Kapitel 1

Softwaretechnik: Überblick

Prof. Dr. Rolf Hennicker

20.10.2009

Ziele

- Verstehen, womit sich die Disziplin der Softwaretechnik (engl. Software Engineering) beschäftigt
- Qualitätskriterien von Software kennen
- Die wichtigsten Vorgehensmodelle in der Software-Entwicklung kennen
- Die Grundprinzipien der objektorientierten Software-Entwicklung verstehen

1.1 Einführung

Software = Menge von Programmen mit begleitenden Dokumenten (einschl. Modellen).

Systemsoftware zum Betrieb und zur Wartung von Rechensystemen (Betriebssysteme, spezielle Dienstprogramme wie Editoren, WWW-Browser, Compiler, Shell, ...)

Anwendungssoftware zur Lösung bestimmter, kundenspezifischer Aufgaben in

- Wirtschaft (z.B. Banken, Versicherungen, Reisebüros)
- Verwaltung (z.B. KFZ-, Uni-, Lagerverwaltung)
- Technik (z.B. Steuerungssysteme, eingebettete Systeme, Monitoring, naturw. und Umwelt-Simulationen)

Beachte: Die Erschliessung neuer Anwendungsbereiche und der Umfang von Software nimmt ständig zu.

Problem: Komplexität der Software, häufige Änderungen von Umgebungen und Anforderungen.

Großes Softwaresystem $\geq$ 50.000 lines of code ($\geq$ 10 PJ)
Sehr großes Softwaresystem $\geq$ 1 Mio lines of code ($\geq$ 200 PJ)

Beispiele:
Handy: 200.000 loc, System R/3: 7 Mio loc, Windows95: 10 Mio loc

Konsequenz: Mangelnde Software-Qualität, Fehler!!

Beispiele:
Handy bis zu 600 Fehler (dh. 3 Fehler pro 1000 loc),
Windows95 bis zu 200.000 Fehler (dh. 20 Fehler pro 1000 loc)
Space Shuttle weniger als 1 Fehler pro 10.000 Zeilen.

Man spricht von
normaler SW bei 25 Fehlern pro 1000 loc
guter SW bei 2 Fehlern pro 1000 loc

Historisches:

- 1965 Softwarekrise:
  Umfang der SW für komplexe Anwendungen wird nicht mehr beherrscht,
  Methoden des "Programmierens im Kleinen"greifen nicht mehr
- 1968/69:
  Der Begriff "Software Engineering"wird geprägt
  Idee: Anwendung ingenieurmäßiger Methoden bei der SW-Entwicklung

Definition (nach Fairley 1985):
Software Engineering ist die technische und organisatorische Disziplin zur systematischen Herstellung und Wartung von Softwareprodukten, die zeitgerecht und innerhalb vorgegebener Kostenschranken hergestellt und modifiziert werden.

Ziele des SW Engineering

- Software hoher Qualität
  (aus Sicht des Benutzers und des Entwicklers)
- Beherrschung der Kosten und der Entwicklungszeit
  (aus wirtschaftlicher Sicht)

Bereiche des Software Engineering

- **Software-Entwicklung**
  - Definition von Anforderungen
  - Entwurf von Lösungen
  - Implementierung und Wartung
  - Erstellen von Modellen und Dokumenten
- **Qualitätssicherung**
  - Analytische Maßnnahmen wie Codeinspektion, Testen, Validieren, Verifizieren
  - Konstruktive Maßnahmen wie Einhaltung methodischer Richtlinien, Wiederverwendung qualitativ hochwertiger Komponenten
- **Projektmanagement**
  - Projektplanung (Projektbeschreibung, Zeitplan mit Meilensteinen, Kosten- und Personalplanung)
  - Projektkoordination (nach innen und nach aussen)
  - Projektkontrolle (Bewertungen, Risikoabschätzung, Berichte)

Zur Lösung dieser Aufgaben verwendet man

- **Vorgehensmodelle** (Aus welchen Phasen besteht der SW-Lifecycle?)
- **Techniken** (Notationen)
  zur konkreten Beschreibung und Lösung einzelner Aufgaben (z.B. UML, Java)
- **Methoden**
  zum systematischen Einsatz der Techniken
- **Werkzeuge** (CASE-Tools)
  zur Projektplanung, zum Editieren von Programmen und Grafiken, zur Codegenerierung, zur Compilierung und Interpretation, zur Spezifikations- und Programmanalyse, zum Testen, zum Konfigurationsmanagement

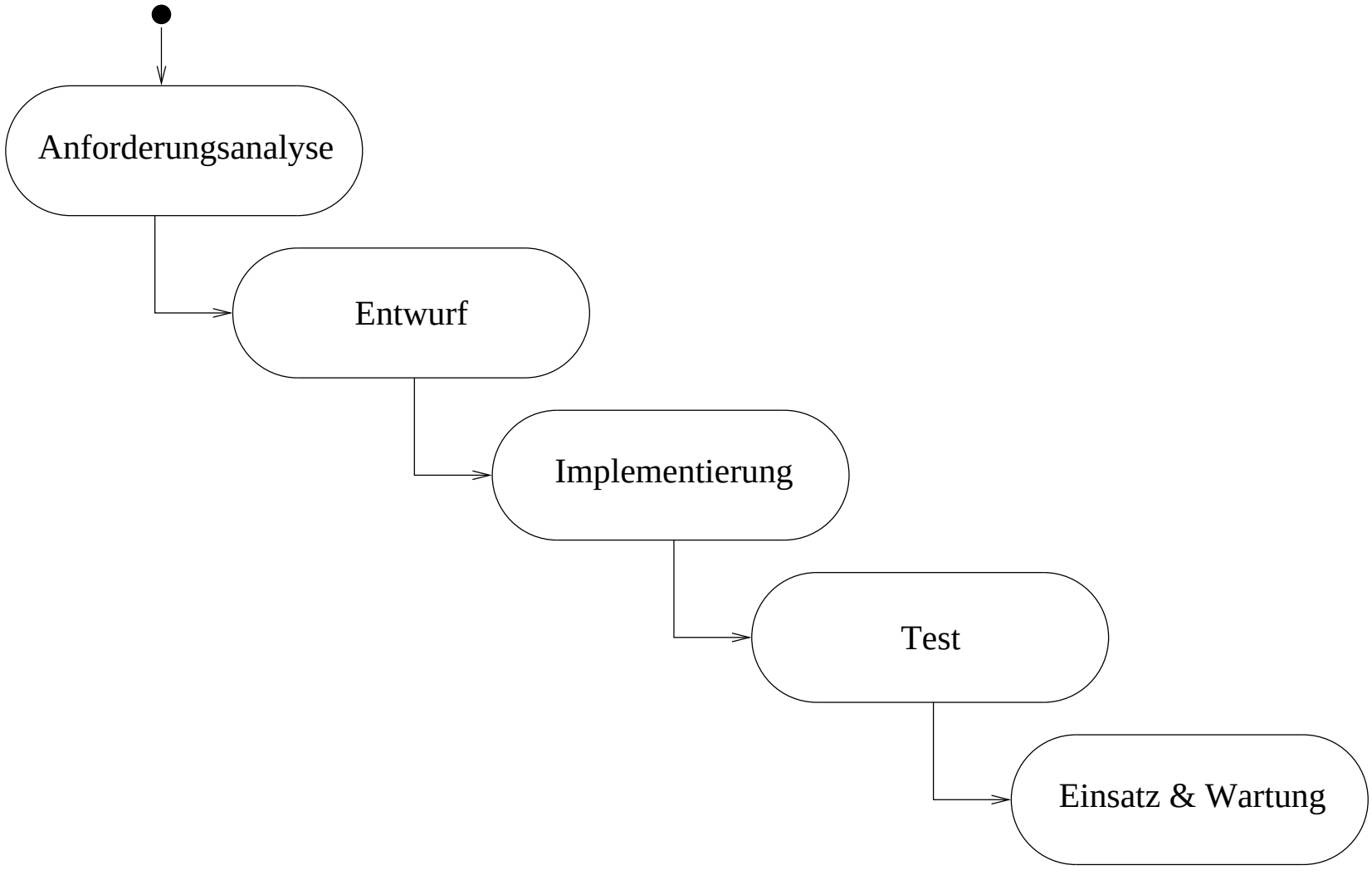
1.2 Qualitätskriterien von Software

1. **Korrektheit:**
   Ein Softwareprodukt erfüllt die in einer Spezifikation beschriebenen Anforderungen
   Beachte:
   - Der Nachweis der Korrektheit ist nur mit formalen Methoden möglich.
   - Falls die (formale) Spezifikation nicht die Anforderungen des Anwenders wiedergibt, leistet auch korrekte Software nicht das Gewünschte.
2. **Zuverlässigkeit:**
   Wahrscheinlichkeit, dass ein SW-Produkt eine gewünschte Funktion in einer bestimmten Zeit erfüllt. Hohe Zuverlässigkeit bedeutet, dass Fehler selten auftreten und nur geringe Auswirkungen haben.
3. **Robustheit:**
   Sinnvolle Reaktion bei Fehlern, die in der Umgebung auftreten (z.B. Bedienungsfehler, Fehler anderer Systeme)

4. **Benutzerfreundlichkeit:**
   Einfache Erlernbarkeit und Bedienbarkeit eines SW-Systems (Übersichtliche und adäquate Benutzerschnittstelle, Lernprogramme, Hilfen, verständliche Fehlermeldungen).
5. **Wartbarkeit:**
   - Möglichst einfache Lokalisierung und Behebung von Fehlern.
   - Möglichst einfache **Änderbarkeit** und **Erweiterbarkeit**.
6. **Performanz:**
   Angemessenes Laufzeitverhalten und Speicherplatzbedarf
7. **Dokumentation:**
   Sowohl für den Benutzer (Handbücher) als auch für den (Weiter-)Entwickler (Dokumentation der einzelnen Entwicklungsschritte, der Implementierung, von Testplänen, ...).
8. **Portabilität:**
   Übertragbarkeit der Software auf andere Rechner.

1.3 Vorgehensmodelle

Das Wasserfallmodell



Anforderungsanalyse (engl. requirements analysis):

- Analyse des Problembereichs
- Festlegung der (funktionalen und nicht funktionalen) Anforderungen an das System
  Was *soll das System leisten?*
- Festlegung organisatorischer Richtlinien (Aufwandsabschätzung, Terminplanung,...) und von Rahmenbedingungen (z.B. vorhandene Soft- und Hardware).
- Machbarkeitsstudie
- Skizze der Systemarchitektur

Ergebnis der Anforderungsanalyse ist eine Anforderungsbeschreibung (Spezifikation, Modell). Diese dient

- als "Vertrag"zwischen Anwender und SW-Entwickler und
- als Grundlage für die weitere Systementwicklung.

Problem: Mögliche Missverständnisse zwischen Anwender und Entwickler.

Entwurf (engl. design):

- Beschreibt die Art und Weise, in der die gestellten Aufgaben gelöst werden sollen.
  Wie *lösen wir das Problem?*
- Festlegung der Systemarchitektur
- Entwurf der einzelnen Systemkomponenten (Wahl von Datenrepräsentationen und Algorithmen)

Ergebnis der Entwurfsphase ist eine (konstruktive) Entwurfsbeschreibung (Spezifikation, Modell).

Implementierung:
Codierung des Entwurfs in einer Programmiersprache, ggf unter Wiederverwendung vorhandener Komponenten.

Test:

- Test der einzelnen Komponenten ("unit testing")
- Schrittweises Zusammenfügen einzelner Komponenten mit jeweiligem Integrationstest
- Systemtest
- Abnahmetest (mit "echten"Daten des Anwenders)

Wartung:

- Fehlerbeseitigung nach Inbetriebnahme
- Änderung und Erweiterung des Systems

Schätzung der aktuellen Kosten in der Systementwicklung

Anforderungsanalyse	6%
Entwurf	5%
Implementierung	7%
Integration und Test	15%
Wartung	67% (davon 40% um existierende Programme zu verstehen!)

Bemerkungen zum Wasserfallmodell

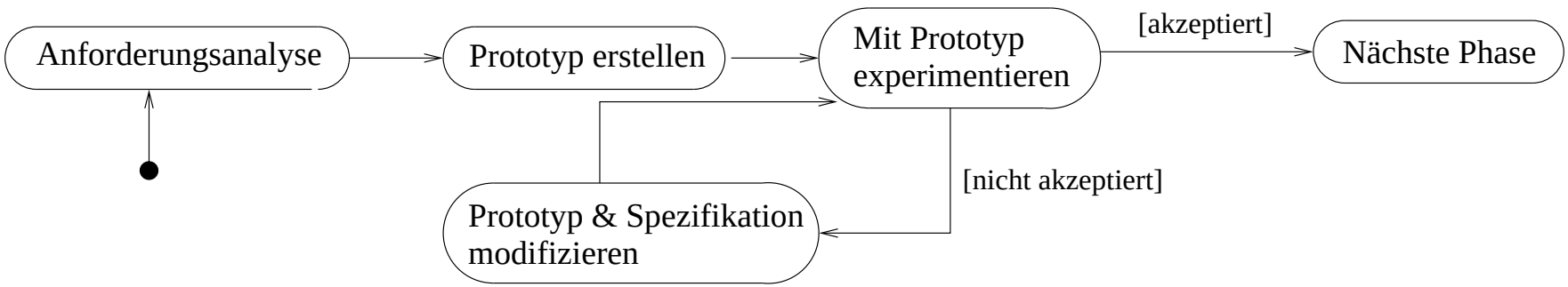
- Klare Trennung der verschiedene Phasen
- Schwierigkeiten in einer Phase verzögern das Gesamtprojekt
- Validierung des Produkts durch den Kunden ist erst nach Abschluss der gesamten Entwicklung möglich
- Streng sequentielle Vorgehensweise ist in der Praxis kaum möglich, da
  - Fehler aus früheren Phasen häufig erst später erkannt werden
  - Anforderungen sich ändern können
- Häufig werden einzelne Phasen weiter verfeinert, z.B. Grob- und Feinentwurf

Das Prototyp-orientierte Modell

Prototyp = Vorabversion (von Teilen) des intendierten Systems.

Charakteristika von Prototypen:

- schnelle und billige Herstellung
- häufig zur Demonstration von Benutzeroberflächen
- eingeschränkte Funktionalität, nicht robust, schlechte Performanz

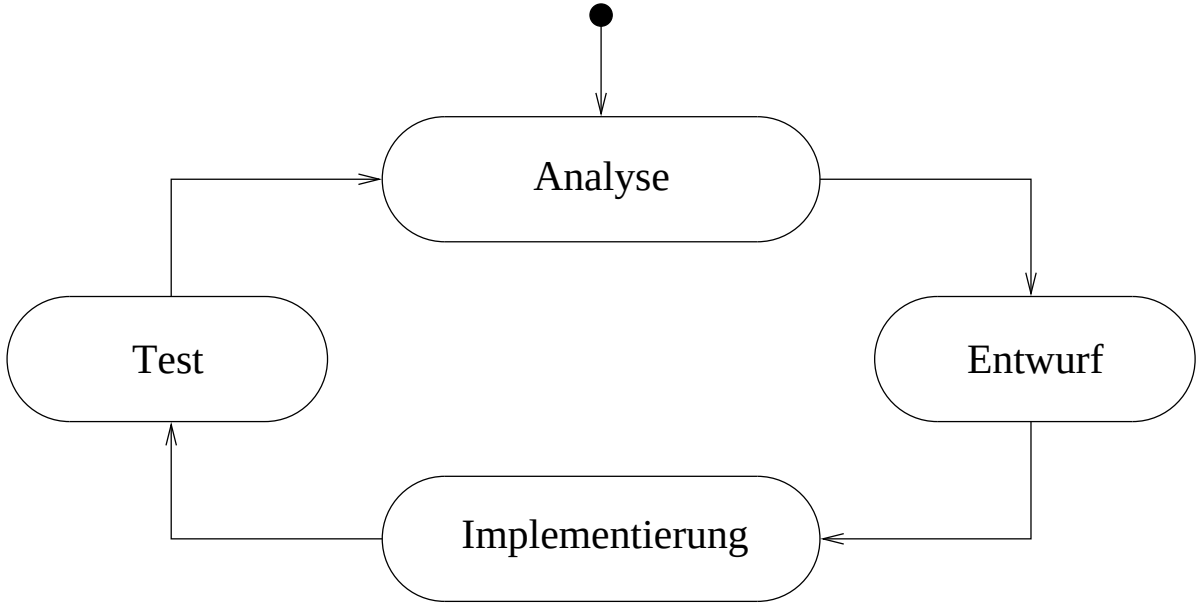


Vorteil: Frühzeitige Validierung durch den Kunden

Unterscheide: Wegwerf-Prototypen und Prototypen zur Weiterentwicklung

Iteratives Vorgehensmodell

Der Entwicklungsprozess besteht aus einer Folge von Zyklen (Iterationen).



Am Ende jedes Zyklus steht eine neue (ausführbare) Version des SW-Produktes, die die vorherige verbessert und erweitert.

Wartung = weiterer Zyklus zur Erstellung eines verbesserten Produkts nach Inbetriebnahme.

Vorteil: Häufiger Kontakt zum Anwender jeweils bei der Erstellung einer neuen Version ("evolutionäres Prototyping")

Nachteil: Risiko, dass die Architektur der bereits implementierten Teile nicht zu den neu hinzukommenden Anforderungen passt.

Spezielle Ausprägung: Unified Process (UP) nach Jacobson, Booch und Rumbaugh (1999).

Charakteristika des Unified Process:

- Anwendungsfall gesteuert ("Use Case driven")
- Architektur-zentriert
- Iterativ, wobei jede Iteration aus den Arbeitsschritten "Anforderung", "Analyse", "Entwurf", "Implementierung"und "Test"besteht.
- Verschiedene Iterationen werden in einzelnen Phasen zusammengefasst, wobei es die 4 Phasen "Beginn", "Ausarbeitung", "Konstruktion"und "Umsetzung"gibt.

Beginn (Inception): Eine "Vision" des Systems wird entwickelt

- Bestimmung der wichtigsten Anwendungsfälle
- Grobe Skizze der Systemarchitektur

Ausarbeitung (Elaboration):

- Genaue Beschreibung der meisten Anwendungsfälle
- Realisierung von essentiellen Anwendungsfällen
- Ausarbeitung der Systemarchitektur

Konstruktion (Construction):

- Implementierung aller Anwendungsfälle
- Stabilisierung der Systemarchitektur

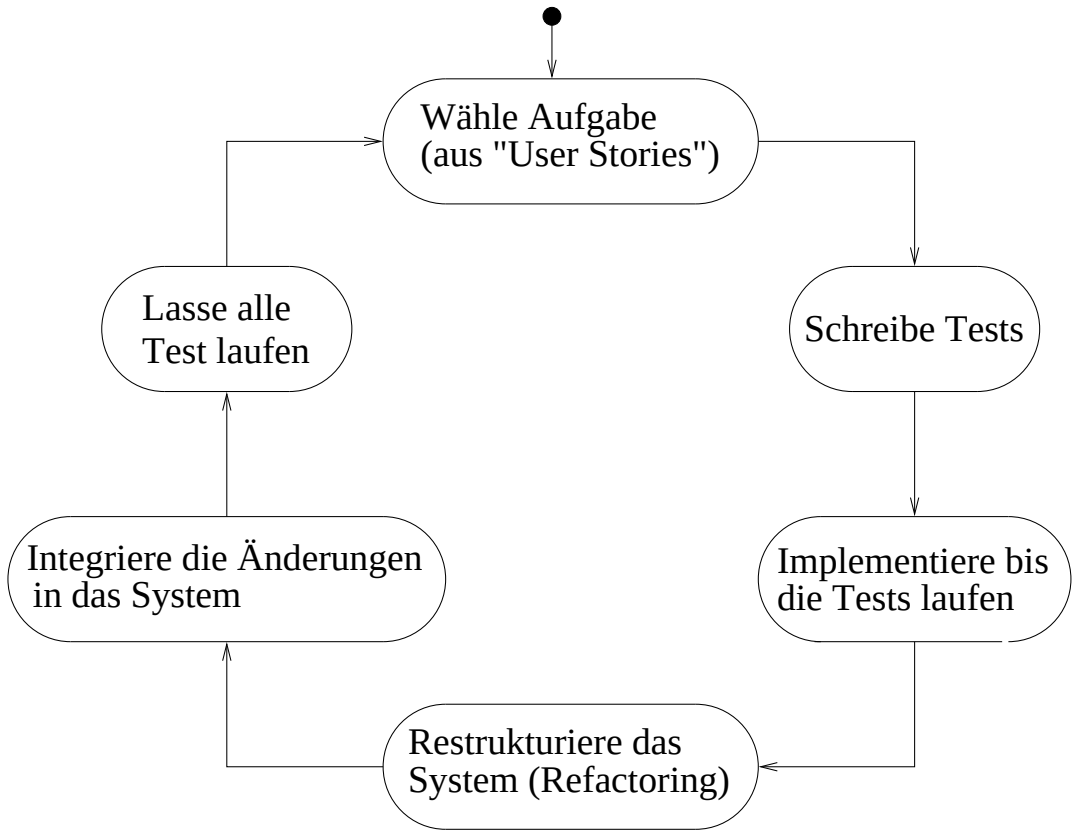
Umsetzung (Transition):

- Erstellen einer β-Version
- Testen und Verbessern
- Übergabe an den Kunden

XP: eXtreme Programming

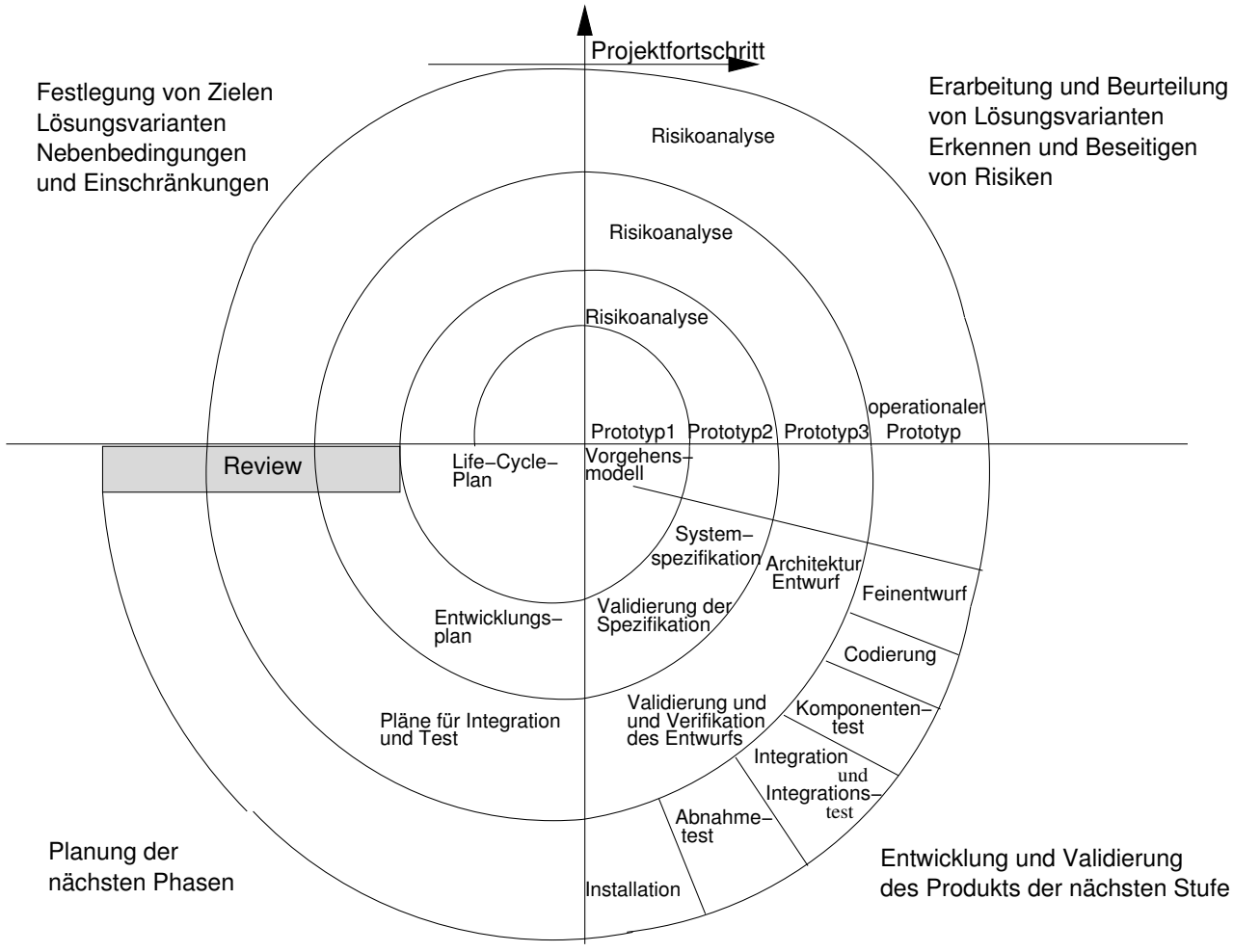
Charakteristika

- Test-getrieben
- viel Kommunikation (paarweises Programmieren, schnelle Rückmeldung durch den Anwender)
- Einfachheit (implementiere das Einfachste, das funktioniert)
- inkrementelle Erweiterung (iterativ)



Das Spiralmodell (nach Boehm, 1988)

Grundidee: Risikobeherrschung



1.4 Grundprinzipien der objektorientierten SW-Entwicklung

- Ein System besteht aus einer Menge von Objekten, die sich (gegenseitig) Nachrichten schicken. Der Empfang einer Nachricht löst eine Operation des (empfangenden) Objekts aus.
- Ein Objekt ist ein individuelles Exemplar mit einer eindeutigen Identität.
- Eine Klasse beschreibt eine Menge von Objekten mit gemeinsamen Merkmalen.
- Merkmale von Objekten sind:
  - Attribute (z.B. Name, Alter einer Person, ...)
  - Operationen, die jedes Objekt der Klasse ausführen kann (z.B. radfahren, schreiben, ...)
  - (mögliche) Beziehungen zu anderen Objekten (z.B. Auto gehört einer Person)
- Die aktuellen Attributwerte (und Beziehungen) zu einem Zeitpunkt bestimmen den Objektzustand.
- Die aktuellen Zustände aller zu einem Zeitpunkt existierenden Objekte (und deren Beziehungen zu anderen Objekten) bestimmen den Systemzustand.
- Klassen können spezialisiert werden (z.B. Auto ist ein spezielles Fahrzeug) (*Vererbungsprinzip!*)

Vorteile der objektorientierten SW-Entwicklung

- Objekte der "realen Welt" können auf ein objektorientiertes Modell und ein objektorientiertes System abgebildet werden.
- Objektorientierte Techniken können auf verschiedenen Ebenen der Systementwicklung eingesetzt werden.

  $$\text{OO-Analyse} \rightarrow \text{OO-Entwurf} \rightarrow \text{OO-Programm}$$

- Einfache Erweiterbarkeit und Wiederverwendbarkeit (z.B. durch Hinzunahme von Subklassen durch Vererbung).

Zusammenfassung

- Ziele des SW Engineerings sind
  - die Erstellung von Software hoher Qualität
  - die Beherrschung der Kosten und der Entwicklungszeit
- Bereiche des SW Engineerings sind SW-Entwicklung, Qualitätssicherung, Projektmanagement.
- In der SW-Entwicklung verwenden wir Vorgehensmodelle, Methoden, Techniken und Werkzeuge.
- Qualitätskriterien von SW sind Korrektheit, Zuverlässigkeit, Robustheit, Benutzerfreundlichkeit, Wartbarkeit, Effizienz, Dokumentation, Portabilität.
- Bekannte Vorgehensmodelle sind Wasserfallmodell, Prototyp-orientiertes Modell, Iteratives Modell (z.B. UP), XP (eXtreme Programming), Spiralmodell.
- Ein objektorientiertes System besteht aus einer Menge von Objekten, die Nachrichten austauschen und auf den Empfang von Nachrichten reagieren (z.B. durch Änderung des Objektzutands).

Kapitel 2

Objektorientierte Modellierungtechniken

Prof. Dr. Rolf Hennicker

27.10.2009

Ziele

- Klassendiagramme in UML erstellen können.
- Objektdiagramme in UML erstellen können.
- Das Vererbungs- und Subtypprinzip verstehen, insbesondere
  - abstrakte Klassen und Operationen
  - Schnittstellen
  - dynamische Bindung
- Klassendiagramme in Java implementieren können.
- (Flache und hierarchische) Zustandsdiagramme in UML erstellen können.
- Zustände, Ereignisse und Transitionen verstehen.
- Aktivitätsdiagramme in UML erstellen können.
- Das Prinzip der Metamodellierung verstehen.

Grundidee

Modellierung dient dazu, ein System zu verstehen, bevor es gebaut wird.

Wesentliches Prinzip

Abstraktion (auf wesentliche Aspekte konzentrieren, keine Details)

Es werden i.a. verschiedene Sichten auf ein System modelliert:

- *Statisches Modell*: beschreibt strukturelle und datenbezogene Eigenschaften
- *Dynamisches Modell*: beschreibt das Verhalten der Objekte, deren Zustandsänderungen und Interaktionen.

Notation

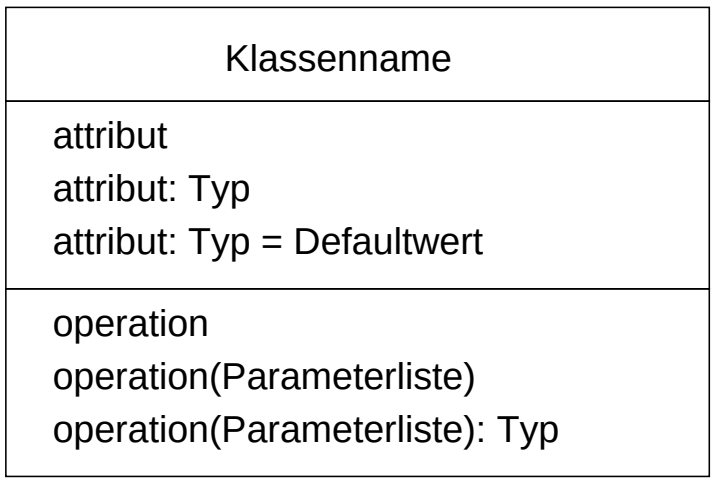
UML (Unified Modeling Language), 1997 Version 1.0 (Booch, Rumbaugh, Jacobson), aktuell UML 2

2.1 Modellierung statischer Systemeigenschaften

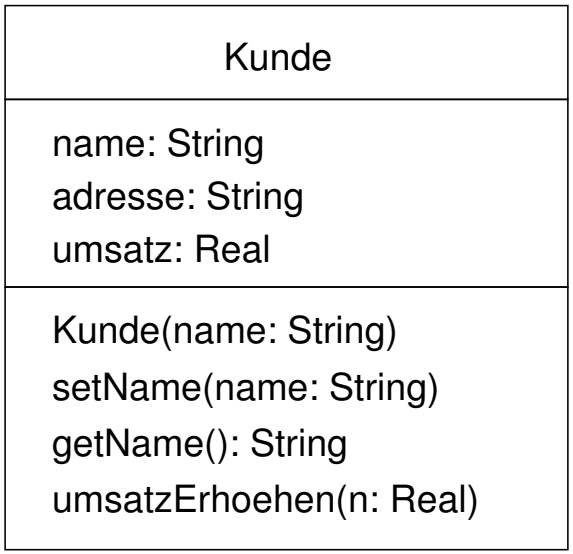
2.1.1 Klassen und Objekte

Klassen

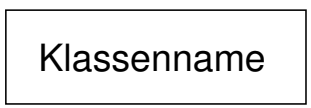
Allgemeine Form:



Beispiel:

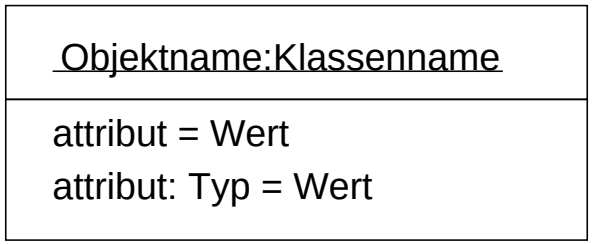


Kurzform:

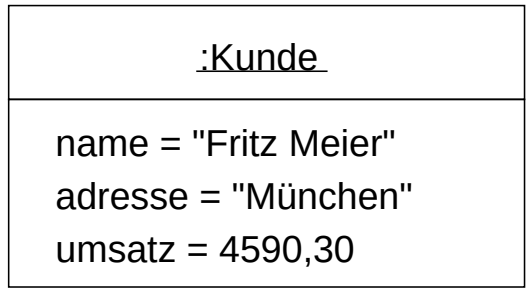


Objekte

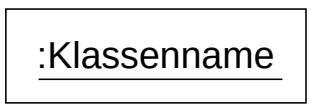
Allgemeine Form:



Beispiel:



Kurzformen:



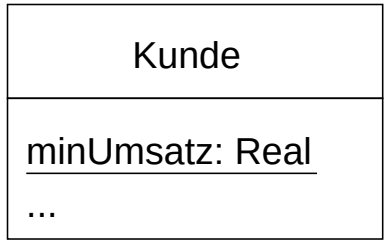
Bemerkungen

- Als Typen verwenden wir Standarddatentypen (Boolean, Integer, Real, String) oder Klassennamen.
- In der Analysephase (und möglichst auch im Entwurf) sollten als Typen von Attributen nur Standarddatentypen verwendet werden.
- Operationen mit Ergebnistyp liefern nach Ausführung einen Wert dieses Typs. Der Rückgabewert kann auch ein Objekt sein.
- Ebenso können aktuelle Parameter von Operationen Objekte sein, wenn der formale Parameter einen Klassentyp hat.
- Operationen ändern i.a. den Zustand eines Objekts.
- Operationen, die den Zustand nicht ändern, nennt man "Queries".

Weiterführende Begriffe und Notationen

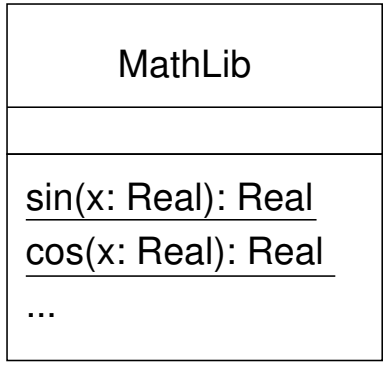
Ein Attribut, das bei jedem existierenden Objekt der Klasse denselben Wert hat, heißt *Klassenattribut* und wird in UML unterstrichen.

Beispiel:



Eine Operation, die vom Zustand konkreter Objekte unabhängig ist, heißt *Klassenmethode* und wird in UML unterstrichen.

Beispiel:

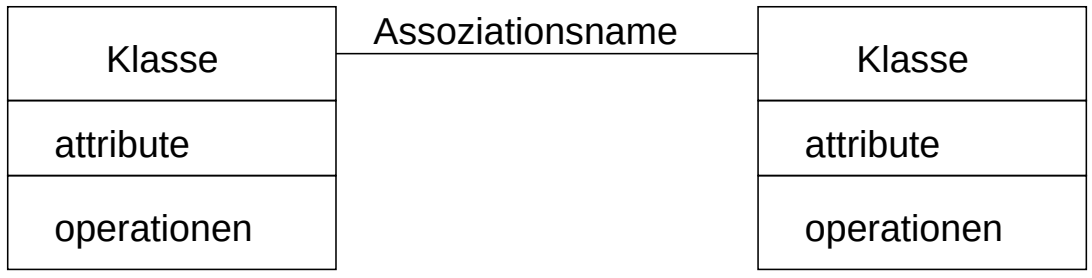


2.1.2 Assoziationen und Objektbeziehungen

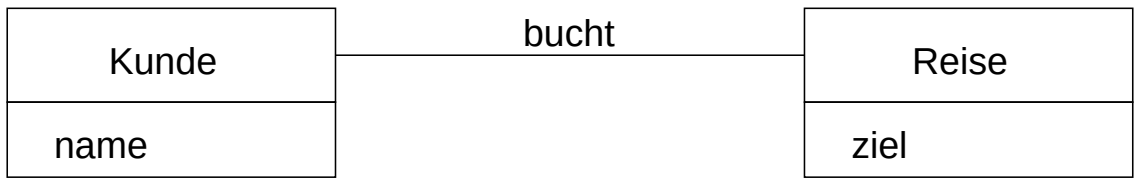
- Eine *Objektbeziehung (Link)* stellt eine semantische (physikalische oder konzeptionelle) Verbindung zwischen Objekten dar.
- Eine *Assoziation* beschreibt eine Menge gleichartiger Beziehungen zwischen Objekten bestimmter Klassen.

Assoziationen

Allgemeine Form:

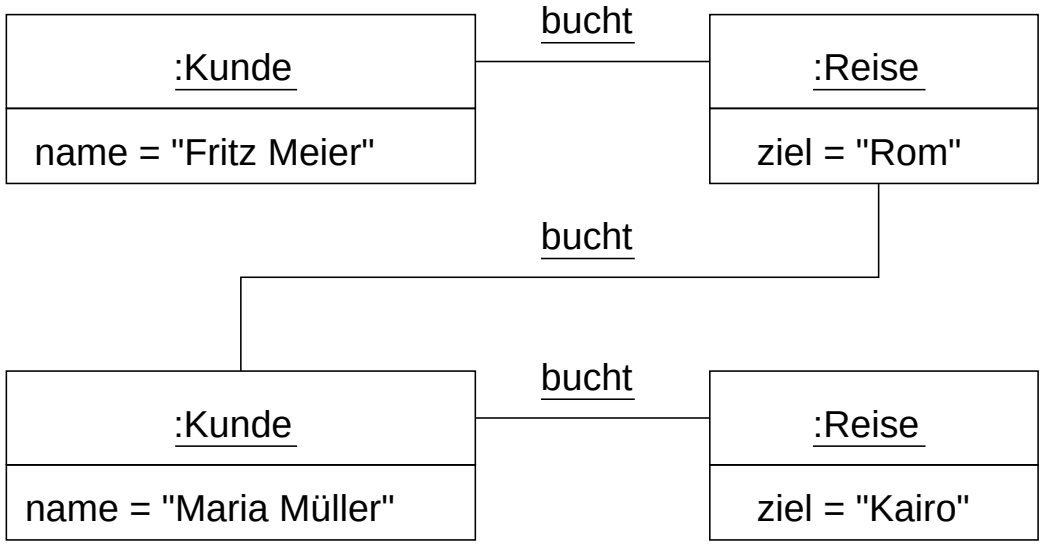


Beispiel:



Objektbeziehungen

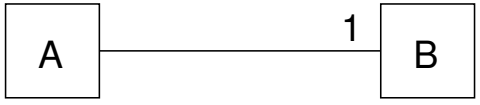
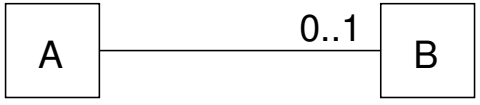
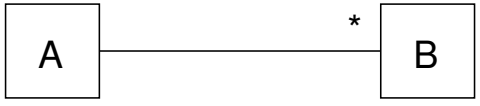
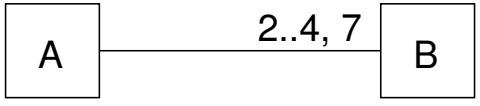
Beispiel:



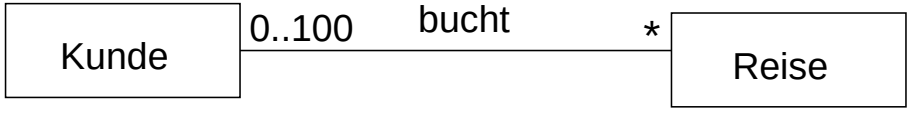
- Ein UML-Diagramm mit Klassen und Assoziationen (und Vererbung) heißt *Klassendiagramm*.
- Ein UML-Diagramm mit Objekten und Objektbeziehungen heißt *Objektdiagramm* (oder *Instanzendiagramm*). Es stellt einen augenblicklichen Systemzustand ("Snapshot") dar.

Multiplizitäten

Geben an, wieviele Objekte einer Klasse mit wievielen Objekten einer (meist anderen) Klasse gemäß einer Assoziation in Beziehung stehen können.

*N*otation:	Bedeutung:
	Jedes Objekt von A steht in Beziehung mit
	genau einem Objekt von B
	keinem oder einem Objekt von B
	einem oder mehreren Objekten von B
	beliebig vielen Objekten von B (auch keinem)
	2 bis 4 oder 7 Objekten von B

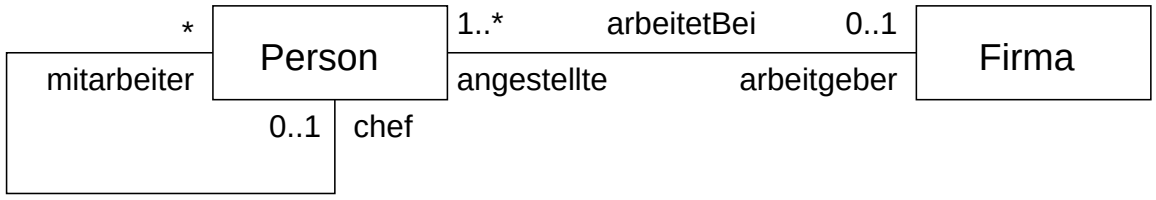
Beispiel:



Assoziationsrollen

Beschreiben, welche Rolle die Objekte einer Klasse in einer Assoziation einnehmen.

Beispiel:



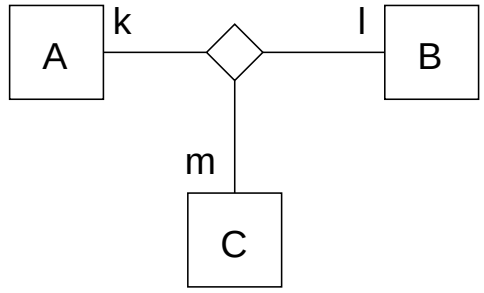
Bemerkung:

Assoziationsnamen und Rollennamen können weggelassen werden. Rollennamen sind dann implizit durch die kleingeschriebenen Klassennamen (evtl. im Plural) gegeben.

Mehrstellige Assoziationen

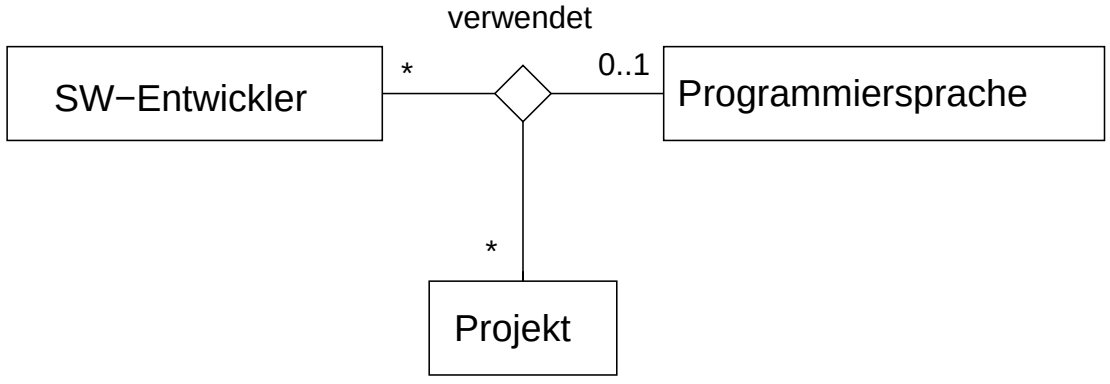
Verbinden drei oder mehr Klassen.

Darstellung:



Die Multiplizität einer Rolle in einer n-stelligen Assoziation spezifiziert, wieviele Objekte mit dieser Rolle mit fest gegebenen (fixierten) n-1 Objekten der anderen Klassen in Beziehung stehen können.

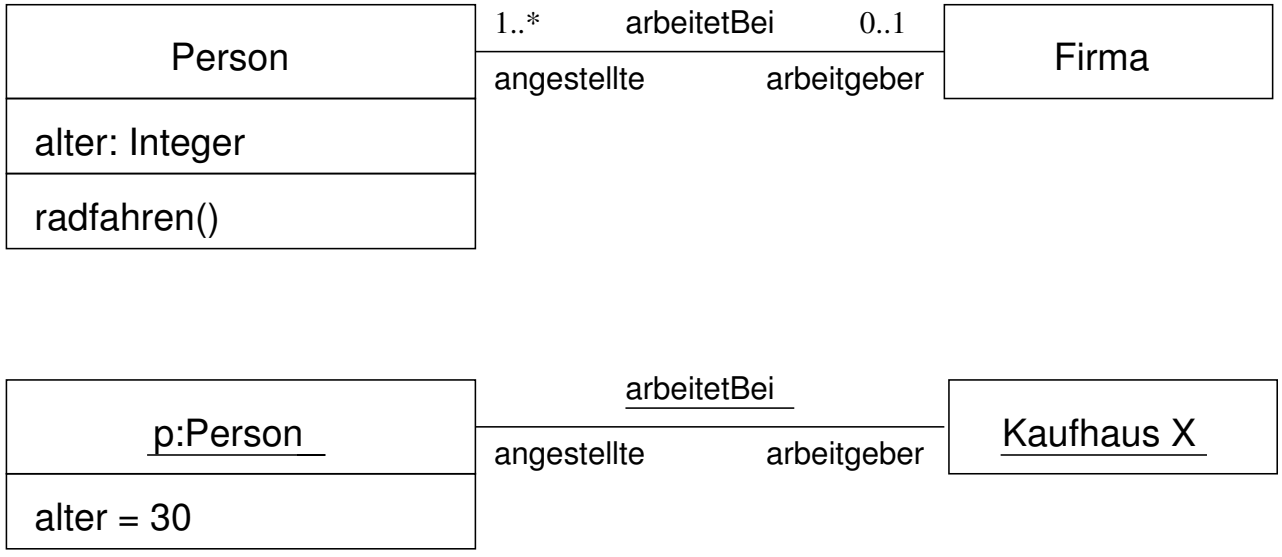
Beispiel:



Zugriff auf die Merkmale eines Objekts

Wird durch die "."-Notation ausgedrückt.

Beispiel:



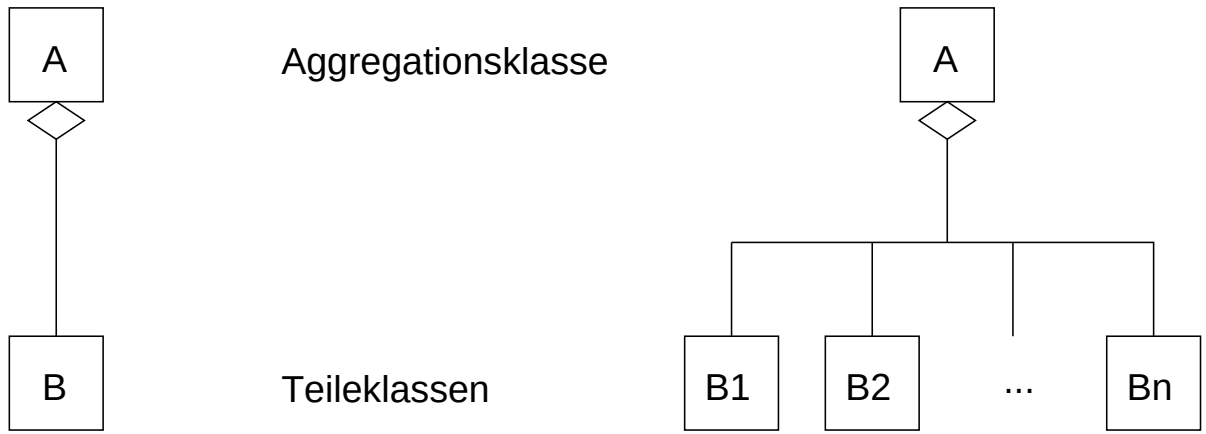

```
p.alter = 30;
p.arbeitgeber = Kaufhaus X;
p.radfahren(); //Aufruf der Operation "radfahren" fuer das Objekt p
```

Aggregation

Spezielle Form der Assoziation, die eine "Gesamtheit-Teil"-Beziehung ausdrückt.

z.B.
ICE-Lok besitzt 6 Motoren (physikalisch),
Stundenplan umfasst mehrere Vorlesungen (konzeptionell)

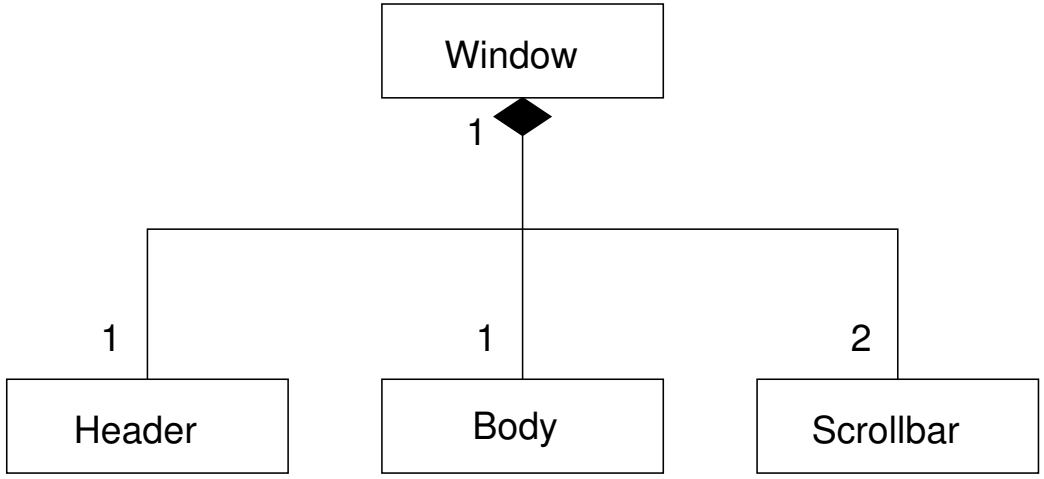
Darstellung:



Komposition

Spezielle Form der Assoziation: das Teil ist existenzabhängig vom Ganzen.

Darstellung (Beispiel):

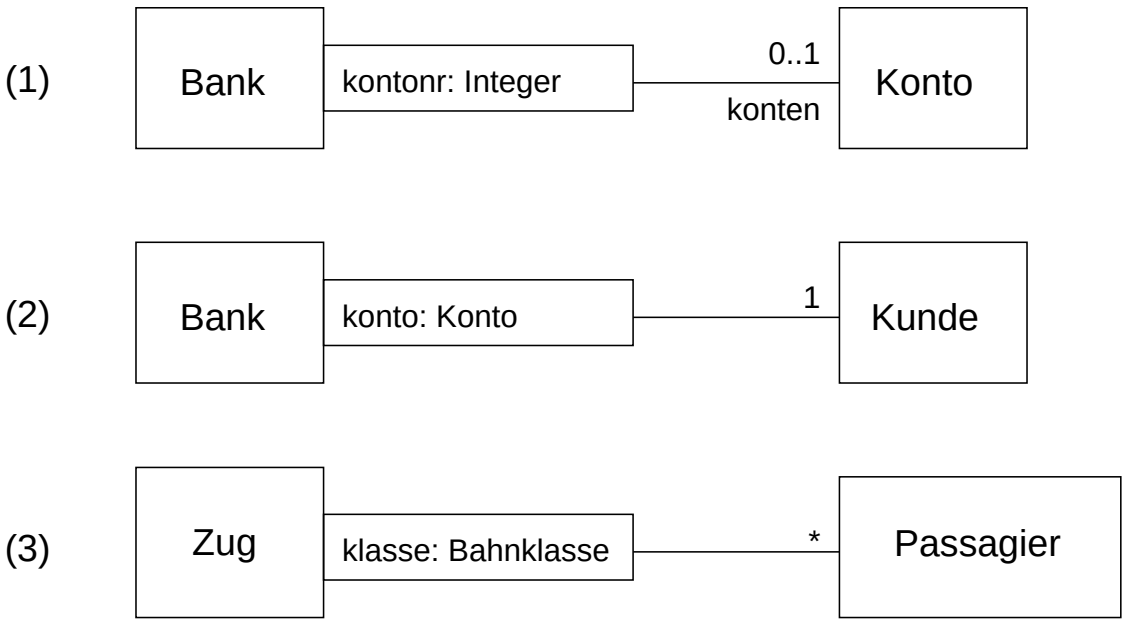


Das Teil ist vollständig im Besitz des Ganzen [OMG 2004]. Das Teil kann erst bei (oder nach) der Erzeugung des Ganzen erzeugt werden und wird mit dem Ganzen gelöscht.

Qualifizierte Assoziation

- Eine qualifizierte Assoziation unterteilt die Menge der Objekte auf einer Seite der Assoziation in Partitionen.
- Qualifizierer haben (wie Attribute) einen Typ, der auch eine Klasse sein kann.

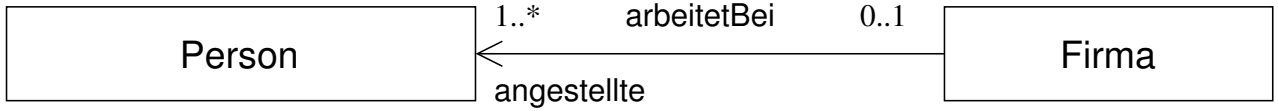
Beispiele:



Gerichtete Assoziation

Falls eine Assoziation nur in einer Richtung durchlaufen wird ("unidirektional"), wird das entsprechende Assoziationsende mit einer Pfeilspitze markiert.

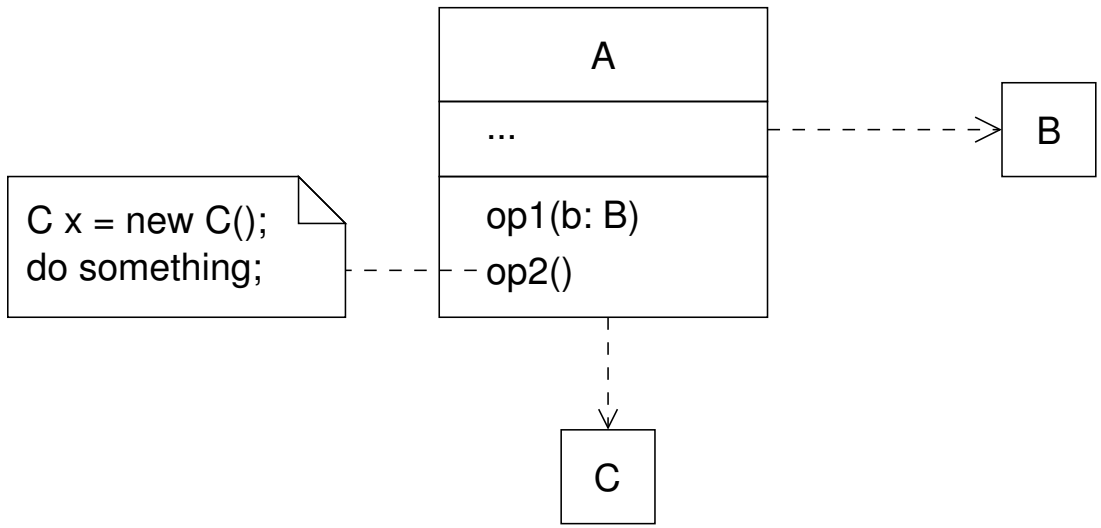
Beispiel:



2.1.3 Abhängigkeiten (Dependencies)

Eine Abhängigkeit ist eine gerichtete Beziehung zwischen Modellelementen, die besagt, dass Änderungen im Zielelement möglicherweise Änderungen im davon abhängigen Element nach sich ziehen.

Beispiel:

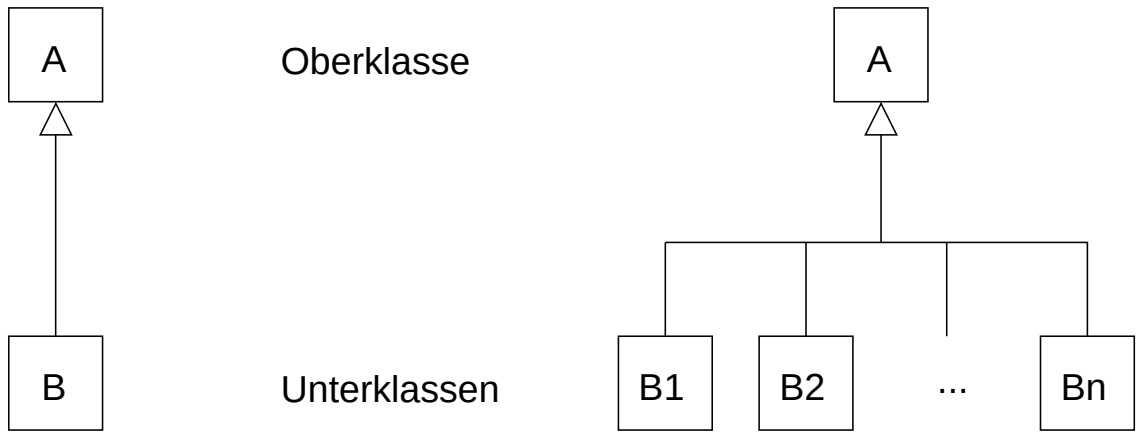


2.1.4 Vererbung

Relation zwischen einer "allgemeineren" Klasse (Ober- bzw. Superklasse) und einer "spezielleren" Klasse (Unter- bzw. Subklasse). Jedes Objekt der Subklasse ist auch ein Objekt der Oberklasse.

