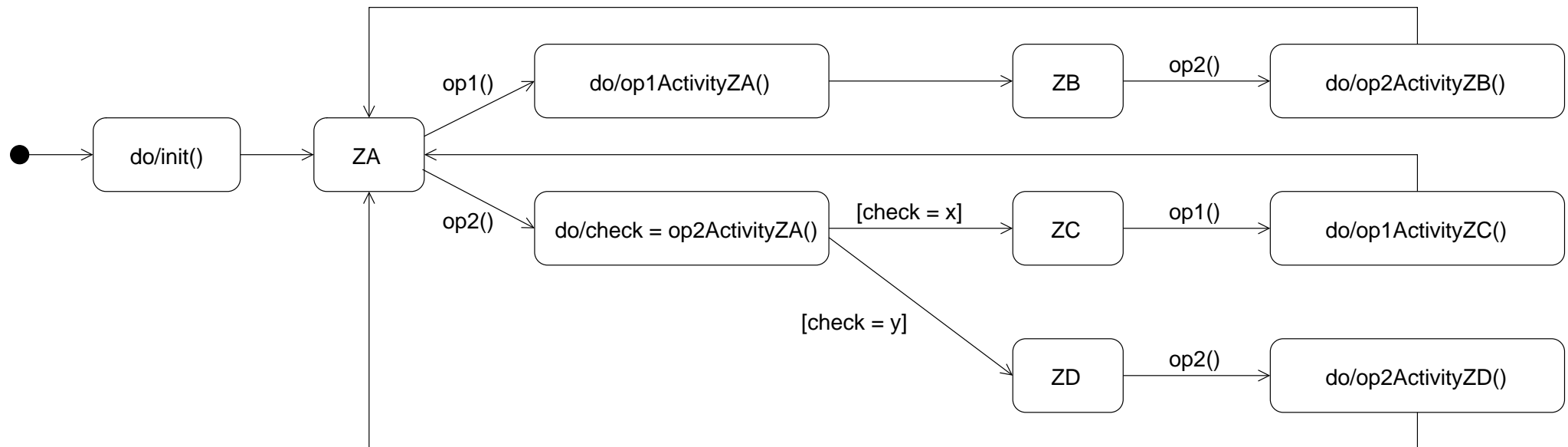
- modalen Dialogen für die (externen) Ereignisse
- bedingten Anweisungen für Verzweigungen
- Wiederholungsanweisungen für Zyklen des Diagramms

## **Bemerkung:**

Flexible Benutzerschnittstellen sind so nur schwer zu realisieren.

## 4.2.2 Realisierung durch Fallunterscheidung

Gegeben sei folgendes Zustandsdiagramm für die Objekte einer Klasse K:



**Gesucht:** Realisierung von op1 und op2 sowie des Konstruktors von K.

## *Vorgehensweise*

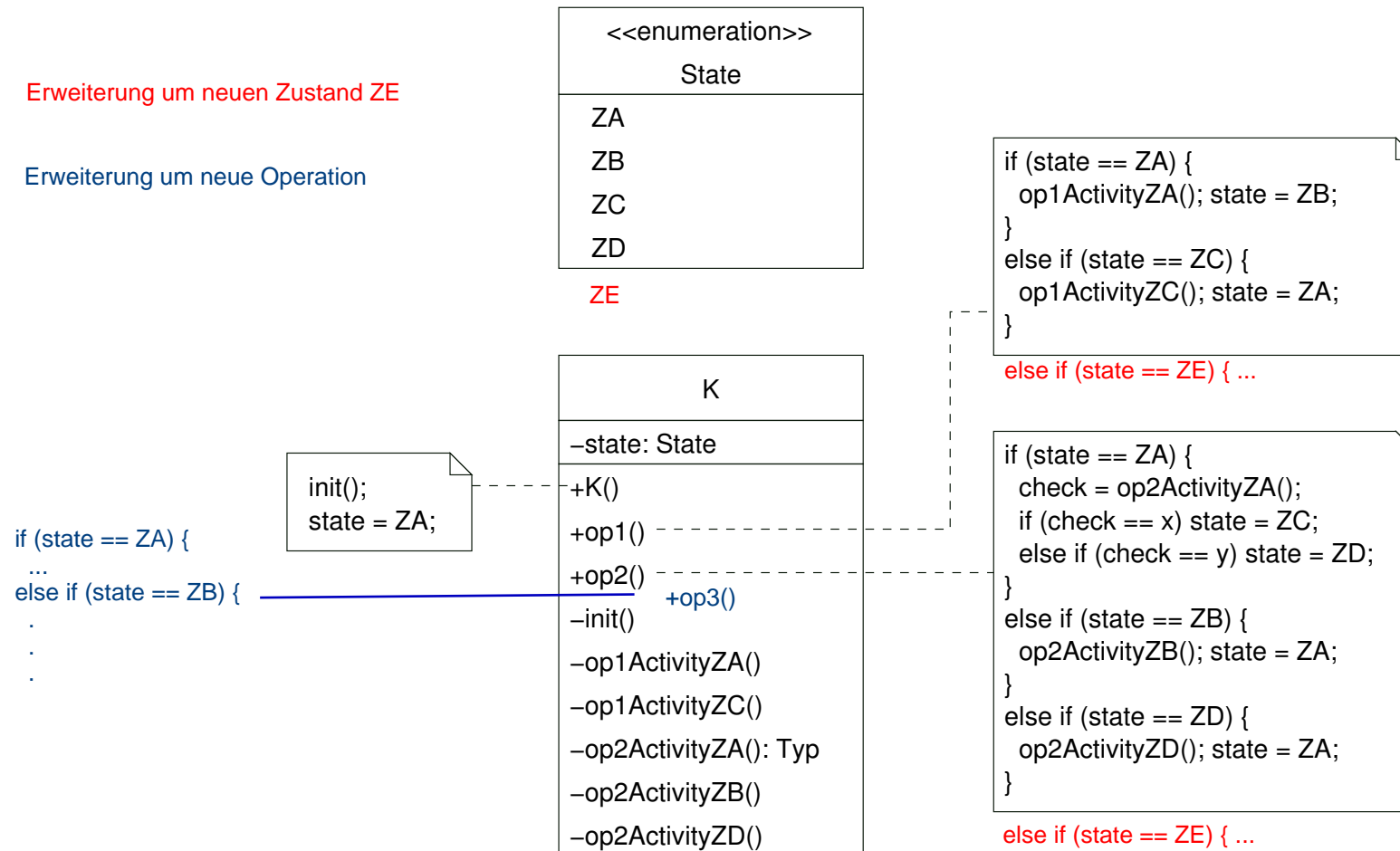
- Verwende einen Enumerationstyp zur Darstellung der (endlich vielen) stabilen Zustände.
- Führe ein explizites Zustandsattribut für die betrachtete Klasse  $K$  ein.
- Realisiere die zustandsabhängigen Operationen durch Fallunterscheidung nach dem aktuellen (stabilen) Zustand.

## *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.

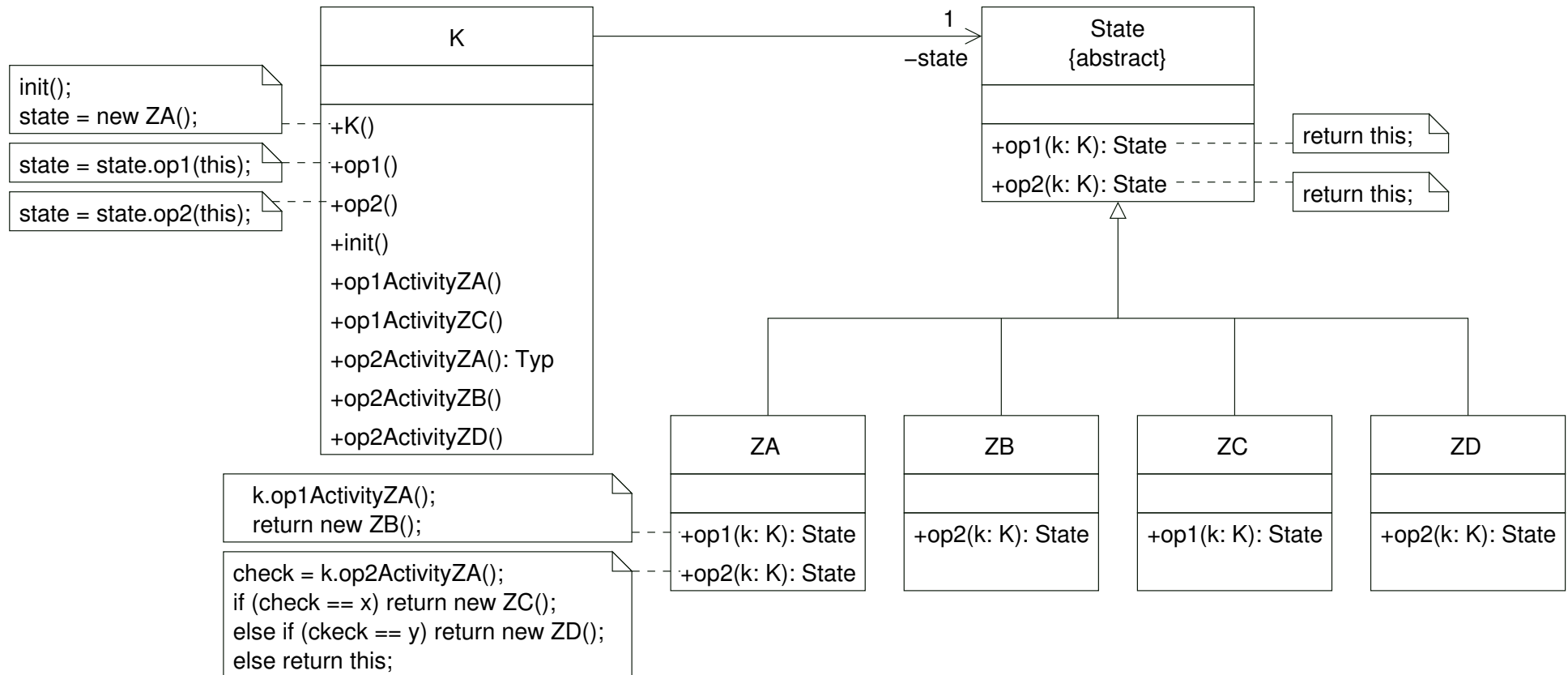
### *Vorteil*

Einfache Erweiterbarkeit bzgl. neuer Zustände.

