

Einführung in die Informatik II
Übersetzung von UML-Modellen
in Java-Quelltexte

Dr. Andreas Harrer, Prof. Bernd Brügge Ph.D.
Institut für Informatik
Technische Universität München

Sommersemester 2001

9. Juli 2001

Einordnung der Vorlesung

- ❖ **9.-11 Juli:** Überleitung zu maschinennaher Programmierung
 - Übersetzung von UML in Java (heute, Montag)
 - Aufbau von Rechnern, Übersetzung von Java in eine maschinennahe Sprache (Mittwoch)
- ❖ **16.-18 Juli:** Maschinennahe Programmierung
 - Laufzeitkeller, Unterprogrammaufruf, Streuspeicherung, Halde
- ❖ **23.-25 Juli:** Entwurf eines größeren Systems, Vergleich der eingeführten Modellierungs- und Programmierkonzepte, Besprechung der Evaluierung

Überblick über Inhalt der Vorlesung

- ❖ Sprachebenen bei der System-Entwicklung
- ❖ manuelle und automatische Übersetzung
- ❖ Codegenerierung in Java aus UML-Diagrammen
 - Anwendungsfall-Diagramme
 - Sequenz-Diagramme
 - Klassen-Diagramme
 - Instanz-Diagramme
 - Zustands-Diagramme
- ❖ Probleme und Herausforderungen
 - Komplementarität und Konsistenz verschiedener Sichten
 - Werkzeuge
 - Verknüpfung von Modellierung und Programmierung

Ziele der Vorlesung

- ❖ Sie verstehen die Grundprinzipien der Übersetzung
- ❖ Sie beherrschen die Generierung von Java-Programmcode aus verschiedenen UML-Diagrammtypen
- ❖ Sie verstehen die Grenzen der Übersetzbarkeit von Modellen, wie z.B. das Problem der Konsistenz verschiedener Sichten
- ❖ Sie kennen Werkzeuge und Vorgehensweisen zur Verknüpfung von Modellierung und Programmierung

Sprachebenen bei der Software-Entwicklung

- ❖ Modellierungssprache (UML, OMT, OCL)
- ❖ Programmiersprache
 - Höhere Programmiersprache (Java, C++, Pascal,.....)
 - Maschinennahe Sprache (C, Forth, Assembler-Dialekte)
- ❖ Maschinensprache

Allgemeines zur Übersetzung

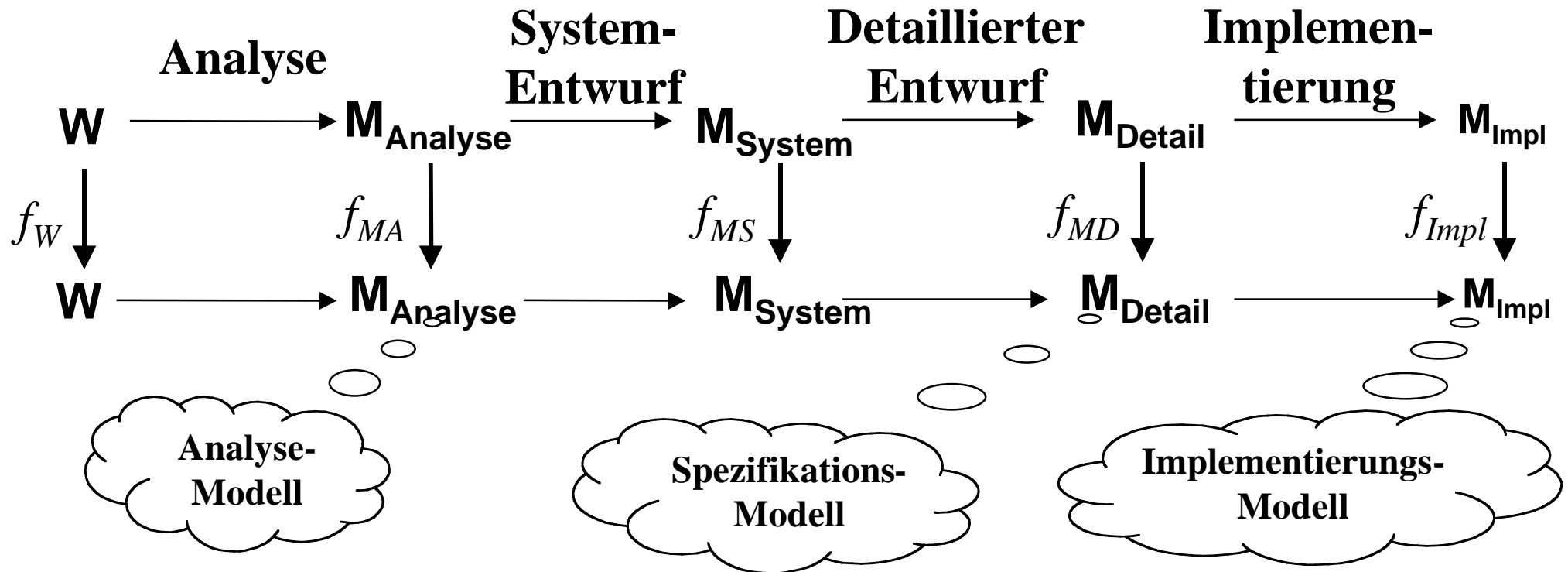
- ❖ Übersetzung findet statt von
 - Quellsprache zu
 - Zielsprache
- ❖ Zuordnung von Konstrukten der Quellsprache zu Konstrukten der Zielsprache
 - Analyse des Quellprogramms: Zerlegung in Quellsprach-Konstrukte
 - Synthese des Zielprogramms: Codegenerierung
- ❖ Durch die Zuordnung wird eine Semantik für die Übersetzung festgelegt
- ❖ Ein Übersetzer (Compiler) nimmt eine automatisierte Übersetzung vor
- ❖ Spezialvorlesung im Hauptstudium: Übersetzerbau
- ❖ Man muss allerdings auch wissen, wie Übersetzen manuell geht, besonders dann, wenn die Automatisierung (noch) nicht gut ist

Grundlage unseres Modells: UML-Diagramme

- ❖ Wir haben in Info I und Info II folgende Diagramme zur Beschreibung wesentlicher Aspekte von Systemen kennengelernt:
 - funktionelle Aspekte: Anwendungsfall-Diagramme
 - Aspekte der Interaktion: Sequenz-Diagramme
 - strukturelle (statische) Aspekte: Klassen-Diagramme
 - strukturelle (momentane) Aspekte: Instanz-Diagramme
 - dynamische Aspekte: Zustands-Diagramme
- ❖ Im Folgenden beleuchten wir, wie diese Diagramme in Programmcode übersetzt werden können
- ❖ Außerdem untersuchen wir, wie sich verschiedene Diagramme ergänzen, ob sie widerspruchsfrei sind und ob eine eindeutige Übersetzung möglich ist.

Rückblick: Entwicklungsaktivitäten und Modelle

(siehe Info I - Vorlesung 2: Informatik-Systeme)

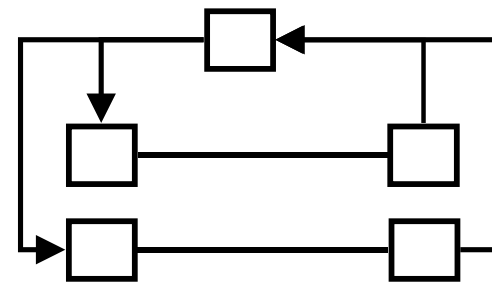
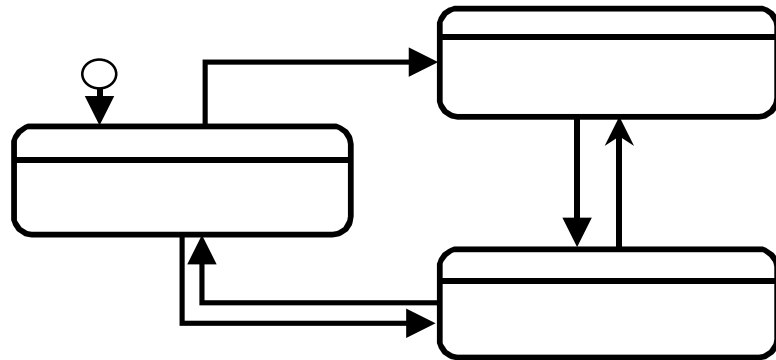
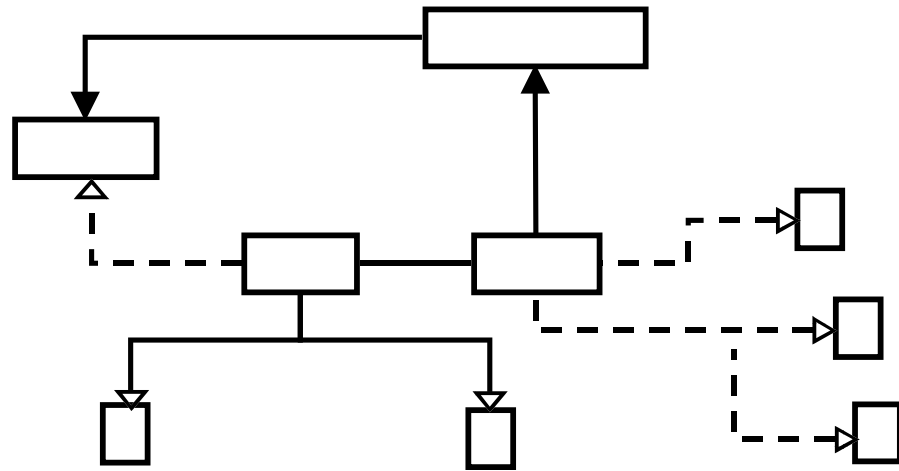
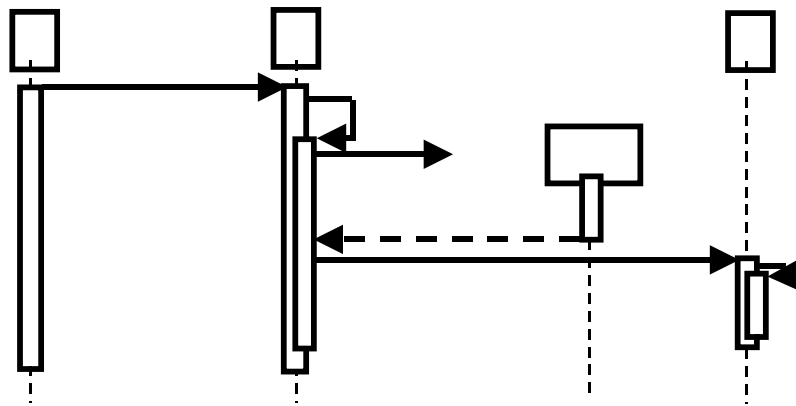
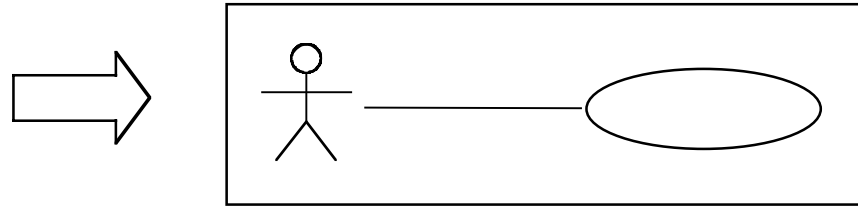


Wir betrachten in dieser Vorlesung Implementierungsmodelle, die sich aus mehreren UML-Diagramm(typ)en zusammensetzen

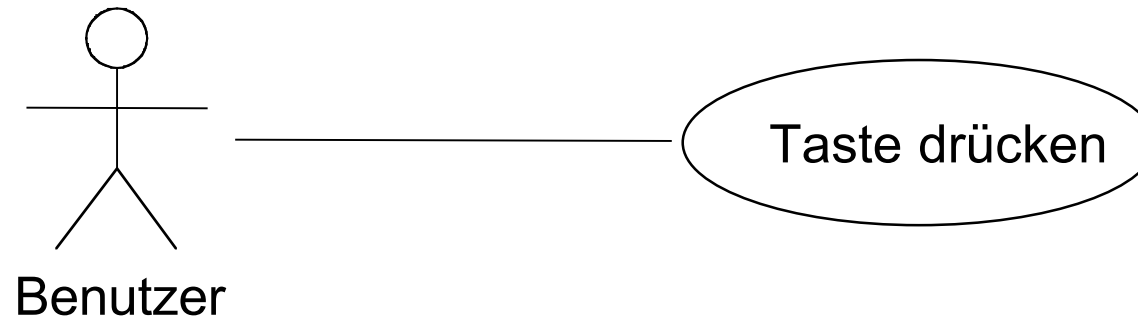
Fallbeispiel zur Übersetzung UML nach Java

- ❖ Wir betrachten nun die Übersetzung von UML-Diagrammen in Java-Programmcode an einem Fallbeispiel
- ❖ Gegeben ist ein UML-Implementierungs-Modell für ein Software-System aufgebaut gemäß der Architektur
 - Modell (*model*)
 - Sicht (*view*)
 - Steuerung (*controller*)
- ❖ Diverse Aspekte des Systems sind durch die uns bekannten UML-Diagramm-Typen abgedeckt

Überblick über unser Implementierungs-Modell (Vogelperspektive)

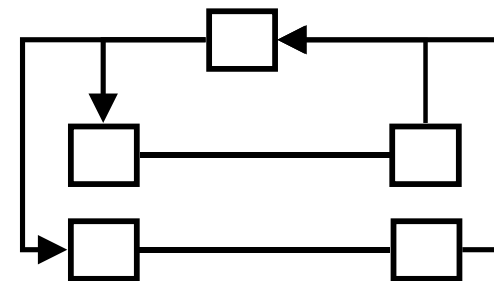
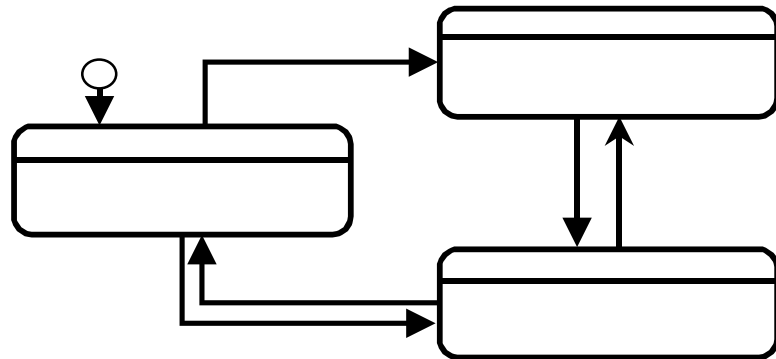
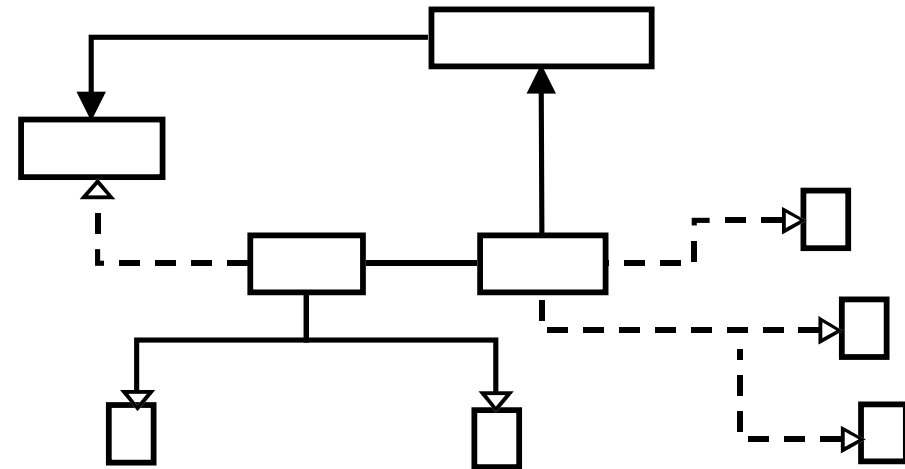
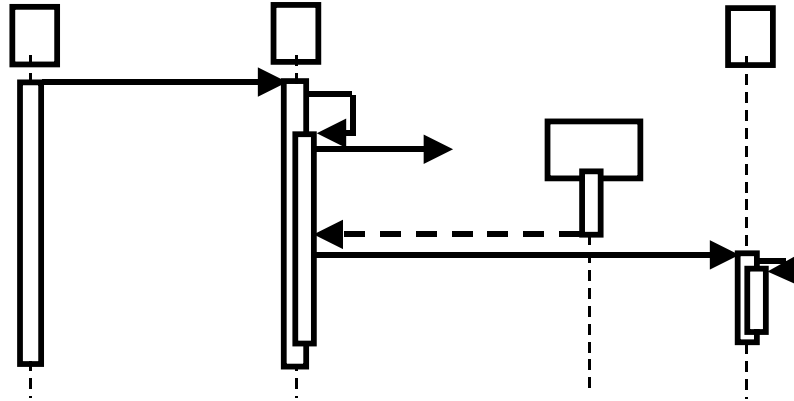
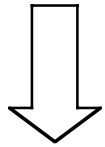
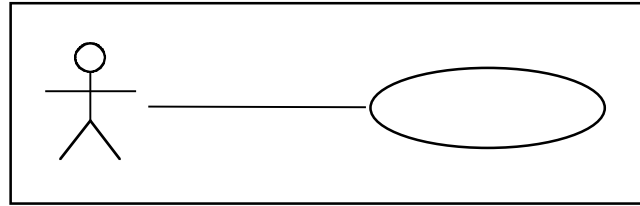


Was machen wir mit Anwendungsfall-Diagrammen?