Beispiel:
Stoppuhr, Standuhr, Armbanduhr, Digitaluhr sind Uhren

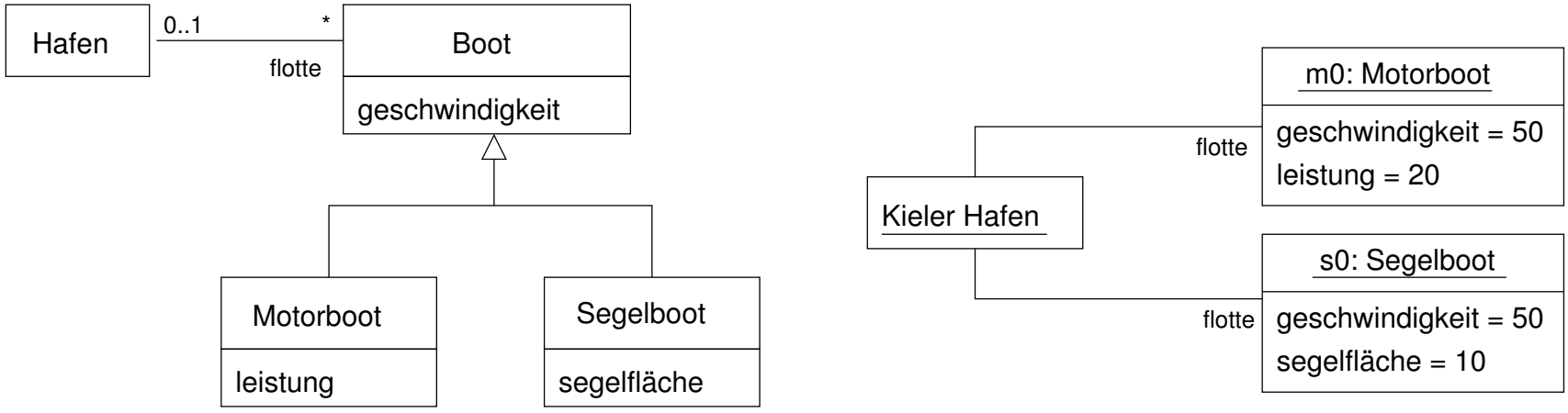
Darstellung:



- A ist eine *Generalisierung* von B.
- B ist eine *Spezialisierung* von A.

Beachte: Die Vererbungsbeziehung ist transitiv.

Beispiel:



Jede Unterklasse besitzt (erbt) alle Attribute, Assoziationen und Operationen der Oberklasse und kann eigene hinzufügen.

Substitutionsprinzip

Immer wenn ein Objekt einer Oberklasse A erwartet wird, kann ein Objekt einer Unterklasse B von A eingesetzt werden.

Beachte:

Klassennamen sind Typen. Ist A ein Klassenname und B der Name einer Unterklasse von A, dann ist B ein *Subtyp* von A.

Abstrakte Klasse

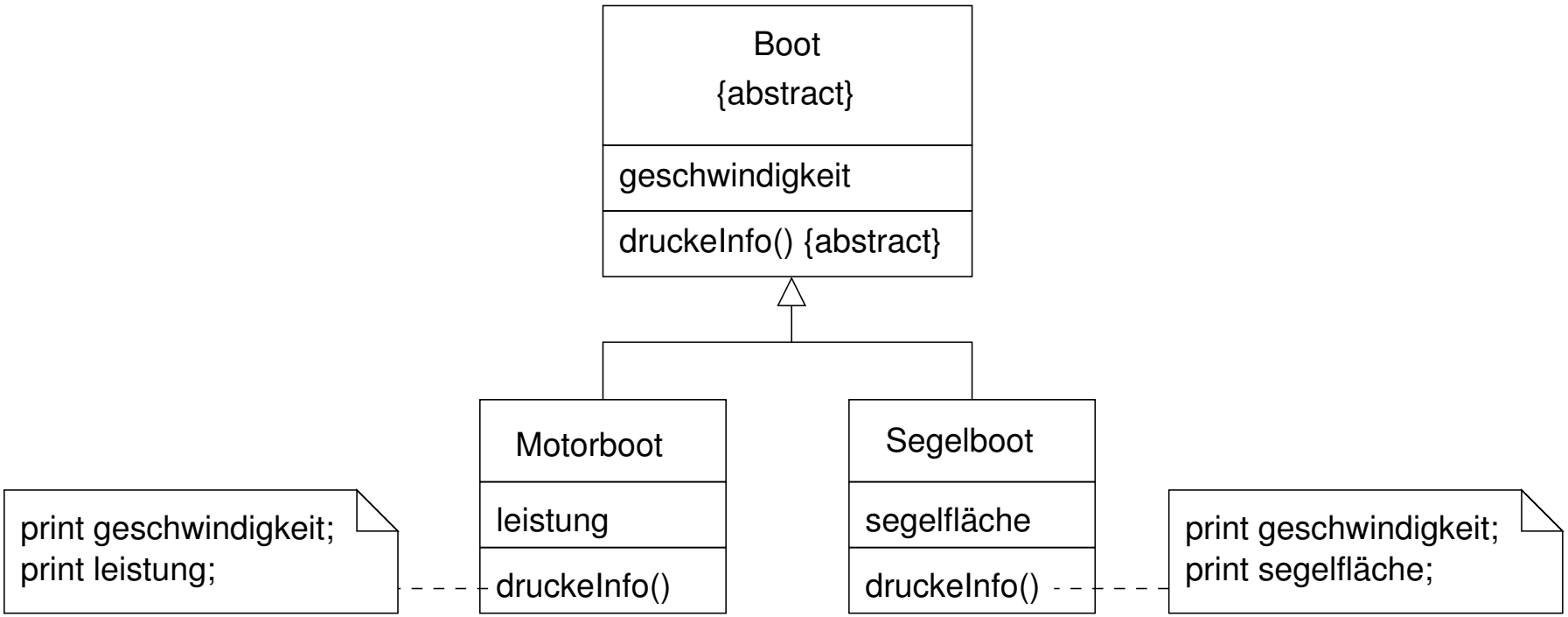
Klasse, von der keine Instanzen erzeugt werden können.

Abstrakte Operation

Operation ohne Implementierung.

Beachte:
Eine Klasse mit mindestens einer abstrakten Operation ist abstrakt.

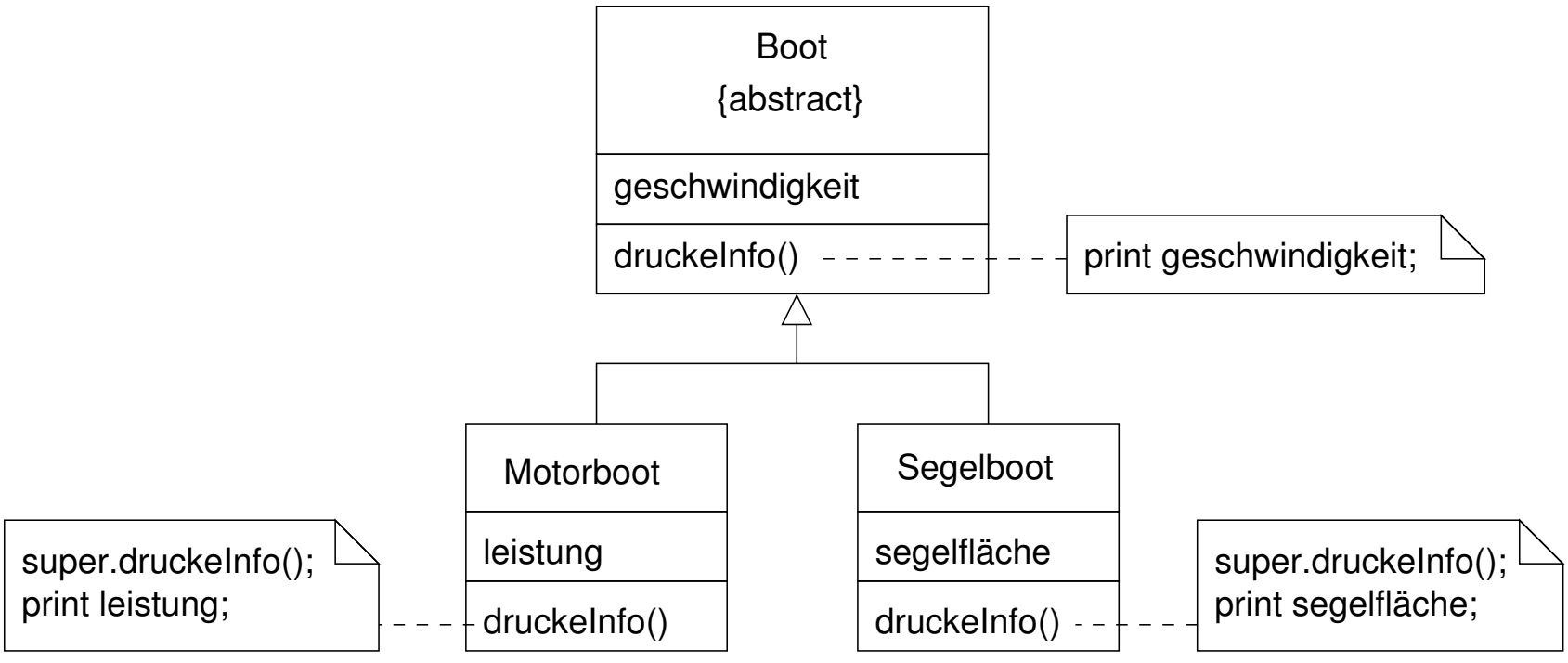
Beispiel:



Überschreiben

Eine in einer Oberklasse implementierte Operation wird in einer Unterklasse neu implementiert (redefiniert).

Beispiel:

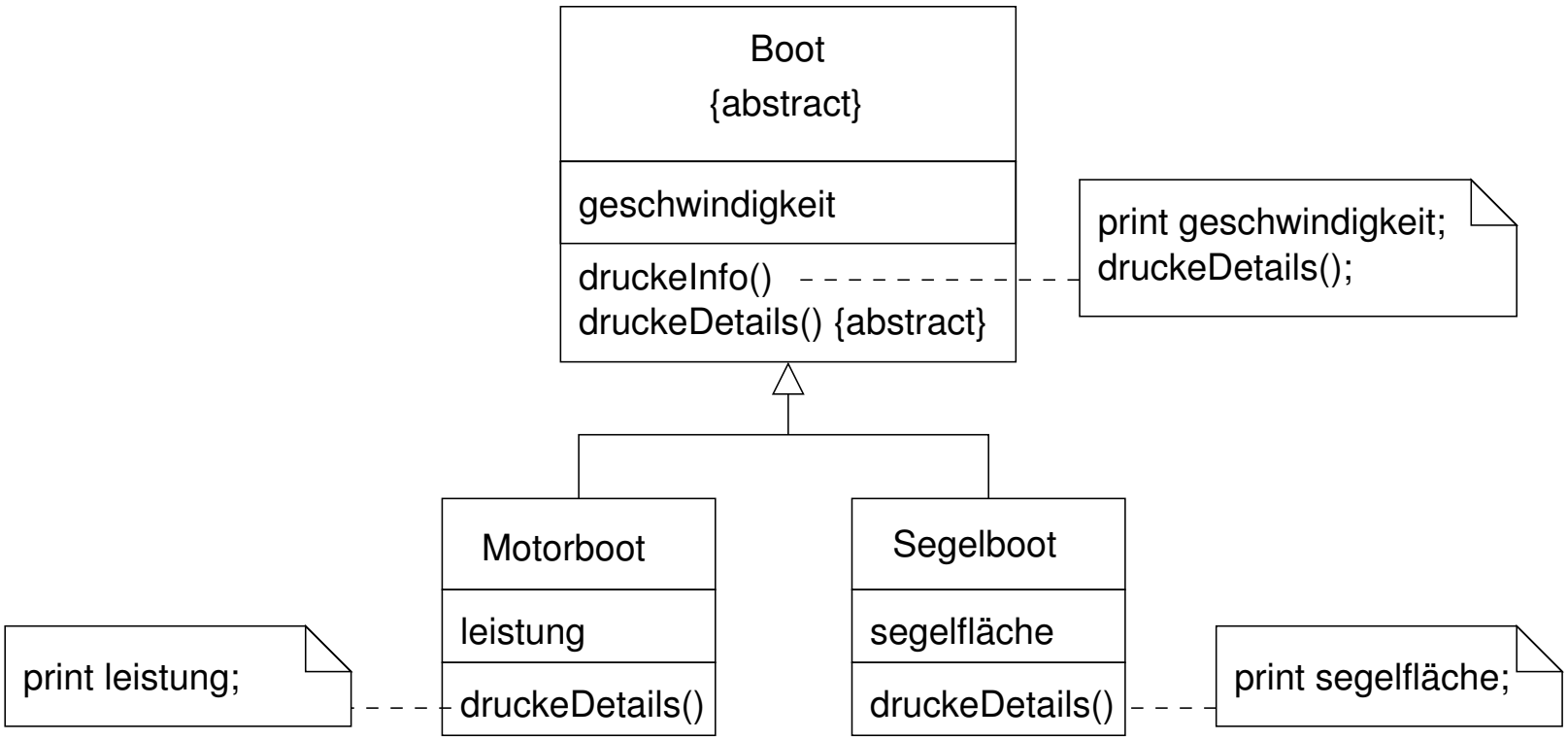


Bemerkung:
Durch Überschreiben soll die Semantik einer Operation nicht verändert werden.

Template Operation

Operation, deren Implementierung eine oder mehrere abstrakte Operationen aufruft.

Beispiel:

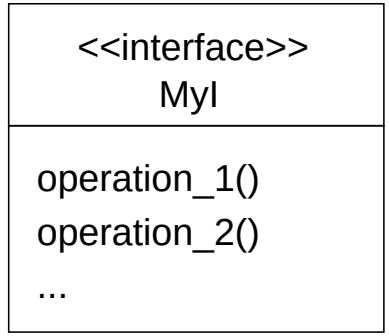


Schnittstellen

Sind abstrakte Klassen, deren sämtliche Operationen abstrakt sind und die keine Attribute und keine bidirektionalen oder wegführenden Assoziationen besitzen.

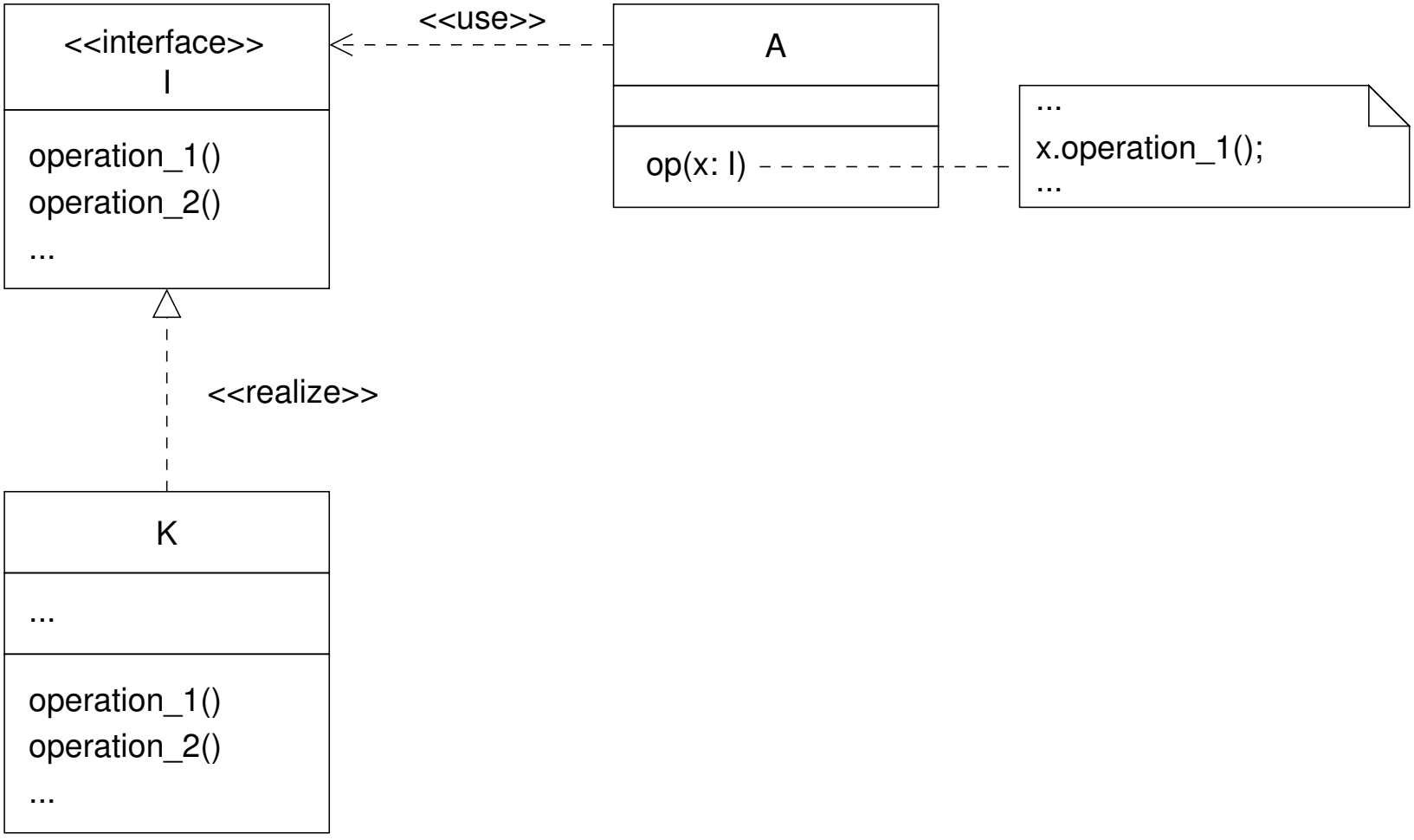
Folglich sind Interface-Namen auch Typen.

Darstellung:

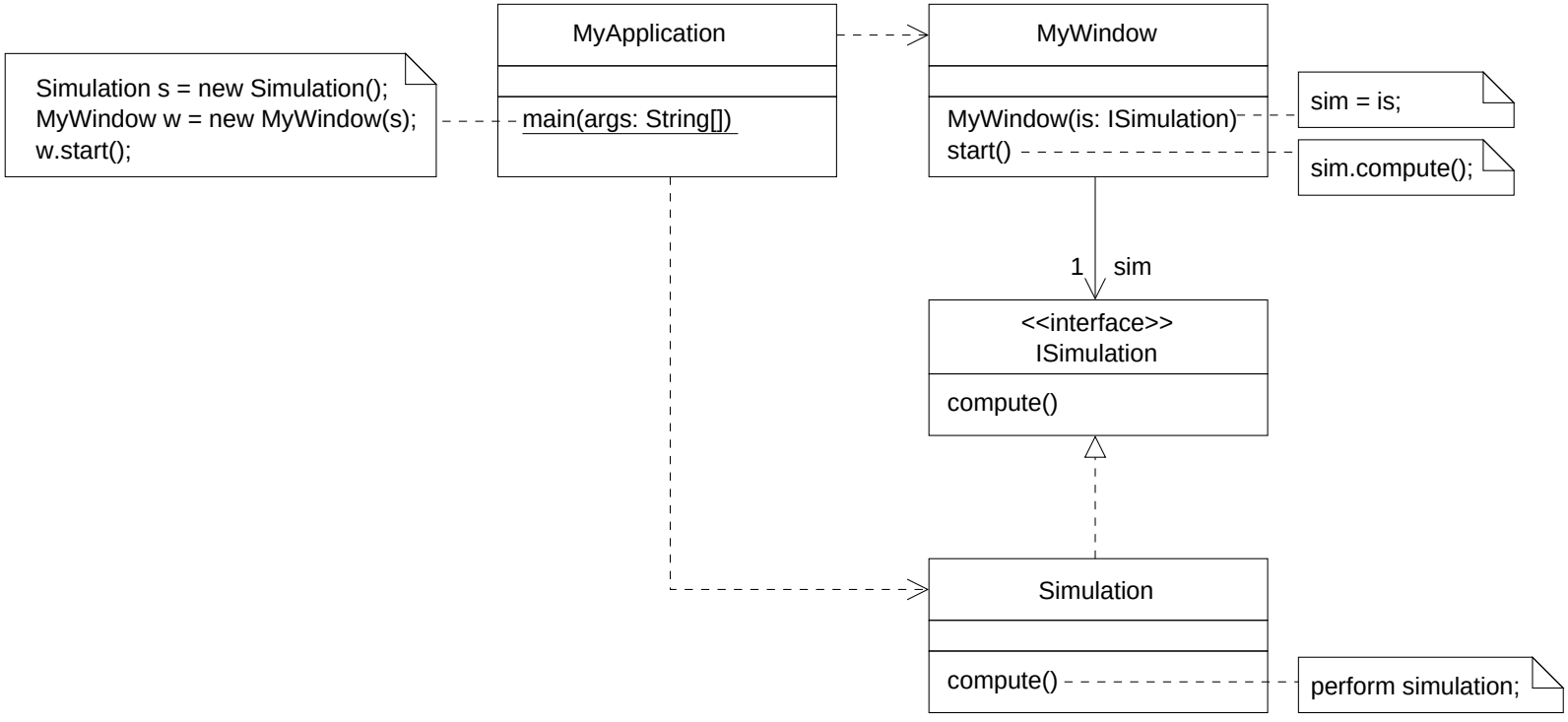


Realisierung und Benutzung von Schnittstellen

Schnittstellen bieten Dienste an, die von verschiedenen Klassen realisiert (implementiert) werden können und die von anderen Klassen genutzt werden können.



Beispiel:


MyApplication
MyWindow
Simulation s = new Simulation();
MyWindow w = new MyWindow(s);
w.start();
main(args: String[])
MyWindow(is: ISimulation)
start()
sim = is;
sim.compute();
1
sim
<<interface>>
ISimulation
compute()
Simulation
compute()
perform simulation;

Bemerkungen

- Überall, wo ein Bezeichner mit einem Interface-Typ verwendet wird, kann ein Objekt einer realisierenden Klasse eingesetzt werden.
- Ist I ein Interface-Name und K der Name einer realisierenden Klasse von I, dann ist K ein Subtyp von I.
- Durch Verwendung von Schnittstellen (oder von Oberklassen) können Objekte bestimmter Klassen zur Laufzeit in Beziehung stehen, obwohl die betreffenden Klassen zur Programmierzeit voneinander unabhängig sind.
- Implementierungen von Schnittstellen können ausgetauscht werden ohne Änderungen am Nutzer (Client) vornehmen zu müssen.
- Schnittstellen sind ein wichtiges Strukturierungsmittel für flexible Software-Architekturen.
- Häufig gibt es statt der Benutzungs-Abhängigkeit eine gerichtete Assoziation vom “Client” zum Interface.

Dynamisches Binden

Beispiel:

```
Boot b;
Motorboot m = new Motorboot();
Segelboot s = new Segelboot();
int x;
... "x einlesen" ...
if (x > 0) b = m;
else b = s;
(*) b.druckeInfo();
```

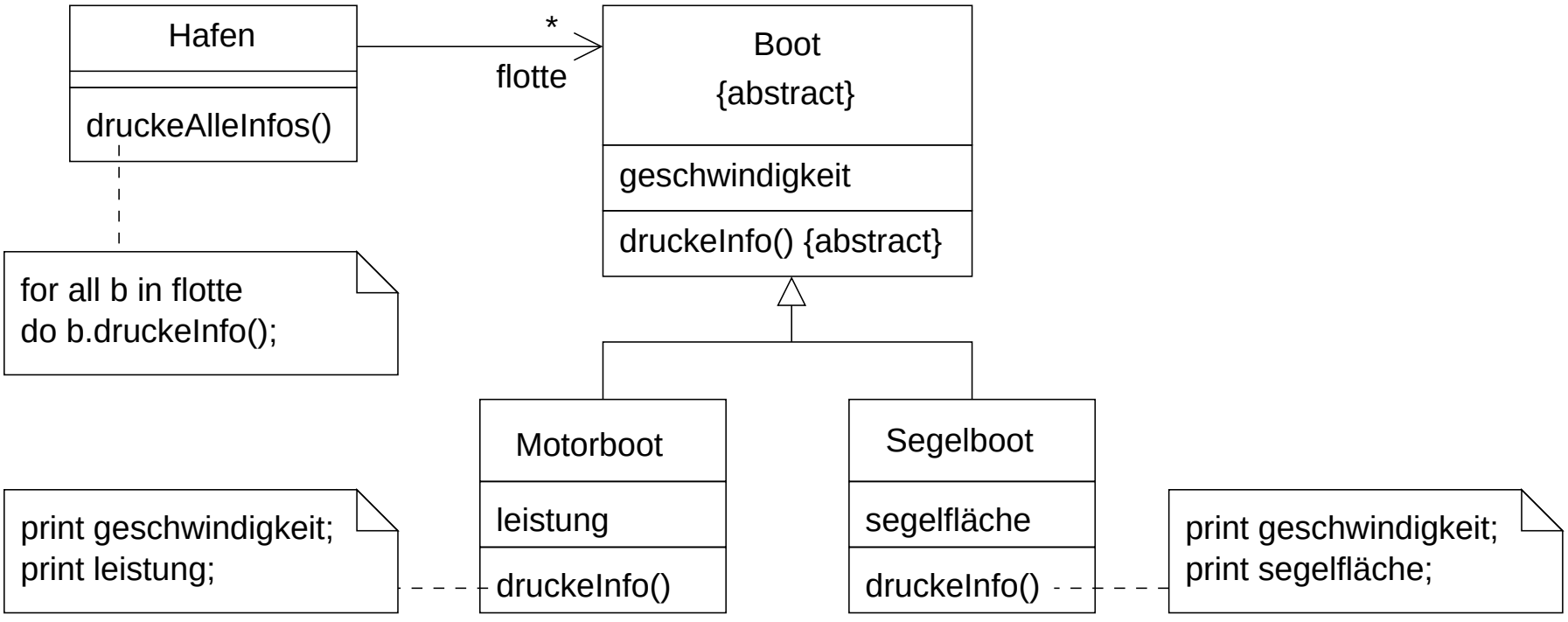
Zur Programmierzeit kann nicht festgestellt werden, welche Implementierung von "druckeInfo()"an der Stelle (*) gültig ist.

Zur Laufzeit wird festgestellt, welchen Typ das mit b bezeichnete Objekt hat. Der in der entsprechenden Klasse implementierte Code wird dann ausgeführt.

Subtyp-Polymorphismus

Eine Operation einer Oberklasse (bzw. einer Schnittstelle) kann für alle Objekte von Unterklassen (bzw. realisierenden Klassen) aufgerufen werden.

Beispiel:

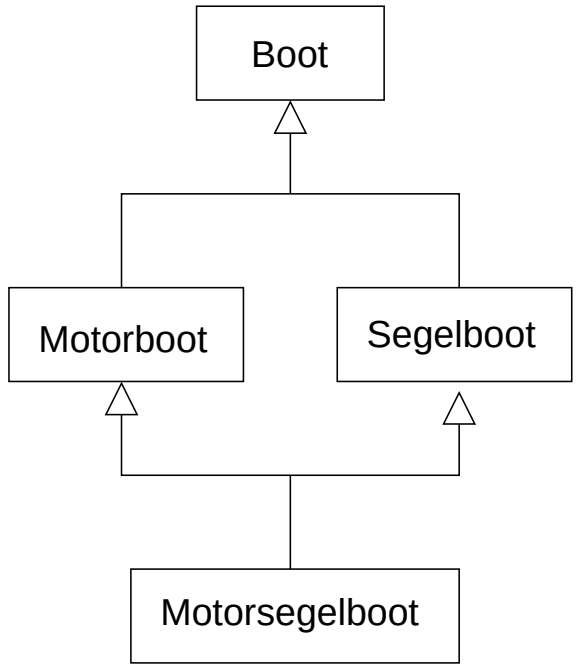


Vorteil:
Einfache Erweiterbarkeit durch Hinzunahme neuer Subklassen.

Mehrfachvererbung

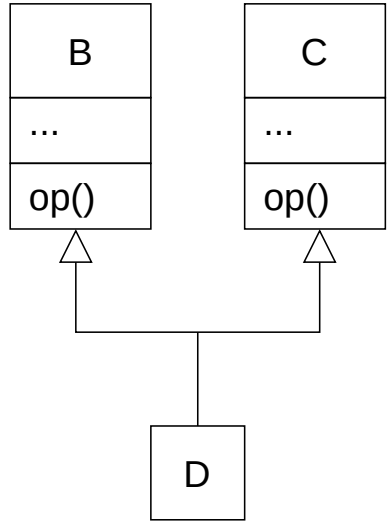
Eine Subklasse hat mehr als eine Oberklasse.

Beispiel:



Vorteil:
Zusammenfügen von Informationen aus mehreren Quellen.

Problem:
Mögliche Konflikte

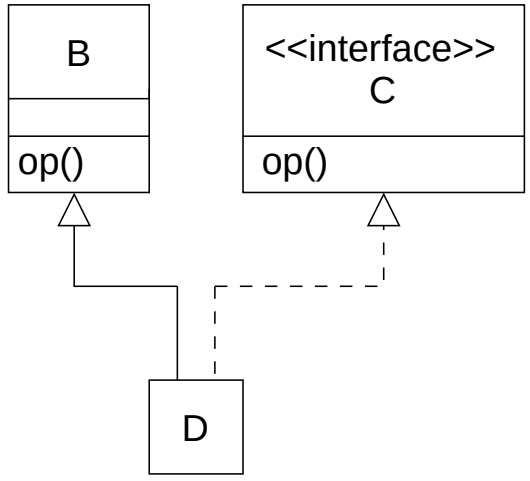


Welche Implementierung von op gilt für D-Objekte?

Konfliktauflösung:
D implementiert op neu durch Überschreiben oder op ist in B oder in C abstrakt.

Bemerkungen

- In Java ist Mehrfachvererbung nur bei Verwendung von Schnittstellen möglich.



- Mehrfachvererbung von Klassen muss beim Entwurf aufgelöst werden, wenn die Zielsprache keine Mehrfachverebung unterstützt.

Vorteile des Vererbungsprinzips

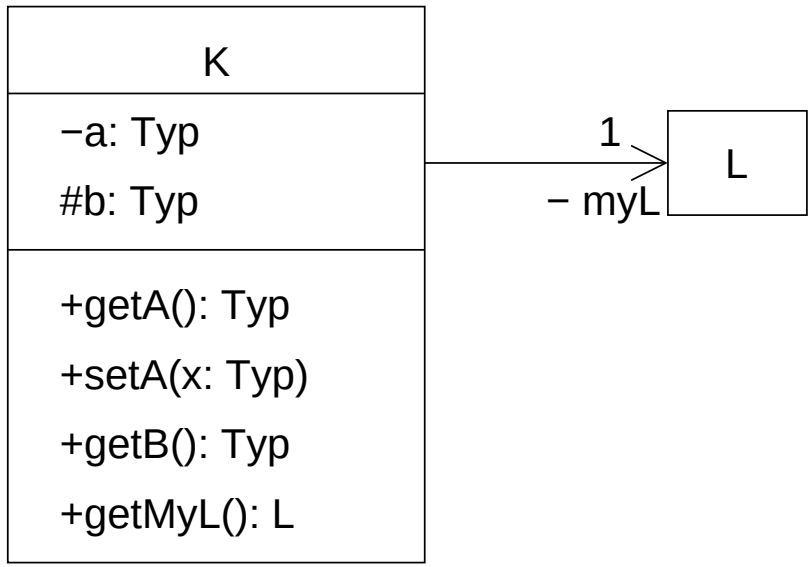
- Konzeptionelle Vereinfachung durch Zusammenfassen gemeinsamer Merkmale verwandter Klassen in einer Oberklasse (*Generalisierung*)
- Wiederverwendung bereits vorhandener Klassen durch Subklassenbildung (*Spezialisierung*)
- Einfache Erweiterbarkeit von Vererbungshierarchien durch Hinzunahme von Subklassen

Vorteile von Schnittstellen

- Keine Abhängigkeit von konkreten Implementierungen.
- Einfache Austauschbarkeit von Realisierungen von Schnittstellen.

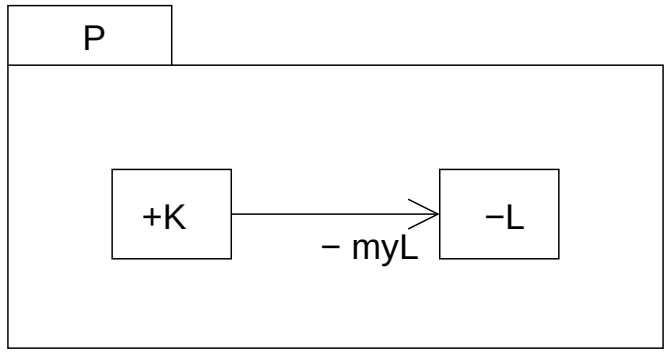
2.1.5 Zugriffsrechte (Sichtbarkeiten)

- 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 und Rollennamen sollten nicht öffentlich zugreifbar sein!



Bemerkung

Innerhalb von *Paketen* (vgl. später) können auch Klassen und Interfaces mit Sichtbarkeiten "+"oder "-"versehen werden.



Wirkung:

- Die Klasse K und deren öffentliche Elemente sind auch außerhalb des Pakets P sichtbar.
- Die geschützten Elemente von K sind auch in Subklassen außerhalb des Pakets P sichtbar.
- Die Klasse L und deren öffentliche Elemente sind nur innerhalb des Pakets P sichtbar.
- Die geschützten Elemente von L sind nur in Subklassen innerhalb des Pakets P sichtbar.

Bemerkung

Attribute, Rollennamen und Operationen, die nur innerhalb eines Pakets sichtbar sein sollen, werden mit "∼"markiert ("komponenten-privat").

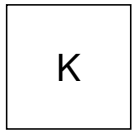
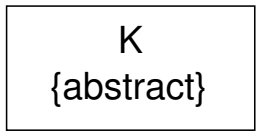
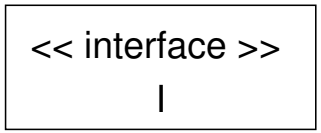
Zusammenfassung von Abschnitt 2.1

- Klassendiagramme werden aus Klassen (einschl. Schnittstellen), Assoziationen, Abhängigkeiten, Vererbungs- und Realisierungsbeziehungen gebildet.
- Objektdiagramme werden aus Objekten und Objektbeziehungen gebildet.
- Mit Hilfe des Vererbungskonzepts kann spezialisiert und generalisiert werden.
- Es gilt das Substitutionsprinzip (bzgl. der Subtypbeziehung).
- Durch dynamische Bindung wird zur Laufzeit festgestellt, welchen Typ ein Objekt hat und der dementsprechende Code ausgeführt.
- Schnittstellen bieten ein wichtiges Strukturierungsmittel für flexible Software-Architekturen.
- Zugriffsrechte können mit Hilfe von Sichtbarkeitsmarkierungen spezifiziert werden.

2.2 Implementierung von Klassendiagrammen in Java

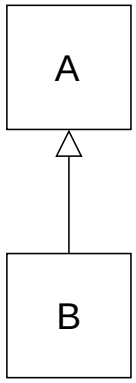
- Die statischen Informationen eines Klassendiagramms können direkt nach Java übersetzt werden.
- Das entstehende Codegerüst enthält noch keine Methodenimplementierungen.

2.2.1 Klassen und Schnittstellen deklarieren

UML	Java
	`class K {...}`
	`abstract class K {...}`
	`interface I {...}`

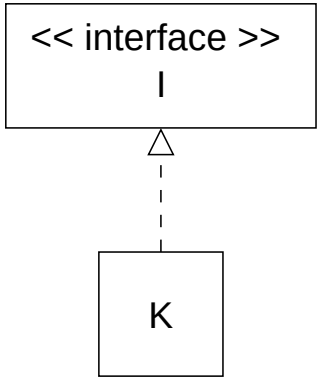
UML

Java




```
class A {...}
```

```
class B extends A {...}
```




```
interface I {...}
```

```
class K implements I {...}
```

2.2.2 Attribute deklarieren

UML	Java
attribut:Typ	JavaTyp attribut;

wobei die verwendeten Standarddatentypen folgendermaßen in Java-Typen übersetzt werden:

UML	Java
Boolean	boolean
Integer	int
Real	float oder double
String	String

2.2.3 Methodenköpfe deklarieren

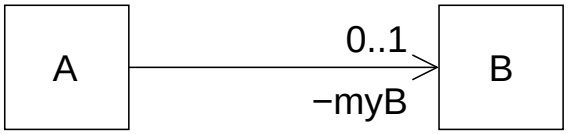
UML	Java
op(x: Typ)	void op(JavaTyp x) {...}
op(x: Typ): ResTyp	JavaResTyp op(JavaTyp x) {...}
op() {abstract}	abstract void op();
K(x: Typ)	K(JavaTyp x) {...} //Konstruktor

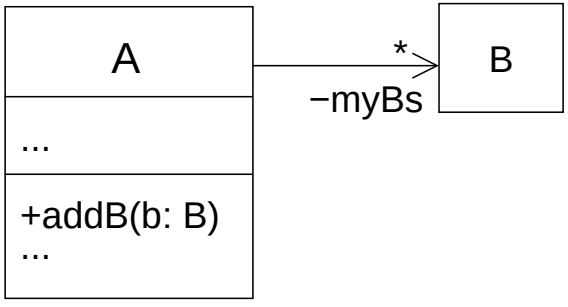
2.2.4 Zugriffsrechte bestimmen

UML	Java	
-	private	//in derselben Klasse sichtbar
$\sharp$	protected	//in Subklassen und im selben Paket sichtbar
+	public	//außerhalb der Klasse sichtbar
$\sim$		//Java Default-Visibility: im selben Paket sichtbar

2.2.5 Assoziationen darstellen

UML



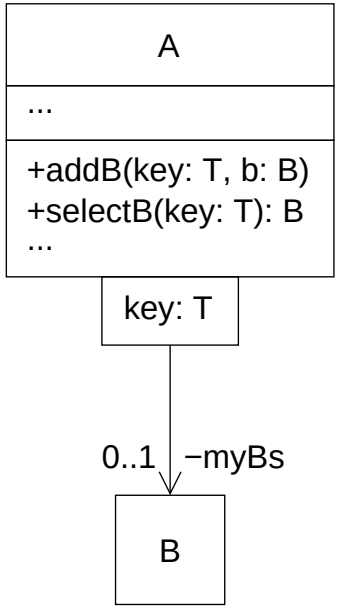


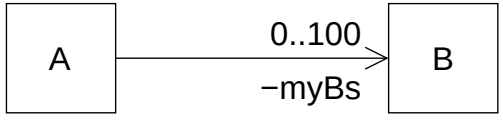
Java

```
class A {
  private B myB; //Referenzattribut
  ...
}
```

```
//Set = Interface für Mengen
//HashSet = Implementierung v. Set

import java.util.*;

class A {
  private Set<B> myBs = new HashSet<B>();
  ...
  public void addB(B b) {
    myBs.add(b);
  }
  ...
}
```

UML



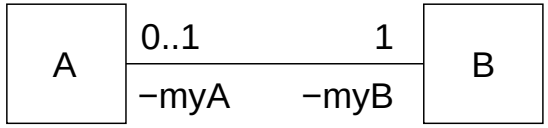


Java

```
//Map = Interface für Schlüssel/Element-Paare
import java.util.*;
class A {
  private Map<T,B> myBs = new HashMap<T,B>();
  ...
  public void addB(T key, B b) {
    myBs.put(key, b);
  }
  public B selectB(T key) {
    return myBs.get(key);
  }
  ...
}
```

```
class A {
  private B[] myBs = new B[100];
  ...
}
```

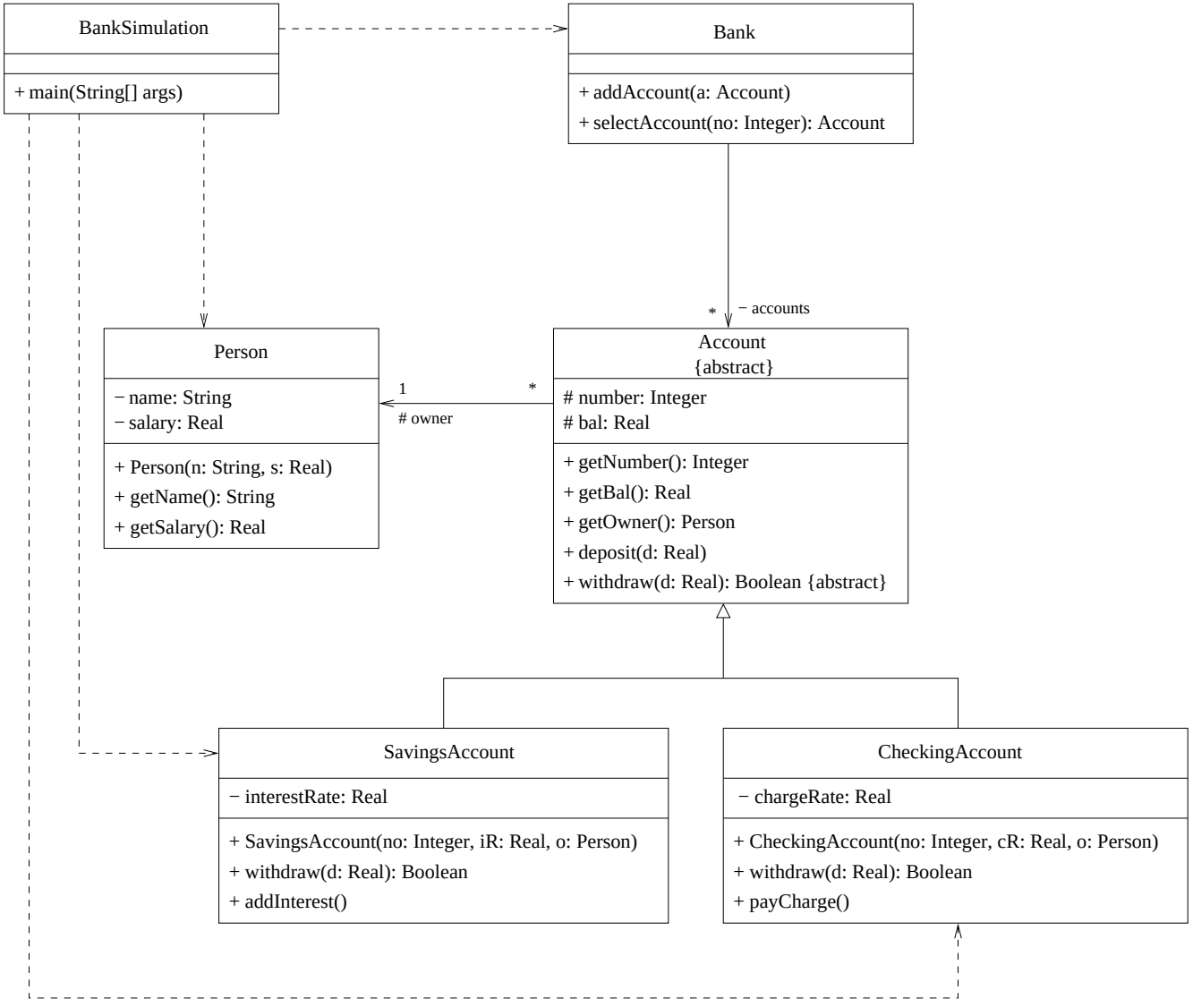
Bidirektionale Assoziationen implementieren




```
class A {
  private B myB;

  public A(B b) {
    myB = b;
    myB.setMyA(this);
  }
  public B getMyB() {
    return myB;
  }
  public void relate(B b) {
    myB = b;
    myB.setMyA(this);
  }
  public void unrelate() {
    myB.unset();
    myB = null;
  }
}
```

```
class B {
  private A myA;

  public A getMyA() {
    return myA;
  }
  void setMyA(A a) {
    myA = a;
  }
  void unsetMyA() {
    myA = null;
  }
}
```

Beispiel (Klassendiagramm)



Beispiel (Codegerüst)

```
class BankSimulation {

    public static void main(String[] args) {
        //Rumpf einfuegen
    }
}

import java.util.*;
class Bank {

    private Set<Account> accounts = new HashSet<Account>();

    public void addAccount(Account a) {
        //Rumpf einfuegen
    }
    public Account selectAccount(int no) {
        //Rumpf einfuegen
    }
}
```

```
class Person {

    private String name;
    private double salary;

    public Person(String n, double s) {
        //Rumpf einfuegen
    }
    public String getName() {
        //Rumpf einfuegen
    }
    public double getSalary() {
        //Rumpf einfuegen
    }
}
```

```
abstract class Account {

    protected int number;
    protected double bal;
    protected Person owner;

    public int getNumber() {
        //Rumpf einfuegen
    }
    public double getBal() {
        //Rumpf einfuegen
    }
    public Person getOwner() {
        //Rumpf einfuegen
    }
    public void deposit(double d) {
        //Rumpf einfuegen
    }
    public abstract boolean withdraw(double d);
}
```

```
class SavingsAccount extends Account {

    private double interestRate;

    public SavingsAccount(int no,double iR,Person o) {
        //Rumpf einfuegen
    }
    public boolean withdraw(double d) {
        //Rumpf einfuegen
    }
    public void addInterest() {
        //Rumpf einfuegen
    }
}
```

```
class CheckingAccount extends Account {

    private double chargeRate;

    public CheckingAccount(int no,double cR,Person o) {
        //Rumpf einfuegen
    }

    public boolean withdraw(double d) {
        //Rumpf einfuegen
    }

    public void payCharge() {
        //Rumpf einfuegen
    }
}
```

2.3 Modellierung des dynamischen Verhaltens

Techniken

- *Interaktionsdiagramme:*
  Beschreiben die Kommunikation und die Zusammenarbeit von *mehreren* Objekten.
- *Zustandsdiagramme:*
  Beschreiben das Verhalten *eines* Objekts einer bestimmten Klasse während seiner Lebenszeit.
- *Aktivitätsdiagramme:*
  Beschreiben die (evtl. parallelen) Abläufe von Aktivitäten.

2.3.1 Zustände und Ereignisse

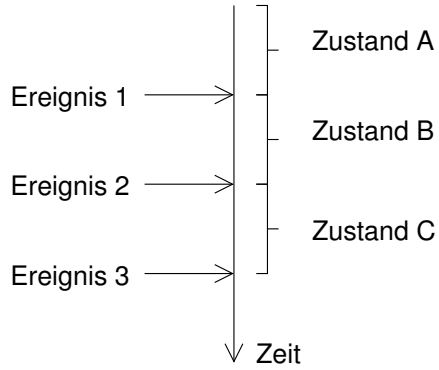
Zustände

- Die aktuellen Attributwerte und Beziehungen eines Objekts zu einem Zeitpunkt bestimmen den *aktuellen Objektzustand*.
- Objektzustände, in denen Objekte qualitativ die gleichen Reaktionen auf eintreffende Ereignisse haben, sind äquivalent. Sie werden zu *einem* (abstrakten) Zustand zusammengefasst.

Beispiel: 4 Briefzustände

In jedem einzelnen Zustand wird dieselbe Briefmarke aufgeklebt.

Ereignis *(event)* = Vorfall, der zu einem bestimmten Zeitpunkt stattfindet.



Arten von Ereignissen

- Empfang eines Signals: *signal event* (z.B. Button klicken, Telefonhörer abheben)
- Aufruf einer Operation: *call event* (z.B. `myKonto.einzahlen(1000)`)
- Eine Bedingung wird wahr: *change event* (z.B. **when** (temperature < 0))
- Ablauf einer Zeit: *time event* (z.B. **after**(5 sec))
- Beendigung einer Aktivität: *completion event* (z.B. WWW-Seite ist geladen)

Beachte:
Ereignisse haben keine Dauer (im Gegensatz zu Zuständen)!

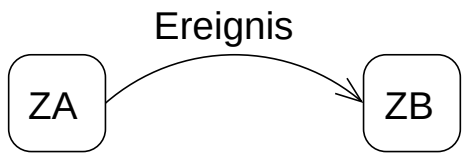
2.3.2 Flache Zustandsdiagramme

Gerichteter Graph mit

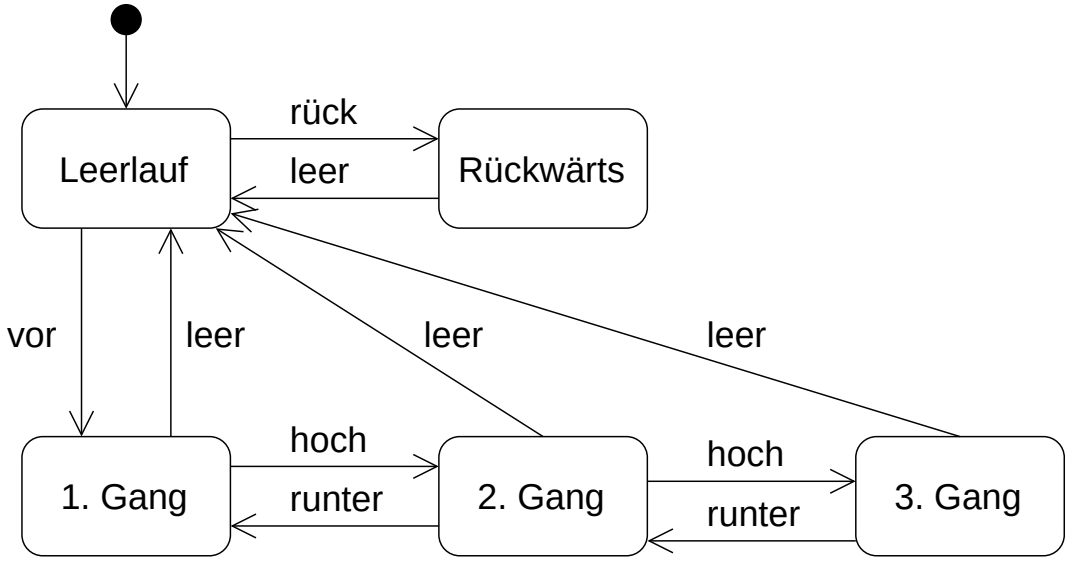
- Knoten = Zustände
- Kanten = Transitionen

Transition

Beschreibt den durch ein Ereignis verursachten Übergang von einem "alten" in einen "neuen" Zustand.



Beispiel: Automatikgetriebe



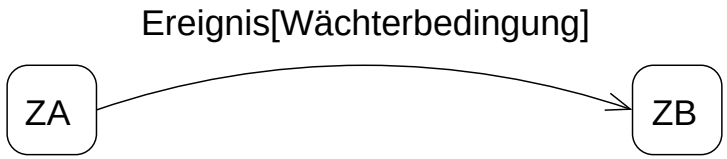
Bemerkungen

- Ein Ereignis, für das es von einem Zustand aus keine Transition gibt, wird in diesem Zustand "ignoriert".
- Das Symbol • bezeichnet den Anfangszustand ("Pseudozustand").
- Das Symbol ⊙ bezeichnet einen Endzustand (Destruktion des Objekts oder Beendigung einer Aktivität).

Wächter *(guards)*

- Eine Bedingung (boolescher Ausdruck) kann als Wächter für eine Transition verwendet werden.
- Die Transition feuert, wenn das Ereignis eintritt *und* die Bedingung erfüllt ist.

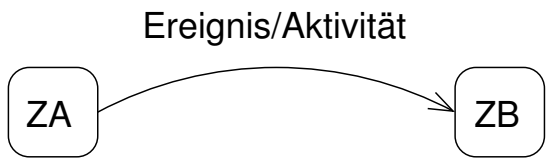
Syntax:



Aktivität

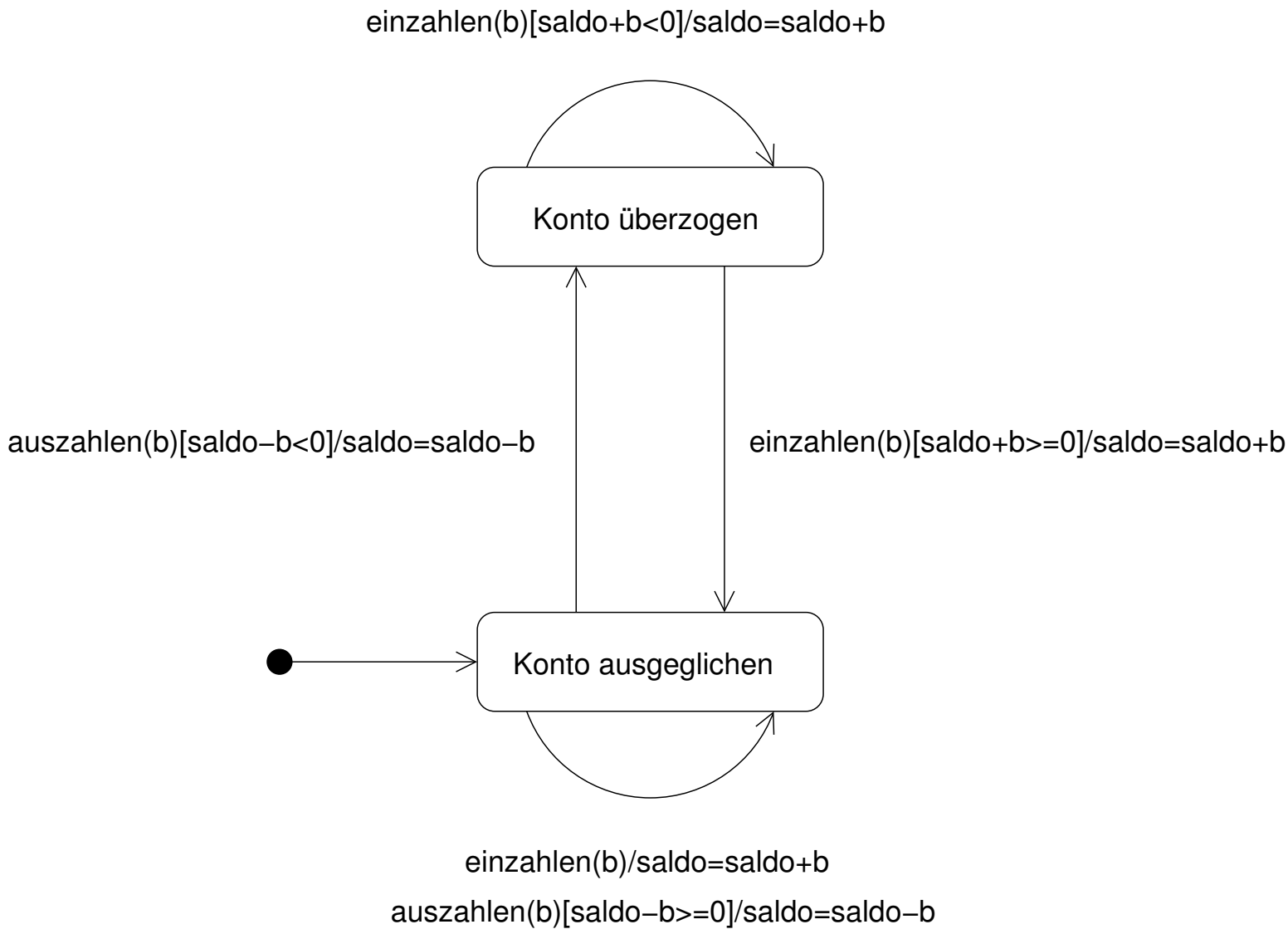
- Tätigkeit (die Zeit in Anspruch nehmen kann).
- Kann als Reaktion auf ein Ereignis erfolgen.

Syntax:

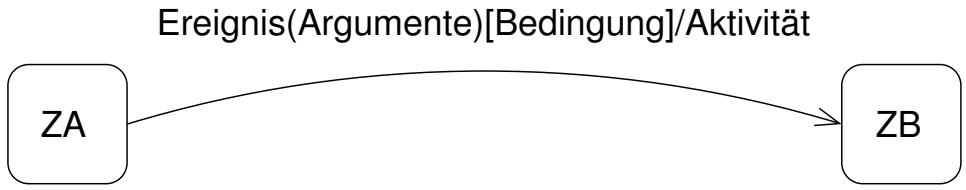


Beachte:
Eine auf einer Transition vorkommende Aktivität kann nicht durch ein Ereignis unterbrochen werden.

