

Software Engineering

Modeling Behavior

Authors of slides:

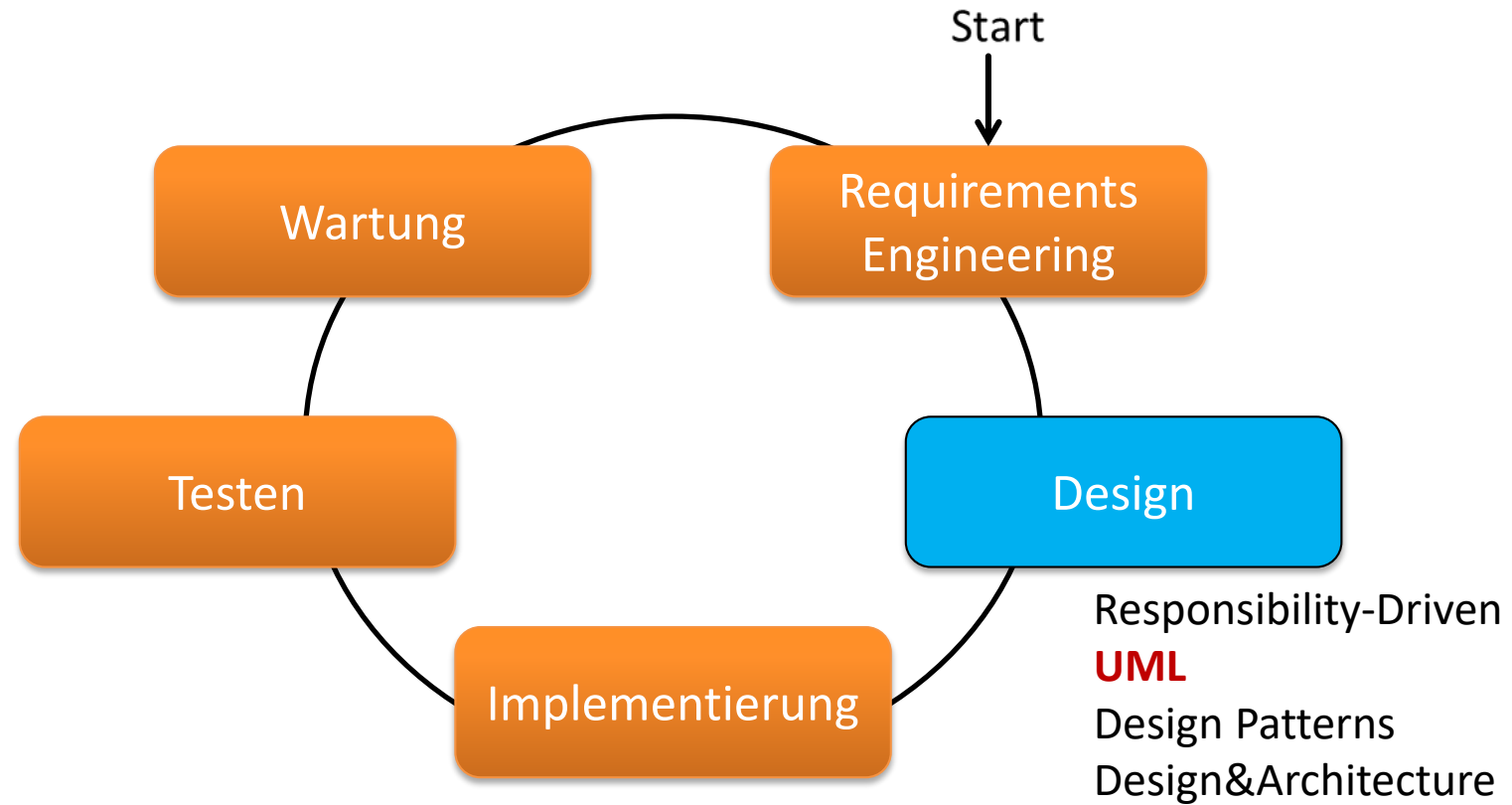
Norbert Siegmund

Janet Siegmund

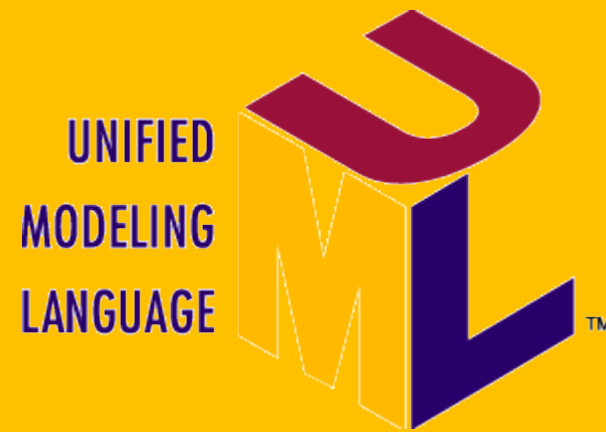
Oscar Nierstrasz

Sven Apel

Einordnung



Übersicht UML



UML

Was ist UML?

- Uniform notation: Booch + OMT + Use Cases (+ state charts)
 - UML ist *nicht* eine Methode oder ein Prozess
 - ... Der *Unified Development Process* hingegen schon...

Warum eine grafische Modellierungssprache?

- Software Projekte werden durch *Teams* bearbeitet
- Team Mitglieder müssen *kommunizieren*
 - ... manchmal sogar mit den Endbenutzern
- “Ein Bild sagt mehr als tausend Worte”
 - Die Frage ist nur *welche Worte*
 - Notwendigkeit *verschiedene Sichten* auf das selbe Software Artefakt (z.B. Code)

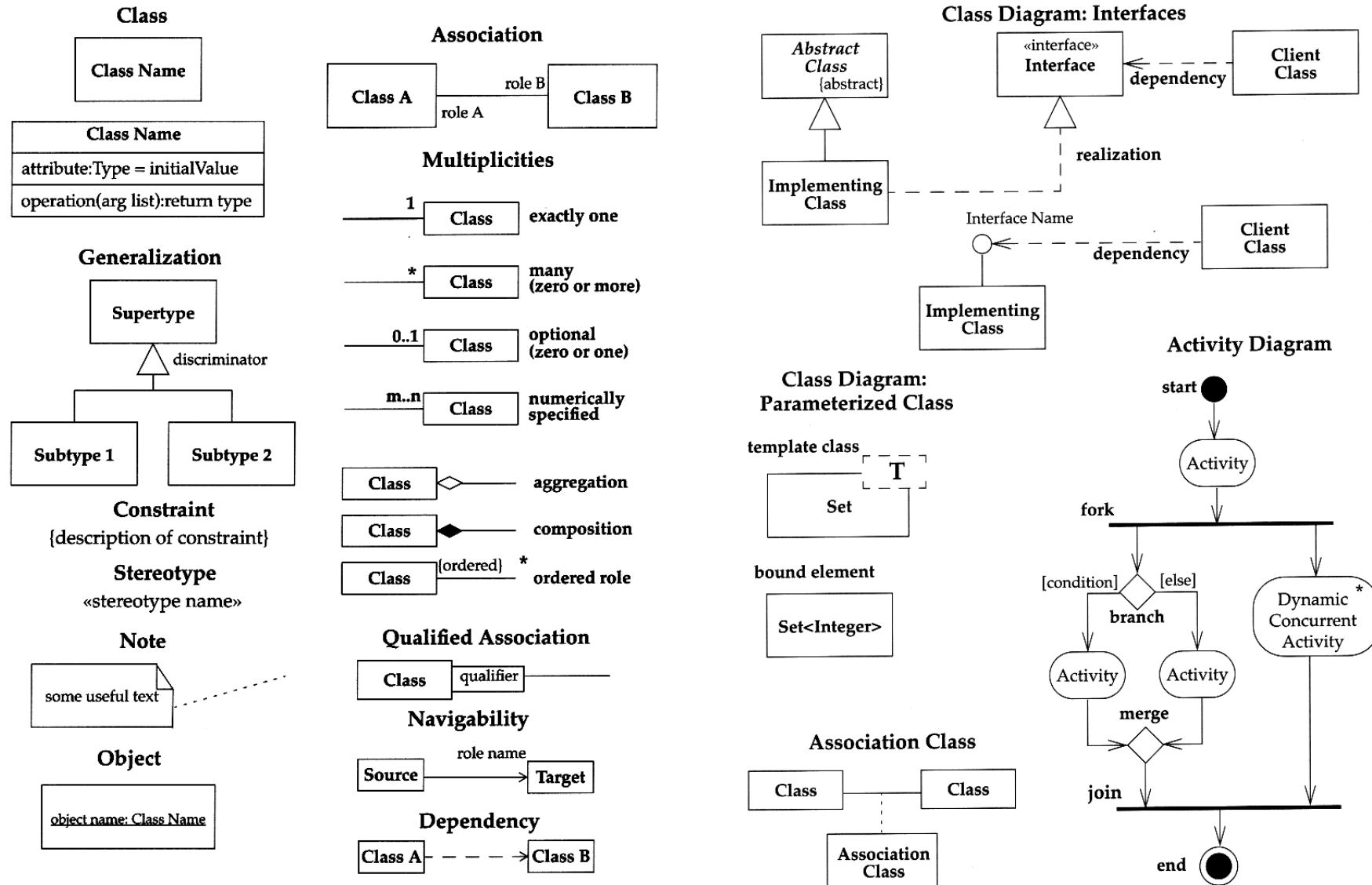
Warum UML?

Warum UML?

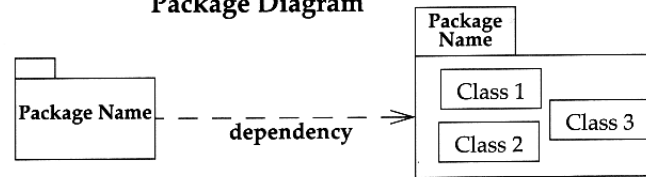
- Reduziert *Risiken* durch das Dokumentieren von Annahmen
 - Domänenmodelle, Requirements, Architektur, Design, Implementation ...
- Repräsentiert Industriestandard
 - Mehr Toolunterstützung, mehr Leute verstehen die Diagramme, weniger Ausbildung
- Ist hinreichend *gut-definiert*
 - ... obwohl es einige Interpretationen und Dialekte gibt
- Ist *offen*
 - Stereotypen, Tags und Bedingungen zur Erweiterung von Basiskonstrukten
 - Hat ein Meta-meta-modell für komplexe Erweiterungen

UML Geschichte

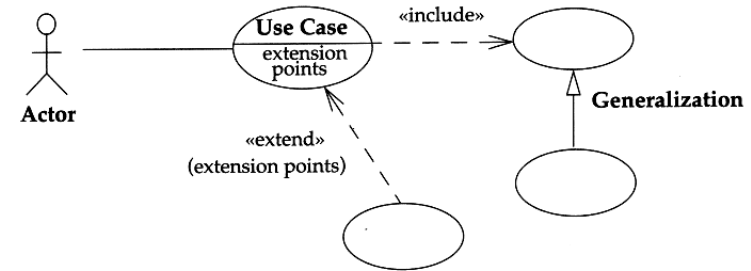
- 1994: Grady Booch (Booch method) + James Rumbaugh (OMT) in der Firma Rational
- 1994: Ivar Jacobson (OOSE, use cases) tritt Rational bei
 - “The three amigos”
- 1996: Rational gründet ein Konsortium, um UML zu unterstützen
- 1997: UML 1.0 bei der OMG eingereicht
- 1997: UML 1.1 als OMG-Standard akzeptiert
 - Aber, OMG benannte es UML 1.0
- 1998-...: Revisionen UML 1.2 - 1.5
- 2005: Hauptrevision zu UML 2.0, beinhaltet OCL (object constraint language)