### *Nachteil*

Schlechte Erweiterbarkeit bzgl. neuer Operationen.

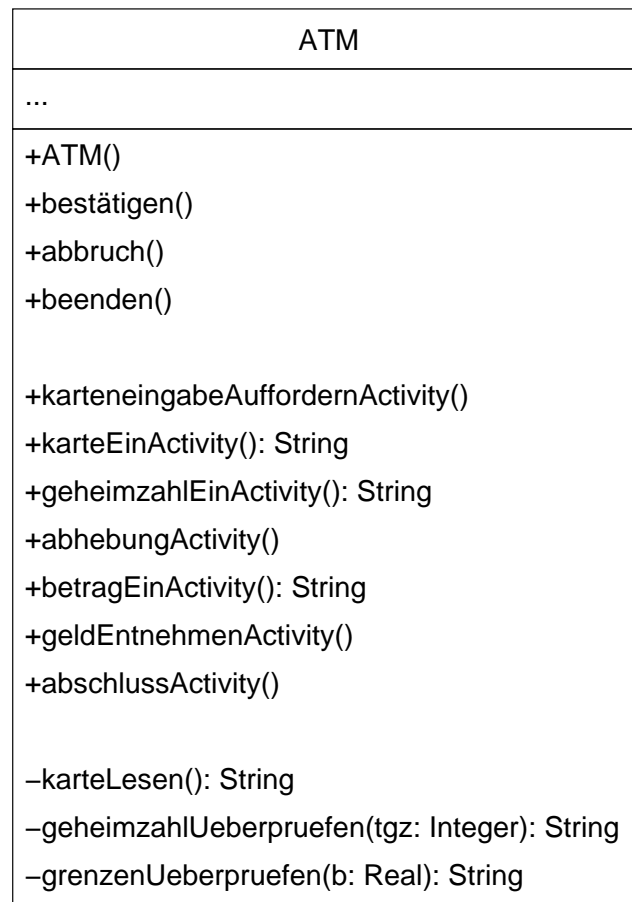
Das Zustandsdiagramm von oben wird folgendermaßen realisiert:



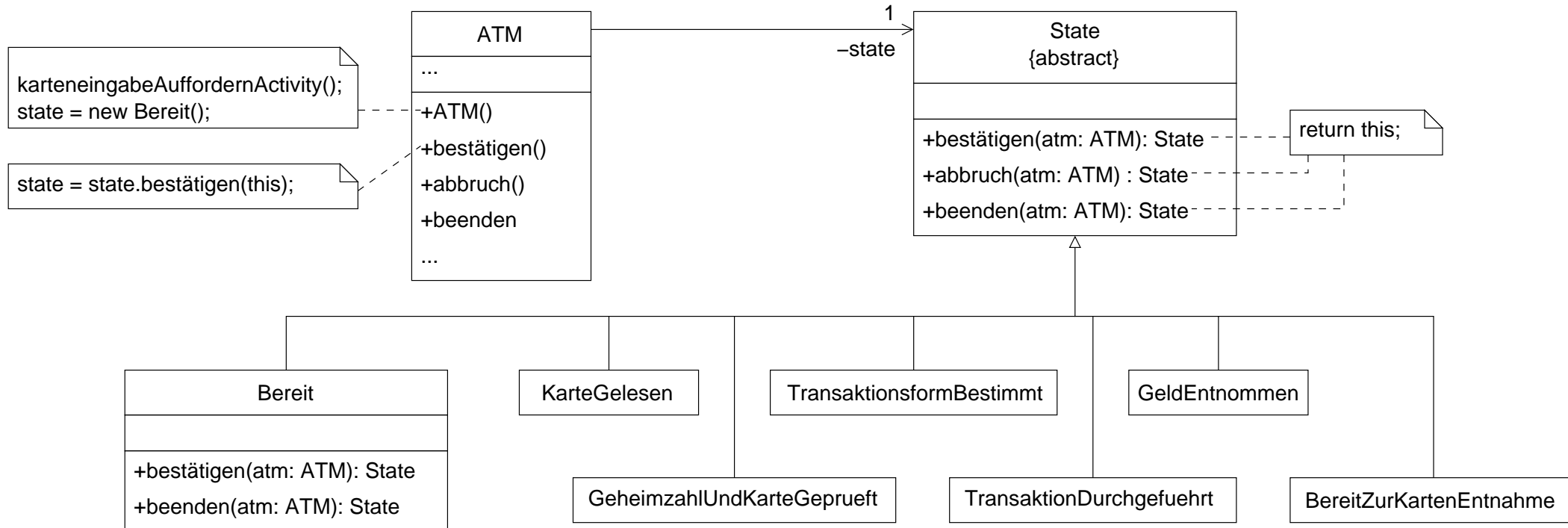


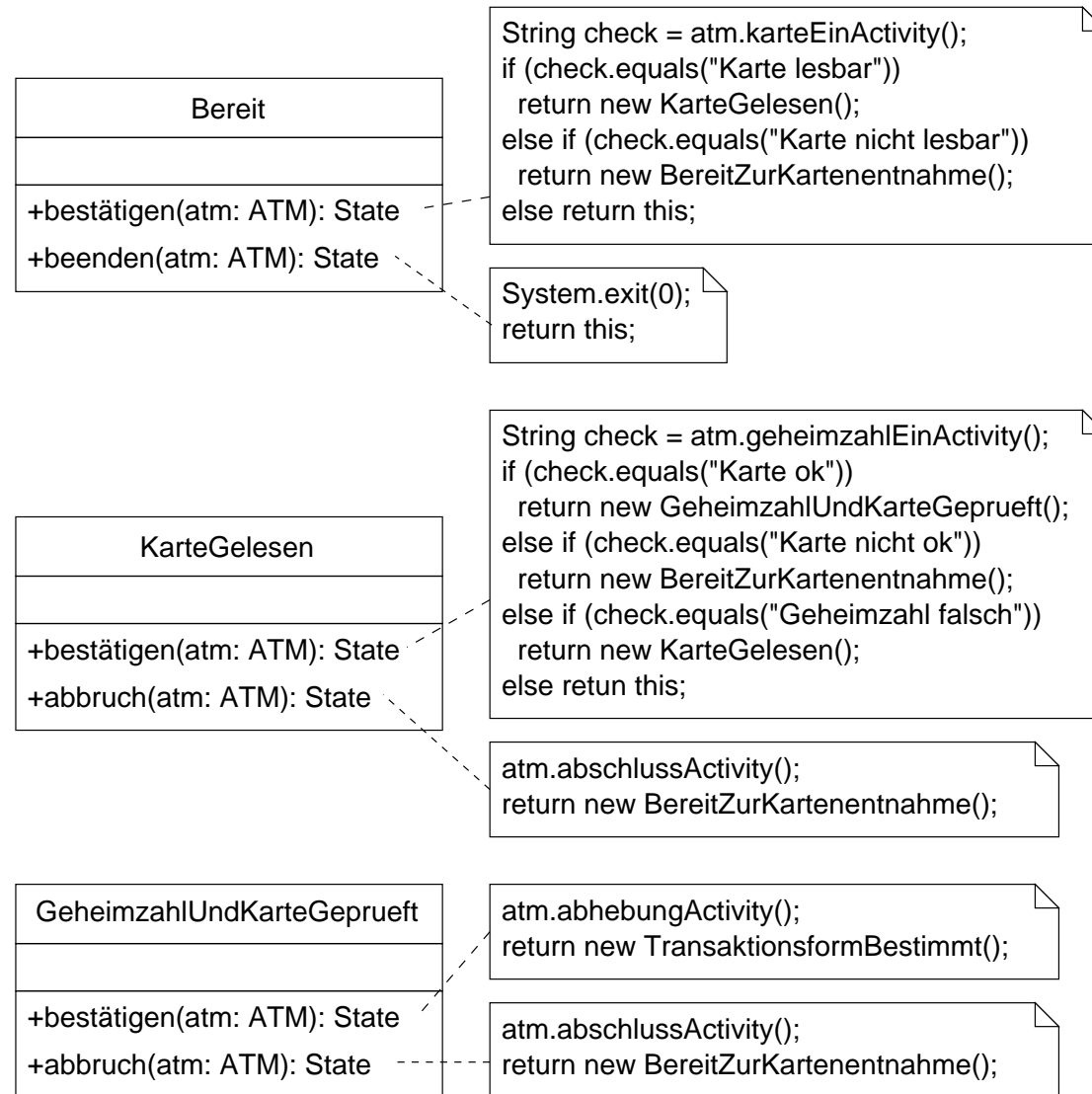
## Überarbeitete Klasse ATM

Die Operationen "karteEin", "geheimzahlEin", ..., "karteBelegEntnehmen" von früher werden entfernt und durch die zustandsabhängige Operation "bestätigen" realisiert.