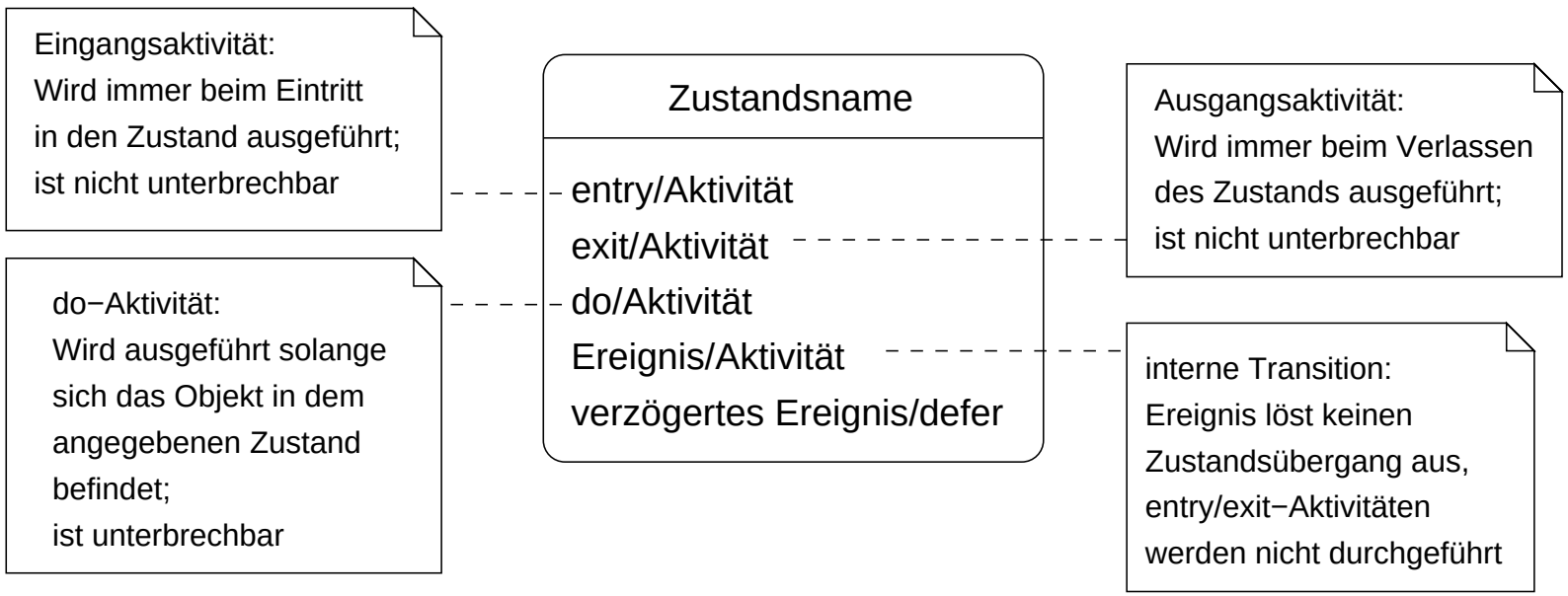
Beispiel:


einzahlen(b)[saldo+b<0]/saldo=saldo+b
Konto überzogen
auszahlen(b)[saldo−b<0]/saldo=saldo−b
einzahlen(b)[saldo+b>=0]/saldo=saldo+b
Konto ausgeglichen
einzahlen(b)/saldo=saldo+b
auszahlen(b)[saldo−b>=0]/saldo=saldo−b

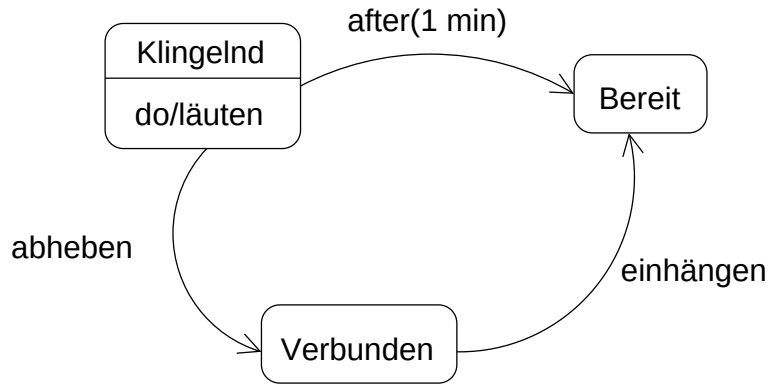
Allgemeine Syntax von Transitionen



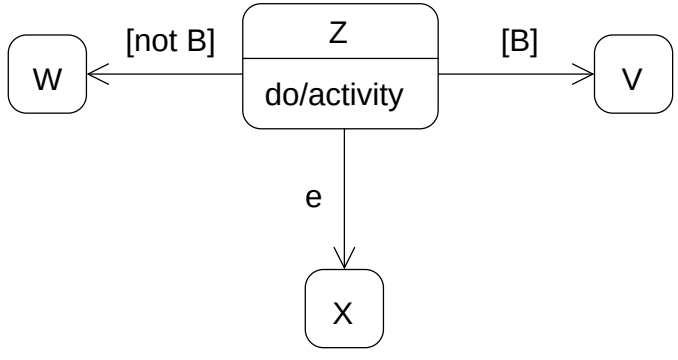
Allgemeine Syntax von Zuständen



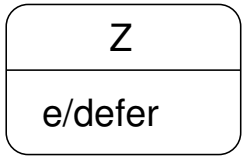
Beispiel: Telefon



Beachte:
Wird die do-Aktivität von selbst beendet, dann erfolgt ein (evtl. bedingtes) Completion Event.

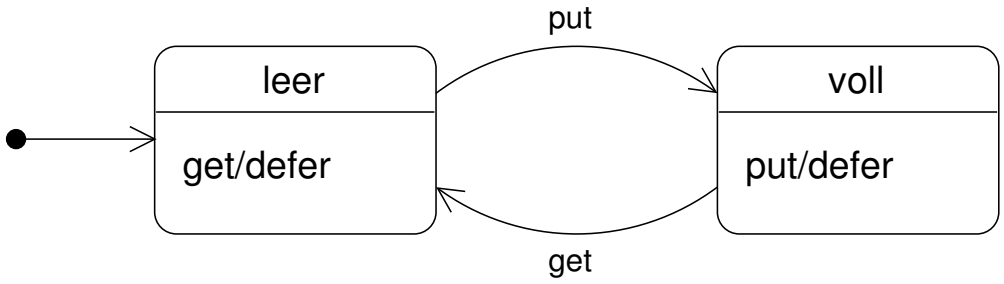


Verzögertes Ereignis



Erfolgt das Ereignis e im Zustand Z und gibt es in Z keine ausgehende Transition mit dem Ereignis e, dann wird das Ereignis aufbewahrt und erst dann verarbeitet, wenn sich das Objekt in einem (anderen) Zustand befindet, in dem das Ereignis verarbeitet werden kann.

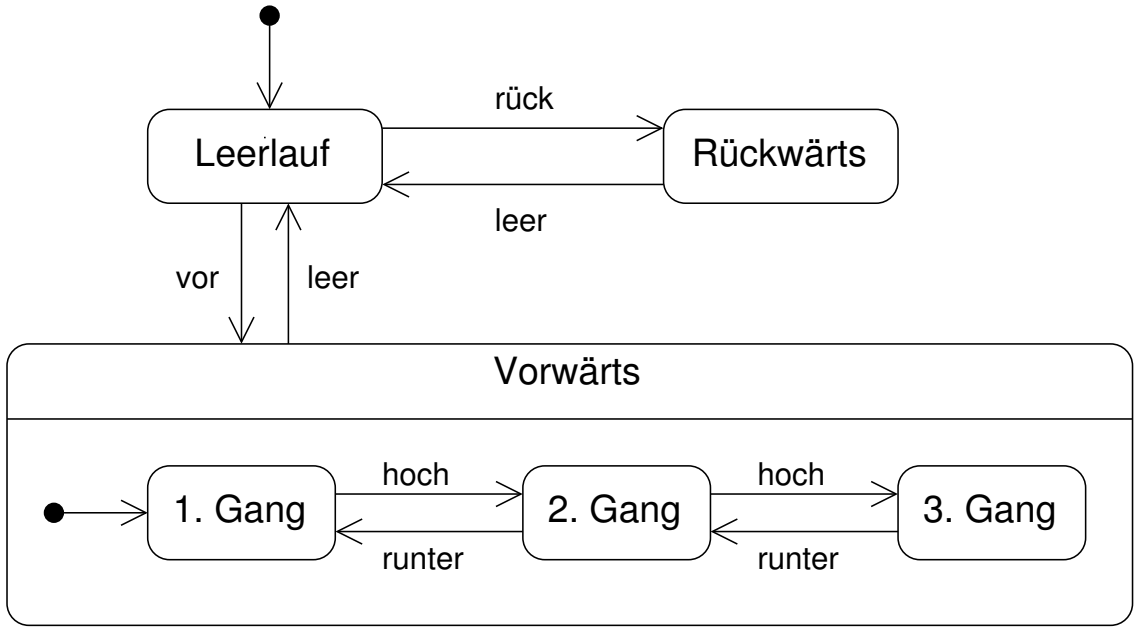
Beispiel: Einelementiger Puffer



2.3.3 Hierarchische Zustandsdiagramme

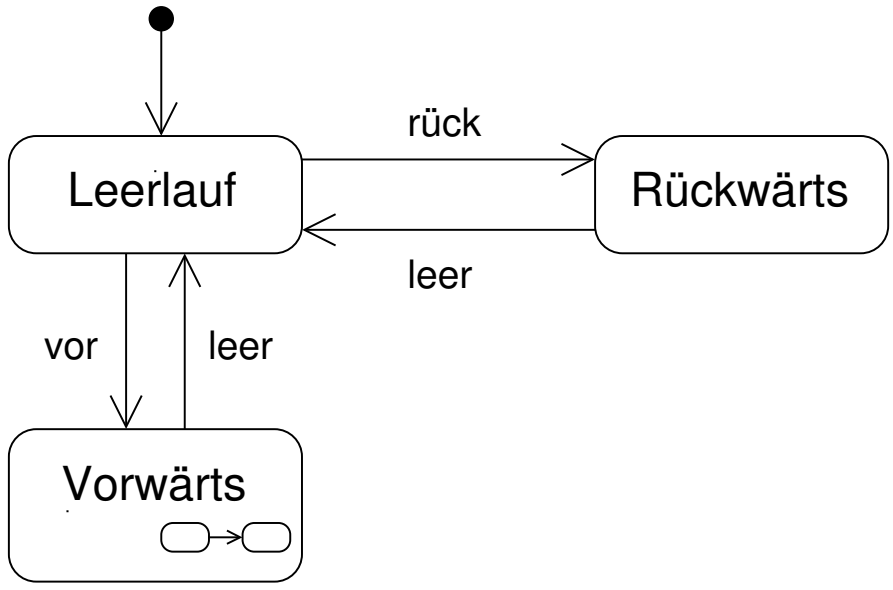
Ein Zustand kann in Unterzustände verfeinert werden.

1. Sequentielle Unterzustände



- Transition in einen Oberzustand (hier: Vorwärts) bedeutet Transition in den Anfangszustand des geschachtelten Diagramms (hier: 1. Gang).
- Transition aus einem Oberzustand bedeutet Transition ausgehend von jedem Unterzustand.

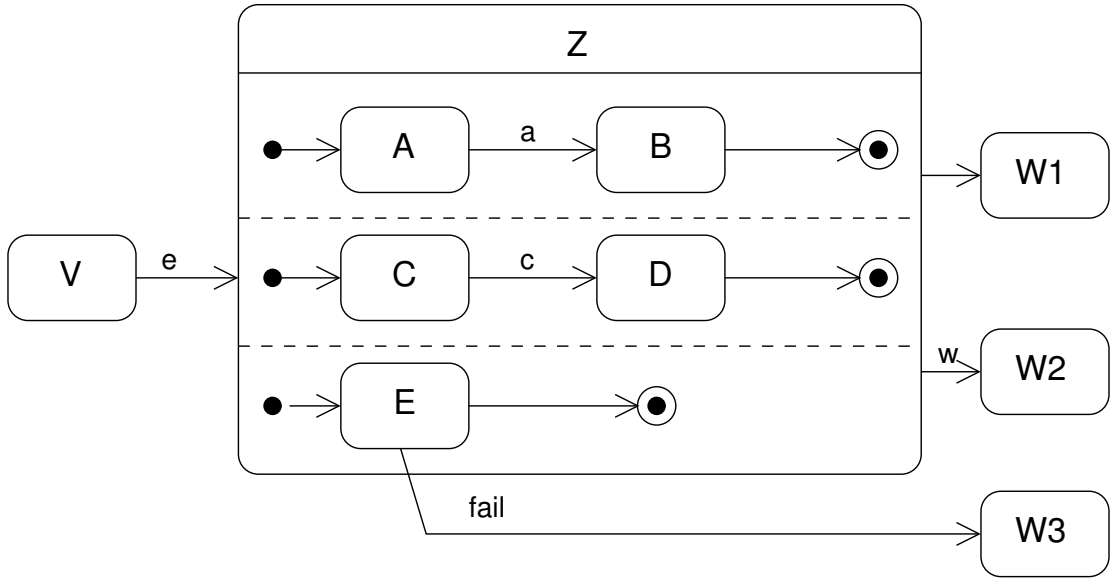
Abstrakte Darstellung eines komplexen Zustands:



Bemerkung

Komplexe Zustände können mit *Einstiegs-* und *Ausstiegspunkten (entry, exit points)* versehen werden.

2. Parallele Unterzustände



- Ein Objekt befindet sich gleichzeitig in mehreren Unterzuständen. Beim Eintritt in den Oberzustand befindet sich das Objekt gleichzeitig in den Anfangszuständen der einzelnen Regionen.
- Der Oberzustand wird verlassen, wenn in jeder Region ein Endzustand erreicht ist oder wenn eine Transition von einem Unterzustand aus direkt nach außen führt oder wenn eine Transition den Oberzustand (wegen eines expliziten Ereignisses) verlässt.

2.3.4 Aktivitätsdiagramme

Können zur Beschreibung der Abläufe von

- Geschäftsprozessen in Unternehmen
- Anwendungsfällen (Use Cases) oder von
- Operationen und Prozessen

verwendet werden.

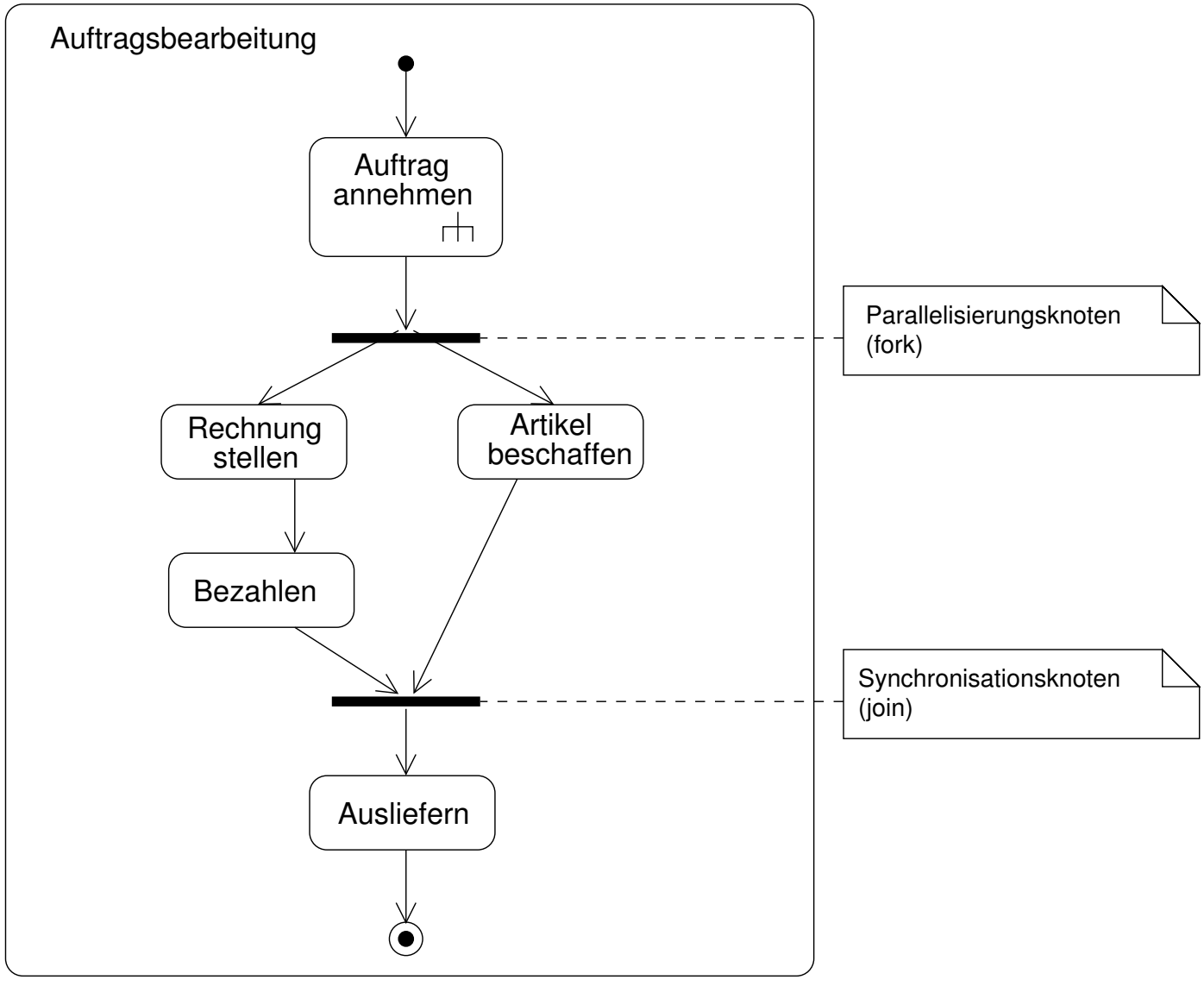
Ein Aktivitätsdiagramm ist ein gerichteter Graph, bestehend aus

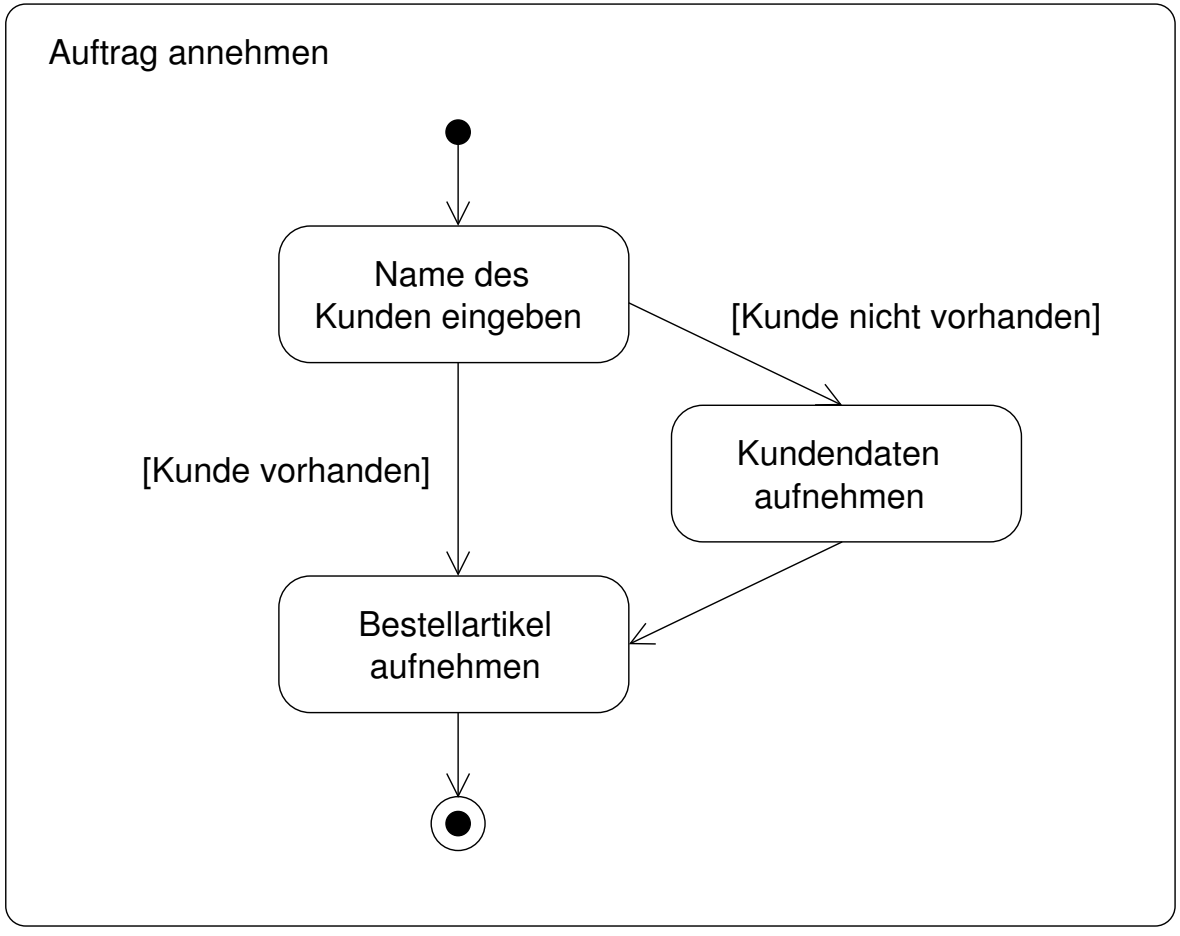
- *Aktivitätsknoten* zur Darstellung von Aktionen, Kontrollstrukturen und Daten,
- *Aktivitätskanten*, die die Aktivitätsknoten verbinden und damit die möglichen Abläufe einer Aktivität beschreiben.

Bemerkung

Beispielsweise können Aktivitäten, die in Form von entry-, exit-. oder do-Aktivitäten an Zustände gebunden sind, durch Aktivitätsdiagramme genauer beschrieben werden.

Beispiel: Geschäftsprozess “Auftragsbearbeitung”





Bemerkung

Fallunterscheidungen können auch durch Verwendung von *Entscheidungsknoten* dargestellt werden.

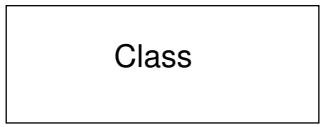
Zusammenfassung von Abschnitt 2.3

- Zustandsdiagramme beschreiben das Verhalten jedes Objekts einer bestimmten Klasse während seiner Lebenszeit.
- Eine Transition beschreibt den durch ein Ereignis verursachten Zustandsübergang.
- Ereignisse sind im Gegensatz zu Zuständen (und Aktivitäten) zeitlos.
- Wir unterscheiden fünf verschiedene Arten von Ereignissen.
- Ereignisse können bewacht sein und auf ein Ereignis kann eine Aktivität folgen.
- Zustandsdiagramme können hierarchisch strukturiert werden.
  Wir unterscheiden
  - sequentielle Unterzustände
  - parallele Unterzustände
- Aktivitätsdiagramme beschreiben Abläufe von Aktivitäten.

2.4 Metamodellierung

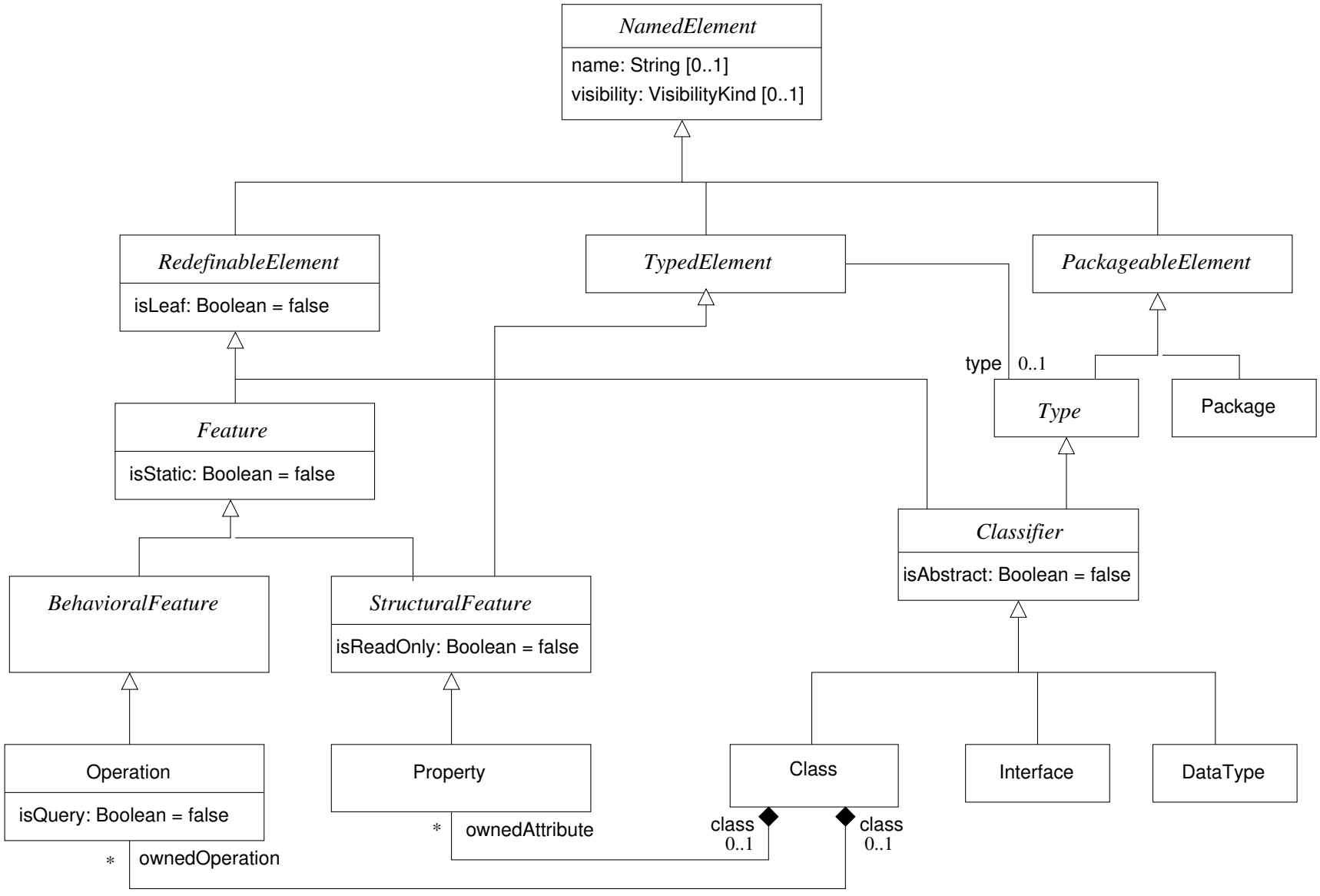
Alle in einem UML-Modell benutzten Konzepte (wie z.B. Klasse, Operation, Zustand, Aktivität, ...) werden selbst durch ein Klassenmodell beschrieben.

Beispiel: Metaklasse Class hat als Instanzen Klassen



- Das Metamodell spezifiziert alle zulässigen UML-Modelle, die Instanzen des Metamodells sein müssen.
- Damit ist
  - die (syntaktische) Korrektheit von UML-Modellen überprüfbar
  - die Basis für ein standardisiertes Austauschformat geschaffen (XMI)
- Das Metamodell kann für die Modellierung bestimmter Anwendungsbereiche (Business Modeling, Web Engineering, ...) erweitert werden.

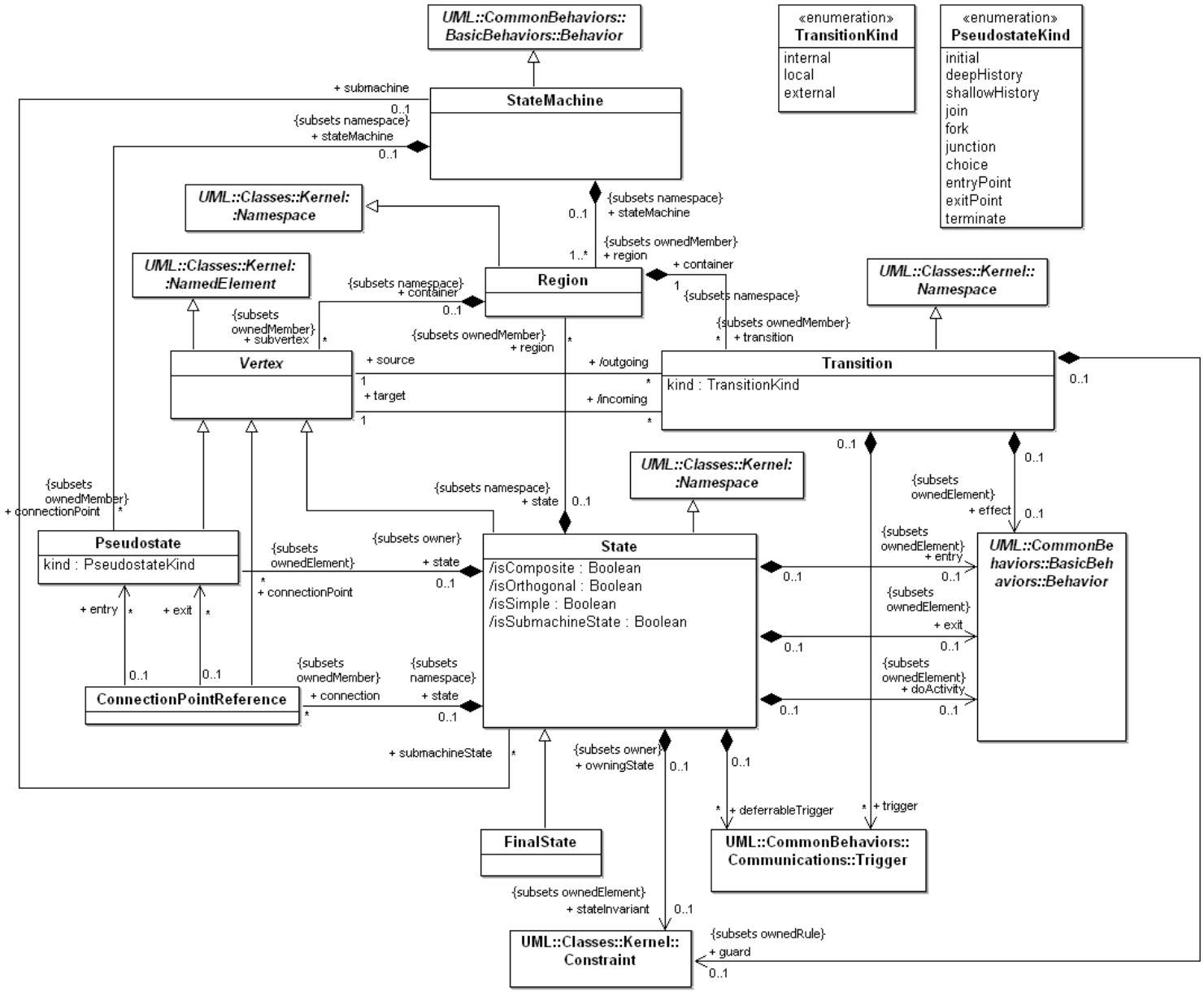
Ausschnitt aus dem UML-Metamodell



Metamodell für Zustandsdiagramme

(aus der UML 2.2 Superstructure Specification)

Package BehaviorStateMachines



Kapitel 3

Objektorientierte Analyse

Prof. Dr. Rolf Hennicker

24.11.2009

Ziele

- Eine Anwendungsfall-Analyse für ein zu entwickelndes System durchführen können.
- Schrittweise ein statisches Modell für ein Anwendungsproblem erstellen können.
- Interaktionsdiagramme für kommunizierende Objekte erstellen können.
- Zustands- und Aktivitätsdiagramme (aus dem Interaktionsmodell) herleiten können.

Die vorgestellte Methode orientiert sich an

- OMT (Rumbaugh et al.)
- "Uses Cases" nach OOSE (Jacobson et al.)

Ziel: Präzise und verständliche Beschreibung der Anforderungen.

Schritte dazu:

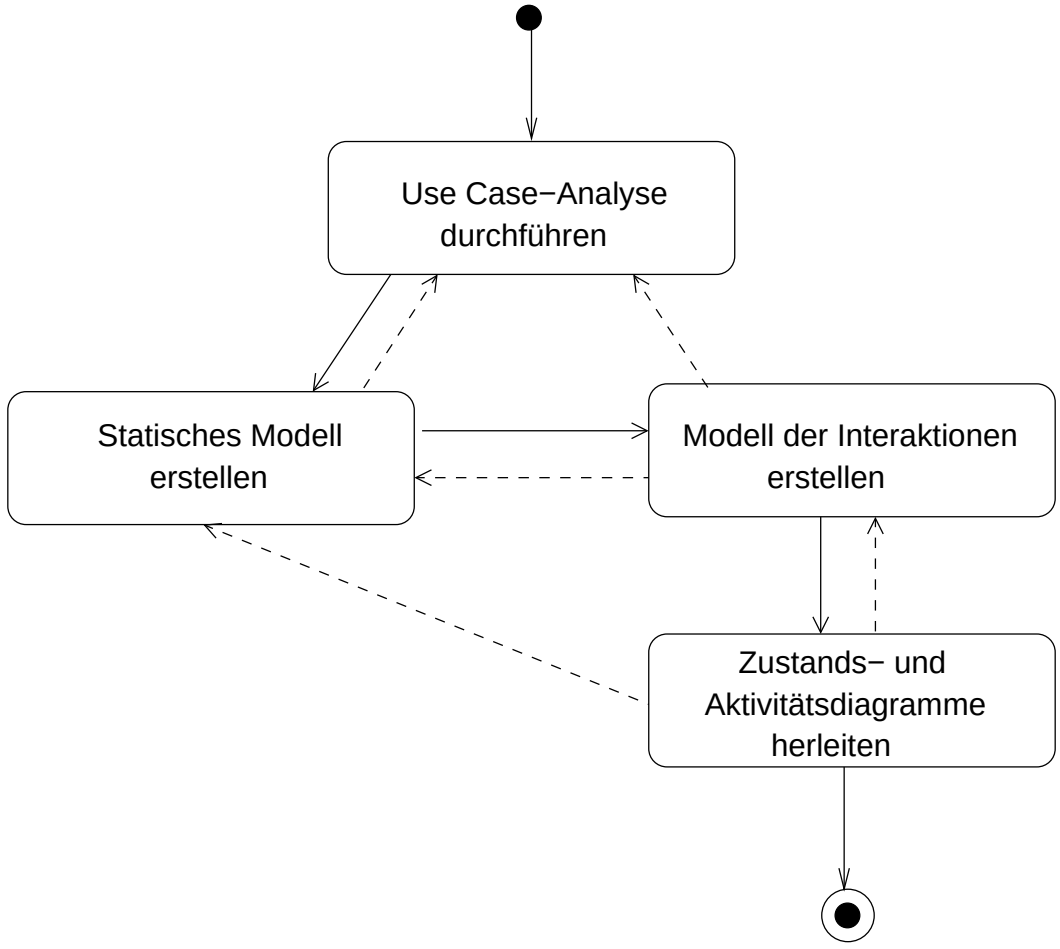
1. "Use Case"-Analyse
2. Entwicklung eines statischen Modells (in Form von Klassendiagrammen)
3. Entwicklung eines dynamischen Modells
   (in Form von Interaktionsdiagrammen, Zustands- und Aktivitätsdiagrammen)
4. Validieren, überarbeiten und erweitern der Modelle (in mehreren Iterationen)

Bemerkung

Während der Analysephase kann zusätzlich

- eine Grobarchitektur des Systems erstellt werden
- die Benutzerschnittstelle skizziert werden

Prozessmodell der objektorientierten Analyse



3.1 Anwendungsfall-Analyse

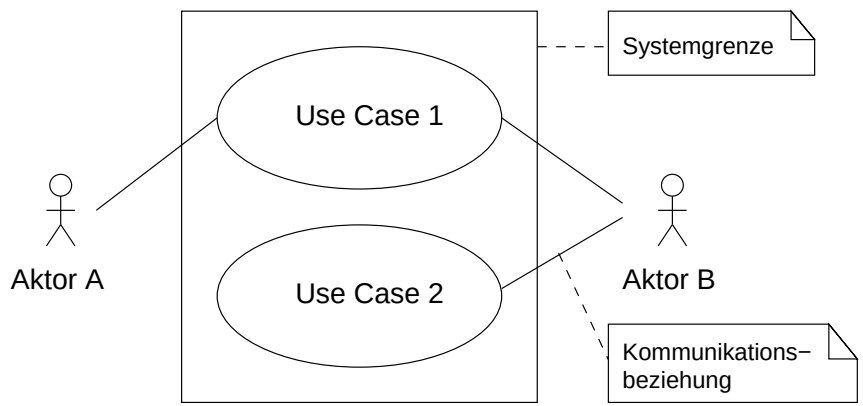
Ausgangspunkt: informelle, möglichst knappe Problembeschreibung

Ziel:
Beschreibung der gewünschten Funktionalität des zu entwickelnden Systems.

3.1.1 Use Case-Modell

- Besteht aus *Aktoren* und *Use Cases* (*Anwendungsfällen*).
- Beschreibt eine externe Sicht auf das System.

Darstellungsform:



Aktoren

- Tauschen von außen Informationen mit dem System aus (Benutzer, andere Systeme, Geräte).
- Aktoren werden durch die *Rolle*, die ein Benutzer gegenüber dem System einnimmt, charakterisiert.

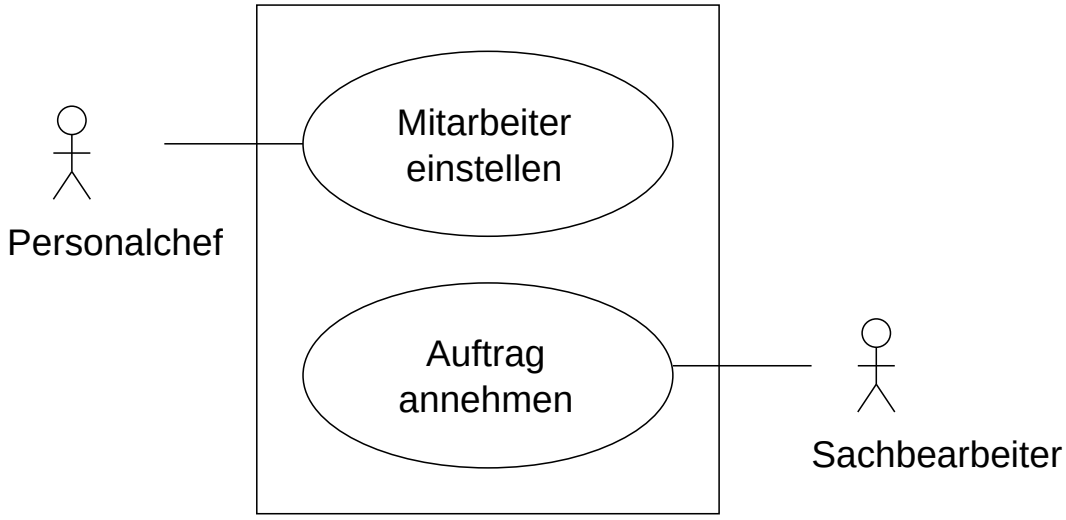
Anwendungsfall

- Beschreibt eine funktionale Anforderung an ein System.
- Beschreibt die Interaktionen zwischen einem (oder mehreren) Aktoren und dem System bei der Bearbeitung einer bestimmten, abgegrenzten Aufgabe.

Definition nach Jacobson:

Ein Anwendungsfall ist eine Menge von verhaltensverwandten Sequenzen von Transaktionen, die durch ein System ausgeführt werden und ein messbares Ergebnis liefern.

Beispiel:



Beachte:

Häufig sind Use Cases die computergestützten Teile von Geschäftsprozessen.

3.1.2 Vorgehensweise bei der Erstellung eines Use Case-Modells

1. Bestimmung der *Aktoren*, die mit dem System interagieren.
   (Wer benützt das System? Wer holt/liefert Informationen von dem/für das System?)

2. Bestimmung der *Anwendungsfälle* aufgrund der Aufgaben, die das System für jeden einzelnen Aktor erledigen soll (z.B. durch Interviews).
   Schwierigkeit: richtige Granularität finden

3. Erstellung eines *Anwendungsfall-Diagramms*, ggf. mit kurzer Beschreibung der Aktoren und Use Cases.

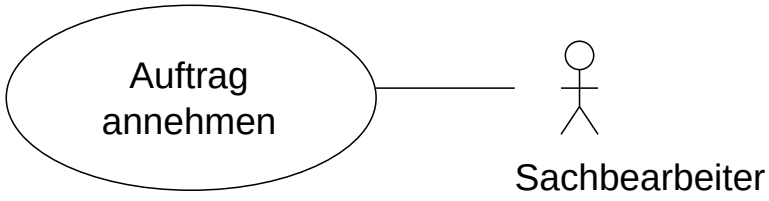
4. Beschreibung der Anwendungsfälle (iterativ).

Eine *Anwendungsfall-Beschreibung* besteht aus:

- Name des Anwendungsfalls
- Kurzbeschreibung
- Vorbedingung
  (Voraussetzung für eine erfolgreiche Ausführung des Anwendungsfalls)
- Nachbedingung
  (Zustand nach erfolgreicher Ausführung)
- einem Standardablauf (Primärszenario)
  (Schritte bzw. Interaktionen, die im Normalfall bei Ausführung des Anwendungsfalls durchlaufen werden)
- mehreren Alternativabläufen (Sekundärszenarien)
  (bei Fehlerfällen und Optionen)

Zusätzlich kann ein Aktivitätsdiagramm für den Anwendungsfall angegeben werden.

Beispiel:



Anwendungsfall: Auftrag annehmen

Kurzbeschreibung:
Ein Sachbearbeiter nimmt für einen Kunden eine Bestellung von Artikeln auf.

Vorbedingung:
Das System ist bereit einen neuen Auftrag anzunehmen.

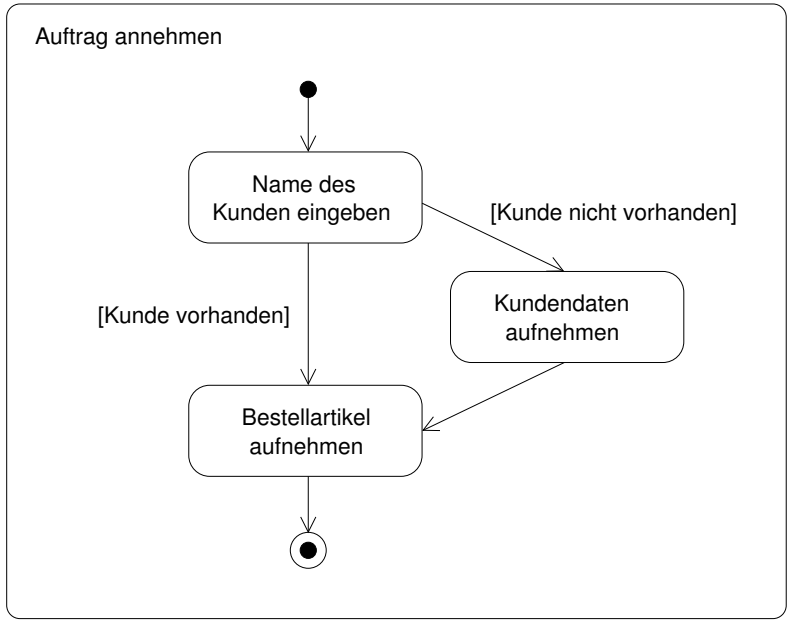
Nachbedingung:
Der Auftrag ist angenommen.

Primärszenario:

1. Der Sachbearbeiter gibt den Namen des Kunden ein.
2. Das System zeigt die Kundendaten an.
3. Der Sachbearbeiter gibt die Daten für die zu bestellenden Artikel ein.

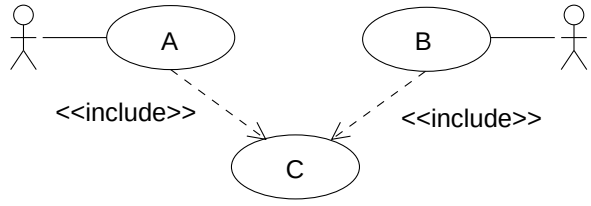
Sekundärszenarien:
Kunde nicht vorhanden

Aktivitätsdiagramm:



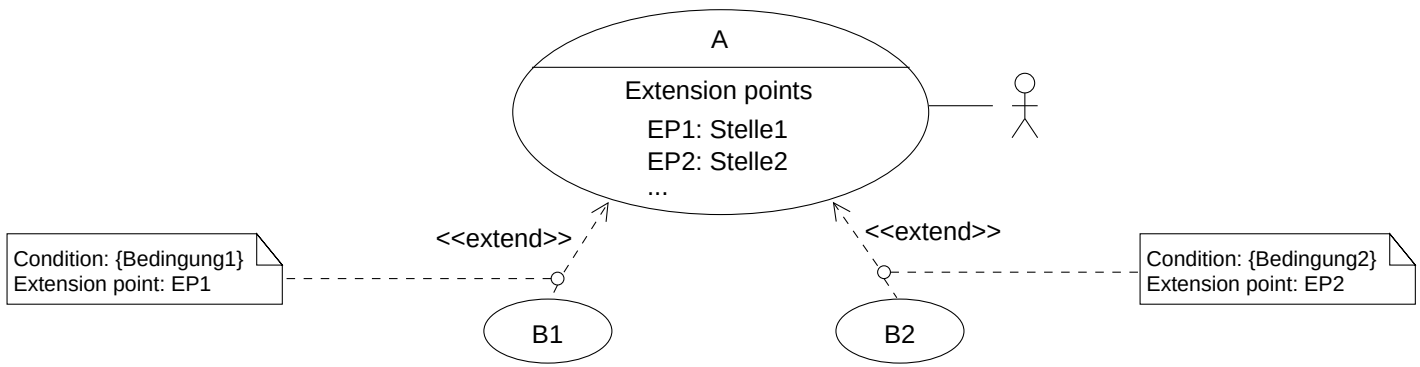
3.1.3 Beziehungen zwischen Anwendungsfällen

1. *Enthält-Beziehung*



Jeder Ablauf von A bzw. B beinhaltet als Teilablauf (notwendigerweise) einen Ablauf von C.

2. *Erweiterungsbeziehung*



Erweitert A um zusätzlich mögliches Verhalten, falls die angegebene Bedingung erfüllt ist.

3.1.4 Beispiel ATM (Automatic Teller Machine)

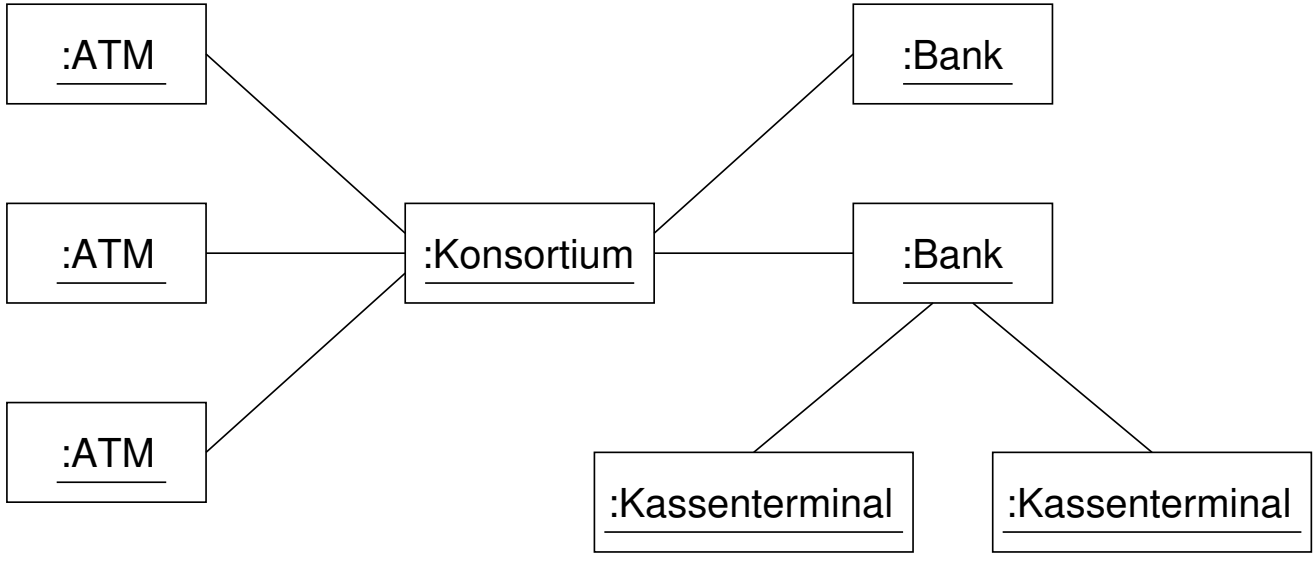
Problembeschreibung: Netzwerk von Bankautomaten
(aus Rumbaugh et al., 1993)

Entwickelt werden soll Software zur Unterstützung eines rechnergesteuerten Bankennetzwerks einschließlich Kassierern und Bankautomaten (ATMs), das sich ein Bankenkonsortium teilt. Jede Bank besitzt einen eigenen Computer, auf dem sie ihre Konten verwaltet und die Transaktionen auf Konten durchführt. Die Banken besitzen Kassenterminals, die direkt mit dem bankeigenen Computer kommunizieren.

Kassierer geben Konto- und Transaktionsdaten ein. ATMs kommunizieren mit einem Zentralrechner, der Transaktionen mit den jeweiligen Banken abklärt. Ein ATM akzeptiert eine Scheckkarte, interagiert mit dem Benutzer, kommuniziert mit dem zentralen System, um die Transaktion auszuführen, gibt Bargeld aus und druckt Belege.

Das System erfordert geeignete Aufzeichnungsmöglichkeiten und Sicherheitsmaßnahmen. Das System muss parallele Zugriffe auf das gleiche Konto korrekt abwickeln. Die Banken stellen die SW für ihre eigenen Computer selbst bereit.

Sie sollen die Software für die ATMs und das Netzwerk entwickeln. Die Kosten des gemeinsamen Systems werden nach Zahl der Scheckkarteninhaber auf die Banken umgelegt.

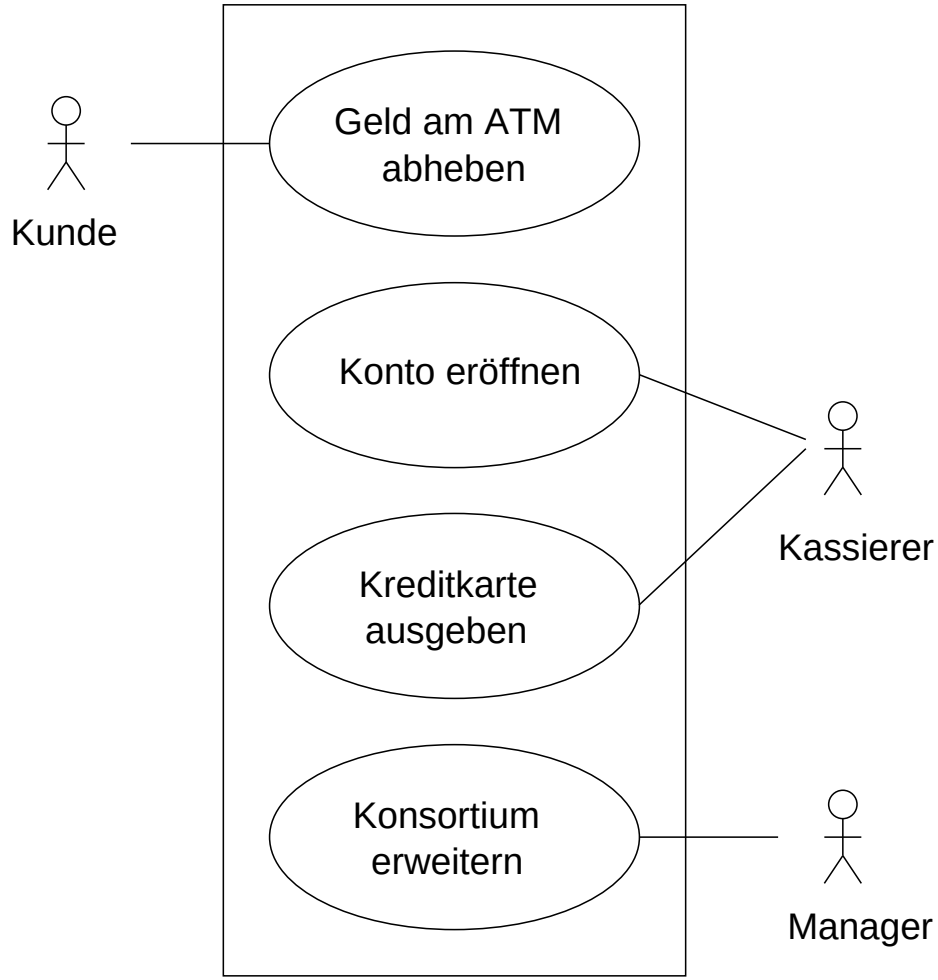


Vorgehensweise

1. *Aktoren:* Kunde, Kassierer, Manager, ...
2. *Use Cases:*
   - Geld am ATM abheben:
     Ein Benutzer hebt am Geldautomaten mit Hilfe seiner Kreditkarte Geld von seinem Konto ab.
   - Konto eröffnen:
     Ein Kassierer richtet ein neues Konto für einen Kunden ein.
   - Neue Kreditkarte ausgeben:
     Ein Kassierer gibt eine neue Kreditkarte für einen Kunden aus.
   - Konsortium erweitern:
     Ein Manager des Konsortiums nimmt eine Bank in das Konsortium auf.

   etc. für weitere Anwendungsfälle wie z.B. "Geld einzahlen", "Überweisung durchführen".

3. *Use Case-Diagramm* (Ausschnitt):



4. *Use Case Beschreibungen:*

Anwendungsfall:
Geld am ATM abheben.

Kurzbeschreibung:
Ein Benutzer hebt am Geldautomaten mit Hilfe seiner Kreditkarte Geld von seinem Konto ab.

Vorbedingung:
Das ATM ist bereit für einen Benutzer eine Transaktion durchzuführen.

Nachbedingung:
Der Benutzer hat die Kreditkarte, das Geld und den Beleg entnommen. Das ATM ist erneut bereit eine Transaktion durchzuführen.

Primärszenario (Standardablauf):

1. Der Kunde gibt seine Kreditkarte ein.
2. Das ATM liest die Kreditkarte und fordert daraufhin die Geheimzahl an.
3. Der Benutzer gibt die Geheimzahl ein.
4. Das ATM liest die Geheimzahl, überprüft sie und lässt dann die BLZ und die Kartennummer beim Konsortium überprüfen.
5. Das Konsortium überprüft die BLZ, gleicht die Kartennummer mit der Bank des Kunden ab und gibt dem ATM sein OK.
6. Das ATM fordert den Benutzer auf die Transaktionsform (Abhebung, Einzahlung, Überweisung, Kontoauszug) zu wählen.
7. Der Benutzer wählt "Abhebung", woraufhin das ATM den Betrag erfragt.
8. Der Benutzer gibt den gewünschten Betrag ein.
9. Das ATM liest den Betrag, überprüft, ob er innerhalb vordefinierter Grenzen liegt, und fordert dann das Konsortium auf die Transaktion zu verarbeiten.
10. Das Konsortium leitet die Anforderung an die Bank weiter, die den Kontostand aktualisiert und die Ausführung bestätigt.
11. Das Konsortium teilt dem ATM den erfolgreichen Abschluss der Transaktion mit.
12. Das ATM gibt den gewünschten Betrag Bargeld aus und fordert den Benutzer auf es zu entnehmen.

13. Der Benutzer entnimmt das Bargeld, woraufhin das ATM fragt, ob der Benutzer eine weitere Transaktion durchführen will.
14. Der Benutzer verneint.
15. Das ATM druckt einen Beleg, gibt die Karte aus und fordert den Benutzer auf sie zu entnehmen.
16. Der Benutzer entnimmt den Beleg und die Karte.
17. Das ATM fordert den nächsten Benutzer auf eine Kreditkarte einzugeben.

Sekundärszenarien (abweichende Fälle):

Karte gesperrt
In Schritt 5, wenn die Bank feststellt, dass die Karte gesperrt ist, teilt sie dies dem Konsortium mit. Das Konsortium leitet die Nachricht an das ATM weiter. Das ATM meldet dem Benutzer den Fehler. Der Use Case wird dann bei Schritt 15 fortgesetzt.

Transaktion gescheitert
In Schritt 10, wenn die Bank feststellt, dass der Kreditrahmen überschritten wird, teilt sie dem Konsortium mit, dass die Banktransaktion gescheitert ist. Das Konsortium leitet die Nachricht an das ATM weiter. Das ATM meldet dem Benutzer das Scheitern der Transaktion. Der Use Case wird ab Schritt 6 wiederholt.

Abbruch durch den Benutzer

Karte nicht lesbar

Falsche Geheimzahl

Falsche Bankleitzahl

Grenzen überschritten

Karte abgelaufen

Kein Geld im ATM

Netzwerk unterbrochen

Zusammenfassung von Abschnitt 3.1

- Ein Use Case-Modell besteht aus Anwendungsfällen und Aktoren.
- Ein Aktor tauscht Informationen mit dem System aus.
- Ein Anwendungsfall beschreibt eine Aufgabe, die das System für einen oder mehrere Aktoren durchführen soll.
- Eine Anwendungsfall-Beschreibung beinhaltet u.a. ein Primärszenario und mehrere Sekundärszenarien.
- Anwendungsfälle können mit <<include>> und <<extend>>-Beziehungen strukturiert werden.
- Ein Use Case-Modell wird schrittweise erstellt.