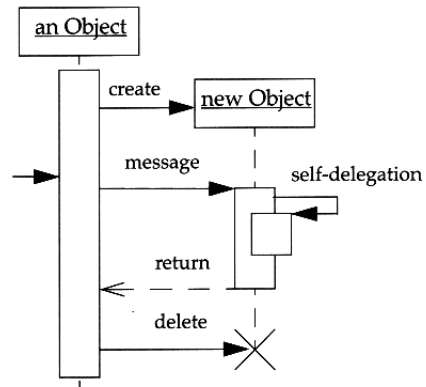
Package Diagram



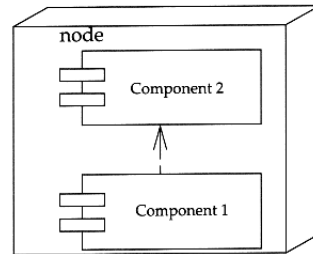
Use Case Diagram



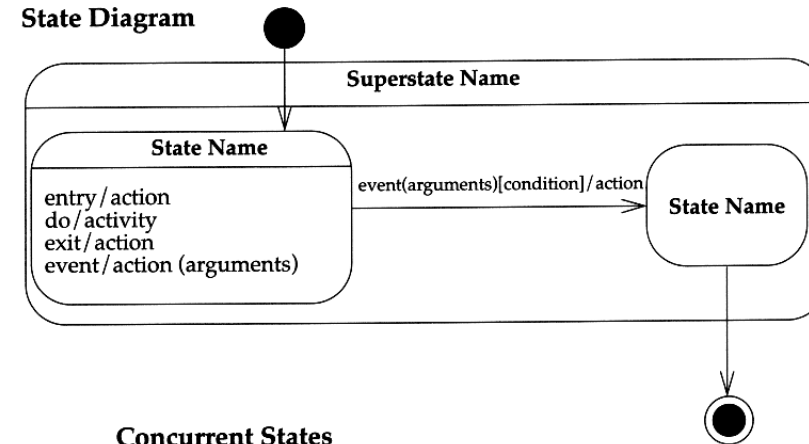
Sequence Diagram



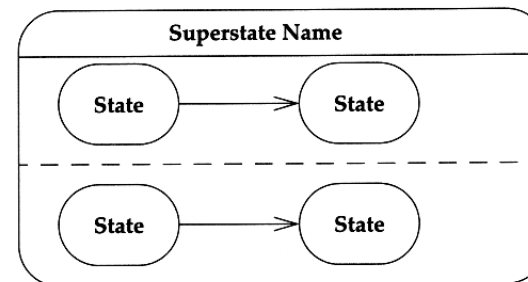
Deployment Diagram



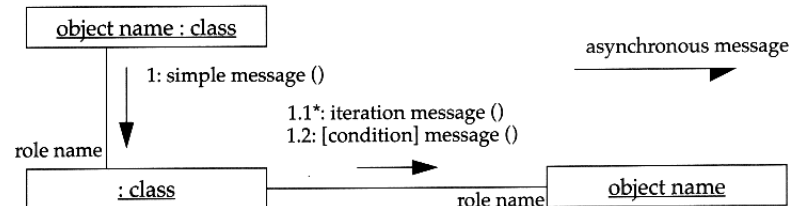
State Diagram



Concurrent States

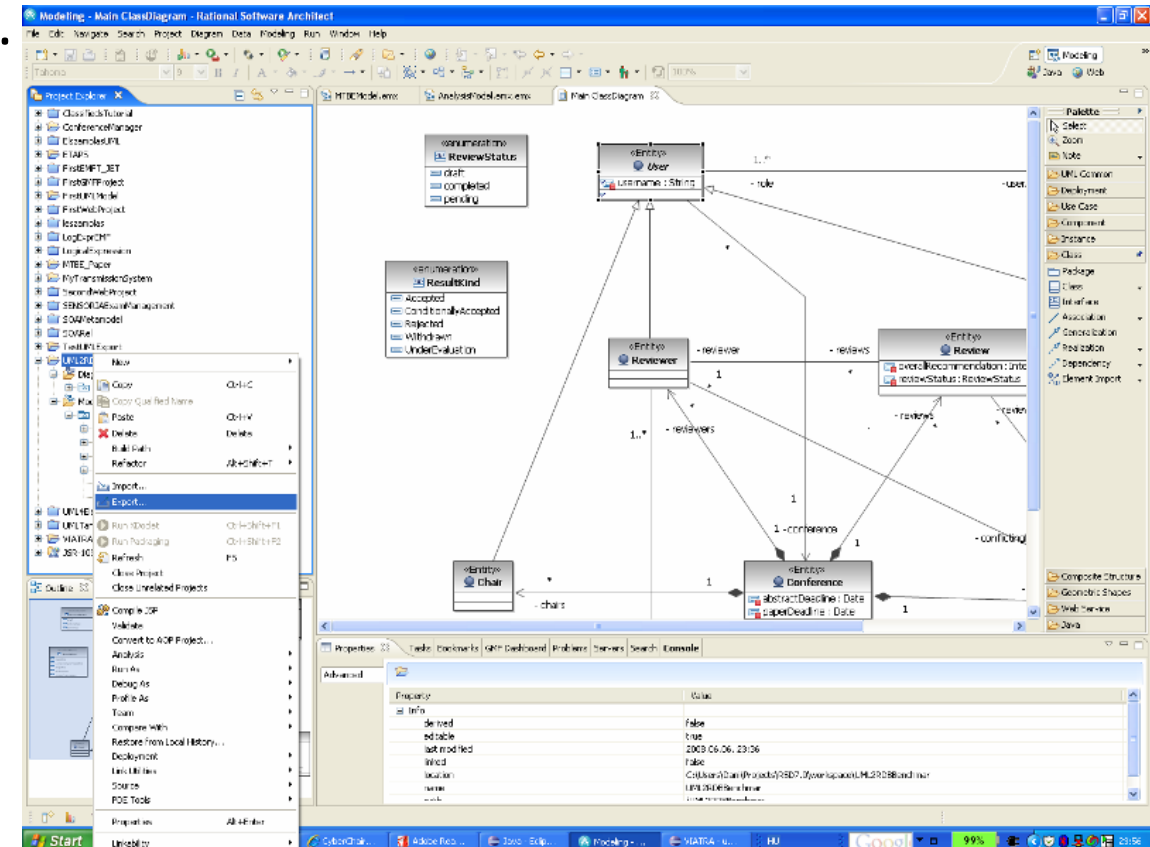


Collaboration Diagram



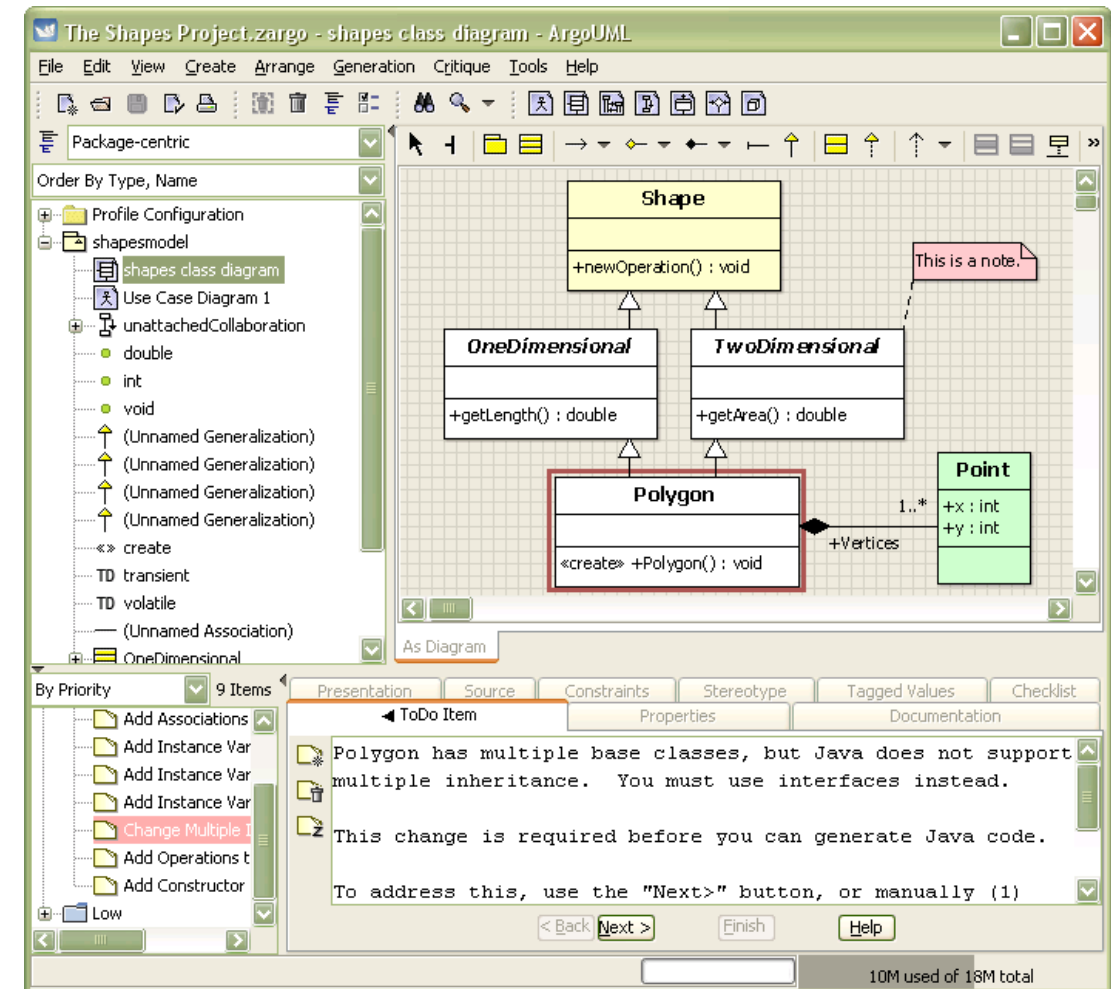
Tools: IBM Rational Software Architect

- Co-Entwicklung von Code und UML Modellen
 - Java, .Net, C++, WSDL, CORBA, ...
- Round-trip engineering
 - Code ↔ model

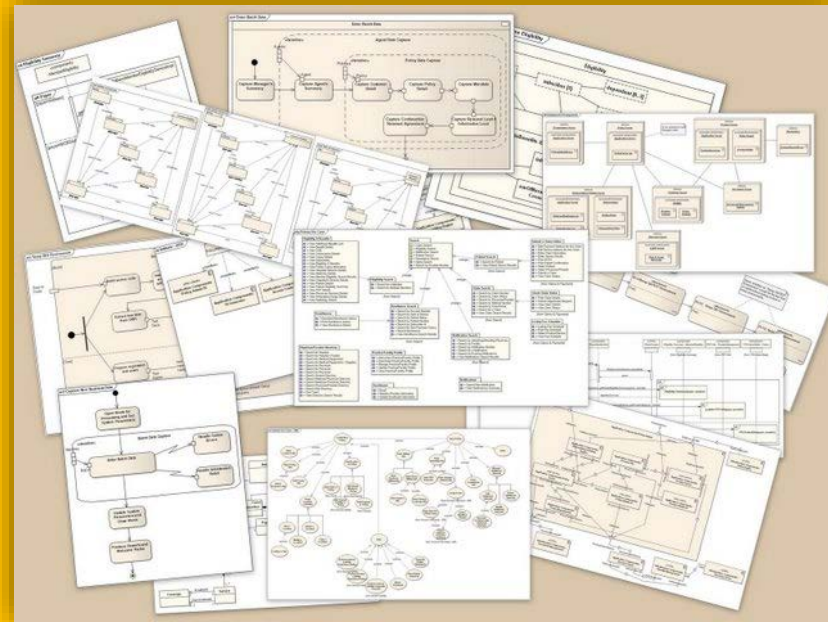


Tools: ArgoUML

- Open-source UML Modellierungswerkzeug
- Round-trip engineering
 - Java code \leftrightarrow model



Klassen, Attribute und Operationen



Klassendiagramme

“Class diagrams show generic descriptions of possible systems, and object diagrams show particular instantiations of systems and their behaviour.”

Attribute and Operationen werden zusammenfassend auch als *Features* bezeichnet.

Achtung: Klassendiagramme können oft in Datenmodelle übergehen. Der Fokus sollte auf dem Verhalten liegen.