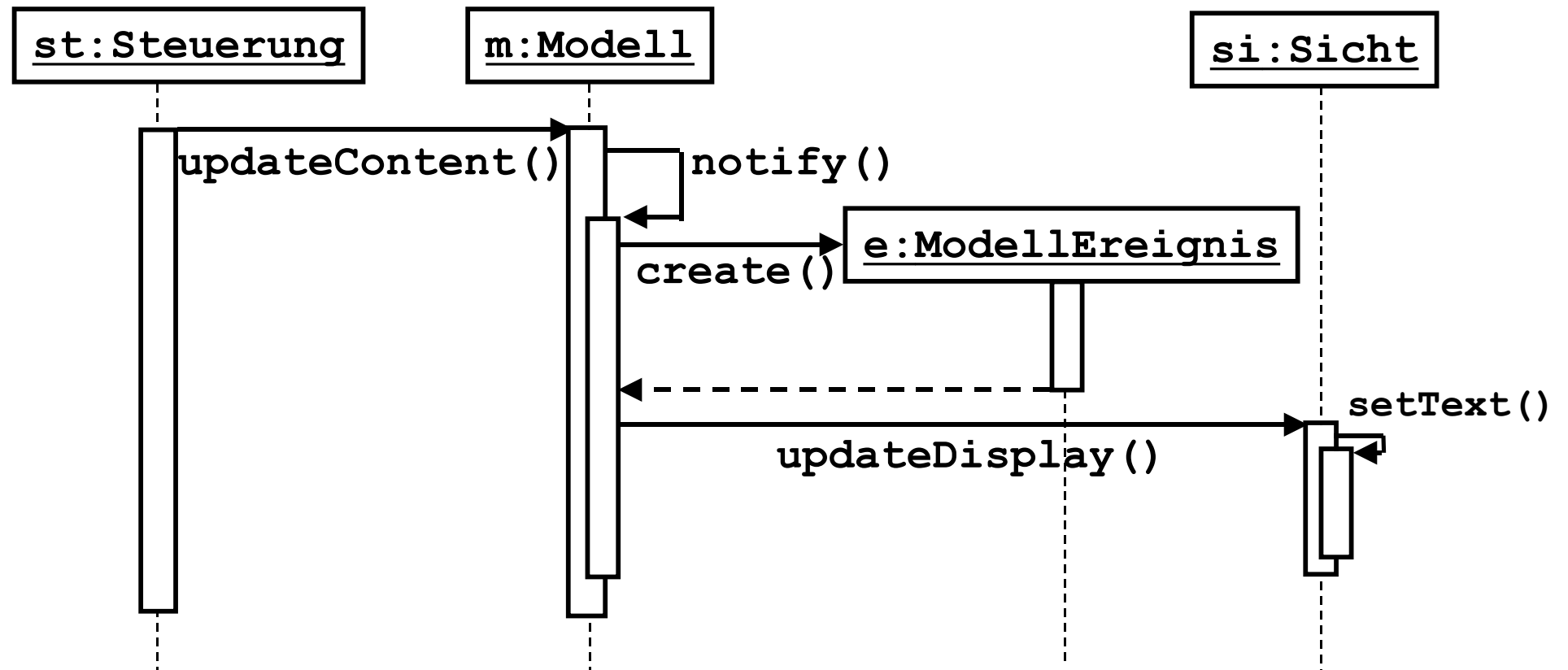


- ❖ Anwendungsfall-Diagramme stellen ein Hilfsmittel der Analyse dar, nicht des Entwurfs
- ❖ daher dienen sie eher als **Vorstufe** für andere Diagramme (insbesondere Sequenz-Diagramme)
- ❖ eine direkte **Übersetzung** in Programmcode ist **wenig ergiebig**
- ❖ bestenfalls Namen von Diensten bzw. Operationen sind davon ableitbar

Überblick über unser Implementierungs-Modell (Vogelperspektive)



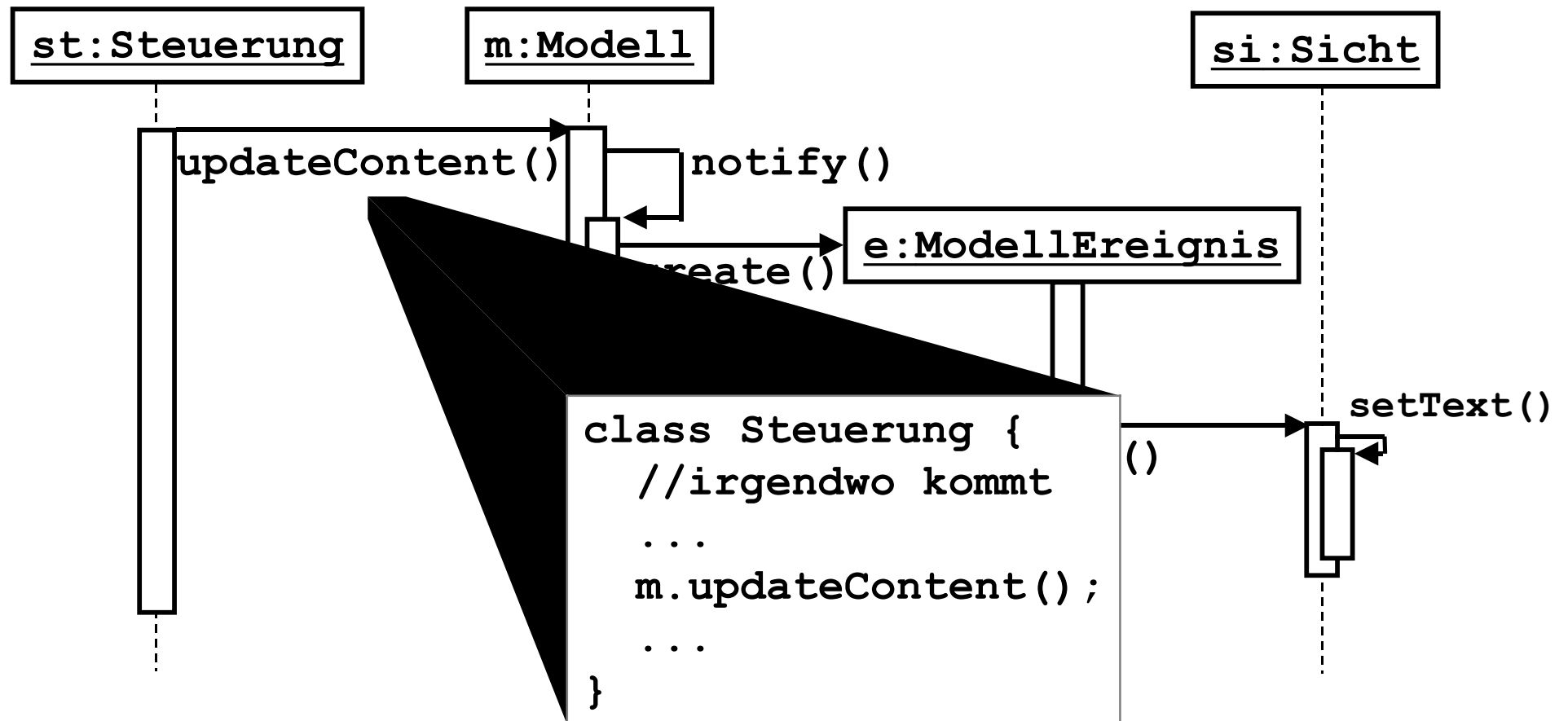
Sequenz-Diagramm unseres Implementierungs-Modells



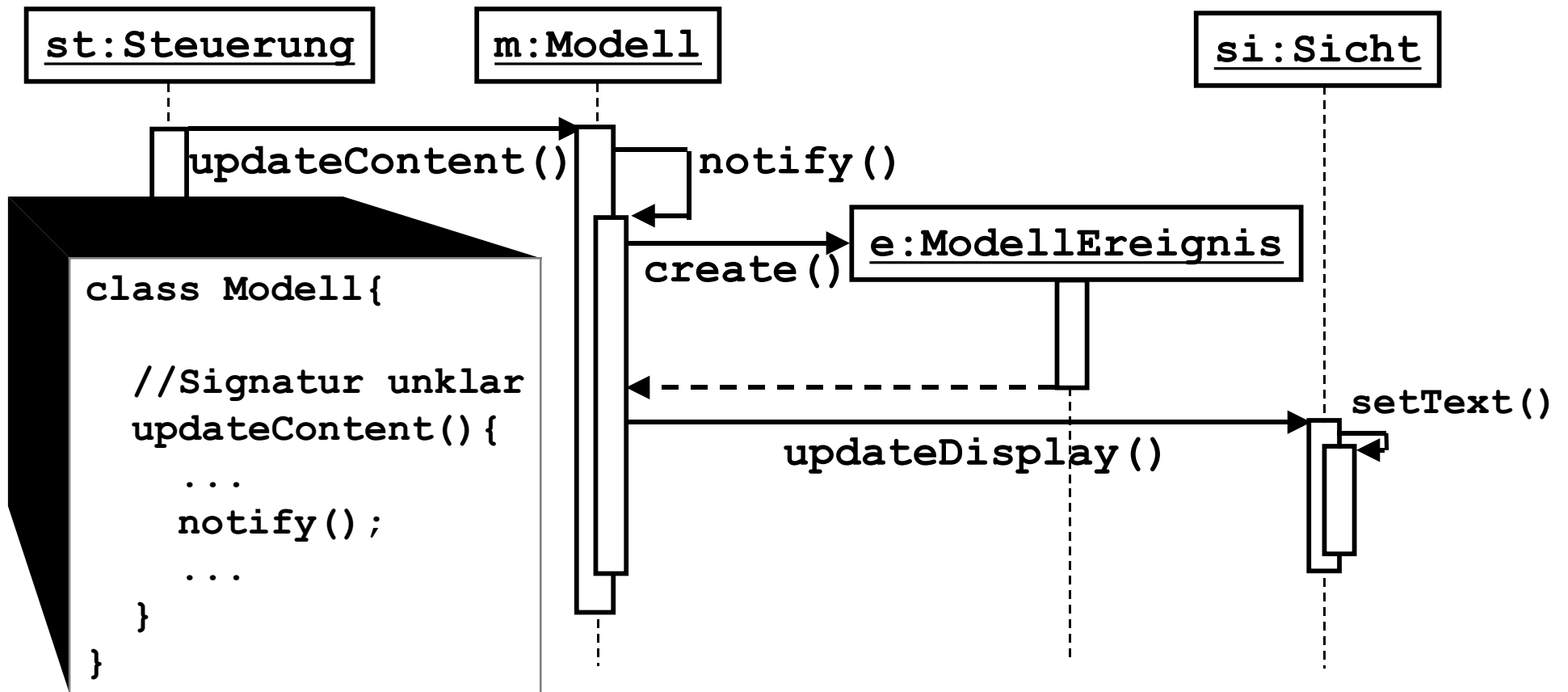
Übersetzung von Sequenz-Diagrammen

- ❖ Sequenz-Diagramme spiegeln Interaktionen zwischen Objekten wider
- ❖ Sequenz-Diagramme stellen Beispielszenarien dar
- ❖ Nachrichten werden in Java in Form von Methoden bzw. Methodenaufrufen implementiert
- ❖ Für die Übersetzung in Programmcode sind folgende Punkte zu überlegen:
 - Wo wird der Code platziert? (in welchen Methoden?)
 - Auf welchen Objektinstanzen wird ein Methodenaufruf durchgeführt?
 - Unter welchen Bedingungen findet dieses Szenarium statt?

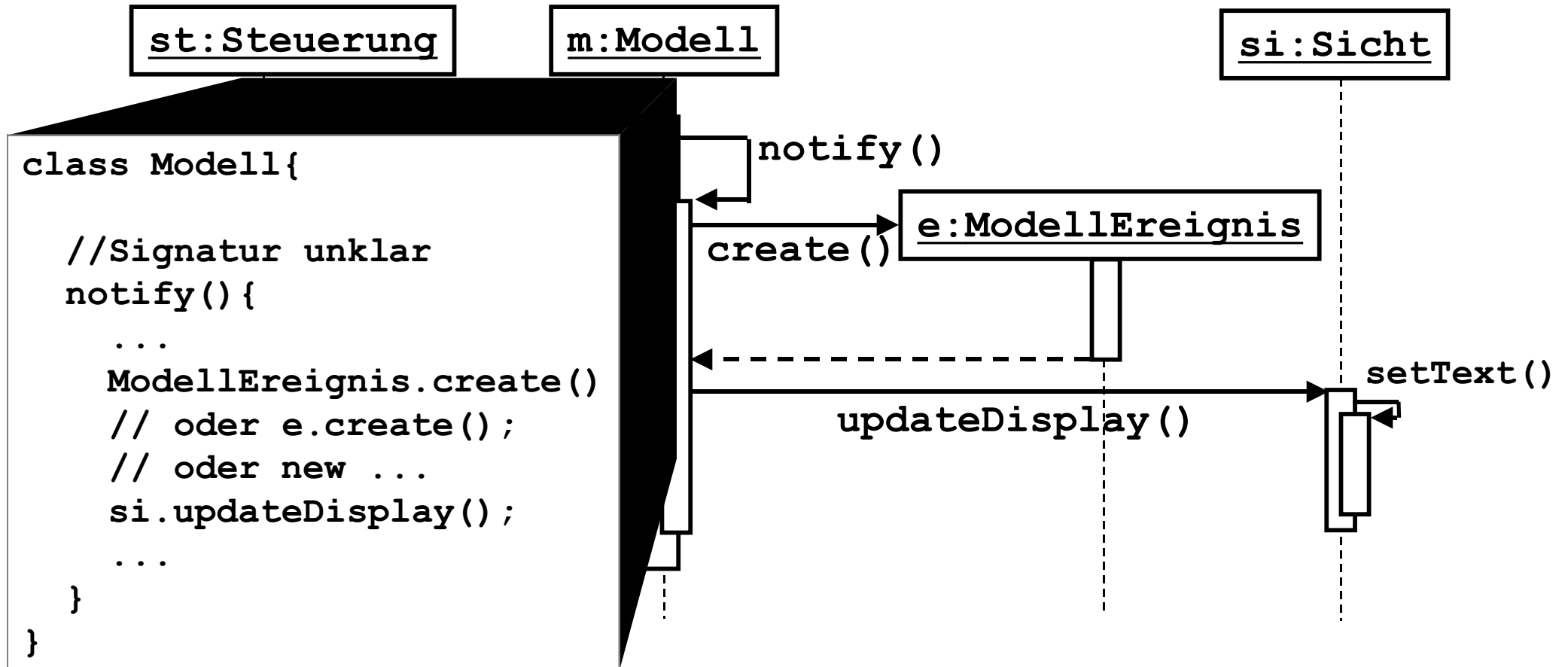
Übersetzung von Sequenz-Diagrammen (Fallbeispiel)