3.2 Entwicklung eines statischen Modells

Input

- Use Case-Modell
- Problembeschreibung
- Expertenwissen über Anwendungsbereich
- Allgemeinwissen

Ziel: Erstellung eines Klassendiagramms (noch ohne Operationen!)

Vorgehensweise (nach OMT)

1. Klassen identifizieren
2. Assoziationen bestimmen
3. Attribute identifizieren
4. Vererbung einführen
5. Modell überarbeiten

3.2.1 Klassen identifizieren

Kandidaten sind

- Personen bzw. Rollen (z.B. Student, Angestellter, Kunde, ...)
- Organisationen (z.B. Firma, Abteilung, Uni, ...)
- Gegenstände (z.B. Artikel, Flugzeug, Immobilie, ...)
- begriffliche Konzepte (z.B. Bestellung, Vertrag, Vorlesung, ...)

1. Schritt: Kandidaten notieren

Möglichkeit: Durchsuche die Use Case-Beschreibungen nach Substantiven.

2. Schritt: Ungeeignete Klassen streichen

Streichungskriterien:

- redundante Klassen
- irrelevante Klassen
- vage Klassen
- Attribute oder Attributwerte
- Operationen (Tätigkeiten)

3. Schritt: Fehlende relevante Klassen hinzunehmen

Hinweise für fehlende Klassen:

- Attribute aus Schritt 2, zu denen es bisher noch keine Klassen gibt
- Problembereichswissen

Beispiel ATM:

1. Schritt:

Substantive im (Primärszenario des) Use Case “Geld am ATM abheben” notieren

Kunde

Kreditkarte

ATM

Geheimzahl

Benutzer

Bankleitzahl

Kartennummer

Konsortium

Bank

Transaktionsform

Abhebung, Einzahlung, Überweisung, Kontoauszug

Betrag

Grenzen

Transaktion

Abschluss

Ausführung

Anforderung

Kontostand

Bargeld

Beleg

Karte

2. Schritt: Ungeeignete Klassen streichen

Kunde

Kreditkarte

ATM

Geheimzahl *(Attribut)*

Benutzer *(redundant)*

Bankleitzahl *(Attribut)*

Kartennummer *(Attribut)*

Konsortium

Bank

Transaktionsform *(Attribut)*

Betrag *(Attribut)*

Grenzen *(Attribut)*

Transaktion

Abschluss *(Tätigkeit)*

Ausführung *(Tätigkeit)*

Anforderung *(Tätigkeit)*

Kontostand *(Attribut)*

Bargeld *(irrelevant)*

Beleg *(vage)*

Karte *(redundant)*

Abhebung, Einzahlung, Überweisung, Kontoauszug *(Attributwert von Transaktionsform)*

3. Schritt: Fehlende Klassen hinzunehmen

- Konto (*folgt aus dem Attribut Kontostand*)
- Kassierer
- Kassenterminal
- Kassierertransaktion
- Aussentransaktion (*statt der o.g. Transaktion*)

3.2.2 Assoziationen identifizieren

Kandidaten sind physische oder logische Verbindungen mit einer bestimmten Dauer, wie

- konzeptionelle Verbindungen (arbeitet für, ist Kunde von, ...)
- Besitz (hat, ist Teil von, ...)
- (häufige) Zusammenarbeit von Objekten zur Lösung einer Aufgabe

1. Schritt: Kandidaten notieren

Möglichkeit: Überprüfe

- Verben
- Substantive im Genitiv (z.B. die Kreditkarte des Kunden)
- Possesivpronomen (z.B. seine Kreditkarte)

Beachte:

- Verben bezeichnen häufig Aktivitäten und keine Beziehungen (z.B. ATM überprüft die Geheimzahl).
- Falls jedoch in einer Aktivität mehrere Objekte gebraucht werden, kann dies ein Hinweis auf eine häufige Zusammenarbeit sein, die eine Assoziation voraussetzt (z.B. ATM lässt Bankleitzahl beim Konsortium überprüfen).
- Assoziationen sind i.a. schwieriger zu bestimmen als Klassen. Wissen über den Problembereich (und Allgemeinwissen) ist wichtig!

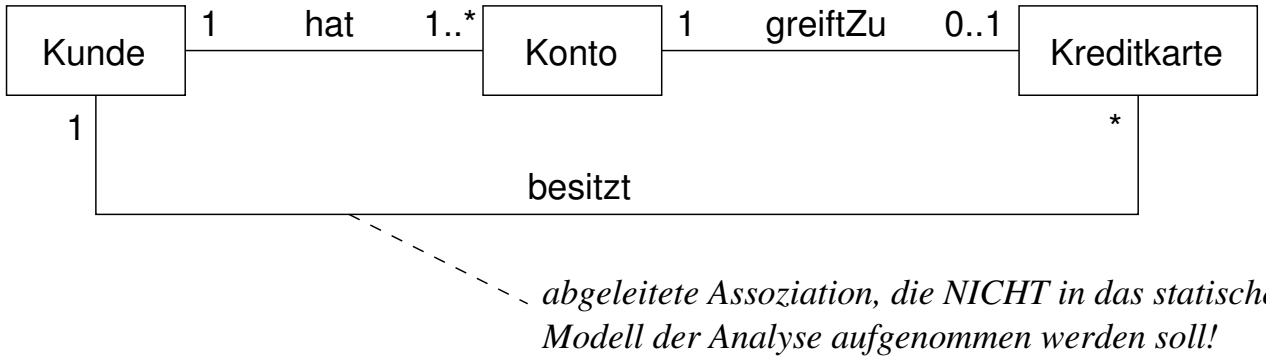
Beispiel ATM:

- Kunde besitzt Kreditkarte (... seine Kreditkarte ...)
- Konsortium besitzt ATM (... lässt überprüfen ...)
- Konsortium besteht aus Banken (... gleicht ab ...)
- Bank führt Konten (Wissen über den Problembereich)
- Kunde hat Konten (Wissen über den Problembereich)

2. Schritt: Ungeeignete Assoziationen streichen

Kriterien wie bei Klassen, inbesondere auf redundante (abgeleitete) Assoziationen möglichst verzichten.

Beispiel ATM:



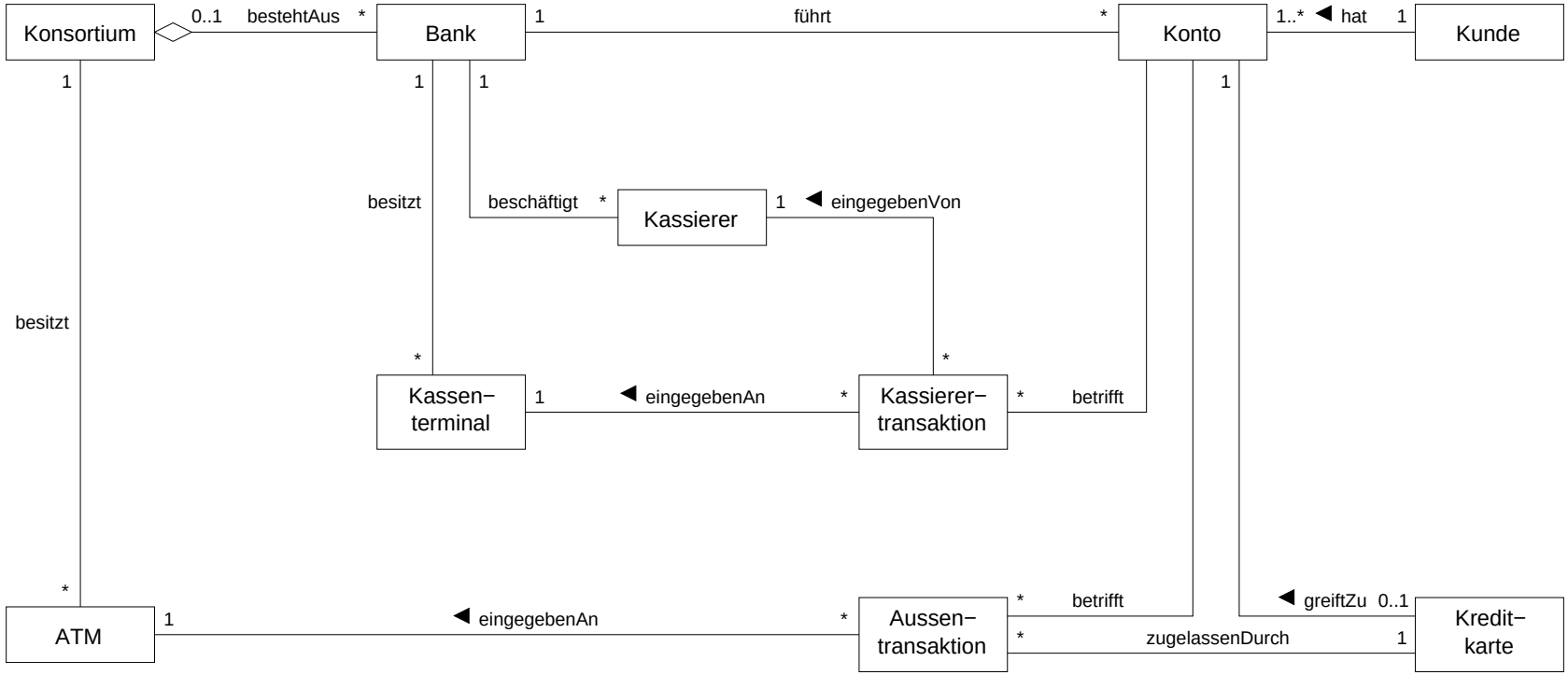
3. Schritt: Fehlende relevante Assoziationen hinzunehmen

4. Schritt: Multiplizitäten und ggf. explizite Rollennamen hinzufügen

Bemerkung

Multiplizitäten können in zweifelhaften Fällen noch weggelassen werden und erst bei der Überarbeitung des Modells bestimmt werden.

Beispiel ATM: Erstes Klassendiagramm



3.2.3 Attribute identifizieren

Richtlinien

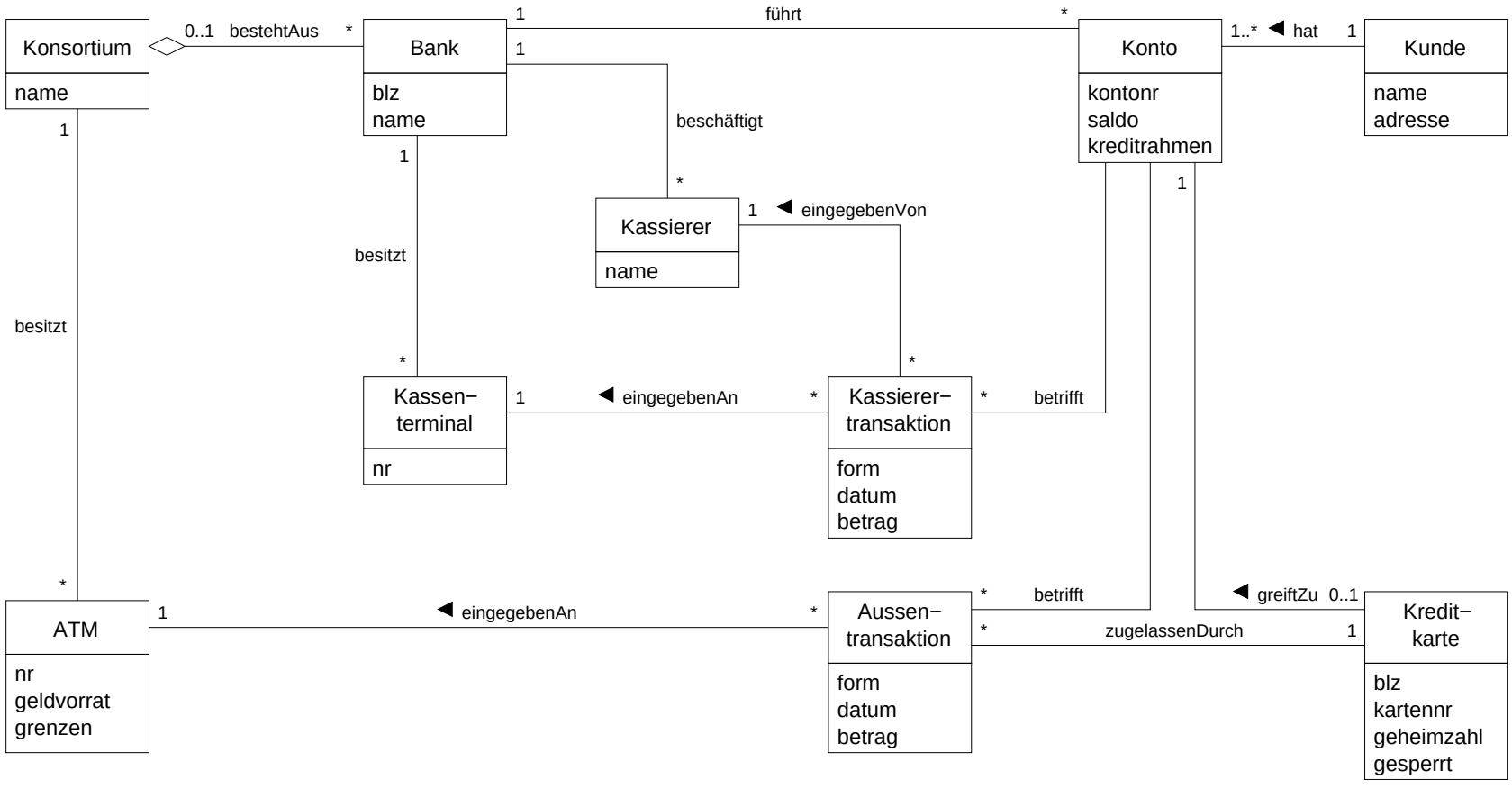
- nur Attribute, die für die Anwendung relevant sind (Problembereichswissen wichtig!)
- keine Attribute, deren Werte Objekte sind (dafür Assoziationen verwenden!)
- Attributtypen können noch weggelassen werden

Beispiel ATM:

Aus dem Primärszenario des Use Case "Geld am ATM abheben" ergeben sich die folgenden Attribute:

- blz ist Attribut von Bank und von Kreditkarte
- kartennr, geheimzahl sind Attribute von Kreditkarte
- form, betrag sind Attribute von Kassierertransaktion und Aussentransaktion
- grenzen ist Attribut von ATM
- saldo (Kontostand) ist Attribut von Konto

Beispiel ATM: Klassendiagramm mit Attributen



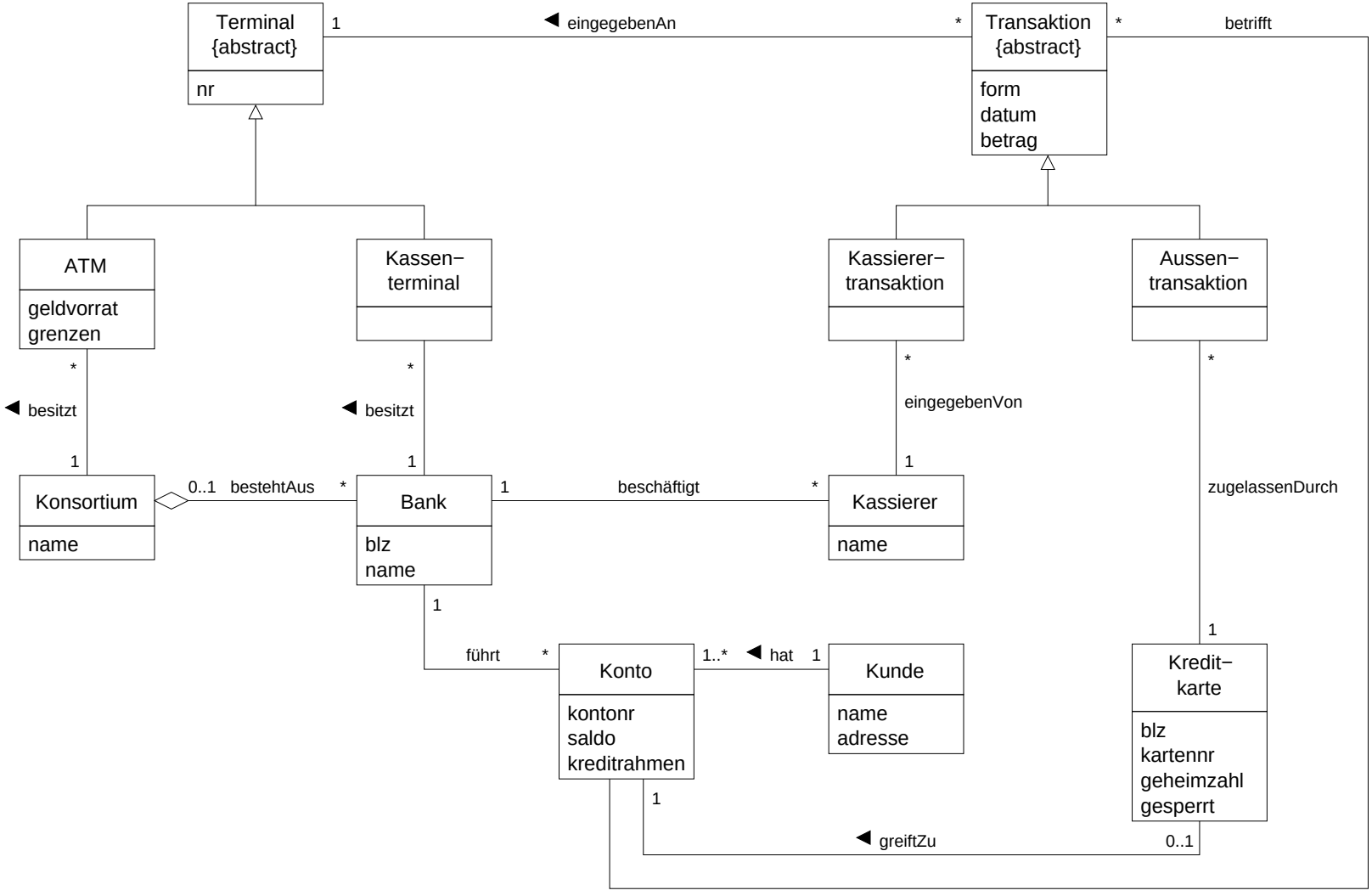
3.2.4 Vererbung einführen

- Vor allem **Generalisierung**!
- Zusammenfassen gemeinsamer Merkmale vorhandener Klassen (Attribute, Assoziationen) in einer Oberklasse.
- Bestimmen, welche Klassen abstrakt sind.

Beispiel ATM:

- Aussentransaktion und Kassierertransaktion haben alle Attribute gemeinsam und eine gemeinsame Assoziation zu Konto
  $\Rightarrow$ Generalisierung zur (abstrakten) Klasse Transaktion
- ATM und Kassenterminal können zur (abstrakten) Klasse Terminal generalisiert werden.

Beispiel ATM: Klassendiagramm mit Vererbung



3.2.5 Modell überarbeiten

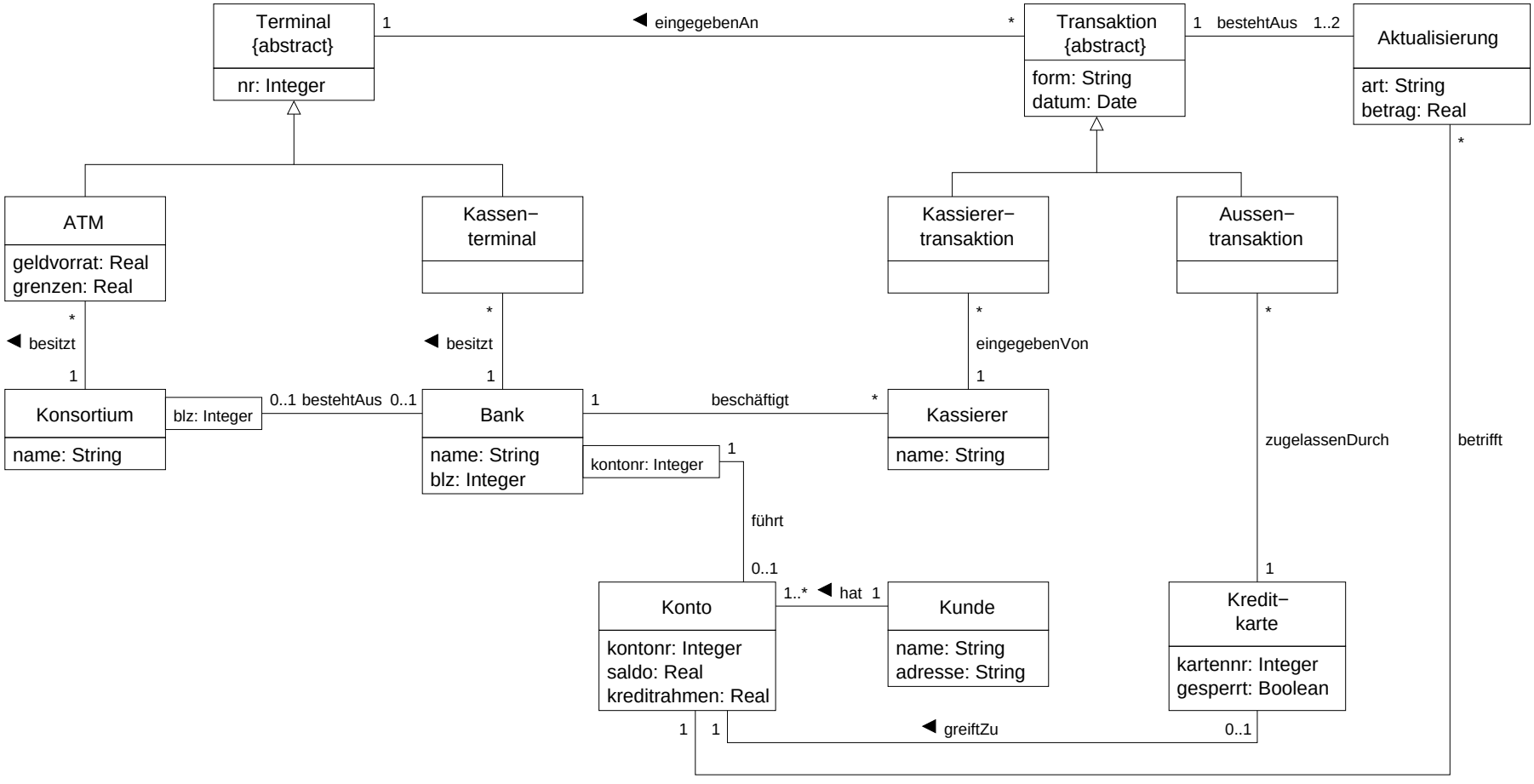
Fragen

- Fehlende Klassen oder Assoziationen?
- Unnötige Klassen oder Assoziationen?
- Falsche Platzierung von Assoziationen und Attributen?
- Qualifizierer für Assoziationen?
- Typen für Attribute?
- Fehlende Multiplizitäten?

Beispiel ATM:

- Eine Überweisungstransaktion betrifft zwei Konten
  $\Rightarrow$ Einführung der Klasse "Aktualisierung".
- Eine Kreditkarte ist ein Stück Plastik! In dem zugehörigen Software-Objekt ist die Geheimzahl *nicht* gespeichert. Ebensowenig die Bankleitzahl (wäre redundant).
- Qualifizierer für die Assoziationen zwischen Konsortium und Bank und zwischen Bank und Konto verwenden.

Beispiel ATM: Klassendiagramm nach Überarbeitung



Zusammenfassung von Abschnitt 3.2

- Das statische Modell beschreibt die strukturellen und datenorientierten Aspekte eines Systems.
- Das statische Modell wird durch Klassendiagramme (ggf. ergänzt um Objektdiagramme) dargestellt.
- Schritte bei der Entwicklung des statischen Modells sind:
  - Klassen identifizieren
  - Assoziationen bestimmen
  - Attribute identifizieren
  - Vererbung einführen
  - Modell überarbeiten

3.3 Modellierung von Interaktionen

Interaktion = spezifisches Muster der Zusammenarbeit und des Nachrichtenaustauschs zwischen Objekten zur Erledigung einer bestimmten Aufgabe (z.B. eines Anwendungsfalls).

Ziel:

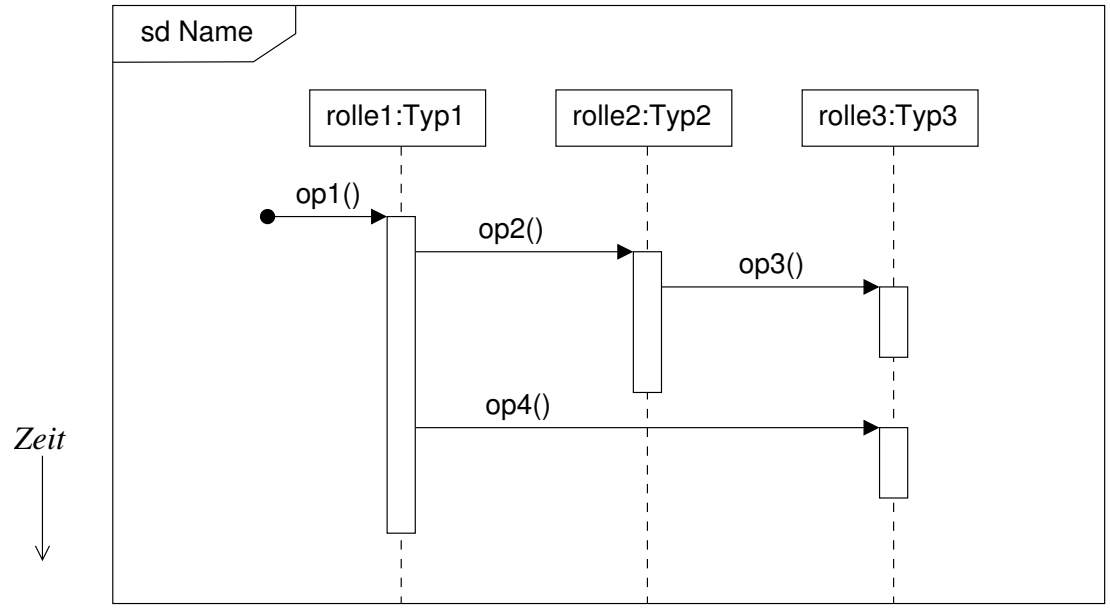
Entwurf einer Menge von *Interaktionsdiagrammen* für jeden Anwendungsfall.

Man unterscheidet zwei Arten von Interaktionsdiagrammen:

- Sequenzdiagramme
- Kommunikationsdiagramme

3.3.1 Sequenzdiagramme

Heben die *zeitliche Reihenfolge* hervor, in der Nachrichten zwischen Objekten ausgetauscht (d.h. gesendet und empfangen) werden.



Beachte:

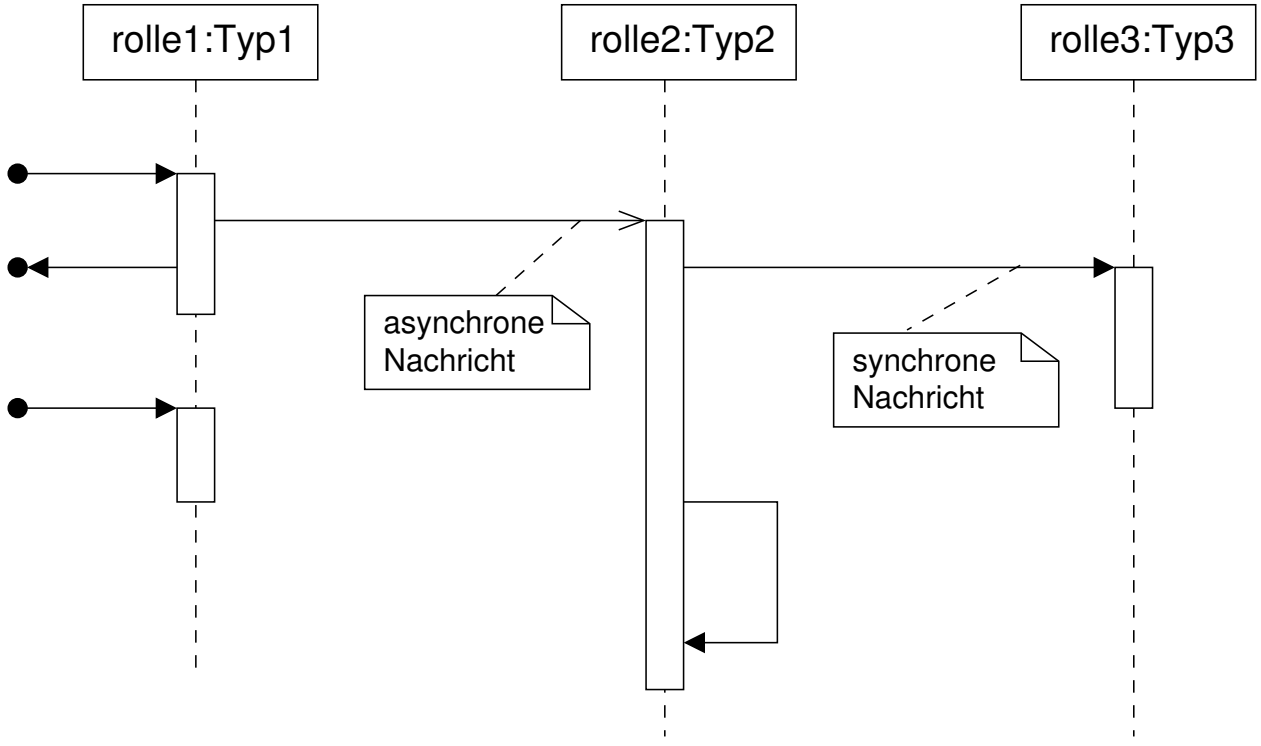
Empfangen einer Nachricht ist ein **Ereignis**.
Senden einer Nachricht ist eine **Aktion**.

Aktivierung

Zeitspanne, innerhalb der ein Objekt aktiv ist, weil es

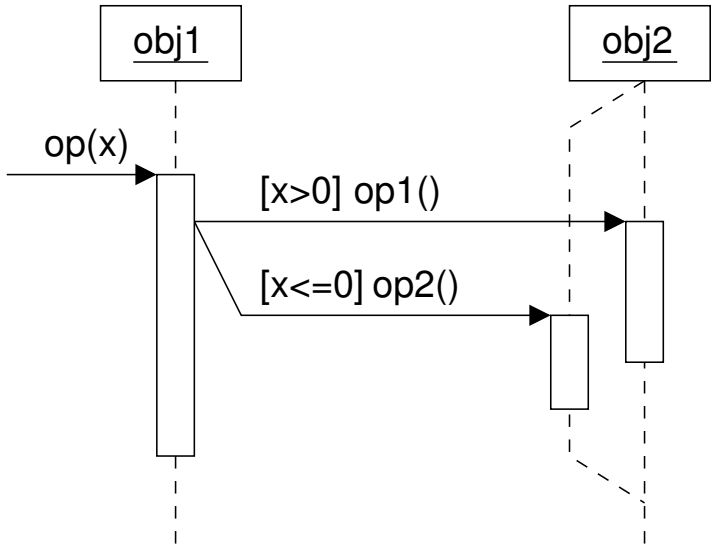
- gerade selbst tätig ist, oder
- auf die Beendigung einer Aktivierung eines (anderen) Objekts wartet, dem es eine (synchrone) Nachricht gesendet hat ("Rückgabe des Steuerungsfokus")

Asynchrone Nachrichten und parallele Ausführung

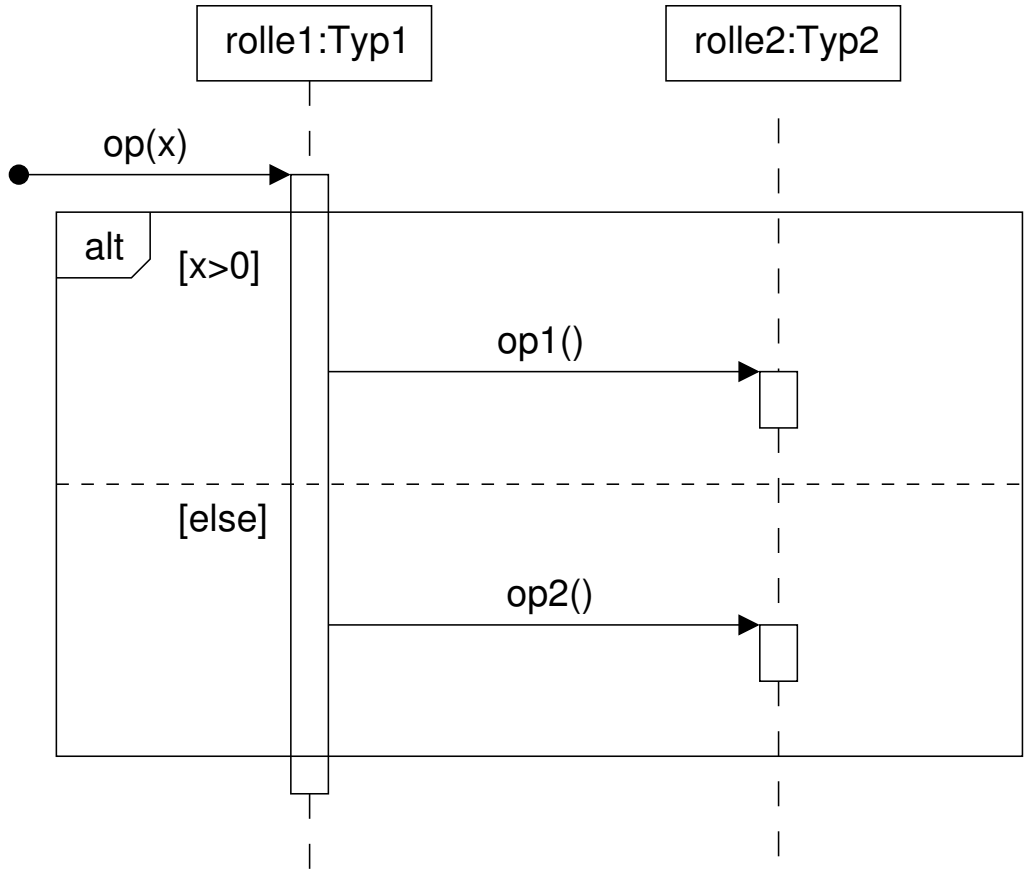


Das sendende Objekt setzt sofort nach dem Senden einer asynchronen Nachricht seine Tätigkeit fort (parallel zur Tätigkeit des Empfängers).

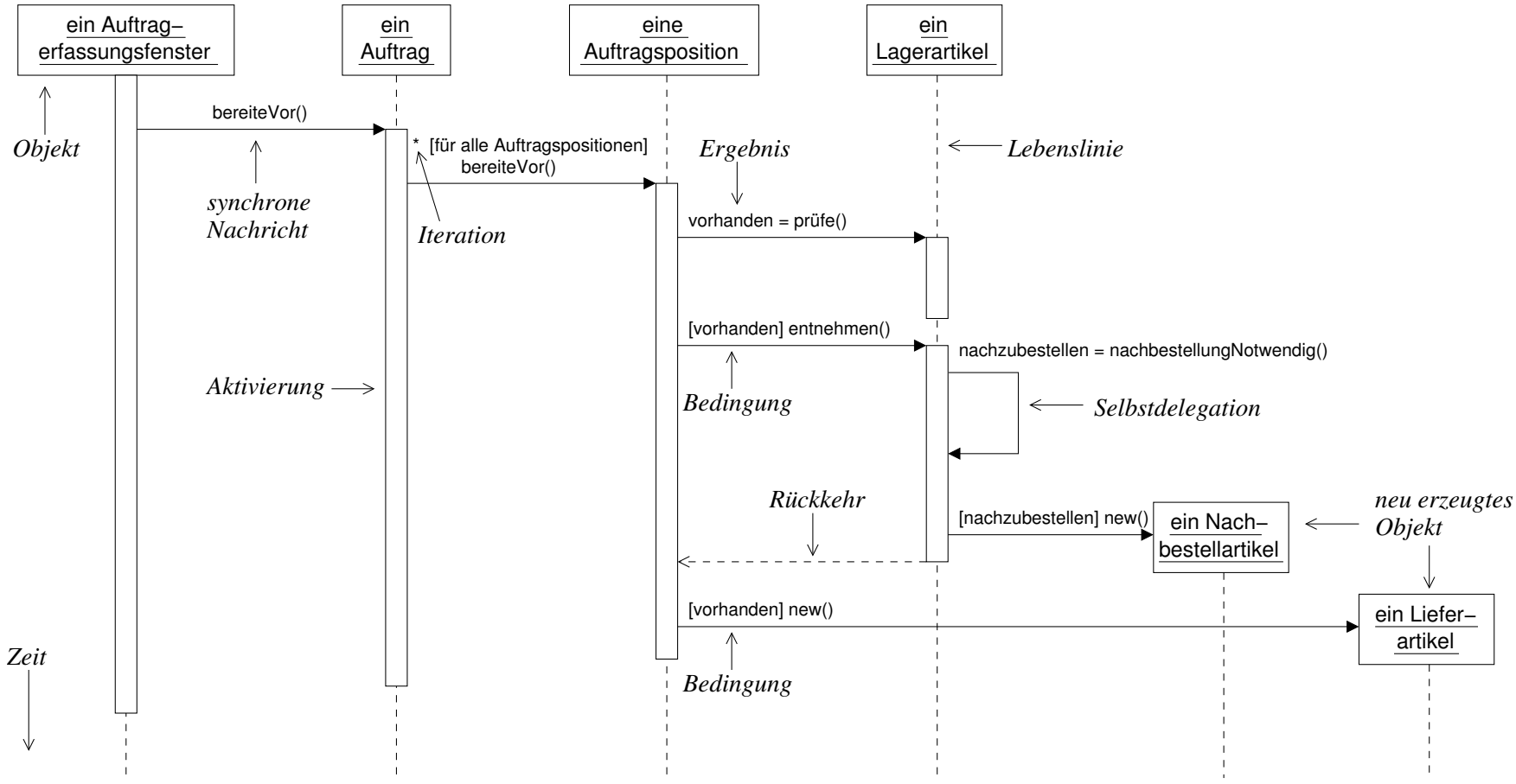
Alternativen (UML 1.X)



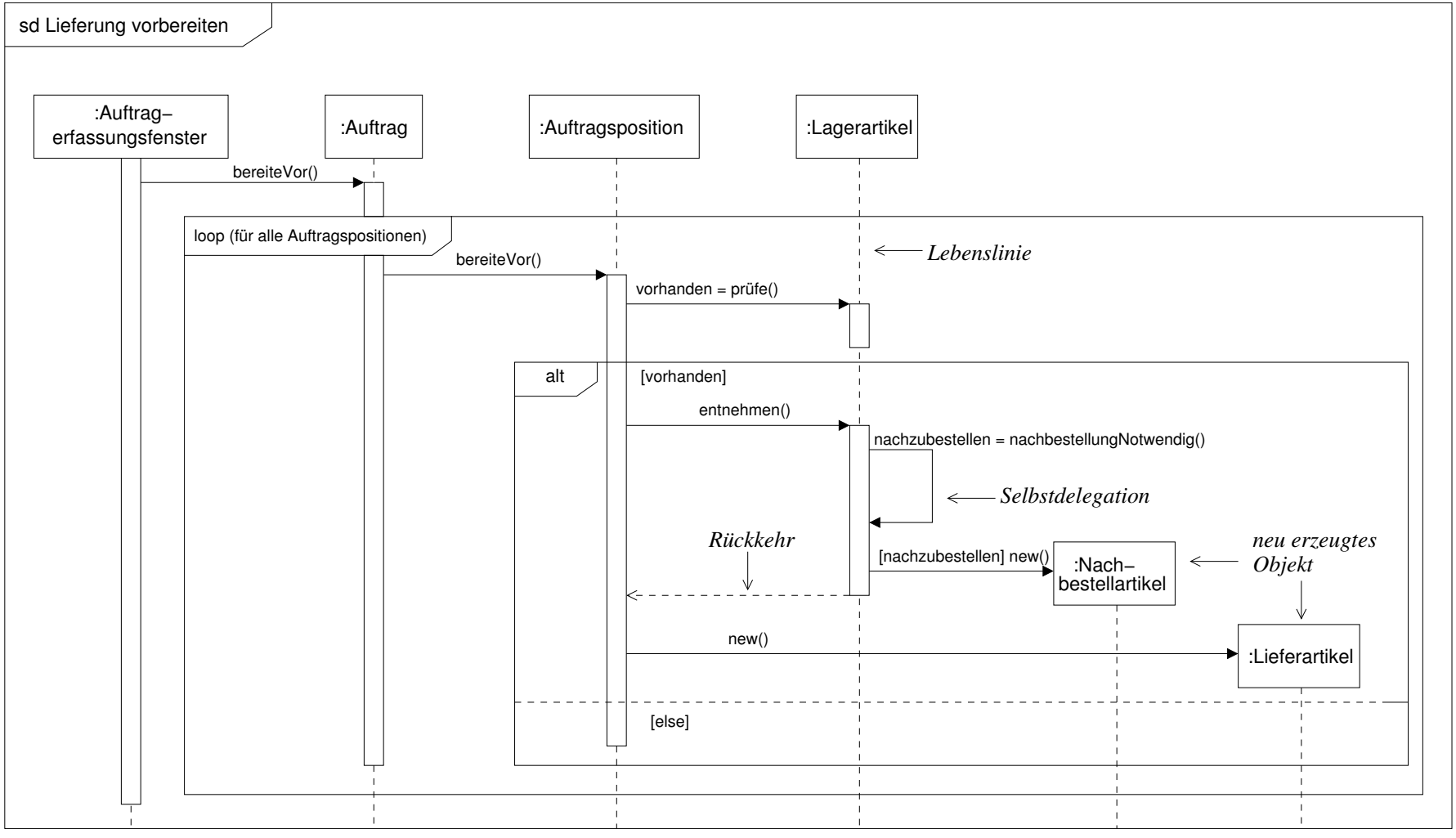
Alternativen (UML 2.0)



Beispiel (UML 1.X): Sequenzdiagramm für "Lieferung vorbereiten"

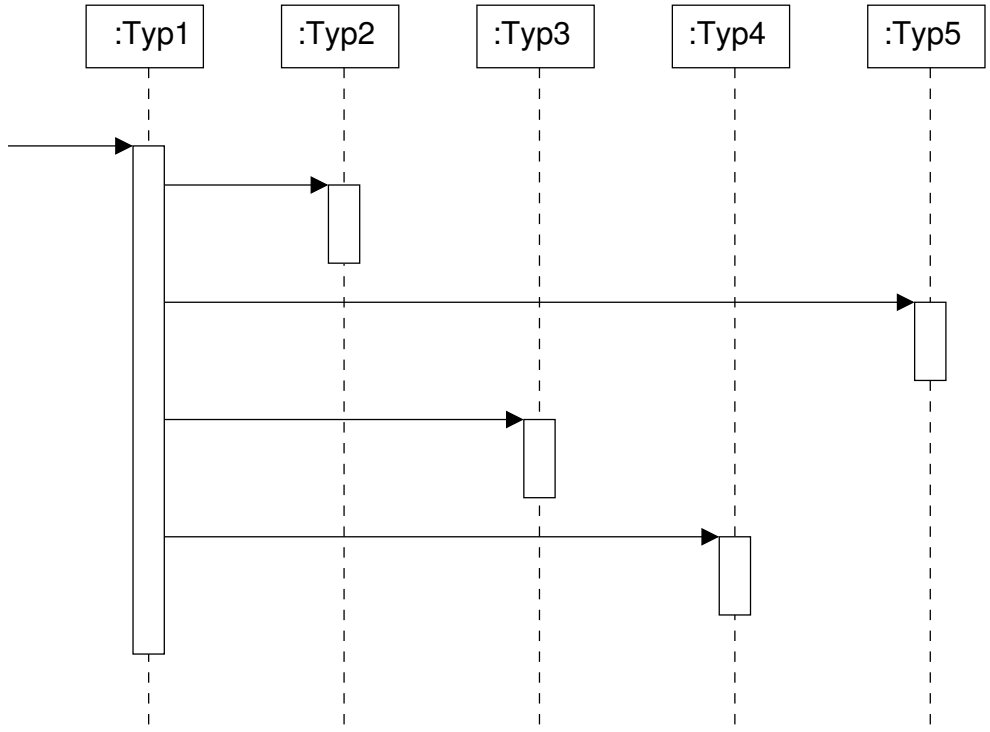


UML 2.0


sd Lieferung vorbereiten
:Auftrag–
erfassungsfenster
:Auftrag
:Auftragsposition
:Lagerartikel
bereiteVor()
loop (für alle Auftragspositionen)
bereiteVor()
Lebenslinie
vorhanden = prüfe()
alt
[vorhanden]
entnehmen()
nachzubestellen = nachbestellungNotwendig()
Selbstdelegation
Rückkehr
[nachzubestellen] new()
:Nach–
bestellartikel
neu erzeugtes
Objekt
new()
:Lieferartikel
[else]

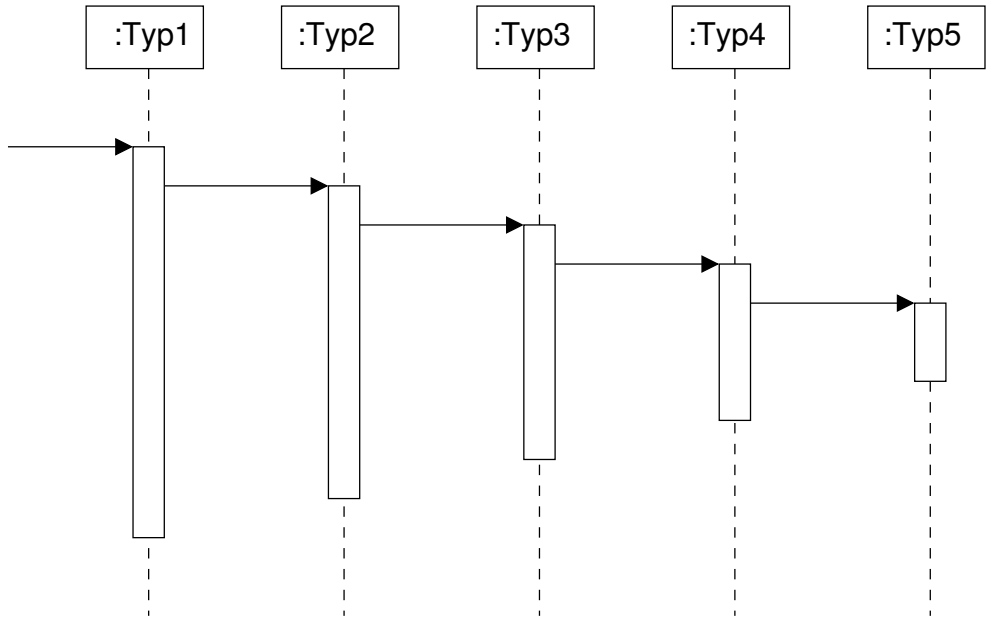
Typische Strukturen von Sequenzdiagrammen

Zentralisierte Struktur ("Fork")



Ein Objekt kontrolliert die anderen Objekte (besitzt die Verantwortung für die erfolgreiche Ausführung).

Dezentralisierte Struktur ("Stair")

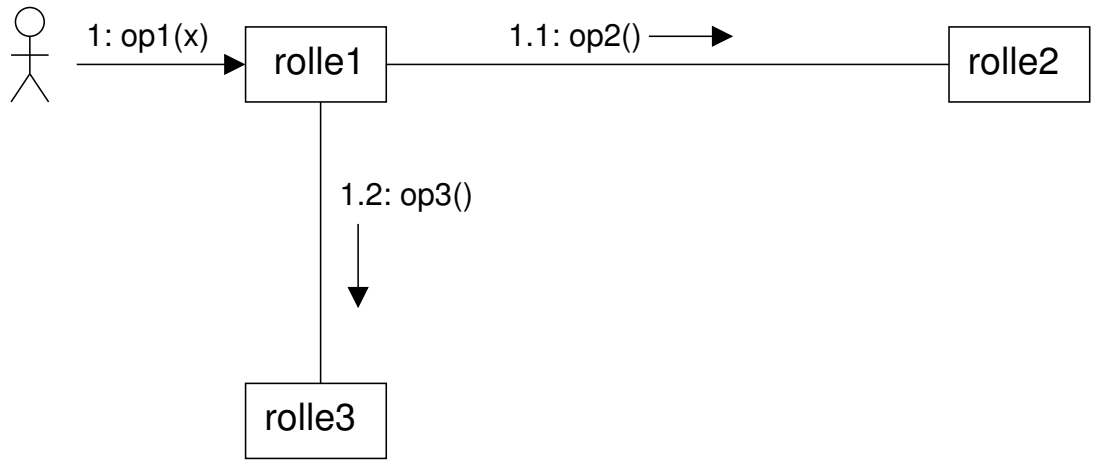


Jedes Objekt trägt eine eigene Verantwortung für die erfolgreiche Ausführung.

3.3.2 Kommunikationsdiagramme

Heben die *strukturellen Beziehungen* (dauerhafte und temporäre) aller an einer Interaktion beteiligten Objekte hervor.

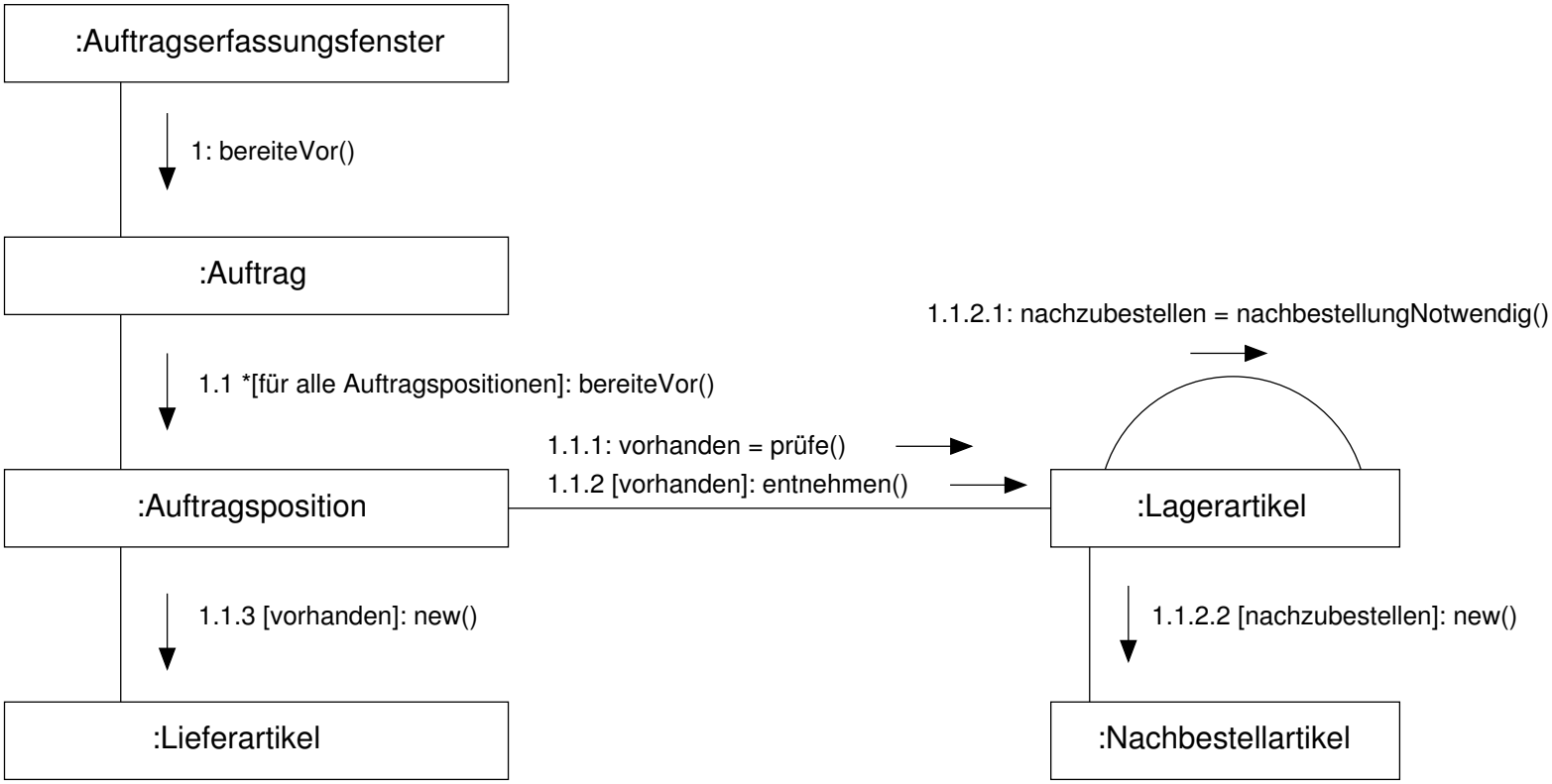
Die zeitliche Reihenfolge wird durch Nummerierung der Nachrichten dargestellt (mit dezimalen Nummern 1.1, 1.2 etc. für geschachtelte Nachrichten).



Bemerkung

Die Links in Kommunikationsdiagrammen sind Instanzen von Assoziationen oder temporäre Beziehungen.

Beispiel: Kommunikationsdiagramm für "Lieferung vorbereiten"



Bemerkung

Sequenz- und Kommunikationsdiagramme stellen im wesentlichen dieselbe Information in verschiedener Form dar.

3.3.3 Entwurf von Interaktionsdiagrammen

Input

- Use Case-Beschreibungen (Szenarien!)
- statisches Modell

Ziel

Modellierung der Zusammenarbeit von Objekten innerhalb eines Anwendungsfalls durch Interaktionsdiagramme.

Vorgehensweise

1. Identifiziere die Nachrichten, die innerhalb eines Anwendungsfalls ausgetauscht werden und die Objekte, die die Nachrichten senden und empfangen.
2. Konstruiere Interaktionsdiagramme für jeden Anwendungsfall.

Möglichkeiten dazu:

(A) Für jedes Szenario eines Use Case ein eigenes Interaktionsdiagramm.

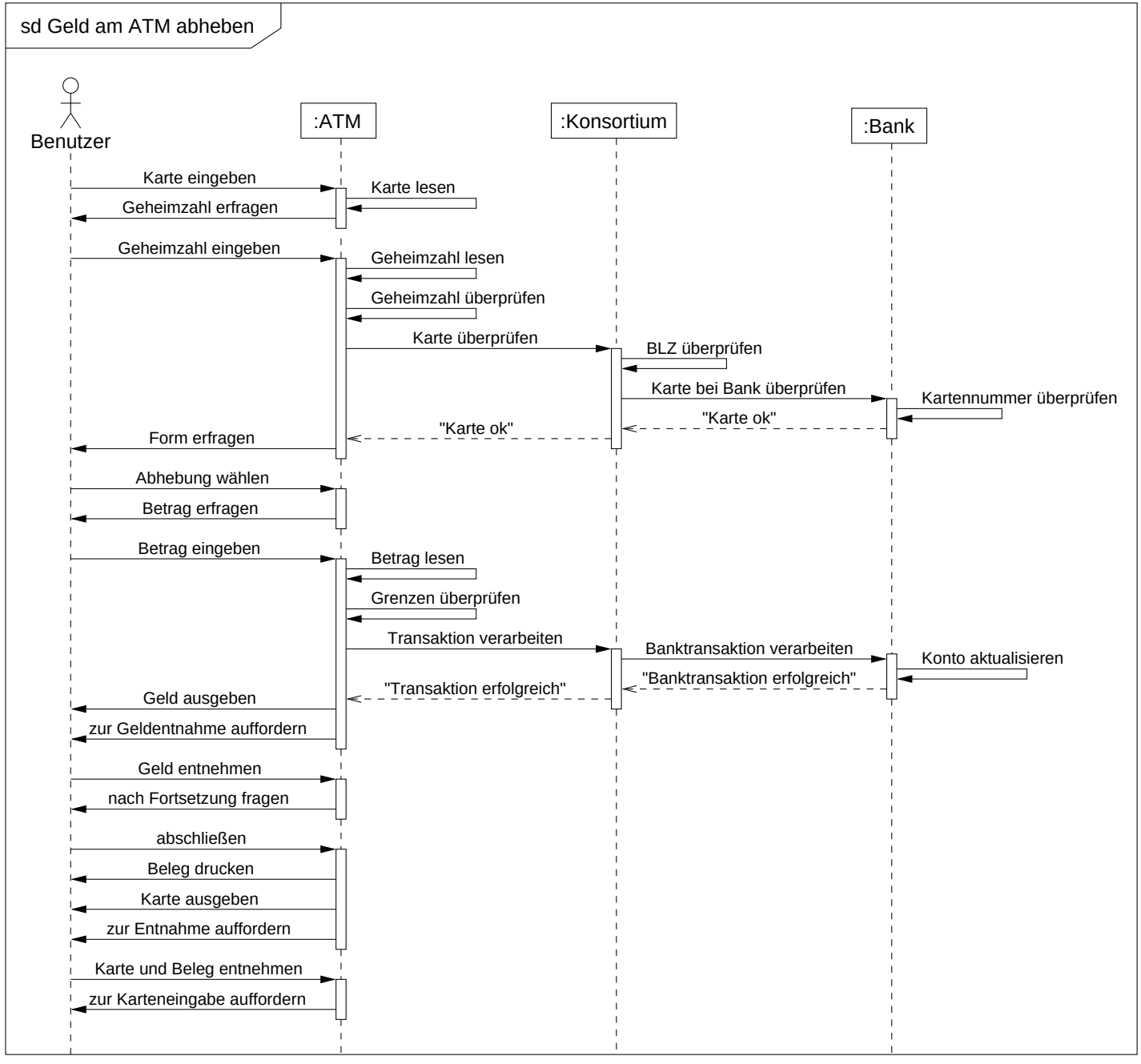
(B) Ein komplexes Interaktionsdiagramm (mit Alternativen, Iterationen (loops), etc.), das alle Szenarien eines Use Case subsummiert.
Nachteil: Wird schnell unübersichtlich!