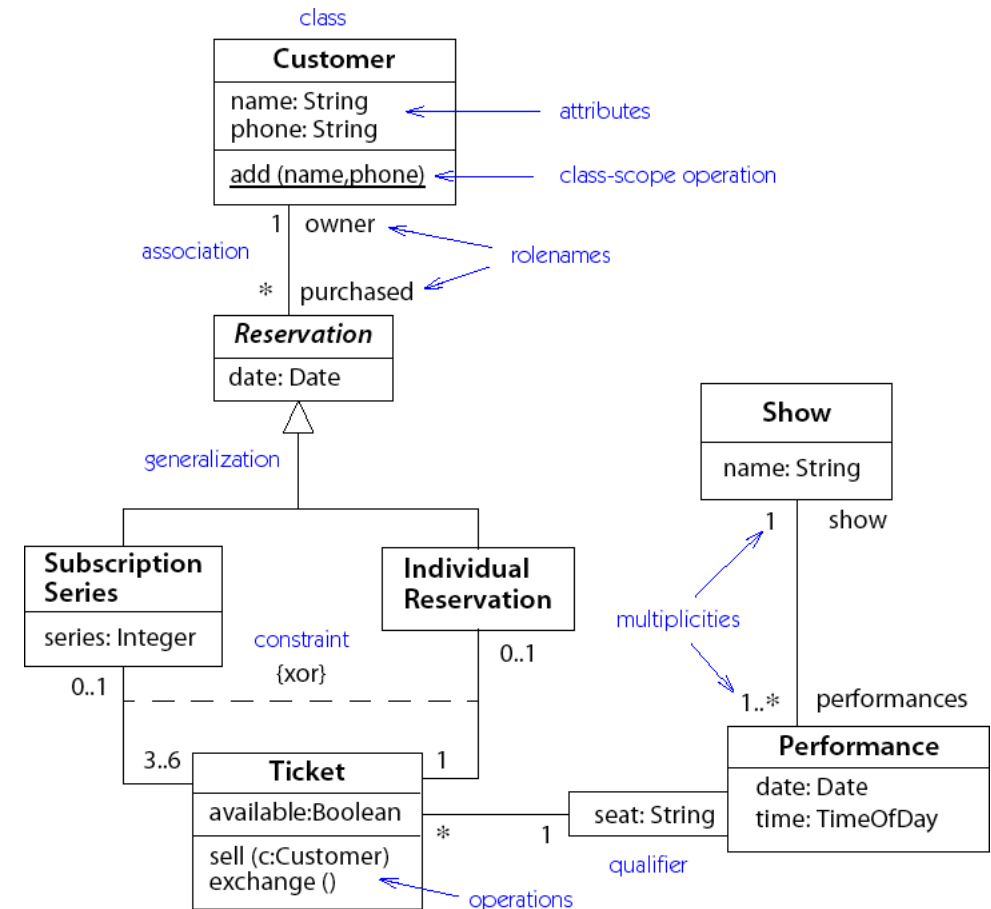


Figure 3-1. Class diagram

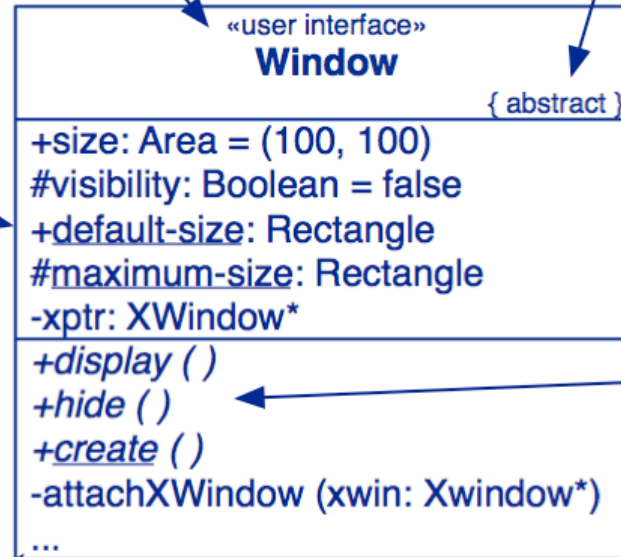
Sichtbarkeit und Scope (Geltungsbereich) von Features

Stereotype
(what “kind” of class is it?)

User-defined properties
(e.g., readonly, owner = “Pingu”)

underlined
attributes have
class scope

+ = “public”
= “protected”
- = “private”



italic attributes
are *abstract*

Kümmert euch
nicht zu früh um
Sichtbarkeit!

An ellipsis signals that further entries are not shown

Attribute und Operationen

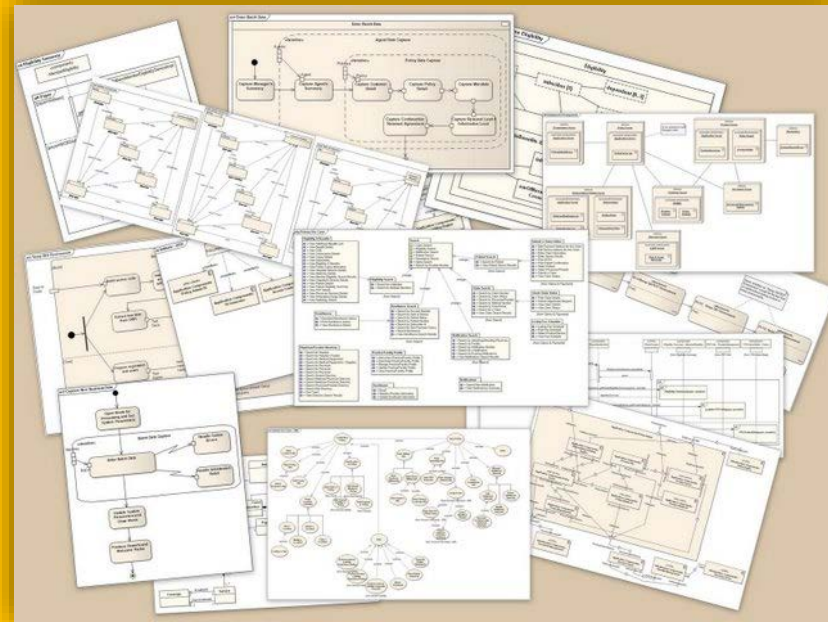
Attribute sind spezifiziert als:

name: type = initialValue { property string }
visibility: boolean = false









Operationen sind spezifiziert als:

direction name (param: type = defaultValue, ...) : resultType
in draw (position: Point): void

UML: Linien und Pfeile



UML Linien und Pfeile

	<i>Constraint</i> (usually annotated)		<i>Association</i> e.g., «uses»
	<i>Dependency</i> e.g., «requires», «imports» ...		<i>Navigable association</i> e.g., part-of
	<i>Realization</i> e.g., class/template, class/interface		<i>“Generalization”</i> i.e., specialization (!) e.g., class/superclass, concrete/abstract class
	<i>Aggregation</i> i.e., “consists of” (Innere Objekte existieren unabhängig von äußeren Objekten)		<i>“Composition”</i> i.e., containment (Existenz der inneren Objekte ist abhängig vom äußeren Objekt)

UML: Parameterisierte Klassen und Interfaces

Parametrisierte Klassen

Parametrisierte (aka “template” oder “generic”) Klassen sind gekennzeichnet durch ihre Parameter in der *gestrichelten Box*.

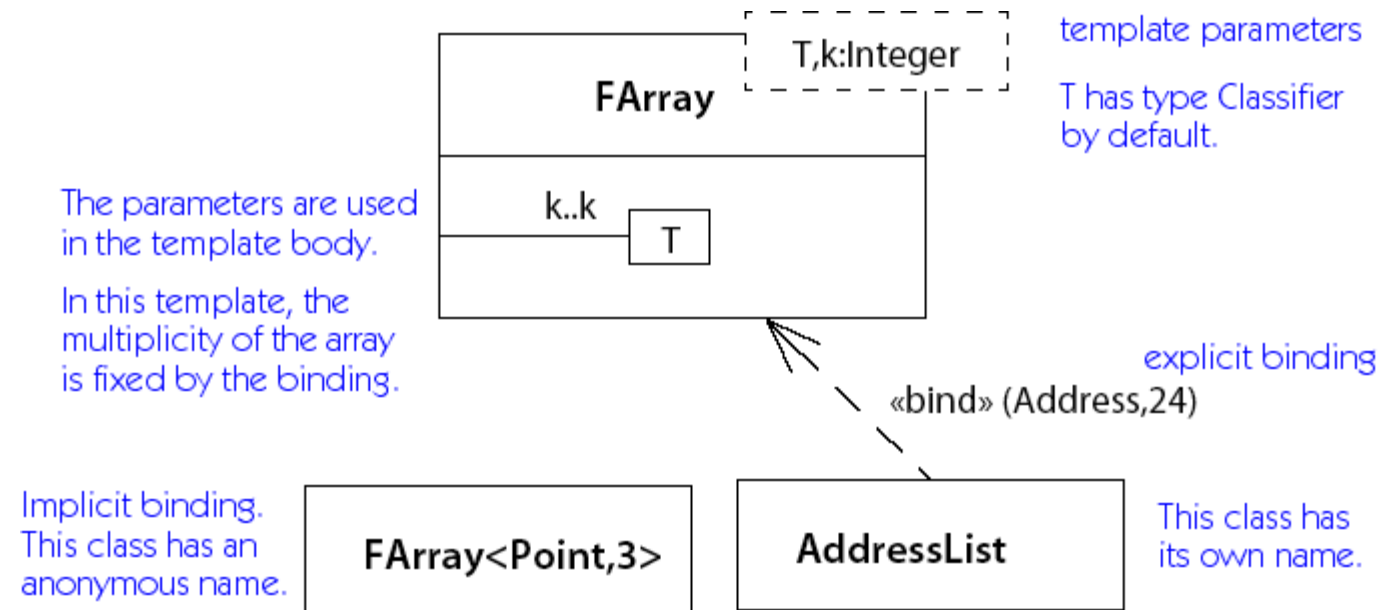


Figure 13-180. Template notation with use of parameter as a reference

Interfaces

Interfaces, äquivalent zu abstrakten Klassen ohne Attribute, werden repräsentiert als Klassen mit dem Stereotyp «interface» oder, alternativ, mit der “Lollipop-Notation”:

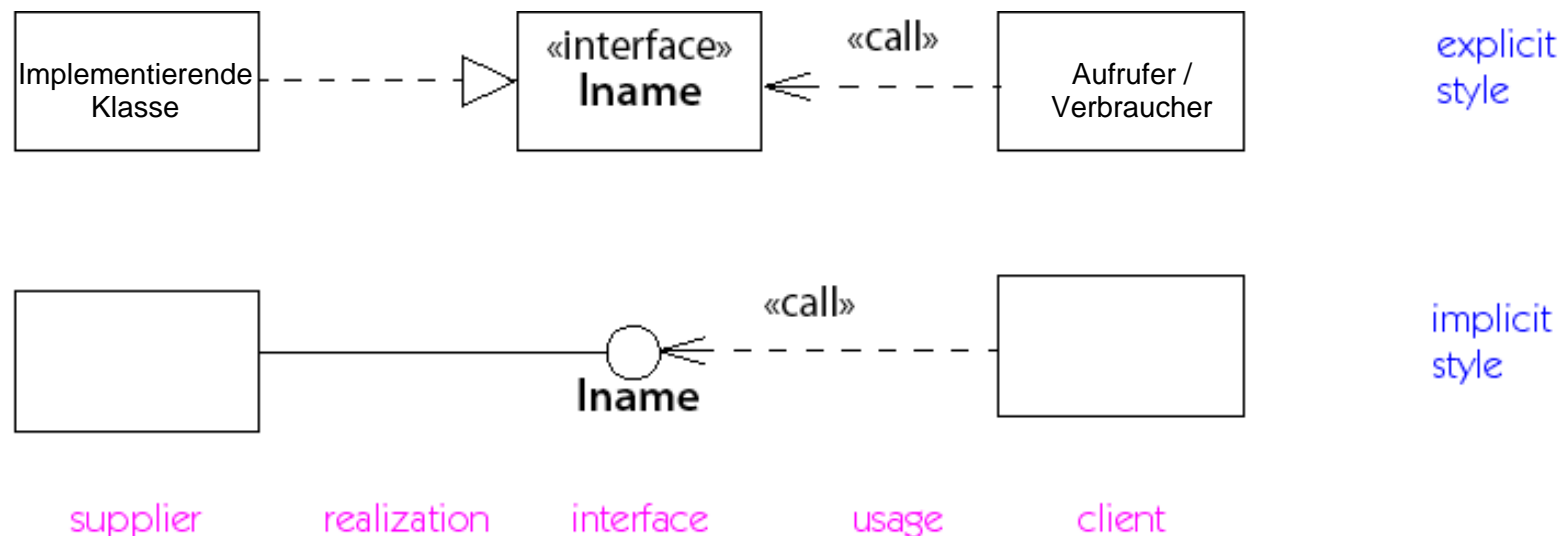


Figure B-5. *Realization of an interface*

Aufgabe

Modellieren Sie ein Interface mit den Methoden put, get, delete, welches durch eine LinkedList realisiert wird. Vergessen Sie den Aufrufer des Interfaces nicht.