## Realisierung des Zustandsdiagramms der ATM-Simulation





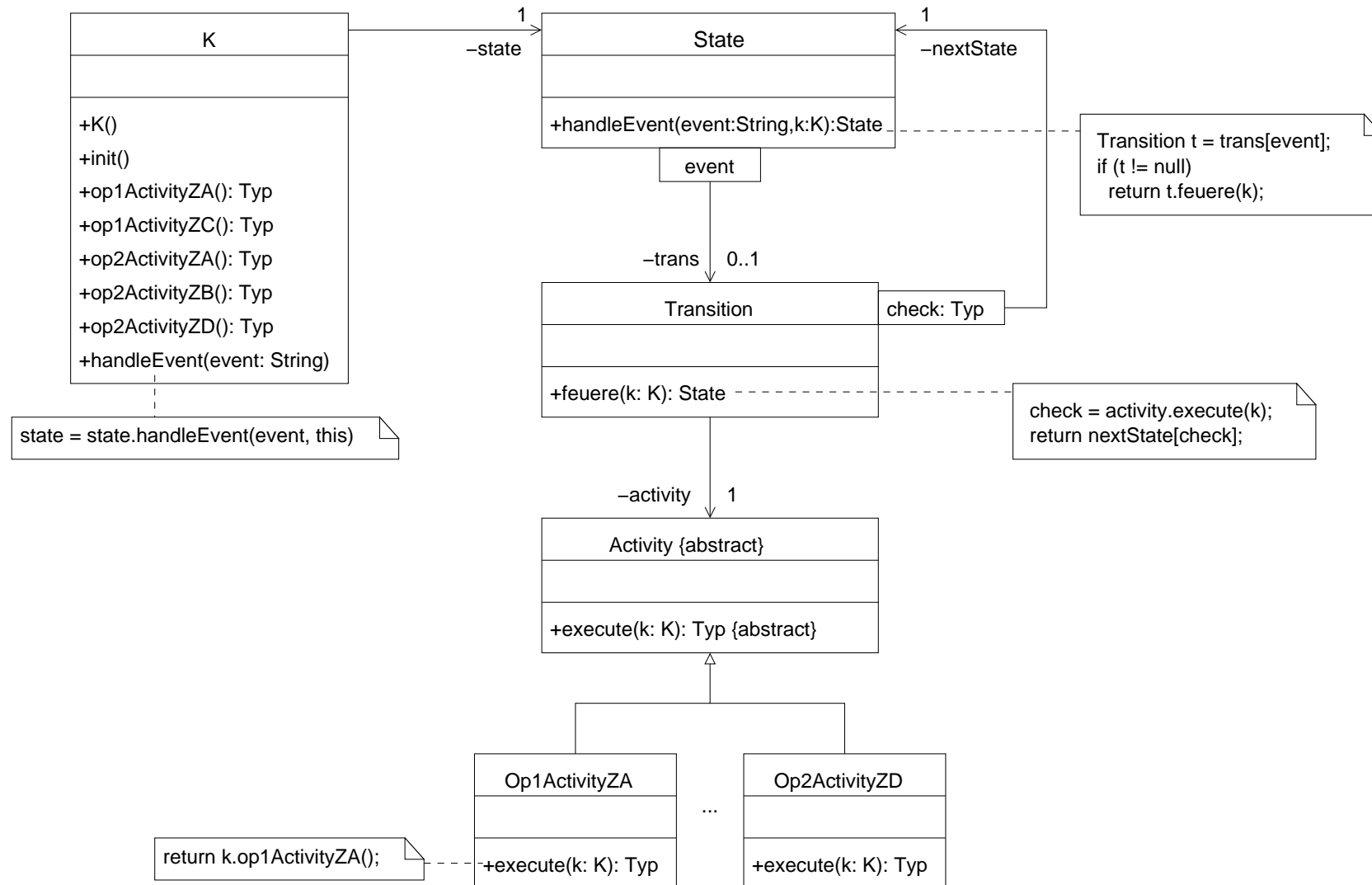
Analog werden die vier übrigen Zustandsklassen implementiert.

## 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 gesamte Zustandsdiagramm wird durch eine (verzeigerte) Objektstruktur repräsentiert.

Das Zustandsdiagramm von oben wird durch folgende Zustandsmaschine realisiert:



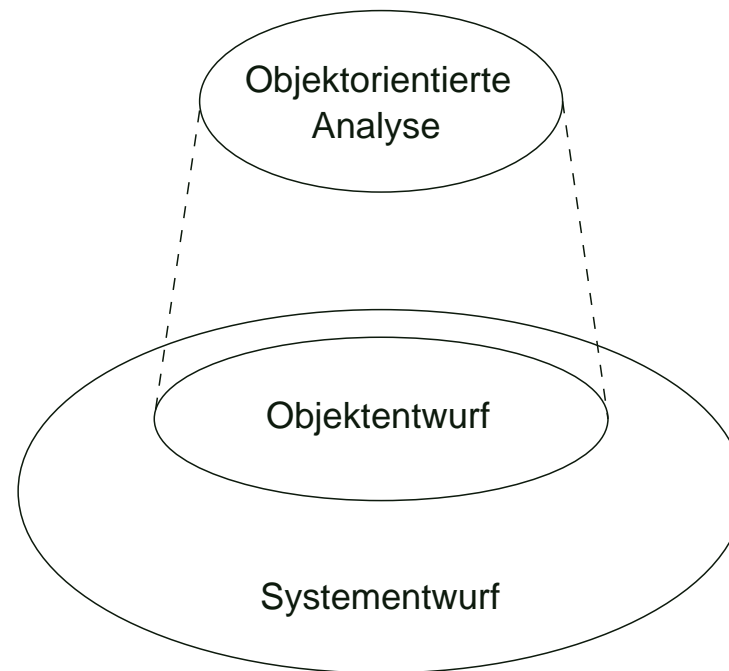
## Zusammenfassung von Abschnitt 4.2

- Zustandsdiagramme können systematisch realisiert und in einen Objektentwurf integriert werden.
- Wir unterscheiden 4 Möglichkeiten der Realisierung:
  - Prozedurgesteuert
  - Fallunterscheidung
  - Zustände als Objekte
  - Zustandsmaschine
- Der gesamte Objektentwurf für das Beispiel der ATM-Simulation besteht nun aus
  - dem Klassendiagramm von Abschnitt 4.1 mit der in Abschnitt 4.2 überarbeiteten Klasse ATM.
  - den in Abschnitt 4.1 entwickelten Algorithmen.
  - der Realisierung des Zustandsdiagramms der ATM-Simulation (Abschnitt 4.2).

## 4.3 Systementwurf

### Ziele

- Einbettung des Objektentwurfs in die Systemumgebung
- Festlegung der Systemarchitektur



## 4.3.1 Pakete und Komponenten

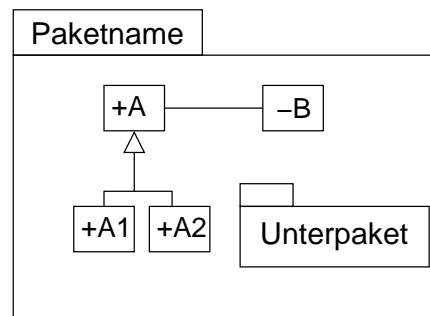
Pakete dienen zur Strukturierung von Modellen größerer Systeme. Sie fassen mehrere Modellelemente in einer Einheit (Gruppe) zusammen.

### Darstellung von Paketen in UML

*Paket ohne Darstellung der Inhalte:*



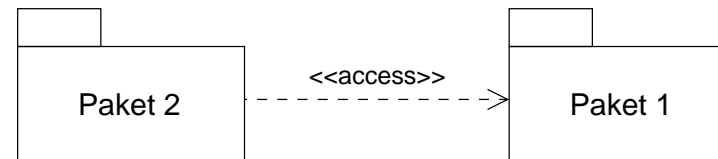
*Paket mit Darstellung der Inhalte:*



## Import-Beziehungen zwischen Paketen

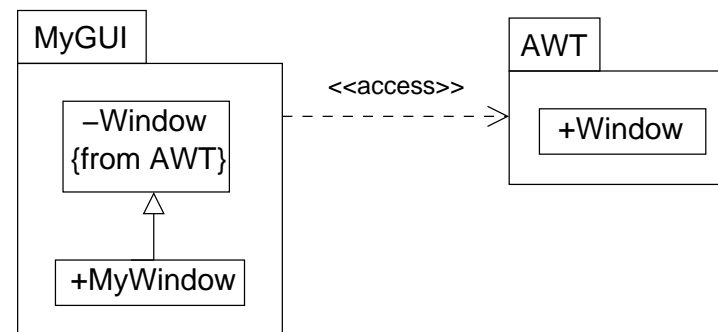
Durch Import-Beziehungen können die Namen von öffentlichen Modellelementen eines (importierten) Pakets in den Namensraum eines anderen (importierenden) Pakets übernommen werden.

### 1. Privater Paket-Import:



Die Sichtbarkeit der importierten Elemente wird auf "privat" gesetzt.

*Beispiel:*

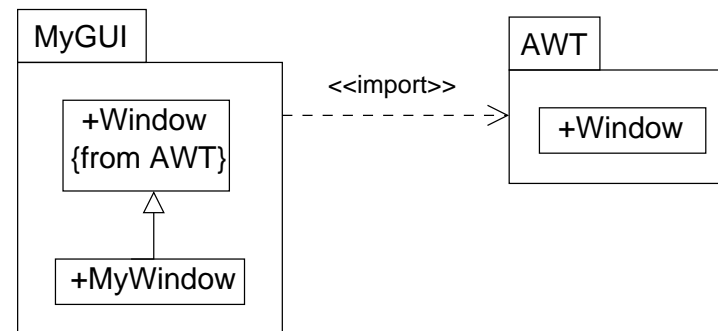


## 2. Öffentlicher Paket-Import:



Die Sichtbarkeit der importierten Elemente wird auf “public” gesetzt. Die importierten Elemente können damit transitiv weiter importiert werden.

