Course "Softwaretechnik"
Book Chapter 2

Modeling with UML

Lutz Prechelt, Bernd Bruegge, Allen H. Dutoit
Freie Universität Berlin, Institut für Informatik

- Modeling, models and UML
- Static view:
  - Class diagrams
- Dynamic view:
  - Sequence diagrams
  - State machine diagrams
  - Activity diagrams
- Other UML diagram types
  - component d., collaboration use d., deployment d., communication d., interaction overview d.
- UML Metamodel, Profiles
- Some notation details
  - Classes, associations, interfaces, states

Lernziele

- Einen groben Überblick über Grundideen und die wichtigsten Diagrammarten der UML gewinnen.
- Erkennen: UML kann informell oder präzise eingesetzt werden.

Taxonomie "Die Welt der Softwaretechnik"

Welt der Problemstellungen:

- Produkt (Komplexitätsprob.)
  - Anforderungen (Problemraum)
  - Entwurf (Lösungsraum)
- Prozess (psycho-soziale P.)
  - Kognitive Beschränkungen
  - Mängel der Urteilskraft
  - Kommunikation, Koordination
  - Gruppendynamik
  - Verborgene Ziele
  - Fehler

Welt der Lösungsansätze:

- Technische Ansätze ("hart")
  - **Abstraktion**
  - Wiederverwendung
  - Automatisierung
- Methodische Ansätze ("weich")
  - Anforderungsermittlung
  - Entwurf
  - Qualitätssicherung
  - Projektmanagement

What is modeling?

- Modeling consists of building an abstraction of reality
  - Models ignore irrelevant details (i.e., they simplify)
  - and only represent the relevant details
- What is *relevant* or *irrelevant* depends on the purpose of the model. We typically want to
  - draw complicated conclusions about reality with simple steps in the model in order to
  - get insights into the past or presence or make predictions

- Reality R:
  - Real things, people, etc.
  - Processes happening during some time
  - Relationships between things etc.
- Model M:
  - Abstractions of any or all of the above

What is a "good" model?

- In a good model, relationships which are valid in reality R are also valid in model M (if they exist in M at all).
  - I : Mapping of reality R to the model M (abstraction)
  - f_M: relationship between abstractions in M
  - f_R: equivalent relationship between real things in R

- In a good model, the following diagram is commutative:

$$\begin{tikzcd}
\mathbf{m_a} \arrow[r, "f_M"] & \mathbf{m_b} \\
\mathbf{r_a} \arrow[u, "\mathbf{I}"] \arrow[r, "f_R"'] & \mathbf{r_b} \arrow[u, "\mathbf{I}"']
\end{tikzcd}$$

Models of models of models...

- **We can think of a model as reality and can build another model from it (with additional abstractions)**
  - The development of software systems is a transformation of models:
    Requirements elicitation (→ req's document M_1),
    Requirements analysis (→ analysis model M_2),
    Design (→ design model M_3),
    Implementation (→ source code M_4)
    ....

"Model-Driven Development"
uses this idea for its engineering approach.

$$\begin{tikzcd}
\mathbf{M_2} \arrow[r, "f_{M_2}"] & \mathbf{M_2} \\
\mathbf{M_1} \arrow[u, "\text{Analysis}"] \arrow[r, "f_{M_1}"] & \mathbf{M_1} \arrow[u, "I_2"'] \\
\mathbf{R} \arrow[u, "\text{Requirements Elicitation}"] \arrow[r, "f_R"'] & \mathbf{R} \arrow[u, "I_1"']
\end{tikzcd}$$

Systems, models and views

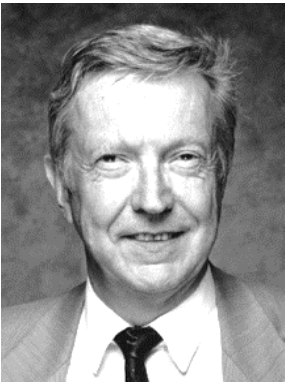
- A ***model*** is an abstraction describing relevant aspects of a system
- A ***view*** ("Sicht") depicts selected aspects of a model
  - Any view is a model itself
  - Calling a model a view makes clear it is part of a larger model
  - Complex models are often shown as many views only
    - never as a whole
- A ***notation*** is a set of rules for depicting models
  - graphically or textually

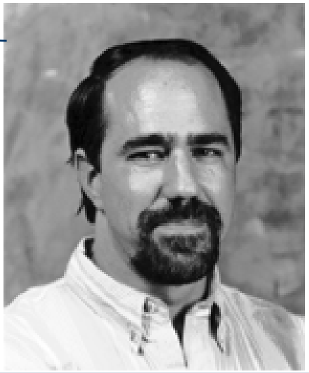
- Example:
  - System: Aircraft
  - Models: Flight simulator, scale model, construction plan, ...
  - Views: All blueprints (e.g. electrical wiring, fuel system)

What is UML?

UML (**Unified Modeling Language**):

- The most-used standard for software modeling
  - For both requirements modeling (application domain)
  - and software modeling (solution domain)
- A set of related graphical notations
  - Quite complex, we will use a subset only
- Resulted from the convergence of notations from three leading object-oriented methods:
  - OMT (James Rumbaugh)
  - OOSE (Ivar Jacobson)
  - Booch method (Grady Booch)
  - The authors are known as *"The Three Amigos"*
- Supported by CASE tools
  - http://de.wikipedia.org/wiki/UML-Werkzeug





Common UML diagram types

- Use Case diagrams (functional view)
  - Catalog scenarios that describe the functional behavior of the system as seen by the user [see lecture "use cases"]
- Class diagrams / Object diagr. (static view and examples)
  - Describe the static structure of the system: Classes, attributes, object associations (class diagram) or
    snapshots of possible resulting configurations (object diagram)
- Sequence diagrams (dynamic view examples)
  - Describe *examples* of the dynamic behavior between objects of the system (and possibly actors)
- State machine diagrams (dynamic view)
  - Describe some aspects of the dynamic behavior of the individual object of a class by a finite state automaton
- Activity diagrams (dynamic view)
  - Model the dynamic behavior of a system, in particular the workflow (essentially a flowchart, but with concurrency)

Less common UML diagram types

Hardly covered in this course:

- Implementation diagrams
  - Component diagrams
  - Deployment diagrams
- Communication diagrams
  - Equivalent to sequence diagrams, but embedded in an object diagram (shows both static structure and dynamic interaction)
- Interaction overview diagrams
  - Related to activity diagrams, for describing control flow

There is also a non-graphical language for expressing conditions:

- Object constraint language (OCL)
  - Introduced in lecture on Object Design

UML core conventions

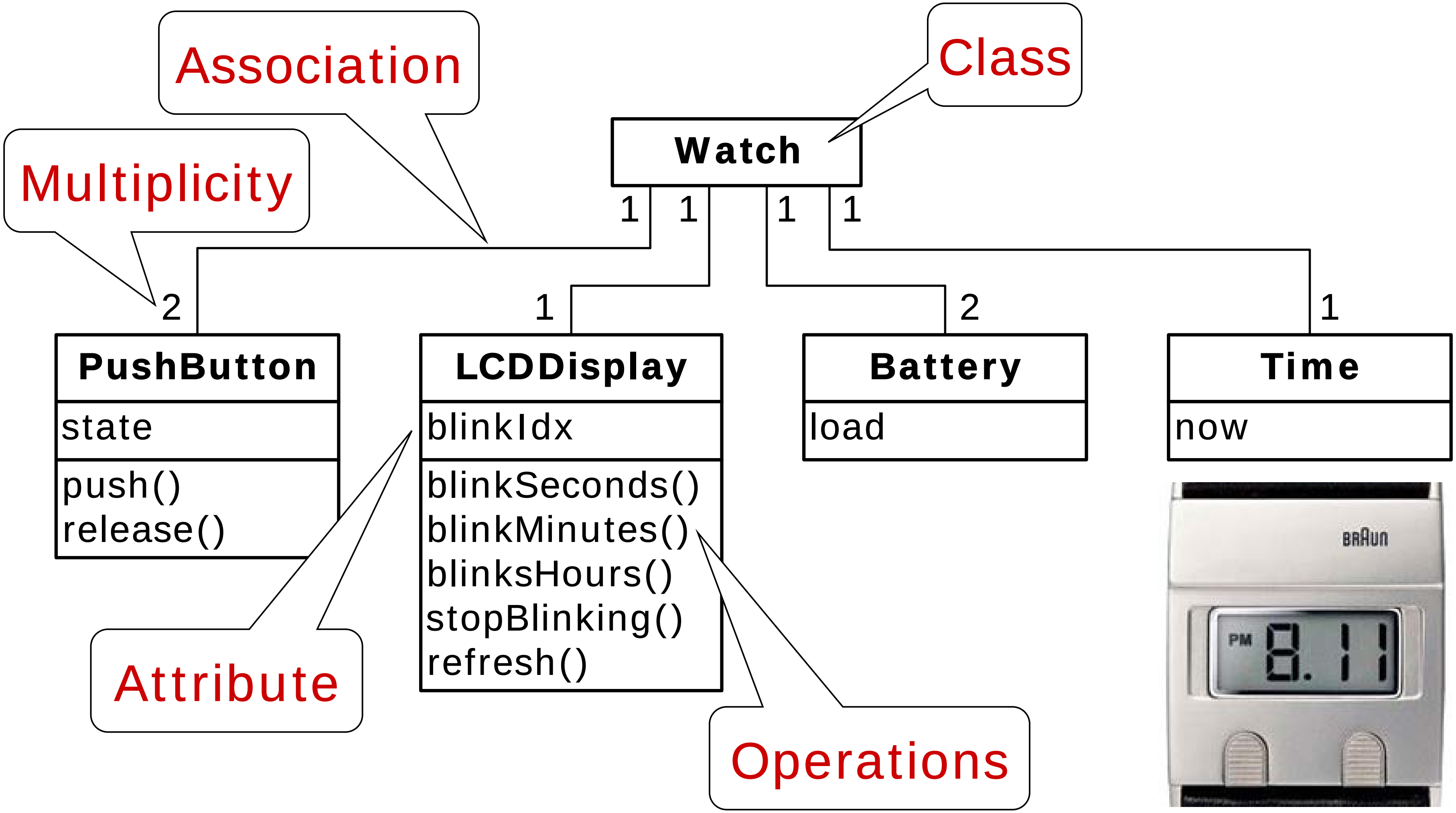
- Diagrams are mostly graphs
  - Nodes are entities
  - Edges are relationships between entities

- Rectangles are classes or instances
- Ovals are functionalities or use cases

- An instance is denoted with an underlined name
  - myWatch:SimpleWatch or with no classifier: myWatch:
  - Jane:Firefighter or with no name: :Firefighter
  - (Anonymous instance of unnamed classifier: :
    - Please don't use this ...)
- A classifier is denoted with a non-underlined name
  - SimpleWatch
  - Firefighter

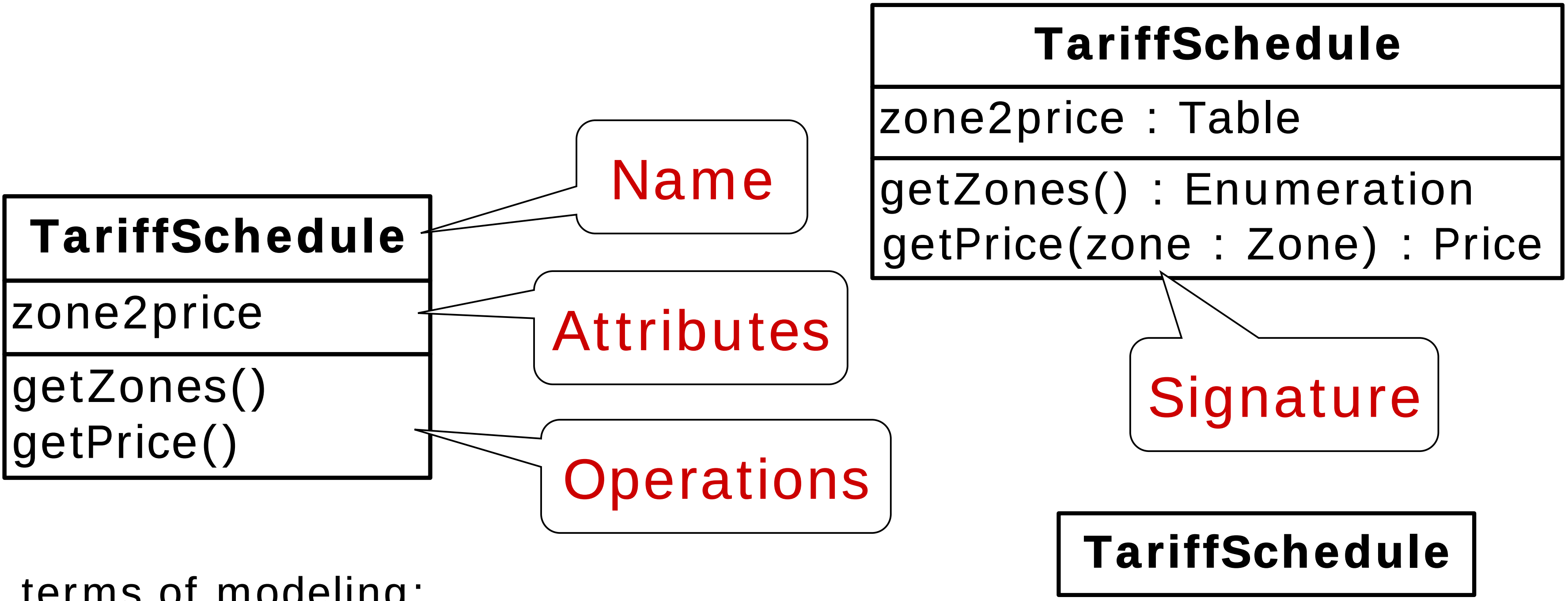
UML class diagrams

Class diagrams represent the structure of the system



[4]

Class diagrams: Classes



TariffSchedule

zone2price

getZones()
getPrice()

Name

Attributes

Operations

TariffSchedule

zone2price : Table

getZones() : Enumeration
getPrice(zone : Zone) : Price

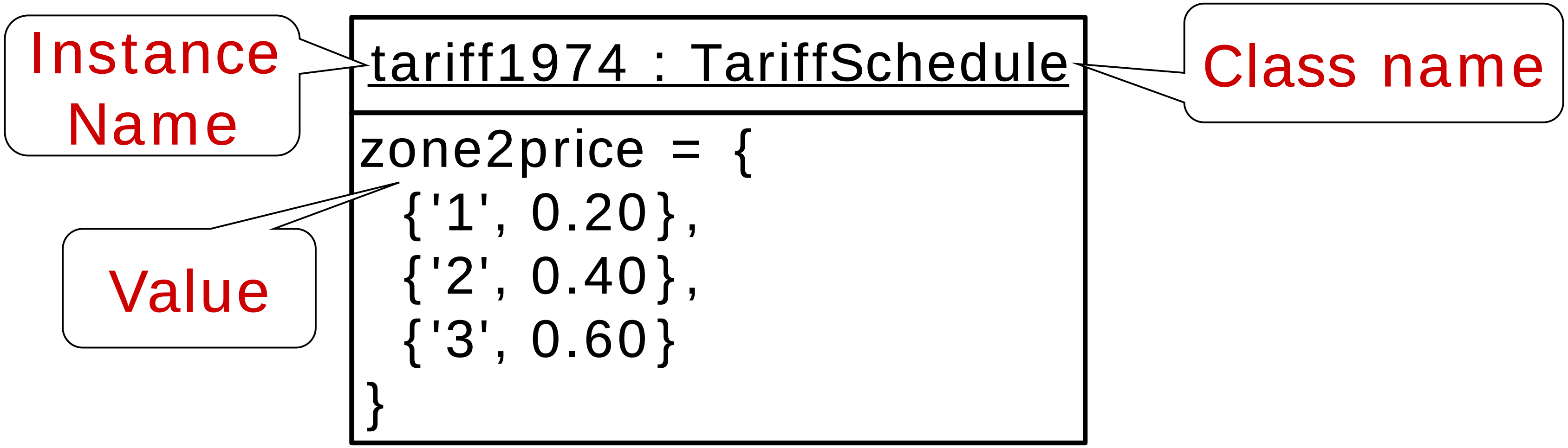
Signature

TariffSchedule

In terms of modeling:

- Problem domain: A class has a name and represents a concept
- Solution domain: A class encapsulates state (attributes) and behavior (operations)
  - Each attribute has a type
  - Each operation has a signature

Instances ("Exemplare", "Objekte")



Instance Name

tariff1974 : TariffSchedule

Class name

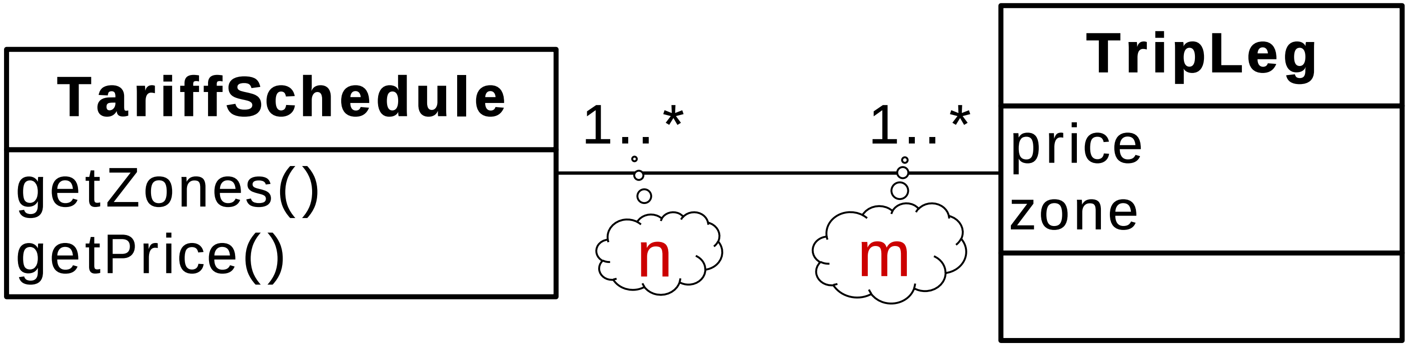
```
zone2price = {
  {'1', 0.20},
  {'2', 0.40},
  {'3', 0.60}
}
```

Value

- An instance represents a phenomenon
- The name of an instance is underlined and may indicate the class of the instance
  - May indicate instance name or class or both
- Attributes may be represented with their values

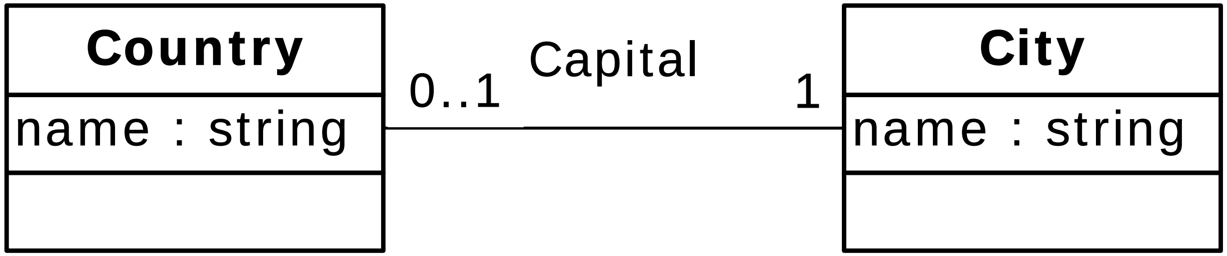
- What is the fundamental difference between a class diagram and an object diagram?