UML: Objekte und Assoziationen

Objekte

Objekte werden als Rechtecke mit unterstrichenem Namen und Typ in einem Unterbereich dargestellt; Attributwerte optional in einem 2. Bereich.

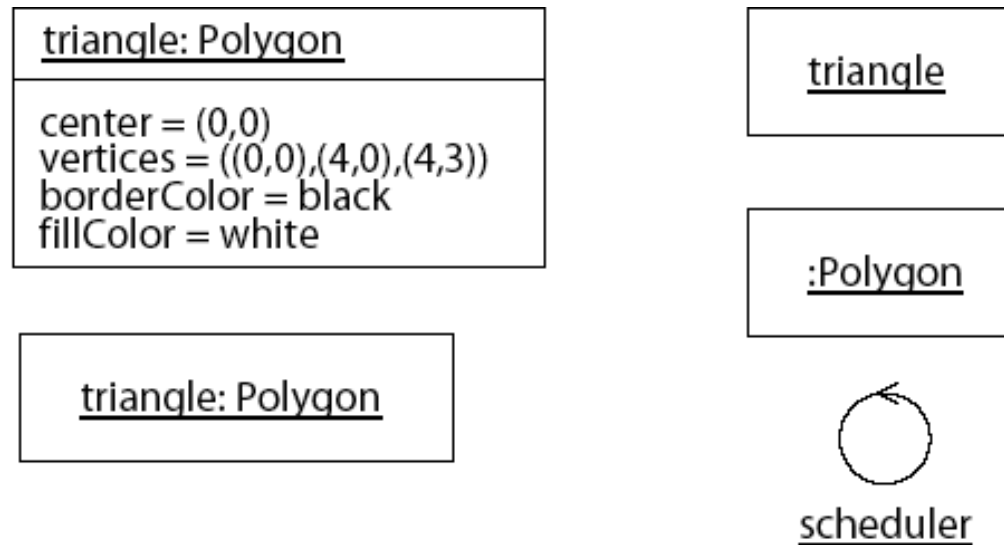


Figure 13-134. *Object notation*

Mindestens der Name oder der Typ muss angegeben werden.

Assoziationen

Assoziationen repräsentieren *strukturelle Beziehungen* zwischen Objekten

- gewöhnlich *binär* (aber möglich auch tertiär etc.)
- Optional *Name* und *Richtung*
- (unique) *Rollennamen* und *Multiplikatoren* an Endpunkten

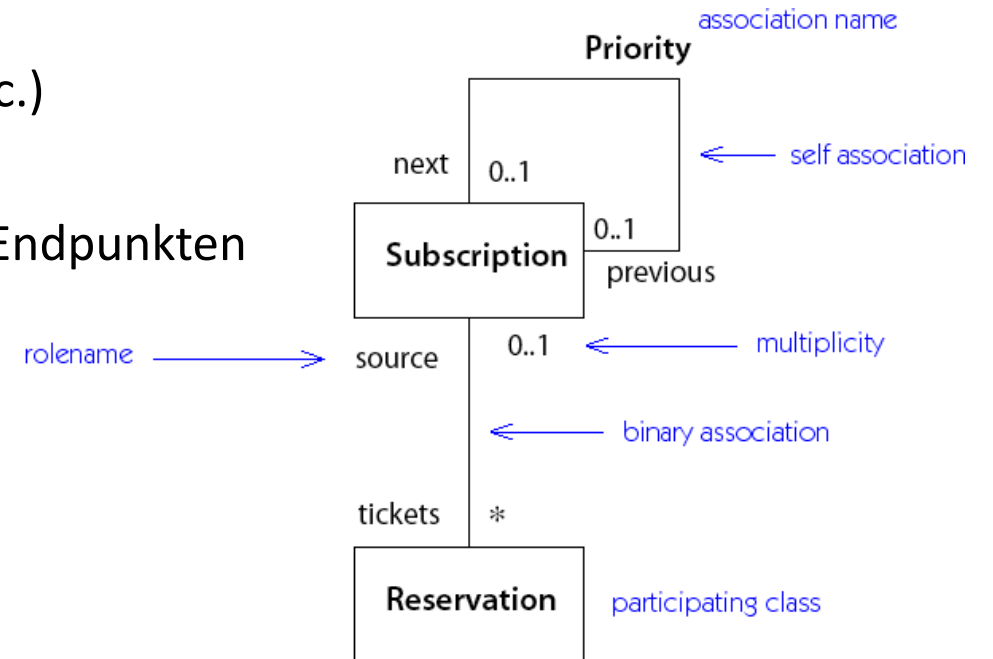


Figure 4-2. Association notation

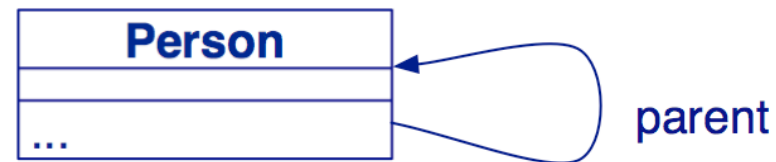
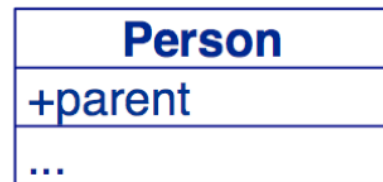
Multiplikatoren

- Multiplikatoren einer Assoziation bestimmen, mit wie vielen Entitäten man assoziiert wird
 - Beispiele:

0..1	Zero or one entity
1	Exactly one entity
*	Any number of entities
1..*	One or more entities
1..n	One to n entities
	<i>And so on ...</i>

Assoziationen und Attribute

- Assoziationen können als Attribute dargestellt werden, müssen aber nicht (abhängig von der Übersicht im Diagramm)



Aggregation und Komposition

Aggregation ist durch eine *Raute* gekennzeichnet und weist auf eine „*part-whole*“ *Abhängigkeit* hin:

Eine *durchsichtige Raute* bezeichnet eine *Referenz*; eine *gefüllte Raute* eine *Implementierung* (d.h., Besitzer).

Aggregation: parts may be shared.

Composition: one part belongs to one whole.

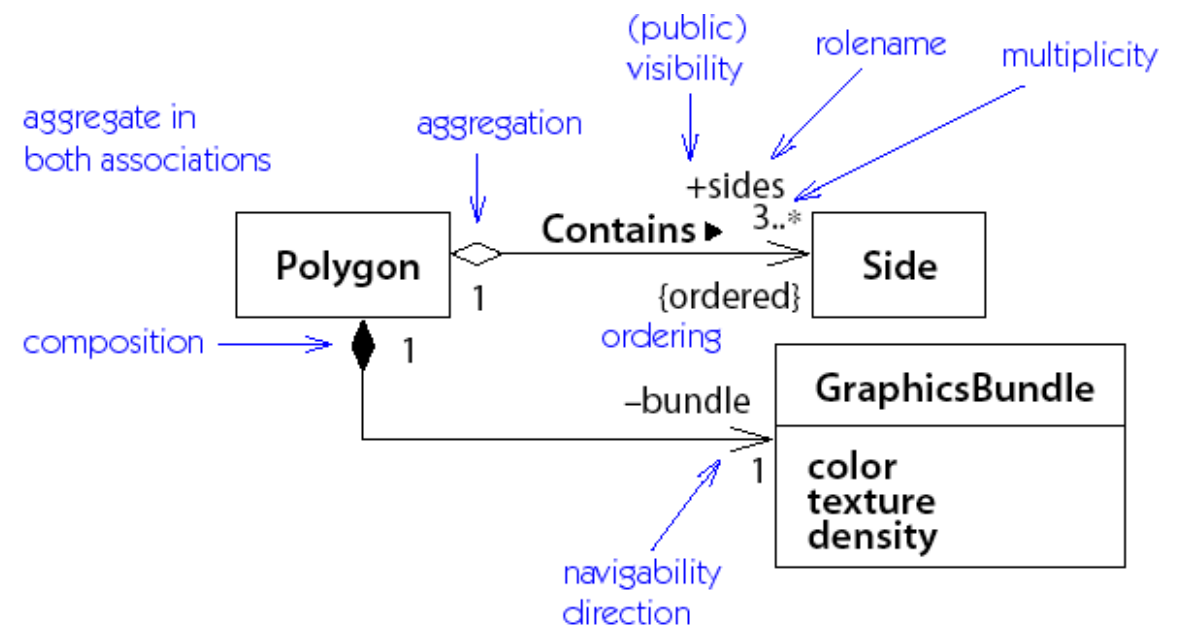


Figure 13-29. Various adornments on association ends

Aggregation vs. Komposition 2

Komposition:

- Klasse A „besitzt“ Klasse B: B hat keine Bedeutung ohne A

Aggregation:

- Klasse A „benutzt“ Klasse B: B existiert unabhängig von A

Beispiel:

- Eine Firma ist eine Aggregation ihrer Mitarbeiter. Aber Ihre Kunden-Accounts sind eine Komposition. Falls die Firma nicht mehr existiert, existieren noch die Mitarbeiter, aber die Kundenaccounts haben dann keine Bedeutung mehr.

Assoziierungsklassen

Eine Assoziierung kann eine Instanz einer Assoziierungsklasse sein:

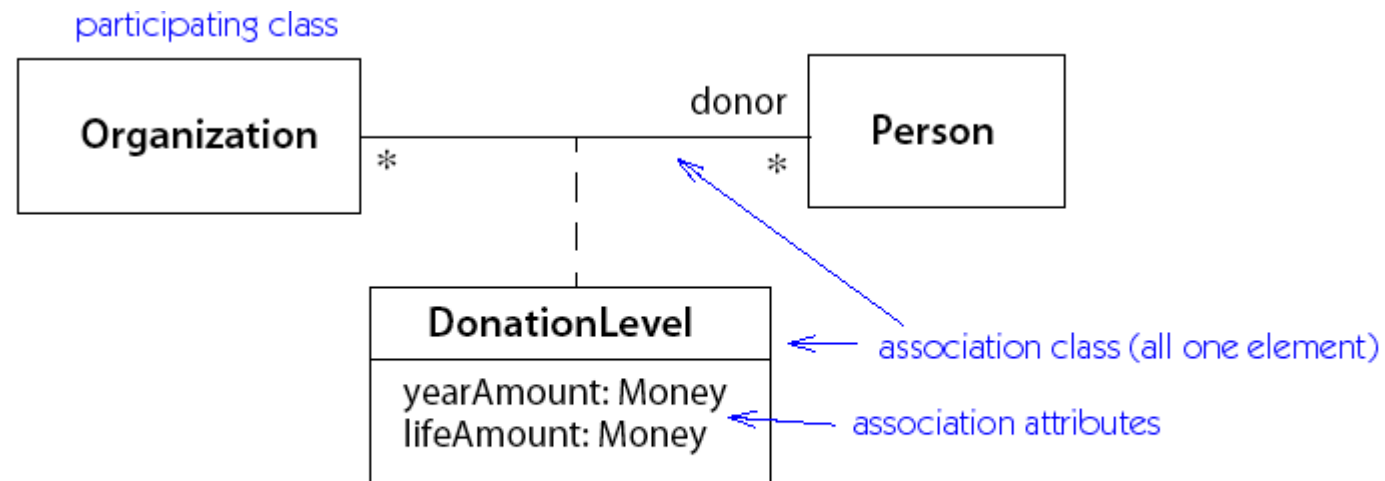


Figure 4-3. Association class

In den meisten Fällen speichert eine Assoziierungsklasse lediglich Attributwerte, so dass der Name oft weggelassen werden kann.