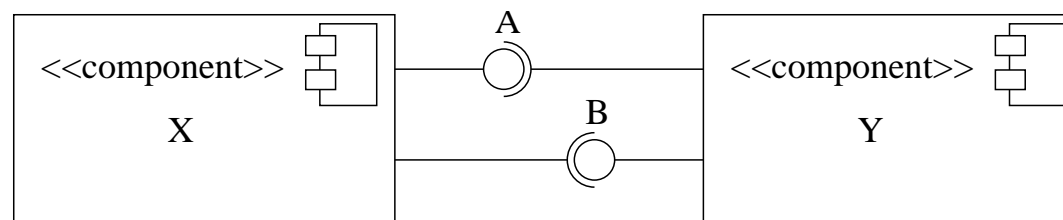
*Beispiel:*



## Komponenten

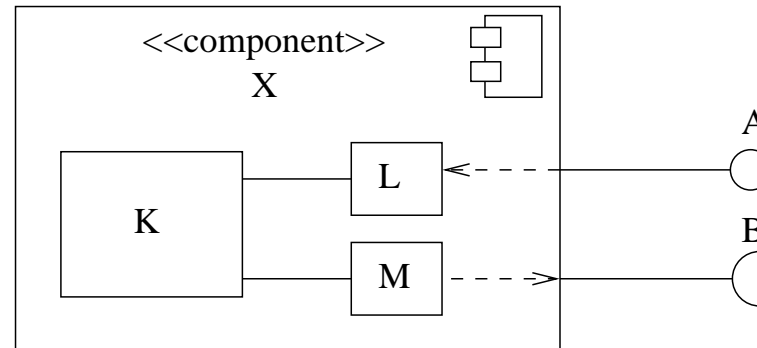
Eine Komponente ist ein modularer Teil eines Systems, der eine komplexe (interne) Struktur verkapselt und mit der Umgebung über Schnittstellen kommuniziert.

### *Externe Sicht von Komponenten*



Das Interface A wird von X zur Verfügung gestellt (*provided interface* von X) und von Y benutzt (*required interface* von Y).

## Interne Sicht von Komponenten



## Bemerkungen

- Komponenten können über *Ports* verfügen, die (evtl. mehrere angebotene und benutzte) Interfaces gruppieren. Das Verhalten von Ports kann durch UML *protocol state machines* spezifiziert werden.
- Für die Realisierung einer Komponente unterscheiden wir zwischen *indirekter* und *direkter Implementierung*. Bei indirekter Implementierung wird die interne Sicht durch Klassendiagramme beschrieben (vgl. oben), bei direkter Implementierung durch *Kompositionsstrukturdiagramme*.
- Komponenten können auch eigene Attribute und Operationen besitzen.

## 4.3.2 Grundlagen der Systemarchitektur

Die Systemarchitektur beschreibt die Gesamtstruktur des SW-Systems durch Angabe von Subsystemen und von Beziehungen zwischen den Subsystemen (ggf. unter Verwendung von Schnittstellen).

### *Bemerkungen*

- Eine grobe Systemarchitektur wird häufig schon zu Beginn der Systementwicklung angegeben.
- Subsysteme werden dargestellt durch Pakete oder, falls Schnittstellen verwendet werden, durch Komponenten.

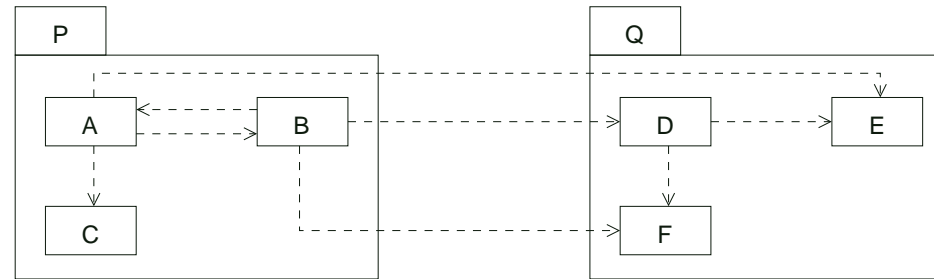
### *Grundregeln*

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

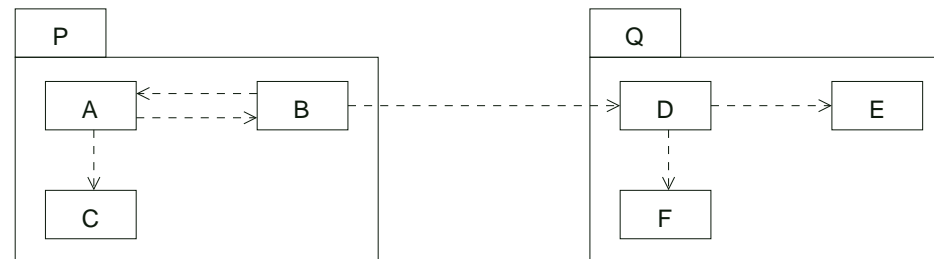
*Vorteil:* Leichte Änderbarkeit und Austauschbarkeit von einzelnen Teilen.

*Beispiel:*

## Subsysteme mit hoher Kopplung



## Subsysteme mit geringer Kopplung



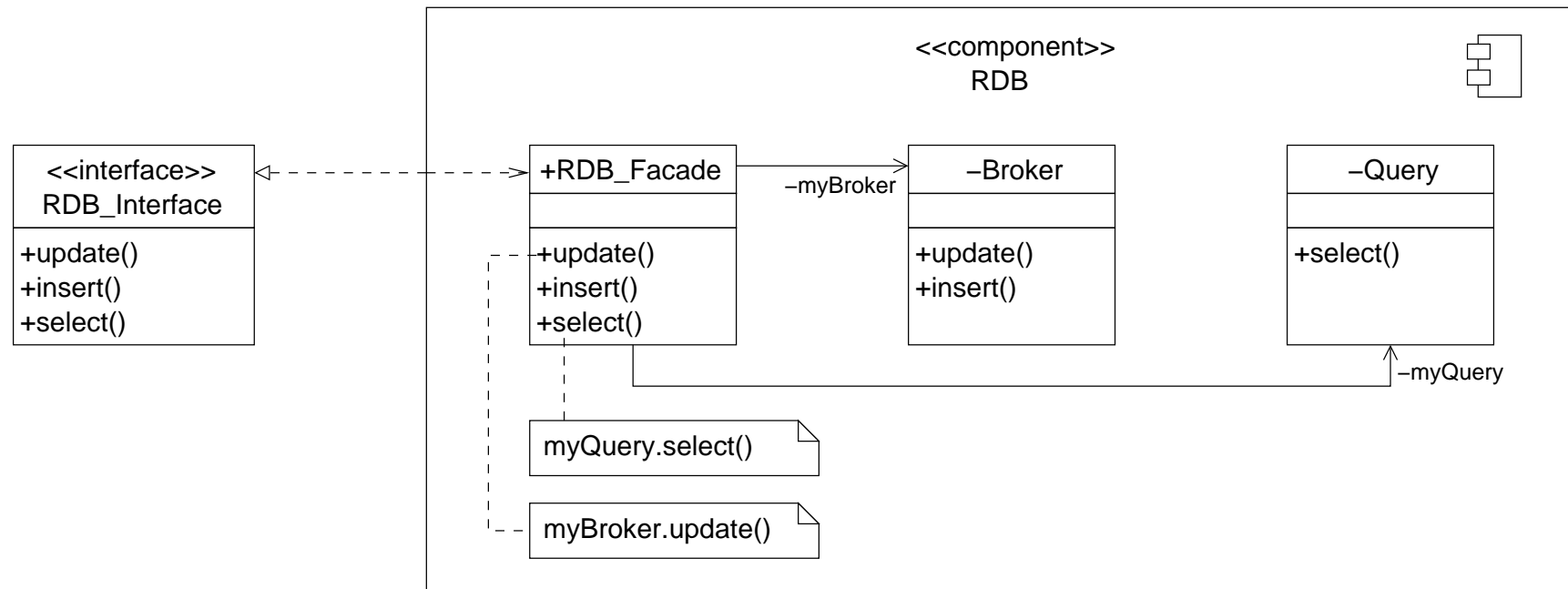
**Beachte**

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 eines Subsystems zusammen und delegieren Aufrufe an die “zuständigen” Objekte.
- Realisieren häufig ein Interface einer Komponente.

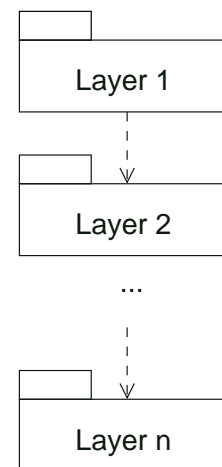
*Beispiel:*



## Schichtenarchitekturen

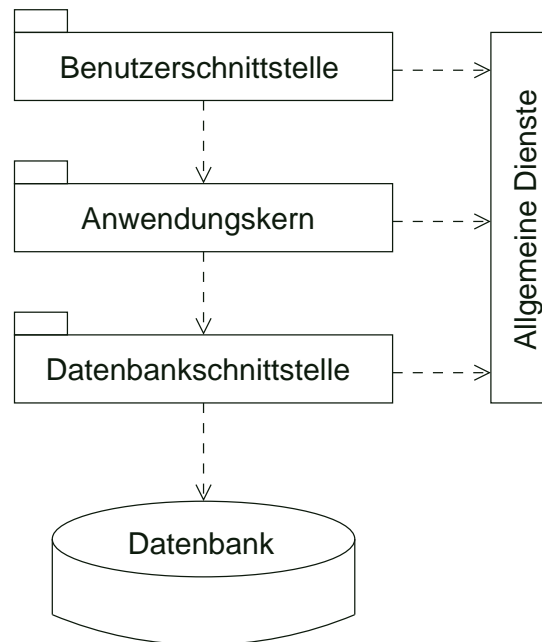
In vielen Systemen findet man *Schichtenarchitekturen*, wobei jede untere Schicht Dienste für die darüberliegende(n) Schicht(en) bereitstellt.