Associations



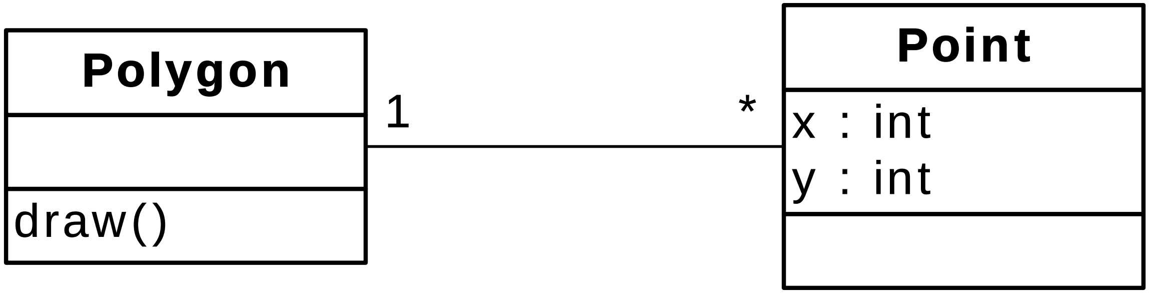
- Associations denote relationships between classes
- The multiplicity of an association end denotes how many objects the source object can legitimately reference:
  - Any one TariffSchedule object is associated with at least one TripLeg object
  - Any one TripLeg object is associated with at least one TariffSchedule object
  - **n** and **m** can be numbers ("5") or ranges (closed/open: "1..5" or "2..*")
  - A missing annotation can mean "don't know yet", "is specified elsewhere" or (by special convention) "1"
  - "*" alone means "arbitrarily many" (zero, one, or several)

1-to-1 and 1-to-many associations



One-to-one association

(Too restrictive?: Some countries have a separate seat of government)

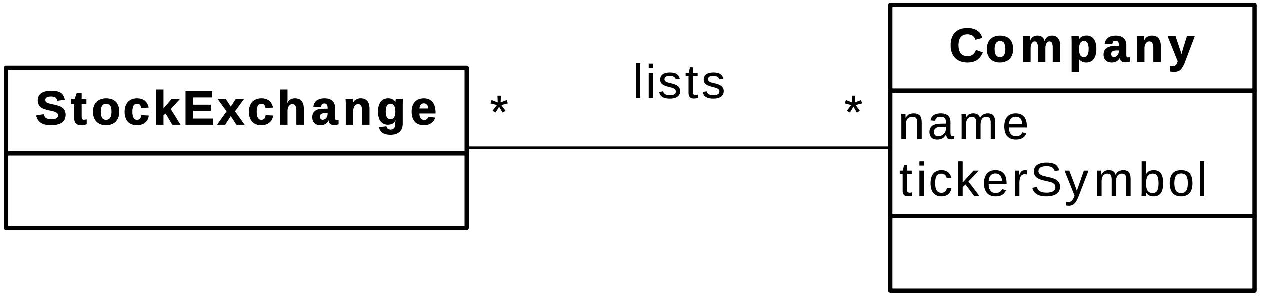


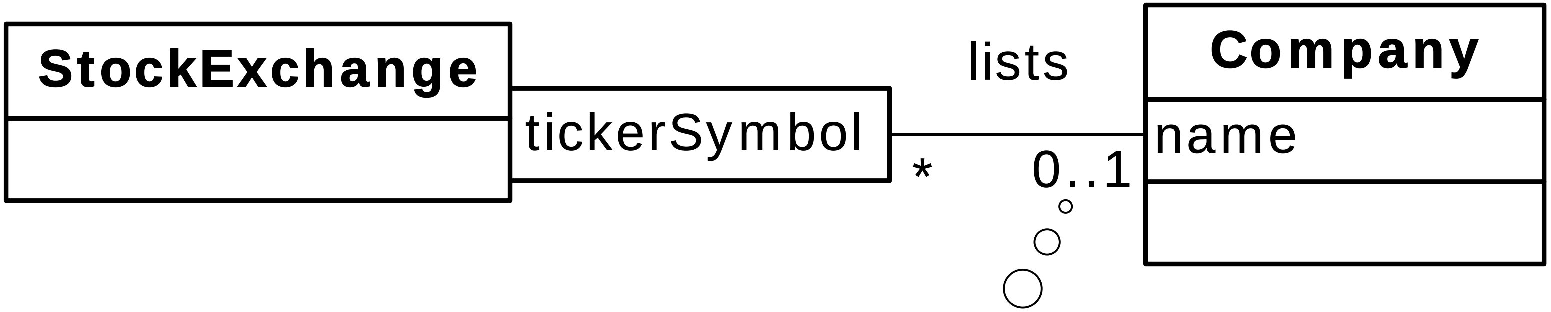
One-to-many association

(Too flexible?: Does a Polygon with 0, 1, or 2 Points really make sense?)

Many-to-many associations

Problem Statement: "A stock exchange lists many companies. Each company is uniquely identified by a ticker symbol."



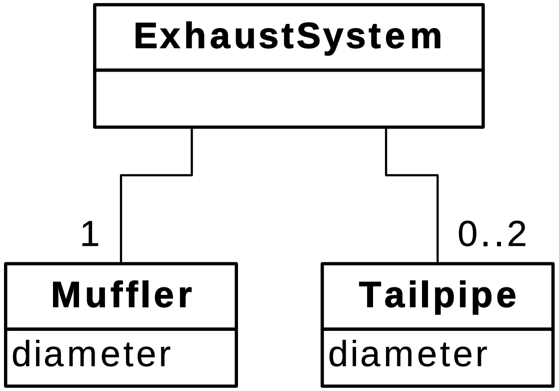


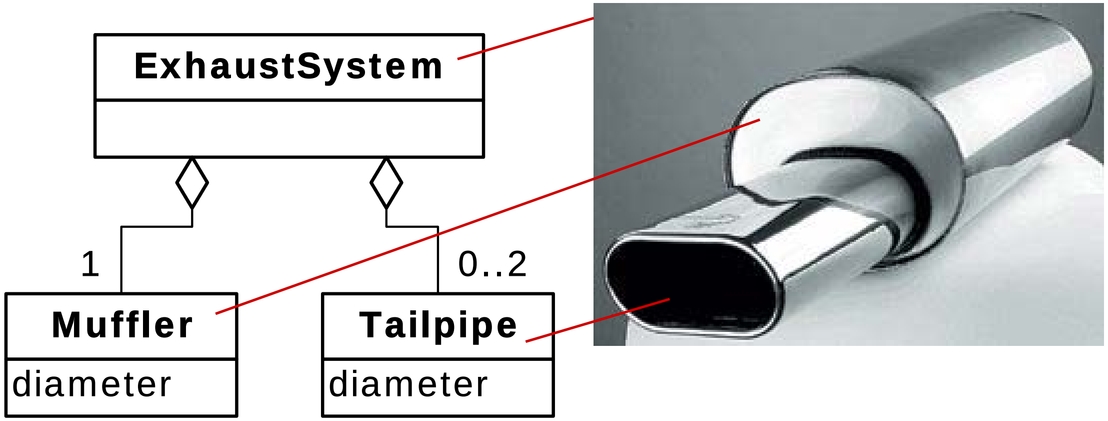
Qualified multiplicity

(Now a Company could have different tickerSymbols at each StockExchange)

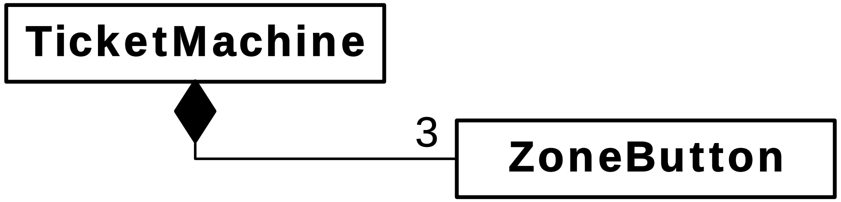
Aggregation

- An ***aggregation*** is a special case of association denoting a "consists of"/"is part of" hierarchy
- The object representing the whole is called the ***aggregate,*** the objects representing the parts are called ***components***

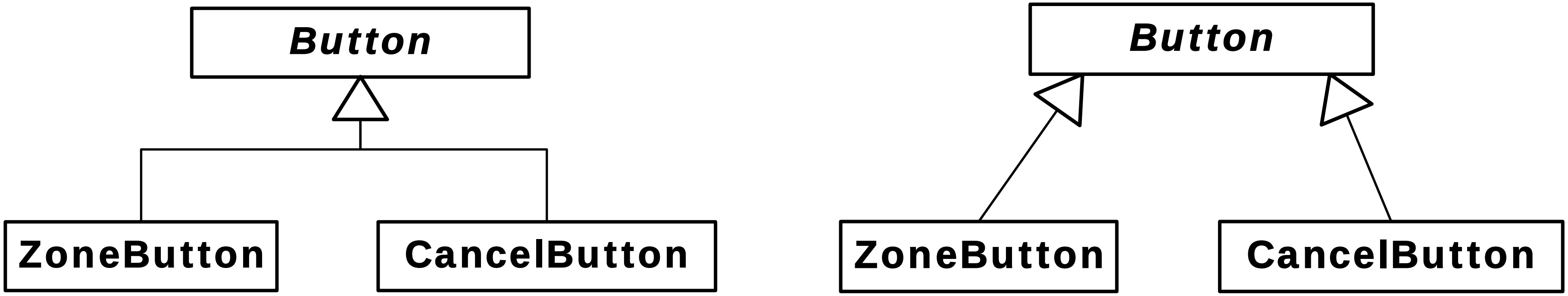




- A solid diamond denotes ***composition***, a strong form of aggregation where the parts never exist without the composite
  - The association is in force throughout the life of the parts objects



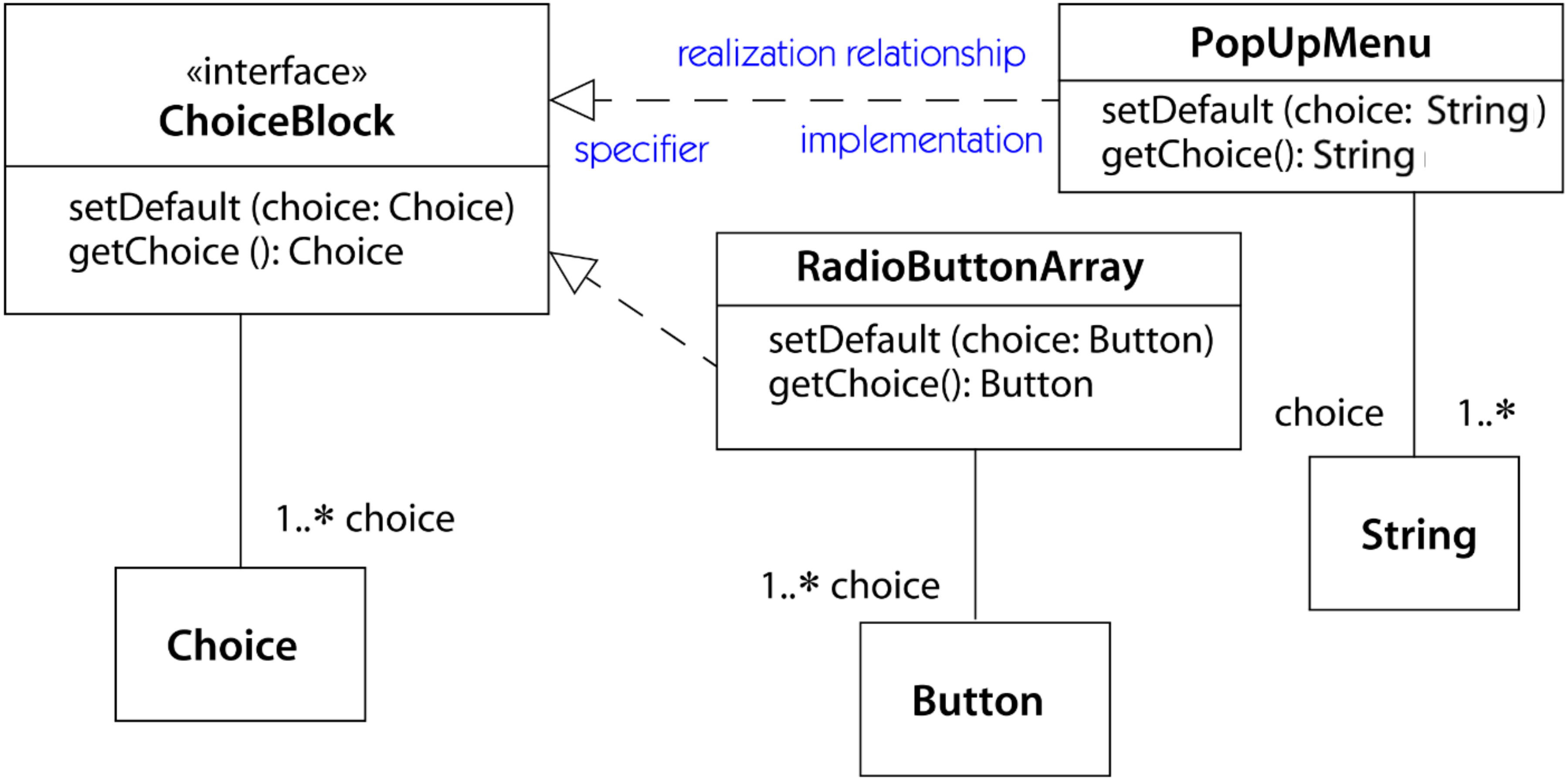
Inheritance (Java: "extends")



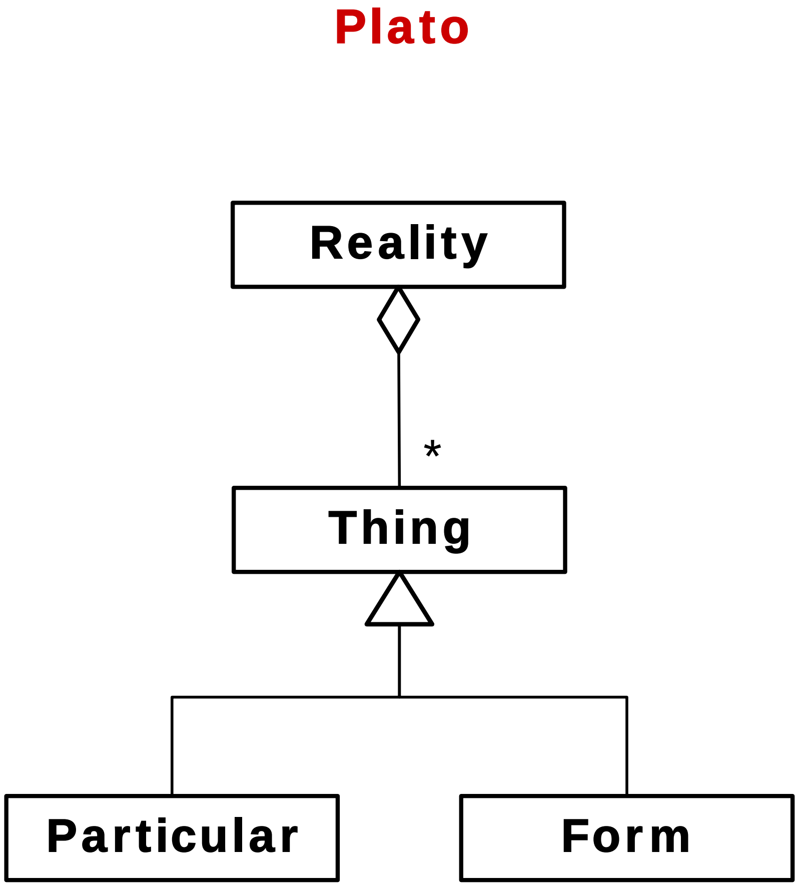
- The **children classes** inherit the attributes and operations of the **parent class**