Code: Assoziation, Aggregation, Komposition

Assoziation:

```
public class Zoo
{
    void Animal(Animal an) {...}
};
```

Verwendung der Klasse Bar
in der Klasse Foo.

Aggregation:

```
public class oo
{
    private Bar bar;
    Foo(Bar bar) {
        this.bar = bar;
    }
};
```

Verwendung der Klasse Bar
in der Klasse Foo.

Komposition:

```
public class Foo
{
    private Bar bar = new Bar();
};
```

Verwendung der Klasse Bar
in der Klasse Foo.

Aufgabe

Modellieren Sie ein Buch, welches aus einem Inhaltsverzeichnis, einem Index sowie mehreren Kapiteln besteht, die wiederum mehrere Abschnitte haben und diese wiederum mehrere Absätze.



Vererbung (Inheritance)

Generalisierung / Vererbung

Eine *Unterklasse* (*Subklasse* / *Kindklasse*) spezialisiert ihre Superklasse (Elternklasse / Oberklasse):

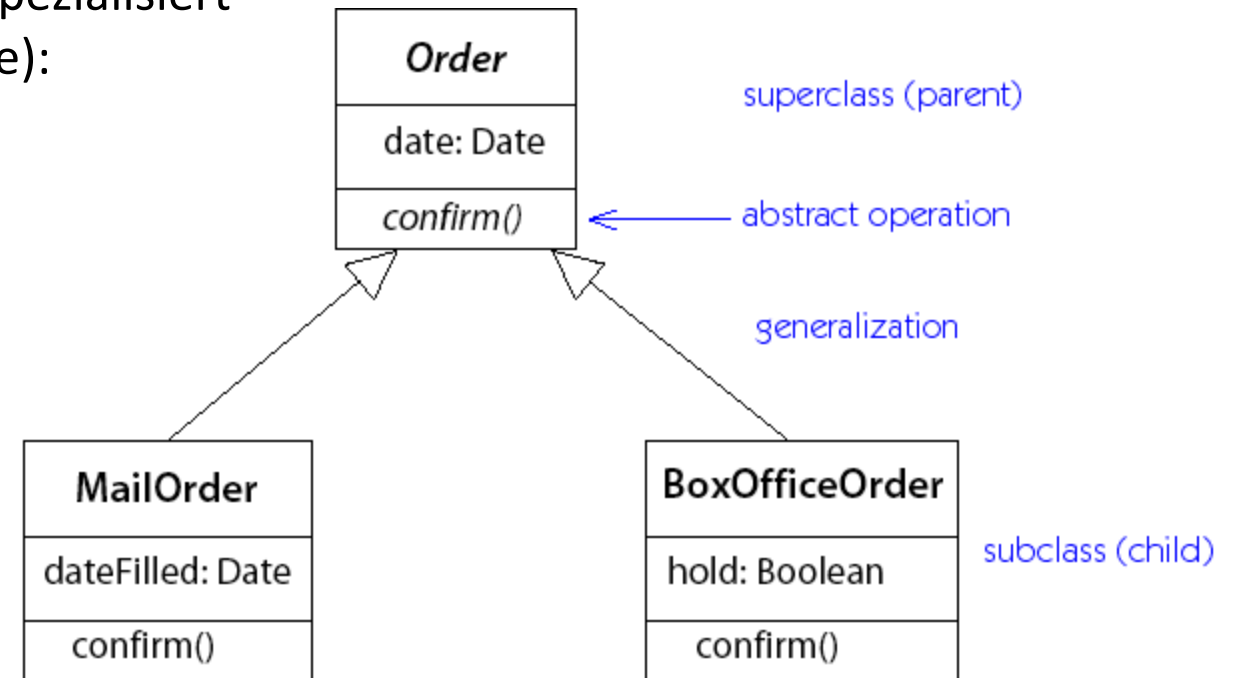


Figure 4-7. Generalization notation

Wofür ist Vererbung gut?

- Neue Software baut oft auf alter Software durch *Nachahmung*, *Verfeinerung* oder *Kombination* auf.
- Genauso: Klassen können basierend auf existierenden Klassen *erweitert*, *spezialisiert* oder *kombiniert* werden

Generalisierung Beschreibt ...

Konzeptuelle Hierarchie:

- Konzeptuell verwandte Klassen können in *Spezialisierung* hierarchien organisiert werden
 - people, employees, managers
 - geometric objects ...

Polymorphie:

- Objekte von unterschiedlichen, aber verwandten Klassen können *uniform* (gleich) durch einen Benutzer benutzt werden
 - array of geometric objects

Software Wiederverwendung:

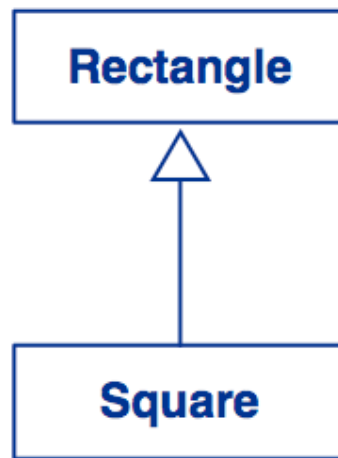
- Verwandte Klassen können Interfaces, Datenstrukturen und Verhalten *teilen*
 - geometric objects ...

Aufgabe

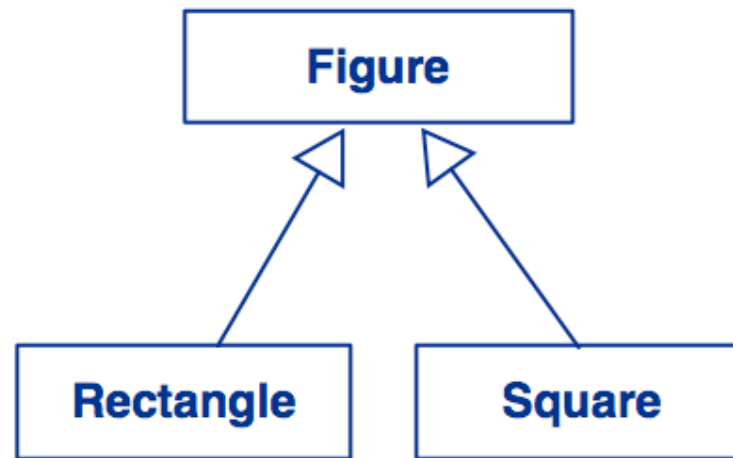
Modellieren Sie ein Auto, welches aus Einzelteilen besteht. Verwenden Sie möglichst viele Klassen, aber achten Sie auf ein gutes Design!



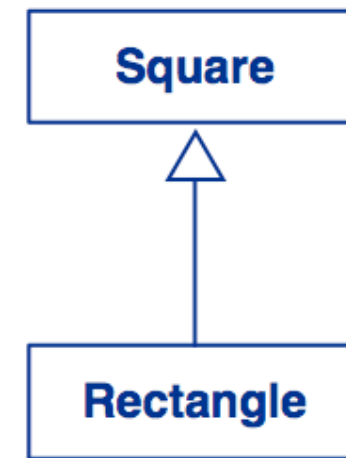
Unterschiedliche Arten von Vererbung



Is-a



Polymorphism



Reuse

Use-Case Diagramme

Use-Case Diagramme

Ein use case ist eine *generische Beschreibung einer gesamten Transaktion*, welche mehrere Akteure involviert.

Ein use-case Diagramm präsentiert eine *Menge von use cases* (Ellipsen) und deren externe Akteure, die mit dem System interagieren.

Abhängigkeiten und *Assoziationen* zwischen use cases können dargestellt werden.

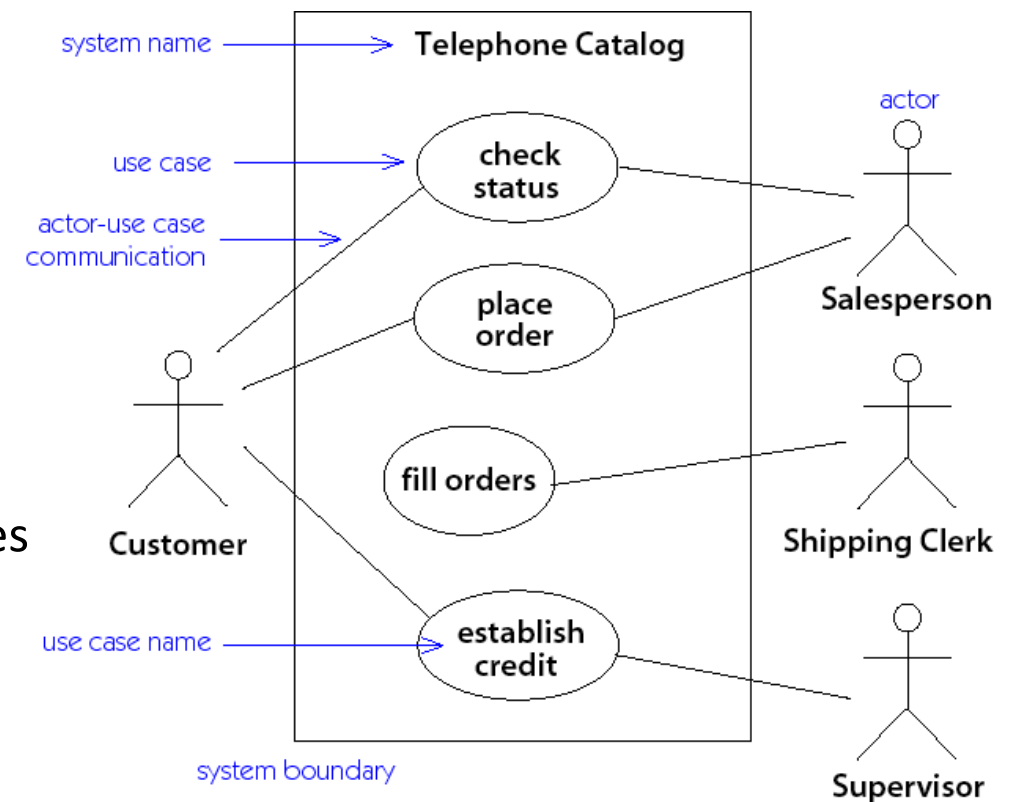
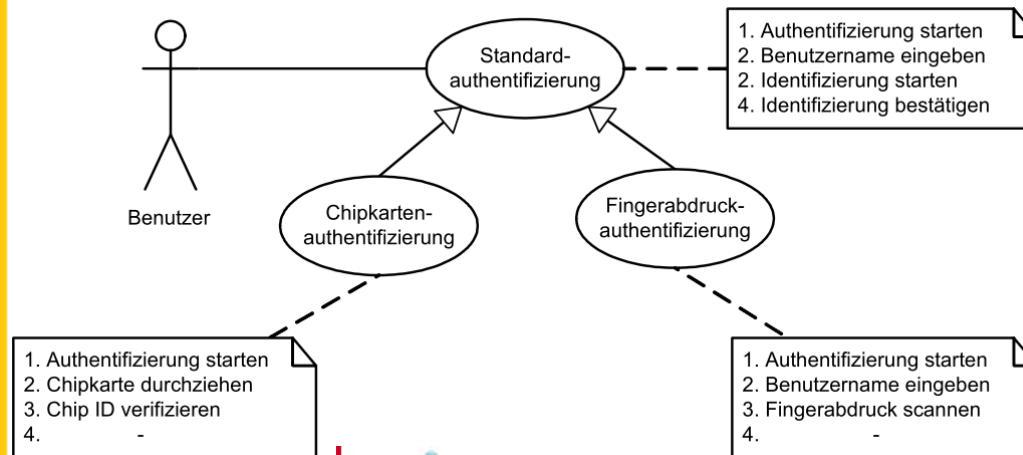


Figure 5-1. Use case diagram

Verwendung: Use-Case Diagramm

“A use case is a *snapshot of one aspect* of your system. The sum of all use cases is *the external picture* of your system ...”