*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 Subsystemen 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*

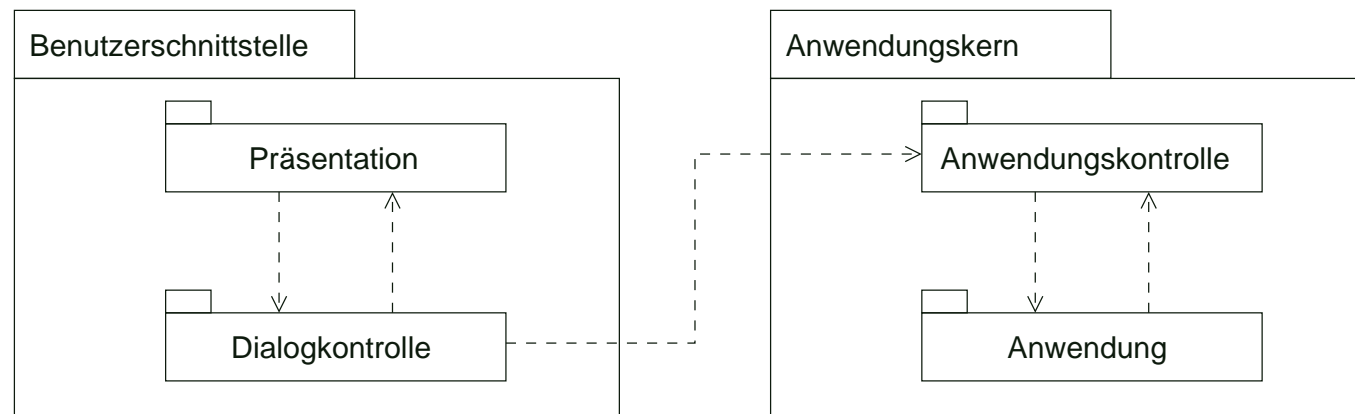
Sorgt für die Speicherung von und den Zugriff auf persistente Daten der Anwendung.

## *Allgemeine Dienste*

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

## Kontrollobjekte

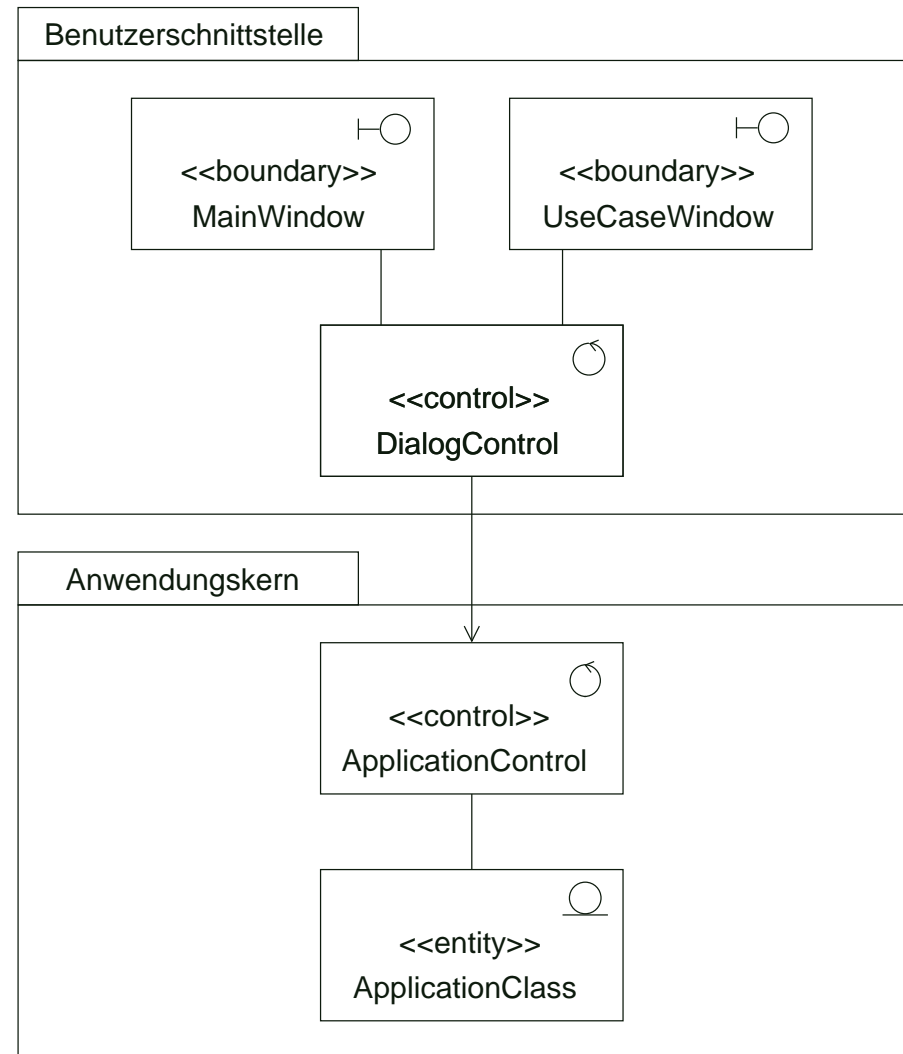
Häufig werden eigene Objekte zur Dialogsteuerung (z.B. Verwaltung mehrerer Fenster) oder zur Steuerung der Aufgaben des Anwendungskerns verwendet (z.B. ein Kontrollobjekt pro Use Case).



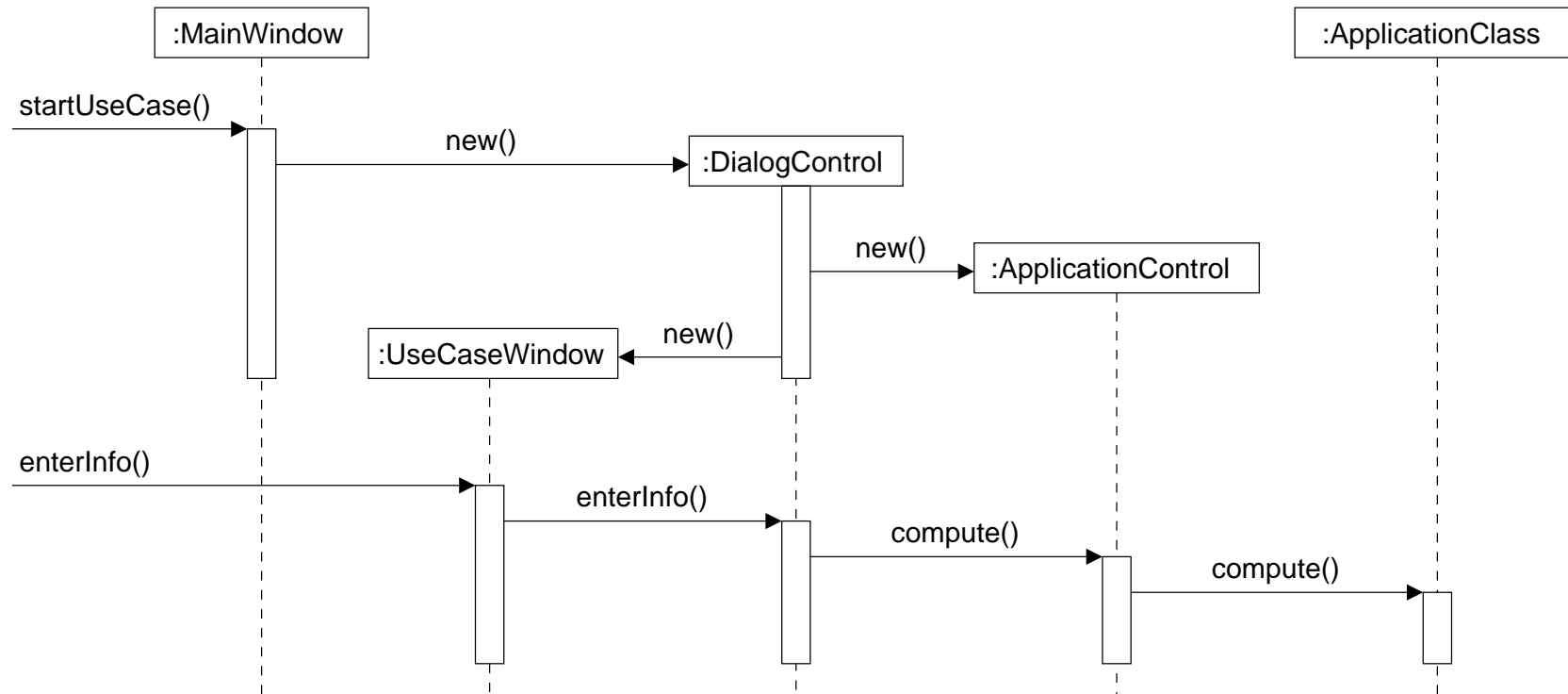
### *Bemerkung*

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

## Schnittstellen-, Kontroll- und Entity-Klassen



## Typisches Interaktionsmuster mit Kontrollobjekten



## 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!

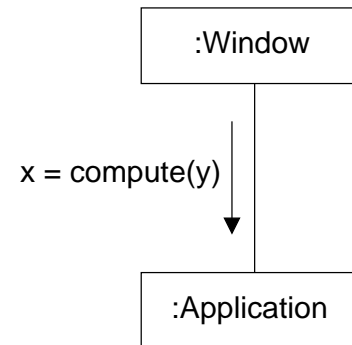
### *Problem*

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).