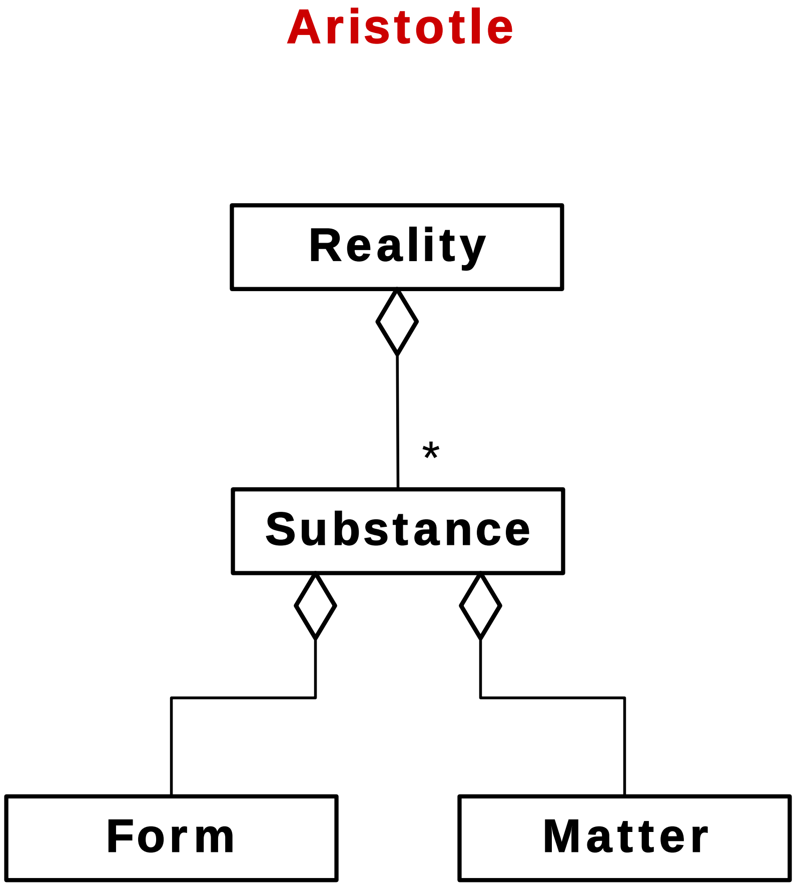
- Read the triangle as an arrowhead, meaning "inherits from" (just like "extends")
  - *CancelButton* inherits from *Button*
  - *ZoneButton* inherits from *Button*

Realization (Java: "implements")



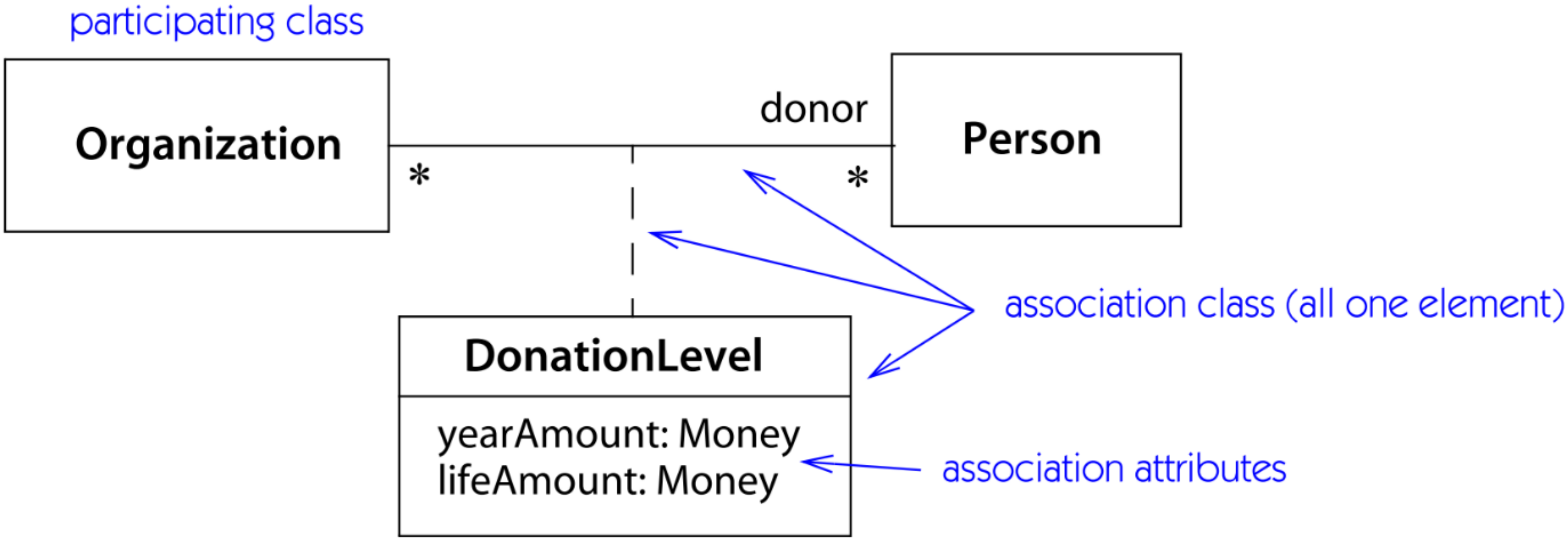
Example: Plato’s and Aristotle’s world views





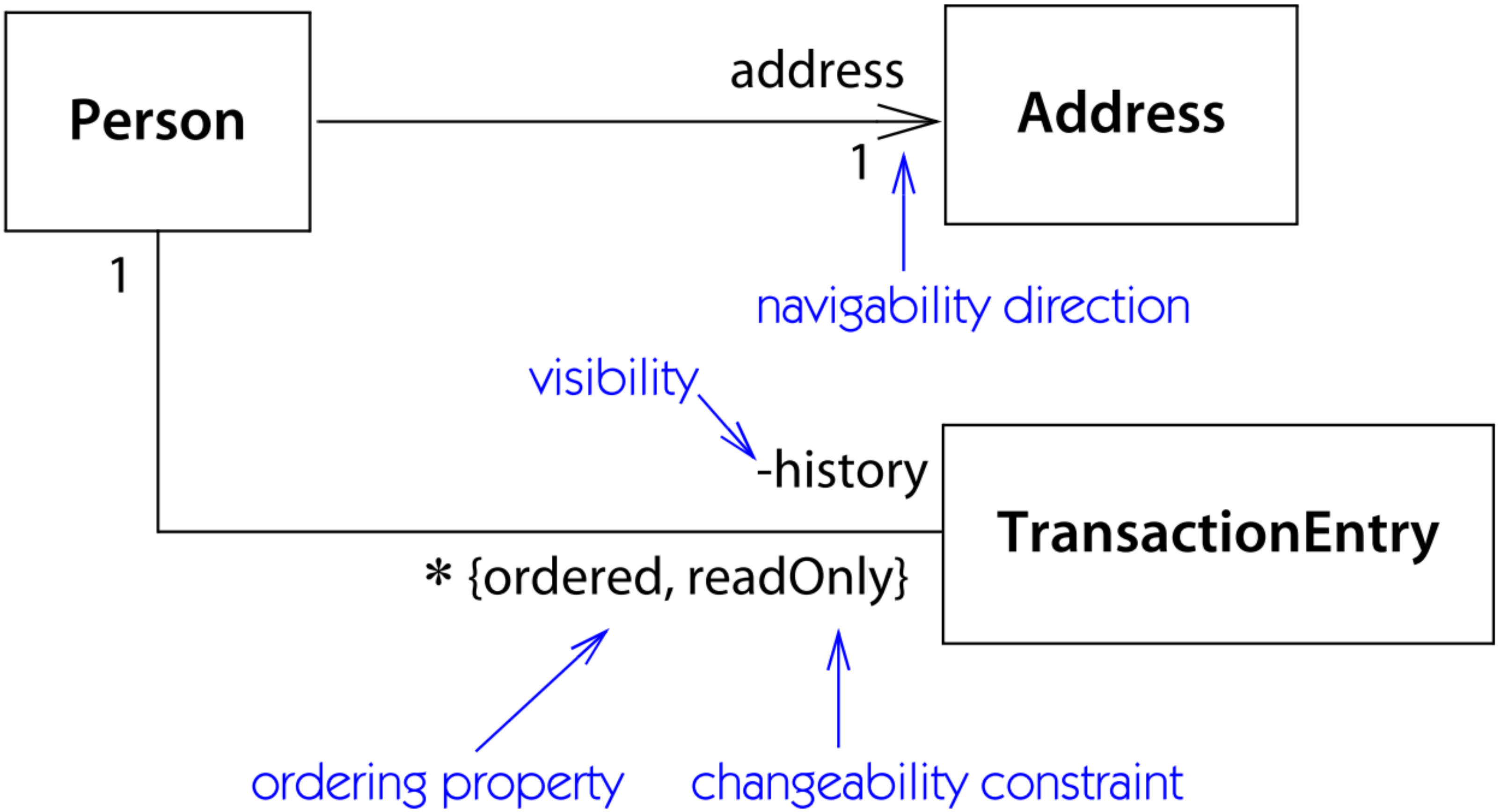
Association classes

- Individual associations between objects can have attributes
  - Described by an association class



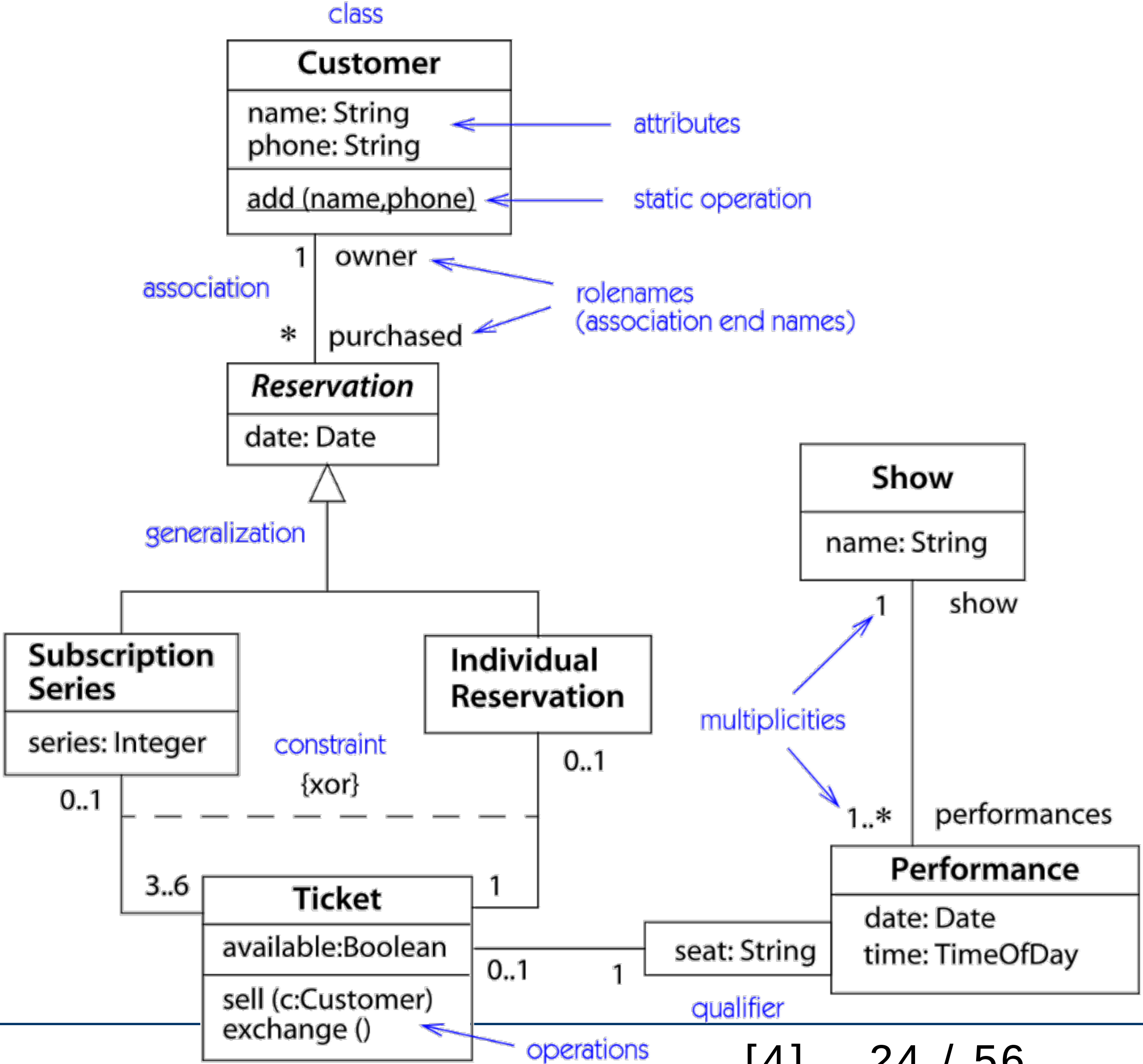
Association constraints

- Associations can be described by further details:



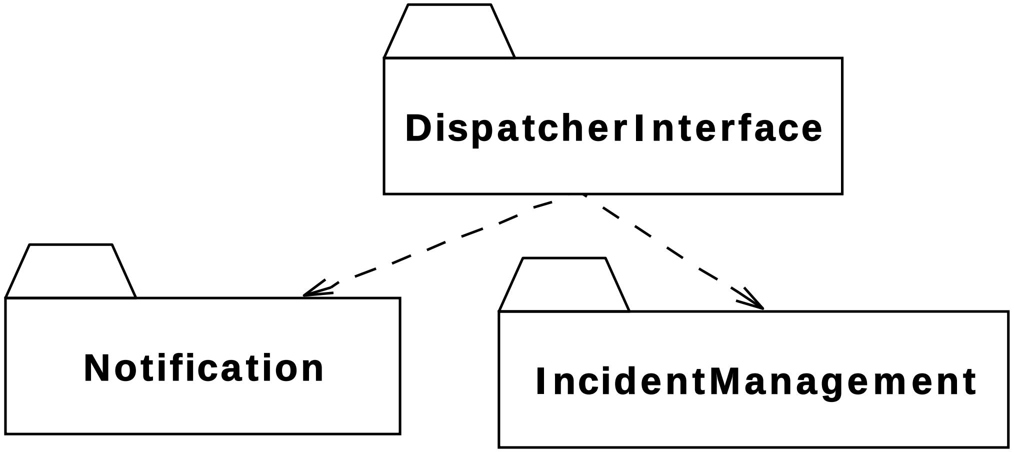
Class diagrams: theater example

- ..and some more notation details:
  - role name
  - XOR constraint
  - static operation



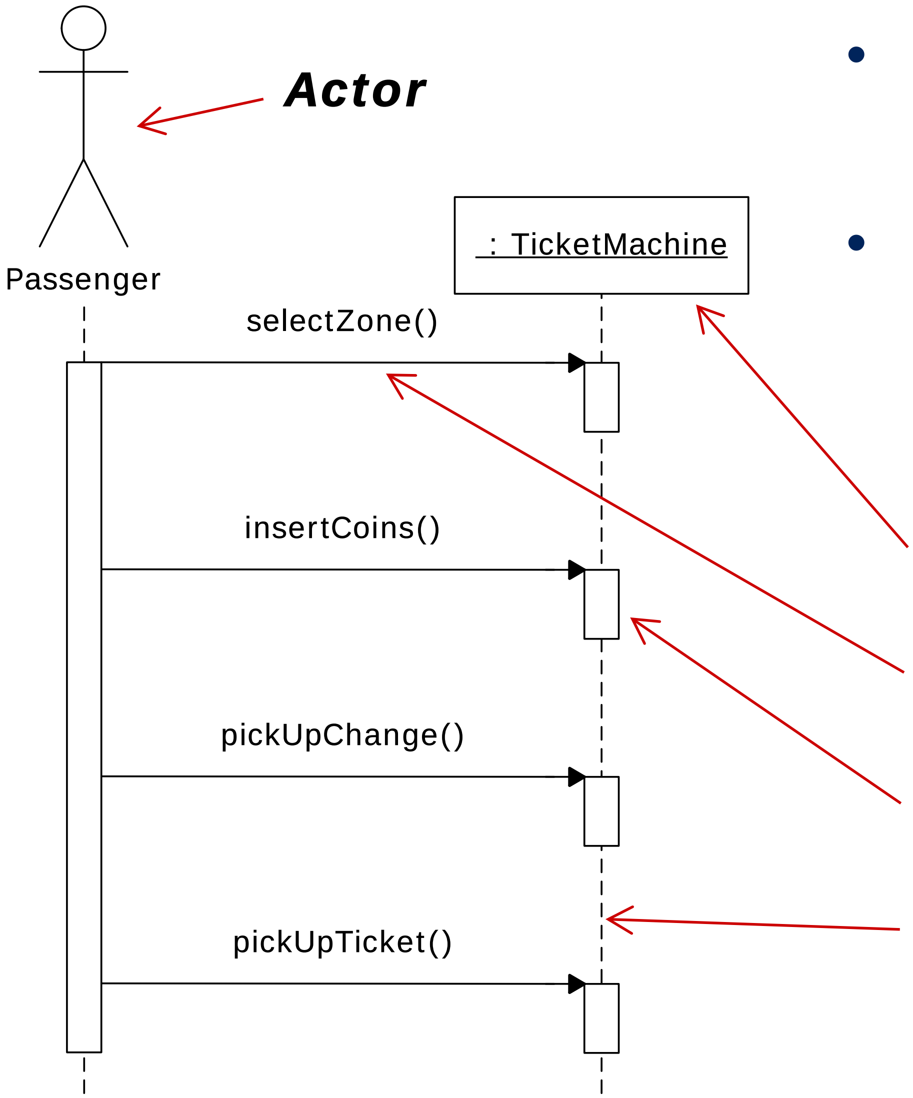
Packages

- A package is a UML mechanism for organizing elements (e.g. classes or whole class diagrams) into groups
  - to simplify model readability and handling