Generalisierung und Kommentare

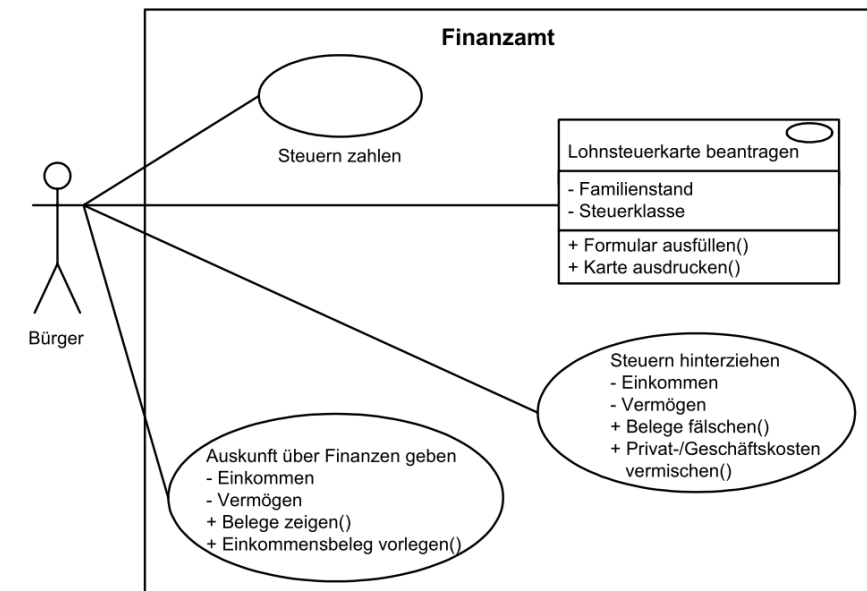


Quelle:



—UML Distilled

Auch Attribute und Operationen möglich



Sequenz-Diagramme

Szenarien

Ein Szenario ist eine *Instanz* von einem use case, das ein *typisches Beispiel* einer Ausführung zeigt.

Szenarien können durch UML repräsentiert werden, entweder durch *Sequenzdiagramme* oder *Kollaborationsdiagramme*

*Wichti: Ein Szenario beschreibt nur **ein** Beispiel eines use cases, so dass Besonderheiten oder Bedingungen nicht ausgedrückt werden können!*

Sequenzdiagramme

Ein Sequenzdiagramm beschreibt ein Szenario durch das Zeigen von Interaktionen zwischen einer Menge von Objekten in einer *zeitlichen Abfolge*.

Objekte (keine Klassen!) werden als *vertikale Balken* gezeichnet. *Events* oder Nachrichtensendungen werden als horizontale (oder schräge) *Pfeile* vom Sender zum Empfänger gezeichnet.

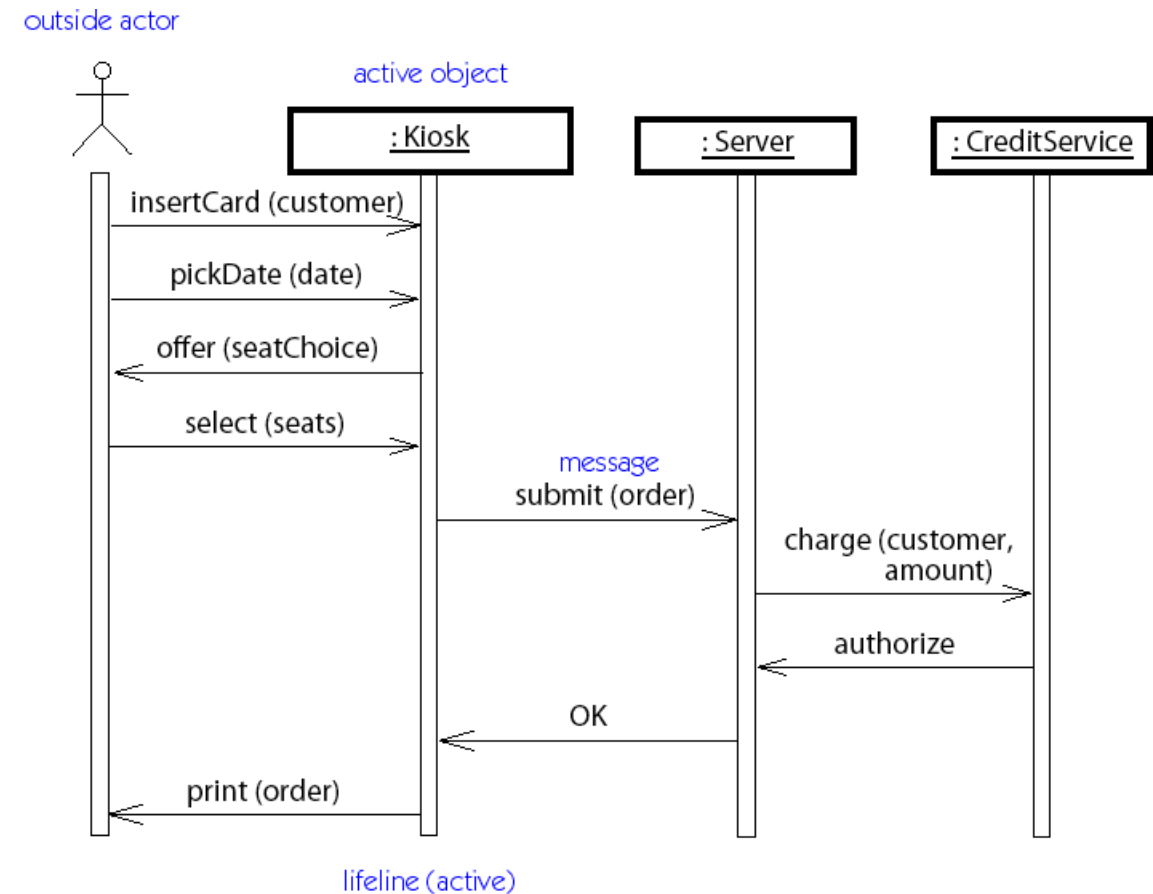
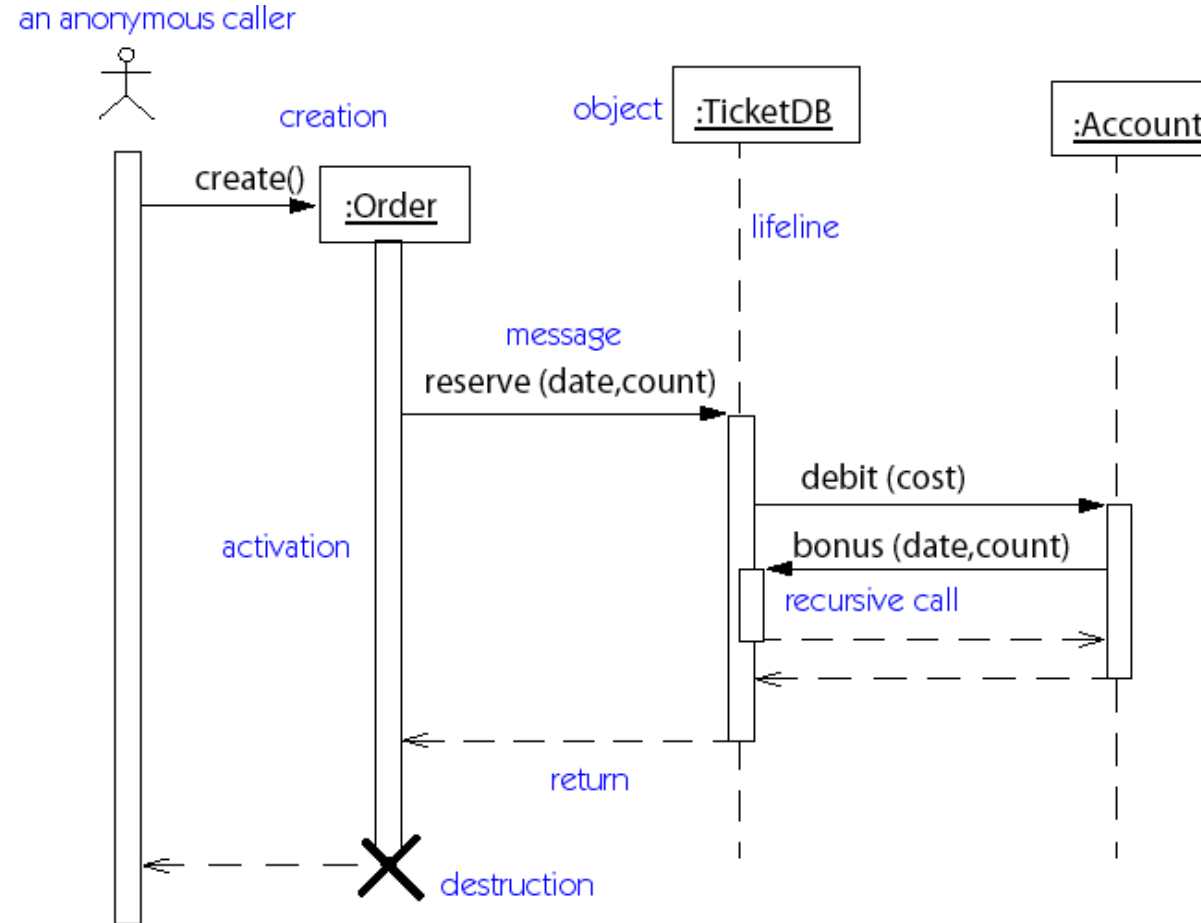


Figure 8-1. Sequence diagram

Szenario: Sitzplatz im Kino reservieren

Aktivierungen



Return-Statements sind optional. Abhängig vom Detailgrad evtl. wichtig.

Figure 8-2. Sequence diagram with activations

Asynchronität und Bedingungen

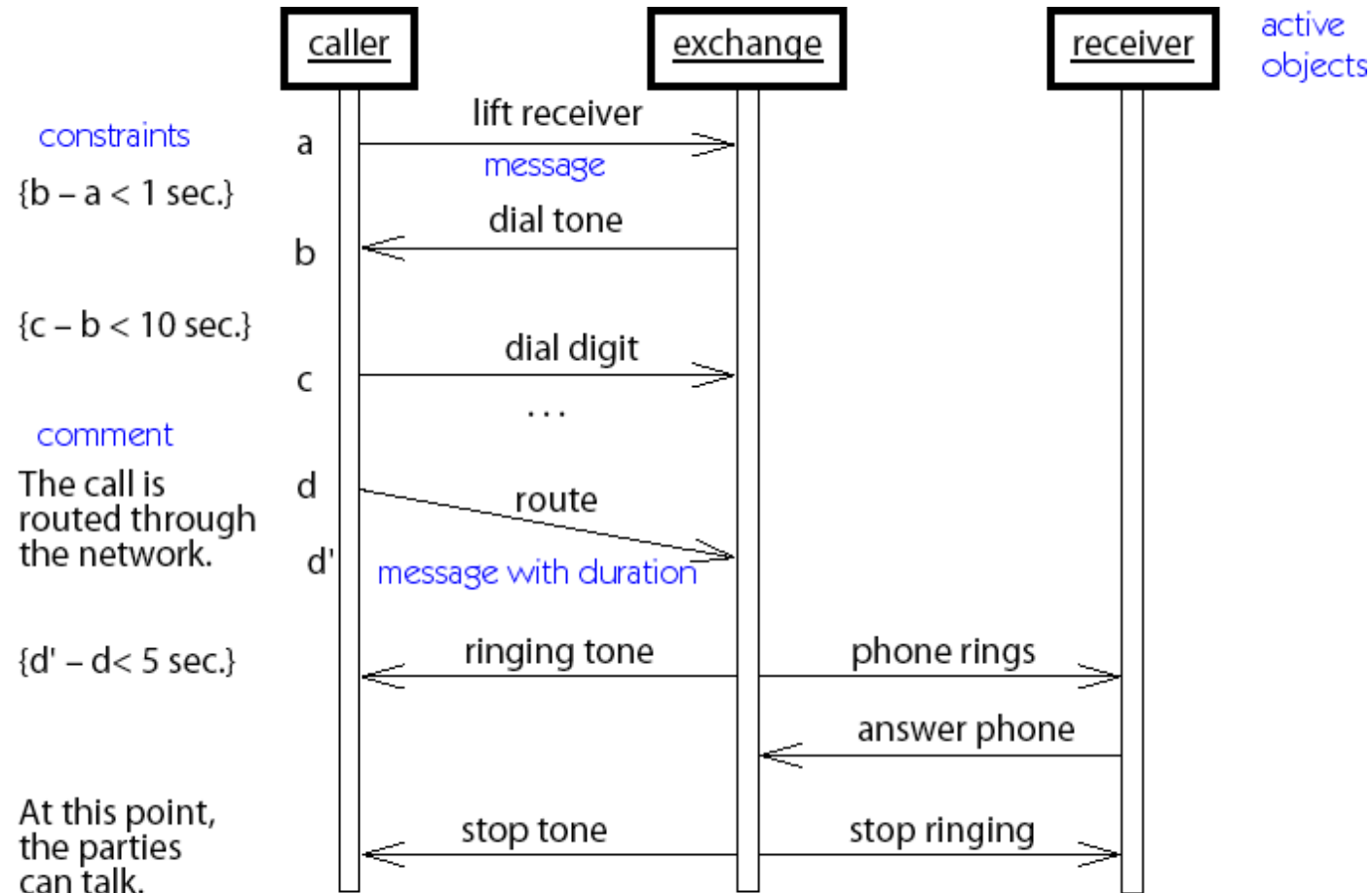
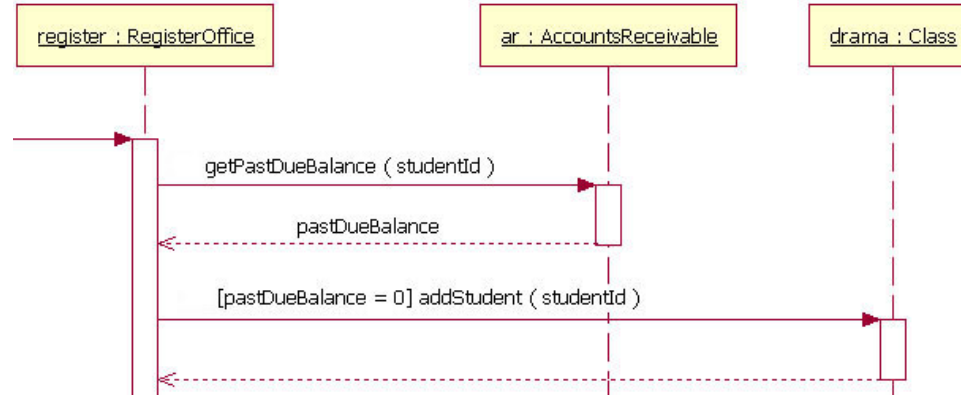