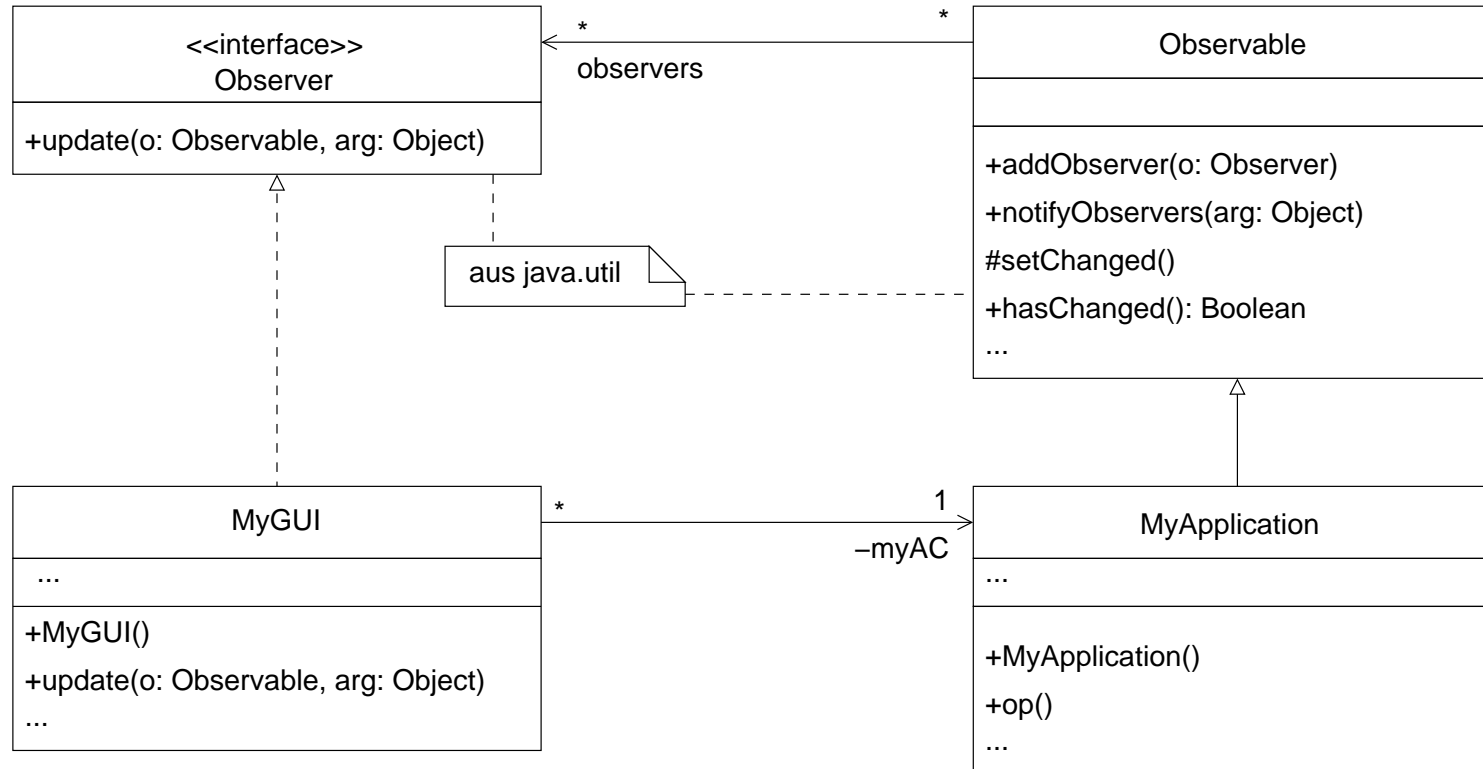
- Indirekte Kommunikation: Event-Manager oder Observer

## Indirekte Kommunikation durch Verwendung von Beobachtern

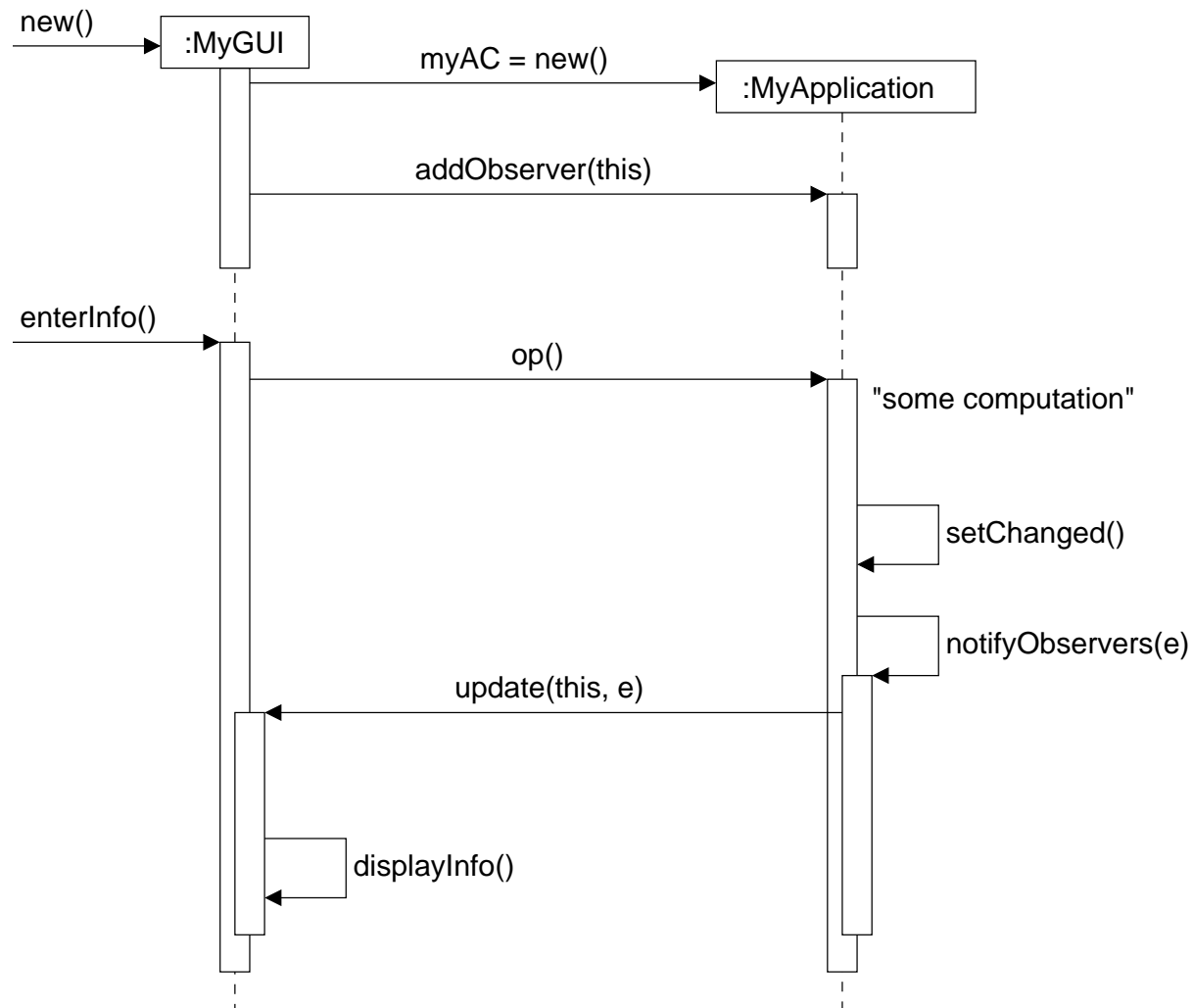
### *Idee*

- 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 Subsystemen (Komponenten) und Beziehungen zwischen den Subsystemen.
- 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!)

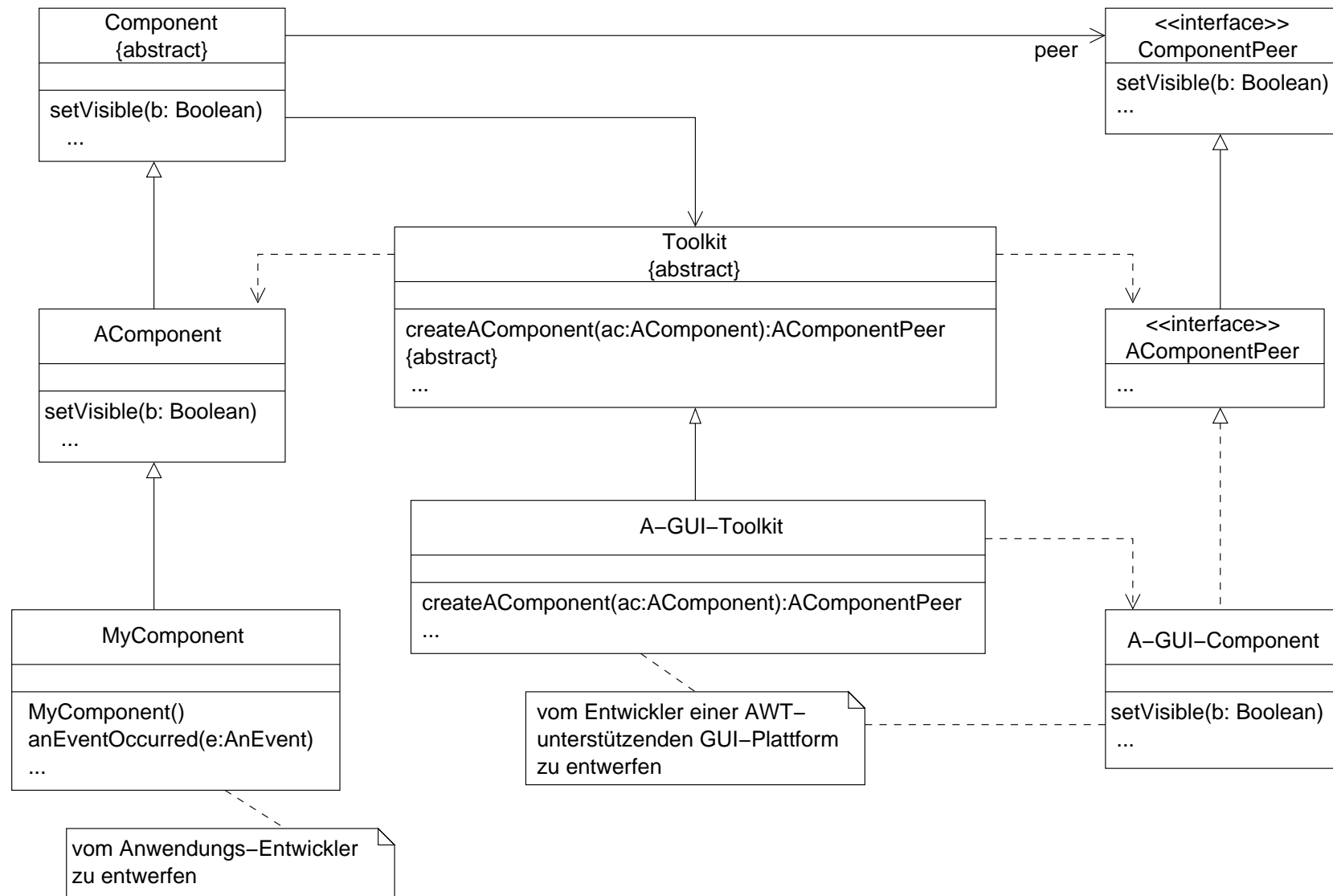
## 4.4 Entwurf von grafischen Benutzerschnittstellen

- 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 Ausprägungen für Java-Programme:
  - Swing/AWT ("abstract window toolkit"),
  - SWT ("standard widget toolkit").