- A complex system can be decomposed into subsystems, where each subsystem is modeled as a package

UML sequence diagrams



Actor

- Used during requirements analysis and system design
  - to refine the model
- Used for explaining behavior to other stakeholders

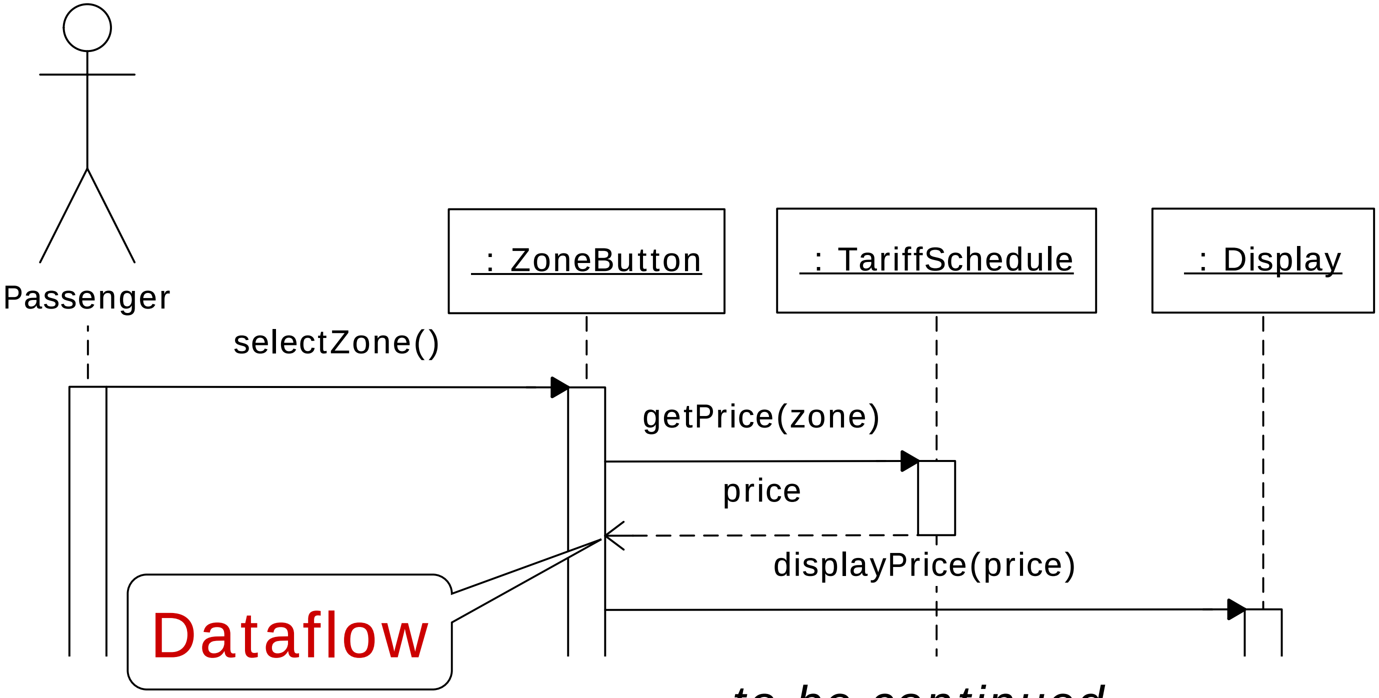
Objects are represented by columns (*objname:classname*)

Messages are represented by arrows

Activations are represented by narrow rectangles

Lifelines are represented by dashed lines

Nested messages

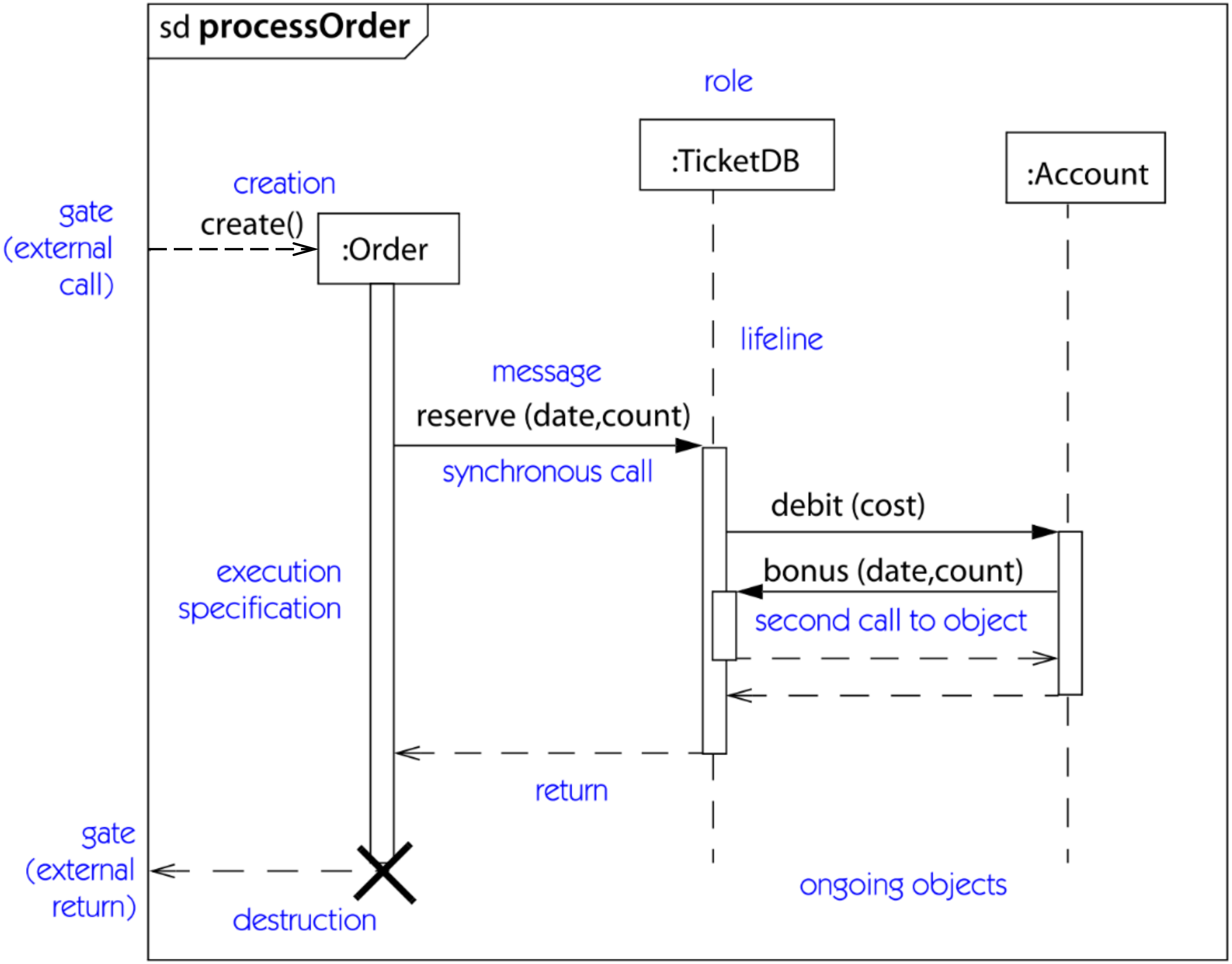


...to be continued...

- The source of an arrow indicates the activation which sent the message
- An activation is as long as all nested activations (for normal calls)
- Horizontal dashed arrows indicate data flow
- Vertical dashed lines indicate lifelines

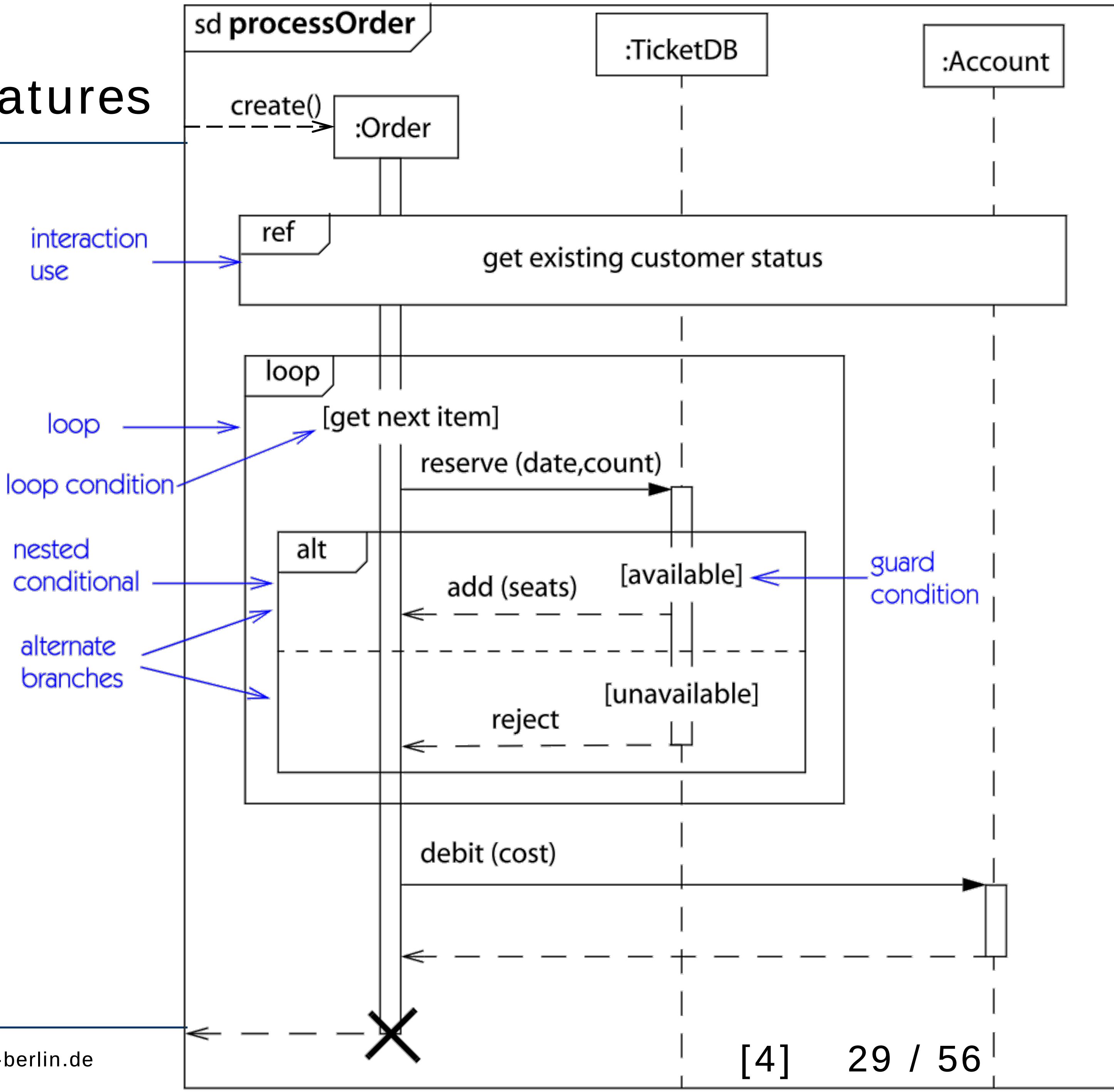
Sequence diagram: theater example

external call,
external return



Advanced features

- creation
- nesting
- iteration
- conditions, branching
- destruction



sd processOrder

create()

:Order

:TicketDB

:Account

ref

get existing customer status

interaction use

loop

[get next item]

loop

loop condition

reserve (date,count)

alt

nested conditional

add (seats)

[available]

guard condition

alternate branches

[unavailable]

reject

debit (cost)

[4]

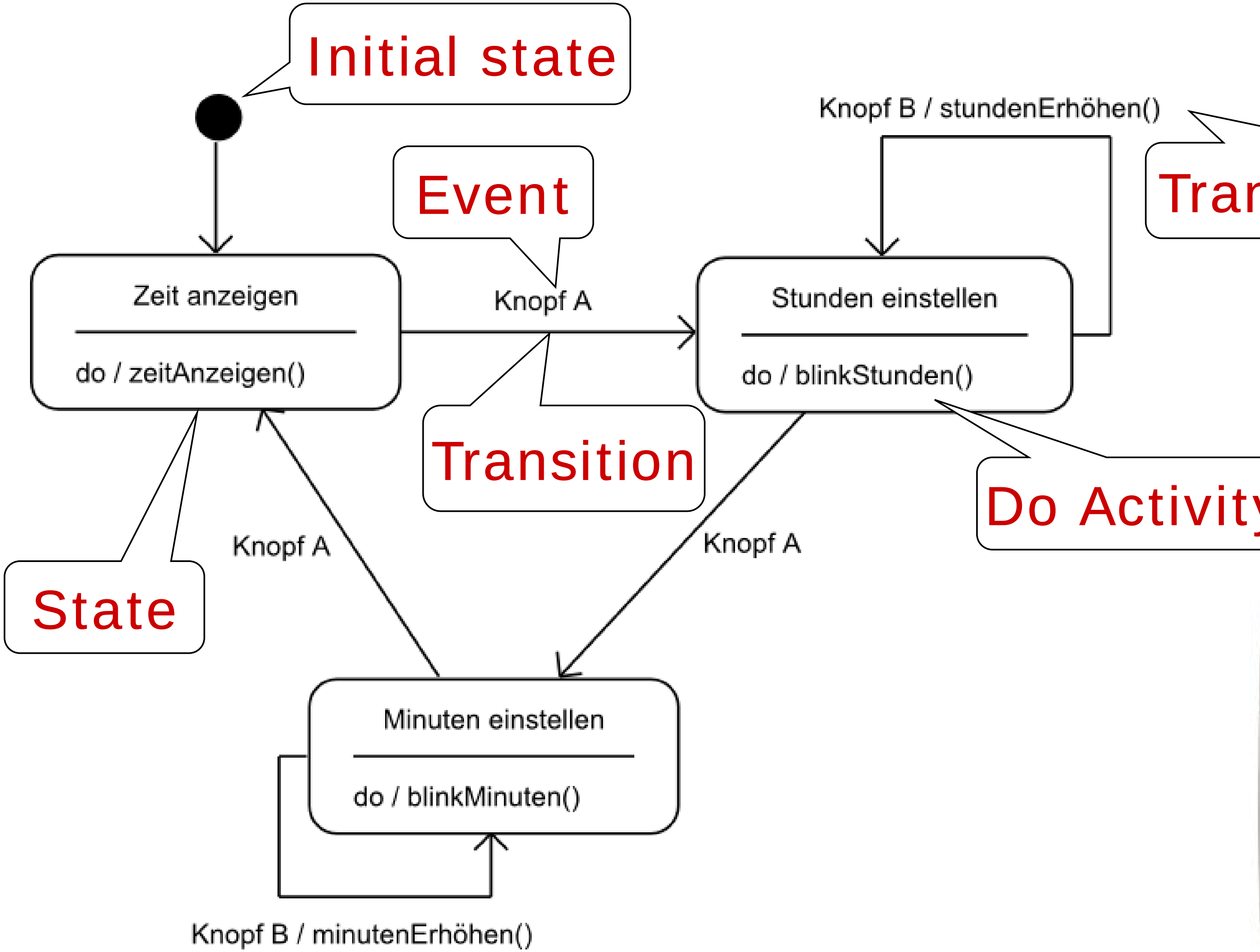
Sequence diagram summary

- UML sequence diagrams represent behavior in terms of object interactions
  - Useful to find missing objects
  - Useful for explaining design ideas
    - Describes examples only, no general specification
- Time-consuming to build, but may be worth it
- Complement the class diagrams (which represent structure)