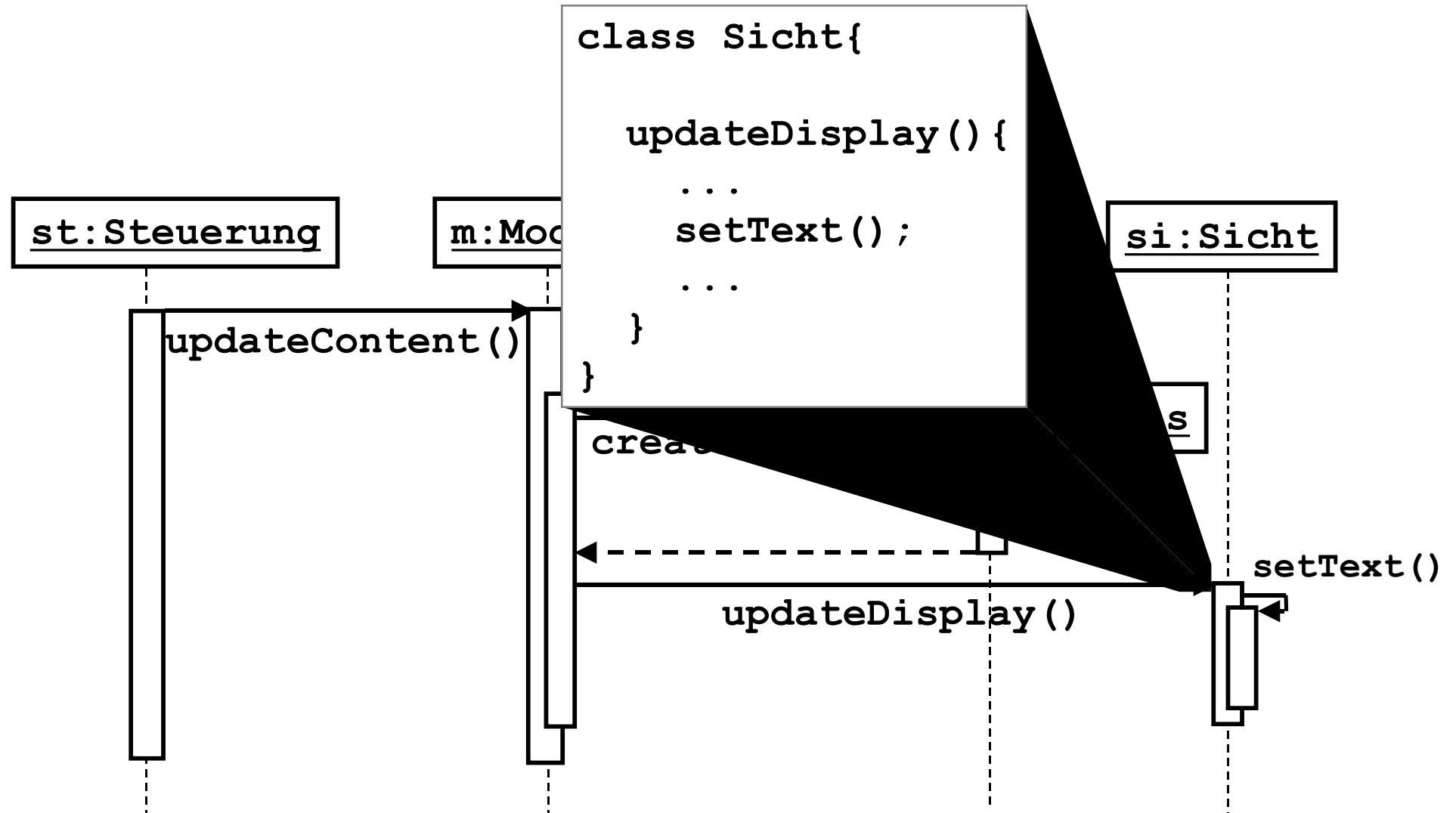
Übersetzung von Sequenz-Diagrammen (Fallbeispiel)



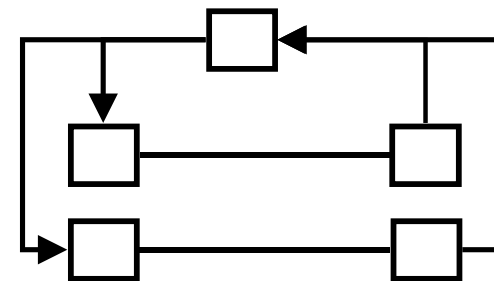
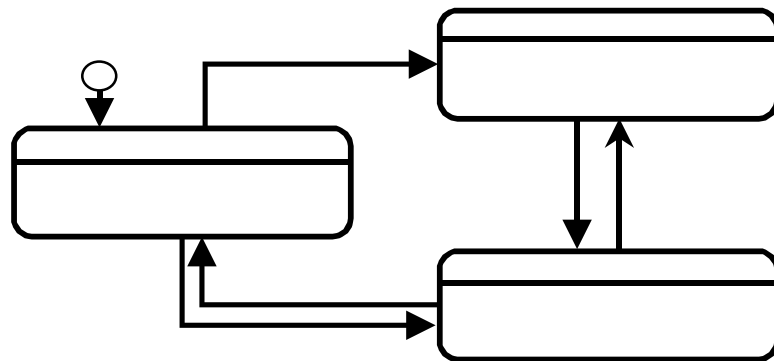
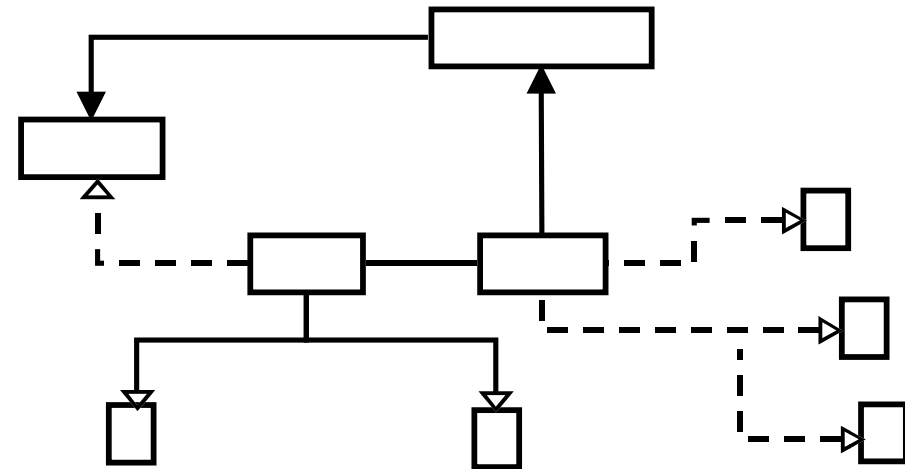
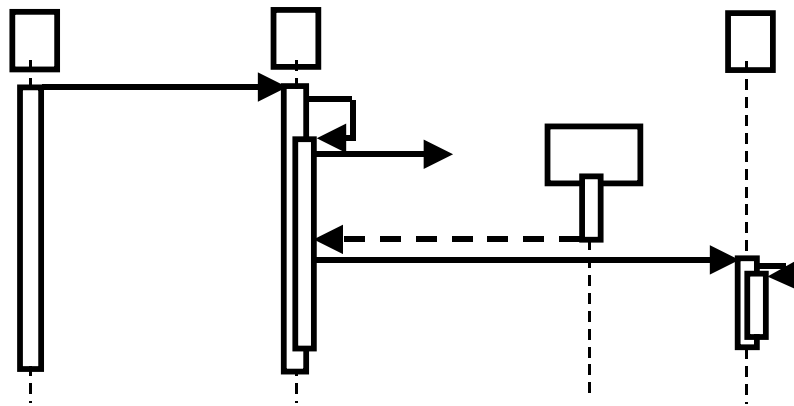
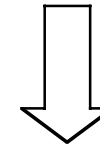
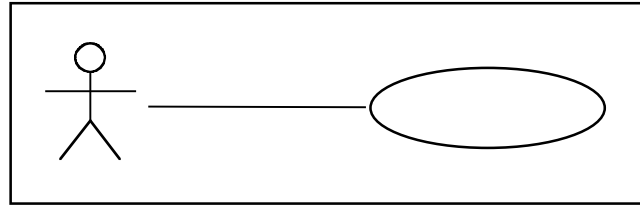
Übersetzung von Sequenz-Diagrammen (Fallbeispiel)



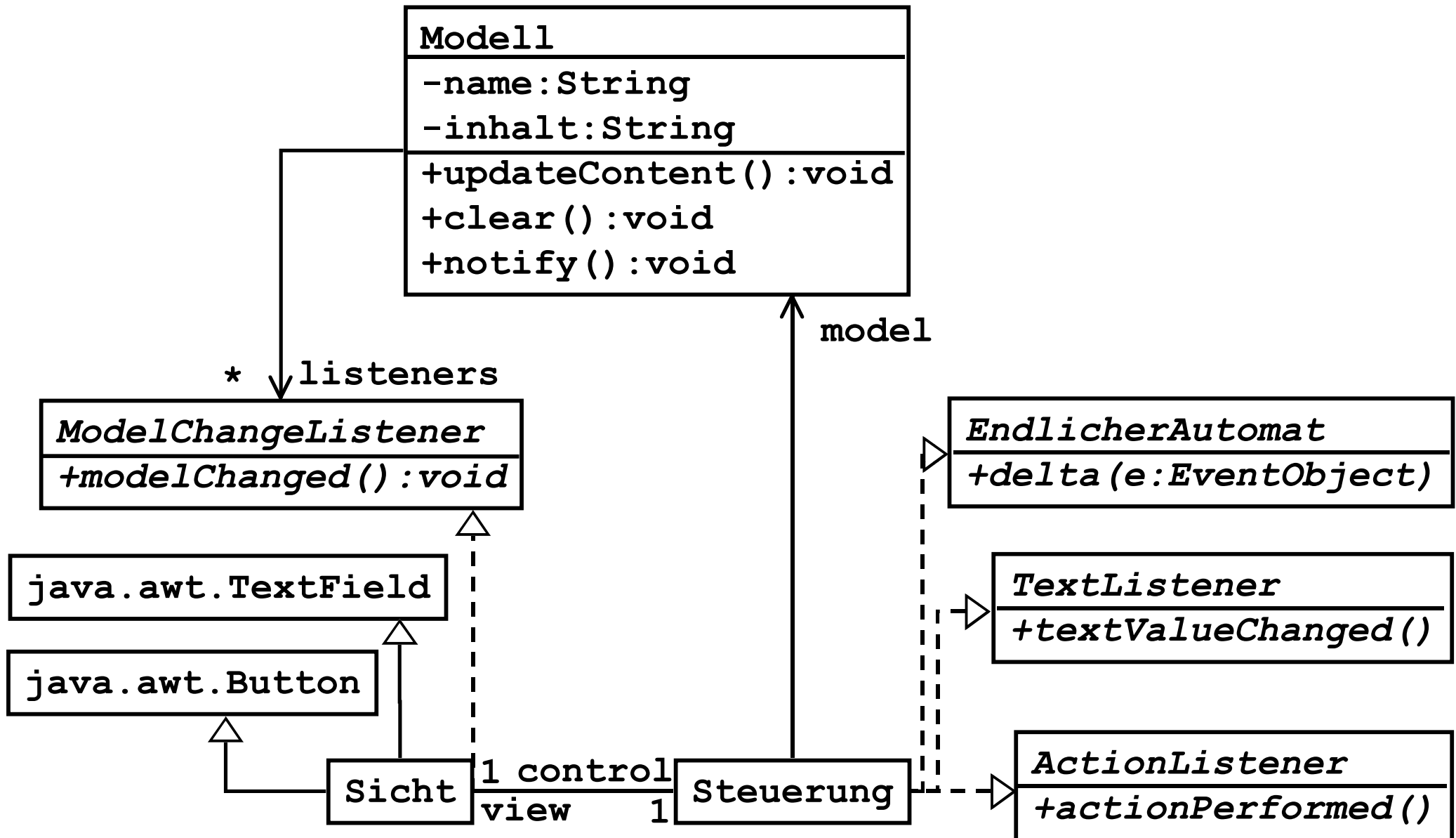
Übersetzung von Sequenz-Diagrammen (Fallbeispiel)



Überblick über unser Implementierungs-Modell (Vogelperspektive)



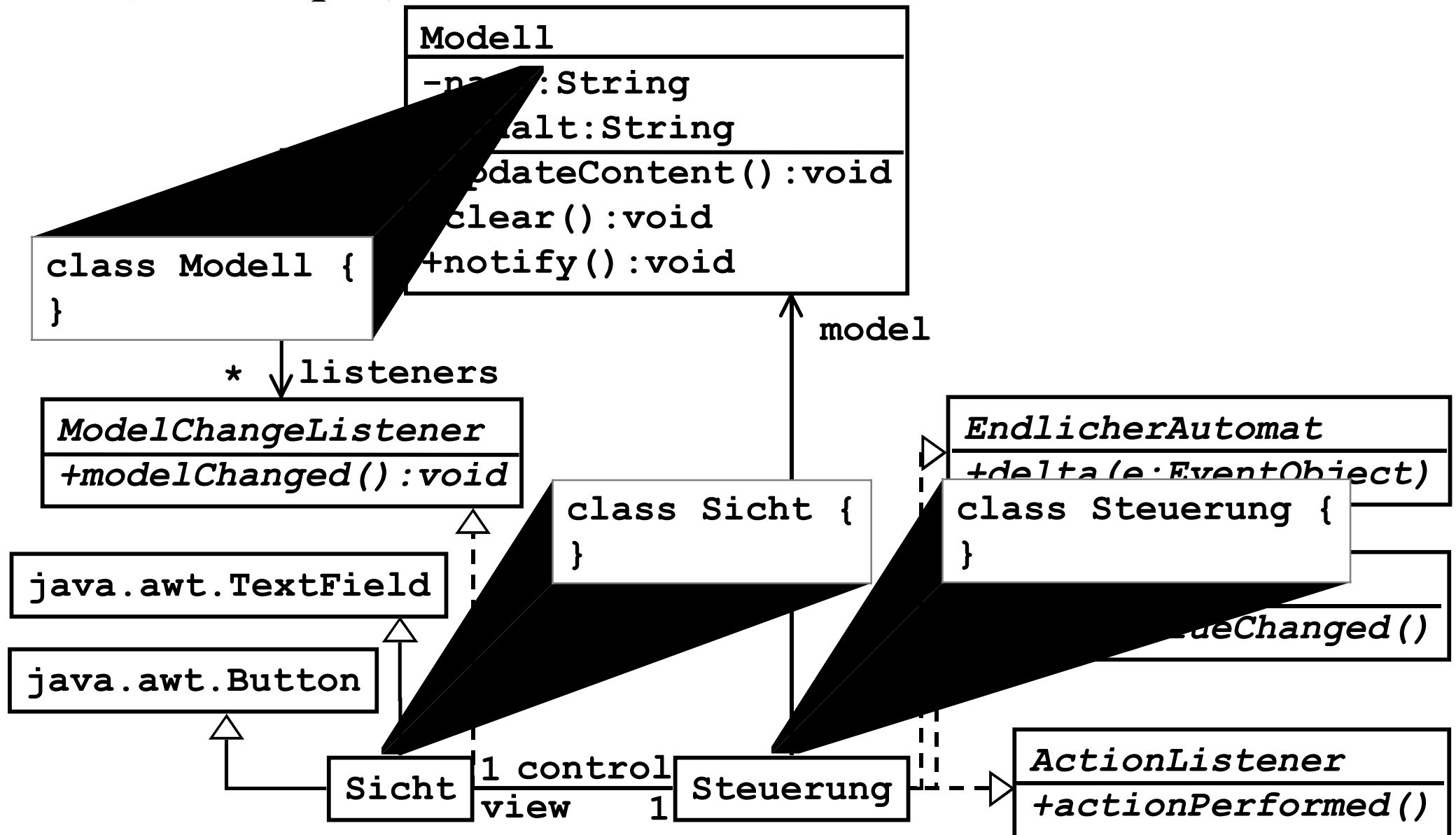
Klassen-Diagramm unseres Implementierungs-Modells



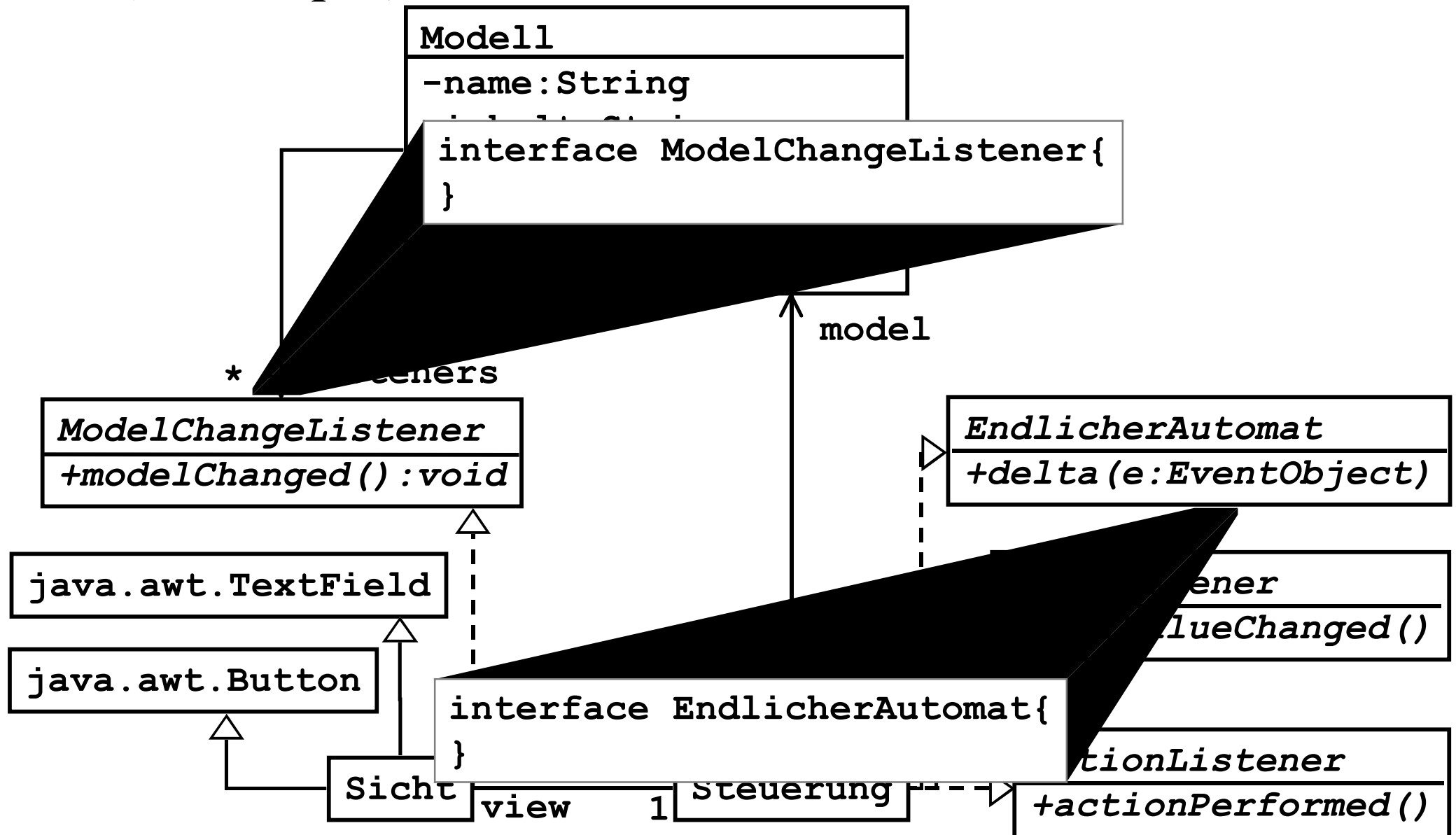
Übersetzung von Klassen-Diagrammen

- ❖ Klassendiagramme beschreiben die allgemeine (statische) Struktur von Klassen und deren Beziehungen zueinander
- ❖ Klassendiagramme beinhalten Informationen über:
 - Klassen und Schnittstellen
 - Attribute
 - Methoden
 - Assoziationen
 - Vererbung
 - Aggregation
- ❖ Die Übersetzung in Java-Programmcode ergibt sich meist unmittelbar