Figure 13-161. Sequence diagram with asynchronous control

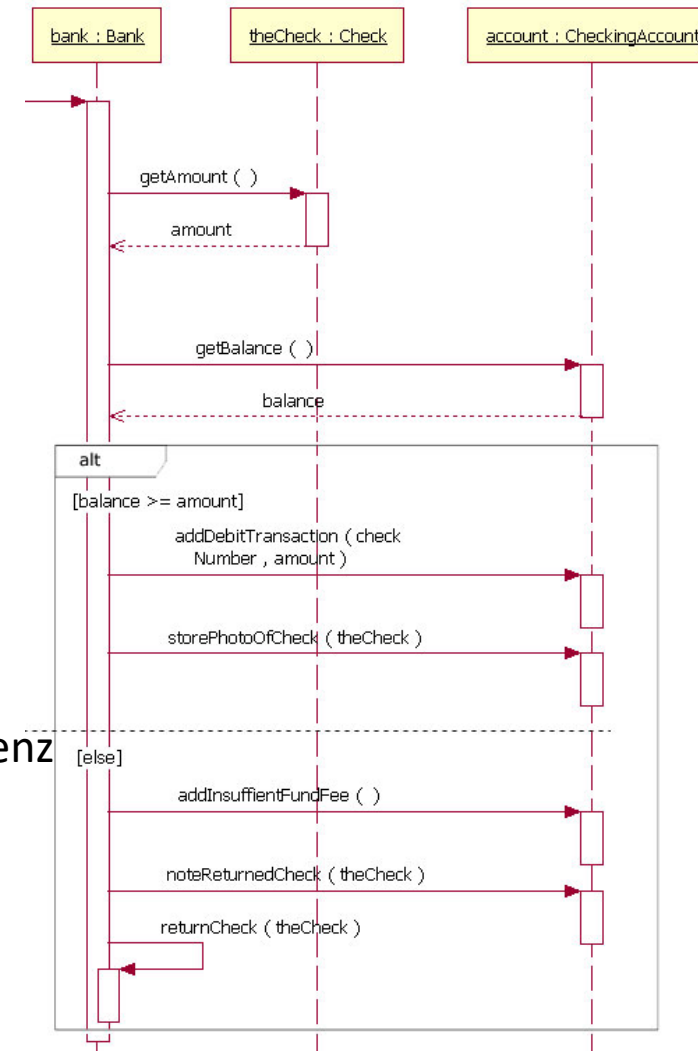
Alternativen und Guards

Guard: Bedingung muss erfüllt sein, bevor eine Nachricht verschickt wird.



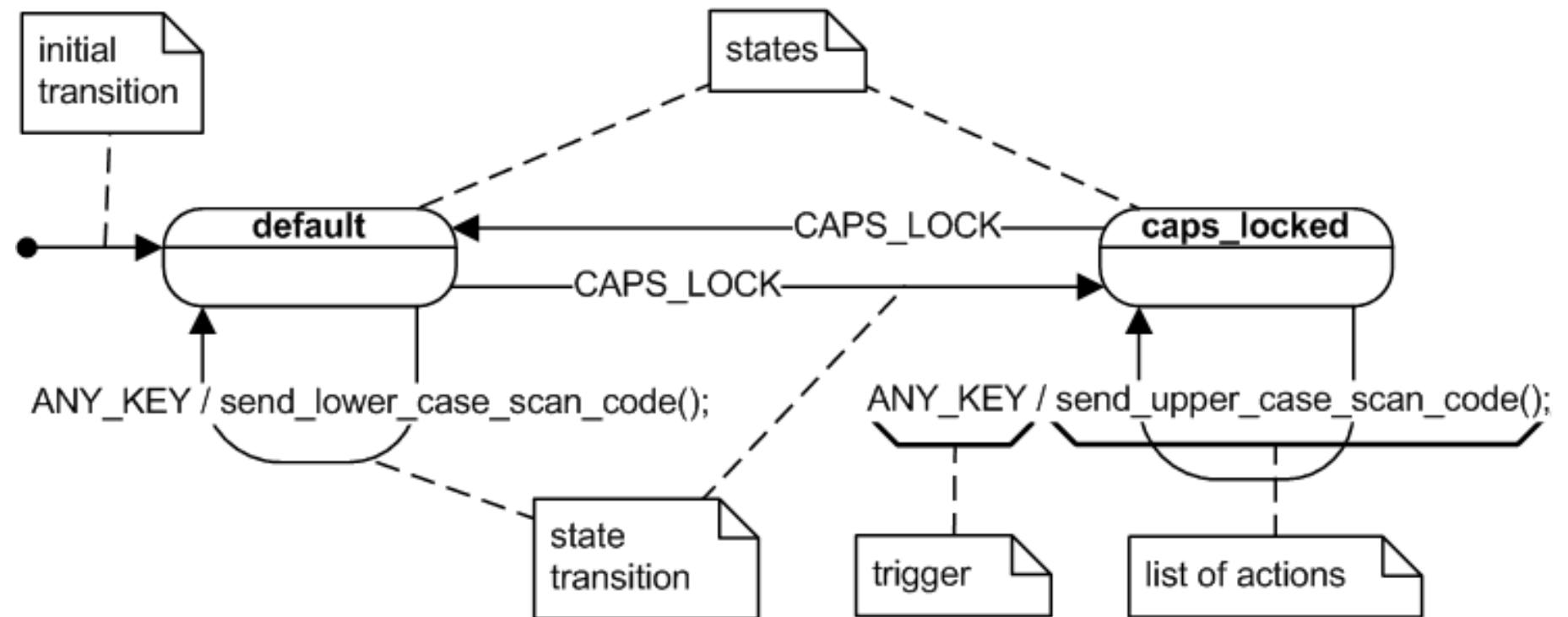
Syntax: [Boolean Test]

Alternative Sequenz



Statechart (Zustands-)Diagramme

Beispiel



Definition I

Ein Zustandsdiagramm beschreibt die *zeitliche Evolution* eines Objektes von einer gegebenen Klasse in Abhängigkeit von *Interaktionen* mit anderen Objekten innerhalb und außerhalb des Systems.

Ein Event ist eine one-way (asynchrone) Kommunikation von einem Objekt zu einem Anderen:

- *atomar* (nicht unterbrechbar)
- Beinhaltet *Hardware* und Realwelt-Objekte, z.B., Nachrichteneingang, input Ereignis, Zeitüberschreitung, ...
- Notation: ***eventName(parameter: type, ...)***
- Kann das Objekt zu einer *Transition* zwischen Zuständen veranlassen

Definition II

Ein Zustand ist eine Zeitperiode, bei der ein Objekt auf ein Ereignis *wartet*:

- Dargestellt als *abgerundete Box* mit (bis zu) drei Sektionen:
 - *name* — optional
 - *state variables* — name: type = value (valid only for that state)
 - *triggered operations* — internal transitions and ongoing operations
- Kann *geschachtelt* sein

Statusbox mit Regionen

Das *Eingangs-Event* tritt auf, wann immer eine Transition zu diesem Zustand getätigt wird.

Das *Ausgangs-Event* tritt auf, wenn eine Transition aus diesem Zustand hinaus führt.

Die *Hilfs-* und *Zeichenereignisse* lösen interne Transitionen aus ohne den Zustand zu ändern, so dass keine Eingangs- oder Ausgangsoperation durchgeführt wird.

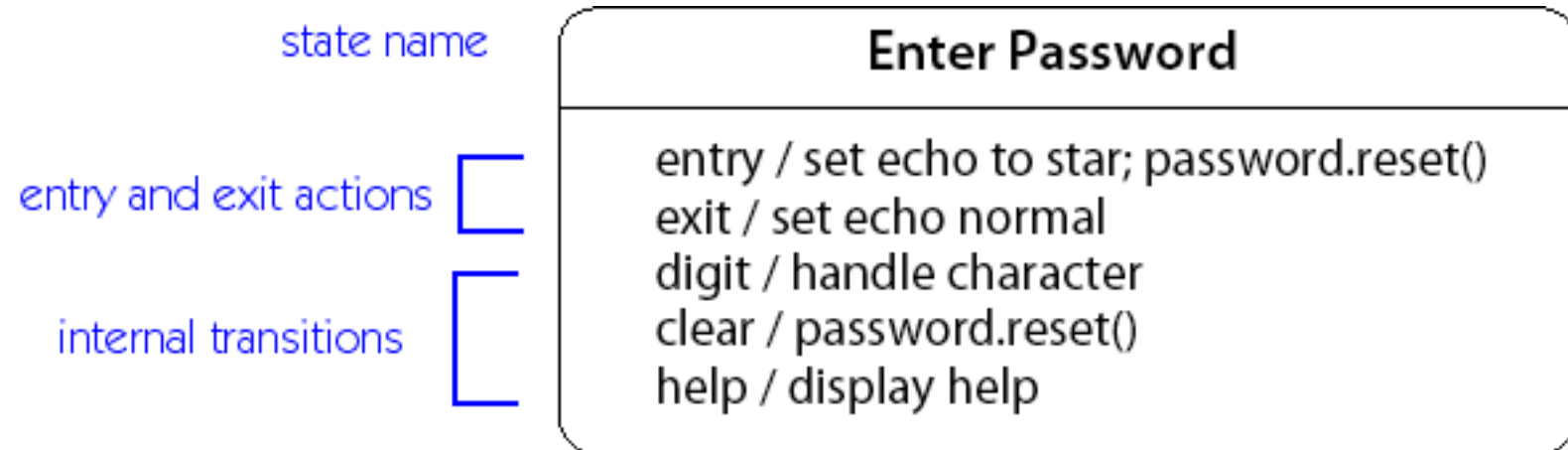


Figure 6-4. *Internal transitions, and entry and exit actions*

Transitionen

Eine Transition ist eine *Antwort auf ein externes Ereignis*, welches das Objekt in einem *bestimmten Zustand* erhalten hat