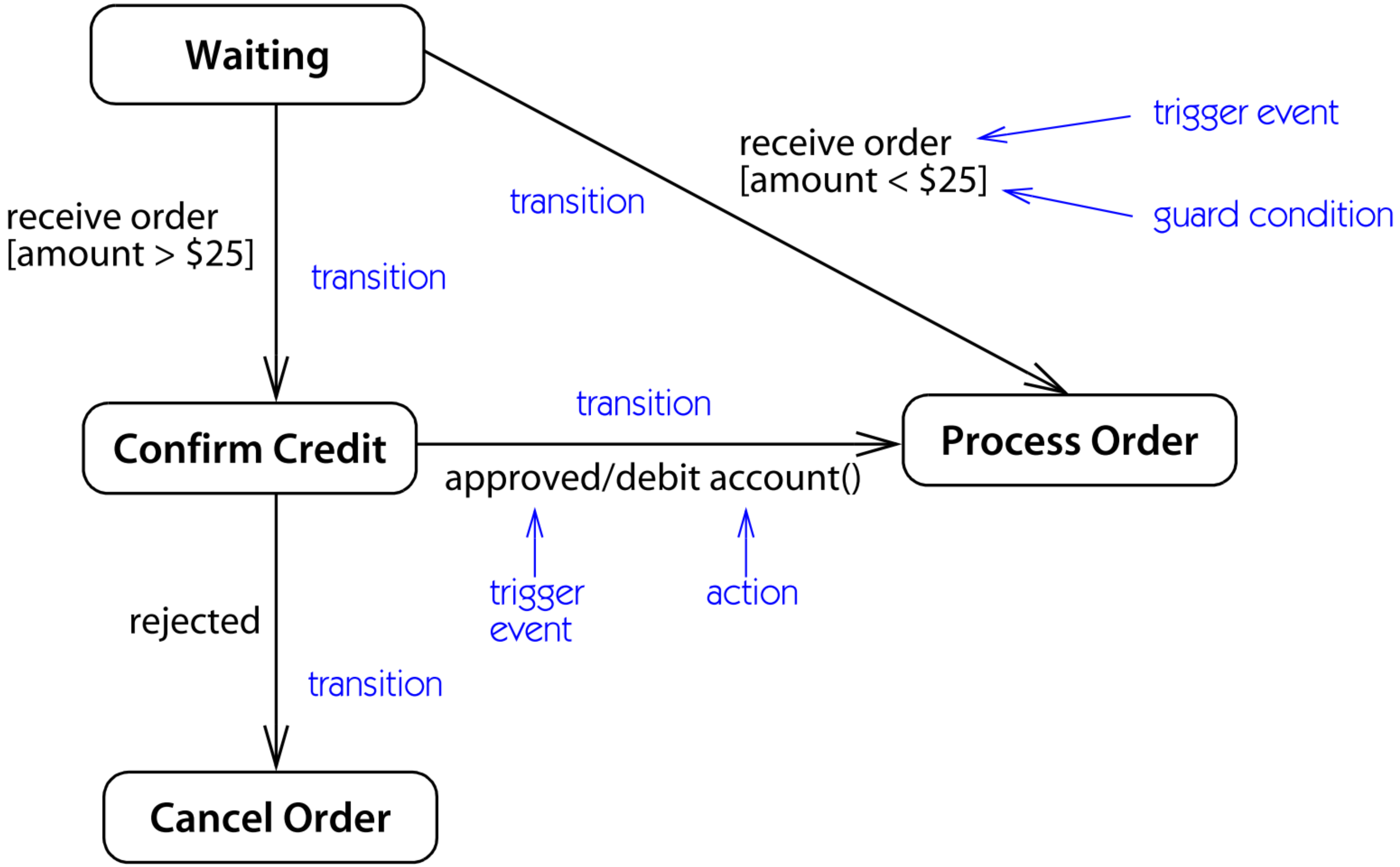
dynamic view

static view

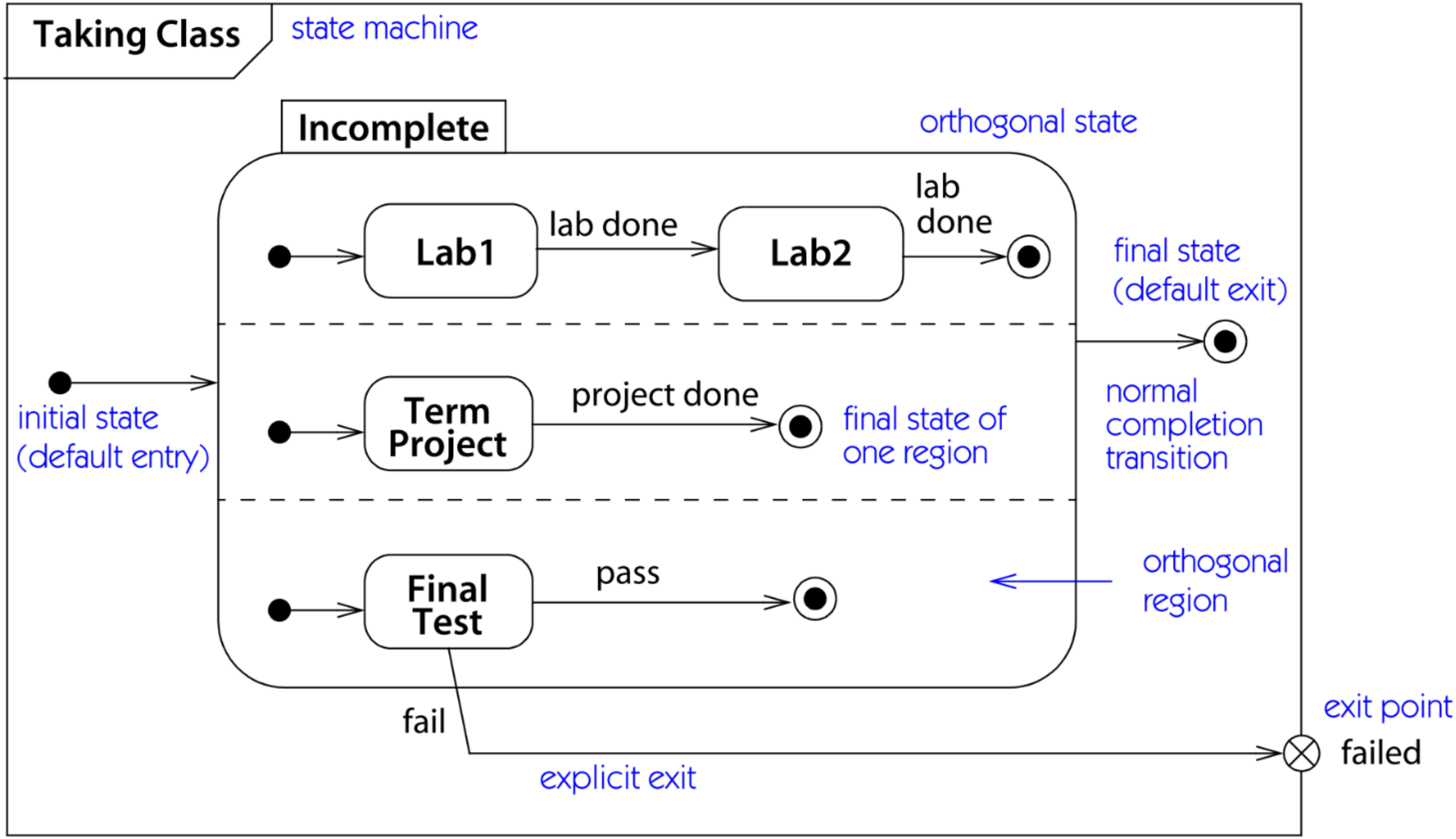
State machine diagrams (statecharts)



Transitions can be subject to guard conditions

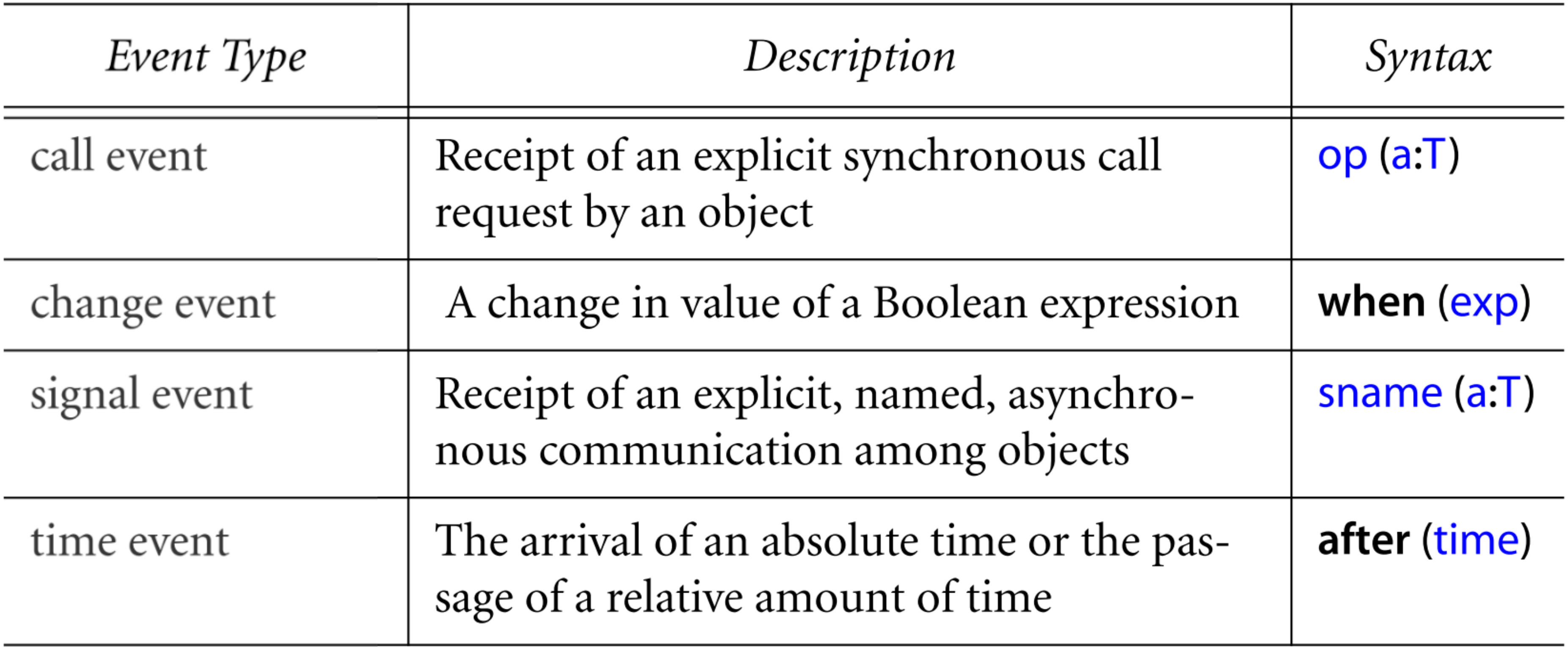


Parallel (orthogonal) states, explicit exits



[4]

A transition is the consequence of an event

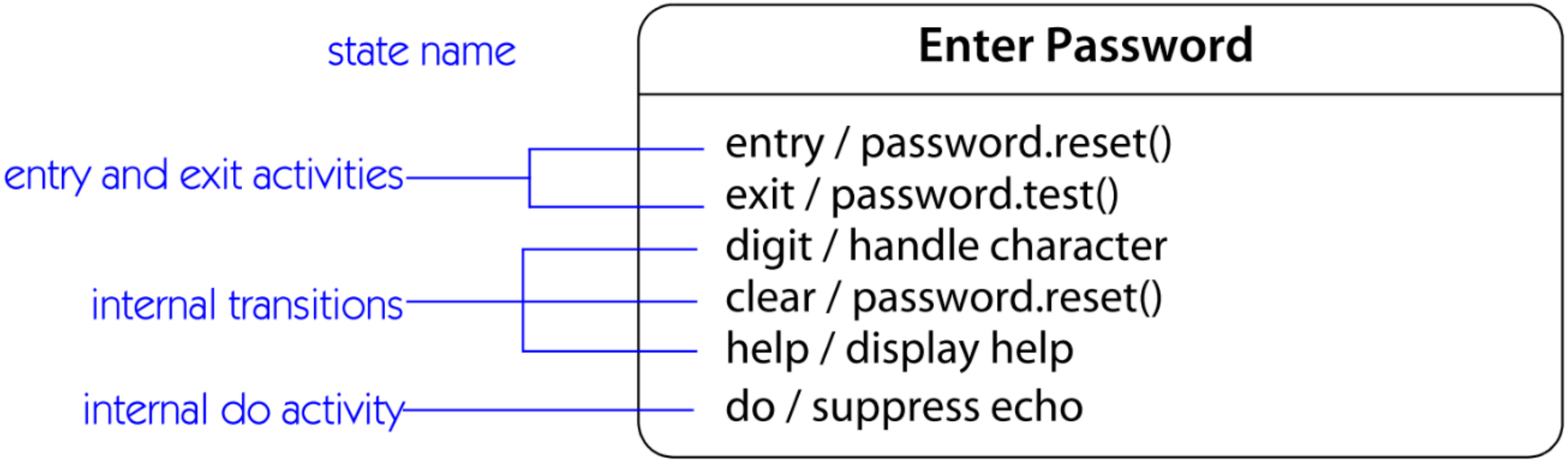


Event Type	*Description*	*Syntax*
call event	Receipt of an explicit synchronous call request by an object	op (a:T)
change event	A change in value of a Boolean expression	**when** (exp)
signal event	Receipt of an explicit, named, asynchronous communication among objects	sname (a:T)
time event	The arrival of an absolute time or the passage of a relative amount of time	**after** (time)

[4]

There can be multiple transitions at one state

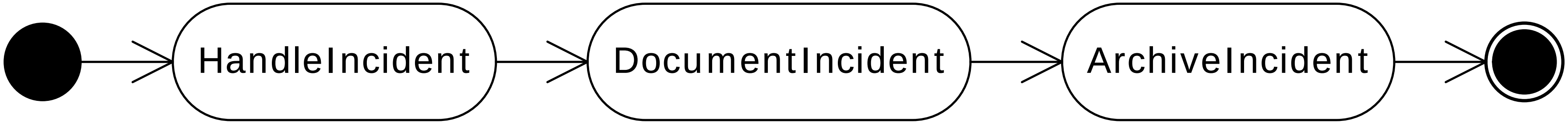
- Internal transitions don't leave the state
- Entry and Exit Activities can be annotated inside the state box
  - to avoid redundancy and encapsulate the state



- also: do / some_activity
  - for a concurrent, abortable, potentially long-running activity occuring throughout the state

Activity Diagrams

- An activity diagram shows flow control (and optionally data flow) within a system

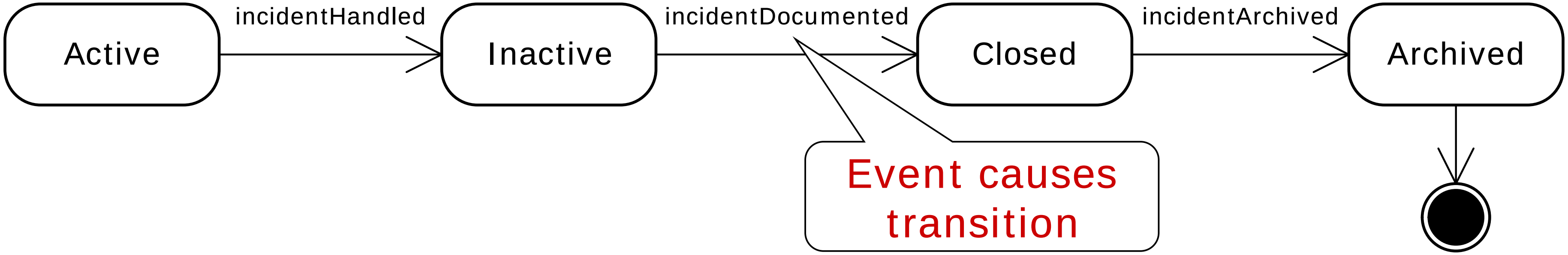


- Two types of (executable) nodes:
  - Action node:
    - Basic activity, cannot be decomposed any further
    - Predefined in UML, e.g. object creation/destruction, accessing/modifying attributes or links, calling operations
  - Activity node:
    - Can be decomposed further
    - The activity is modeled by another activity diagram
- Difference to State Chart?

State machine diagram vs. activity diagram

State machine diagram for Incident

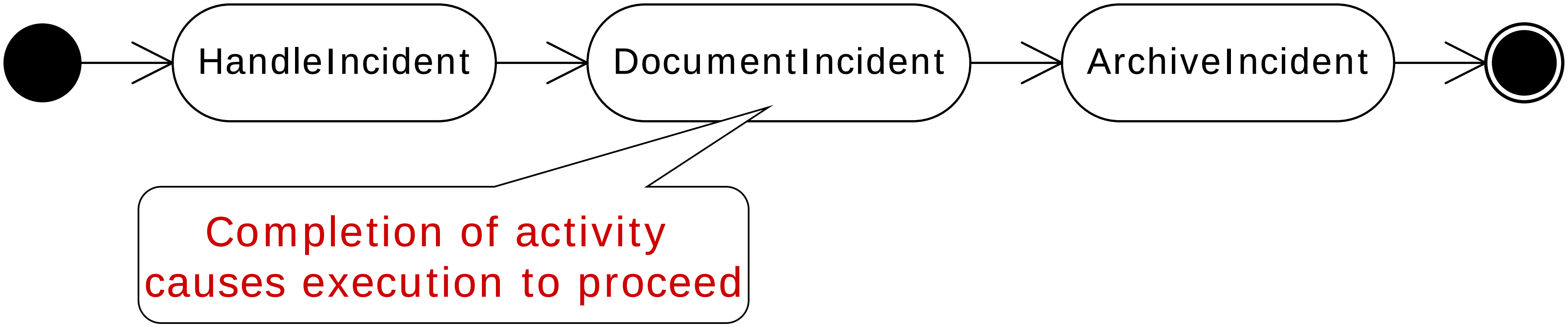
(Node represents some set of attribute values)