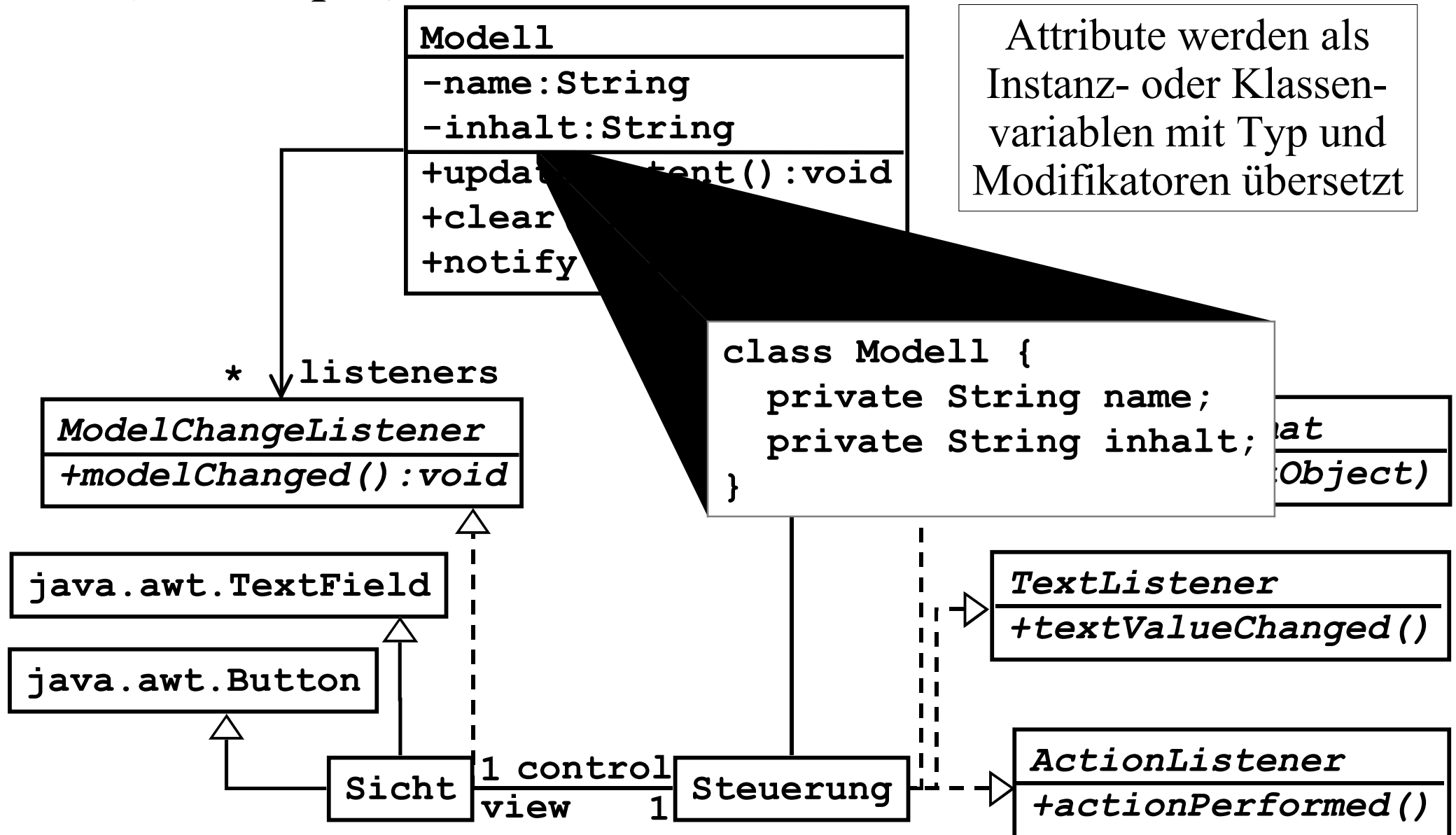
Übersetzung von Klassen im Klassen-Diagramm (Fallbeispiel)



Übersetzung von Schnittstellen im Klassen-Diagramm (Fallbeispiel)



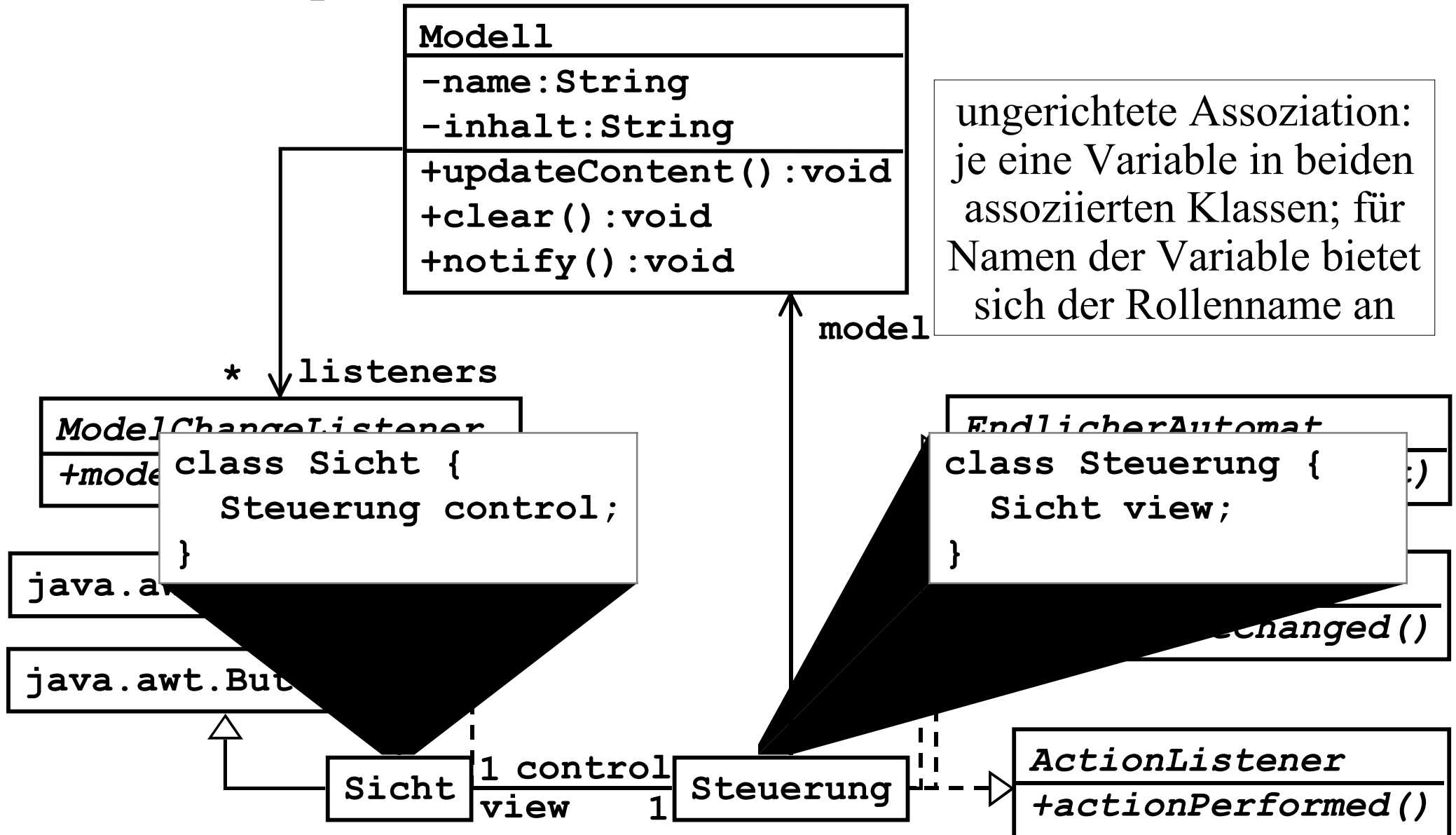
Übersetzung von Attributen im Klassen-Diagramm (Fallbeispiel)



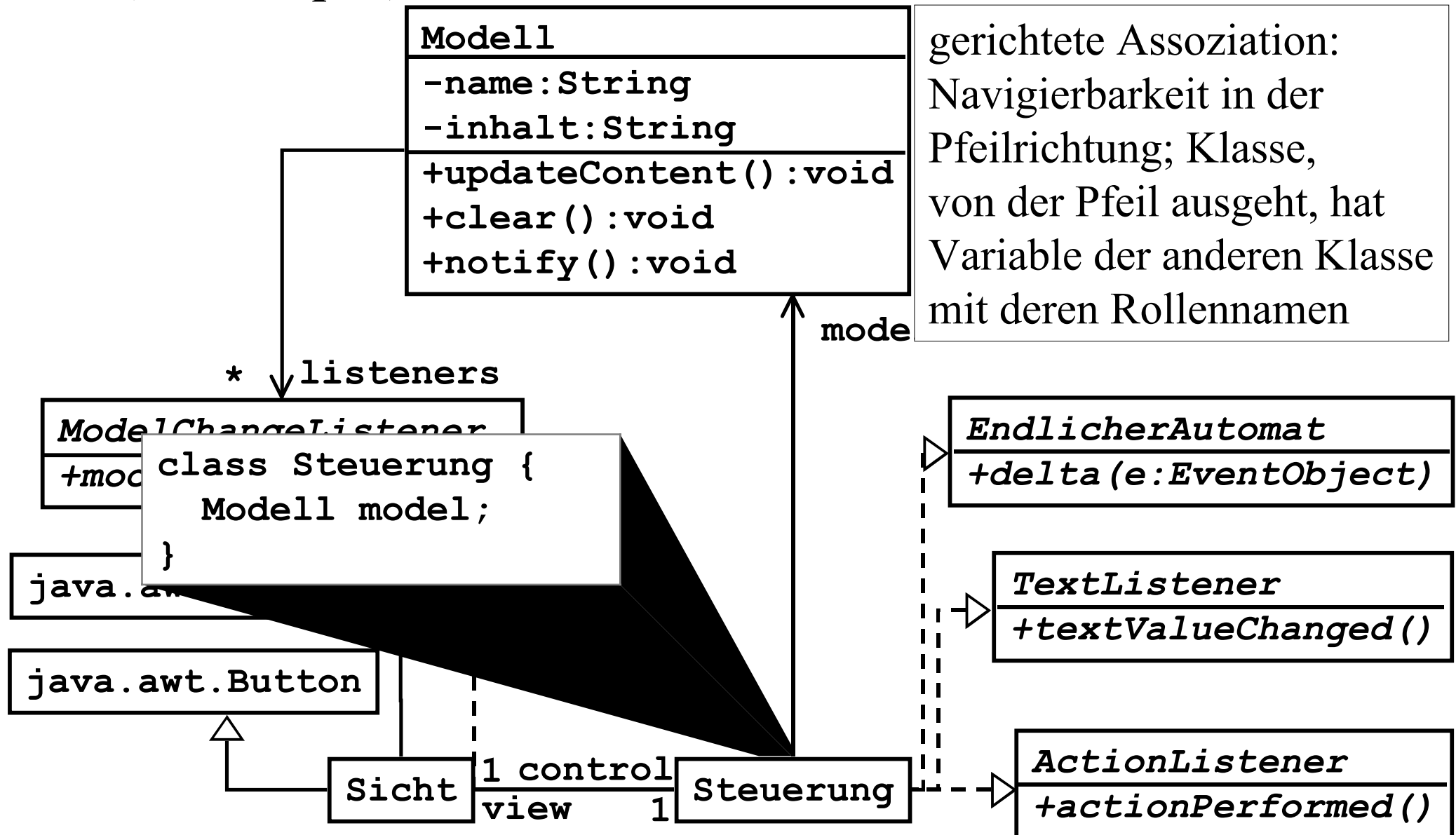
Übersetzung von Assoziationen im Klassen-Diagramm

- ❖ Assoziationen werden analog zu Attributen als Instanz- oder Klassenvariablen übersetzt
- ❖ Allerdings stellen Assoziationen eine Navigationsmöglichkeit zwischen den assoziierten Objekten dar
 - ungerichtete Assoziationen werden durch je eine Variable in beiden assoziierten Klassen realisiert; für den Namen der Variable bietet sich der Rollename an
 - gerichtete Assoziationen stellen eine Navigierbarkeit in der Pfeilrichtung an: die Klasse, von der der Pfeil ausgeht, hat eine Variable der anderen Klasse mit deren Rollennamen

Übersetzung von Assoziationen im Klassen-Diagramm (Fallbeispiel)



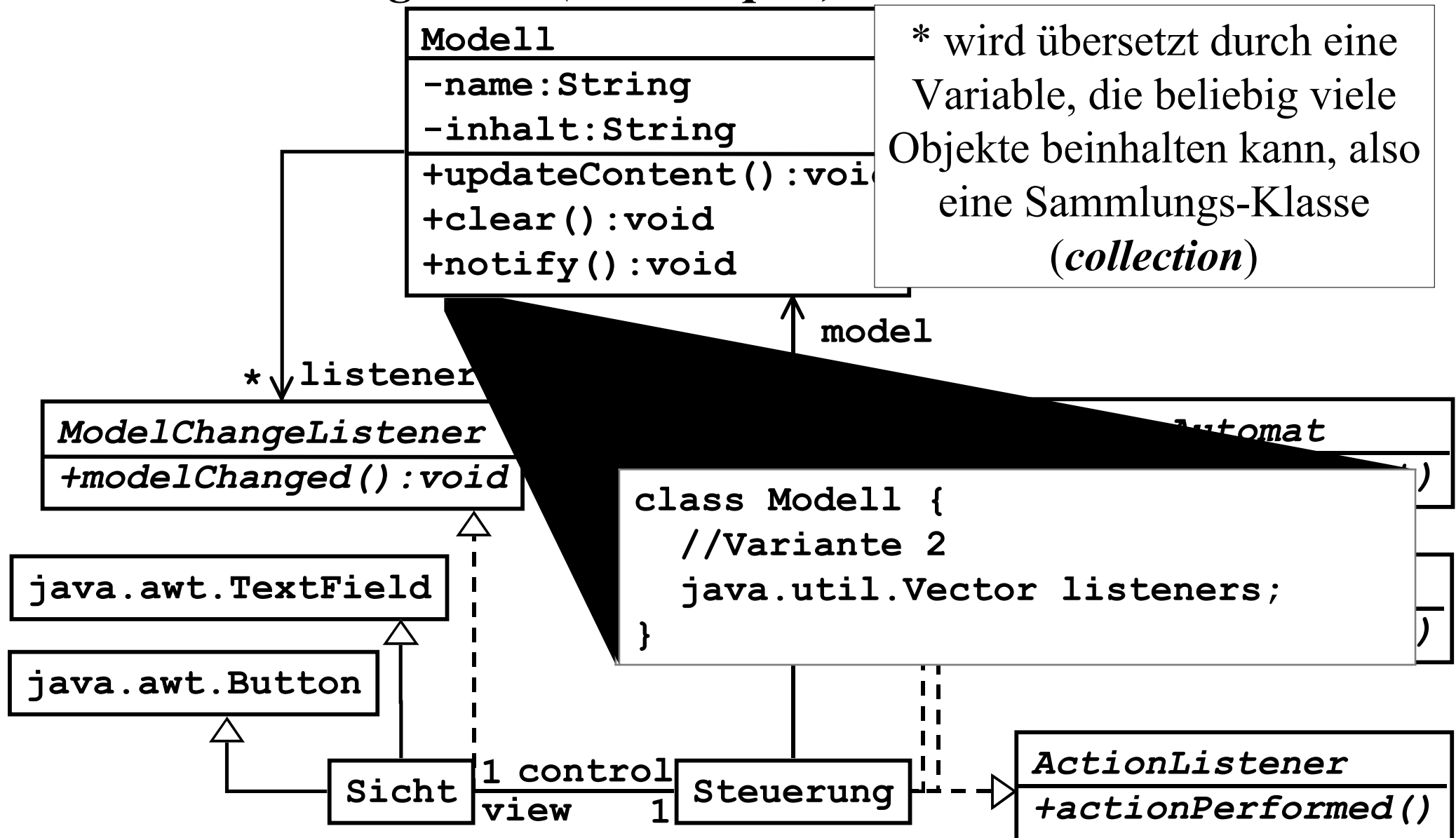
Übersetzung von Assoziationen im Klassen-Diagramm (Fallbeispiel)