- Kann zur *Ausführung* einer Operation und zum Wechsel des Zustands des Objekts führen
- Kann ein Ereignis zu einem anderen externen Objekten *senden*
- Transitionssyntax (jeder Teil ist optional):
 event(arguments) [condition]
 / target.sendEvent operation(arguments)
- *Externe Transitionen* markieren Kreisbögen zwischen Zuständen
- *Interne Transitionen* sind Teil der ausgelösten Operationen eines Zustandes

Operationen und Aktivitäten

Eine Operation ist eine *atomare Aktion*, angestoßen von einer Transition

- *Eingangs- und Ausgangsoperationen* können mit Zuständen assoziiert werden

Eine Aktivität ist eine *laufende Operation*, die läuft, während ein Objekt in einem bestimmten Zustand ist

- Modelliert als “interne Transitionen” markiert mit dem pseudo-event **do**

Schachtelung: Geschachtelte Zustandsdiagramme

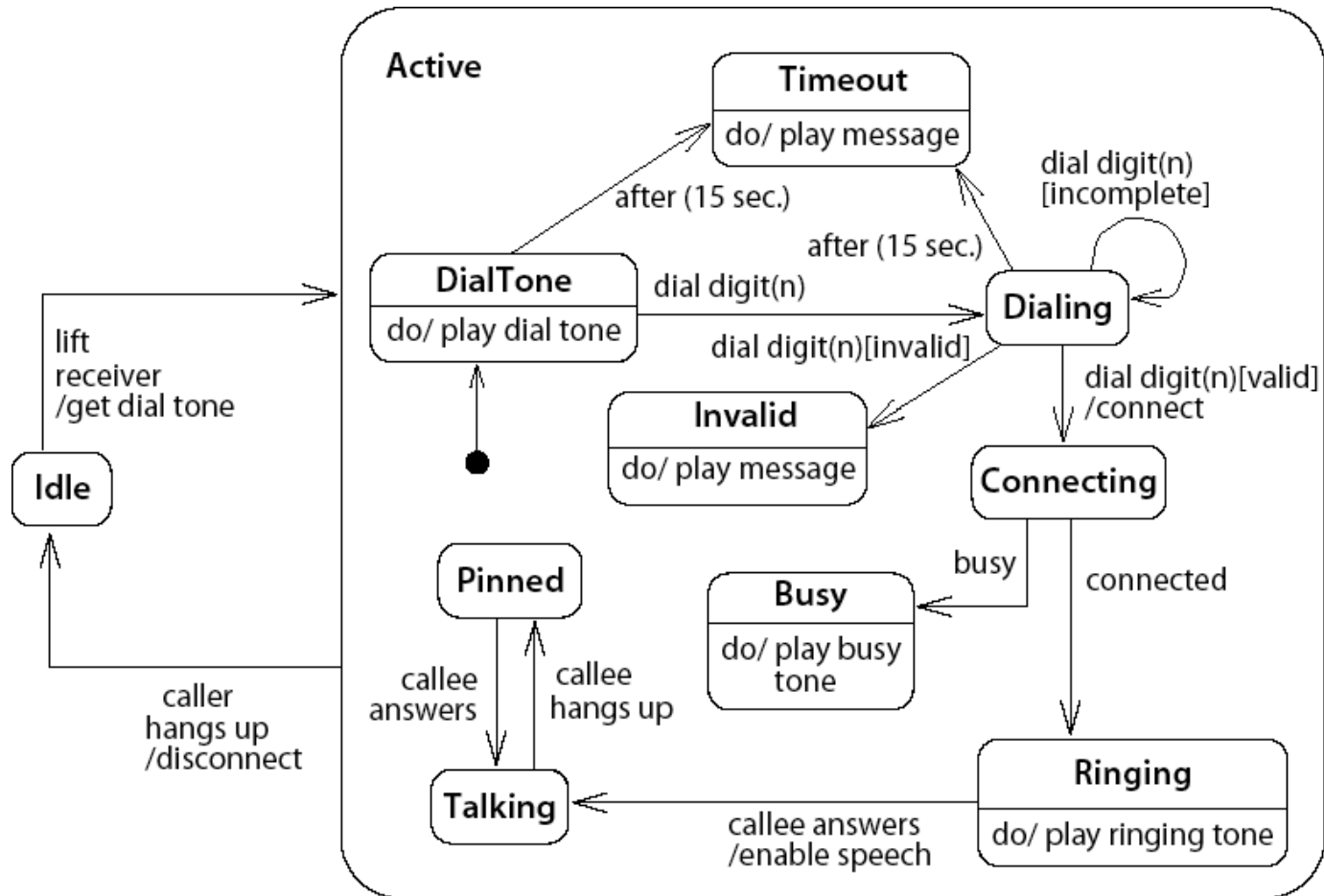


Figure 13-169. State diagram

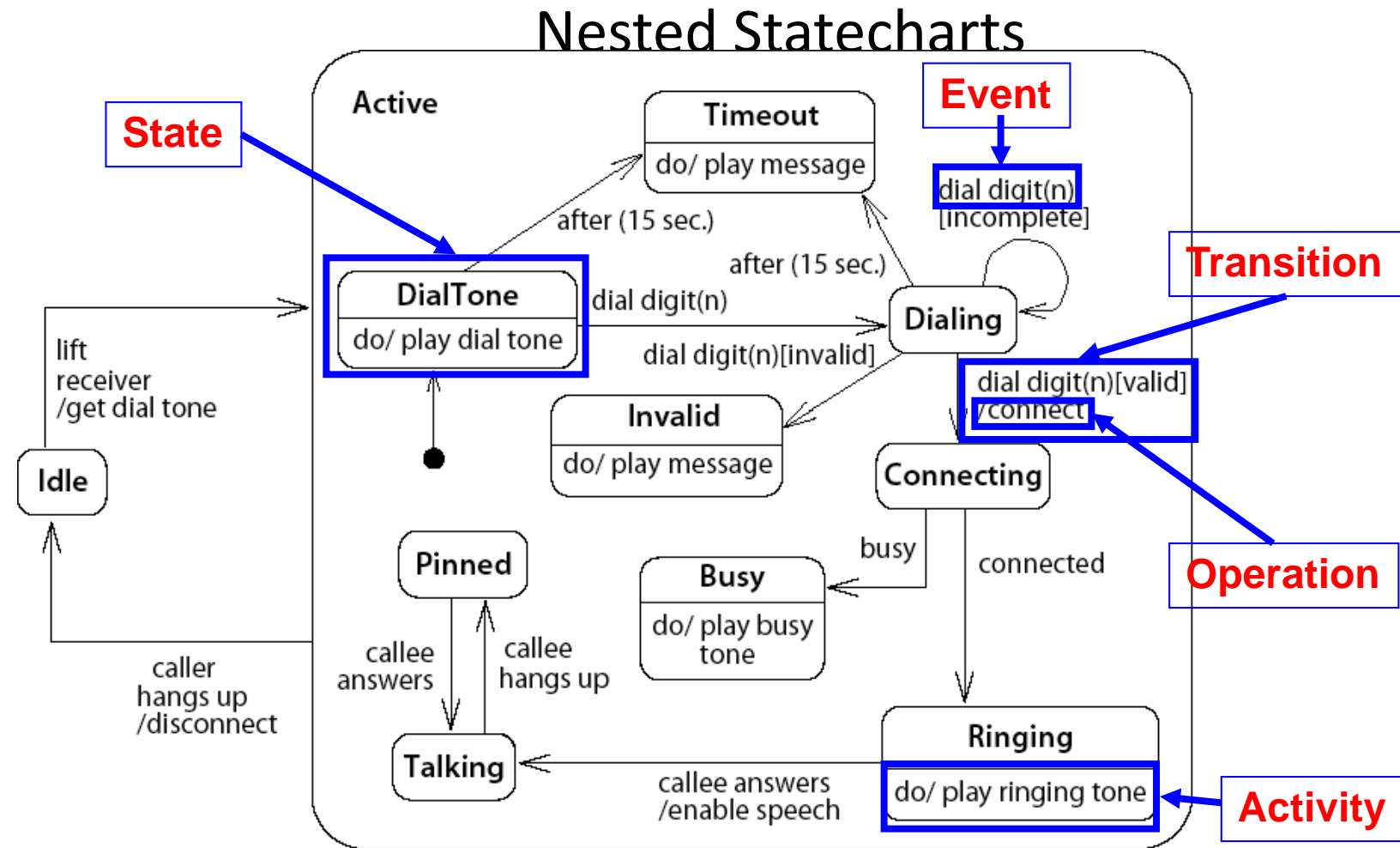


Figure 13-169. State diagram

Aufgabe

- Modellieren Sie ein Flugzeug-Objekt, welches den Zustand des Flugzeuges bzgl. der Platzreservierung wiedergibt. Definieren Sie geeignete Zustandsübergänge und evtl. Bedingungen dafür.



UML Benutzung: Perspektiven

Perspektiven

Drei Perspektiven beim Erstellen von UML Diagrammen:

1. *Konzeptionell*

- Repräsentieren Domänenkonzepte
 - Ignoriere Software Belange

2. *Spezifikation*

- Fokus auf sichtbare Interfaces und Verhalten
 - Ignoriere interne Implementierung

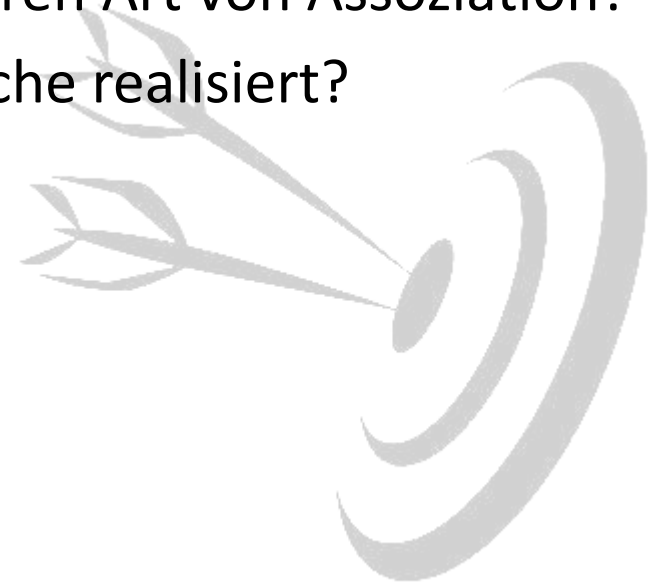
3. *Implementierung*

- Dokumentiere Implementierungsentscheidungen
 - Häufigste, aber am wenigsten nützlichste Perspektive (!)

—*UML Distilled*

Was Sie mitgenommen haben sollten I

- Wie kann ich Klassen, Objekte und Assoziationen repräsentieren?
- Wie kann ich die Sichtbarkeit von Attributen und Operationen bestimmen?
- Warum ist Vererbung in der Analyse und im Design nützlich?
- Was unterscheidet Aggregation von irgendeiner anderen Art von Assoziation?
- Wie werden Assoziationen in einer Programmiersprache realisiert?



Was Sie mitgenommen haben sollten II

- Was ist der Zweck von use case Diagrammen?
- Warum beschreiben Szenarien Objekte und nicht Klassen?
- Wie können zeitliche Bedingungen in Szenarien beschrieben werden?
- Wie spezifiziert und interpretiert man Nachrichten-Labels in einem Szenario?
- Wie benutzt man genestete Zustandsdiagramme, um Objektverhalten zu modellieren?
- Was ist der Unterschied zwischen “externen” und “internen” Transitionen?
- Leiten Sie aus einer Anforderungsbeschreibung/CRC-Karten ein Klassendiagramm, ein Use-Case-Diagramm, ein Sequenz-Diagramm und ein Zustandsdiagramm ab

Literatur

- *The Unified Modeling Language Reference Manual*, James Rumbaugh, Ivar Jacobson and Grady Booch, Addison Wesley, 1999.
- *UML Distilled*, Martin Fowler, Kendall Scott, Addison-Wesley, Second Edition, 2000.