Event causes transition

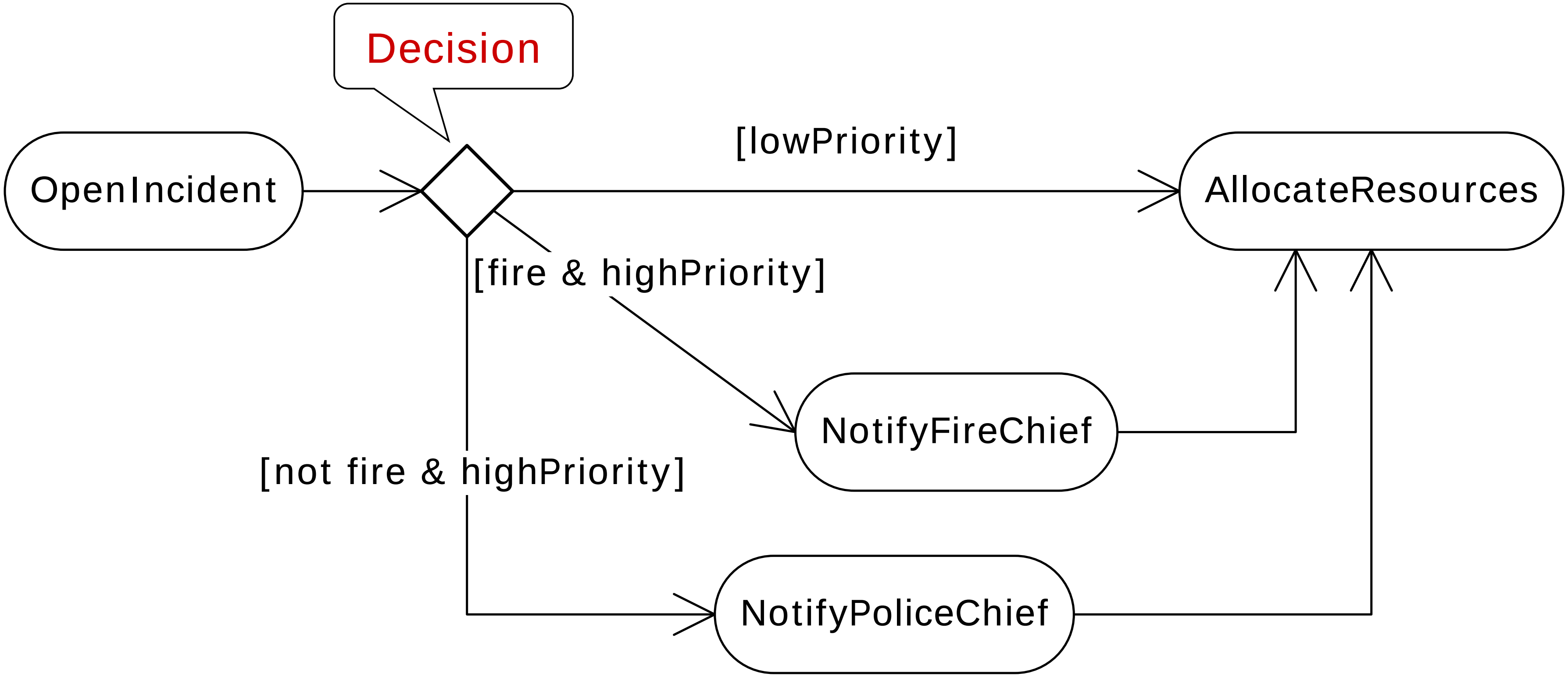
Activity diagram for Incident handling

(Node represents some collection of operations)



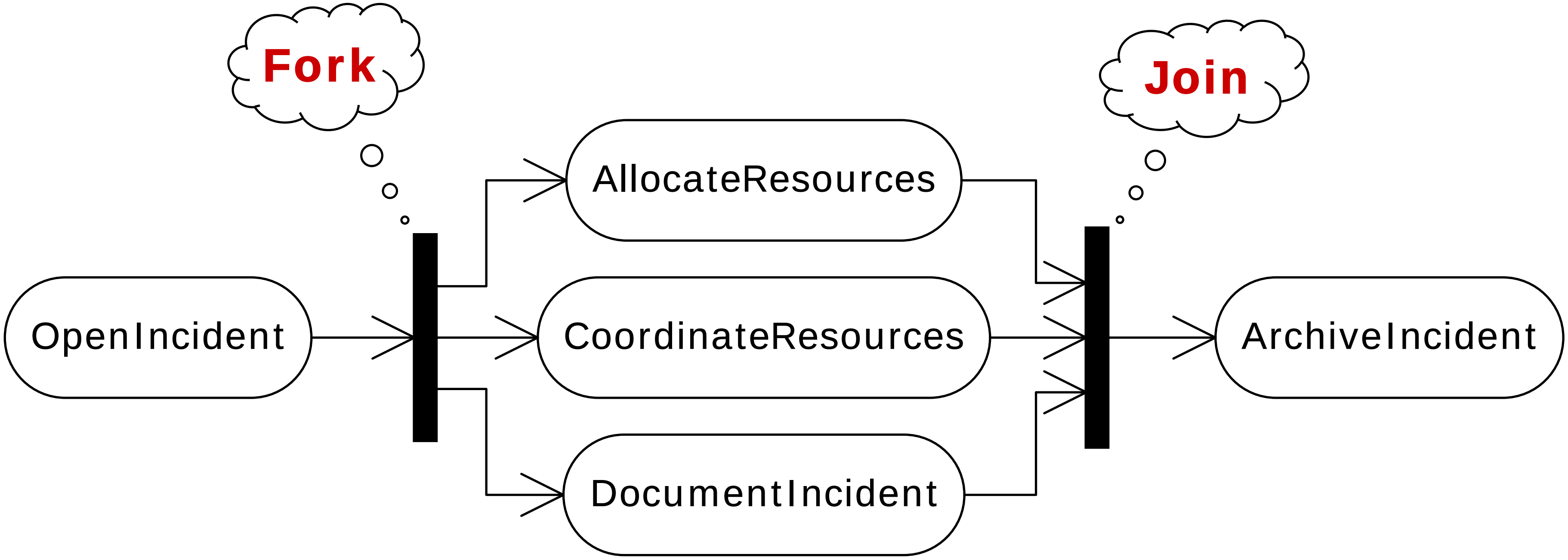
Completion of activity causes execution to proceed

Activity diagram: decisions



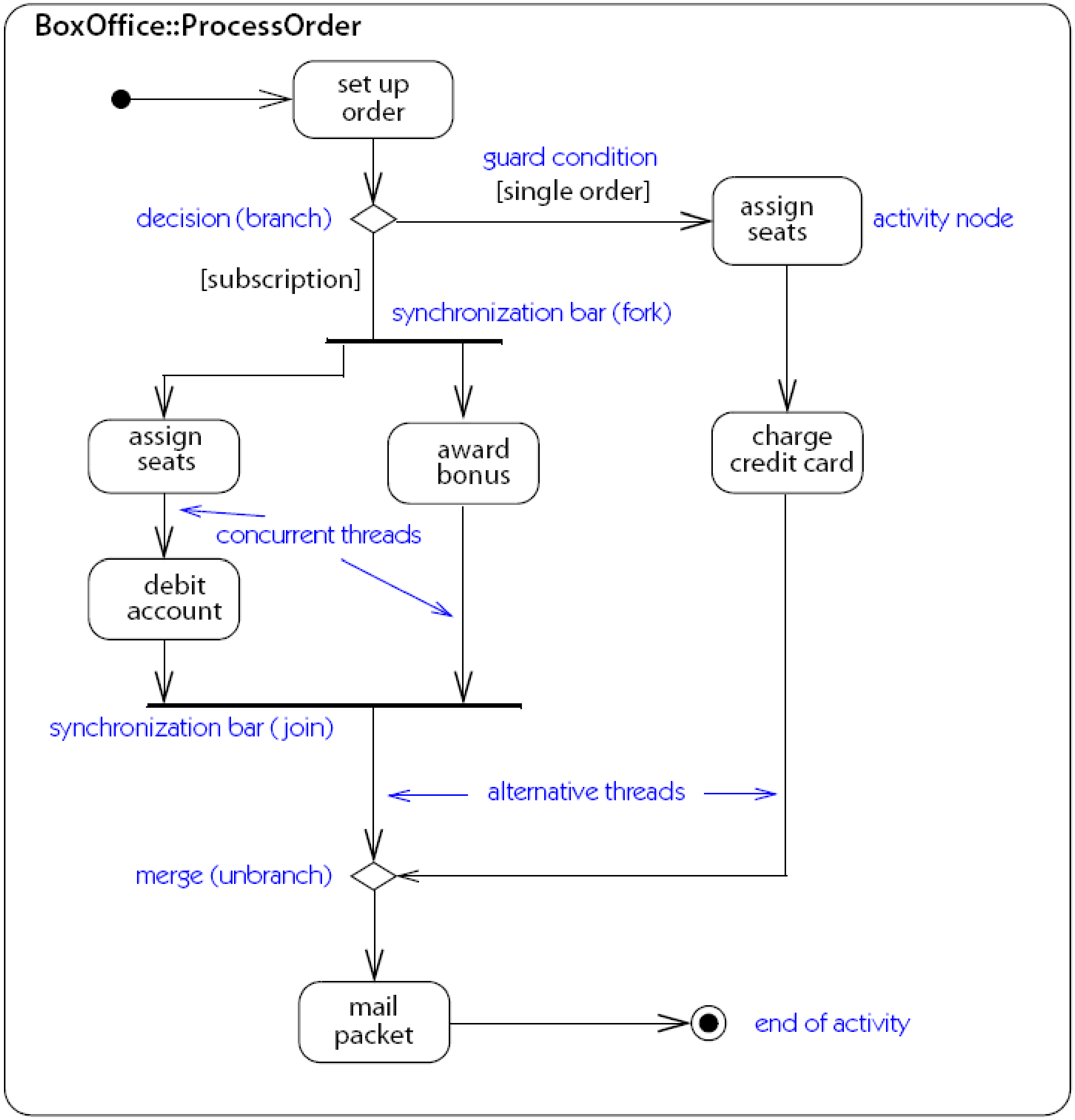
Activity diagrams: concurrency

- Synchronization of multiple activities
- Splitting the flow of control into multiple threads



- Difference between Fork (here)
  and Decision Split (previous slide)?

Activity diagrams: theater example



(ohne neue Konstrukte)

Further UML diagram types

Static view:

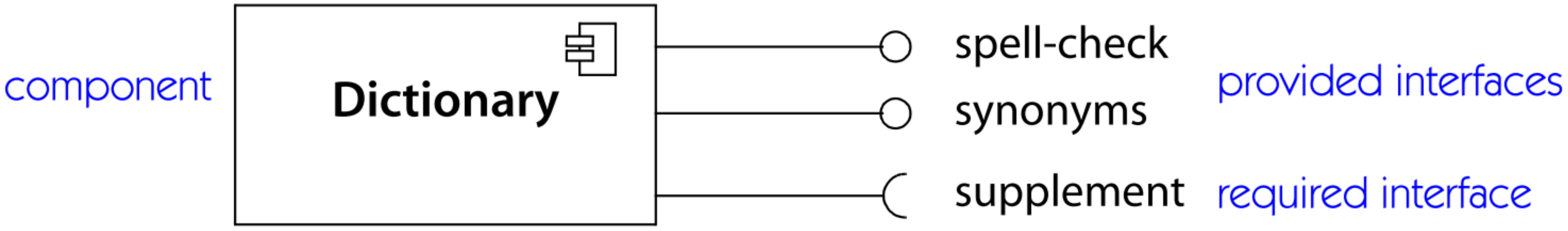
- Component diagrams, internal structure diagrams
  - Subsystems (components) and their interfaces
- Collaboration use diagram
  - A part of a structure that collaborates for a specific purpose
- Deployment diagrams
  - Computers and which part of the system runs on which

Dynamic view:

- Communication diagrams
  - Equivalent to sequence diagrams, but embedded in an object diagram (shows both static structure and dynamic interaction)
- Interaction overview diagrams
  - Related to activity diagrams, for describing control flow

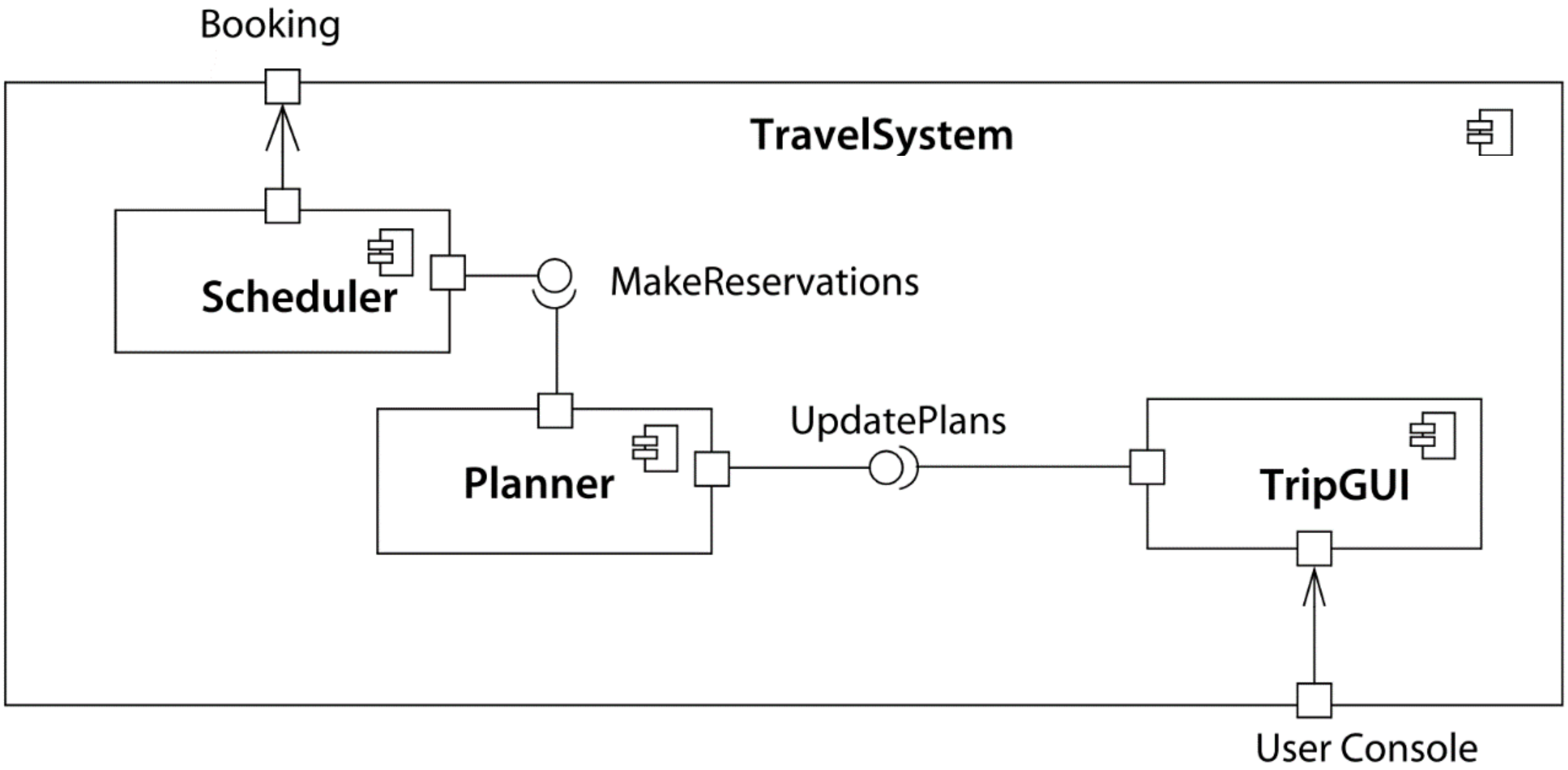
Components

- Components represent classes or subsystems (multiple classes)
  - The focus is on their interfaces
  - All implementation details are ignored



Component diagram, internal structure diagram

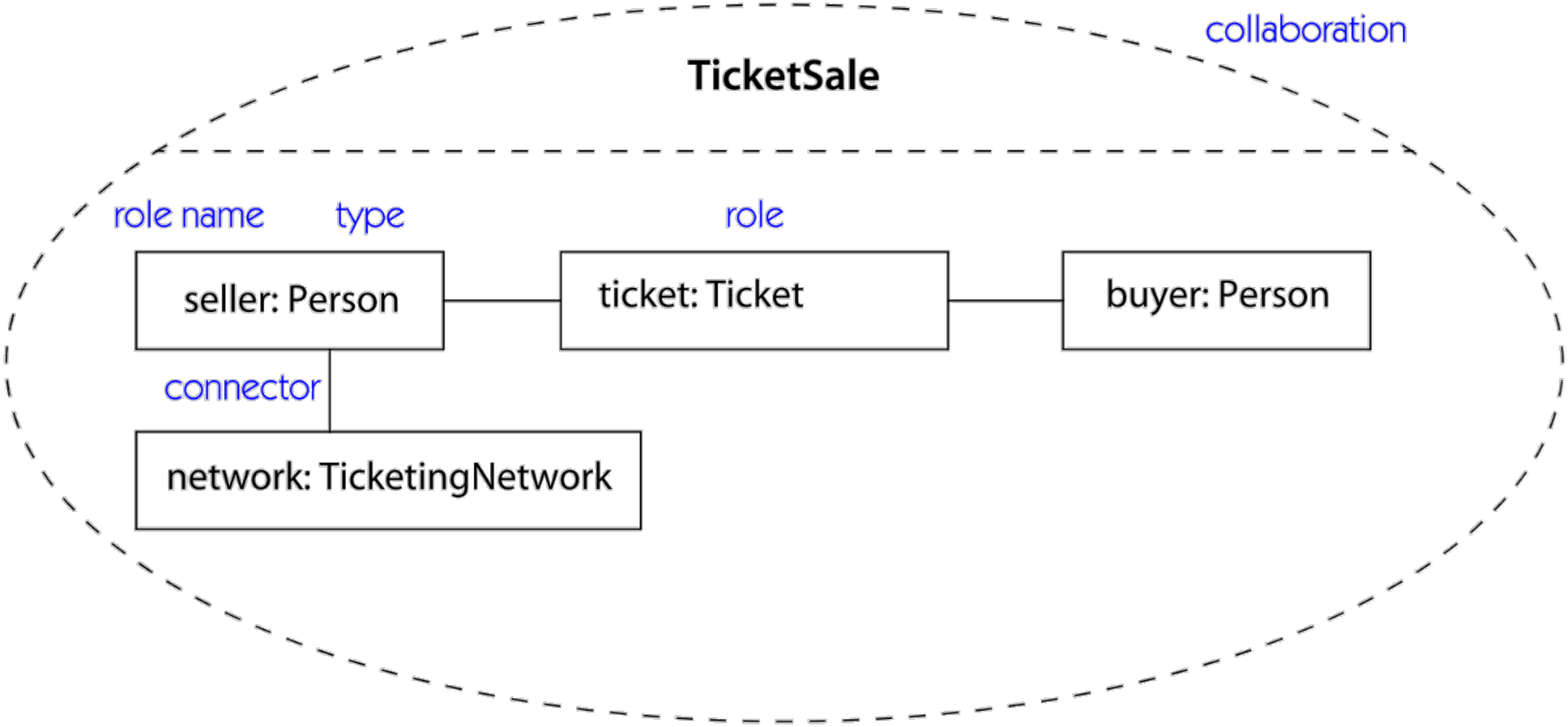
- Compositions of components
  - Component diagram: relationships between components
  - Internal structure diagram: structure of a component (as below) or any other classifier