## AWT (Abstract Window Toolkit) und Swing

- AWT und Swing bieten eine Klassenbibliothek zur Programmierung grafischer Benutzerschnittstellen (GUIs) für Java Programme (Pakete `java.awt`, `java.awt.event`, `javax.swing`)
- *Grundidee*: plattformunanabhängige Konstruktion von GUIs
- *Entwicklung*:
  - AWT 1.0
  - AWT 1.1 (neues Event-Handling mit “Listnern”)
  - Swing (ergänzt AWT und ersetzt die meisten AWT-Komponenten durch “Lightweight”-Komponenten)

# 1. Grundkonzepte von AWT



## Grundkonzepte von AWT (Zusammenfassung)

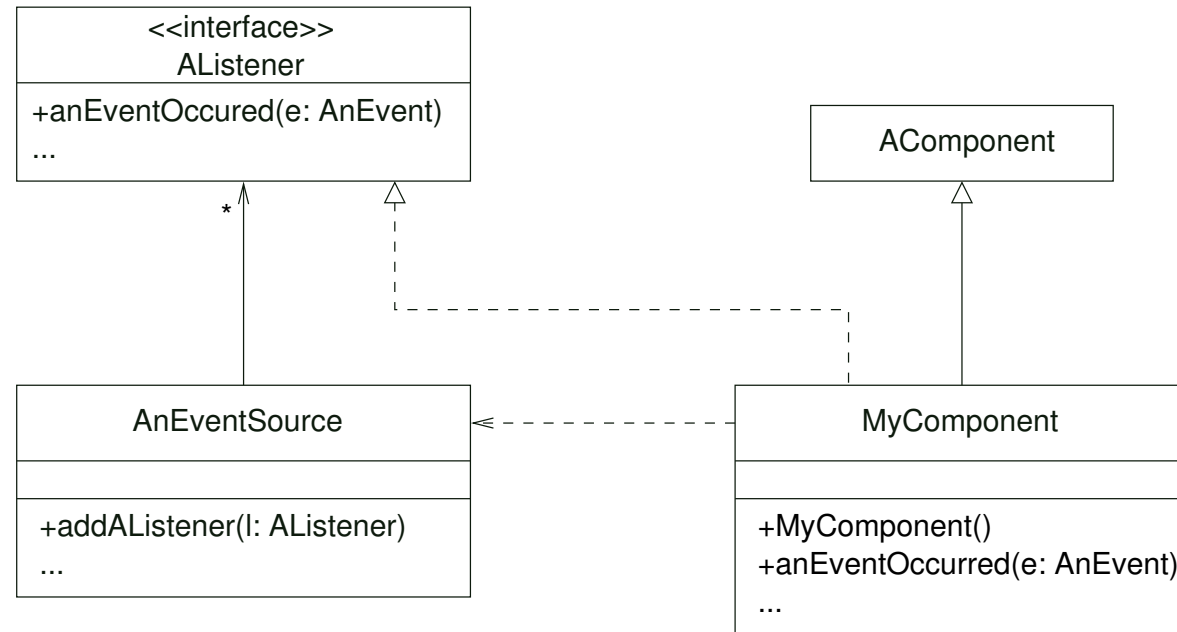
- AWT-Komponenten werden von einem Toolkit in entsprechende GUI-Komponenten einer speziellen Plattform (“native components”) übersetzt.
- Die plattformspezifischen GUI-Komponenten müssen ein entsprechendes Peer-Interface (des AWT) implementieren. Das Peer-Interface beschreibt die Anforderungen an die GUI-Komponente.
- Soll eine spezielle GUI-Plattform AWT unterstützen, dann muss ein entsprechendes GUI-Toolkit implementiert werden. Die abstrakte Klasse Toolkit (des AWT) beschreibt die Anforderungen an das GUI-Toolkit.
- Der Anwendungs-Entwickler muss die zur Anwendung gehörigen (plattformunabhängigen) GUI-Komponenten entwerfen (meist Swing Komponenten).



## Komponenten-Hierarchie (Zusammenfassung)

- Alle mit “J” beginnenden Klassen (und einige weitere) gehören zu Swing.
- In Swing werden unterschieden:
  - Heavyweight-Komponenten (JFrame, JDialog, JWindow, JApplet)
  - Lightweight-Komponenten (alle Spezialisierungen von JComponent)
- Heavyweight-Komponenten werden (wie AWT-Komponenten) in native Komponenten einer konkreten GUI-Plattform übersetzt.
- Heavyweight-Komponenten haben einen Container (Zugriff mit “getContentPane”), in dem die Lightweight-Komponenten gezeichnet werden.
- Jeder Container besitzt einen Layout-Manager.
- Mit “add” können neue Komponenten zu einem Container hinzugefügt werden (entsprechend des eingestellten Layout-Managers).

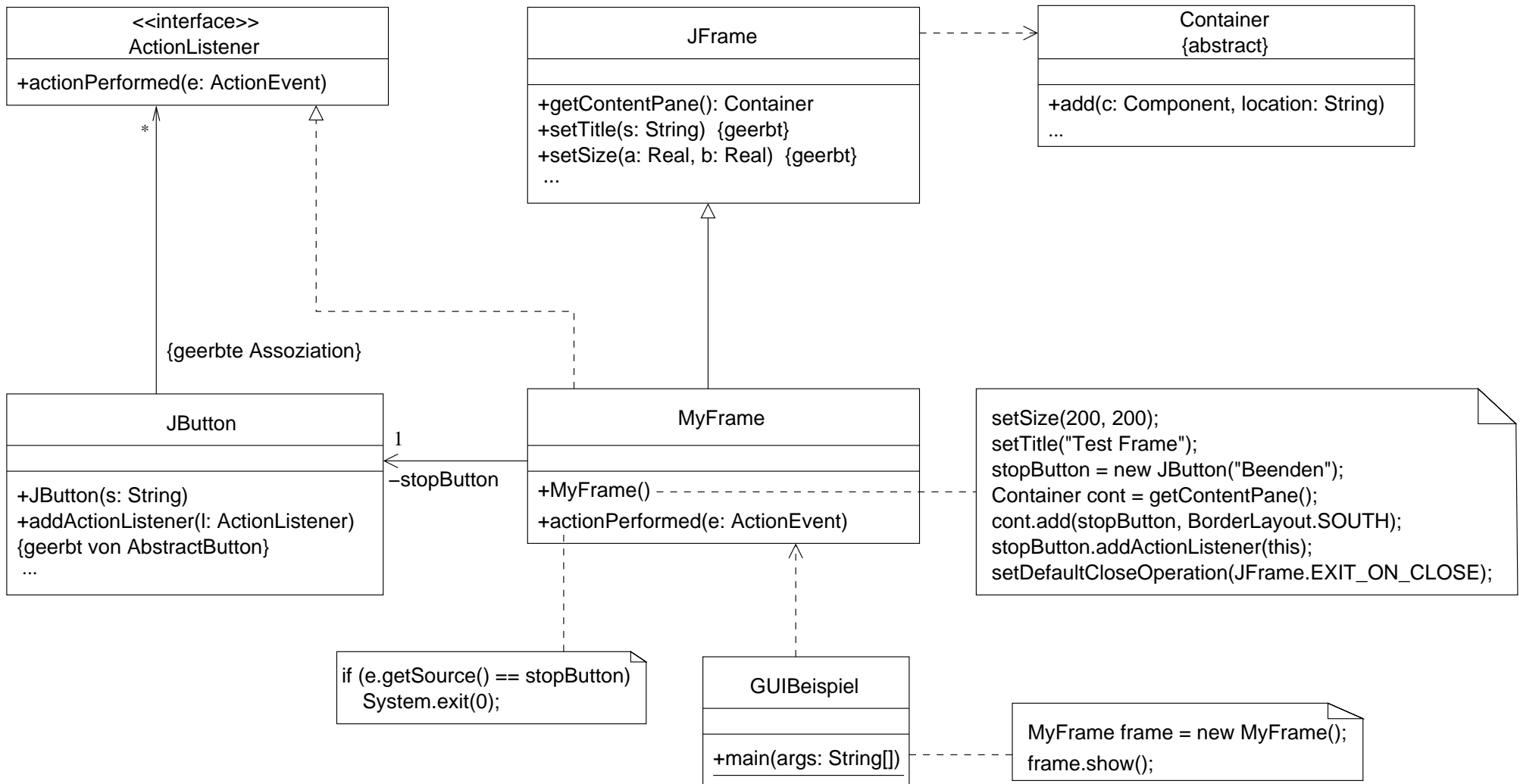
### 3. Ereignisbehandlung in AWT



## Ereignisbehandlung (Zusammenfassung)

- In AWT/Swing werden verschiedene Ereignisklassen unterschieden: KeyEvent, MouseEvent, ActionEvent, WindowEvent, ...
- Ist eine Komponente an Ereignissen eines bestimmten Typs interessiert, dann muss sie:
  1. sich bei der Komponente, in der ein solches Ereignis auftreten kann (AnEventSource) als "Listener" registrieren (addAListener).
  2. die beim Eintritt eines solchen Ereignisses aufgerufene Operation (anEventOccured) der passenden Listener-Schnittstelle (AListener) implementieren.
- Listener-Schnittstellen sind z.B. KeyListener, MouseListener, ActionListener, WindowListener.
- Operationen von Listener-Schnittstellen sind z.B. actionPerformed (von ActionListener), windowClosing (von WindowListener).