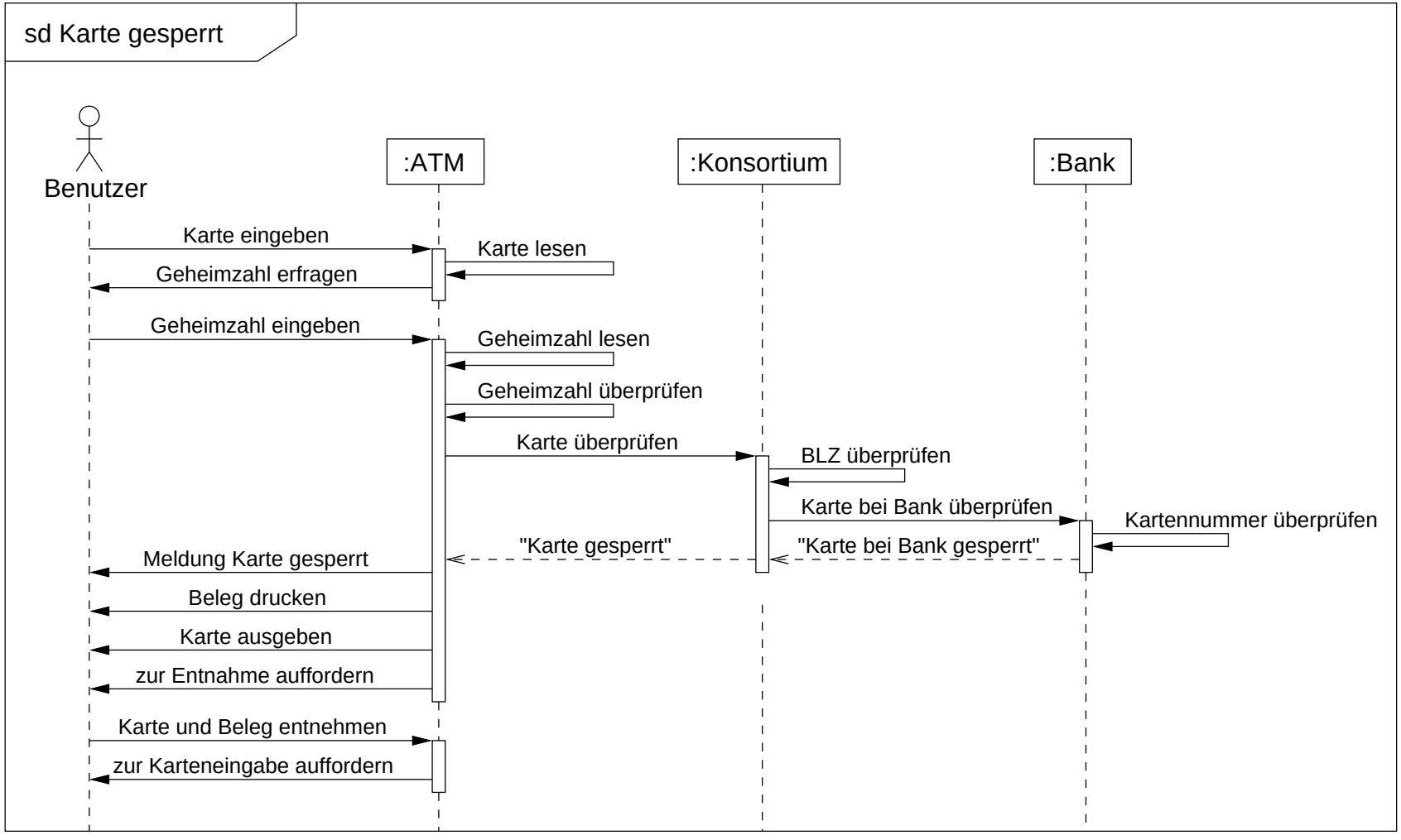
Wir orientieren uns an Möglichkeit (A) und verwenden Sequenzdiagramme (SDs).

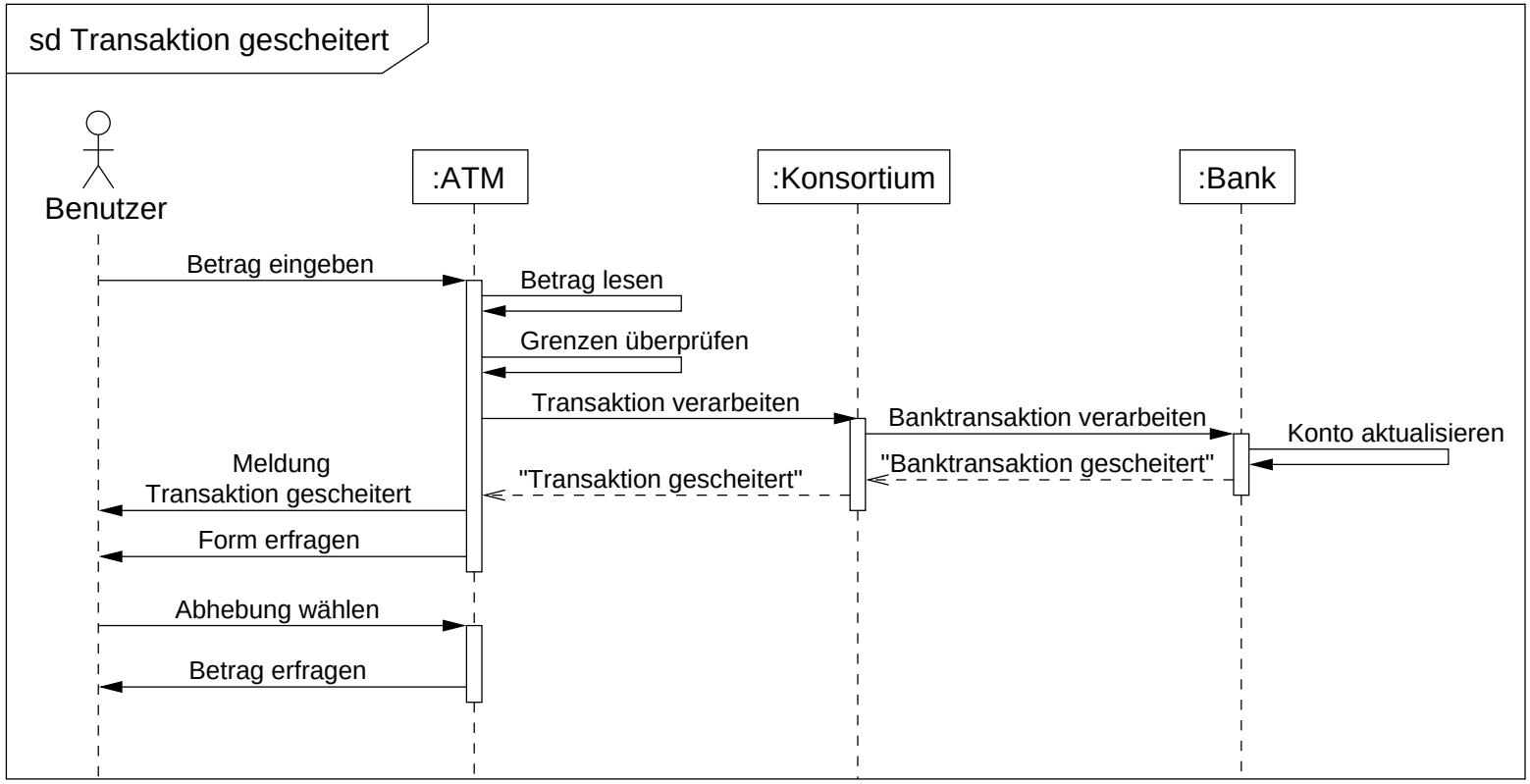
Beispiel: Use Case "Geld am ATM abheben"

Wir konstruieren:

- ein SD für das Primärszenario
- ein SD für das Sekundärszenario "Karte gesperrt"
- ein SD für das Sekundärszenario "Transaktion gescheitert"


sd Geld am ATM abheben
Benutzer
:ATM
:Konsortium
:Bank
Karte eingeben
Karte lesen
Geheimzahl erfragen
Geheimzahl eingeben
Geheimzahl lesen
Geheimzahl überprüfen
Karte überprüfen
BLZ überprüfen
Karte bei Bank überprüfen
Kartennummer überprüfen
"Karte ok"
"Karte ok"
Form erfragen
Abhebung wählen
Betrag erfragen
Betrag eingeben
Betrag lesen
Grenzen überprüfen
Transaktion verarbeiten
Banktransaktion verarbeiten
Konto aktualisieren
"Banktransaktion erfolgreich"
"Transaktion erfolgreich"
Geld ausgeben
zur Geldentnahme auffordern
Geld entnehmen
nach Fortsetzung fragen
abschließen
Beleg drucken
Karte ausgeben
zur Entnahme auffordern
Karte und Beleg entnehmen
zur Karteneingabe auffordern


sd Karte gesperrt
Benutzer
:ATM
:Konsortium
:Bank
Karte eingeben
Karte lesen
Geheimzahl erfragen
Geheimzahl eingeben
Geheimzahl lesen
Geheimzahl überprüfen
Karte überprüfen
BLZ überprüfen
Karte bei Bank überprüfen
Kartennummer überprüfen
"Karte gesperrt"
"Karte bei Bank gesperrt"
Meldung Karte gesperrt
Beleg drucken
Karte ausgeben
zur Entnahme auffordern
Karte und Beleg entnehmen
zur Karteneingabe auffordern


sd Transaktion gescheitert
Benutzer
:ATM
:Konsortium
:Bank
Betrag eingeben
Betrag lesen
Grenzen überprüfen
Transaktion verarbeiten
Banktransaktion verarbeiten
Konto aktualisieren
Meldung
Transaktion gescheitert
"Transaktion gescheitert"
"Banktransaktion gescheitert"
Form erfragen
Abhebung wählen
Betrag erfragen

Zusammenfassung von Abschnitt 3.3

- Interaktionsdiagramme beschreiben die Zusammenarbeit und den Nachrichtenaustausch zwischen *mehreren* Objekten.
- Wir unterscheiden Sequenzdiagramme (heben die zeitliche Reihenfolge hervor) und Kommunikationsdiagramme (heben die strukturellen Beziehungen hervor).
- Ein Modell der Interaktionen basiert auf den Anwendungsfall-Beschreibungen und dem statischen Modell.
- Pro Anwendungsfall werden i.a. mehrere Interaktionsdiagramme erstellt (z.B. für das Primärszenario und für jedes Sekundärszenario ein Sequenzdiagramm.)

3.4 Entwicklung von Zustands- und Aktivitätsdiagrammen

Ausgangspunkt: Menge von Sequenzdiagrammen (SDs) zu den Use Cases.

Ziel

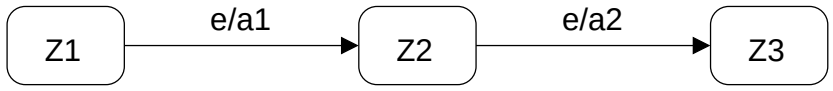
- Je ein Zustandsdiagramm für jede Klasse mit "interessantem Verhalten" (d.h. die Objekte der Klasse haben einen nicht-trivialen Lebenszyklus).
- Aktivitätsdiagramme zur Beschreibung der Abläufe von Operationen.

Bemerkung

- Zustands- und Aktivitätsdiagramme erfassen das (vollständige) Verhalten *eines jeden* Objekts einer Klasse über viele Szenarien hinweg.
- Auf Aktivitätsdiagramme kann verzichtet werden, wenn das Ablauf-Verhalten einer Operation schon vollständig in *einem* Interaktionsdiagramm beschrieben ist.

Kriterien für "interessantes Verhalten" eines Objekts

- Es gibt mindestens ein Ereignis, das in Abhängigkeit vom Objektzustand *unterschiedliche Reaktionen* auslösen kann.



Beispiel: Stellen einer Digitaluhr

- Mindestens ein Ereignis wird in bestimmten Zuständen ignoriert (bzw. kann in bestimmten Zuständen nicht auftreten).

Beispiel: ATM

Typische zustandsabhängige Objekte sind:

- Kontrollobjekte für Anwendungsfälle und Benutzerschnittstellen
- Geräte (z.B. Videorecorder, Digitaluhr, Thermostat, ...)
- Objekte mit einem beschränkten Fassungsvermögen (voll, leer, ...)

Klassifizierung von Zuständen

- Einen Zustand, dem eine Aktivität zugeordnet ist und der **nur** durch ein (evtl. bedingtes) Completion Event verlassen werden kann, nennen wir *Aktivitätszustand*.
- Einen Zustand, dem keine Aktivität zugeordnet ist, nennen wir *stabilen Zustand* (oder *inaktiven Zustand*).

Vorgehensweise bei der Konstruktion eines Zustandsdiagramms

1. Wähle ein SD, das eine typische Interaktion für ein Objekt der betrachteten Klasse zeigt (üblicherweise das SD für das Primärszenario).
2. Betrachte die Projektion des SD auf die Lebenslinie dieses Objekts.
3. Bilde der Lebenslinie des Objekts folgend eine Kette von Zuständen und Transitionen, so dass
   - die Phasen in denen das Objekt nicht aktiv ist, durch stabile Zustände modelliert werden,
   - Aktivierungsphasen durch Aktivitätszustände modelliert werden (in denen lokale Operationen zur Durchführung der Aktivität aufgerufen werden),
   - eintreffende Ereignisse durch entsprechend markierte Transitionen von stabilen Zuständen in Aktivitätszustände modelliert werden,
   - die Beendigung einer Aktivität durch eine Transition mit einem Completion Event von einem Aktivitätszustand in einen stabilen Zustand modelliert wird.
4. Bilde Zyklen für Folgen, die wiederholt werden können.

{ein SD verarbeitet, noch ohne Detaillierung der Aktivierungen}

5. Solange es weitere SDs für Objekte der betrachteten Klasse gibt,
   - wähle ein solches SD und projiziere es auf die Lebenslinie des Objekts,
   - finde den Aktivitätszustand, wo die Sequenz von dem bisher beschriebenen Verhalten abweicht,
   - hänge die neue Folge als Alternative an diesen Zustand an,
   - finde (wenn möglich) einen stabilen Zustand, in dem die alternative Folge mit dem bestehenden Zustandsdiagramm vereinigt werden kann.

{alle SDs verarbeitet, noch ohne Detaillierung der Aktivierungen}

6. Konstruiere Aktivitätsdiagramme für lokale Operationen (die in Aktivitätszuständen aufgerufen werden; vgl. Schritt 3).

7. Verfeinere das bestehende Zustandsdiagramm ggf. durch Einfügen von Bedingungen bei den von den Aktivitätszuständen ausgehenden Transitionen.

{alle SDs mit allen Aktivierungen verarbeitet}

8. Integriere (soweit noch nötig) alle angegebenen Sekundärszenarien, für die kein SD vorhanden ist.

Bemerkung

- Der nach den Schritten 1-5 erhaltene Entwurf eines Zustandsdiagramms ist i.a. nicht-deterministisch. Dieser Entwurf wird in Schritt 7 zu einem deterministischen Zustandsdiagramm verfeinert.
- Eine genaue Formalisierung der angegebenen Methodik ist zu finden in:

  R. Hennicker, A. Knapp: Activity-Driven Synthesis of State Machines. Konferenzband FASE 2007, Fundamental Approaches to Software Engineering, Springer Lecture Notes in Computer Science 4422, 87-101, 2007.

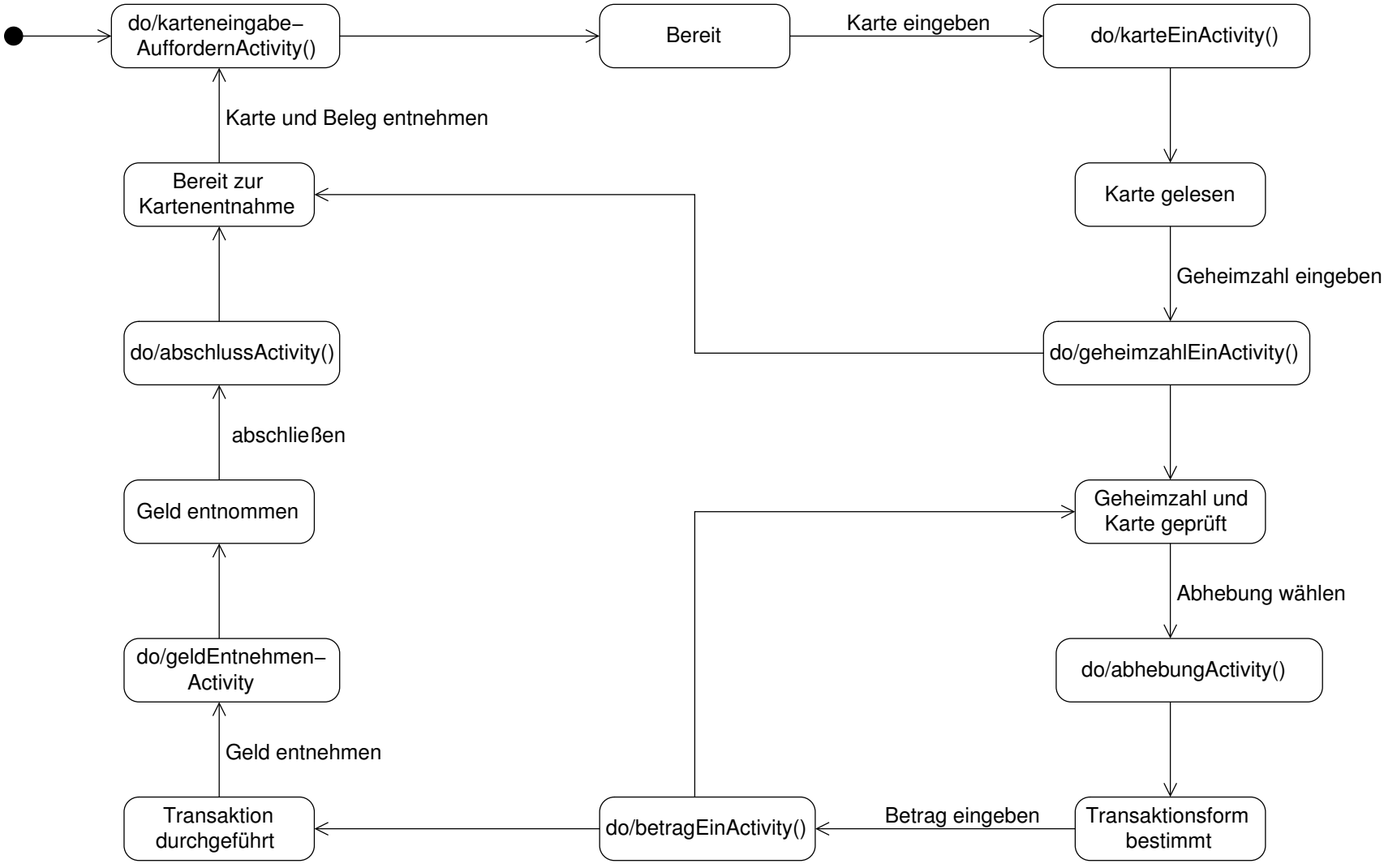
Beispiel ATM:

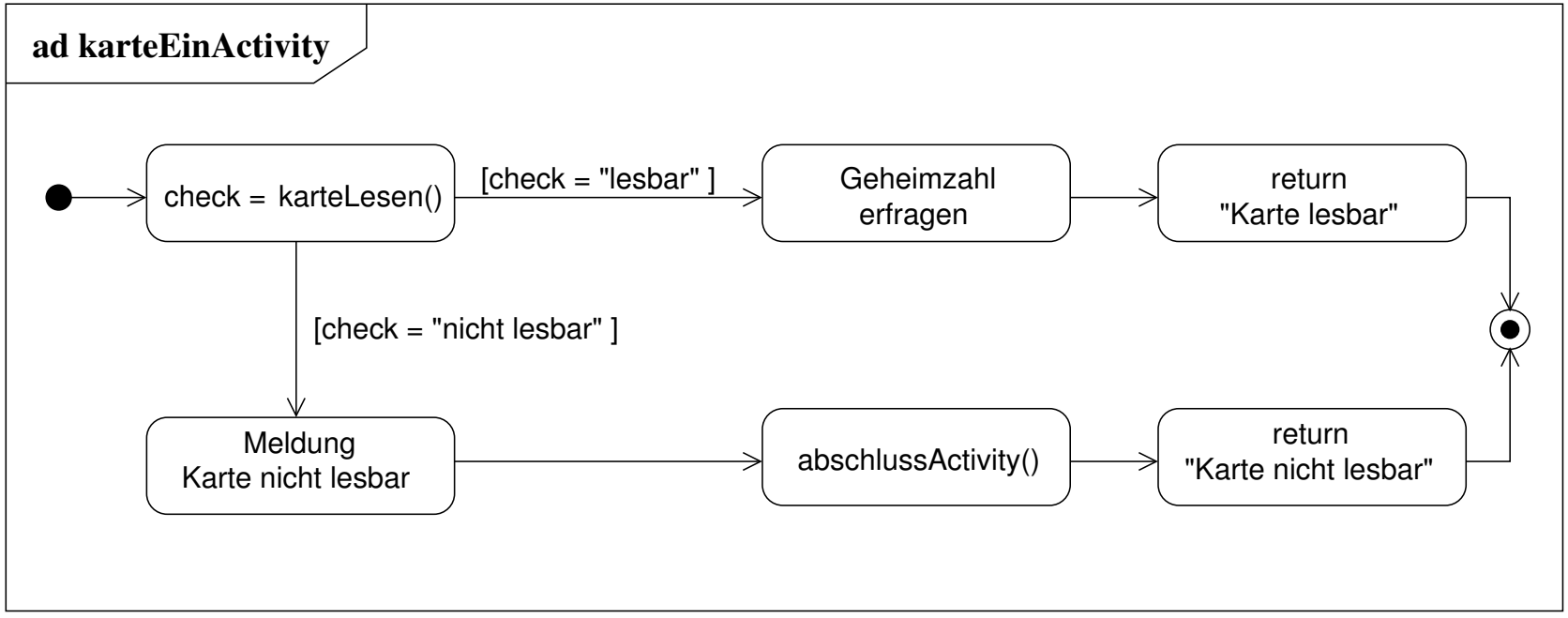
- "interessantes Verhalten": ATM (*Zustandsdiagramm*)
- "Operationen": (*Aktivitätsdiagramme*)
  - für ATM:
    - lokale Operationen
  - für Konsortium:
    - Karte überprüfen
    - Transaktion verarbeiten
  - für Bank:
    - Karte bei Bank überprüfen
    - Banktransaktion verarbeiten

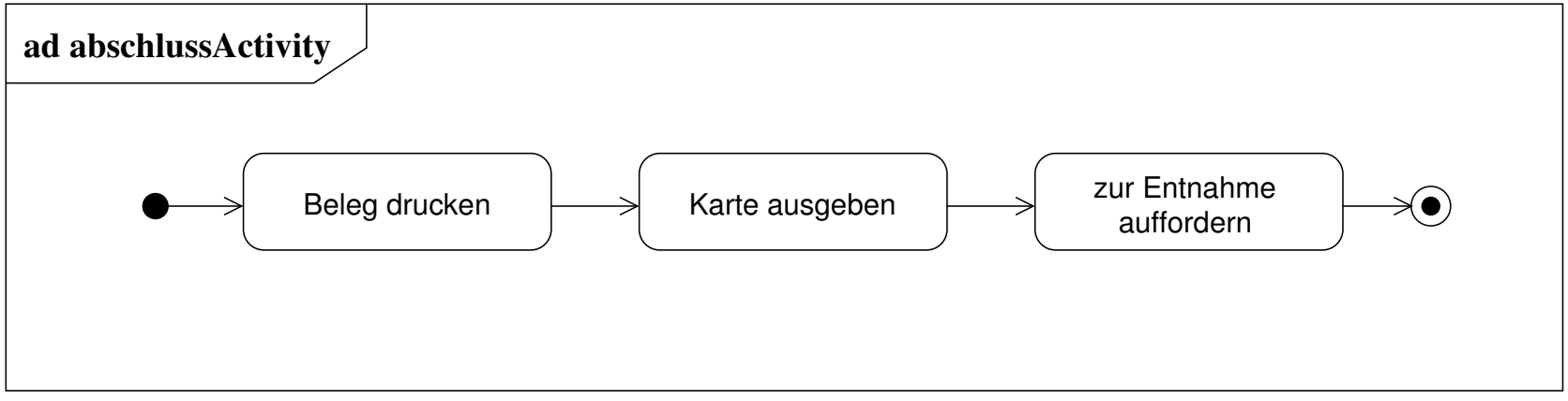
Zustandsdiagramm für ATM konstruieren

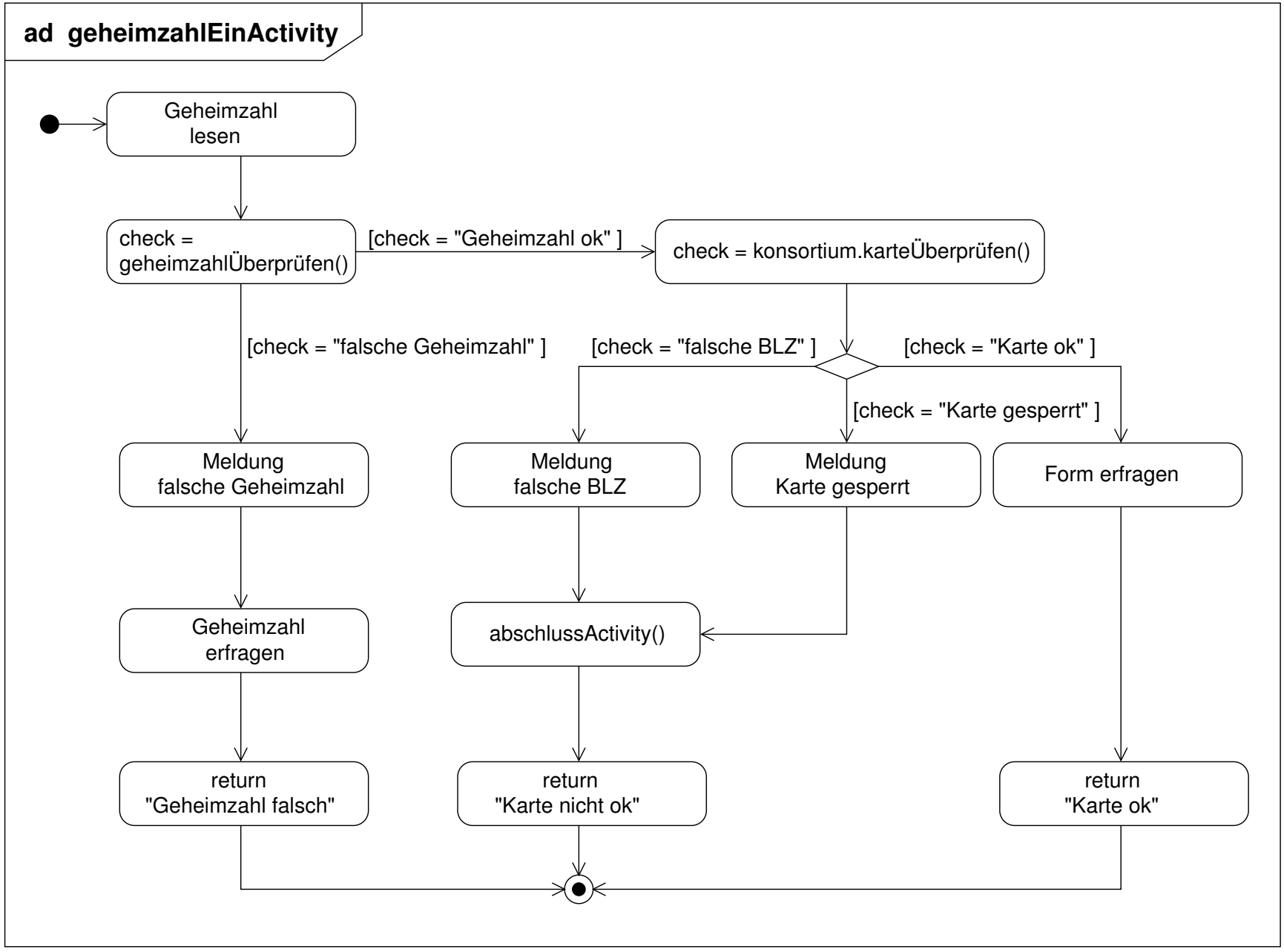
1. Wähle SD für das Primärszenario von "Geld am ATM abheben"
2. Betrachte die Lebenslinie von ":ATM"
3. Kette von Zuständen und Transitionen bilden mit einem Aktivitätszustand nach jeder Benutzerinteraktion
4. Schleife bei Zustand "Bereit"
5. SD für "Karte gesperrt" integrieren,
   SD für "Transaktion gescheitert" integrieren
6. Aktivitätsdiagramme für die in den Aktivitätszuständen aufgerufenen lokalen Operationen konstruieren
7. Bedingungen einfügen bei den von o.g. Aktivitätszuständen ausgehenden Transitionen und das bestehende Zustandsdiagramm verfeinern
8. Noch nicht berücksichtigte Sekundärszenarien integrieren
   ("Abbruch", etc.)

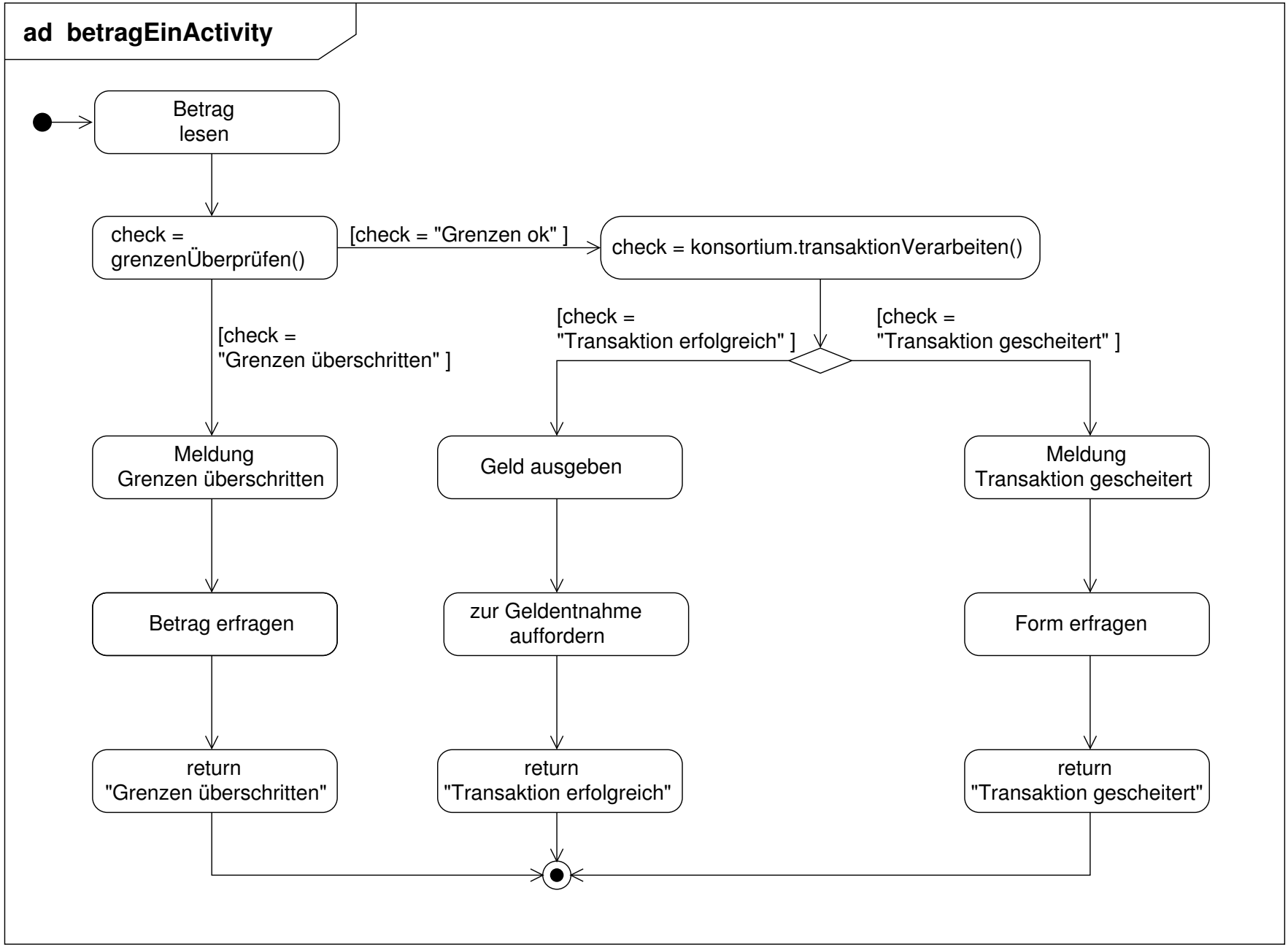
Zustandsdiagramm für ATM (nach den Schritten 1-5)

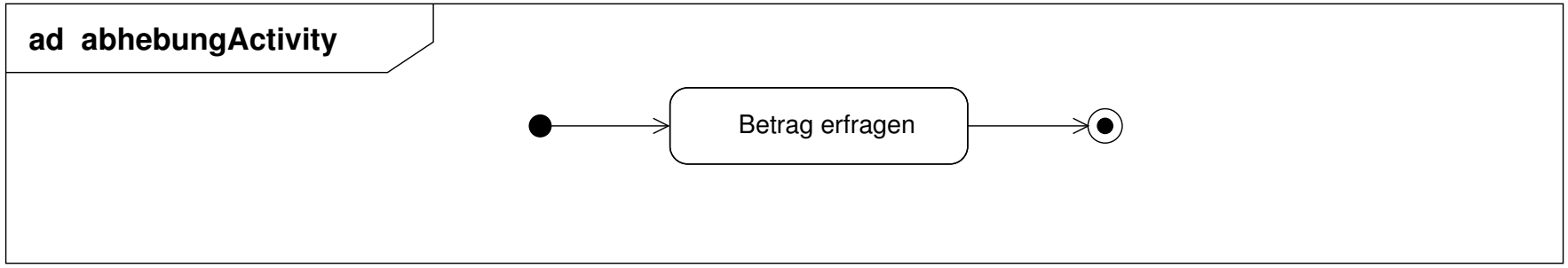


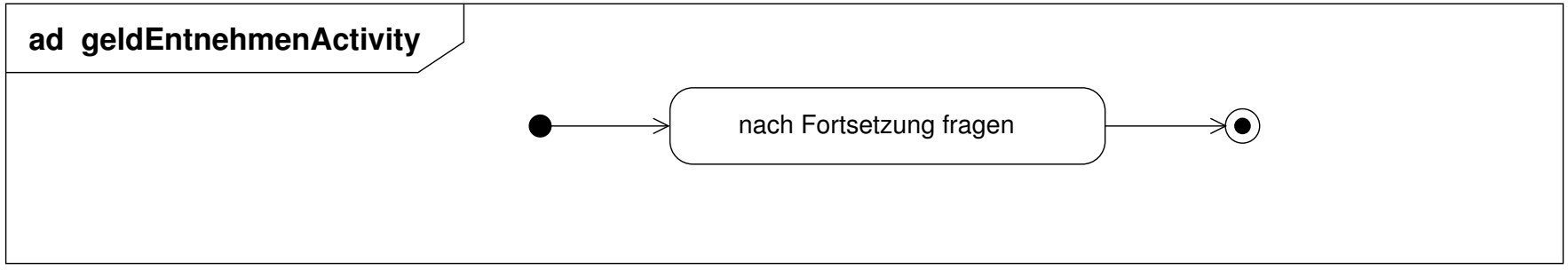

ad karteEinActivity
check = karteLesen()
[check = "lesbar"]
Geheimzahl erfragen
return "Karte lesbar"
[check = "nicht lesbar"]
Meldung Karte nicht lesbar
abschlussActivity()
return "Karte nicht lesbar"

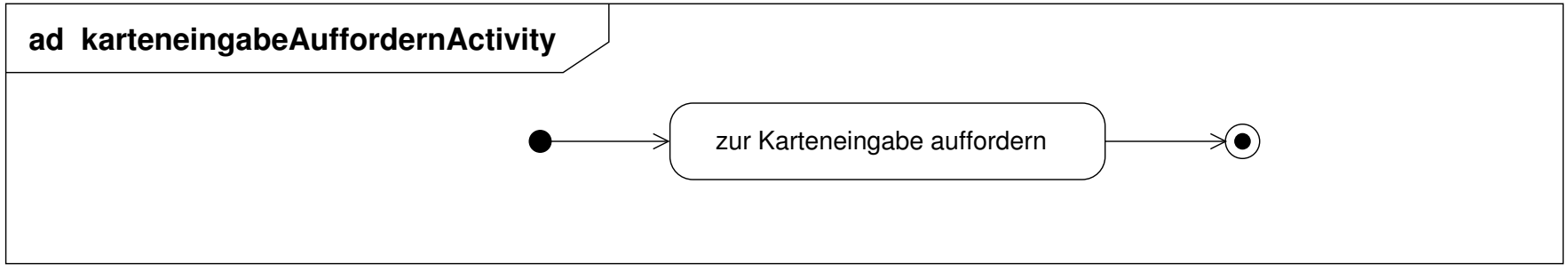

ad abschlussActivity
Beleg drucken
Karte ausgeben
zur Entnahme auffordern


ad geheimzahlEinActivity
Geheimzahl lesen
check = geheimzahlÜberprüfen()
[check = "Geheimzahl ok"]
check = konsortium.karteÜberprüfen()
[check = "falsche Geheimzahl"]
[check = "falsche BLZ"]
[check = "Karte ok"]
[check = "Karte gesperrt"]
Meldung falsche Geheimzahl
Meldung falsche BLZ
Meldung Karte gesperrt
Form erfragen
Geheimzahl erfragen
abschlussActivity()
return "Geheimzahl falsch"
return "Karte nicht ok"
return "Karte ok"

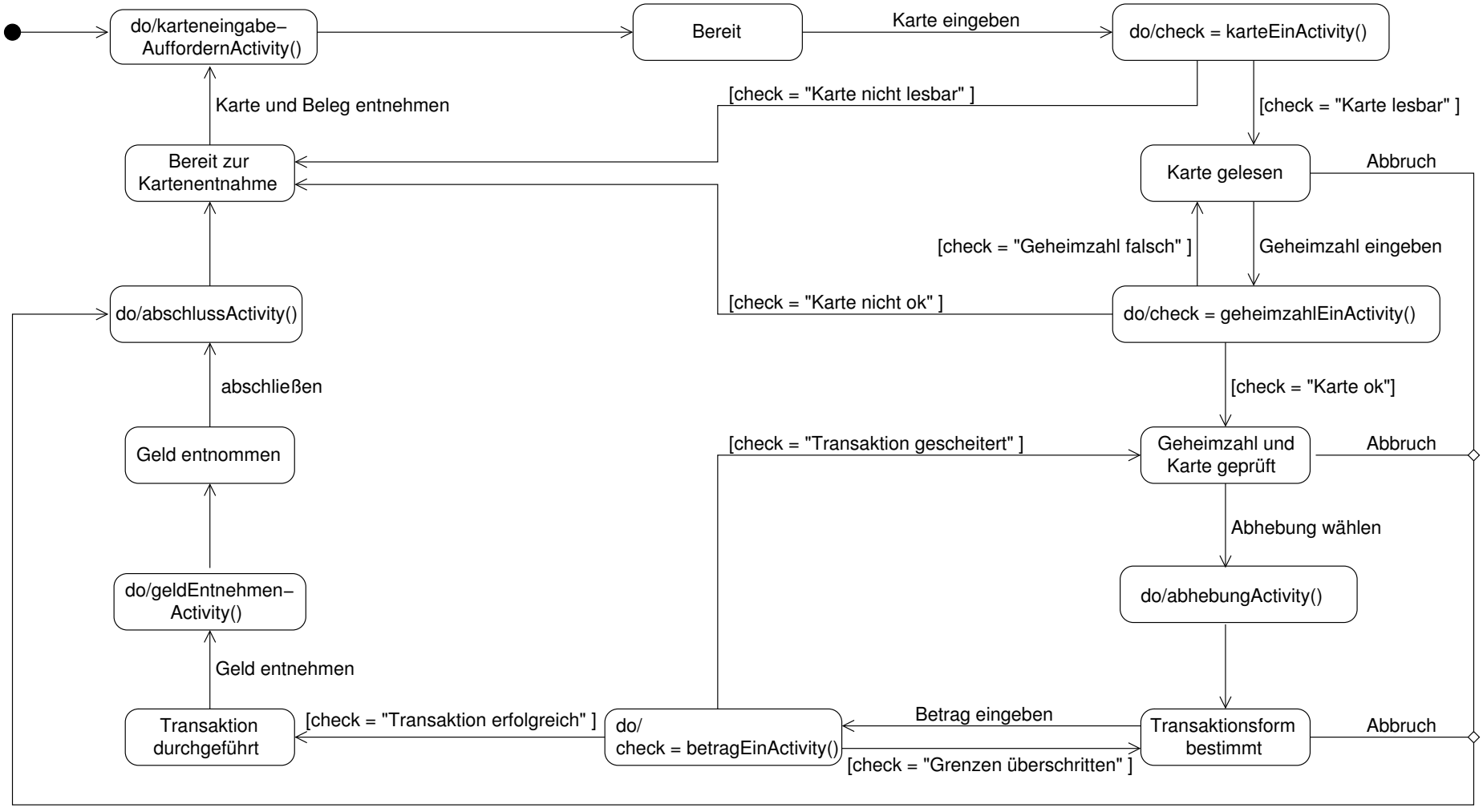

ad betragEinActivity
Betrag lesen
check = grenzenÜberprüfen()
[check = "Grenzen ok"]
check = konsortium.transaktionVerarbeiten()
[check = "Grenzen überschritten"]
[check = "Transaktion erfolgreich"]
[check = "Transaktion gescheitert"]
Meldung Grenzen überschritten
Geld ausgeben
Meldung Transaktion gescheitert
Betrag erfragen
zur Geldentnahme auffordern
Form erfragen
return "Grenzen überschritten"
return "Transaktion erfolgreich"
return "Transaktion gescheitert"


ad abhebungActivity
Betrag erfragen

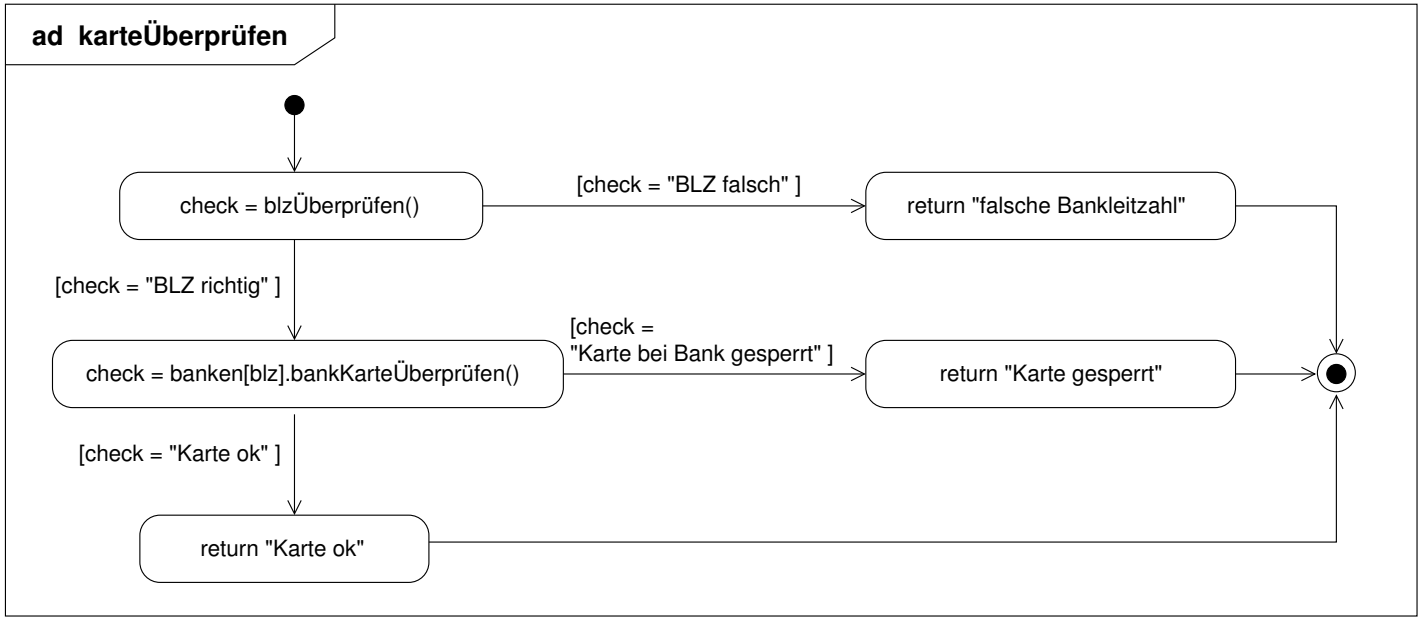

ad geldEntnehmenActivity
nach Fortsetzung fragen

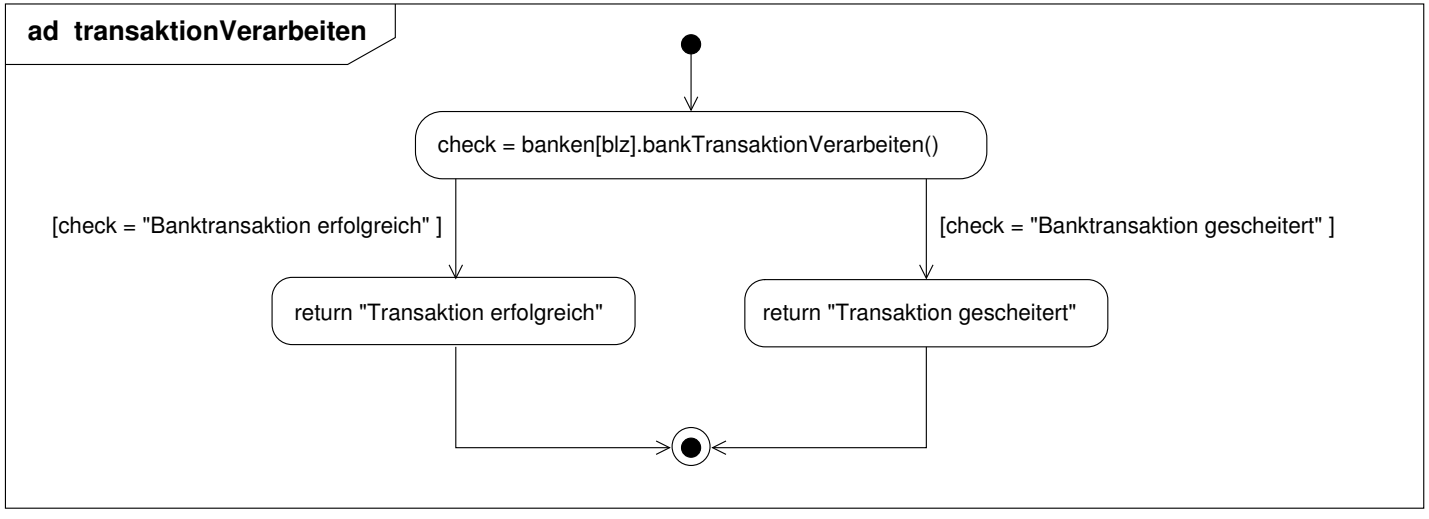

ad karteneingabeAuffordernActivity
zur Karteneingabe auffordern

Zustandsdiagramm für ATM (nach den Schritten 6-8)

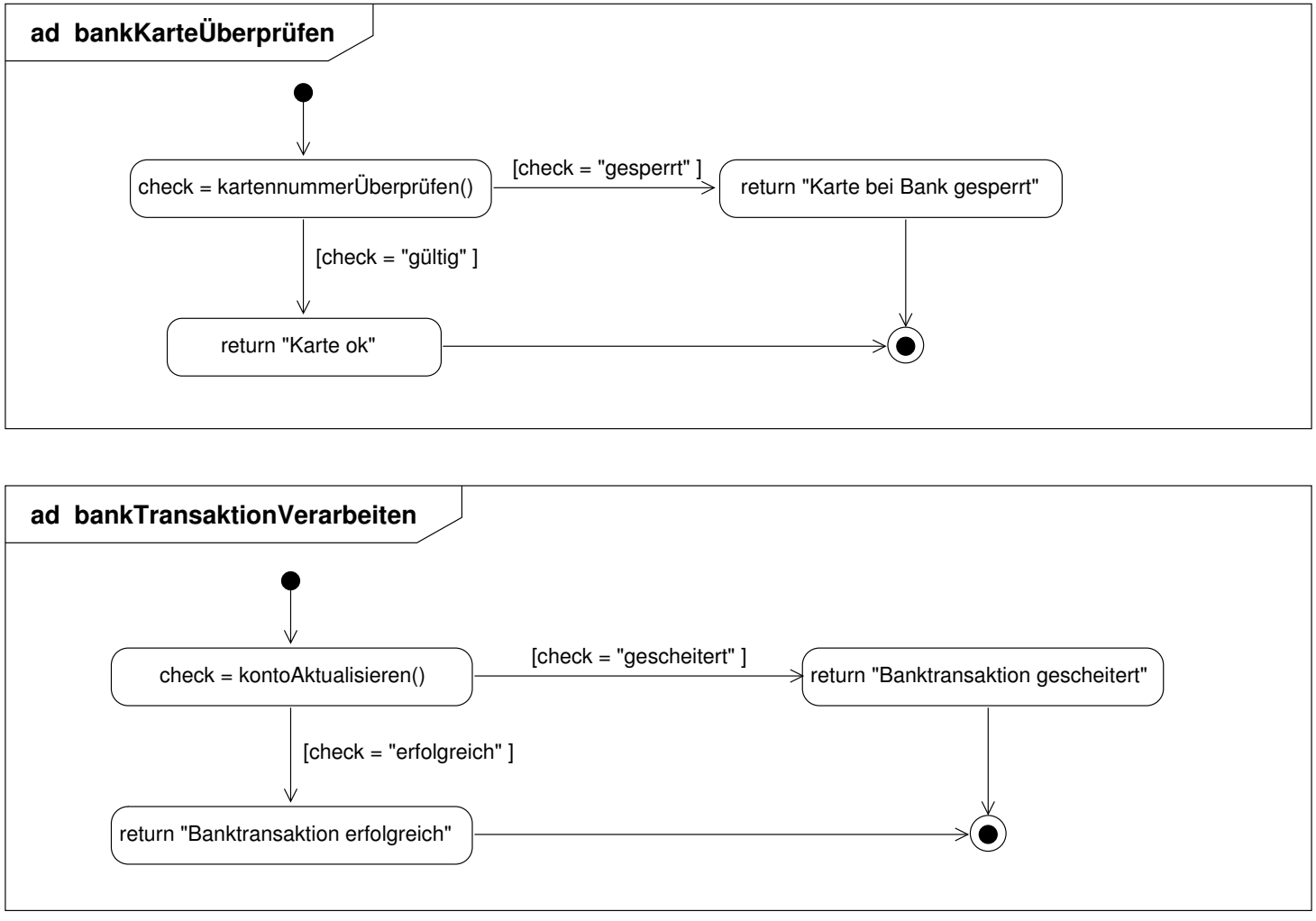


Aktivitätsdiagramme für Operationen der Klasse Konsortium





Aktivitätsdiagramme für Operationen der Klasse Bank



Bemerkung

Die Konsistenz der Diagramme untereinander kann leicht überprüft werden.

Zusammenfassung von Abschnitt 3.4

- Zustands- und Aktivitätsdiagramme können ausgehend von dem Modell der Interaktionen systematisch entwickelt werden.
- Zustandsdiagramme werden für Klassen, deren Objekte einen nicht-trivialen Lebenszyklus haben, entwickelt.
- Bei der schrittweisen Entwicklung von Zustandsdiagrammen werden stabile Zustände und Aktivitätszustände unterschieden.
- Zur Beschreibung von Operationen werden Aktivitätsdiagramme erstellt.

Kapitel 4

Objektorientierter Entwurf

Prof. Dr. Rolf Hennicker

15.12.2009

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

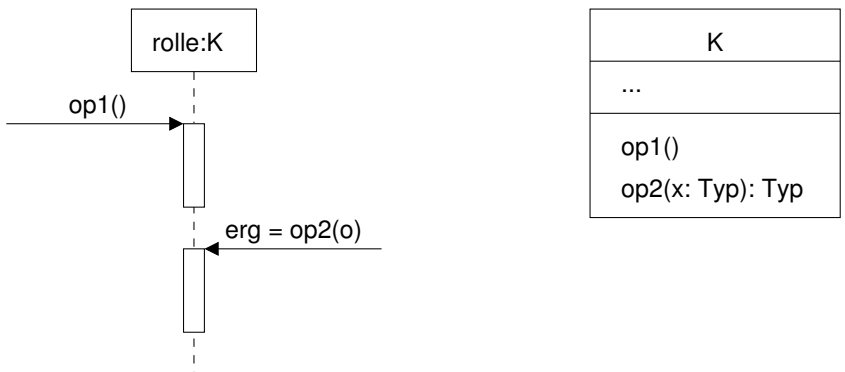
1. Operationen hinzufügen
2. Assoziationen ausrichten
3. Zugriffsrechte bestimmen
4. Mehrfachverebung auflösen
5. Wiederverwendung von Klassen

4.1.1 Operationen hinzufügen

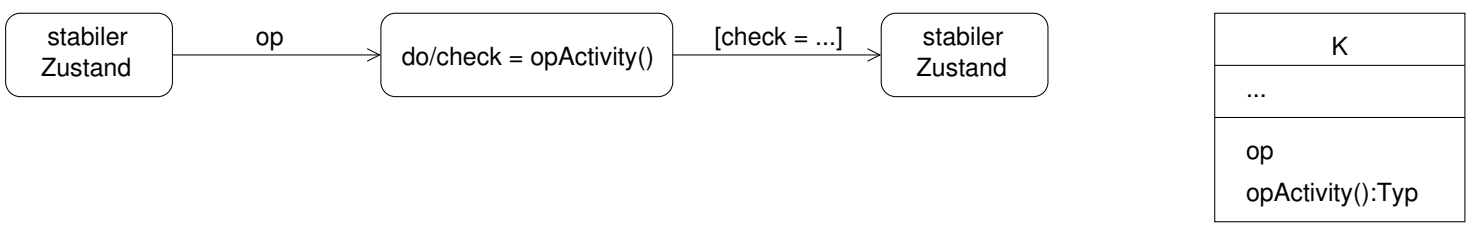
Vorgehensweise

Sei K eine Klasse des Objektmodells.

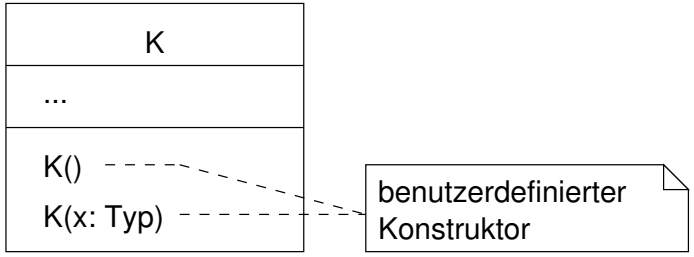
- Führe eine Operation für jede an ein Objekt von K gesendete Nachricht ein.



- 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).