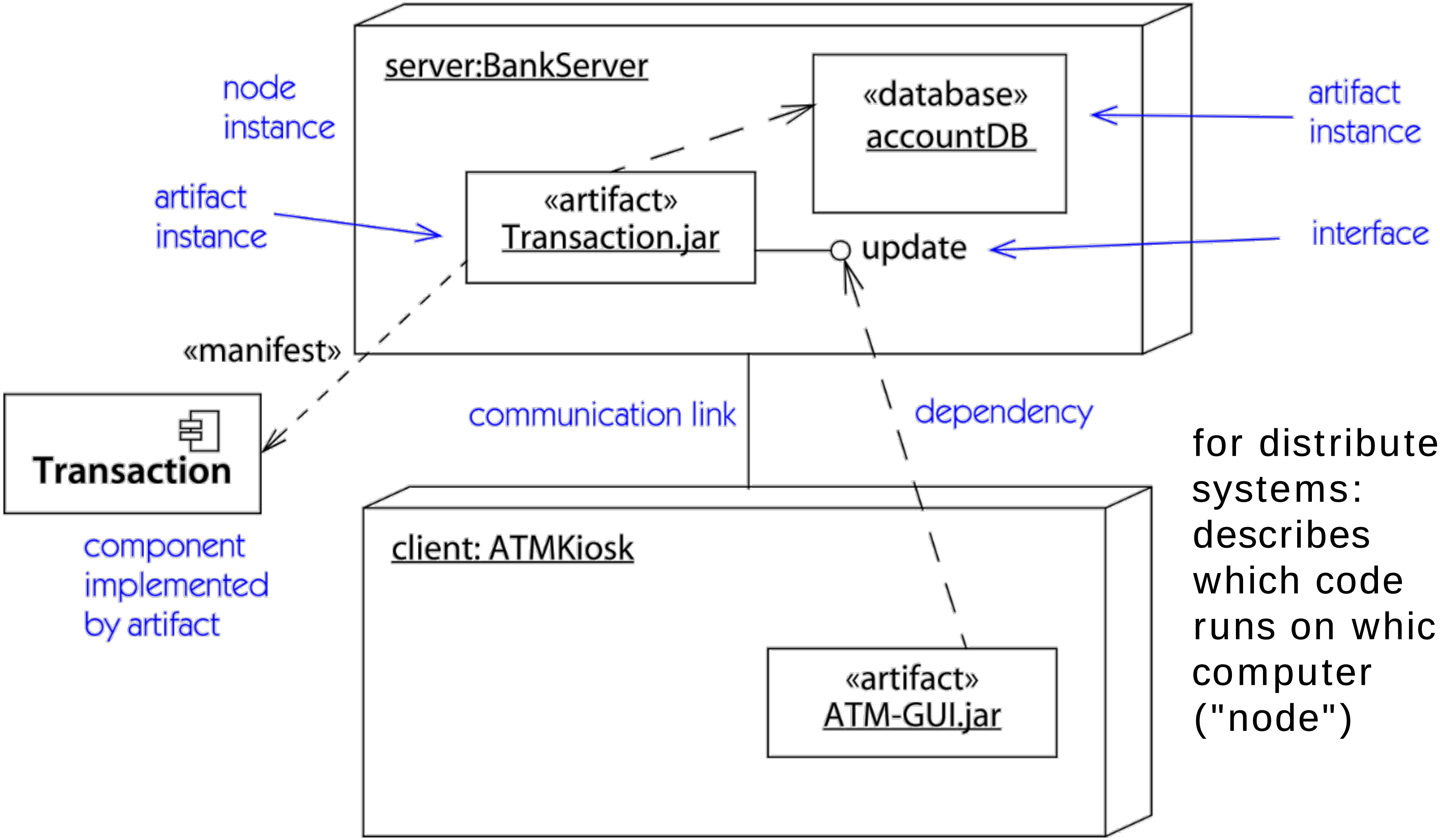
[4]

Collaboration use diagram

- A view describing the roles different parts play for one specific purpose
  - Can be on class level or on instance level



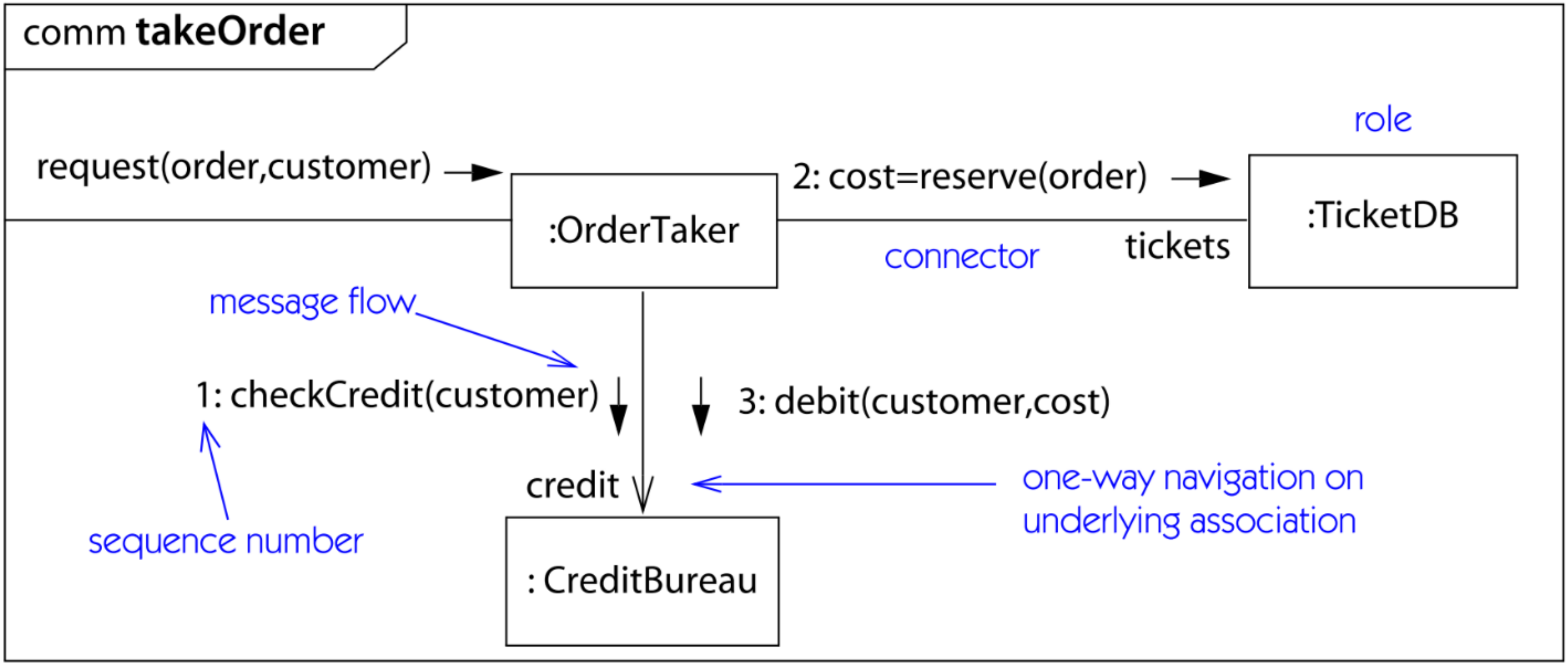
Deployment diagram



for distributed systems: describes which code runs on which computer ("node")

Communication diagram

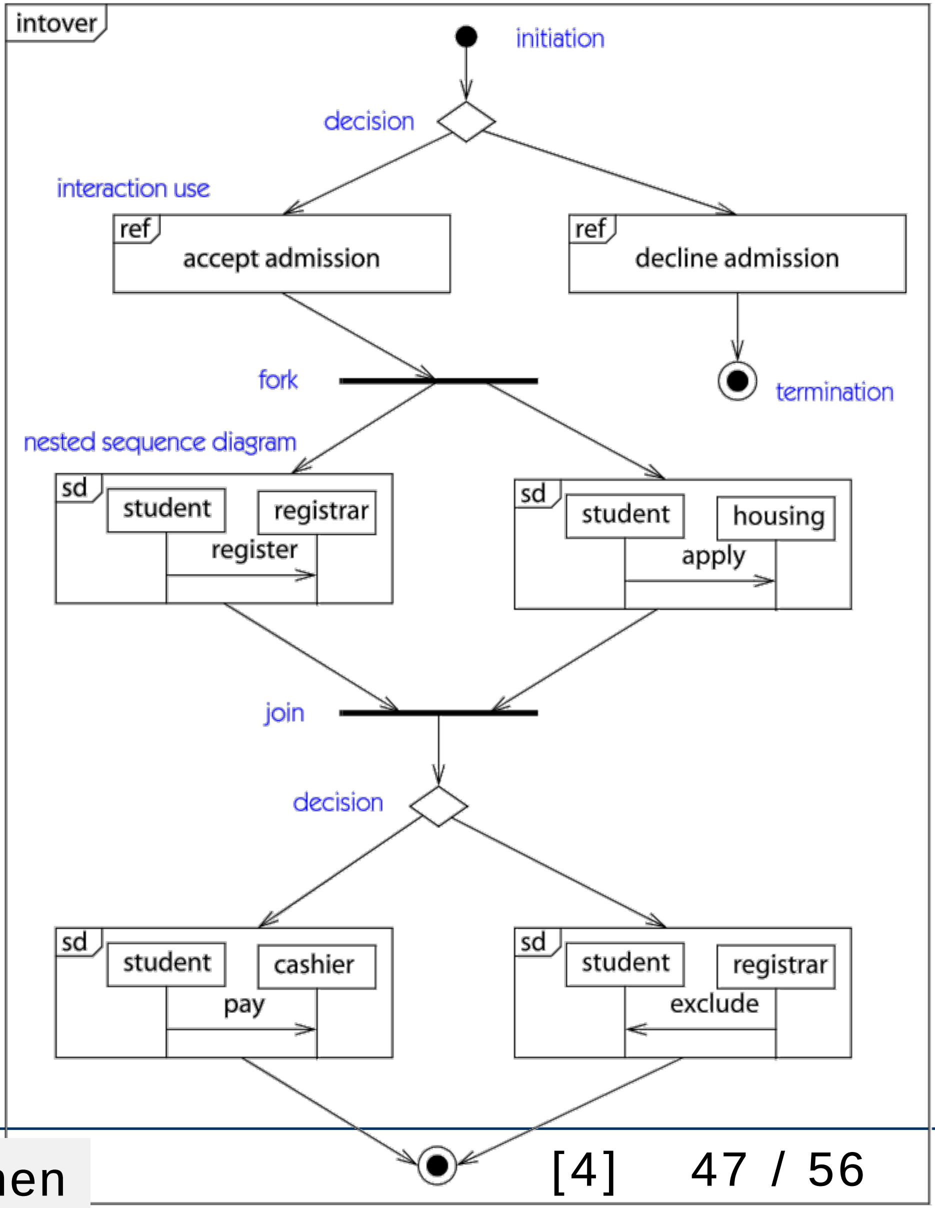
- An object diagram with interaction annotations
  - Indicates interactions (like a sequence diagram) as well as object relationships (by the object diagram)



Interaction overview diagram

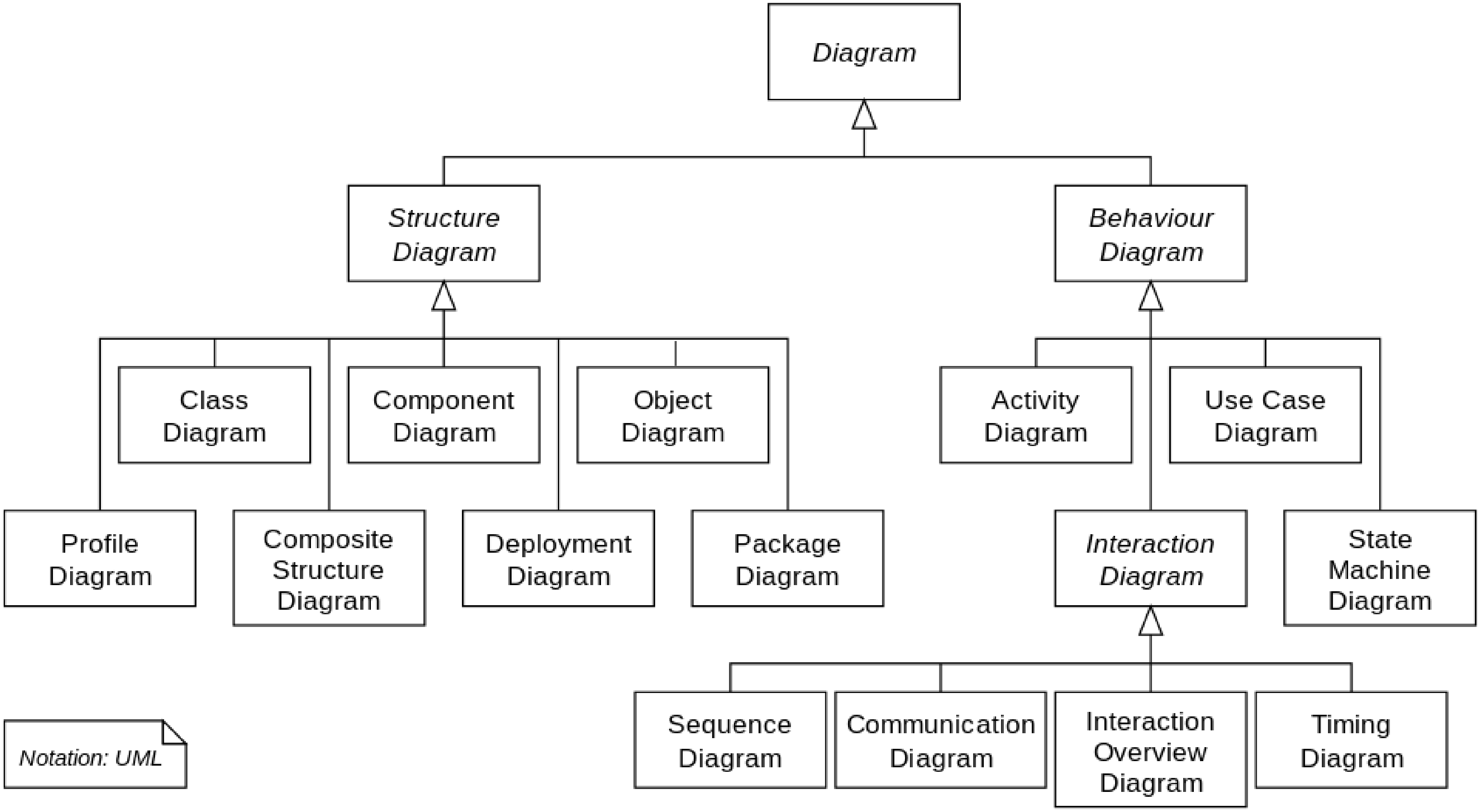
- A combination of two other diagram types. Which?