# 4. GUI-Modellierung: Ein einfaches Beispiel





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

class GUIBeispiel {

    public static void main (String[] args) {
        MyFrame frame = new MyFrame();
        frame.setVisible(true);
    }
}
```

```
class MyFrame extends JFrame implements ActionListener {

    private JButton stopButton;

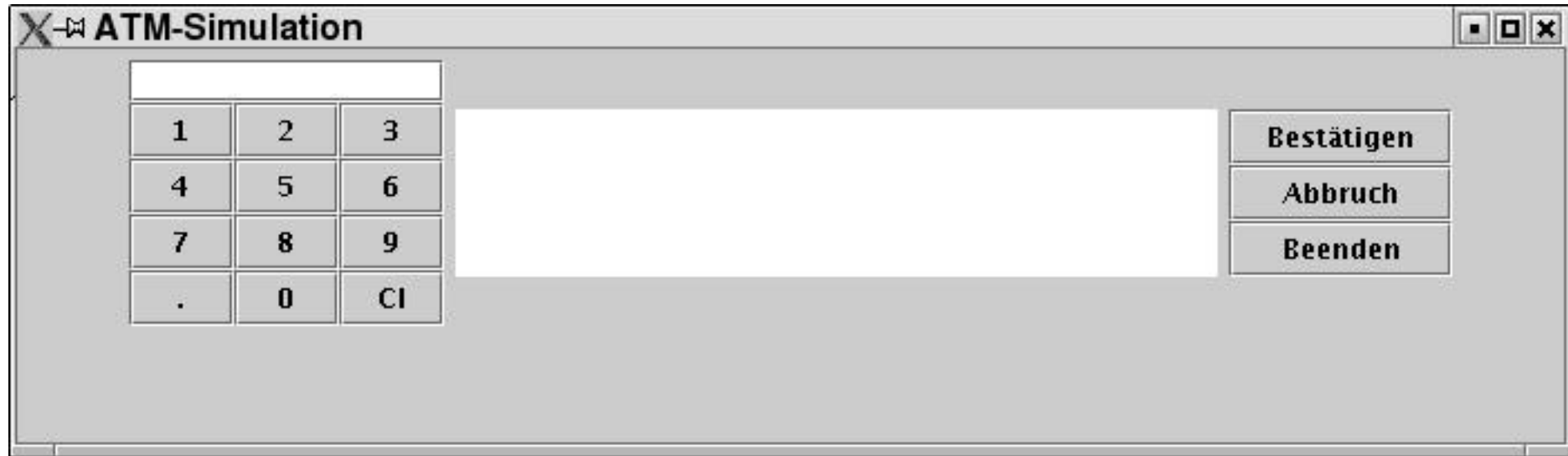
    public MyFrame() {
        setSize(200,200);
        setTitle("TestFrame");
        stopButton = new JButton("Beenden");

        Container cont = getContentPane();
        cont.add(stopButton, BorderLayout.SOUTH);

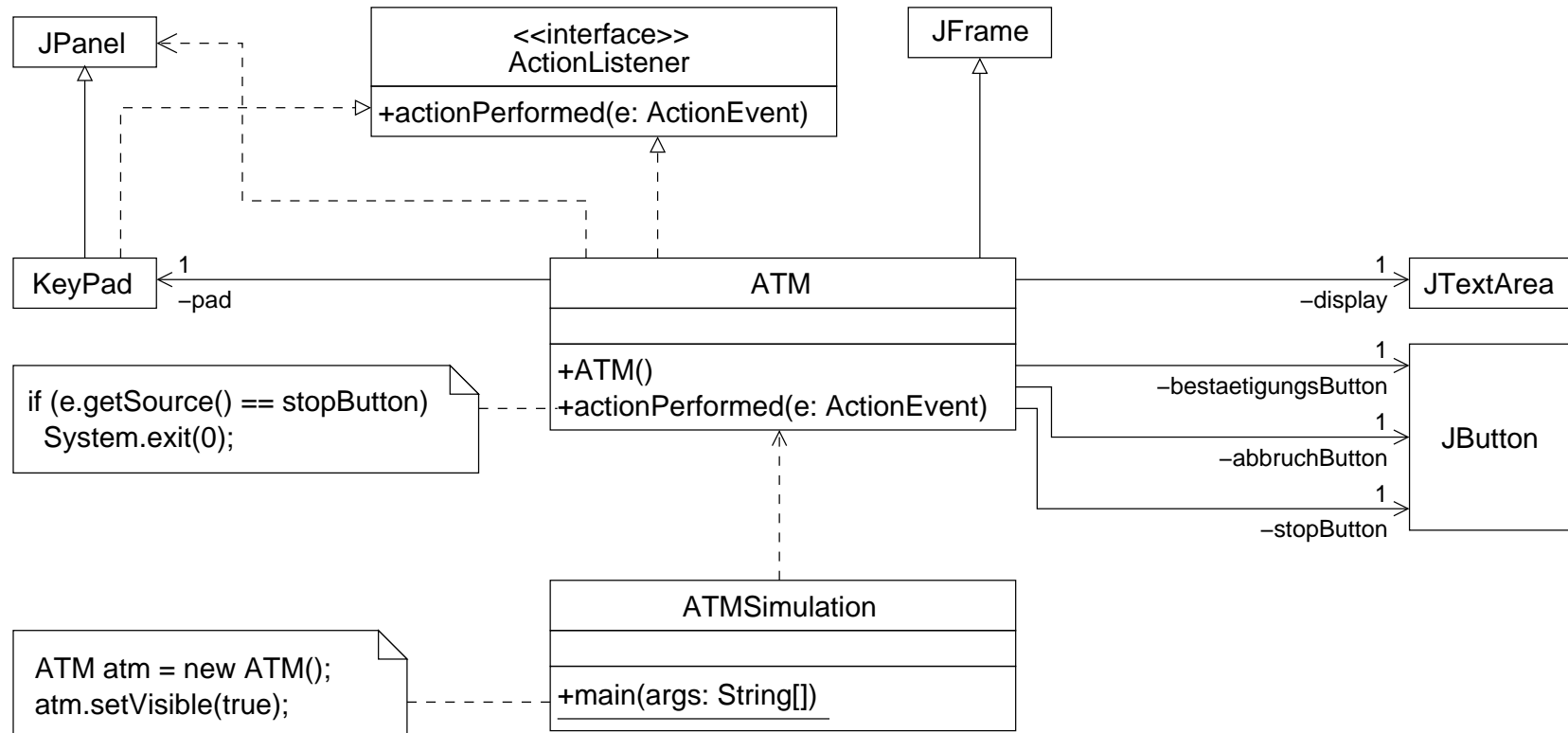
        stopButton.addActionListener(this);

        //Damit mit dem Schließen des Fensters auch das Programm beendet wird
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == stopButton)
            System.exit(0);
    }
}
```

## 5. 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.setVisible(true);
```

```
    }
```

```
}
```

```
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("Bestätigen");  
    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);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}

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

//angelehnt an "C. Horstmann: Computing Concepts with Java2 Essentials" (S. 601)

```
class KeyPad extends JPanel implements ActionListener {

    private JTextField display;

    public KeyPad() {
        setLayout(new BorderLayout());

        display = new JTextField();
        add(display, BorderLayout.NORTH);

        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(4, 3));
        addButton(buttonPanel, "1");
        addButton(buttonPanel, "2");
        addButton(buttonPanel, "3");
        addButton(buttonPanel, "4");
        addButton(buttonPanel, "5");
        addButton(buttonPanel, "6");
        addButton(buttonPanel, "7");
```

```
    addButton(buttonPanel, "8");
    addButton(buttonPanel, "9");
    addButton(buttonPanel, ".");
    addButton(buttonPanel, "0");
    addButton(buttonPanel, "C1");
    add(buttonPanel, BorderLayout.CENTER);
}

private void addButton(JPanel buttonPanel, String label) {
    JButton button = new JButton(label);
    buttonPanel.add(button);
    button.addActionListener(this);
}

public void actionPerformed (ActionEvent e) {
    JButton source = (JButton)e.getSource();
    String label = source.getText();
    if (label.equals("C1"))
        clear();
    else
```

```
        display.setText(display.getText()+label);
    }

    public double getValue() {
        return Double.parseDouble(display.getText());
    }

    public void clear() {
        display.setText("");
    }
}
```

## Zusammenfassung von Abschnitt 4.4

- Zur Programmierung von Benutzerschnittstellen verwendet man GUI-Toolkits.
- Anwendungsspezifische GUI-Elemente können durch Spezialisierung gegebener GUI-Klassen definiert werden.
- Swing/AWT bietet eine Klassenbibliothek zur plattformunabhängigen Programmierung von GUIs für Java Programme.
- Wesentliche Aufgaben bei der Realisierung einer GUI sind
  - die (statische) Konstruktion der GUI-Komponenten
  - die Programmierung der Ereignisbehandlung durch Implementierung entsprechender “Listener-Interfaces”.