Übersetzung von Multiplizitäten an Assoziationsenden im Klassen-Diagramm

- ❖ Übersetzung der Vielfachheit (Multiplizität) eines Assoziationsendes:
 - 1 wird übersetzt durch genau eine Variable, die auch instantiiert sein muss (im Konstruktor?)
 - 0..1 wird übersetzt durch eine Variable, die gegebenenfalls auch nicht instantiiert (**null**) sein darf
 - * wird übersetzt durch eine Variable, die beliebig viele Objekte beinhalten kann, also ein Objekt vom Typ einer Sammlungs-Klasse (*collection*)
 - Reihung
 - Sequenz
 - Baum
 - **java.util.Vector**

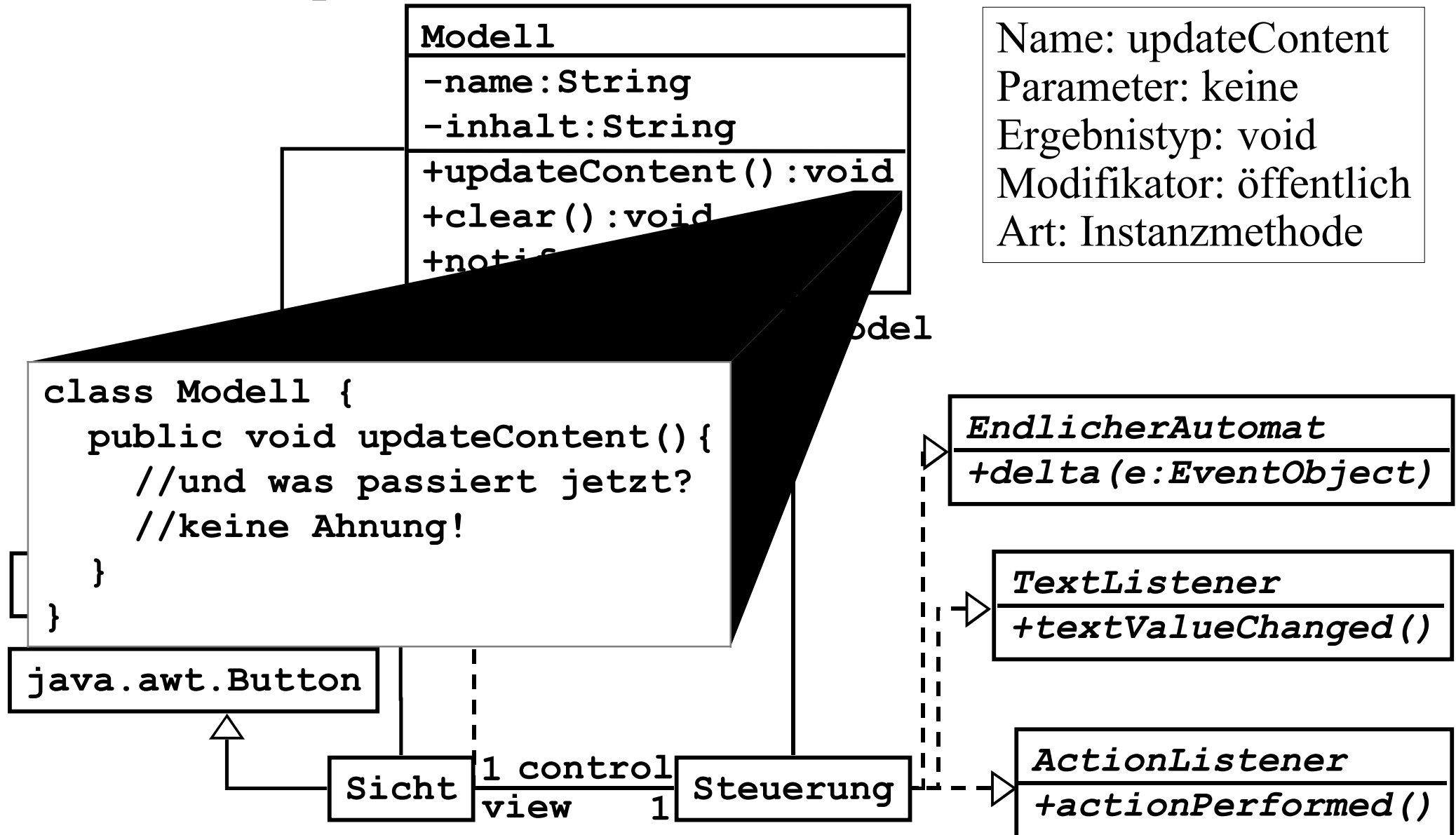
Übersetzung von Multiplizitäten an Assoziationsenden im Klassen-Diagramm (Fallbeispiel)



Übersetzung von Methoden im Klassen-Diagramm

- ❖ Aus einer Beschreibung einer Methode im UML-Klassendiagramm (Implementierungsmodell) kann man folgende Informationen zur Code-Generierung entnehmen:
 - Name der Methode
 - Parameter
 - Ergebnistyp, falls vorhanden
 - Modifikatoren (private, protected, public, abstract)
 - Art der Methode: Instanz-Methode oder Klassen-Methode
- ❖ Damit kann also der gesamte **Methodenkopf** erstellt werden
- ❖ Über den Rumpf kann allerdings daraus nichts geschlossen werden, dazu können Informationen aus anderen Diagrammen dienen

Übersetzung von Methoden im Klassen-Diagramm (Fallbeispiel)



Name: updateContent
 Parameter: keine
 Ergebnistyp: void
 Modifikator: öffentlich
 Art: Instanzmethode

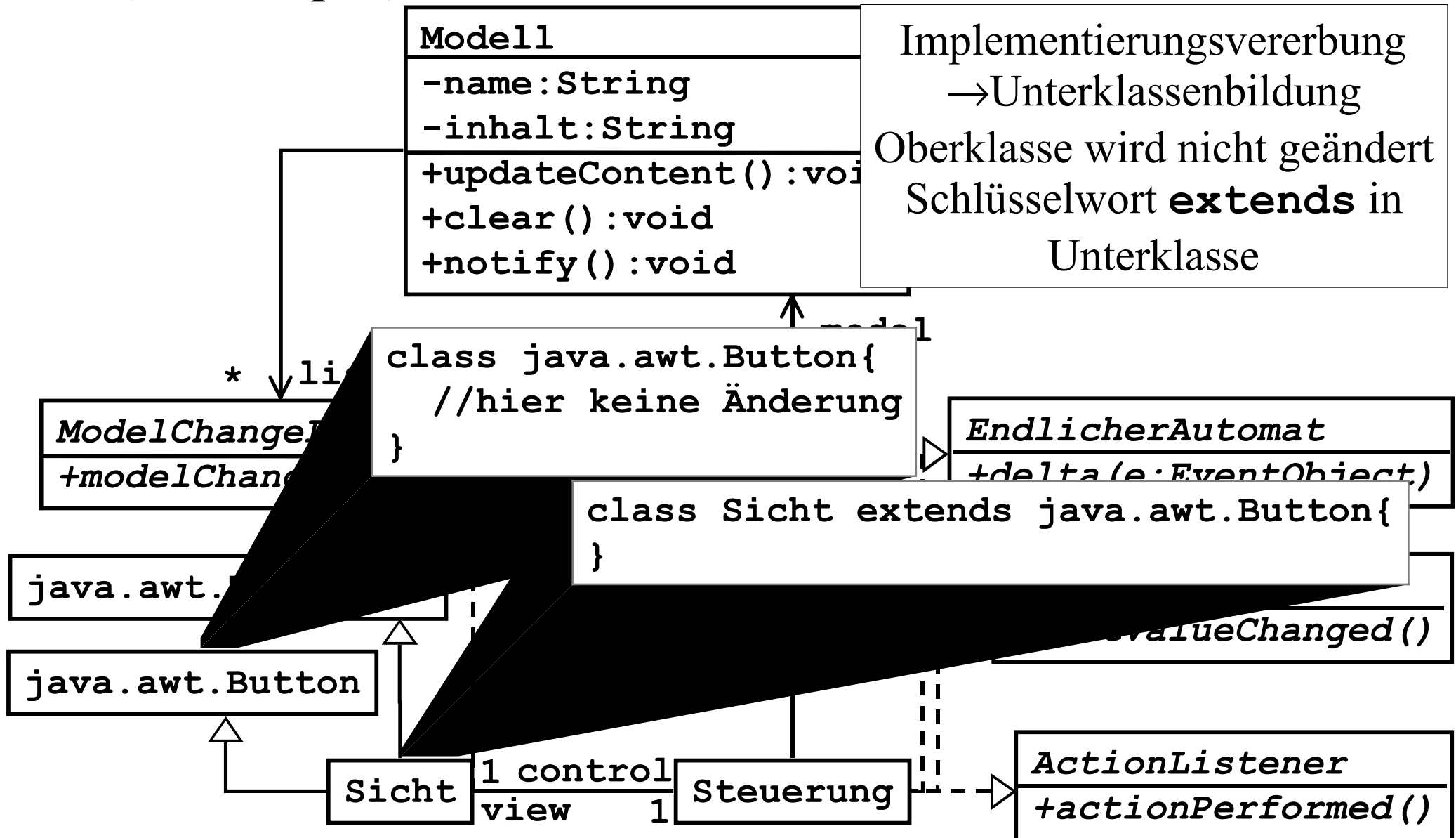
```

class Modell {
  public void updateContent() {
    //und was passiert jetzt?
    //keine Ahnung!
  }
}
  
```

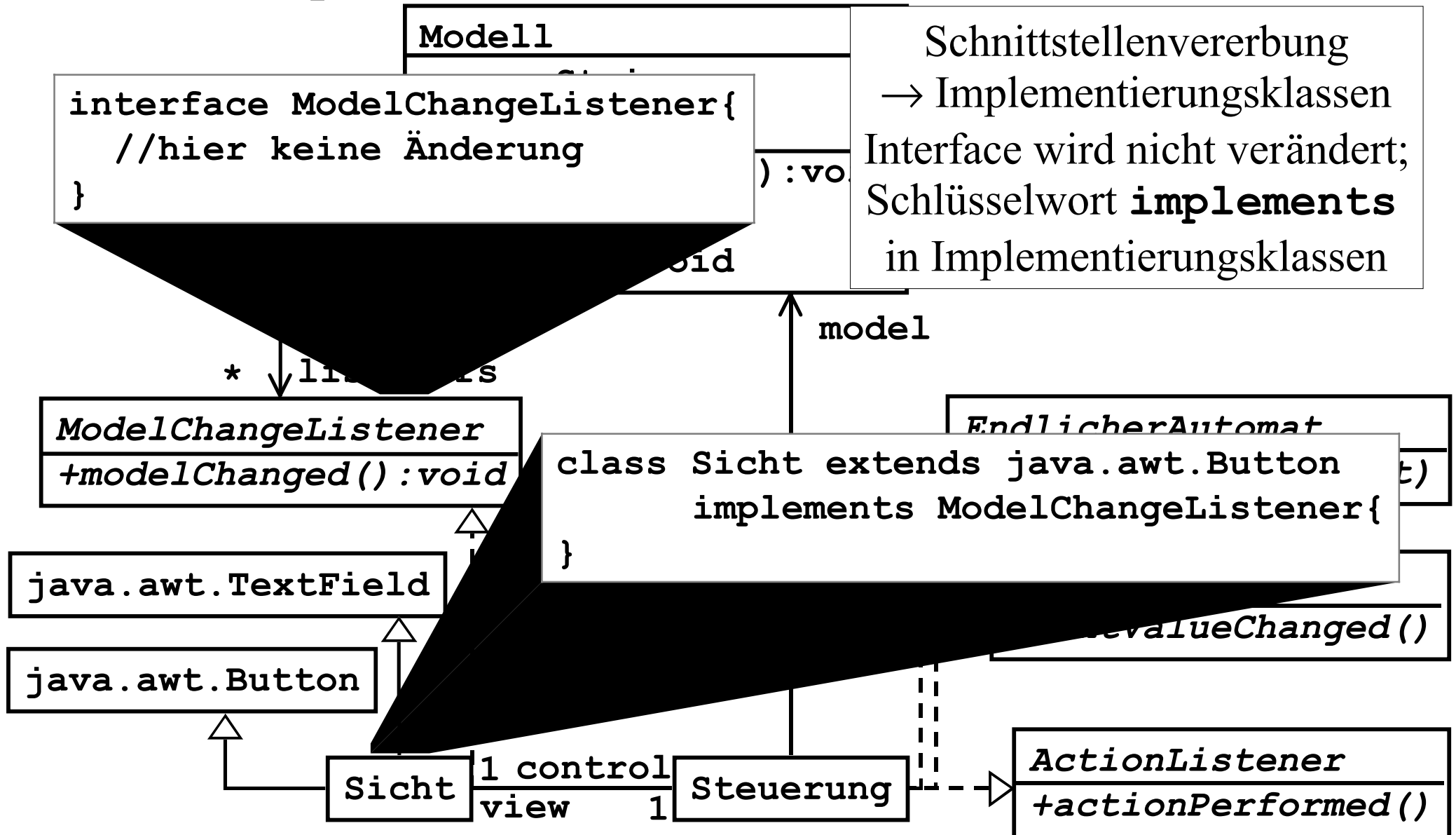
Übersetzung von Vererbung im Klassen-Diagramm

- ❖ Beide Arten von Vererbung in UML haben ein Gegenstück in Java:
 - Implementierungsvererbung → Unterklassenbildung
 - Code der Oberklasse wird nicht geändert
 - Schlüsselwort **extends** in Unterklasse
 - Schnittstellenvererbung → Implementierungsklassen, Konkretisierung von abstrakten Klassen
 - Code der Schnittstelle bzw. abstrakten Klasse wird nicht geändert
 - Schlüsselwort **implements** in Implementierungsklasse bzw. Konkretisierung von abstrakten Methoden

Übersetzung von Vererbung im Klassen-Diagramm (Fallbeispiel)



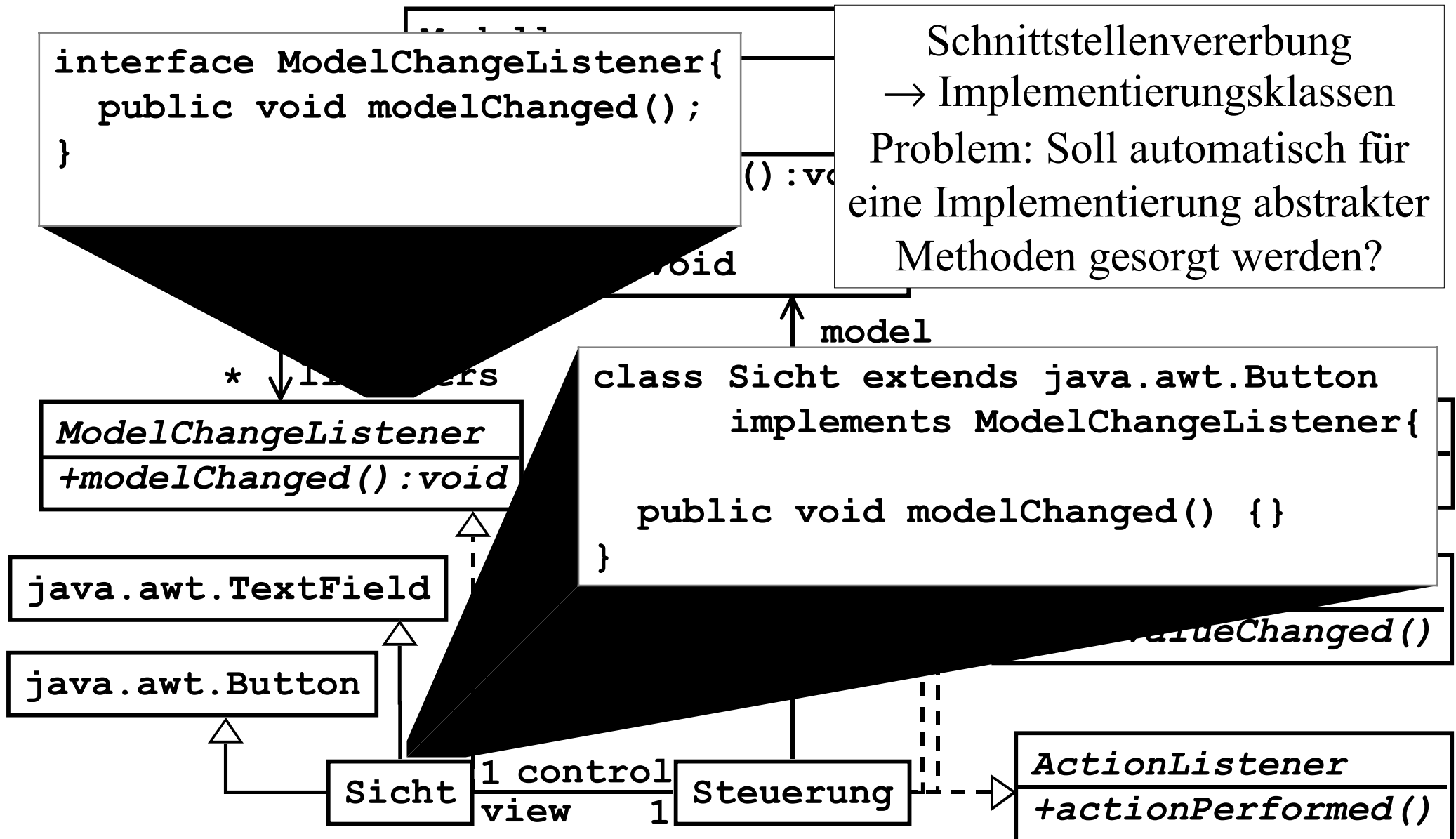
Übersetzung von Vererbung im Klassen-Diagramm (Fallbeispiel)



Probleme der Übersetzung von Vererbung nach Java

- ❖ Bei der Übersetzung von Vererbung in UML nach Java kann es allerdings zu Problemen kommen:
 - ***Schnittstellenvererbung***: Bei abstrakten Methoden und Implementierungsklassen muss für eine Implementierung der entsprechenden Methoden gesorgt werden. Soll diese automatisch erzeugt werden?
 - ***Implementierungsvererbung***: In UML ist Mehrfachvererbung erlaubt und für die Modellierung häufig sinnvoll, in Java ist die Mehrfachvererbung von Implementierungen nicht erlaubt!