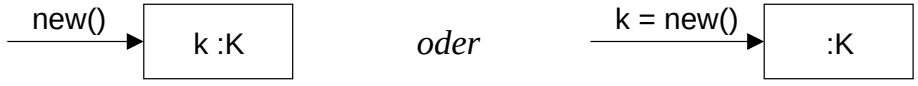


- Benutzerdefinierte Konstruktoren bei nicht abstrakten Klassen hinzufügen



Aufruf eines Konstruktors:

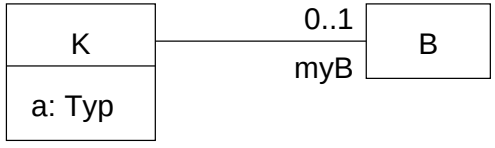
- im Sequenzdiagramm:

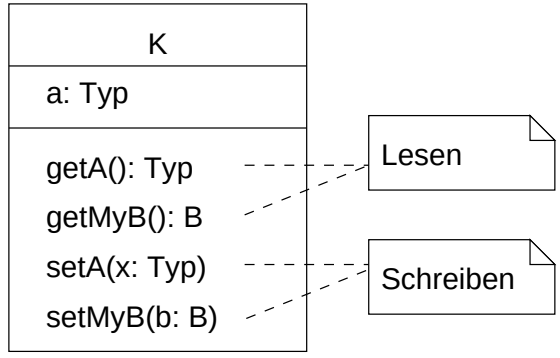


- im Pseudocode:

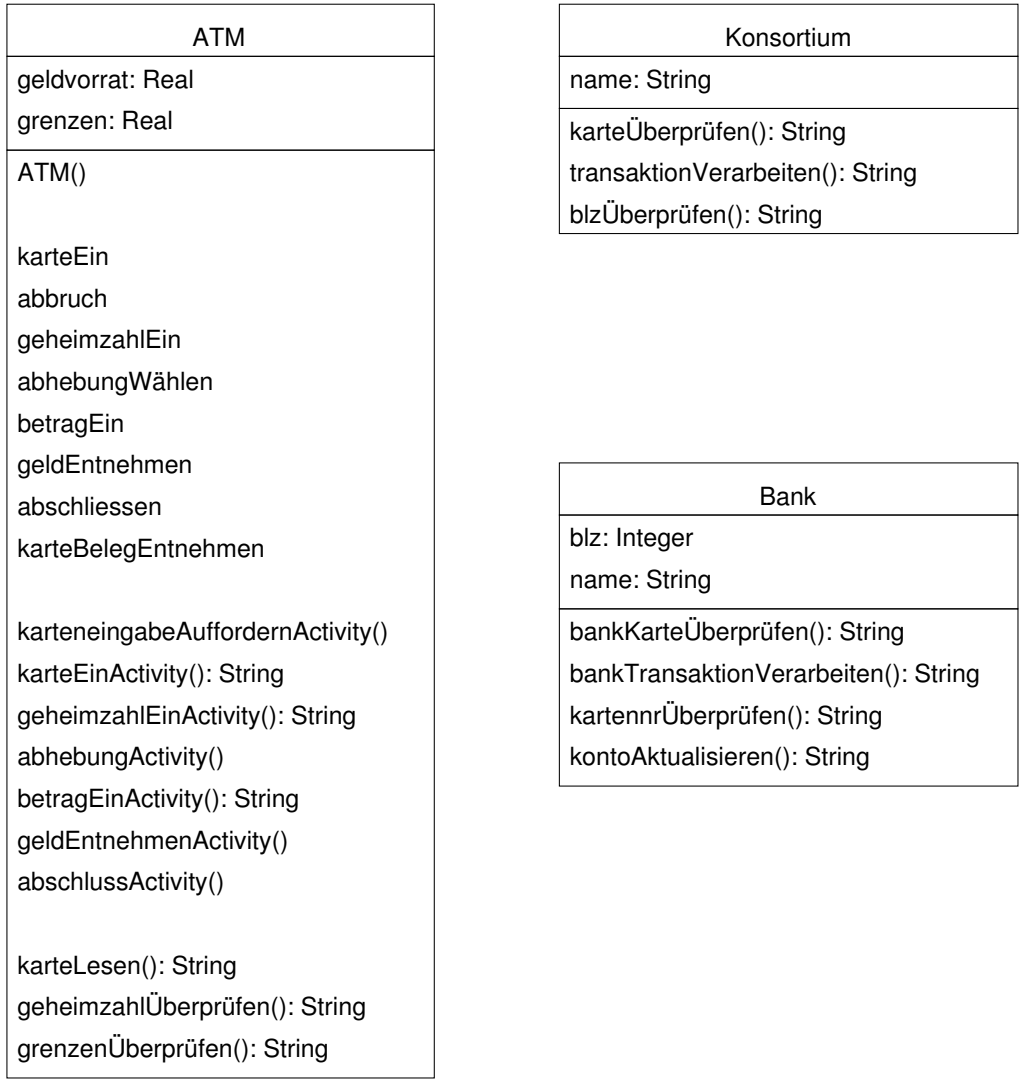
```
k = new K();      //falls k ein Rollenname ist
K k = new K();    //falls k eine lokale Variable ist
```

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





Beispiel ATM:



- 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")

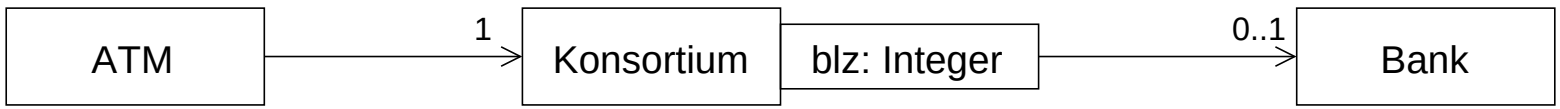
  Beachte:

  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

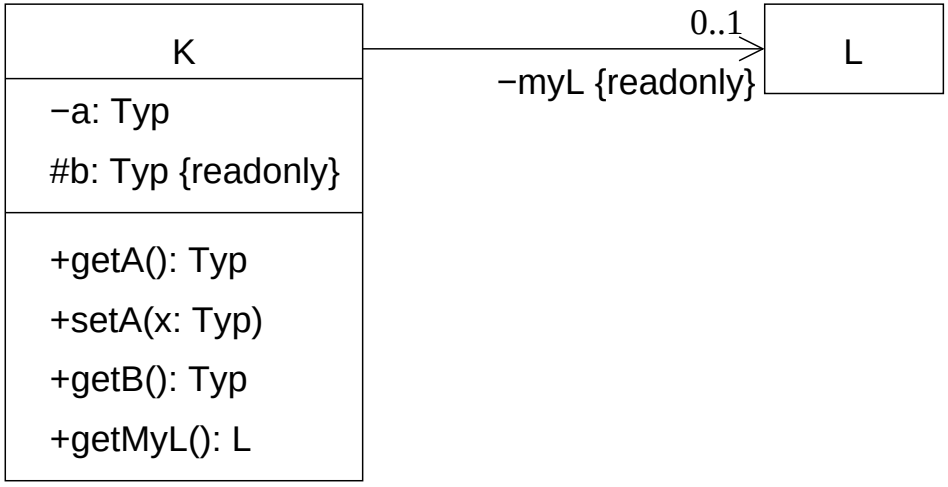
- Analysiere in welcher/welchen Richtung(en) eine Assoziation (beim Senden von Nachrichten bzw. Operationsaufrufen) durchlaufen wird.
- Falls eine Assoziation nur in einer Richtung durchlaufen wird, dann richte sie entsprechend aus.

Beispiel ATM:



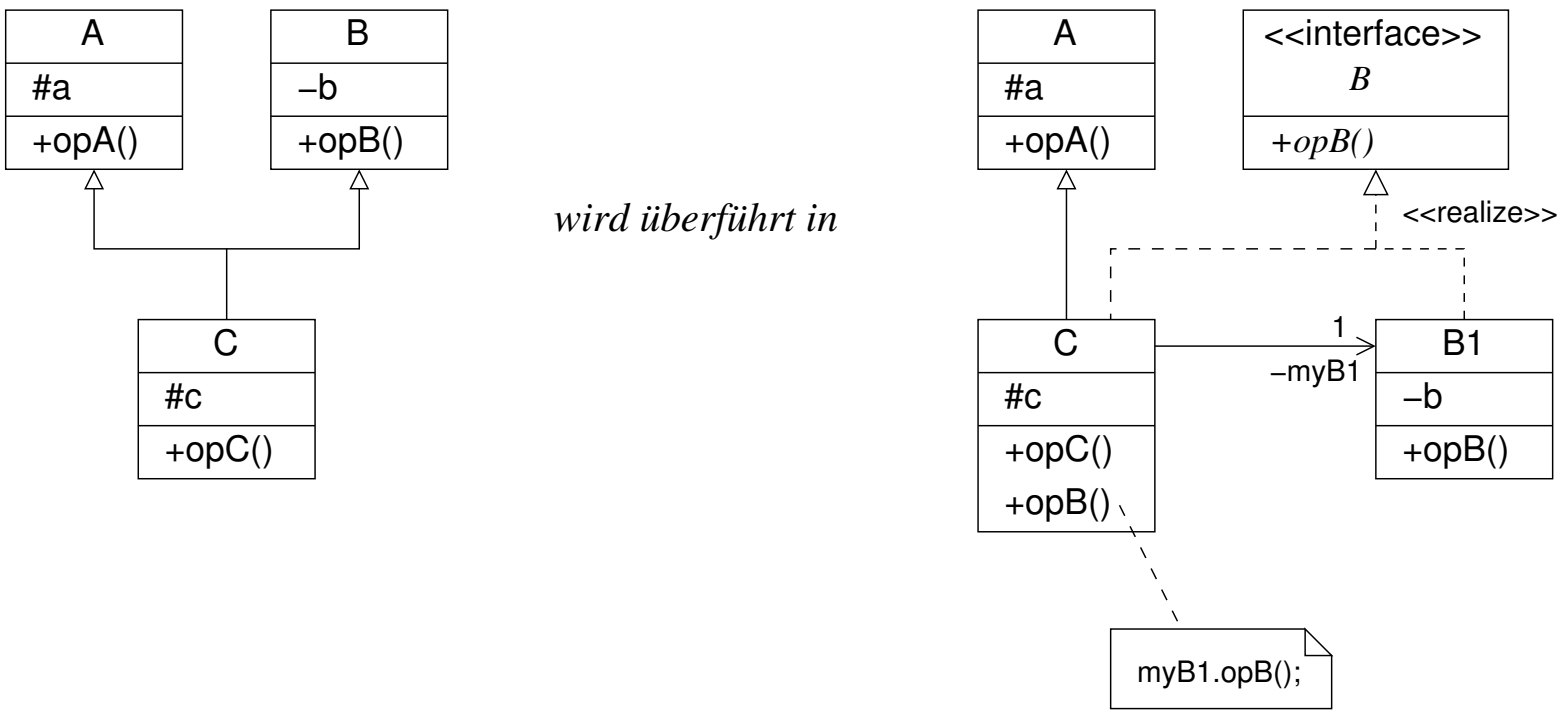
4.1.3 Zugriffsrechte bestimmen

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



4.1.4 Mehrfachvererbung auflösen

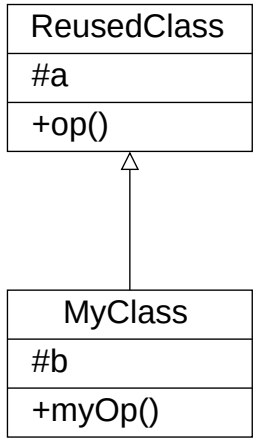
- Ist notwendig, wenn die Zielsprache keine Mehrfachvererbung für Klassen unterstützt (z.B. Java).
- Die Auflösung der Mehrfachvererbung ist möglich durch Einführung einer Schnittstelle.



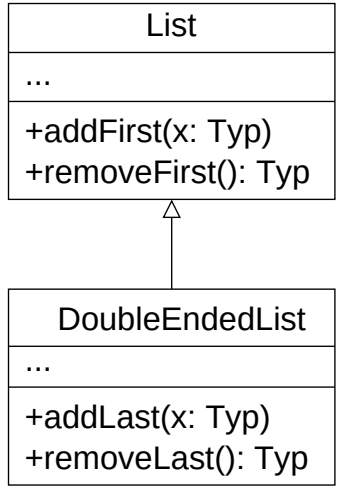
4.1.5 Wiederverwendung von Klassen

- 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:



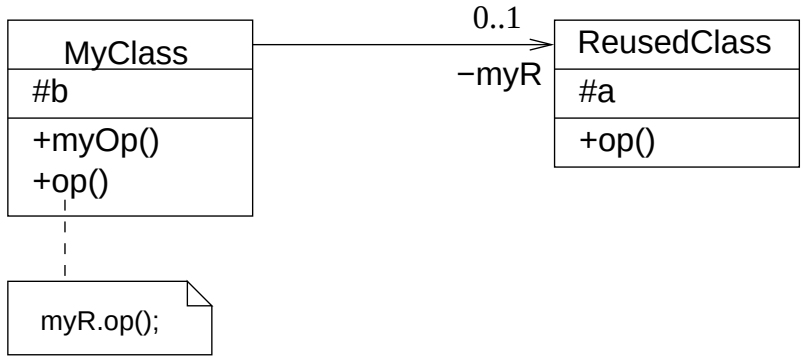
Beispiel:



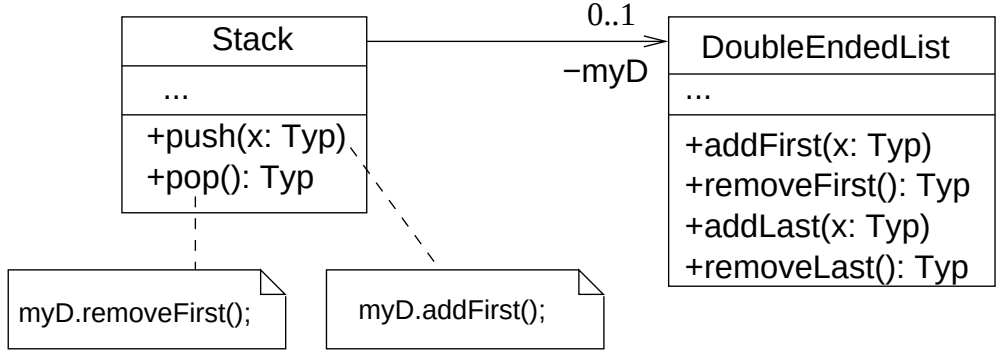
Beachte: Wenn die wiederverwendete Klasse auch Operationen anbietet, die die spezielle Klasse nicht benötigt, ist eventuell das Kapselungsprinzip verletzt. In diesem Fall ist Wiederverwendung durch *Delegation* vorzuziehen.

- 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() {
  output("Karte eingeben?");
}

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

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

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

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

abhebungActivity() {
  output("Betrag?");
```

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

```
geldEntnehmenActivity() {
  output("Fortsetzung?");
}

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

```
karteLesen(): String {
  aktKartennr   = inputKartennr();   // neues Attribut von ATM
  aktBLZ        = inputBLZ();        // neues Attribut von ATM
  aktGeheimzahl = inputGeheimzahl(); // 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 ueberschritten";
}
```

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";
}
```

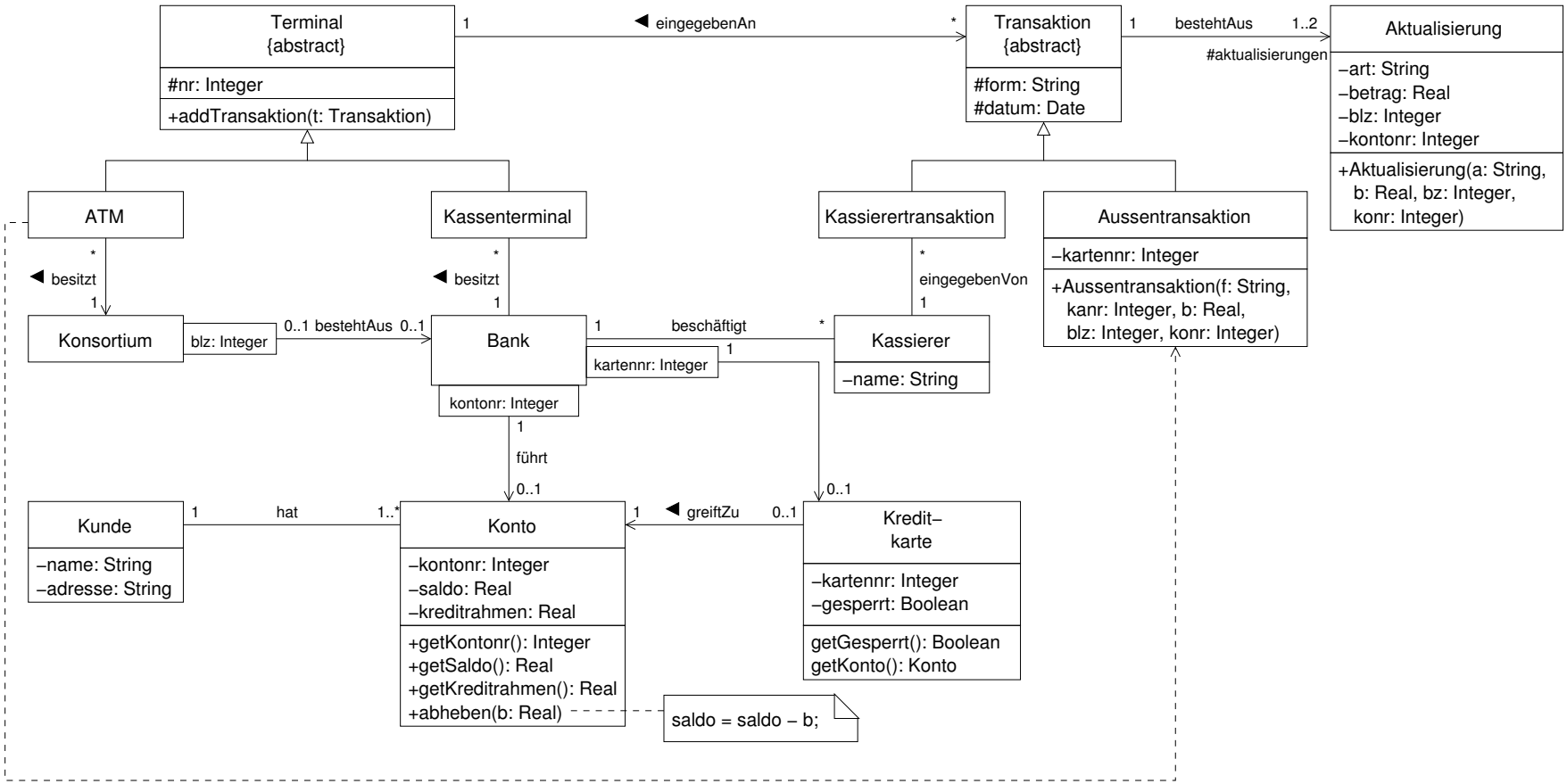
3. Operationen der Klasse Bank


```
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.
  // Dafuer 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.
  // Dafuer 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

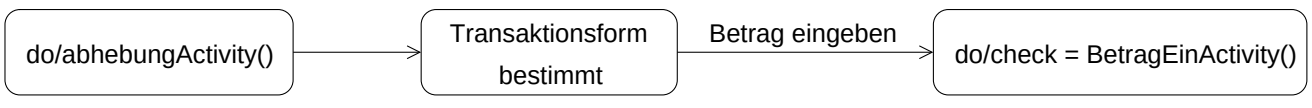
Idee

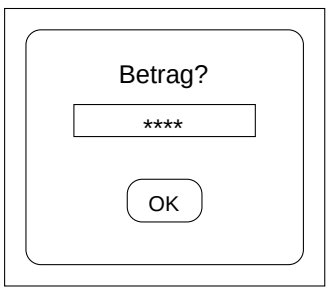
Ereignisse werden durch "modale Dialoge" (erzwungene Benutzereingaben) realisiert.

Voraussetzung

Objekte befinden sich an der Systemgrenze
(Kontrollobjekte zur Dialogsteuerung)

Beispiel:





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

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

Vorgehensweise

Das gesamte Zustandsdiagramm wird überführt in eine Prozedur mit

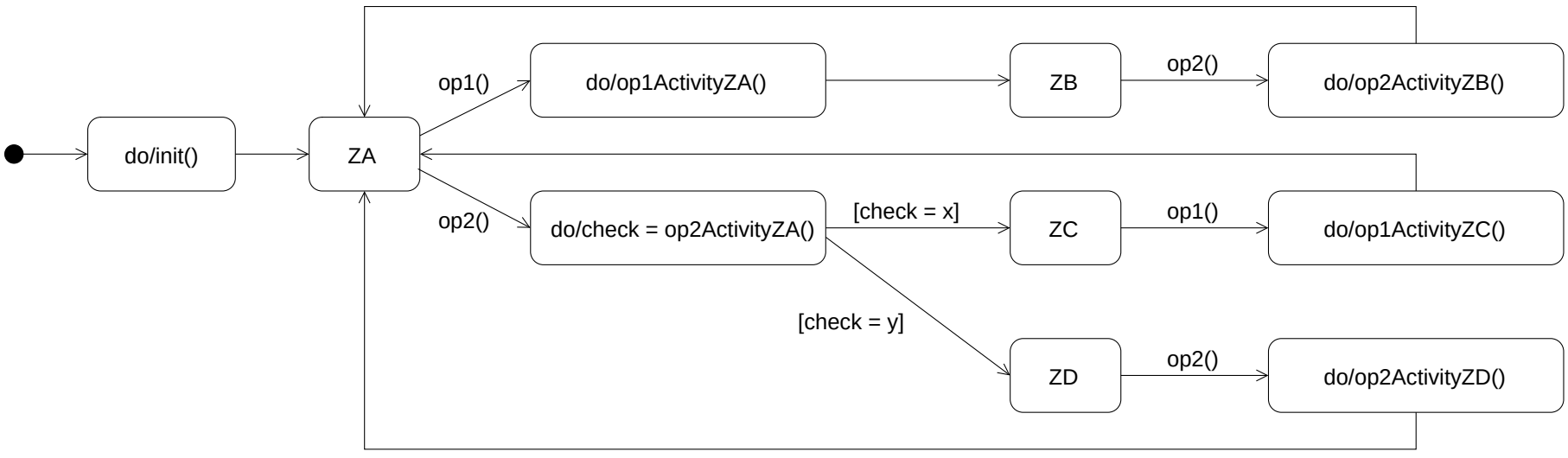
- modalen Dialogen für die (externen) Ereignisse
- bedingten Anweisungen für Verzweigungen
- Wiederholungsanweisungen für Zyklen des Diagramms

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

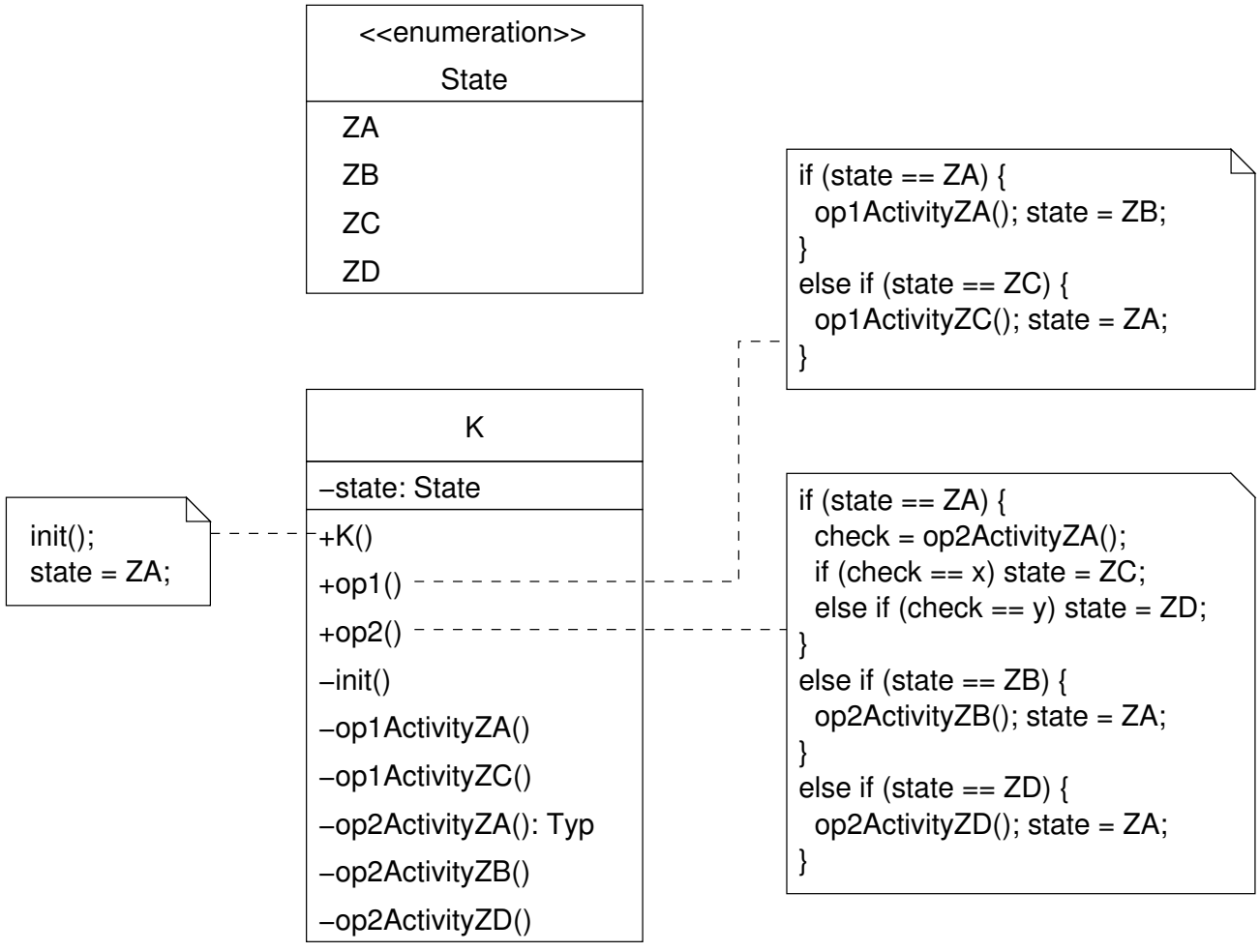
- Verwende einen Enumerationstyp zur Darstellung der (endlich vielen) stabilen Zustände.
- Führe ein explizites Zustandsattribut für die betrachtete Klasse K ein.
- Realisiere die zustandsabhängigen Operationen durch Fallunterscheidung nach dem aktuellen (stabilen) Zustand.

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.

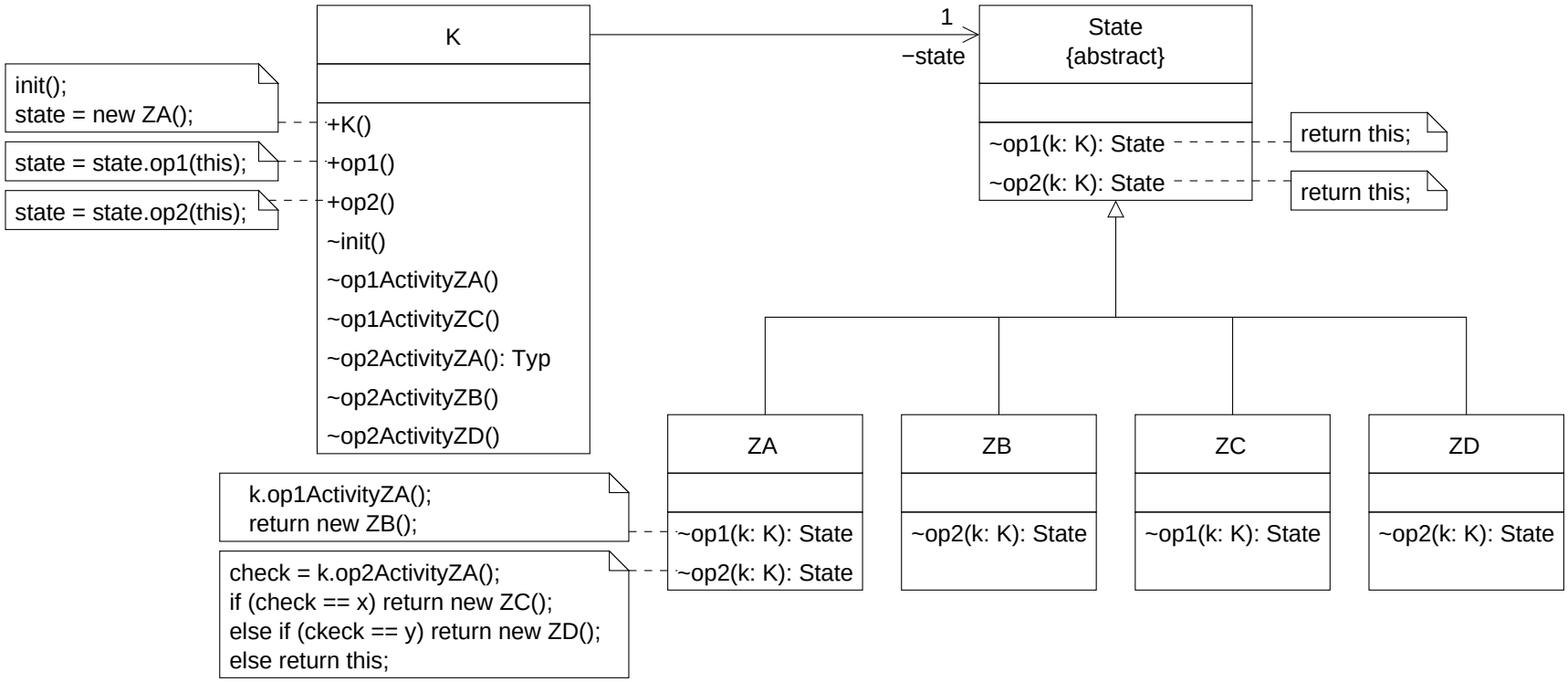
Vorteil

Einfache Erweiterbarkeit bzgl. neuer Zustände.

Nachteil

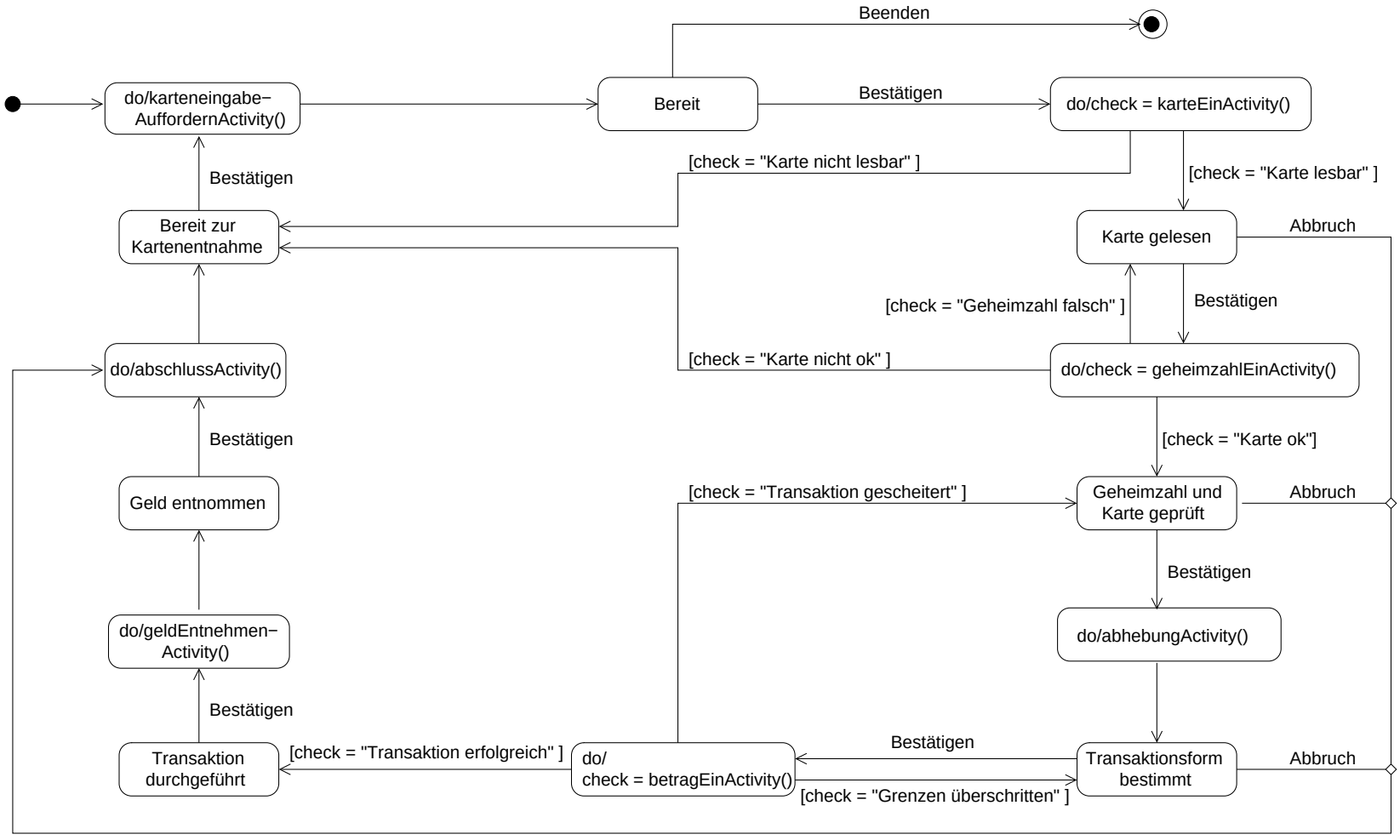
Schlechte Erweiterbarkeit bzgl. neuer Operationen.

Das Zustandsdiagramm von oben wird folgendermaßen realisiert:



Beispiel: Realisierung des ATM durch Zustandsobjekte

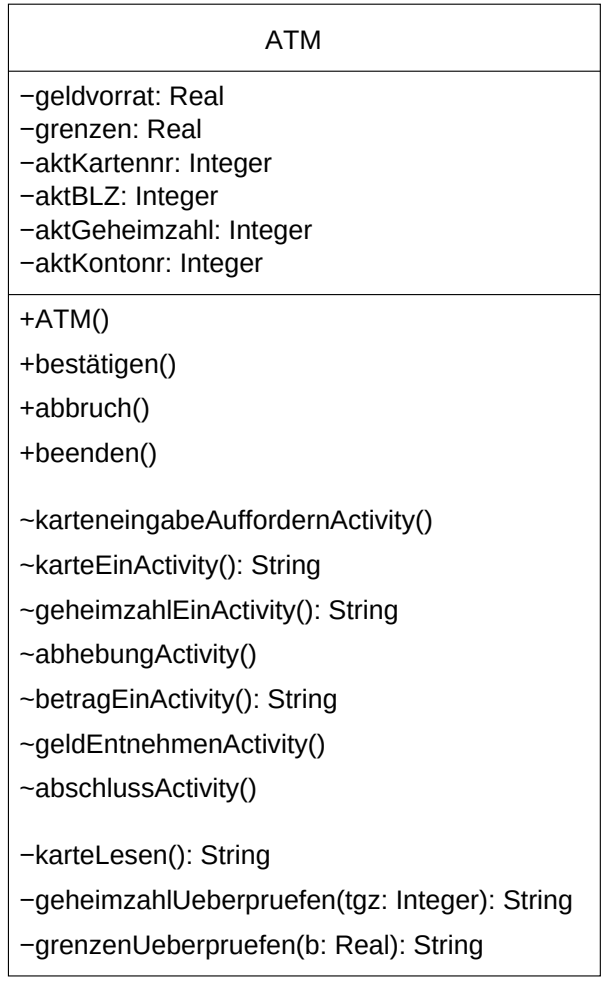
Zustandsdiagramm für ereignisgesteuerte ATM-Simulation



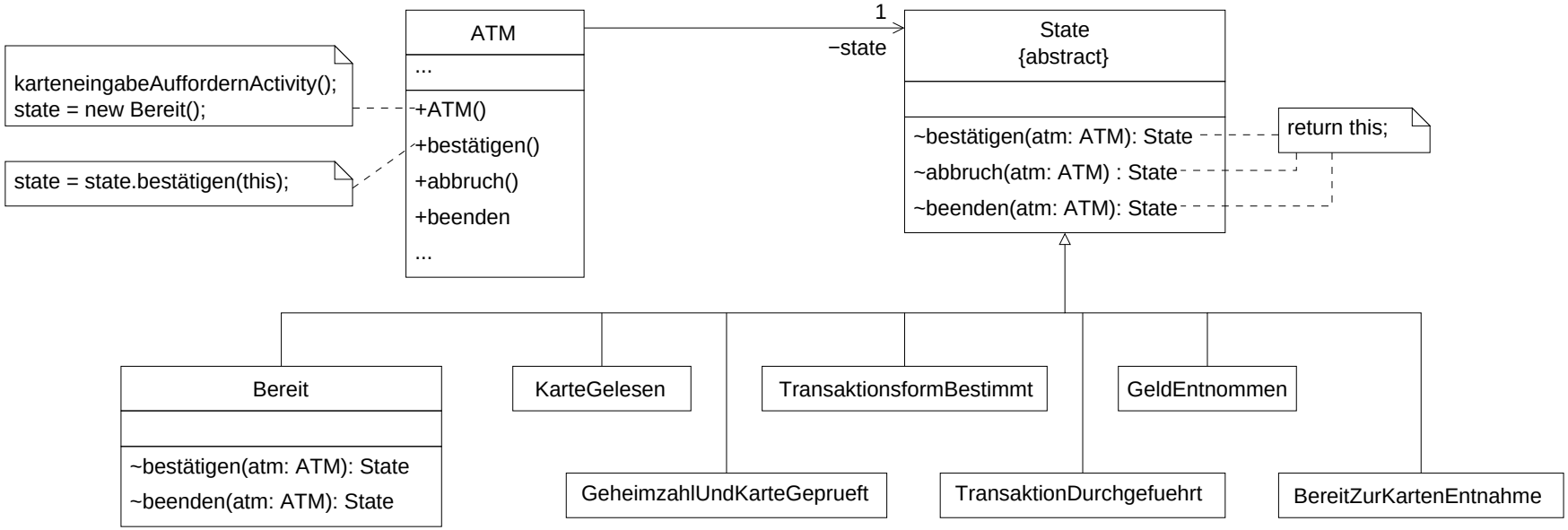
Die Ereignisse “Bestätigen”, “Abbruch” und “Beenden” werden durch die Betätigung entsprechender Buttons der Benutzerschnittstelle hervorgerufen.

Überarbeitete Klasse ATM

Die Operationen “karteEin”, “geheimzahlEin”, ..., “karteBelegEntnehmen”von früher werden entfernt und durch die zustandsabhängige Operation “bestaetigen”realisiert.



Realisierung des Zustandsdiagramms der ATM-Simulation



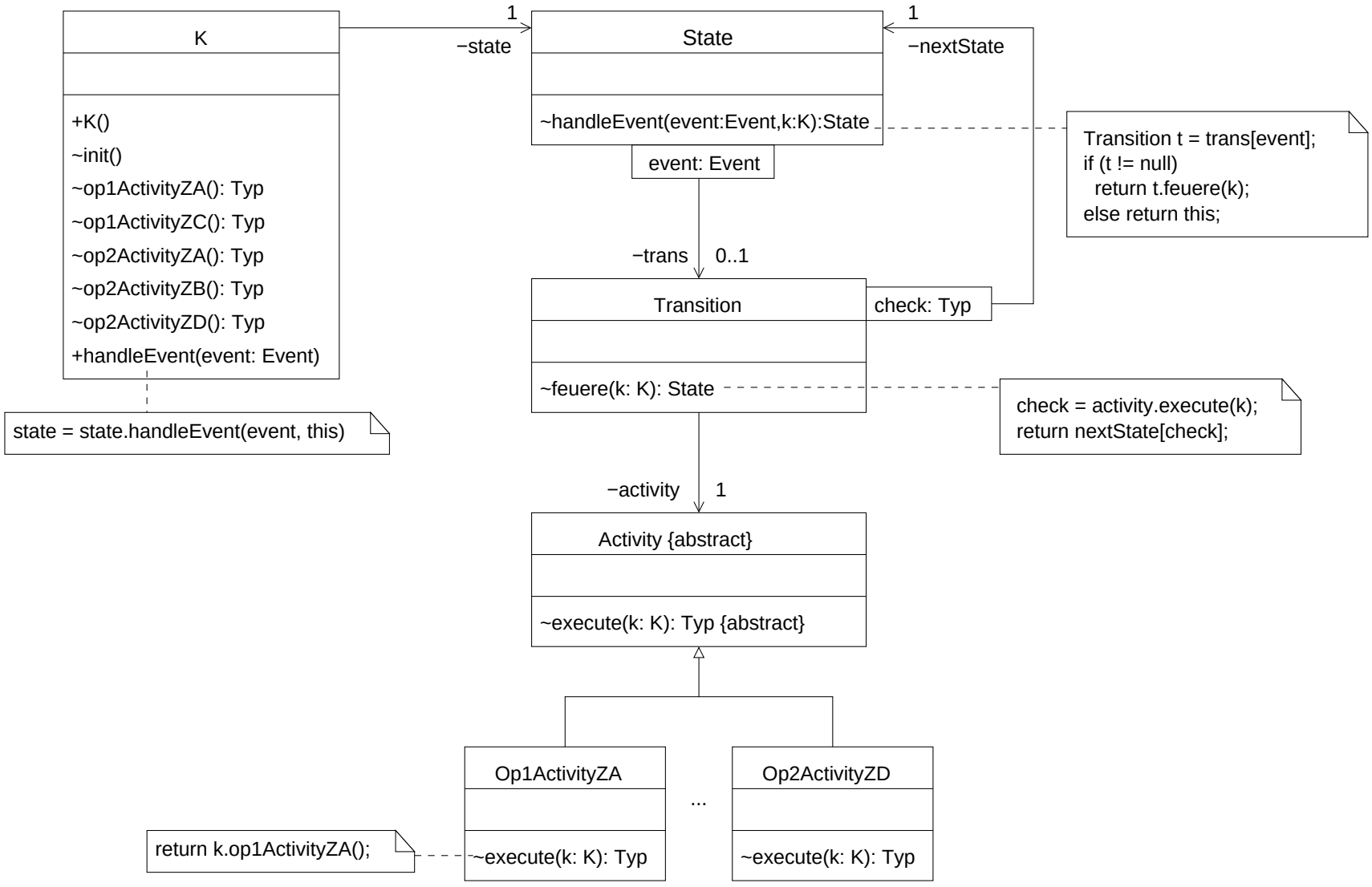
Analog werden die vier übrigen Zustandsklassen implementiert.

4.2.4 Realisierung durch eine Zustandsmaschine

Idee

- Alle in einem Zustandsdiagramm vorkommende Größen (Zustände, Transitionen, Aktivitäten, Ereignisse) 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:



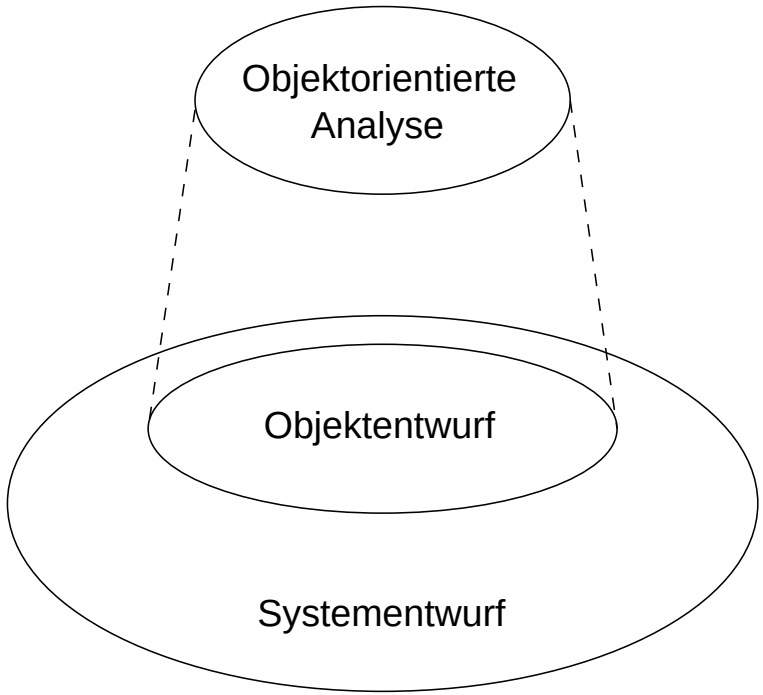
Zusammenfassung von Abschnitt 4.2

- 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 Agorithmen,
  - der Realisierung des Zustandsdiagramms der ATM-Simulation (Abschnitt 4.2).

4.3 Systementwurf

Ziele

- Einbettung des Objektentwurfs in die Systemumgebung
- Festlegung der Systemarchitektur



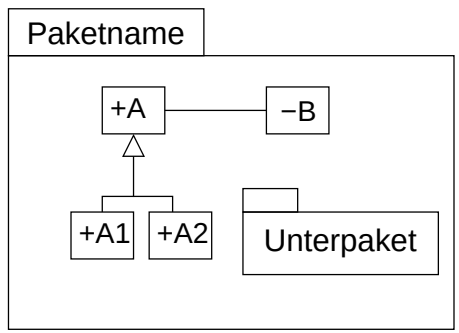
4.3.1 Pakete und Komponenten

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:

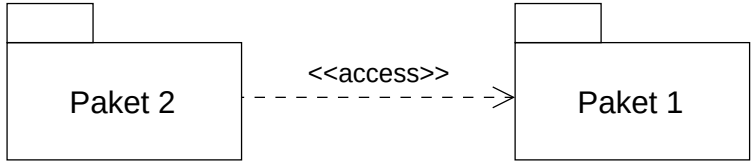


Die öffentlichen Elemente eines Pakets sind außerhalb des Pakets (immer) zugreifbar unter Verwendung ihres qualifizierten Namens, z.B. **Paketname::A**.

Import-Beziehungen zwischen Paketen

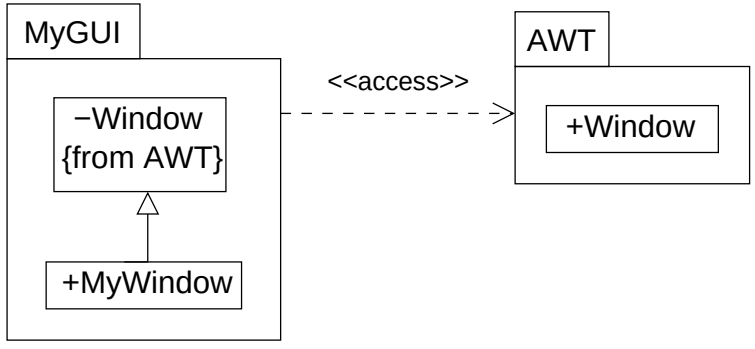
Durch Import-Beziehungen können die Namen von öffentlichen Modellelementen eines (importierten) Pakets in den Namensraum eines anderen (importierenden) Pakets übernommen werden.

1. Privater Paket-Import:

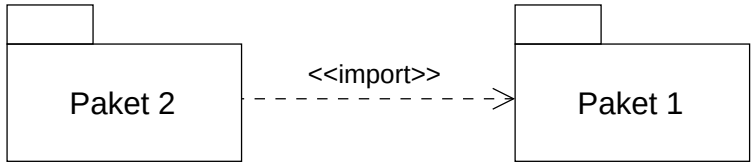


Die Sichtbarkeit der importierten Elemente wird auf "privat" gesetzt.

Beispiel:

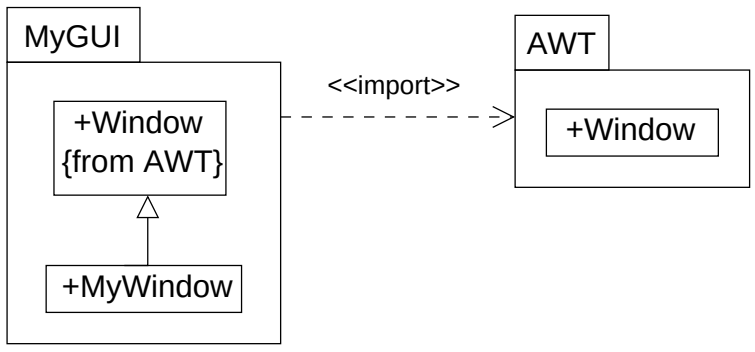


2. Öffentlicher Paket-Import:



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

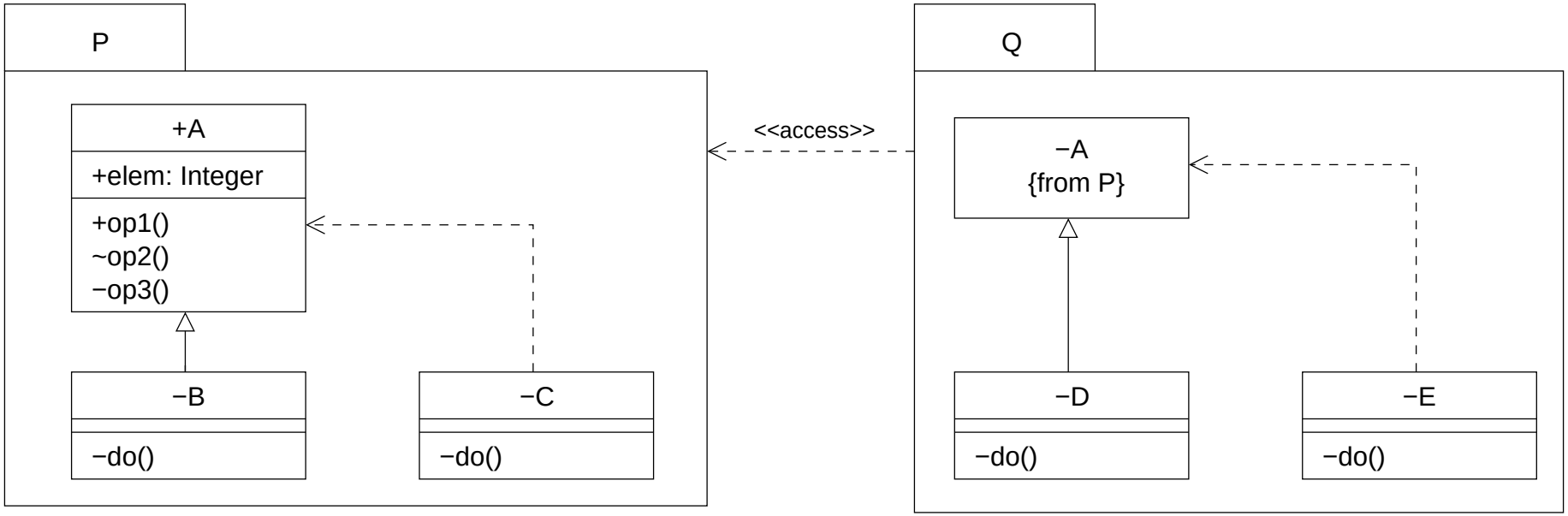
Beispiel:



Implementierung von Paketen in Java

- Für jedes Paket wird ein Verzeichnis mit dem Paketnamen erstellt; für Unterpakete werden Unterverzeichnisse eingerichtet.
- Eine Klasse, die zu einem Paket **P** gehört, wird in einer Datei in dem Verzeichnis **P** implementiert.
- Die Datei darf höchstens eine "**public class**" enthalten. Zu Beginn muss der Paketname angegeben werden: "**package P;**" bzw. "**package P.U;**" falls die Klasse zu einem Unterverzeichnis **U** von **P** gehört.
- Java unterstützt nur den privaten Import von Paketen und Paketelementen: "**import P.*;**" bzw. "**import P.U.*;**" bzw. "**import P.Klassenname;**" bzw. "**import P.U.Klassenname;**"

Beispiel:




```
package P;
public class A {
  public int elem;
  public void op1() { }
  void op2() { }
  private void op3() { }
}
```

```
package P;
class B {
  private void do {
    elem=1;
    op1();
    op2();
    op3();
  }
}
```

```
package P;
class C {
  private void do {
    A a = new A();
    a.elem=1;
    a.op1();
    a.op2();
    a.op3();
  }
}
```

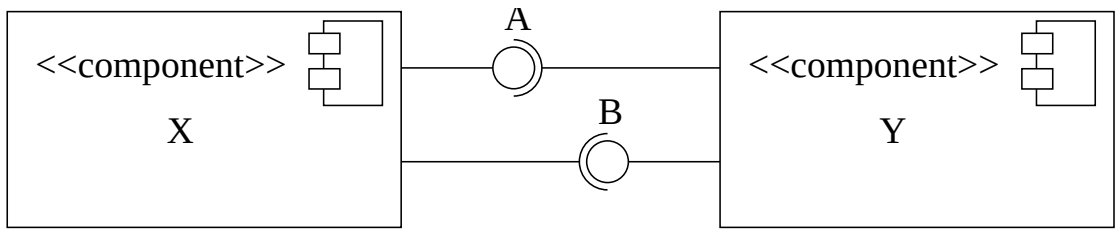
```
package Q;
import P.*;
class D extends A {
  private void do {
    elem=1;
    op1();
    op2();
  }
}
```

```
package Q;
import P.A;
import P.B;
class E {
  private void do {
    A a = new A();
    a.elem=1;
    a.op1();
    a.op2();
    B b = new B();
  }
}
```

Komponenten

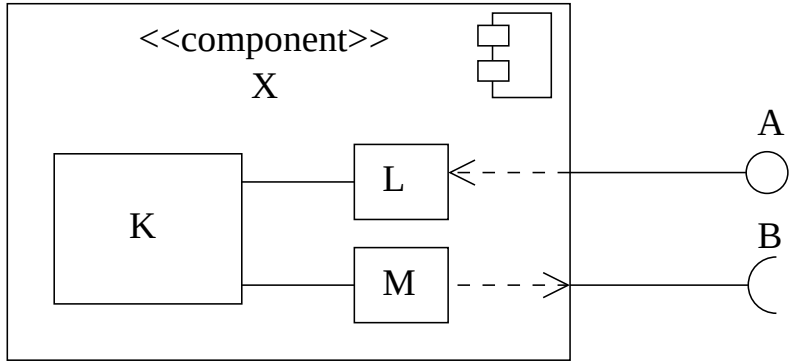
Eine Komponente ist ein modularer Teil eines Systems, der eine komplexe (interne) Struktur verkapselt und mit der Umgebung über Schnittstellen kommuniziert.

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 und von Interfaces 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*.

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 durch Komponenten (mit Schnittstellen).

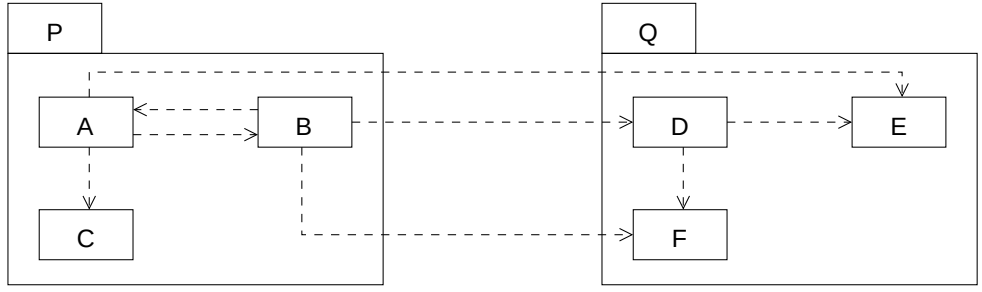
Grundregeln

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

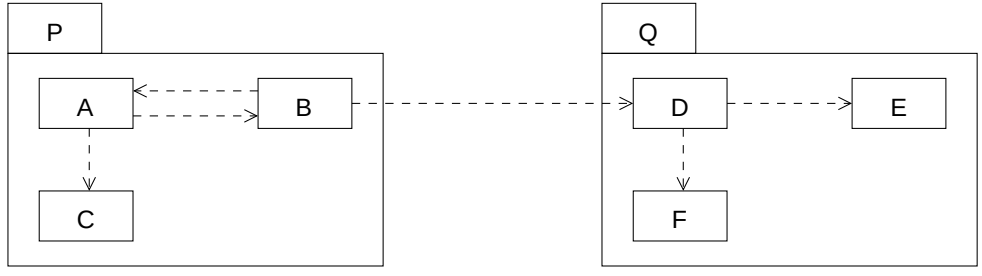
Vorteil: Leichte Änderbarkeit und Austauschbarkeit von einzelnen Teilen.