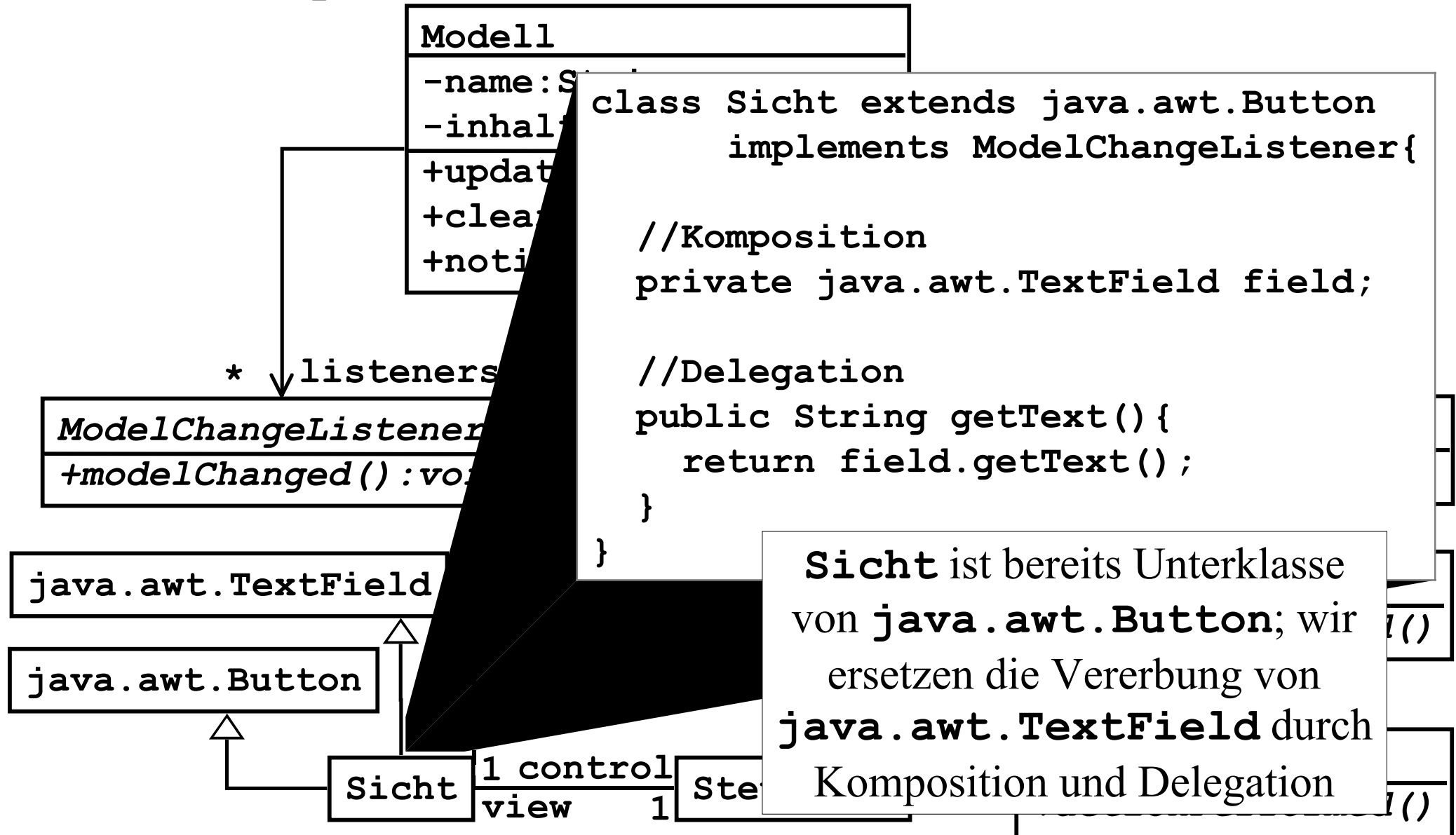
Probleme der Übersetzung von Vererbung (Fallbeispiel)



Probleme der Übersetzung von Mehrfachvererbung in Java

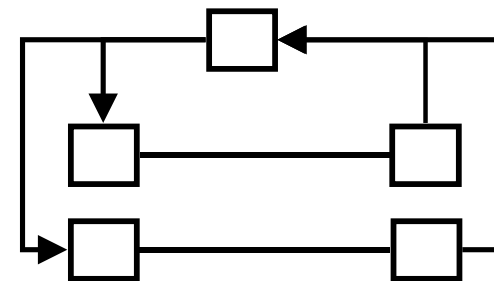
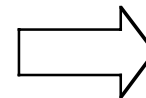
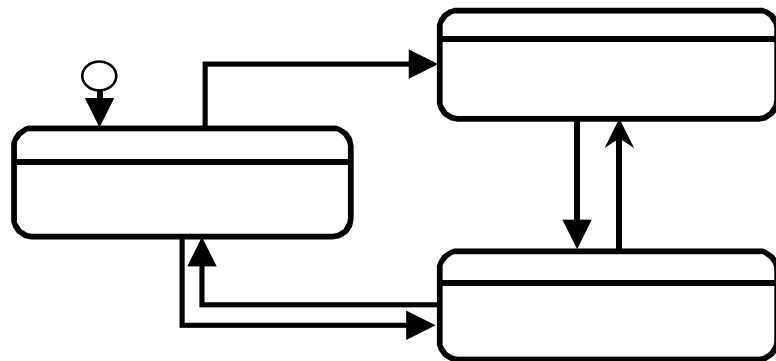
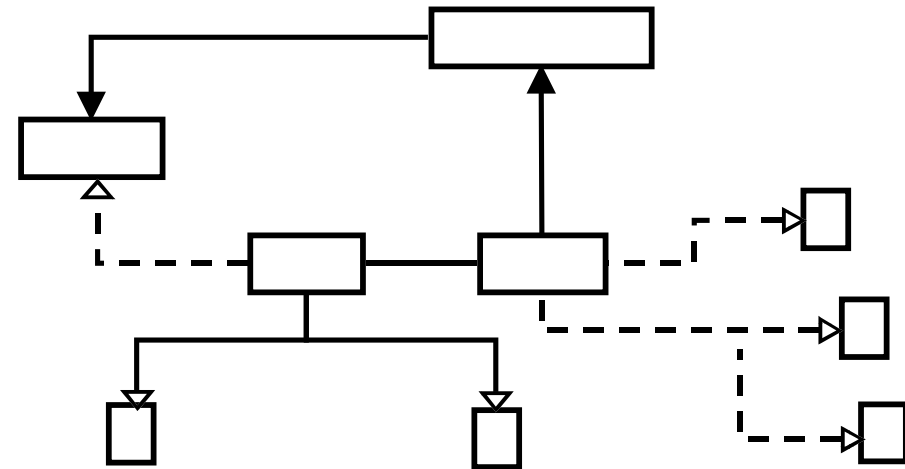
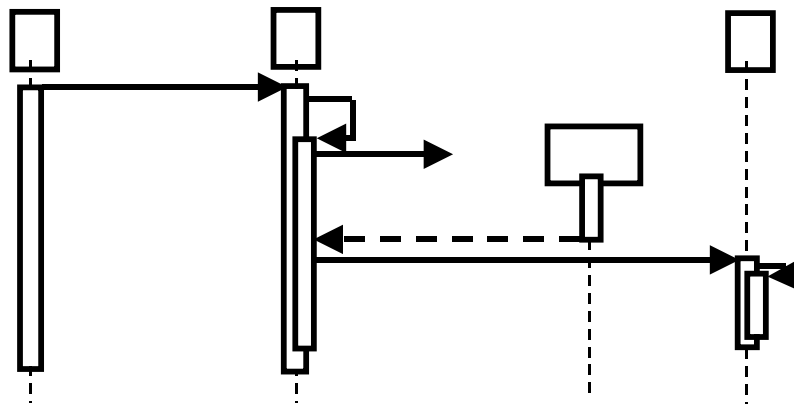
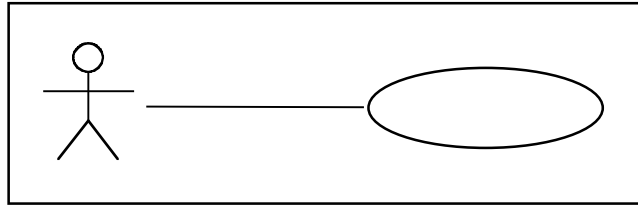
- ❖ Will man die Merkmale mehrerer Klassen in einer Klasse vereinigen, so kann bei der Modellierung Mehrfachvererbung eingesetzt werden
- ❖ Mehrfache Implementierungsvererbung ist in Java nicht möglich
- ❖ Als Hilfsmittel der gleichzeitigen Benutzung von Merkmalen mehrerer Klassen dient uns die Ersetzung der Vererbung durch
 - **Komposition/Aggregation:** von einer oder mehreren der gewünschten Klassen wird eine Instanzvariable angelegt
 - **Delegation:** für jedes gewünschte Merkmal der aggregierten Klasse wird eine Methode definiert (evtl. sogar gleichen Namens), in der auf der Instanzvariable die entsprechende Methode aufgerufen wird. *Der Aufruf wird also weitergegeben.* Diese Weitergabe bezeichnet man als **Delegation.**

Problem der Übersetzung von Mehrfachvererbung (Fallbeispiel)

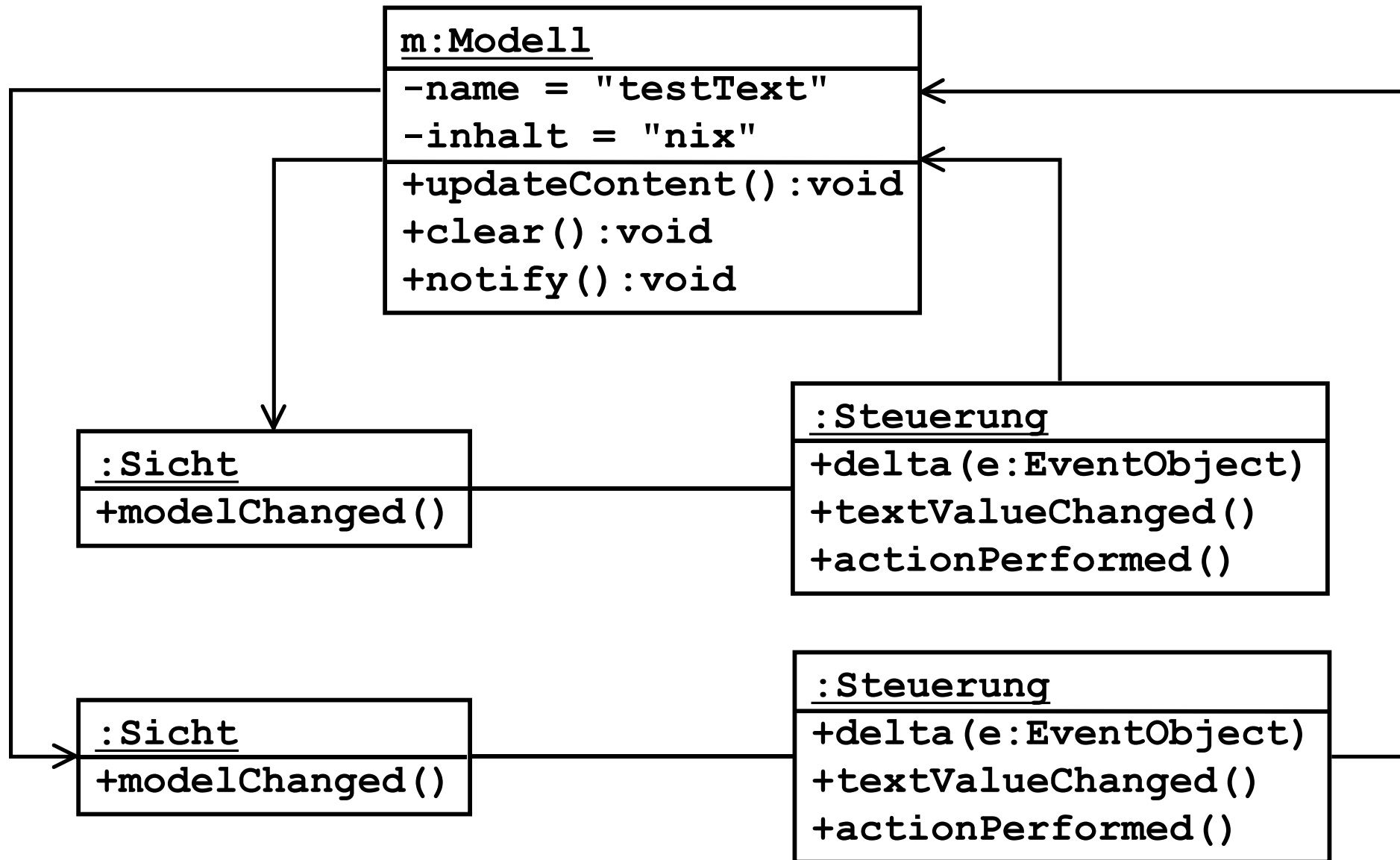


Sicht ist bereits Unterklasse von `java.awt.Button`; wir ersetzen die Vererbung von `java.awt.TextField` durch Komposition und Delegation

Überblick über unser Implementierungs-Modell (Vogelperspektive)



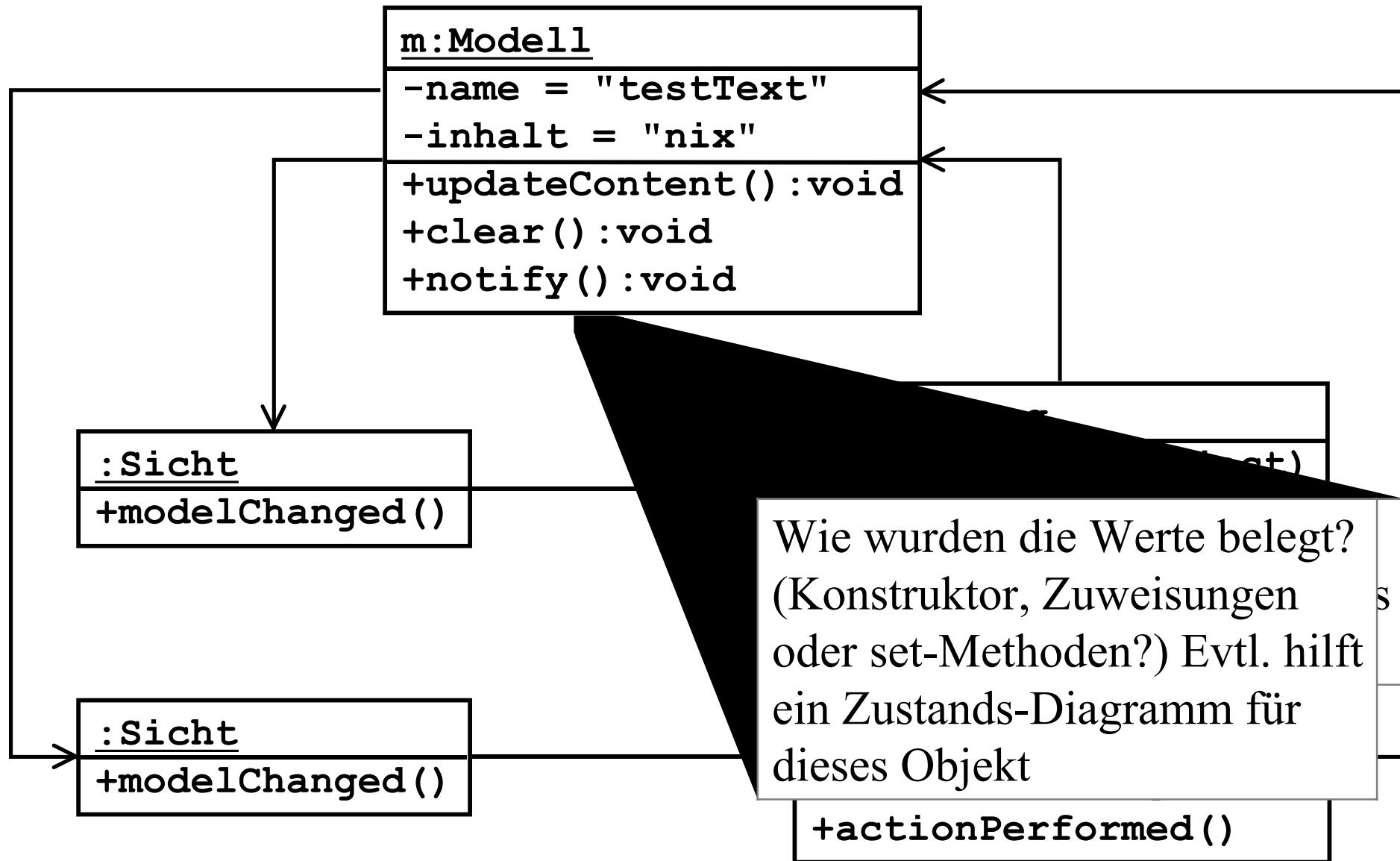
Instanz-Diagramm aus unserem Implementierungs-Modell