Beispiel:

Subsysteme mit hoher Kopplung



Subsysteme mit geringer Kopplung



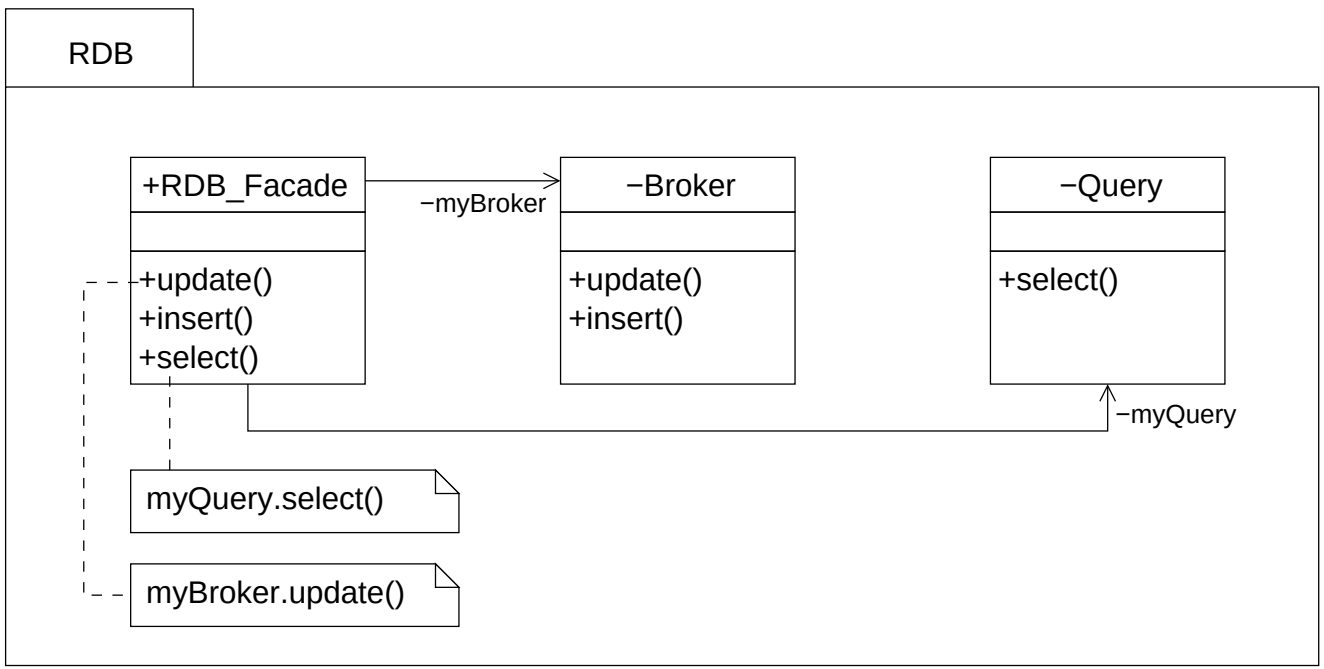
Beachte

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

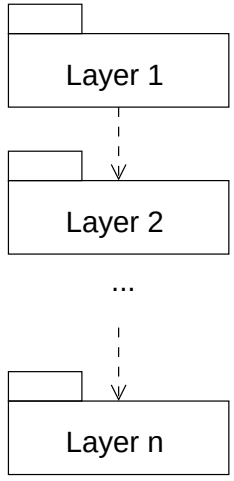
- Hilfsmittel zur Erzielung geringer Kopplung.
- Fassen die Dienste verschiedener Klassen eines Subsystems zusammen und delegieren Aufrufe an die "zuständigen" Objekte.

Beispiel:



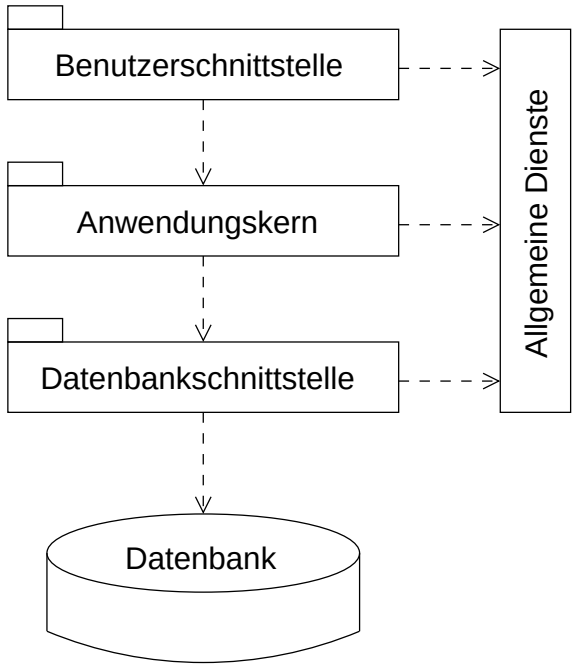
Schichtenarchitekturen

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 (z.B. Web-Anwendungen) 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

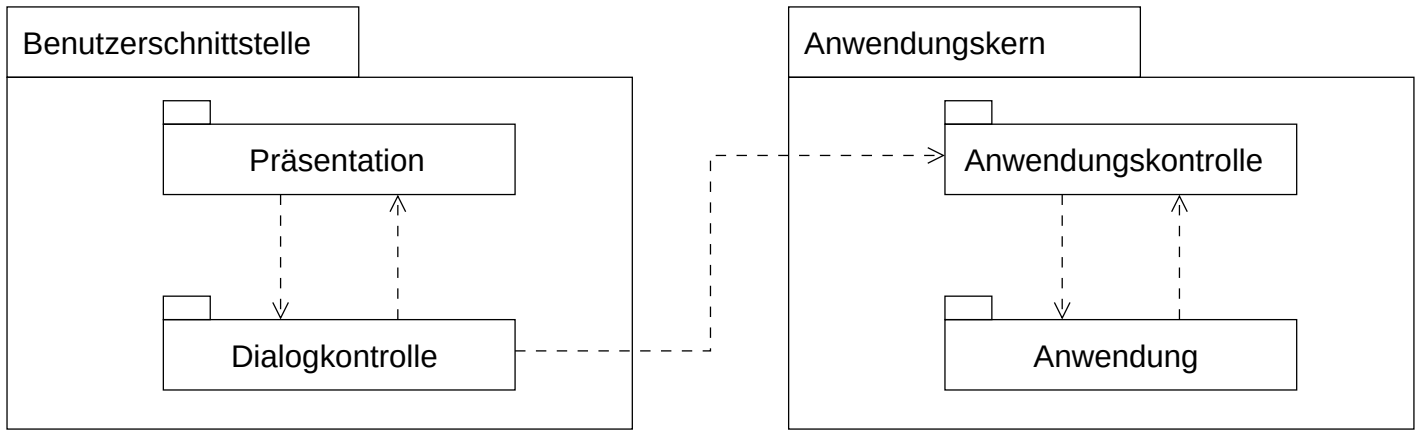
Sorgt für die Speicherung von und den Zugriff auf persistente Daten der Anwendung.

Allgemeine Dienste

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

Kontrollobjekte

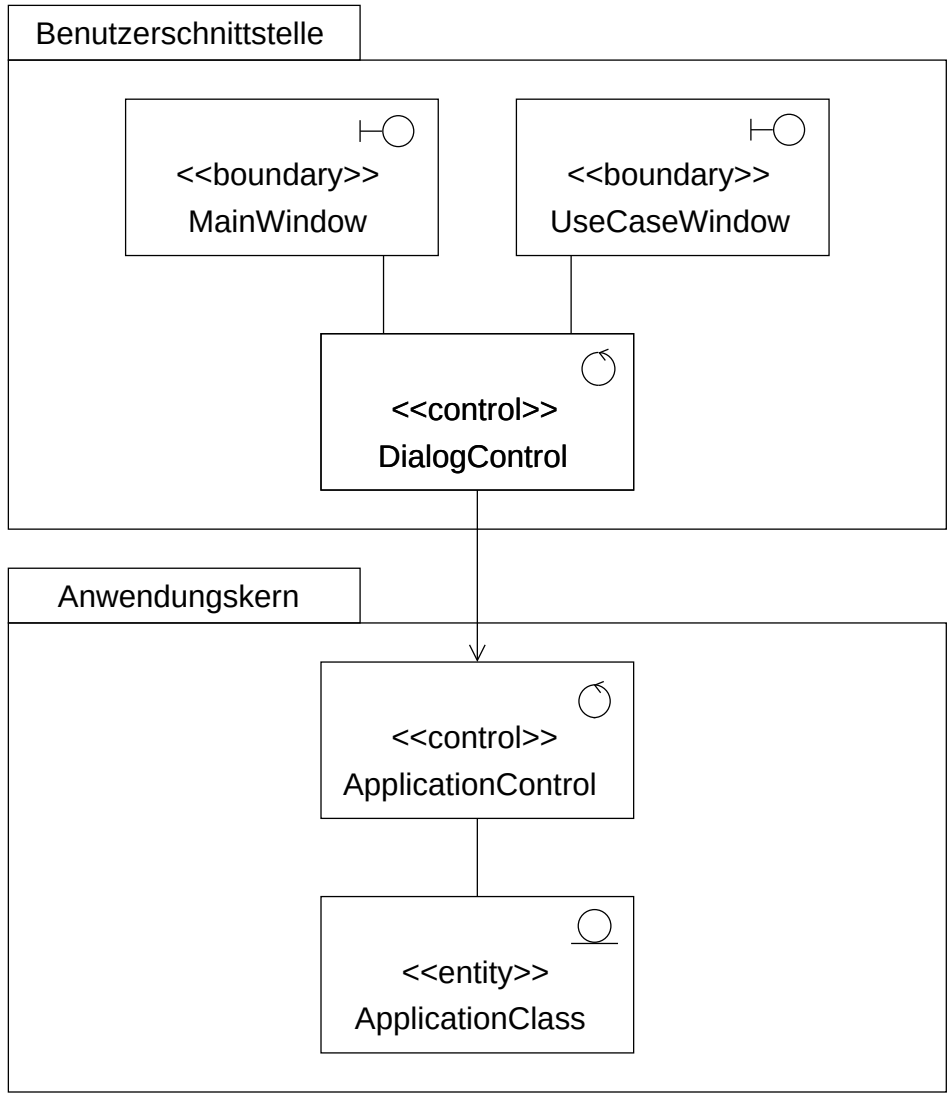
Häufig werden eigene Objekte zur Dialogsteuerung (z.B. Verwaltung mehrerer Fenster) oder zur Steuerung der Aufgaben des Anwendungskerns verwendet (z.B. ein Kontrollobjekt pro Use Case).



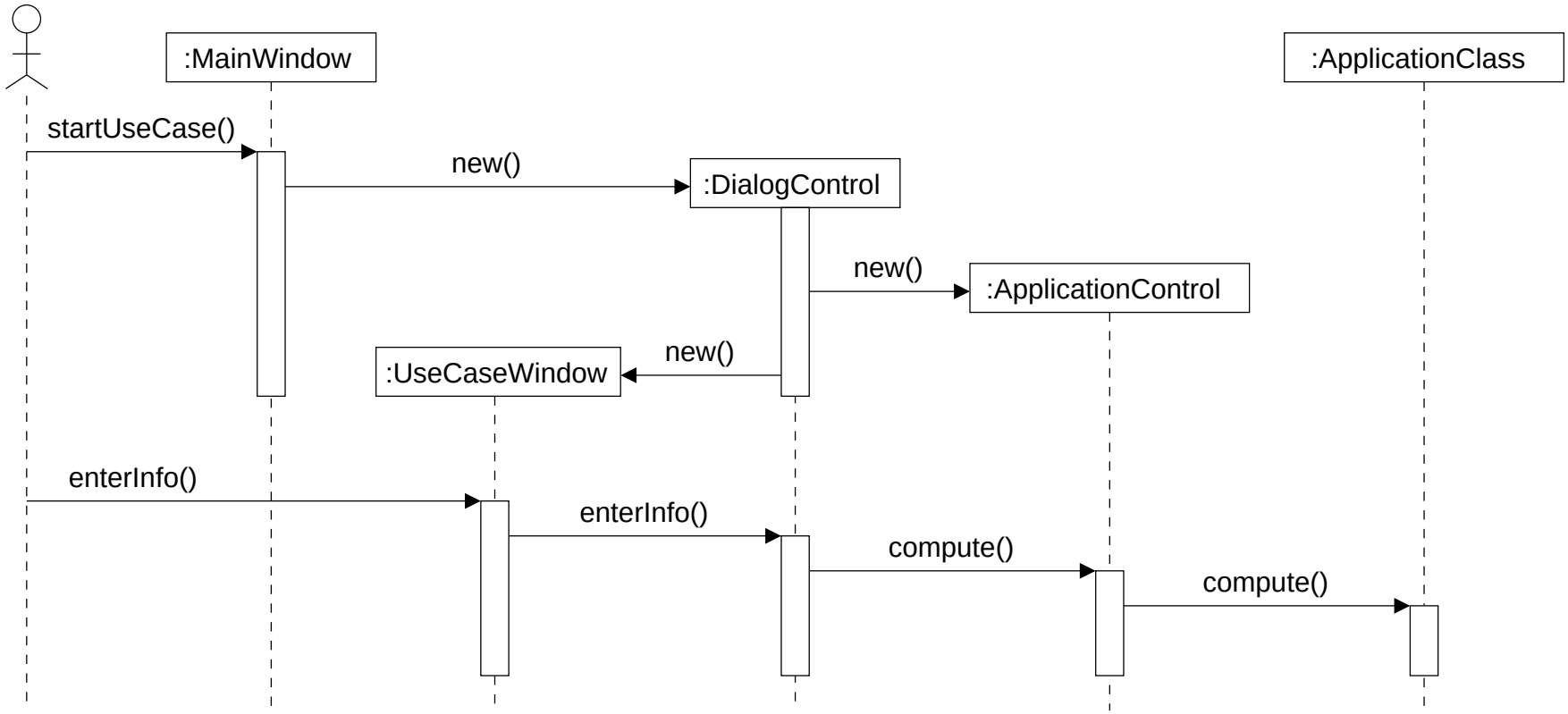
Bemerkung

- Kontrollobjekte haben häufig ein interessantes Verhalten, das durch Zustandsdiagramme beschrieben werden kann (z.B. ATM).
- Mit Hilfe von Kontrollobjekten wird geringe Kopplung unterstützt.

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!

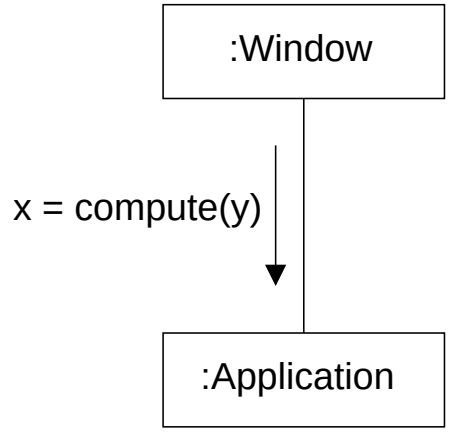
Problem

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).

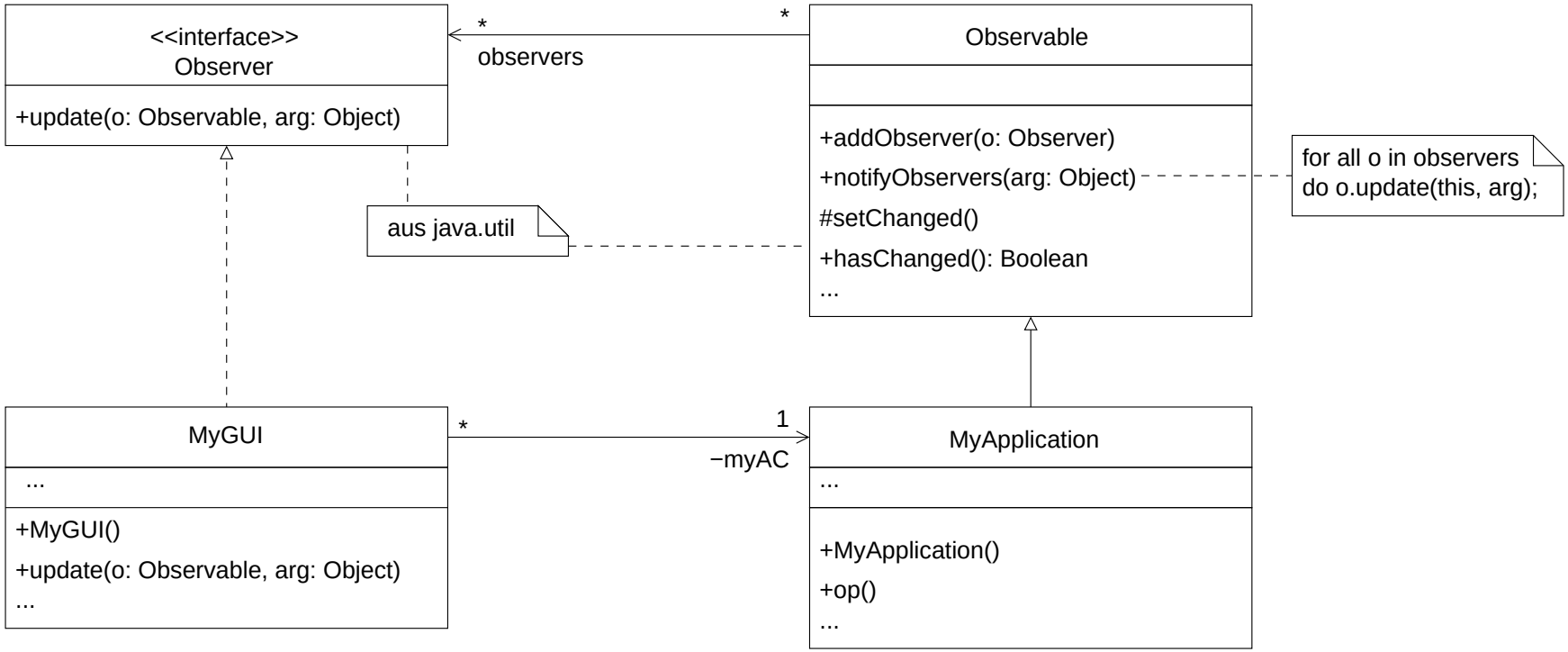
- Indirekte Kommunikation: Event-Manager oder Observer

Indirekte Kommunikation durch Verwendung von Beobachtern

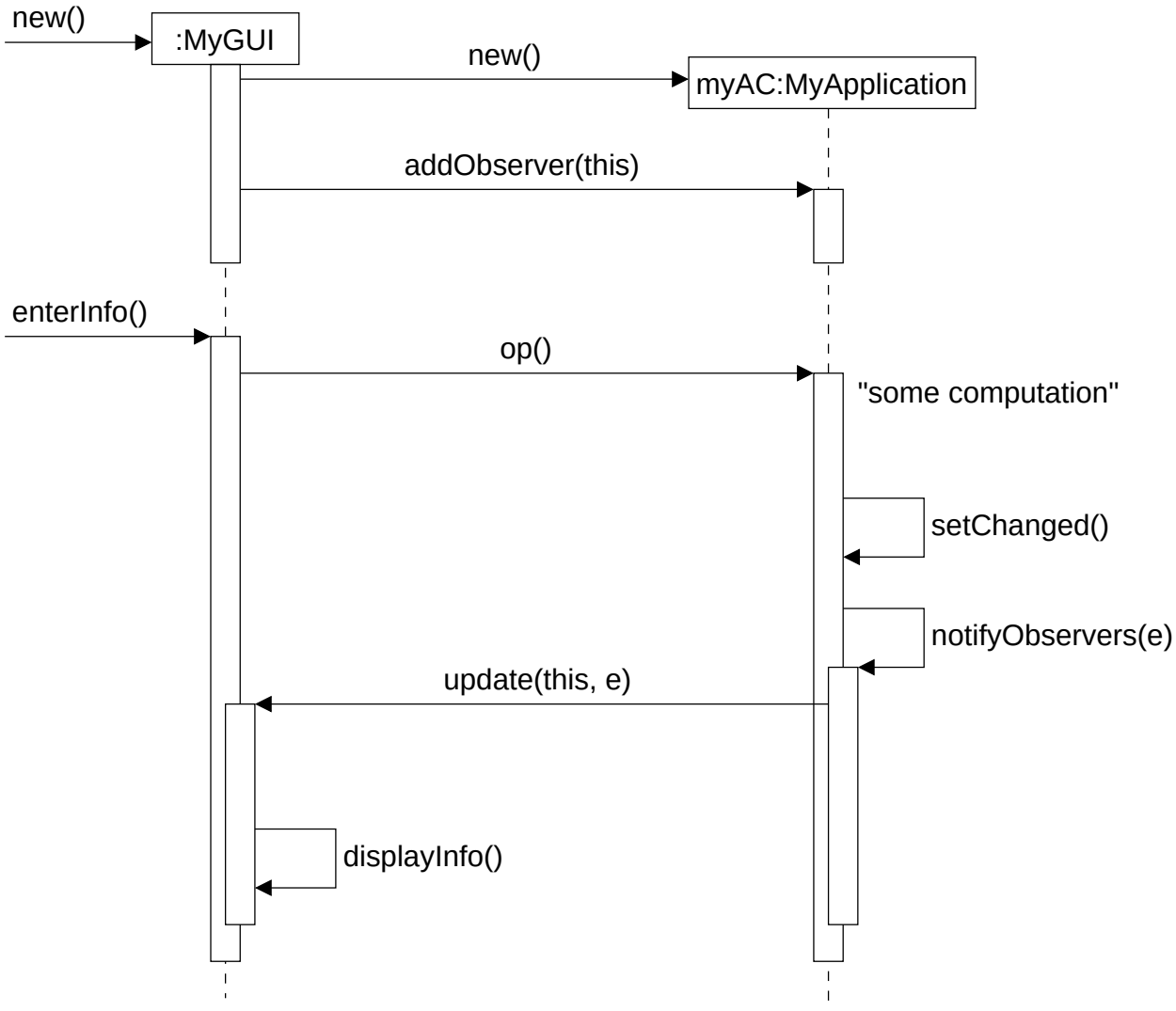
Idee

- 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!)
- Die Sichtbarkeitsregel kann z.B. durch indirekte Kommunikation unter Verwendung von Beobachtern (Observer) realisiert werden.

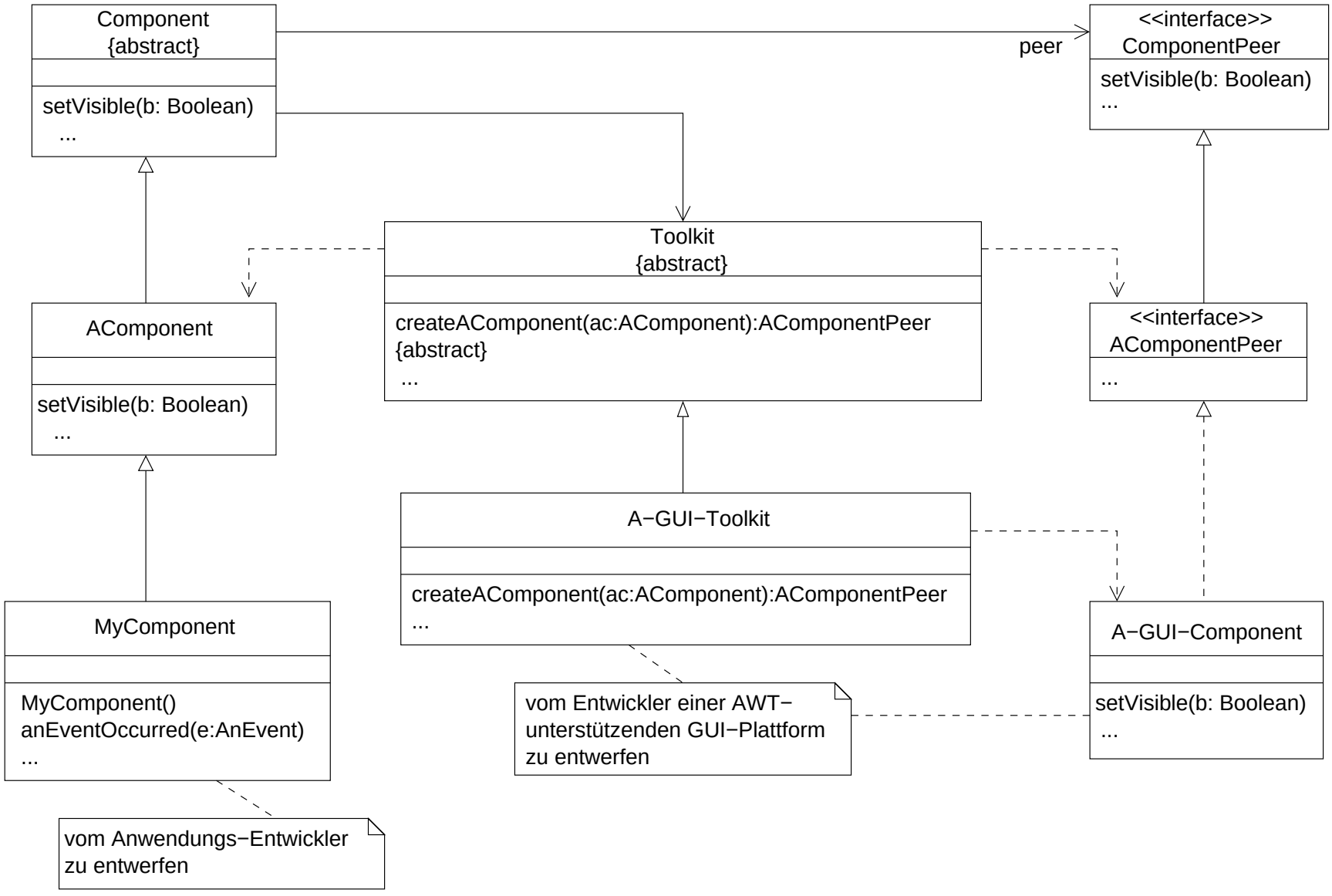
4.4 Entwurf von grafischen Benutzerschnittstellen

- 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:* plattformunanbhängige Konstruktion von GUIs
- *Entwicklung:*
  - AWT 1.0
  - AWT 1.1 (neues Event-Handling mit "Listenern")
  - Swing (ergänzt AWT und ersetzt die meisten AWT-Komponenten durch "Lightweight"-Komponenten)

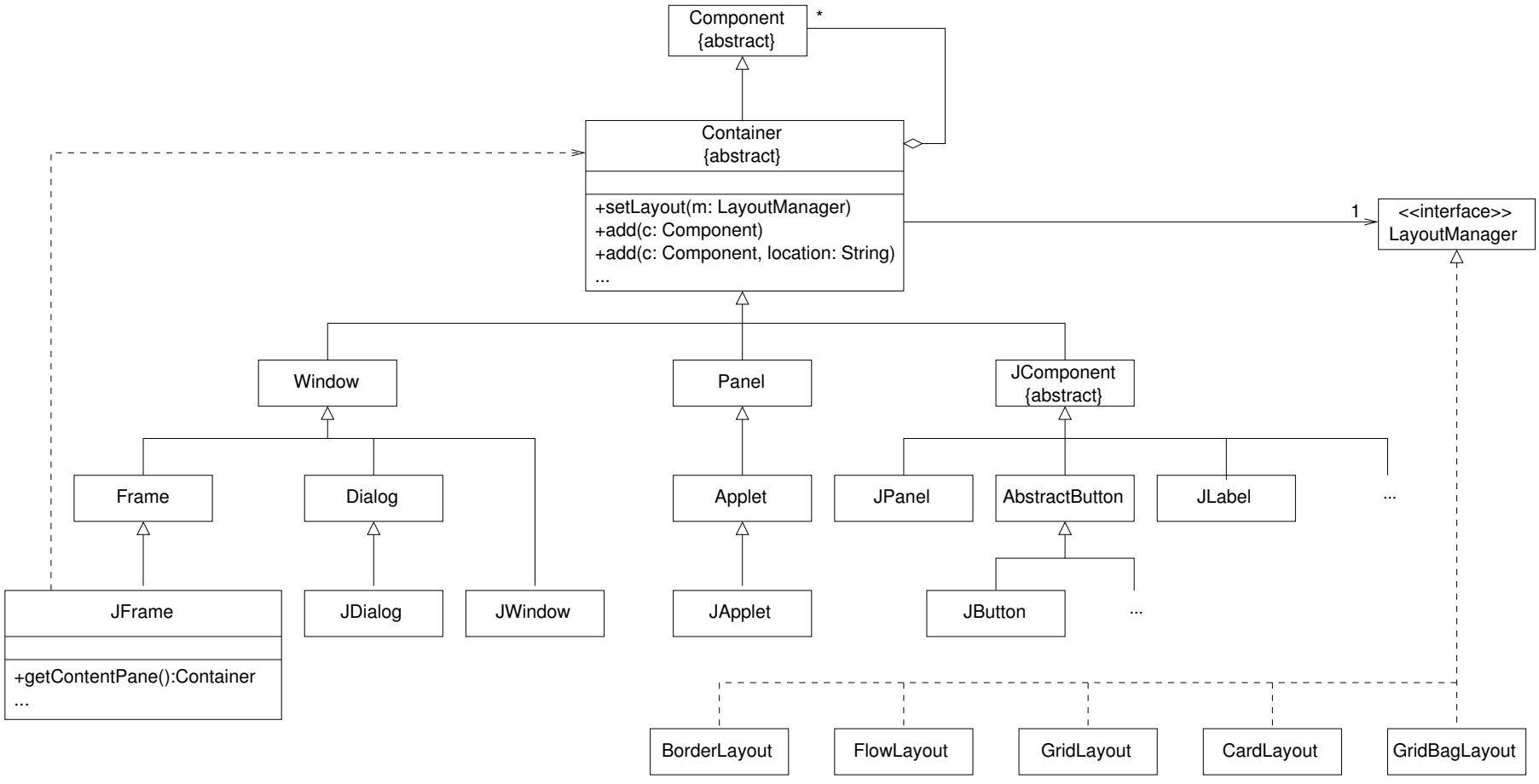
1. Grundkonzepte von AWT



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).

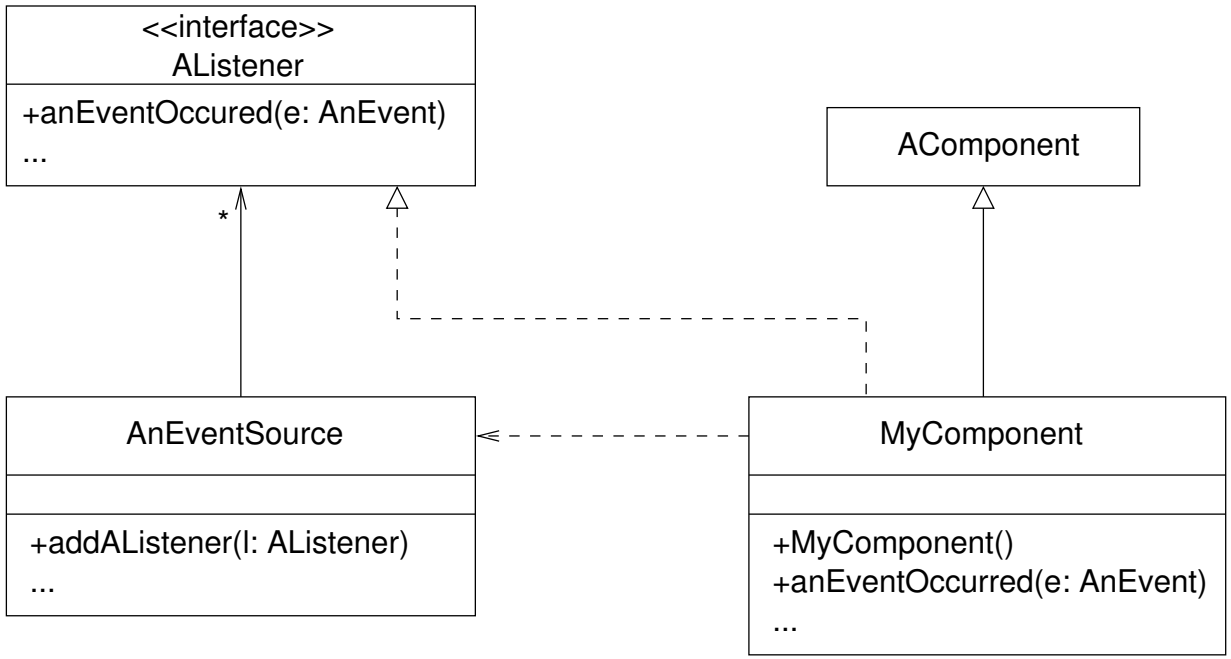
2. Komponenten-Hierarchie von Swing/AWT



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).

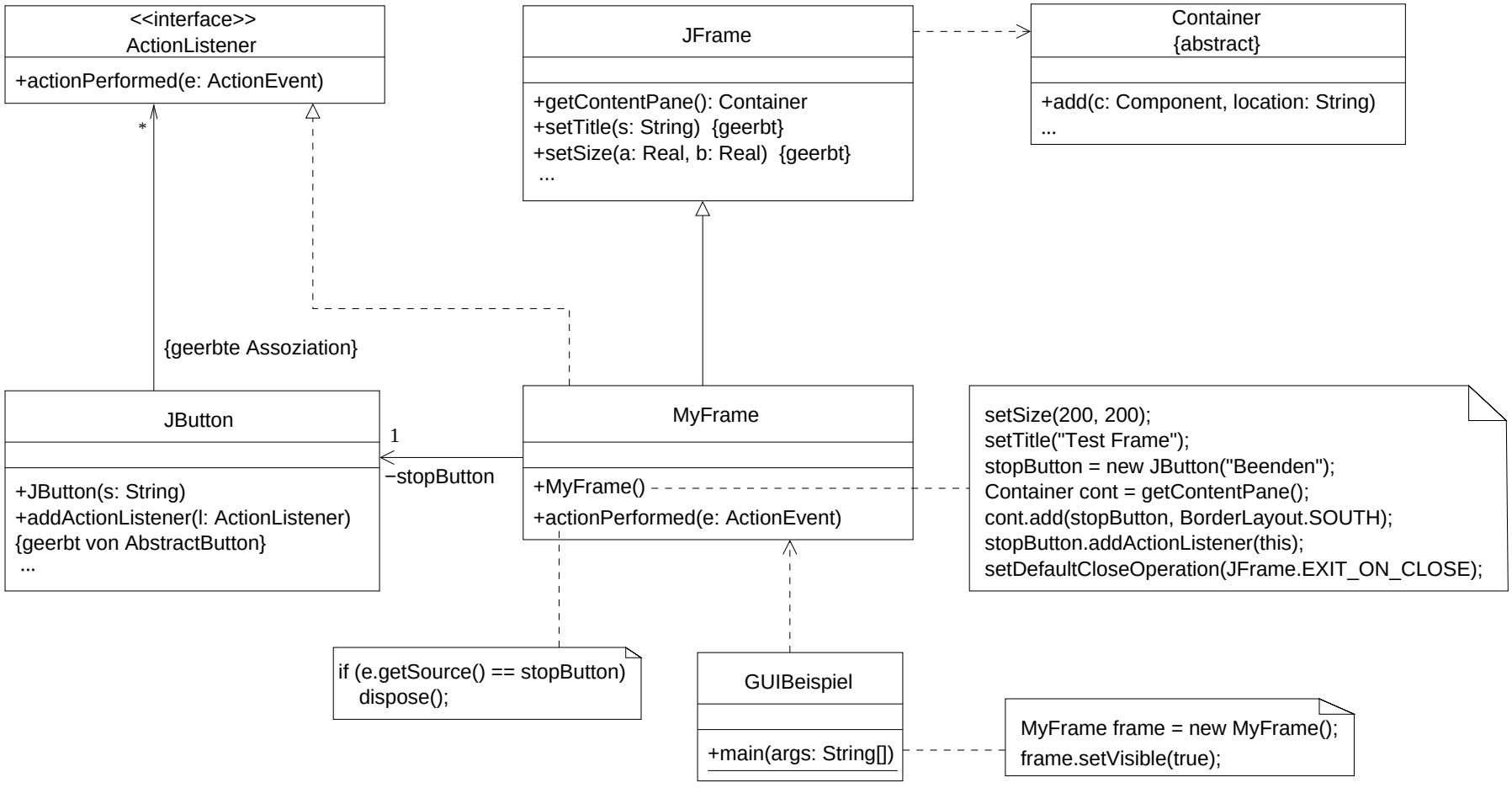
3. Ereignisbehandlung in AWT



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).

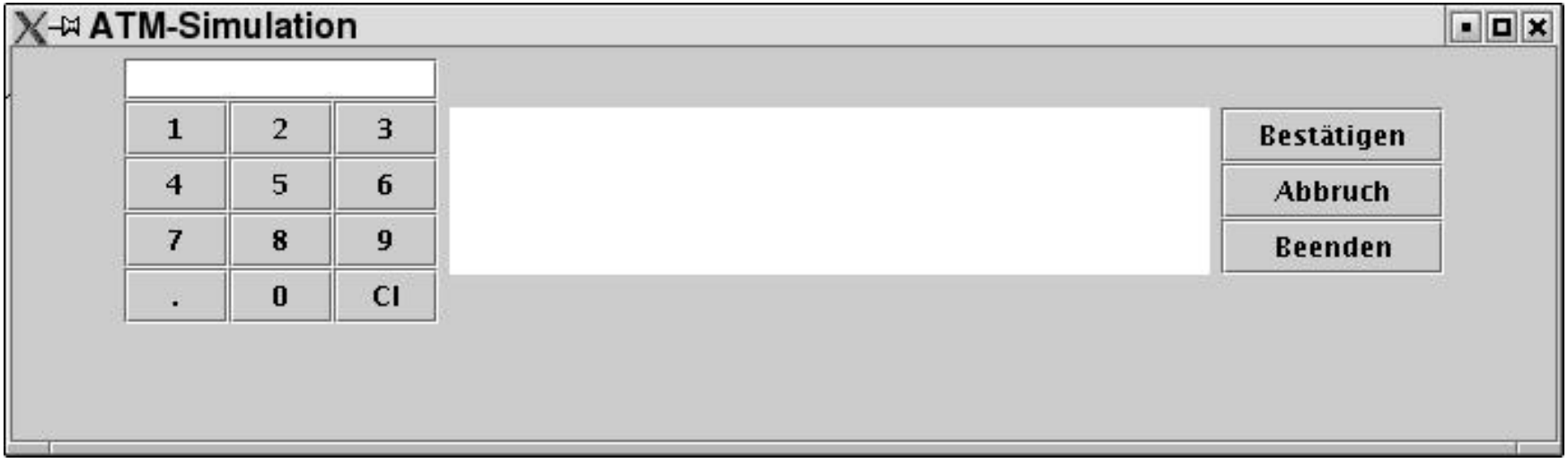
4. GUI-Modellierung: Ein einfaches Beispiel



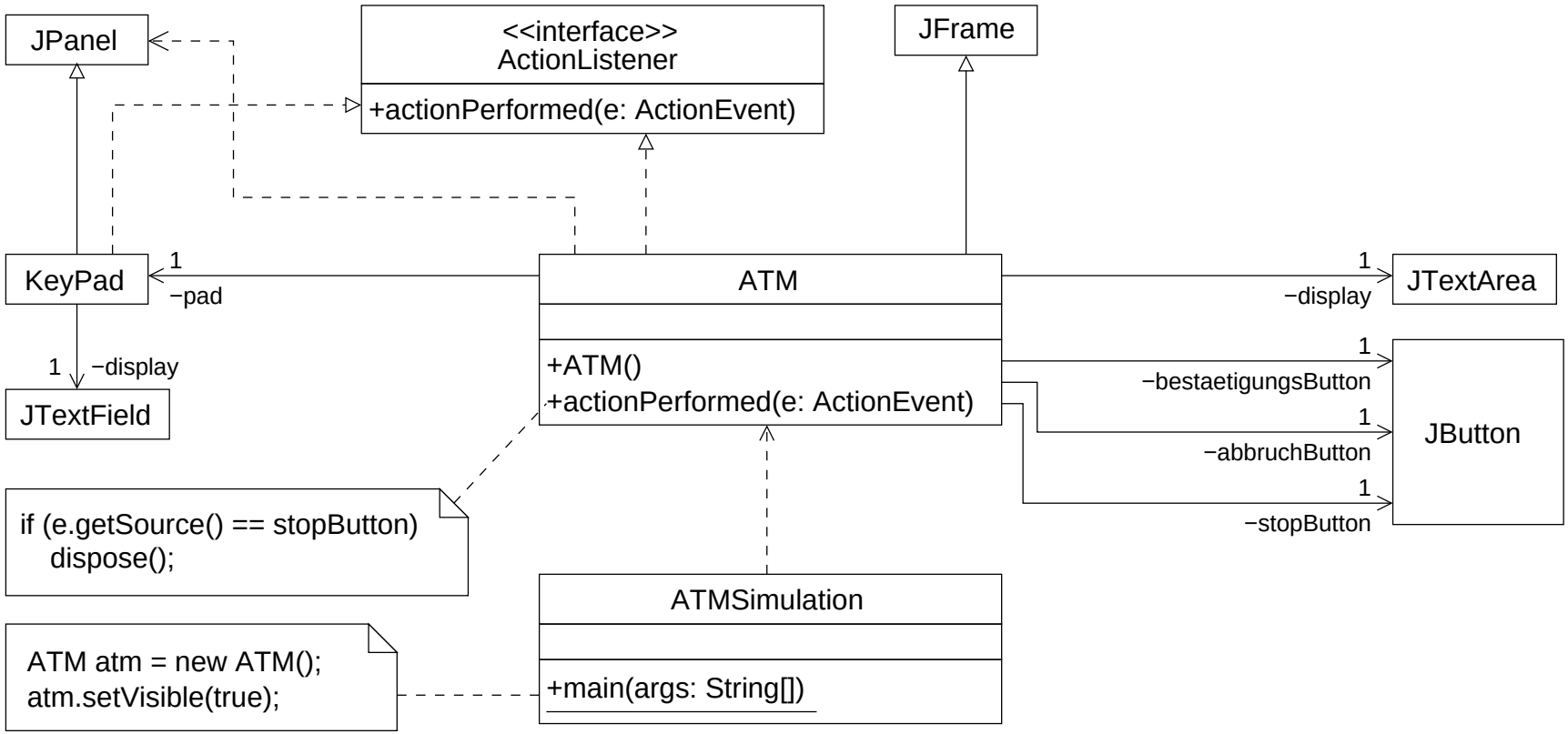

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

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)
      dispose();
  }
}
```

5. Benutzerschnittstelle der ATM-Simulation



Modell der GUI für die ATM-Simulation



```
// Vorlaeufige 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 fuer 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("Bestaetigen");
   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 hinzufuegen
   bestaetigungsButton.addActionListener(this);
   abbruchButton.addActionListener(this);
   stopButton.addActionListener(this);
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
public void actionPerformed(ActionEvent e) {
   if (e.getSource() == stopButton) dispose();
}}
```

```
//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, "Cl");
     add(buttonPanel, BorderLayout.CENTER);
   }
   private void addButton(JPanel buttonPanel, String label) {
     JButton button = new JButton(label);
     buttonPanel.add(button);
     button.addActionListener(this);
   }
```

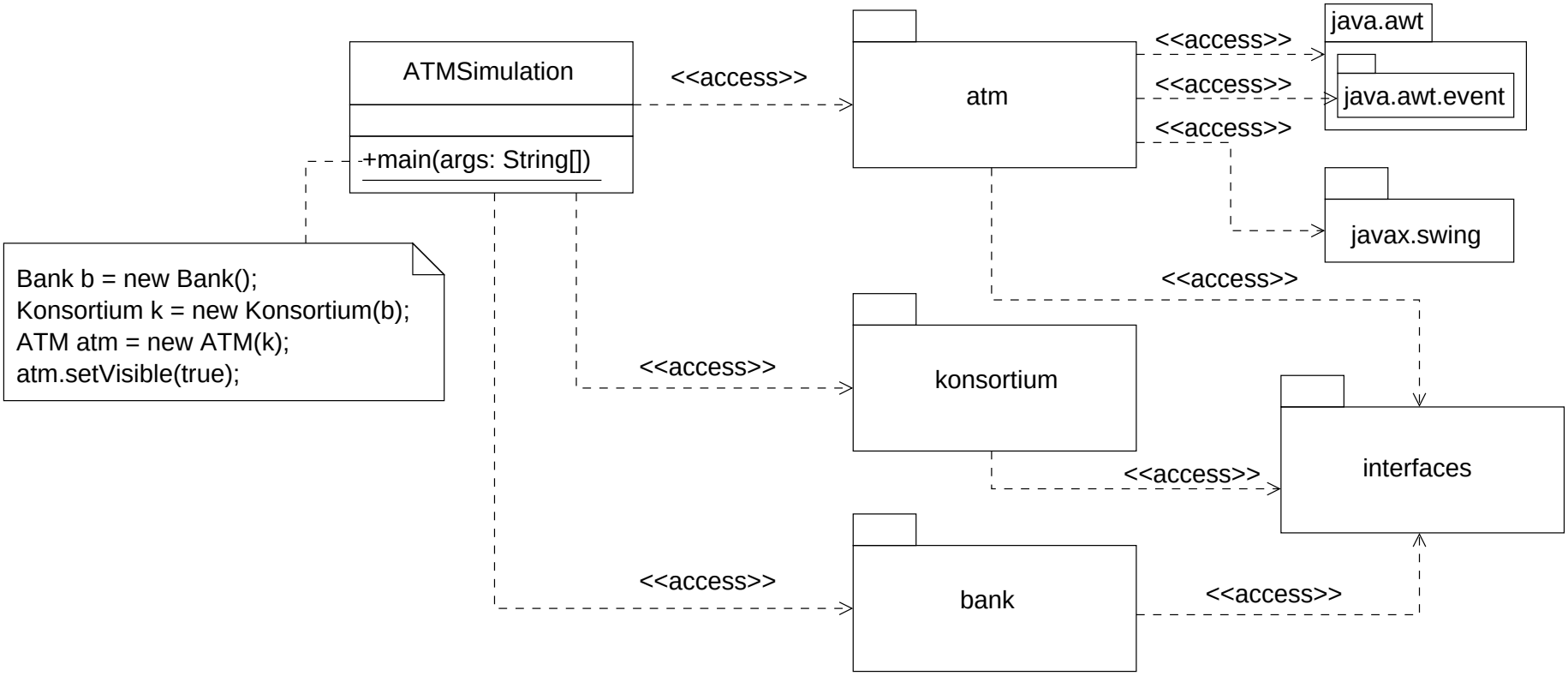
```
  public void actionPerformed (ActionEvent e) {
    JButton source = (JButton)e.getSource();
    String label = source.getText();
    if (label.equals("Cl"))
      clear();
    else
      display.setText(display.getText()+label);
  }
  public double getValue() {
    return Double.parseDouble(display.getText());
  }
  public void clear() {
    display.setText("");
  }
}
```

Zusammenfassung von Abschnitt 4.4

- Zur Programmierung von Benutzerschnittstellen verwendent 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”.

4.5 Realisierung der ATM-Simulation

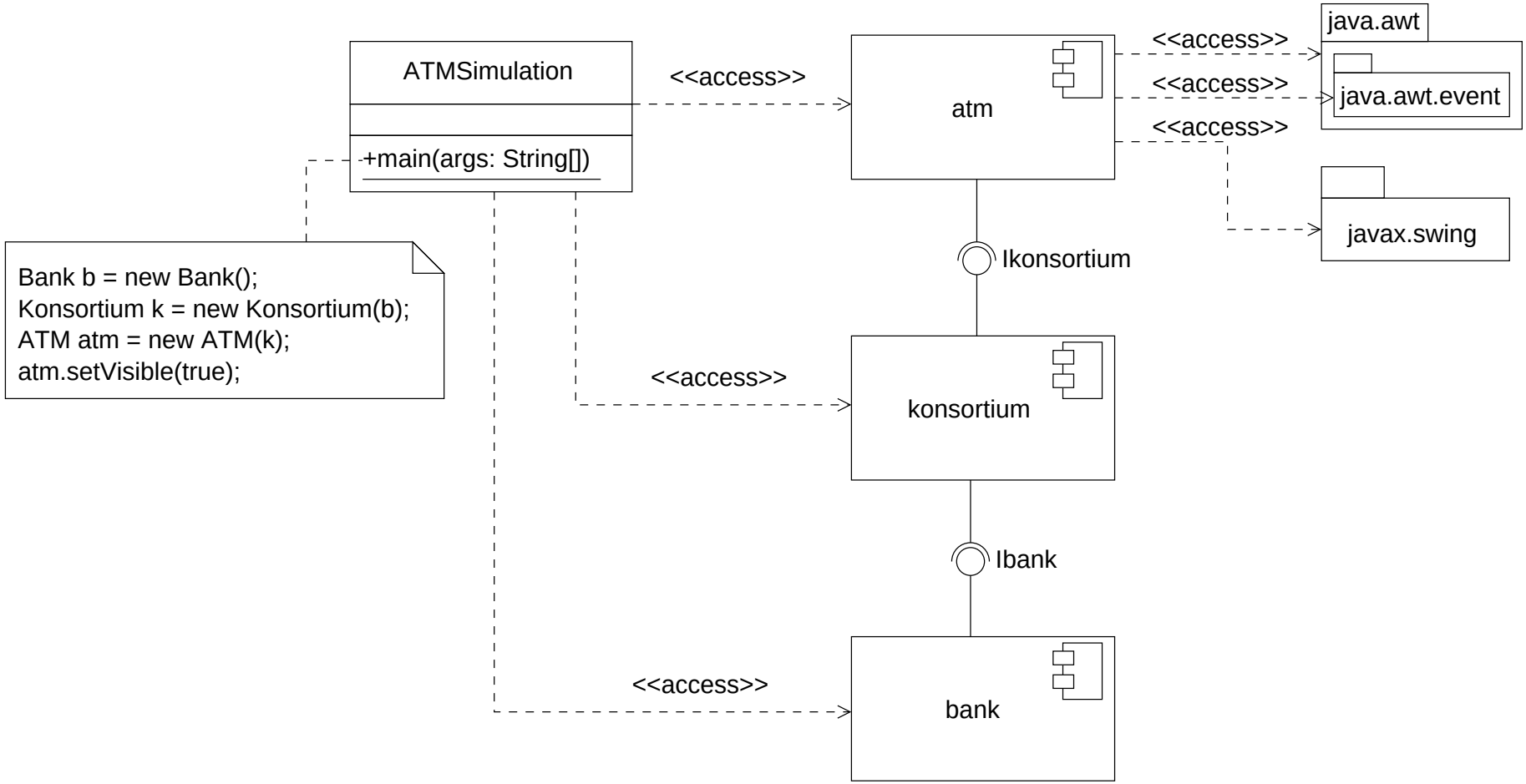
1. Systemarchitektur: Darstellung mit Paketen



Bemerkung:

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

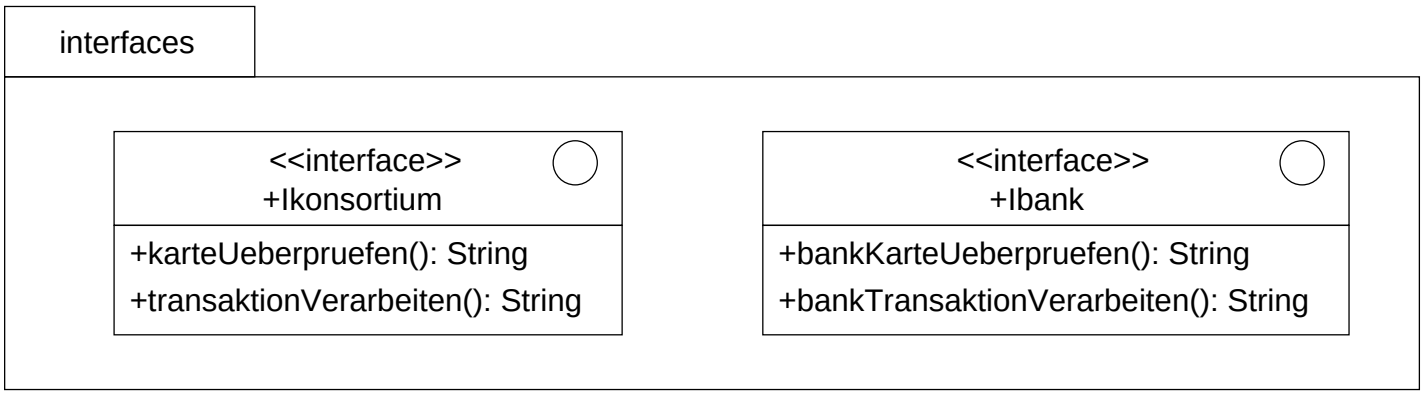
Systemarchitektur: Darstellung mit Komponenten



Vereinfachungen (im Folgenden):

- Beim Lesen der Kreditkarte wird keine Kartennummer 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. Das Paket “interfaces”



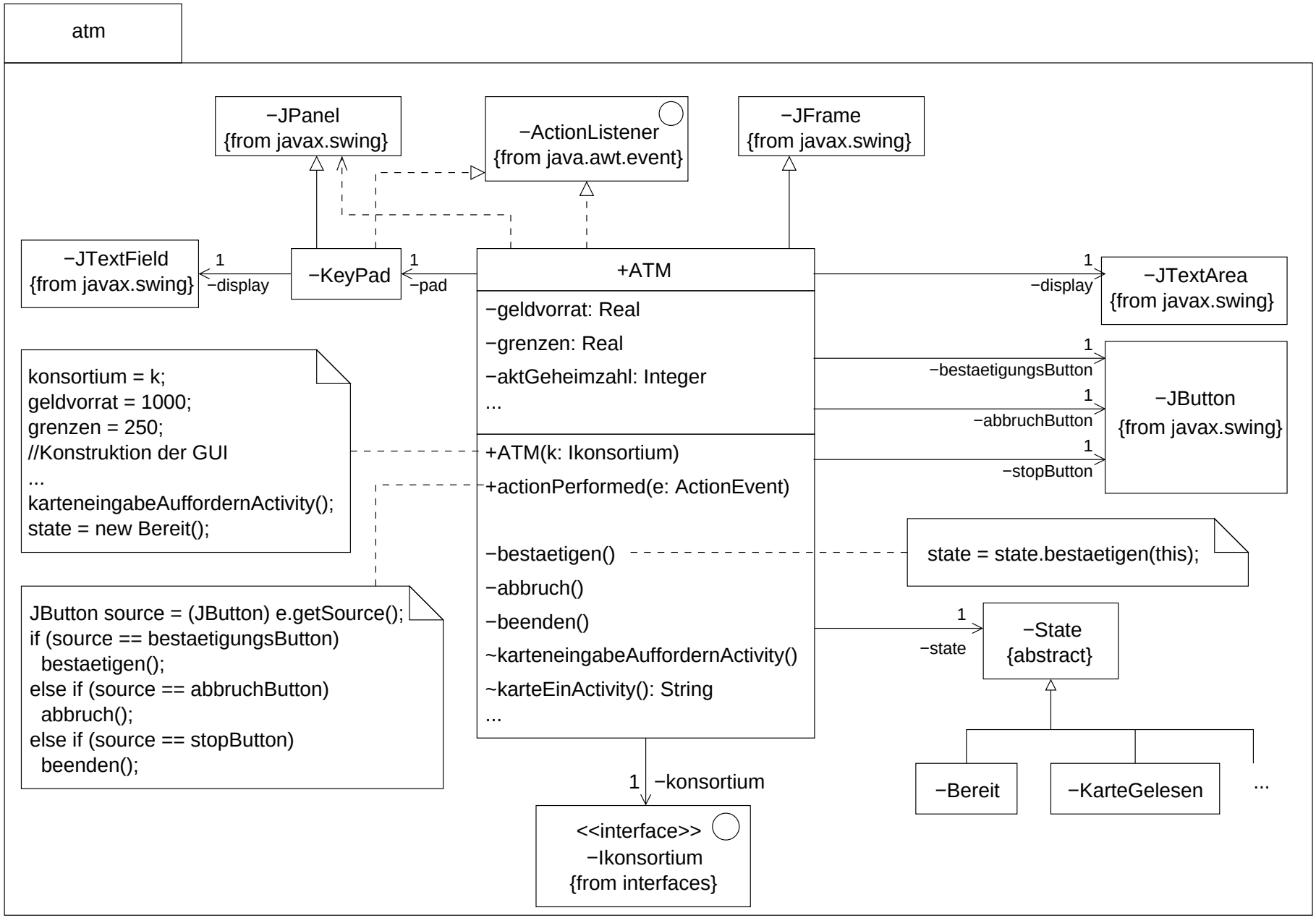
3. Das Subsystem “atm”

Basiert auf

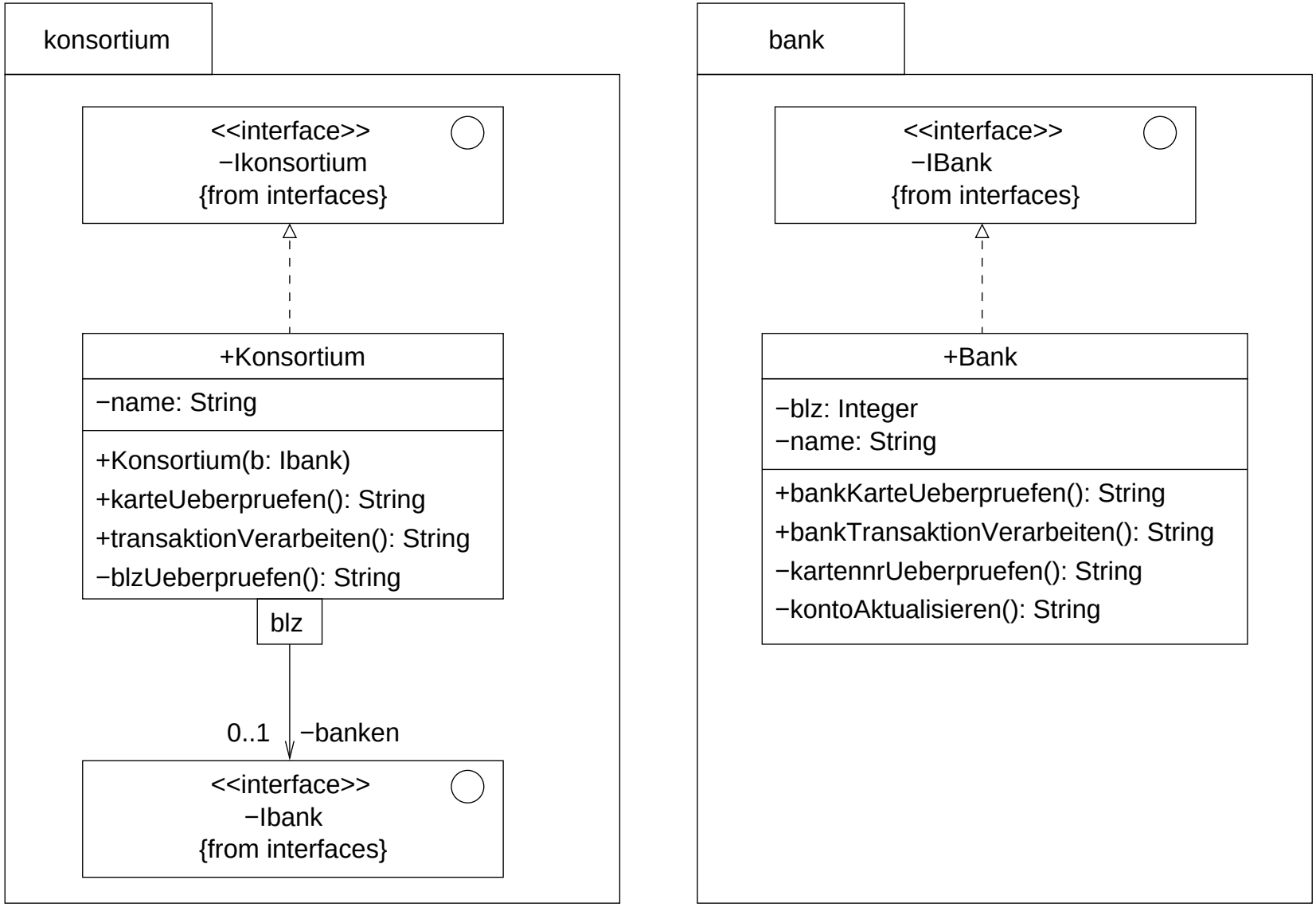
1. Benutzeroberfläche der ATM-Simulation (vgl. Abschnitt 4.4)
2. Pseudo-Code der nicht zustandsabhängigen Operationen des ATM (vgl. Abschnitt 4.1.6)
3. Realisierung des Zustandsdiagramms der ereignisbasierten ATM-Simulation (vgl. Abschnitt 4.2.3).

Beachte:

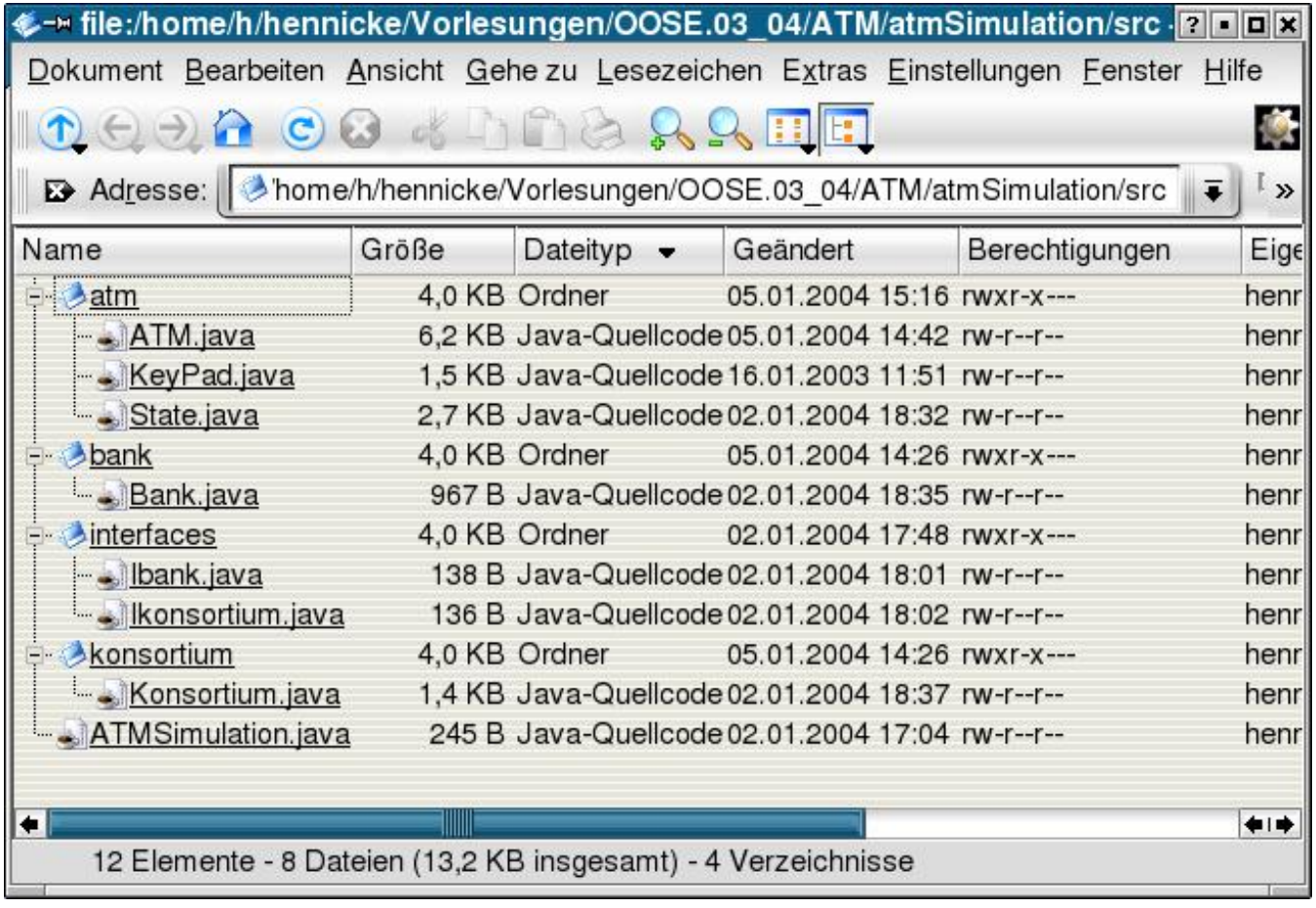
Bei der Integration von 1. und 3. muss der Eventhandling-Mechanismus von AWT berücksichtigt werden, indem eine geeignete Implementierung von “actionPerformed”angegeben wird, so dass bei Drücken eines Buttons die passende Operation der Klasse ATM aufgerufen wird.


atm
−JPanel
{from javax.swing}
−ActionListener
{from java.awt.event}
−JFrame
{from javax.swing}
−JTextField
{from javax.swing}
1
−display
−KeyPad
1
−pad
+ATM
−geldvorrat: Real
−grenzen: Real
−aktGeheimzahl: Integer
...
+ATM(k: Ikonsortium)
+actionPerformed(e: ActionEvent)
−bestaetigen()
−abbruch()
−beenden()
~karteneingabeAuffordernActivity()
~karteEinActivity(): String
...
1
−display
−JTextArea
{from javax.swing}
1
−bestaetigungsButton
1
−abbruchButton
1
−stopButton
−JButton
{from javax.swing}
konsortium = k;
geldvorrat = 1000;
grenzen = 250;
//Konstruktion der GUI
...
karteneingabeAuffordernActivity();
state = new Bereit();
state = state.bestaetigen(this);
JButton source = (JButton) e.getSource();
if (source == bestaetigungsButton)
bestaetigen();
else if (source == abbruchButton)
abbruch();
else if (source == stopButton)
beenden();
1
−state
−State
{abstract}
−Bereit
−KarteGelesen
...
1
−konsortium
<<interface>>
−Ikonsortium
{from interfaces}

4. Die Subsysteme “konsortium”und “bank”



5. Java-Implementierung der ATM-Simulation



Beachte:

Die Verzeichnisstruktur ist konform zur Systemarchitektur.

Klasse ATM Simulation


```
import atm.*;
import konsortium.*;
import bank.*;

class ATMSimulation {
    public static void main(String[] args) {
       Bank b = new Bank();
       Konsortium k = new Konsortium(b);
       ATM atm = new ATM(k);
       atm.setVisible(true);
    }
}
```

Interface Ikonsortium

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

Interface Ibank

```
package interfaces;
public interface Ibank {
   public String bankKarteUeberpruefen();
   public String bankTransaktionVerarbeiten();
}
```

Klasse ATM


```
package atm;

import interfaces.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 fuer GUI
   private KeyPad pad;
   private JTextArea display;
   private JButton bestaetigungsButton;
   private JButton abbruchButton;
   private JButton stopButton;

   //Referenzattribut fuer 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("Bestaetigung");
   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 hinzufuegen
   bestaetigungsButton.addActionListener(this);
   abbruchButton.addActionListener(this);
   stopButton.addActionListener(this);
   //ordnungsgemaesse Beendigung bei Schliessen des Windows
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

   //zur Karteneingabe auffordern
   karteneingabeAuffordernActivity();
   //Zustandsobjekt initialisieren
   state = new Bereit();
}                                          //Ende Konstruktor
public void actionPerformed(ActionEvent e) {
   JButton source = (JButton) e.getSource();
   if (source == bestaetigungsButton)
      bestaetigen();
   else if (source == abbruchButton)
      abbruch();
   else if (source == stopButton)
      beenden();
}
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);
}
```

```
//Operationen fuer die Aktivitaetszustaende des Zustandsdiagramms

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

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

```
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? (Bestaetigung = Abhebung)");
         return "Karte ok";
      }
      else if (check.equals("falsche BLZ")) {
         display.setText("Karte nicht ok: falsche BLZ!");
         abschlussActivity();
         return "Karte nicht ok";
      }
      else if (check.equals("Karte gesperrt")) {
         display.setText("Karte nicht ok: Karte gesperrt!");
         abschlussActivity();
         return "Karte nicht ok";
      }
      else return "Error";
   }
   else if (check.equals("falsche Geheimzahl")) {
      display.setText("falsche Geheimzahl!");
      display.append("\nGeheimzahl? (Eingabe bestaetigen)");
      return "Geheimzahl falsch";
   }
   else return "Error";
}
```

```
void abhebungActivity() {
   display.setText("Betrag? (Eingabe bestaetigen)");
}

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? (Bestaetigung)");
         return "Transaktion erfolgreich";
      }
      else if (check.equals("Transaktion gescheitert")) {
         display.setText("Transaktion gescheitert!");
         display.append("\nTransaktionsform? (Bestaetigung = Abhebung)");
         return "Transaktion gescheitert";
      }
      else return "Error";
   }
   else if (check.equals("Grenzen ueberschritten")) {
      display.setText("Grenzen ueberschritten!");
      display.append("\nBetrag? (Eingabe bestaetigen)");
      return "Grenzen ueberschritten";
   }
   else return "Error";
   }
```

```
    void geldEntnehmenActivity() {
        display.setText("Fortsetzung? (Bestaetigung = Nein)");
    }
    void abschlussActivity() {
        pad.clear();
        display.append("\nBeleg wird gedruckt");
        display.append("\nKarte wird ausgegeben");
        display.append("\nKarte und Beleg entnehmen? (Bestaetigung)");
    }

    //private Operationen fuer Subaktivitaeten

    private String karteLesen() {
        aktGeheimzahl = (int)pad.getValue();
        pad.clear();
        return "lesbar";
    }
    private String geheimzahlUeberpruefen(int tgz) {
        if (tgz == aktGeheimzahl) return "Geheimzahl ok";
        else return "falsche Geheimzahl";
    }
    private String grenzenUeberpruefen(double b) {
        if (b <= grenzen) return "Grenzen ok";
        else return "Grenzen ueberschritten";
    }
}
```

Klasse State mit Unterklassen


```
/* Das folgende Programm realisiert das Zustandsdiagramm der ATM-Simulation
   durch Zustandsobjekte*/

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

abstract class State {
   State bestaetigen(ATM atm) {
      return this;
   }
   State abbruch(ATM atm) {
      return this;
   }
   State beenden(ATM atm) {
      return this;
   }
}
class Bereit extends State {
   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;
   }
   State beenden(ATM atm) {
      atm.dispose();
      return this; }
}
```

```
class KarteGelesen extends State {

   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;
   }

   State abbruch(ATM atm) {
      atm.abschlussActivity();
      return new BereitZurKartenentnahme();
   }
}

class GeheimzahlUndKarteGeprueft extends State {

   State bestaetigen(ATM atm) {
      atm.abhebungActivity();
      return new TransaktionsformBestimmt();
   }

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

```
class TransaktionsformBestimmt extends State {

   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 ueberschritten"))
         return new TransaktionsformBestimmt(); //oder einfacher: return this;
      else return this;
   }

   State abbruch(ATM atm) {
      atm.abschlussActivity();
      return new BereitZurKartenentnahme();
   }
}

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

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

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

Klasse Konsortium

```
package konsortium;

//beide Interfaces werden benoetigt
import interfaces.*;
import java.util.*;

public class Konsortium implements Ikonsortium {

   private String name;
   private Map<Integer,Ibank> banken = new HashMap<Integer,Ibank>();

   //Bei der Konstruktion des Konsortiums wird genau eine Bank eingefuegt
   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.bankKarteUeberpruefen();
         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";
//zum Testen      return "BLZ falsch";
   }
}
```

Klasse Bank


```
package bank;
import interfaces.Ibank;

public class Bank implements Ibank {
   private int blz;
   private String name;

   public String bankKarteUeberpruefen() {
      String check = kartennrUeberpruefen();
      if (check.equals("gueltig")) 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 "gueltig";
//      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, Vererbung und Operationen. Der ODMG-Standard umfasst: ODL (object definition language) für Schemadeklarationen, OQL (object query language) und Sprachanbindungen an C++, Java und Smalltalk.
- *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

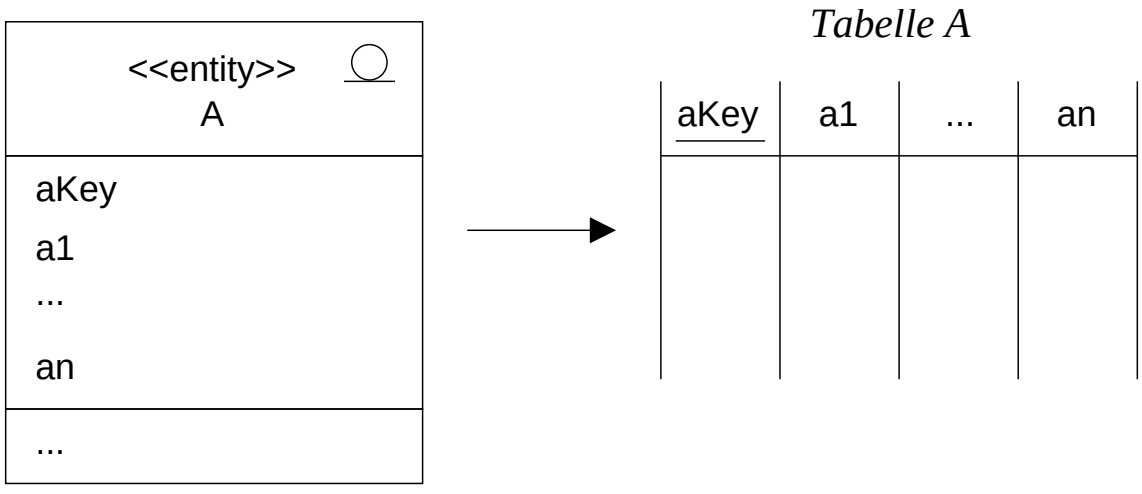
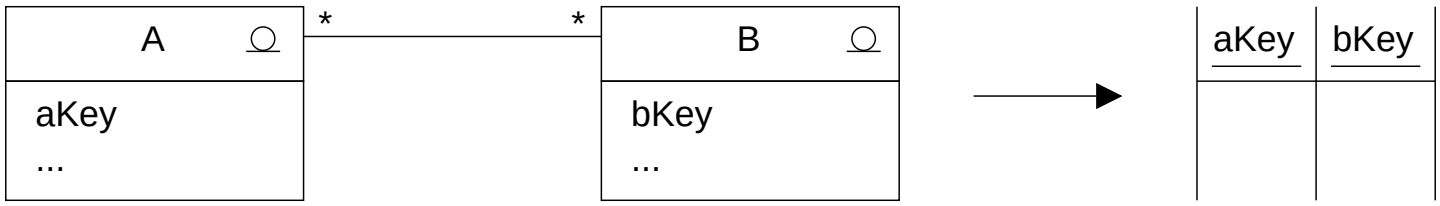


Abbildung von Assoziationen

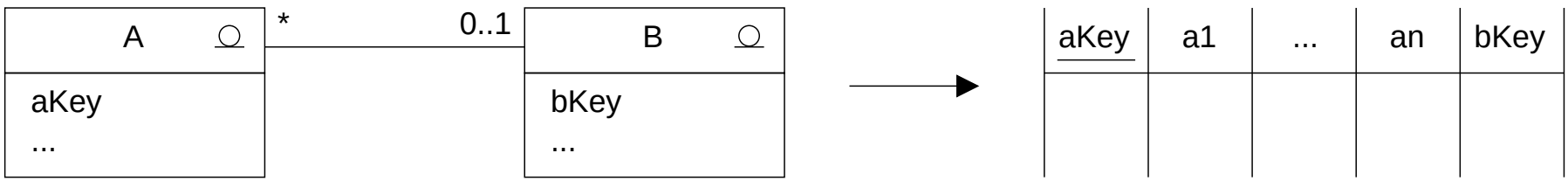
Multiplizität * - *

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



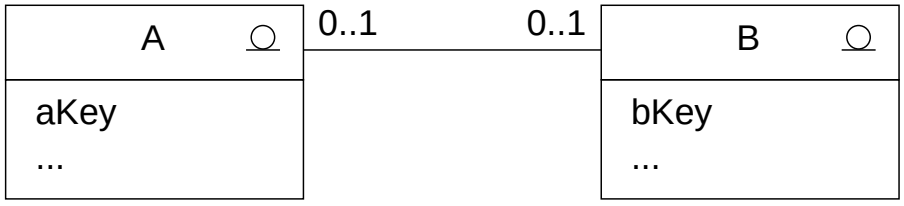
Multiplizität * - 0..1

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.

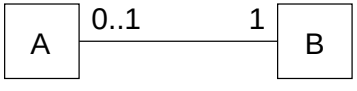


aKey	a1	...	an	bKey

oder

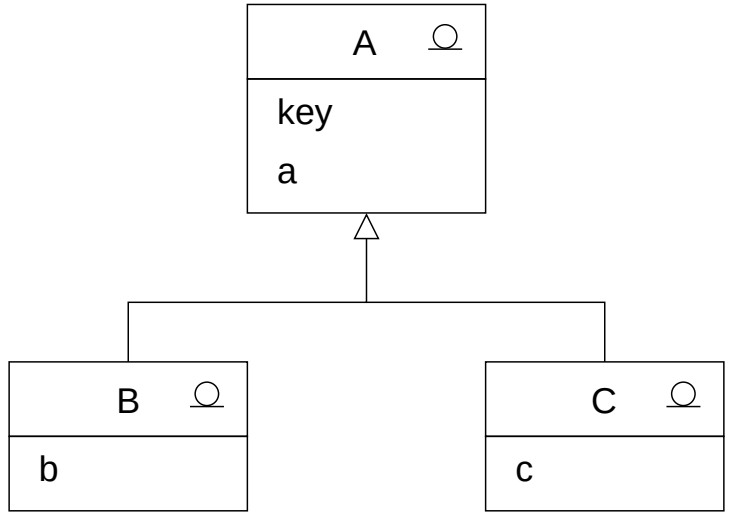
bKey	b1	...	bm	aKey

Bemerkung

Bei 

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

Abbildung von Vererbung



Variante I: Je eine Tabelle pro Klasse

Tabelle A

key	a

Tabelle B

key	b

Tabelle C

key	c

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

Tabelle A

key	a

Tabelle B

key	a	b

Tabelle C

key	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

Tabelle ABC

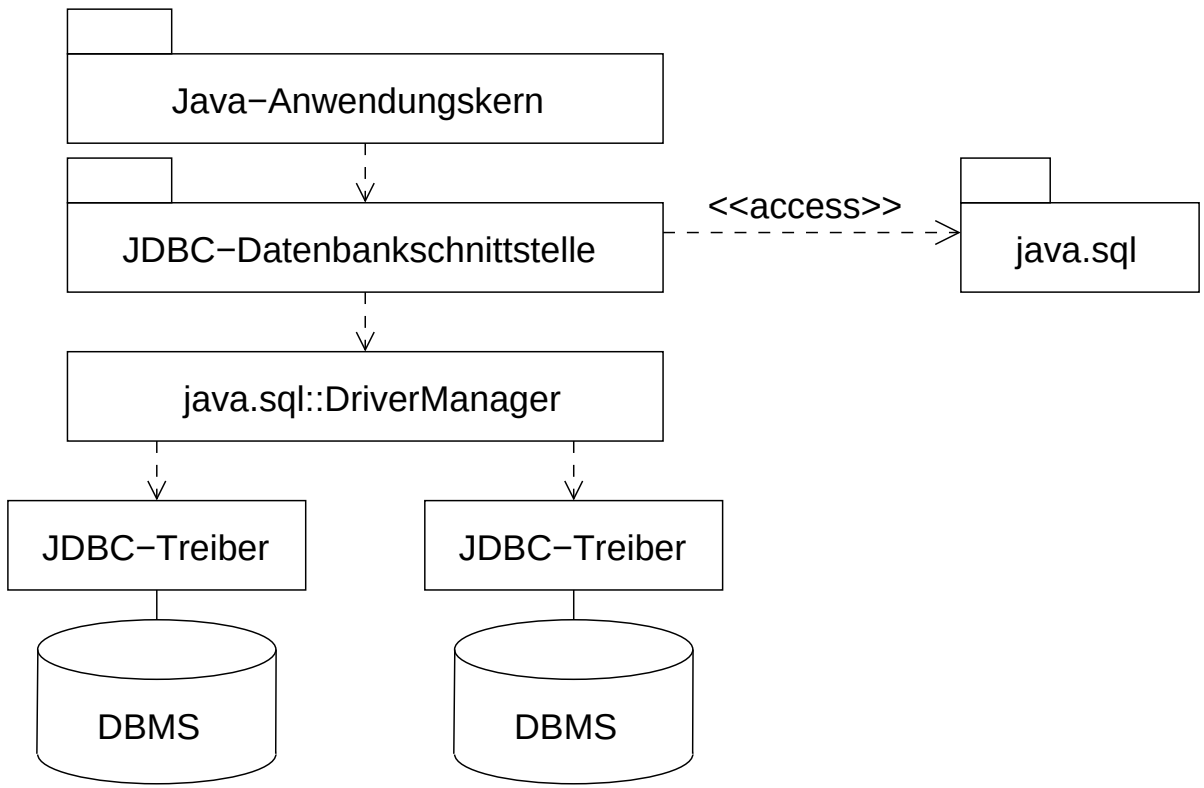
key	a	b	c	Typ

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

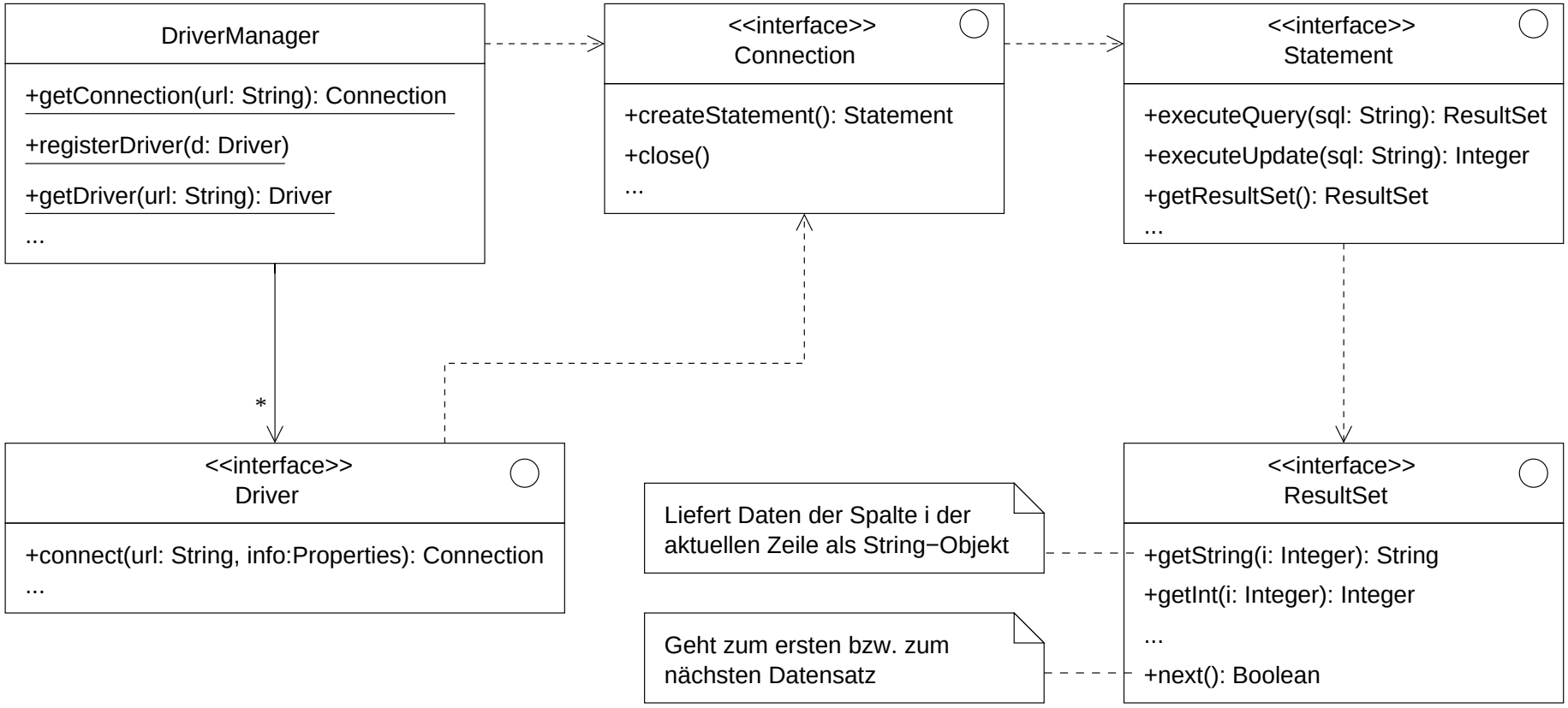
4.6.2 Datenbankanbindung mit der JDBC

- 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



JDBC-Kurzüberblick



- “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 der Operation “next” zeilenweise durchlaufen werden.
- Auf die Felder innerhalb einer Zeile kann mit einer (bzgl. des Spaltentyps) passenden get-Operation zugegriffen werden.

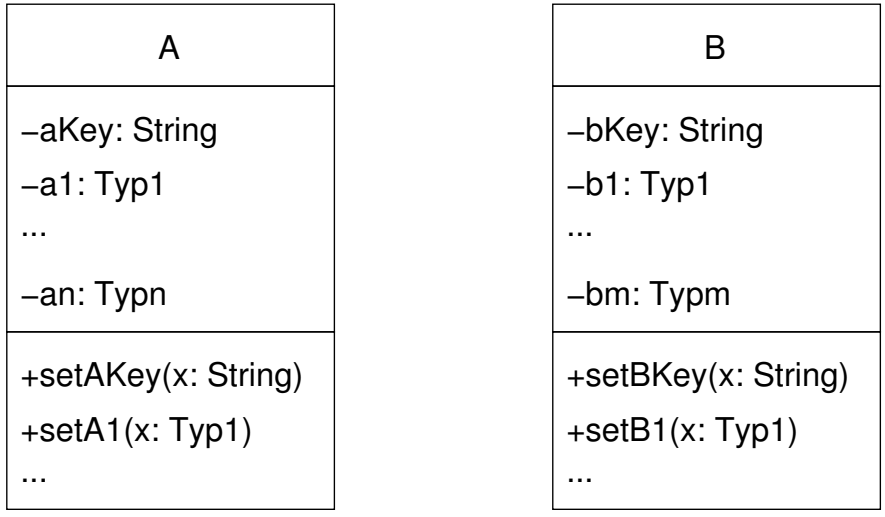
Beispiel:

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

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

4.6.3 Materialisierung von Objekten

Gegeben

Klassen im Anwendungskern:



Tabellen in der relationalen Datenbank:

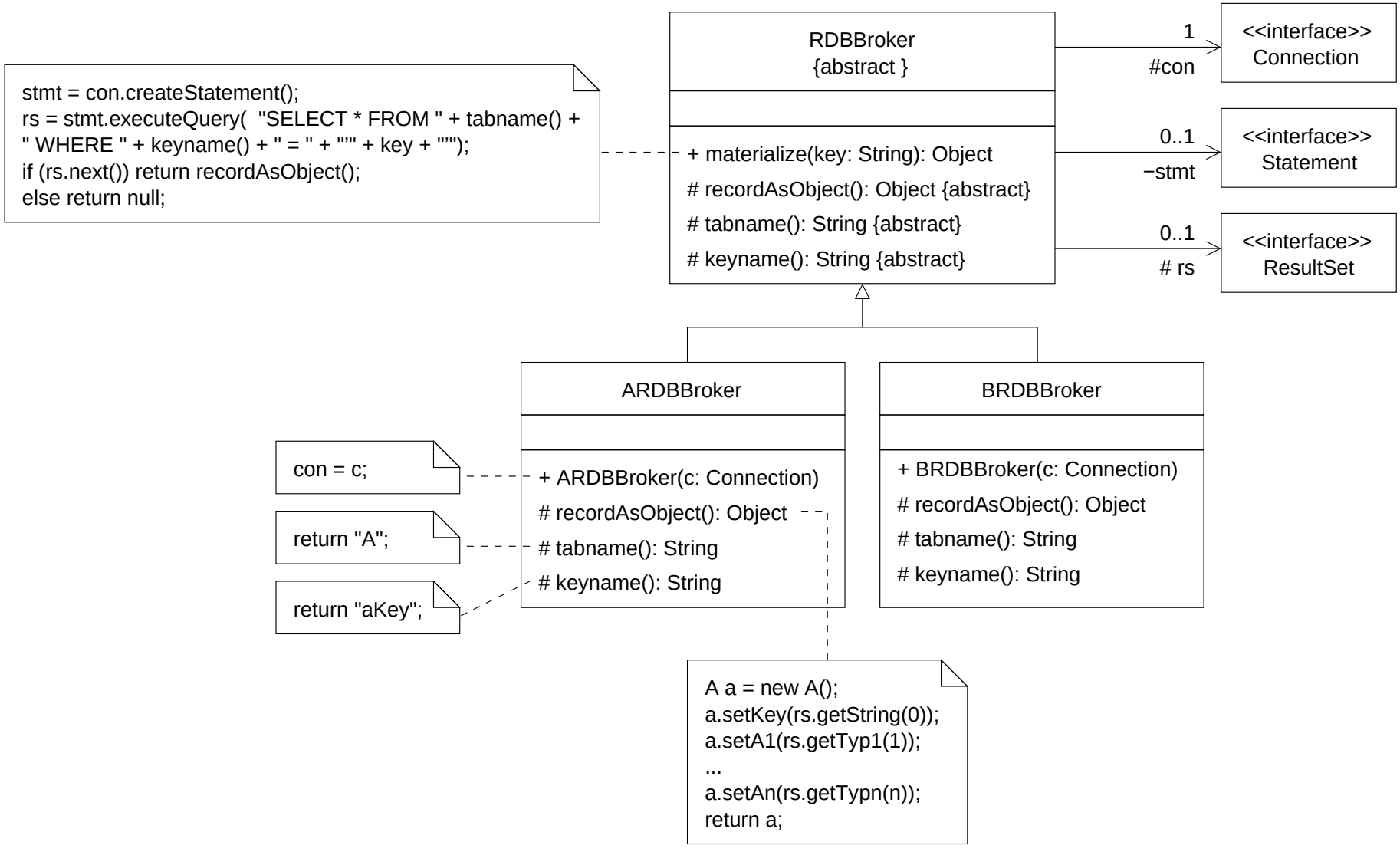
Tabelle A

aKey	a1	...	an

Tabelle B

bKey	b1	...	bm

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();
}
```

Bemerkungen

- Zur Verbesserung der Effizienz werden häufig Zwischenspeicher *(Cache)* verwendet.
- Bei der Materialisierung von Objektstrukturen werden häufig nur die gerade benötigten Objekte geladen *(on-demand materialization)*.
- Persistenz-Frameworks enthalten noch weitere Mechanismen z.B. zur Dematerialisierung und zur Transaktionskontrolle.

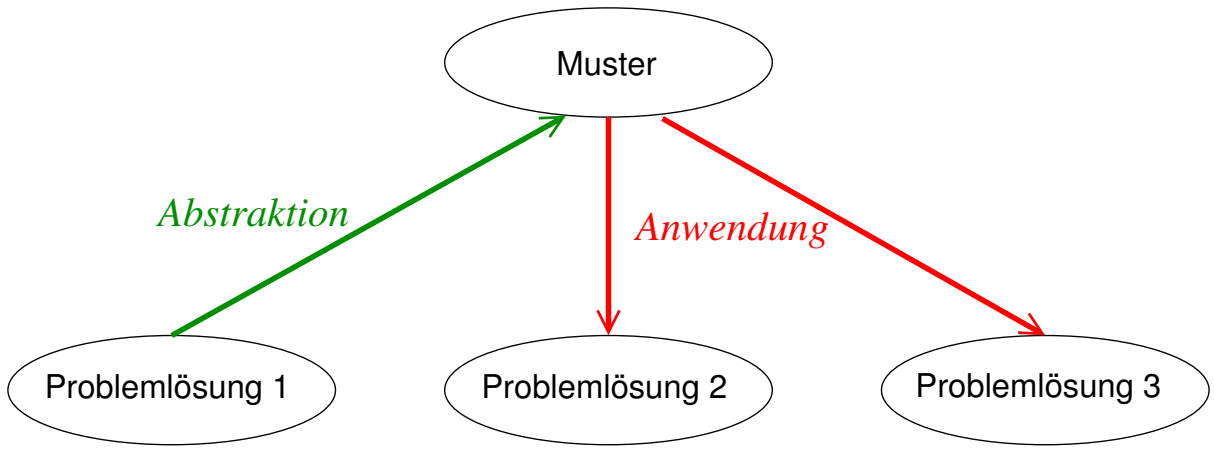
Zusammenfassung von Abschnitt 4.6

- 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.

4.7.1 Grundlagen

Grundidee

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



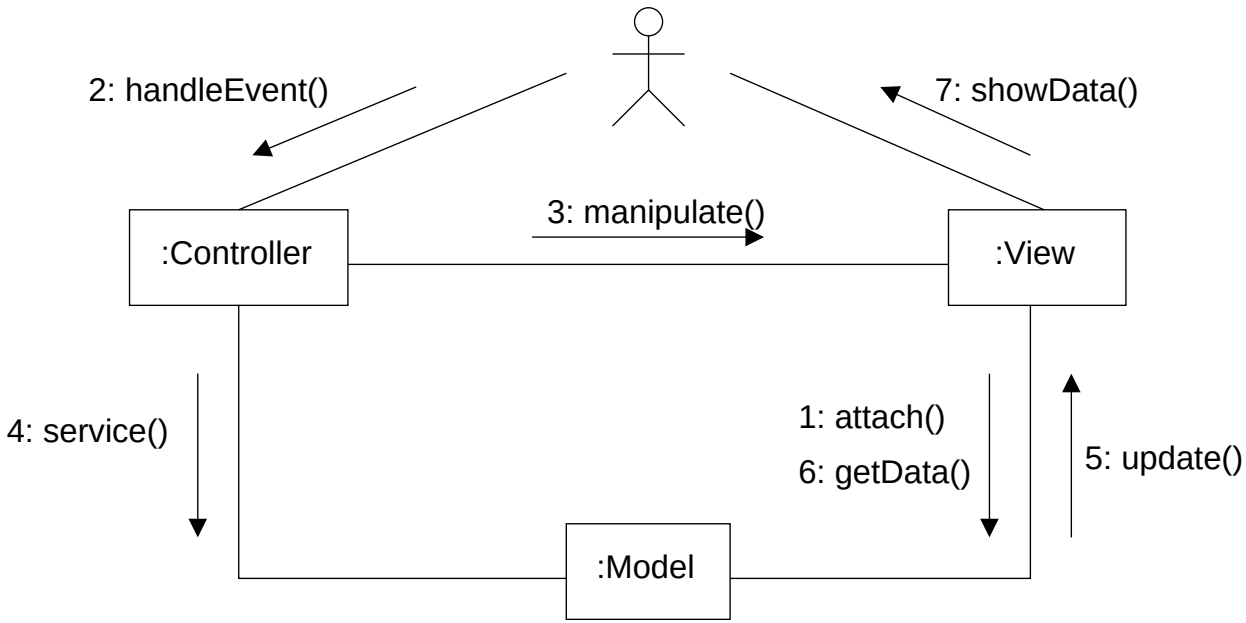
Vorteile

- 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 im Software-Engineering
- 1995 Design Pattern Katalog von Gamma, Helm, Johnson, Vlissides (GoF "Gang of Four")

MVC-Architektur (vereinfacht)



Wesentliche Elemente eines Entwurfmusters

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

4.7.2 Design-Pattern Katalog (GoF)

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 rational 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 oject 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 strukturellen 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.

- **Structural Patterns** (Fortsetzung)
  - **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.

- **Behavorial 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.

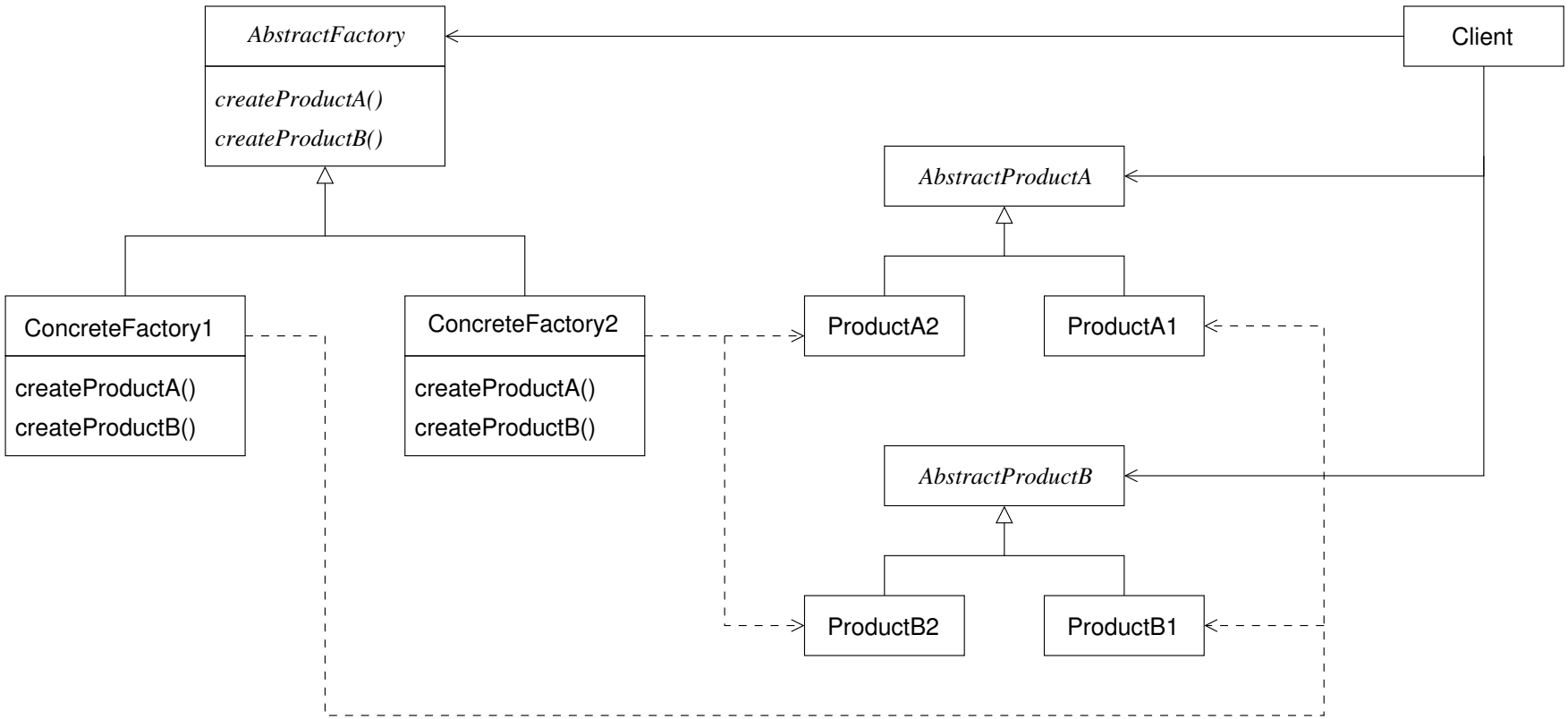
- **Mediator** Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- **Memento** Without violating encapsulation, capture and externalize an 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



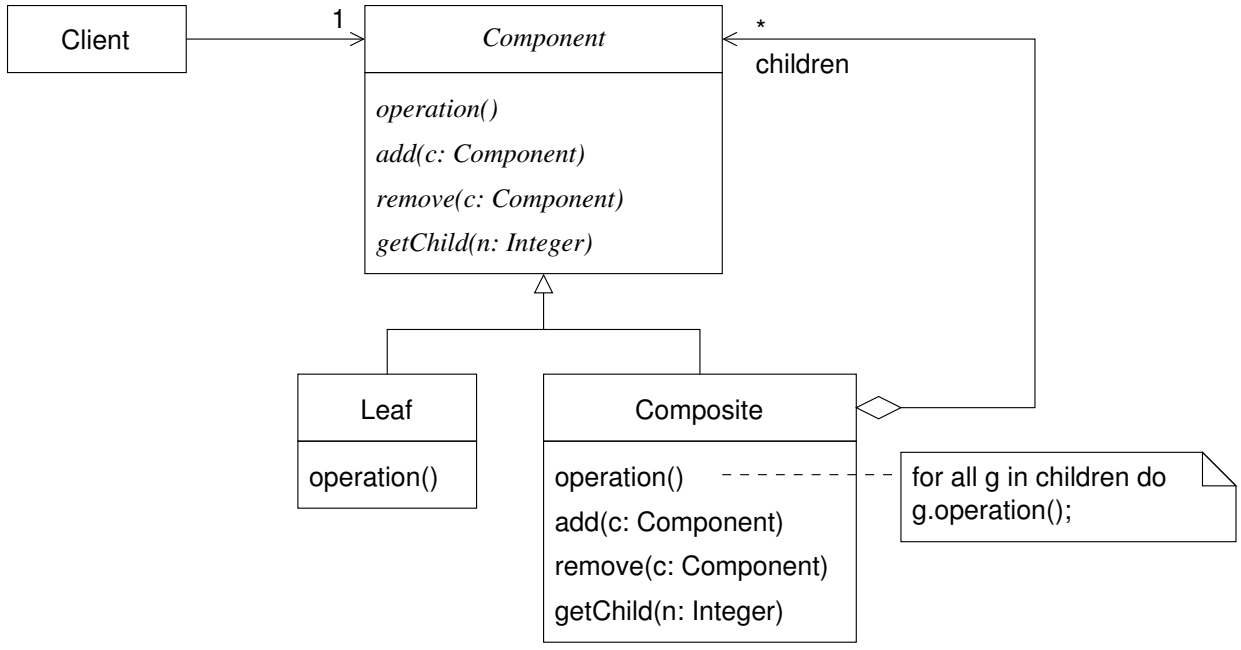
Anwendungsbeispiel: Toolkit in AWT

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



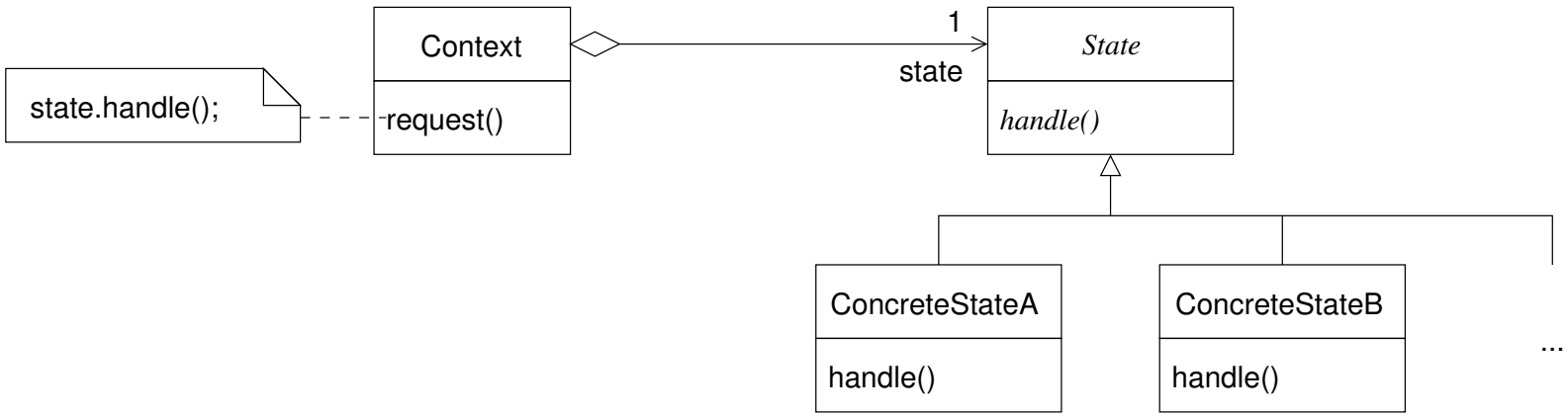
Anwendungsbeispiel: Komponenten in AWT/Swing; geometrische Figuren

Beispiel 3: **State (Behavioral Pattern)**

Zweck

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

Struktur



Anwendungsbeispiel:

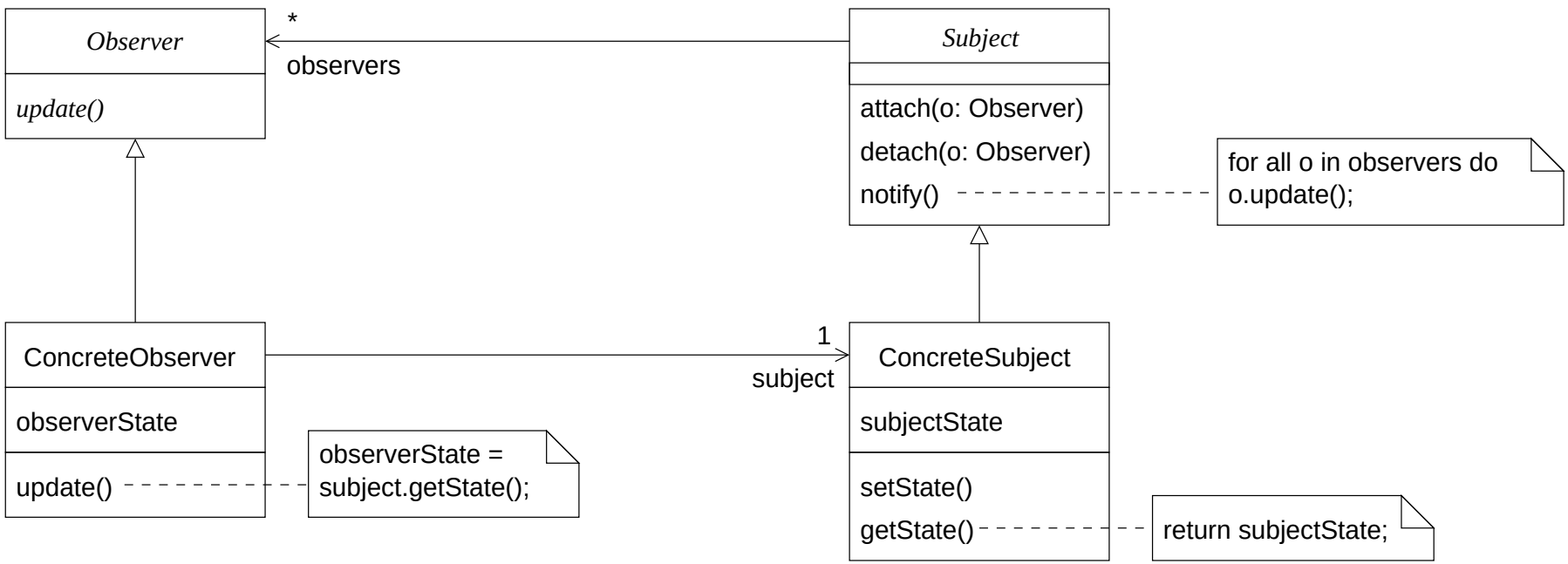
Realisierung von Zustandsdiagrammen durch Zustandsobjekte

Beispiel 4: **Observer (Behavioral Pattern)**

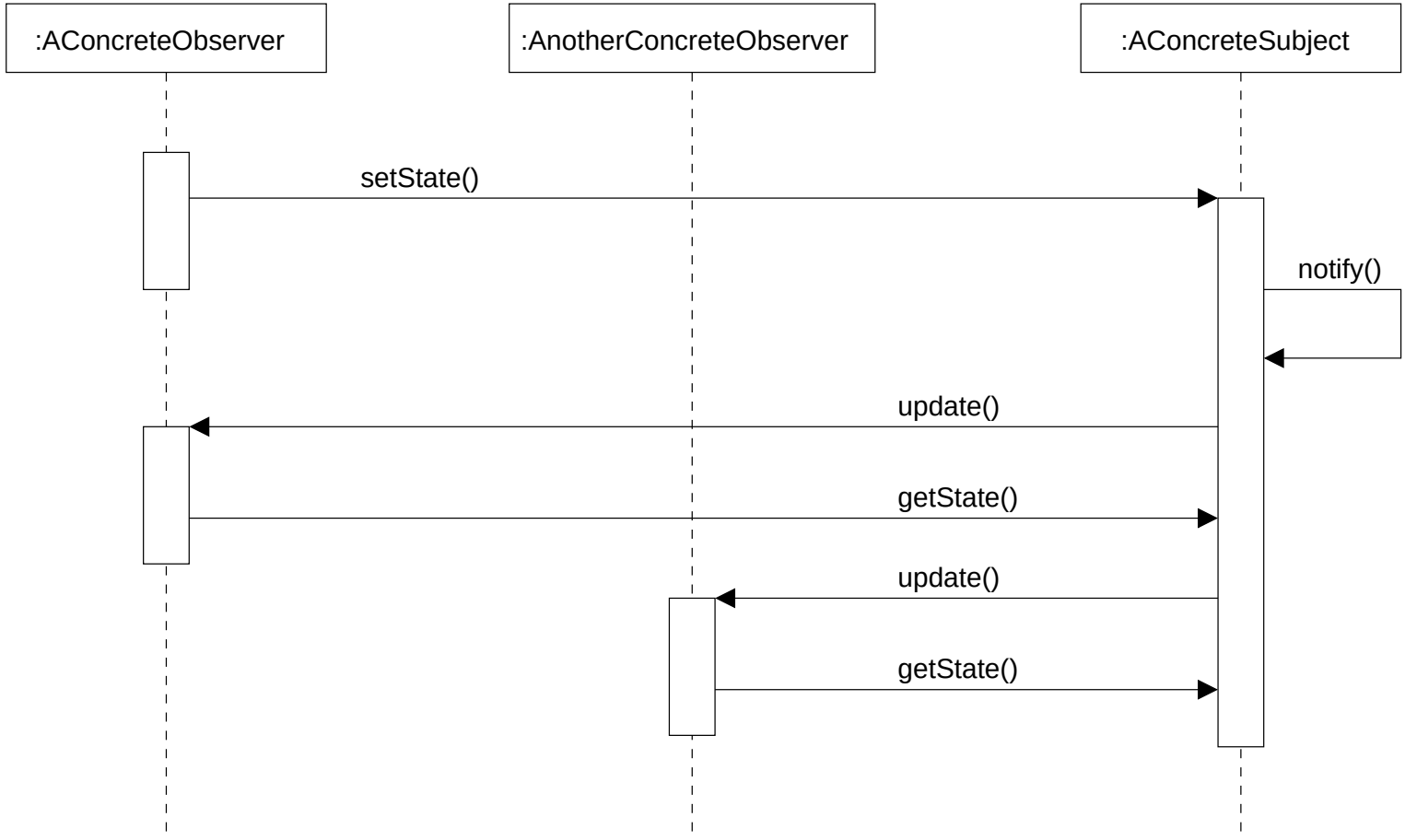
Zweck

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

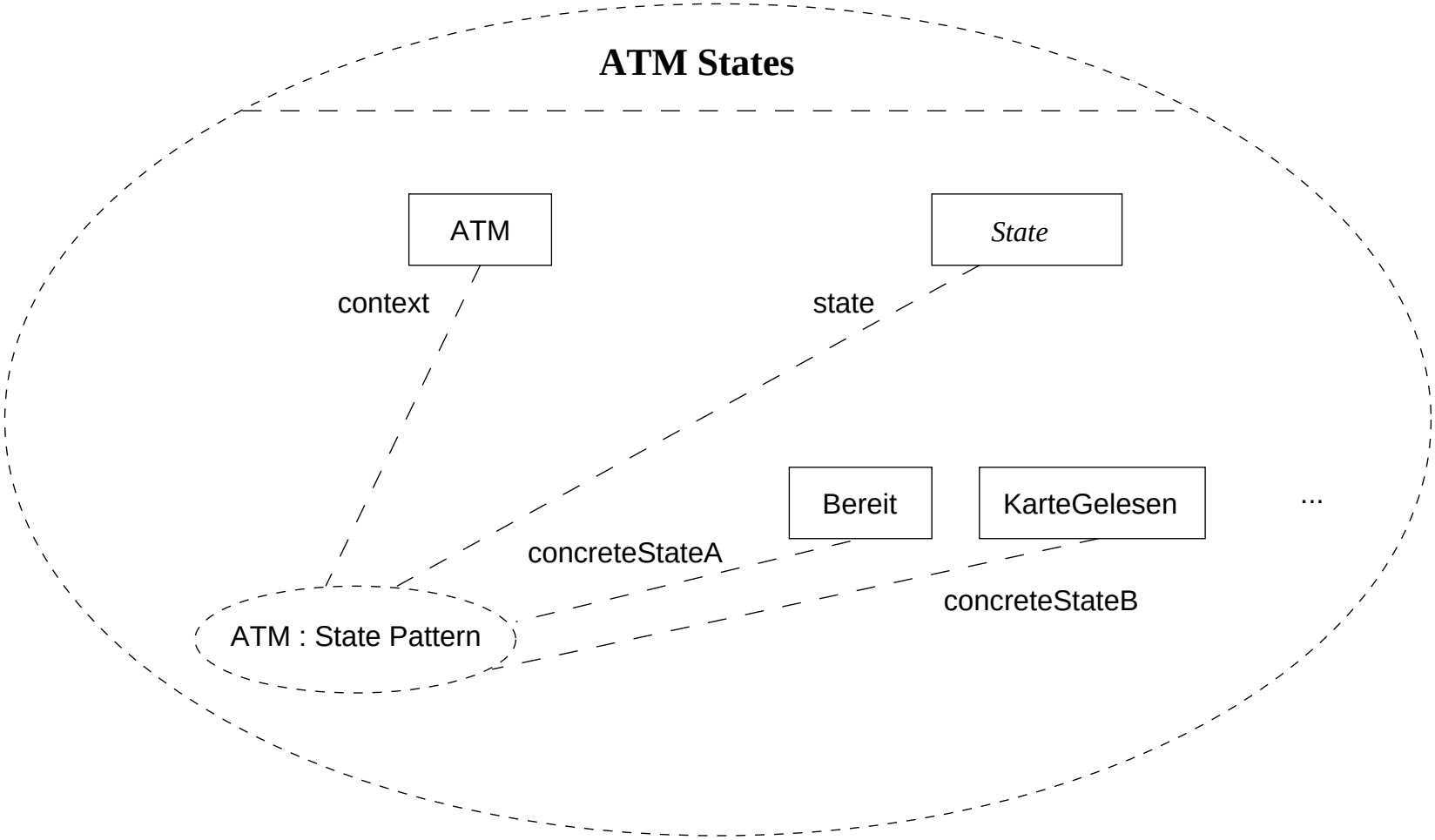


Anwendungsbeispiel:

Indirekte Kommunikation zwischen Anwendungskern und GUI

Anwendung von Mustern in UML

Beispiel: Anwendung des State-Pattern auf die ATM-Simulation



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.