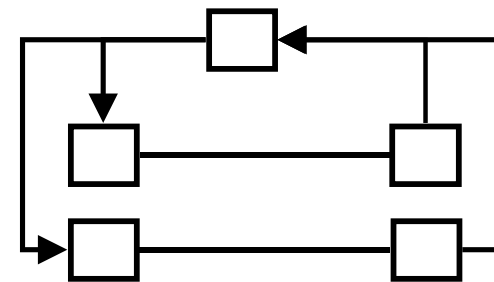
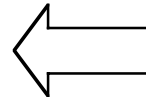
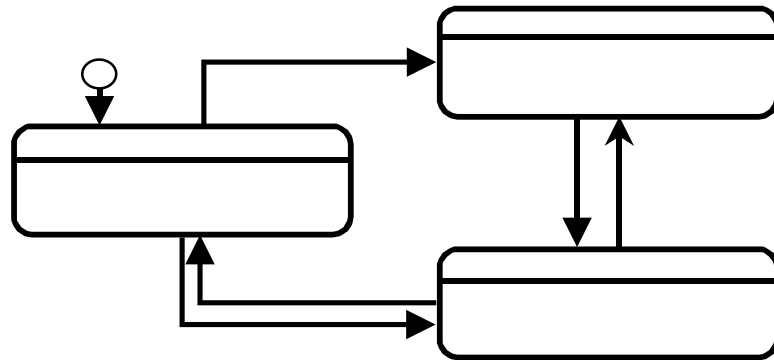
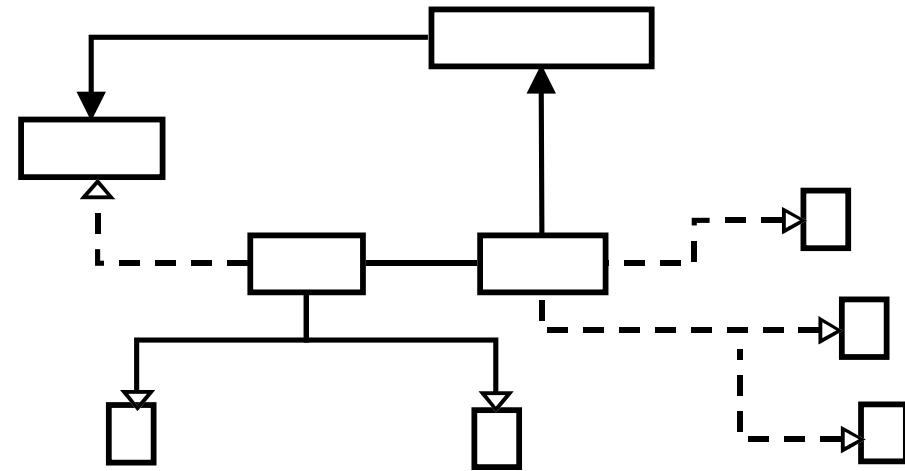
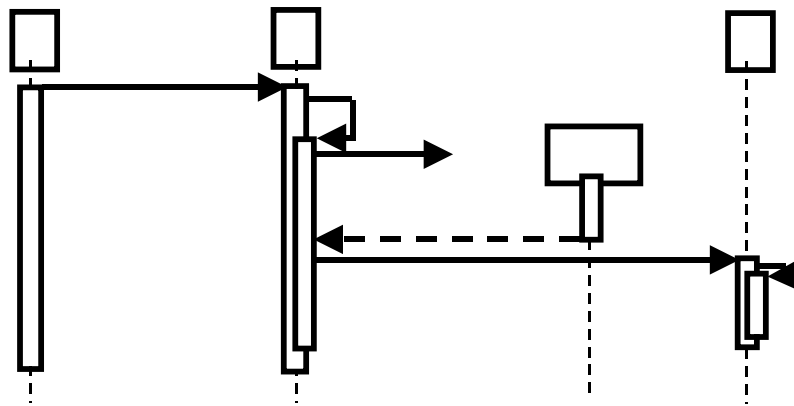
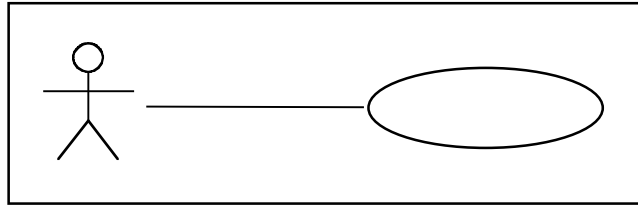
Übersetzung von Instanz-Diagrammen

- ❖ Instanz-Diagramme geben Augenblicksaufnahmen der existierenden Objekte in einem bestimmten Systemzustand an.
- ❖ Für die Umsetzung in Programmcode sind folgende Punkte zu überlegen:
 - Wo bzw. von wem wird das Objekt erzeugt?
 - Wie wird das Objekt erzeugt und initialisiert?
 - Wann wird das Objekt erzeugt? (Bedingungen)
 - Wie bekam das Objekt seine aktuellen Attributwerte?
- ❖ Diese Fragen können meist nur im Zusammenspiel mit anderen Diagrammen beantwortet werden!

Übersetzung von Instanz-Diagrammen (Fallbeispiel)

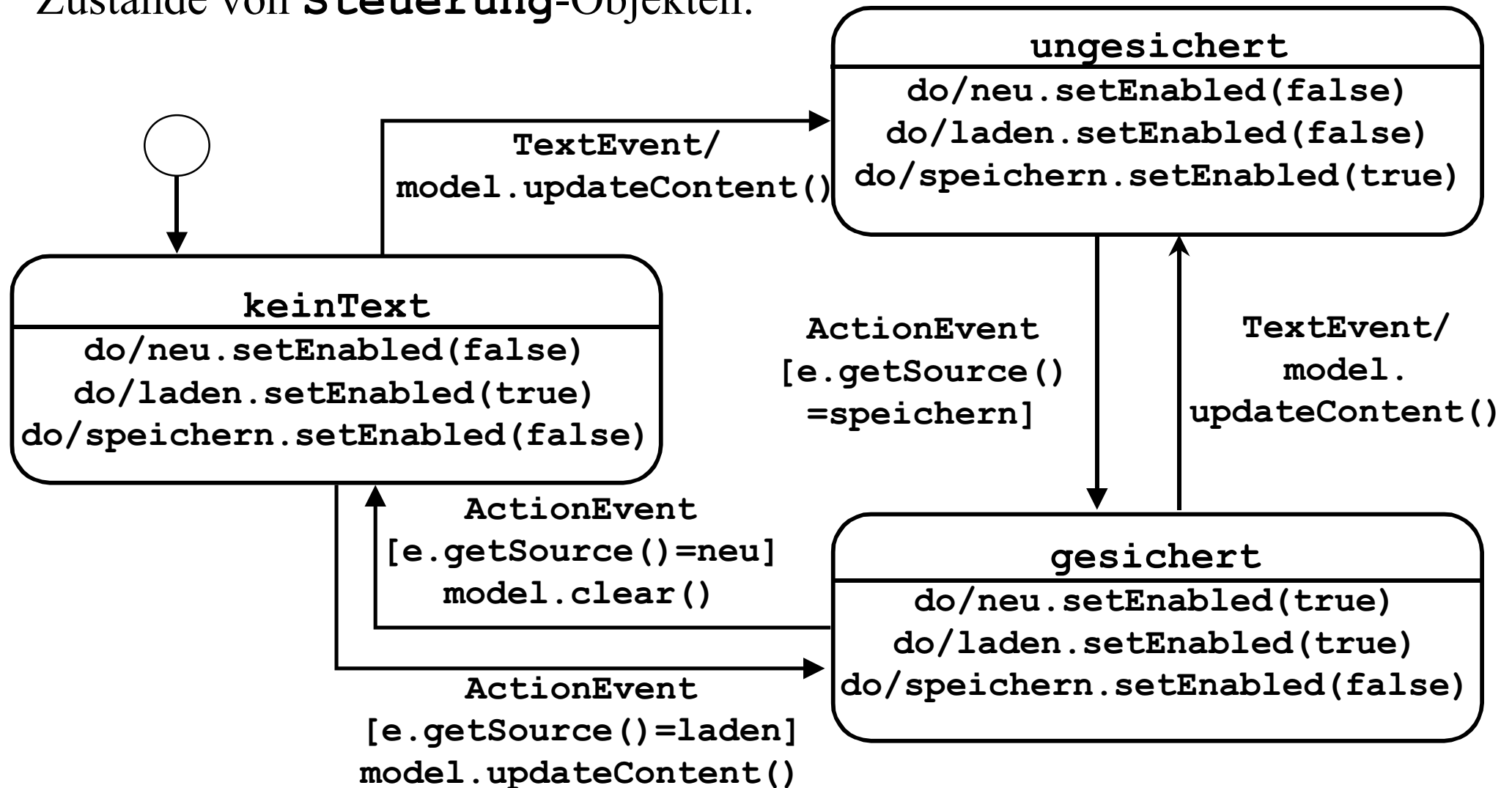


Überblick über unser Implementierungs-Modell (Vogelperspektive)



Zustands-Diagramm der Steuerung

Zustände von **Steuerung**-Objekten:



Übersetzung von Zustands-Diagrammen

- ❖ Für die Umsetzung in Programmcode sind folgende Punkte zu überlegen:
 - wo im Code finden Zustandsübergänge statt? (in welchen Methoden stehen die Zuweisungen, die die Übergänge verursachen?)
 - auf welche Objekte beziehen sich Wächter?
 - welche Objektinstanzen führen Aktionen und Aktivitäten aus?
- ❖ Wir betrachten nun Objekte, für die das Zustands-Diagramm definiert wurde, als Harel-Automaten (daher auch die Implementierung der Automaten-Schnittstelle mit der **delta ()**-Methode im Klassendiagramm)
 - Die gesamte Steuerung findet dann in **delta ()** statt, andere Methodenaufrufe (z.B. die Standardmethoden gewisser Ereignisempfänger) müssen dorthin weitergeleitet werden

Übersetzung von Zustands-Diagrammen

- ❖ Die Zustände müssen irgendwie repräsentiert werden
- ❖ Verschiedene Realisierungsmöglichkeiten der Zustände:
 - als Konstante in der Klasse
 - als eigenes Zustandsobjekt
- ❖ Verschiedene Realisierungsmöglichkeiten des Zustandsübergangs:
 - explizite Fallunterscheidung (**if-else** Verschachtelungen nach Zustand und Ereignissen)
 - Übergangstabelle (z.B. Reihung)
 - Verkapselung der Zustandsübergänge im Zustands-Objekt/Klasse → *state pattern* (Vertiefungsvorlesung)

Wir behandeln hier nur die explizite Fallunterscheidung

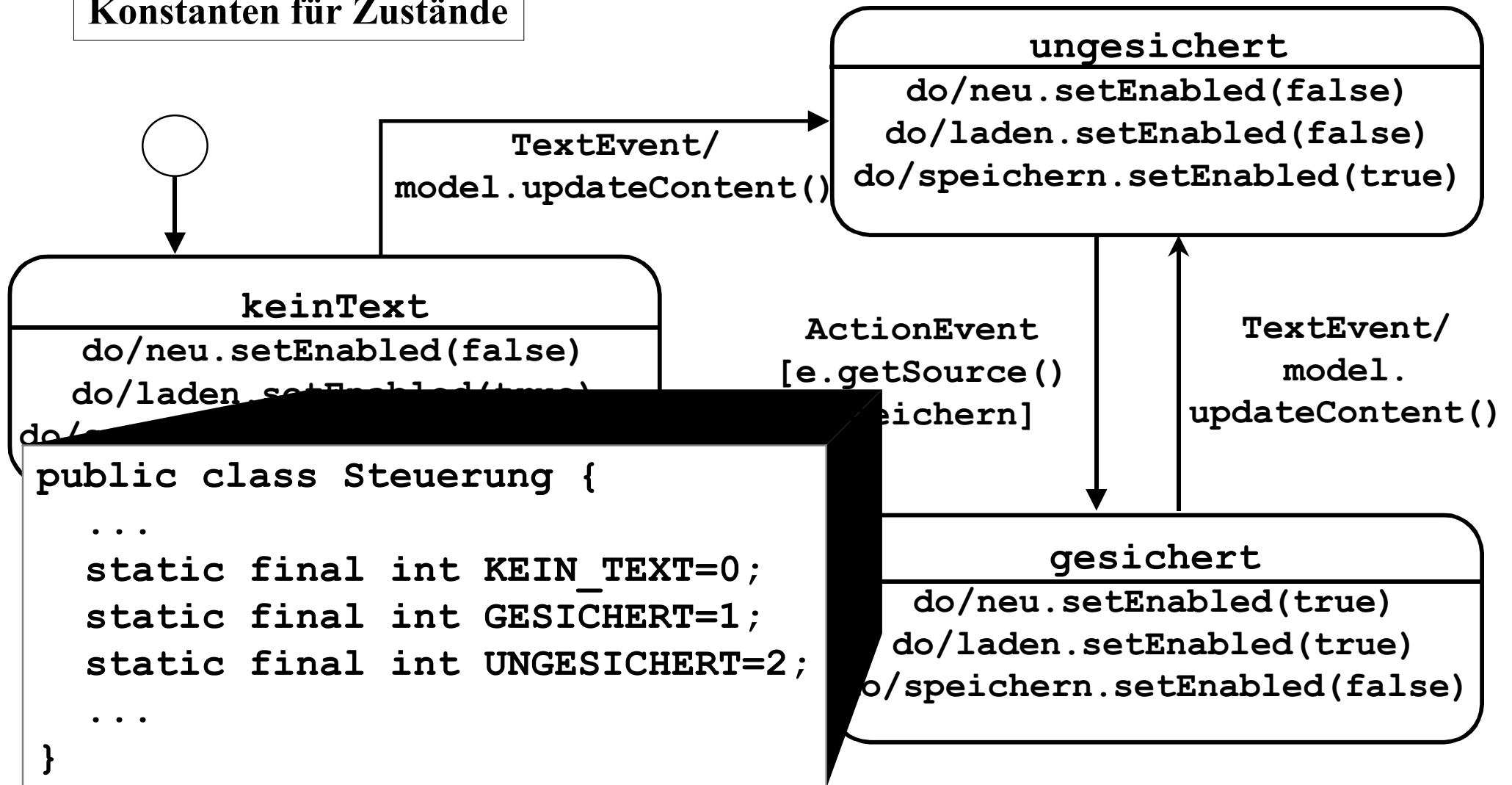
Übersetzung von Zustands-Diagrammen

Realisierungsvorschlag mit expliziter Fallunterscheidung:

- ❖ Implementierung von Zuständen
 - Definition von Konstanten für Zustände
 - Aktueller Zustand wird als Attribut repräsentiert
- ❖ Implementierung der Zustandsübergänge
 - Fallunterscheidung nach Zuständen
 - darin verschachtelt Fallunterscheidung nach Ereignissen (Eingabezeichen)
 - evtl. boolesche Verknüpfung mit einem Wächter
 - Anweisungsfolge für Aktion, Transition, Aktivitäten
- ❖ Delegation von Methoden an Zustandsübergangsfunktion

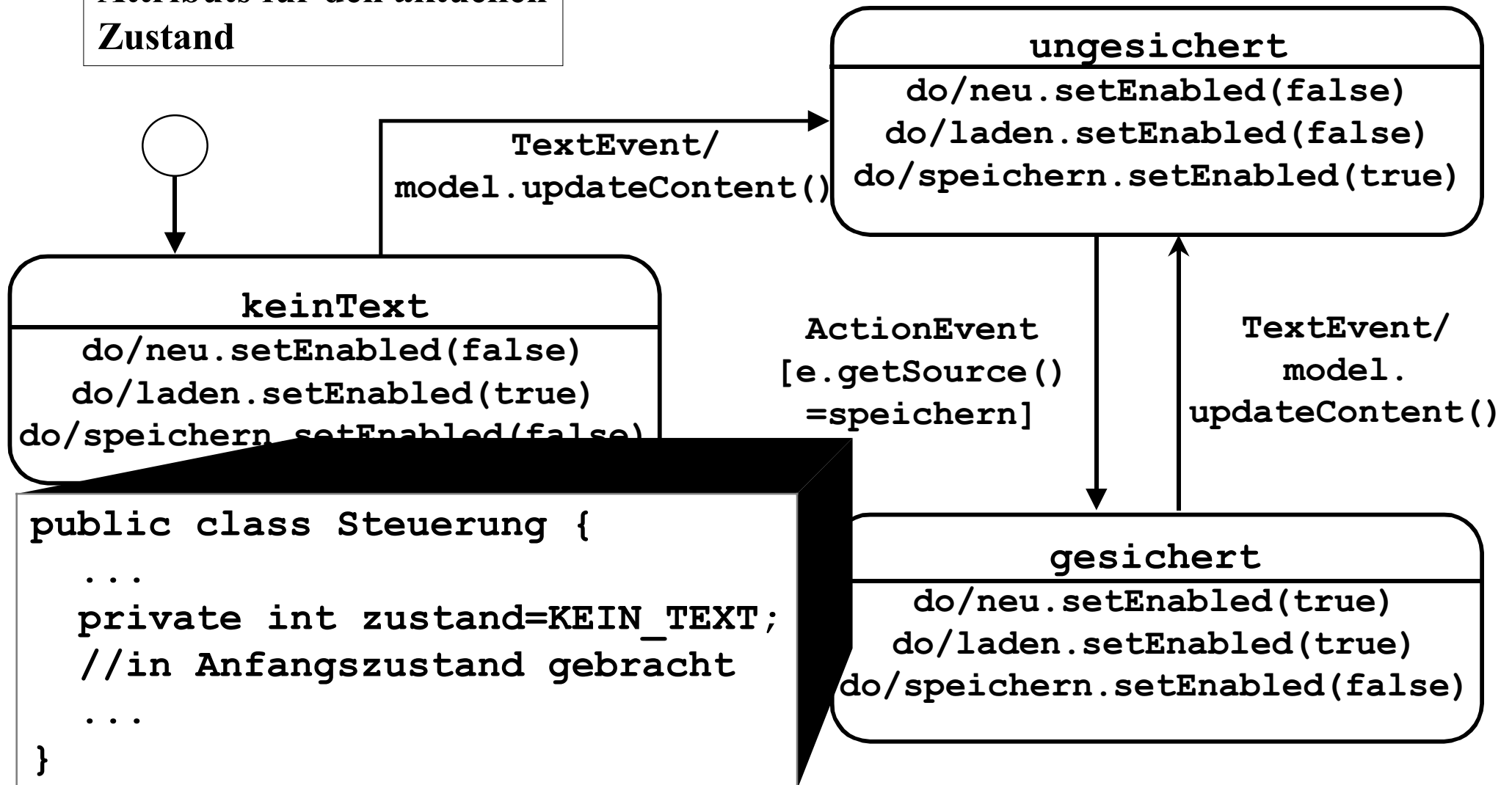
Übersetzung von Zustands-Diagrammen (Fallbeispiel)

1. Schritt: Definition von Konstanten für Zustände



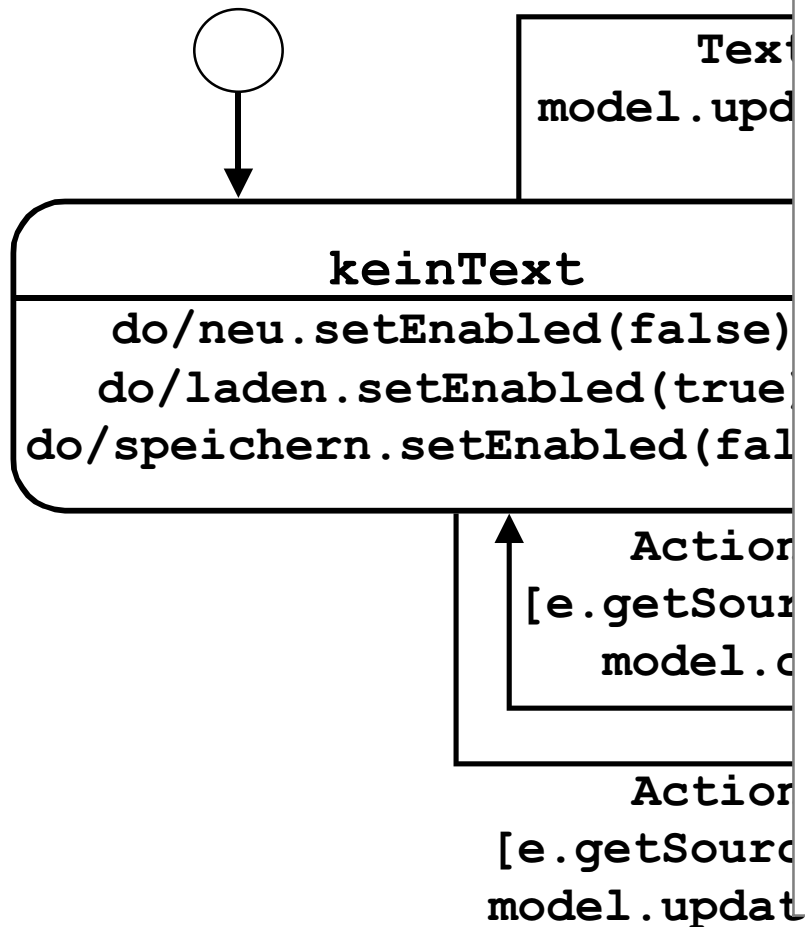
Übersetzung von Zustands-Diagrammen (Fallbeispiel)

2. Schritt: Definition eines Attributs für den aktuellen Zustand



Übersetzung von Zu

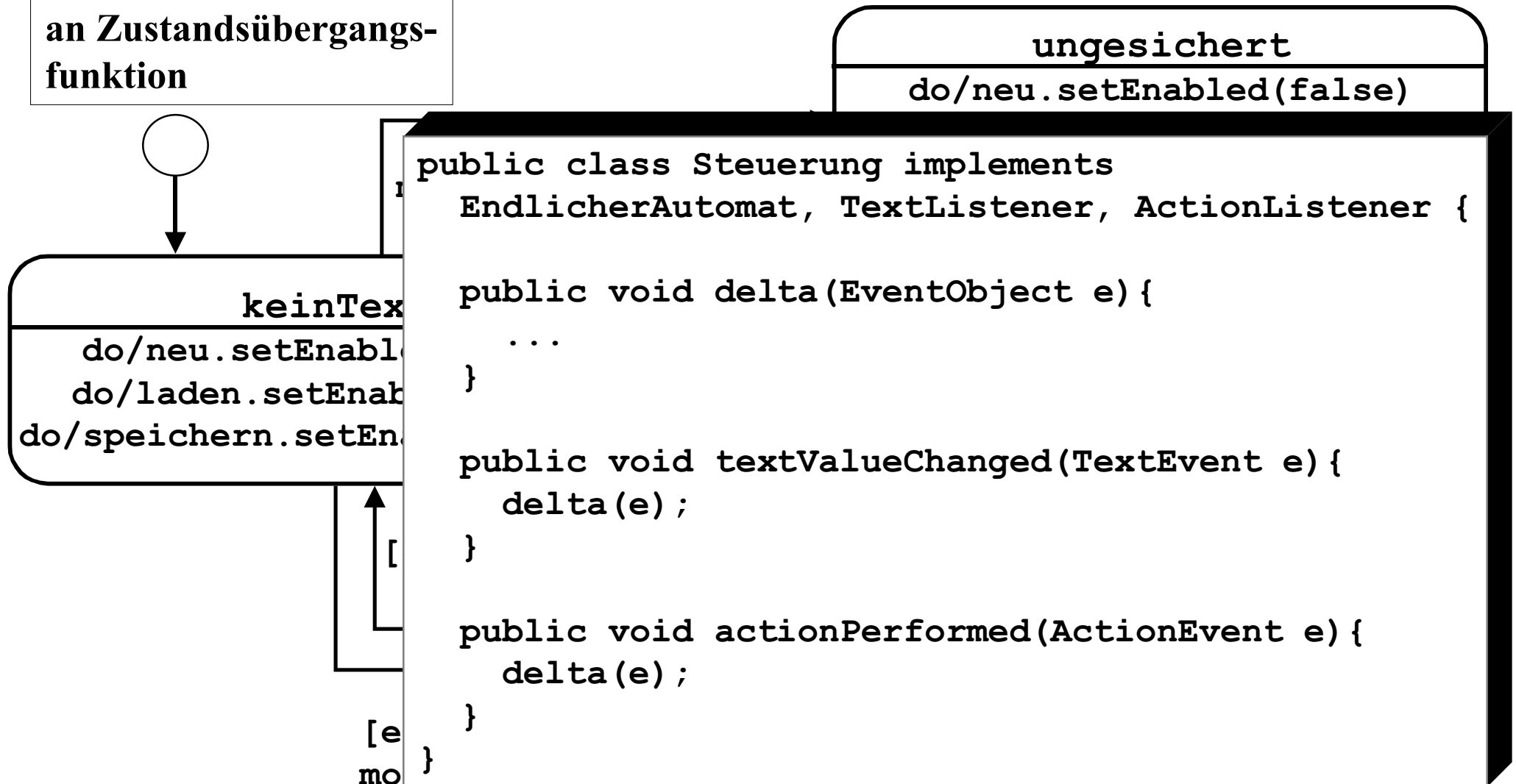
3. Schritt: Implementieren der Zustandsübergänge mit verschachtelten bedingten Anweisungen



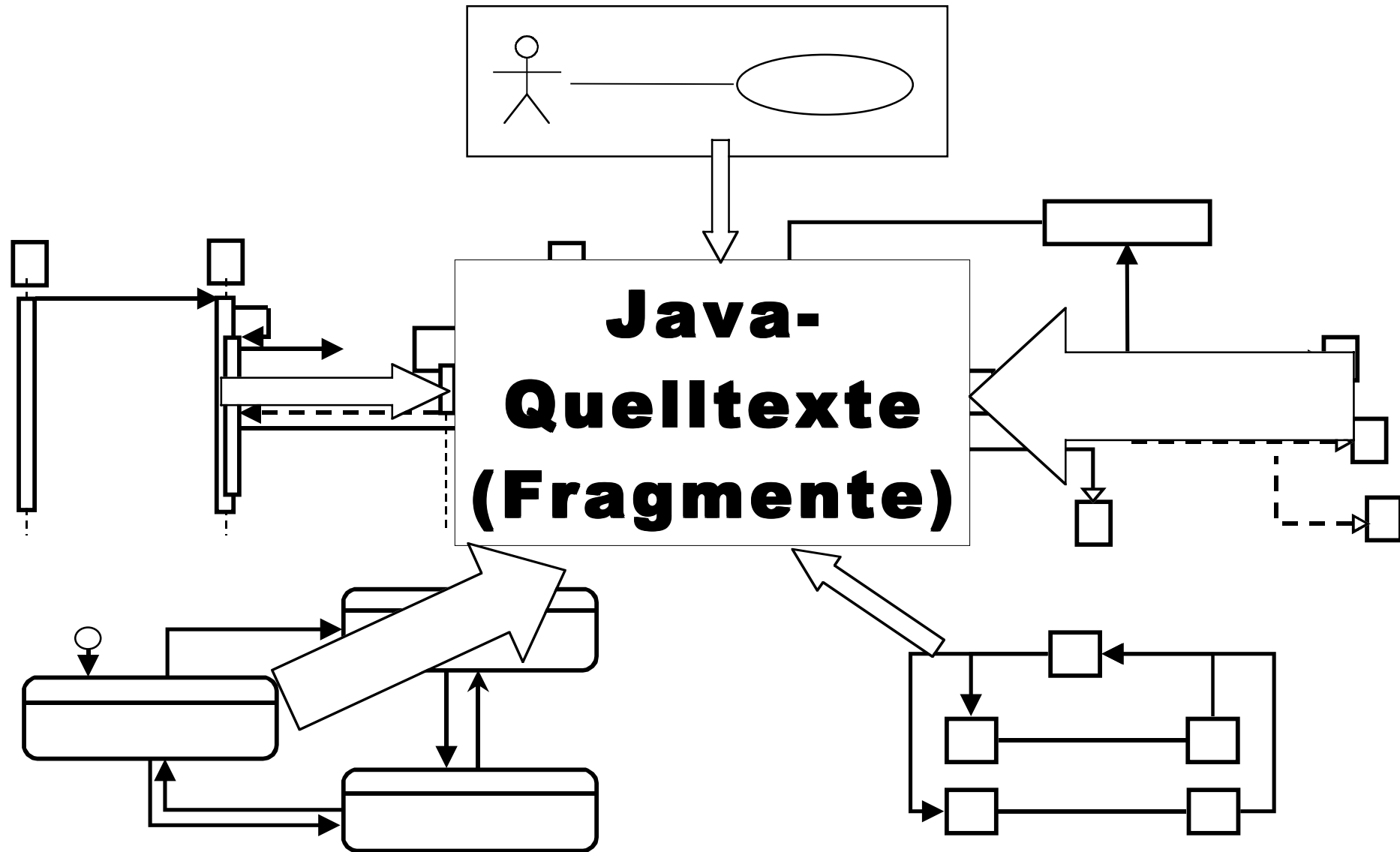
```
public class Steuerung {
    ...
    public void delta(EventObject e) {
        if (zustand == KEIN_TEXT) {
            if (e instanceof TextEvent) {
                model.updateContent();
                zustand = UNGESICHERT;
                neu.setEnabled(false);
                laden.setEnabled(false);
                speichern.setEnabled(true);
            }
            else if ((e instanceof ActionEvent)
                && (e.getSource() == laden)) {
                model.updateContent();
                zustand = GESICHERT;
                ...
            }
        }
        else if (zustand == GESICHERT) ...
    }
    ...
}
```

Übersetzung von Zustands-Diagrammen (Fallbeispiel)

4. Schritt: Delegation
von Methoden
an Zustandsübergangs-
funktion



Vom Modell zu Programm-Quelltexten



Grenzen der Übersetzung vom Modell ins Programm

- ❖ Überprüfung der Kombination der verschiedenen UML Diagrammsichten nach 3 Kriterien
 - ***Komplementarität*** (ergänzen sie sich gegenseitig?)
 - stellen Sequenz-Diagramme alle Anwendungsfälle dar?
 - Informationen über Objekte aus Instanz-Diagrammen
 - ***Konsistenz*** (widersprechen sie sich evtl. gegenseitig?)
 - passen Klassen- und Instanz-Diagramme zusammen?
 - können Nachrichten in Sequenz-Diagrammen überhaupt ausgetauscht werden (Signaturen und Sichtbarkeit)
 - ***Eindeutigkeit*** (gibt es nur eine Implementierung?)
 - i.A. nicht (z.B. unterschiedliche Sammlungs-Objekte)
- ❖ Für einige Diagramm-Notationen ist die Semantik in UML bisher nur unzureichend formalisiert

Werkzeuge zur Code-Generierung aus Modellen

- ❖ Für die Schritte, die wir manuell durchgeführt haben, gibt es zum Teil auch Werkzeuge (CASE-Tools)
- ❖ Einige verfügbare CASE-Tools:
 - Rational Rose, Together
- ❖ Hauptsächlich Generierung von Klassenrümpfen (statische Struktur des Entwurfs)
- ❖ Andere Diagramm-Typen werden meist nur ansatzweise umgesetzt, dienen eher zur Dokumentation
- ❖ Verwendung graphischer Beschreibungstechniken stellt neue Herausforderung dar
- ❖ Solche Werkzeuge sind momentan Gegenstand intensiver Forschung (wer macht mit? :-)

Vorgehensweisen zur Verknüpfung von Modellen und Programmen

- ❖ Übersetzung vom Modell ins Programm (forward engineering)
- ❖ Analyse des Programms führt zum Modell (reverse engineering)
- ❖ Iteratives Vorgehen: Modell und Programm beeinflussen sich in mehreren Zyklen gegenseitig (round trip engineering)
- ❖ Refaktorisieren: Änderungen auf Quellcode-Ebene verursachen auch Änderungen des vorhandenen Modells
- ❖ Themen aktueller Forschung im Bereich des Software-Engineering

Zusammenfassung

- ❖ Verschiedene Sprachebenen bei der Software-Entwicklung
- ❖ Grundlagen der Übersetzung
- ❖ Codegenerierung aus UML-Diagrammen
- ❖ Grenzen und Herausforderungen
- ❖ Vorgehensweisen zur Verknüpfung von Modellierung und Programmierung