- Activity diagram and sequence diagram:
  - activities may be sequence diagram fragments



[4]

Diagram types overview (UML 2.2)



source: Wikimedia Commons

[4]

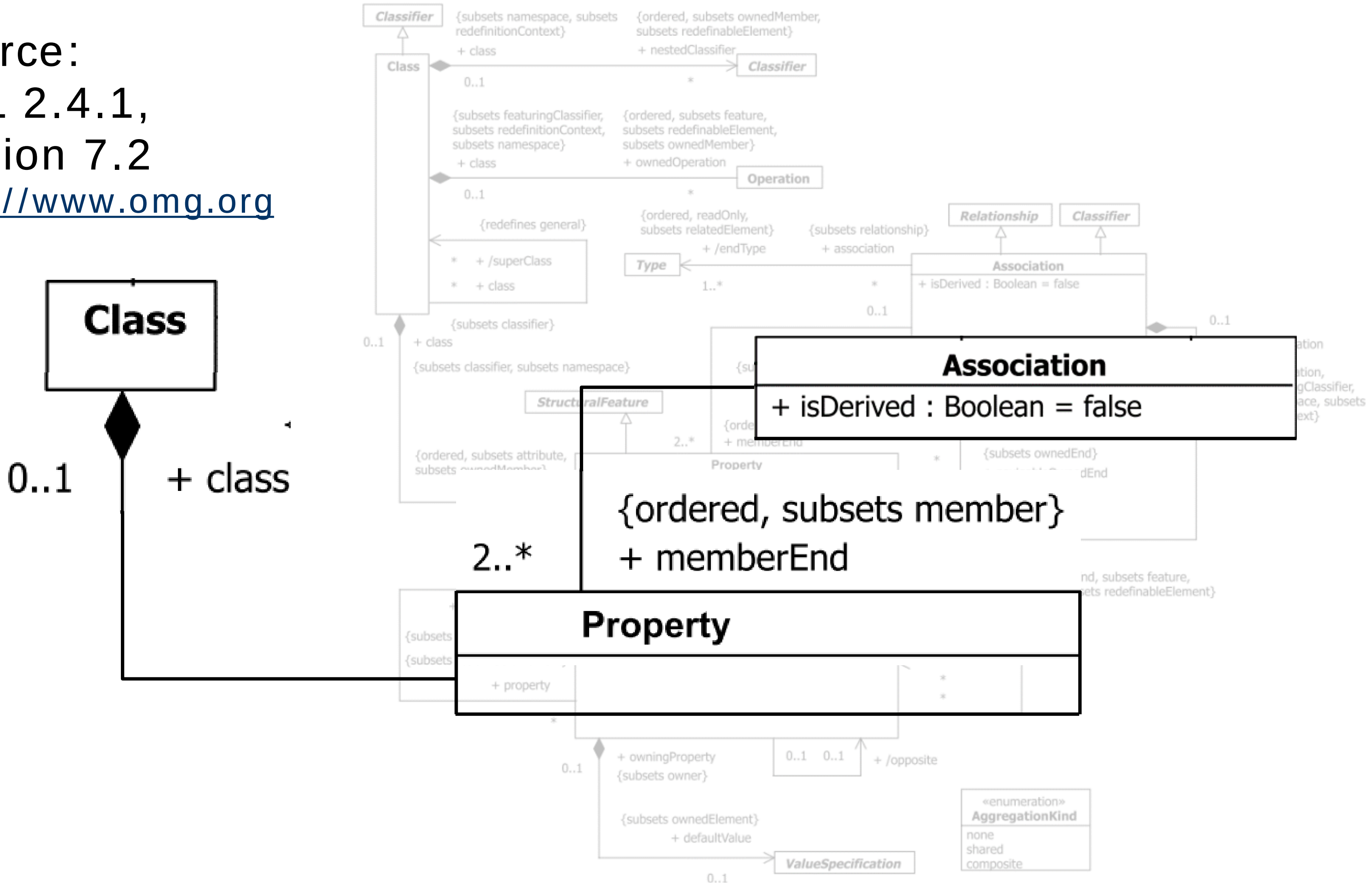
UML is described in UML itself

- The UML model describing UML is called the **UML metamodel**
  - It consists of UML class diagrams plus descriptive text

- Class level: Every kind of UML element (e.g. *"association"*) is a class in that metamodel
  - The characteristics are described by attributes or associated classes
  - e.g. the UML metamodel contains a class *Association*
- Instance level: Every association in a specific UML model can be interpreted as an instance of the *Association* class in the UML metamodel
  - But actually there is much more than just one class:

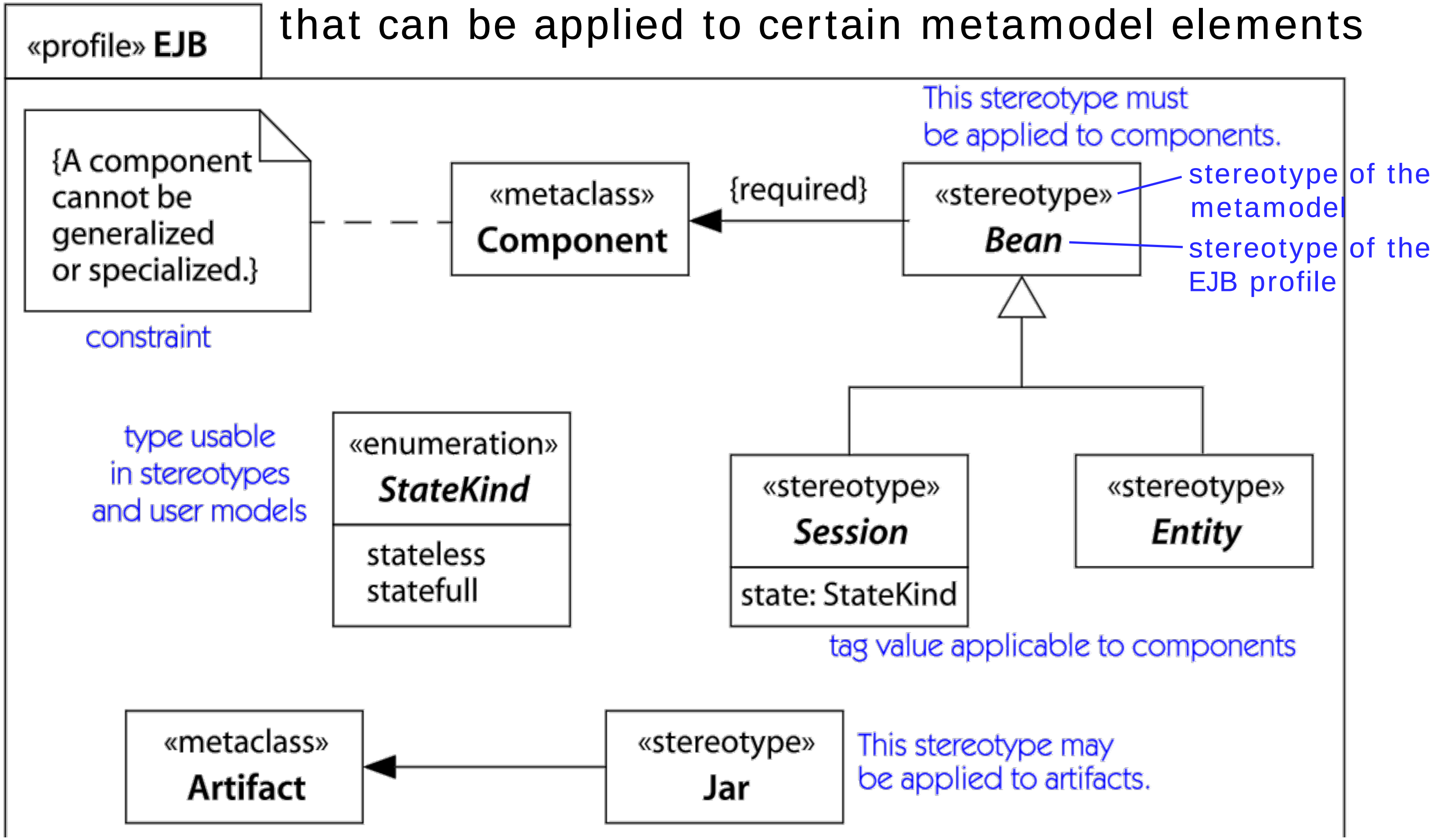
The UML Metamodel of associations

Source:
UML 2.4.1,
section 7.2
http://www.omg.org


Class
0..1
+ class
2..*
{ordered, subsets member}
+ memberEnd
Property
Association
+ isDerived : Boolean = false
Classifier
{subsets namespace, subsets redefinitionContext}
{ordered, subsets ownedMember, subsets redefinableElement}
+ nestedClassifier
{subsets featuringClassifier, subsets redefinitionContext, subsets namespace}
{ordered, subsets feature, subsets redefinableElement, subsets ownedMember}
+ ownedOperation
Operation
{redefines general}
+ /superClass
{ordered, readOnly, subsets relatedElement}
+ /endType
1..*
Type
{subsets relationship}
+ association
Relationship
{subsets classifier}
{subsets classifier, subsets namespace}
StructuralFeature
{ordered, subsets attribute, subsets ownedMember}
{subsets ownedEnd}
+ property
+ owningProperty
{subsets owner}
+ /opposite
{subsets ownedElement}
+ defaultValue
ValueSpecification
«enumeration»
AggregationKind
none
shared
composite

UML is extensible

- **Profiles** add elements to the UML metamodel
  - A profile is a package that defines «stereotypes» and constraints that can be applied to certain metamodel elements


«profile» EJB
{A component cannot be generalized or specialized.}
constraint
«metaclass» Component
{required}
«stereotype» Bean
This stereotype must be applied to components.
stereotype of the metamodel
stereotype of the EJB profile
type usable in stereotypes and user models
«enumeration» StateKind
stateless
statefull
«stereotype» Session
state: StateKind
«stereotype» Entity
tag value applicable to components
«metaclass» Artifact
«stereotype» Jar
This stereotype may be applied to artifacts.

UML is fairly precise

- In this course, we will often use UML in a rather informal and imprecise manner
  - Our models are usually not very detailed
  - They leave many things unspecified (i.e., they are incomplete)
- However, one can produce fairly precise UML models
  - Such models have a reasonably well-defined meaning, as UML itself is specified in a semi-formal manner
    - No complete semantics have been specified for UML overall, though
  - There is **much** more to UML than can be said here
    - UML 2.4 Infrastructure + Superstructure: 200 + 800 pages
    - UML 2.5, rewritten in one document: 800 pages
- Precise UML usage is relevant for automatic code generation from the UML model
  - In some domains, such as telecommunication, complete subsystems are sometimes code-generated from UML models today

What should you know about UML?

- For all application domains:
  - Learn as much as you can about class diagrams (object diagrams help in doing this)
    - (soon maybe also component diagrams)
  - Learn the basics of use case, sequence, communication, state machine, and activity diagrams
- For realtime and formally specifiable (sub)domains:
  - Also learn a lot about state machine diagrams
- If you want to make full use of UML CASE tools:
  - Learn a lot about packages and about profiles
- If you want to build UML CASE tools:
  - Learn about the UML metamodel (Warning: tough!)

UML summary

- UML provides a wide variety of notations for representing many aspects of software development
  - Powerful, but complex language
  - Can be misused to generate unreadable models
  - Can be misunderstood when using too many exotic features
  - Many people who say they "know UML" actually know very little

- We will concentrate on a few notations:

Functional model:	use case diagram (next lecture)
Object model:	class diagram
Dynamic model:	sequence diagrams, state machine and activity diagrams

Literature

- James Rumbaugh, Ivar Jacobson, Grady Booch: *"The Unified Modeling Language Reference Manual",* Second Edition (UML 2.0), Addison-Wesley 2005.
  - this is also the source of the figures with blue annotations

- James Rumbaugh, Ivar Jacobson, Grady Booch: *"The Unified Modeling Language User Guide",* Second Edition (UML 2.0), Addison-Wesley 2005.
  - actually teaches how to *use* the UML
    - this lecture did not do this, but some of the rest of the course will
  - less misleading than some other books on the topic

The current version of UML is 2.5.1 (December 2017).

- http://www.omg.org/spec/UML

Thank you!

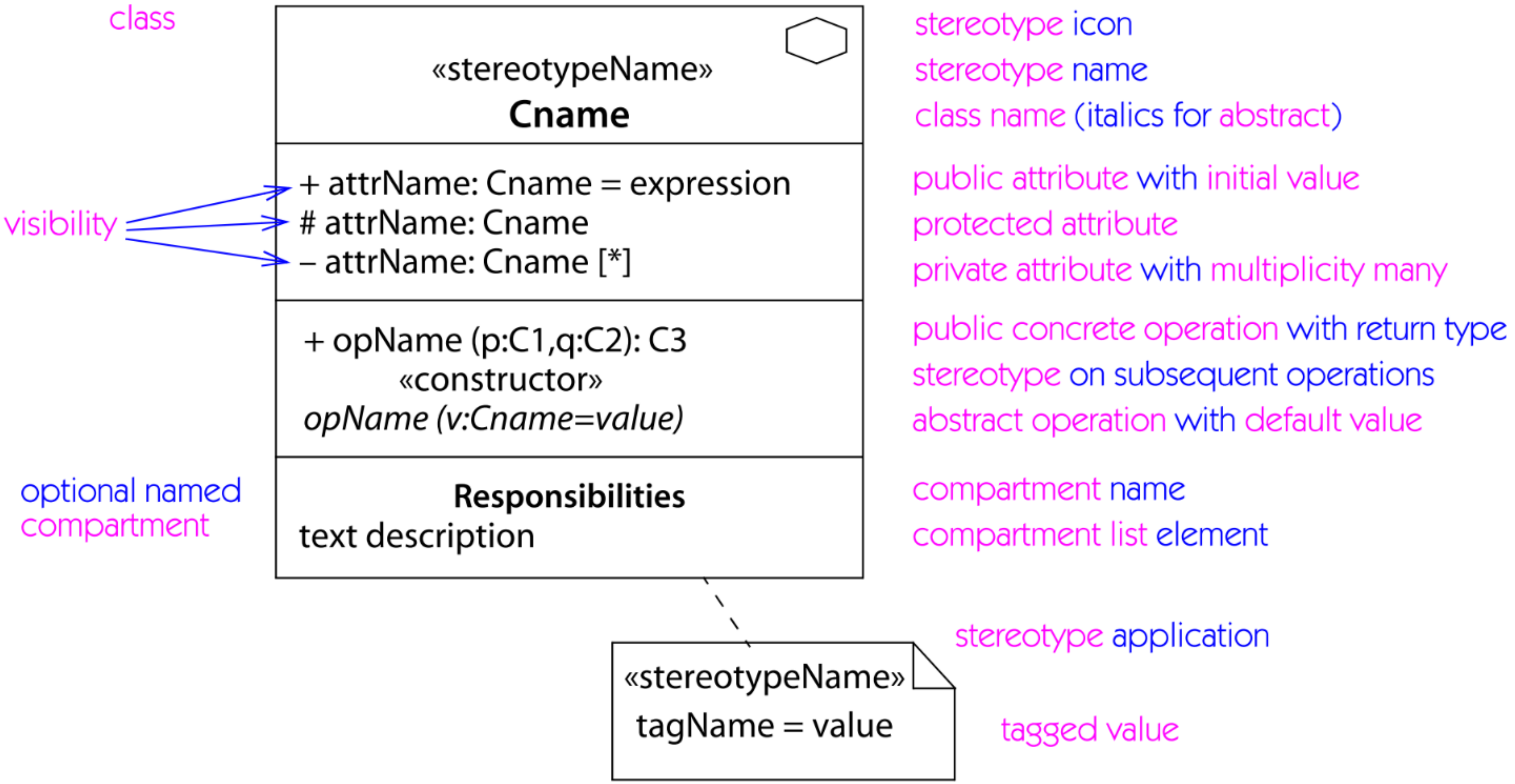
and now: some bonus slides

UML language elements details

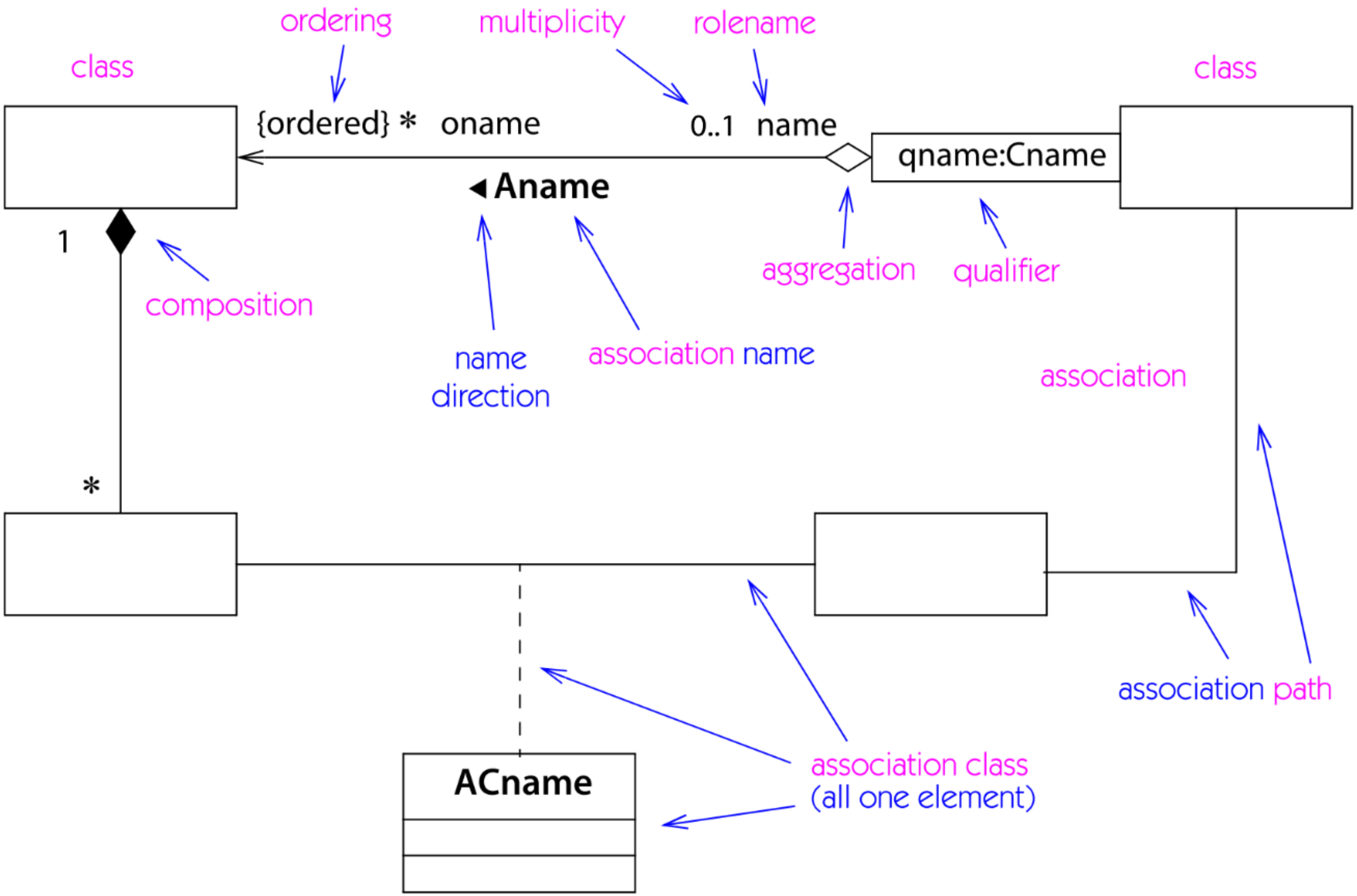
- The next few slides present a number of details in the notation of

Classes	(Class diagrams)
Associations	(Class diagrams)
Interfaces	(Class diagrams)
States	(State machine diagrams)

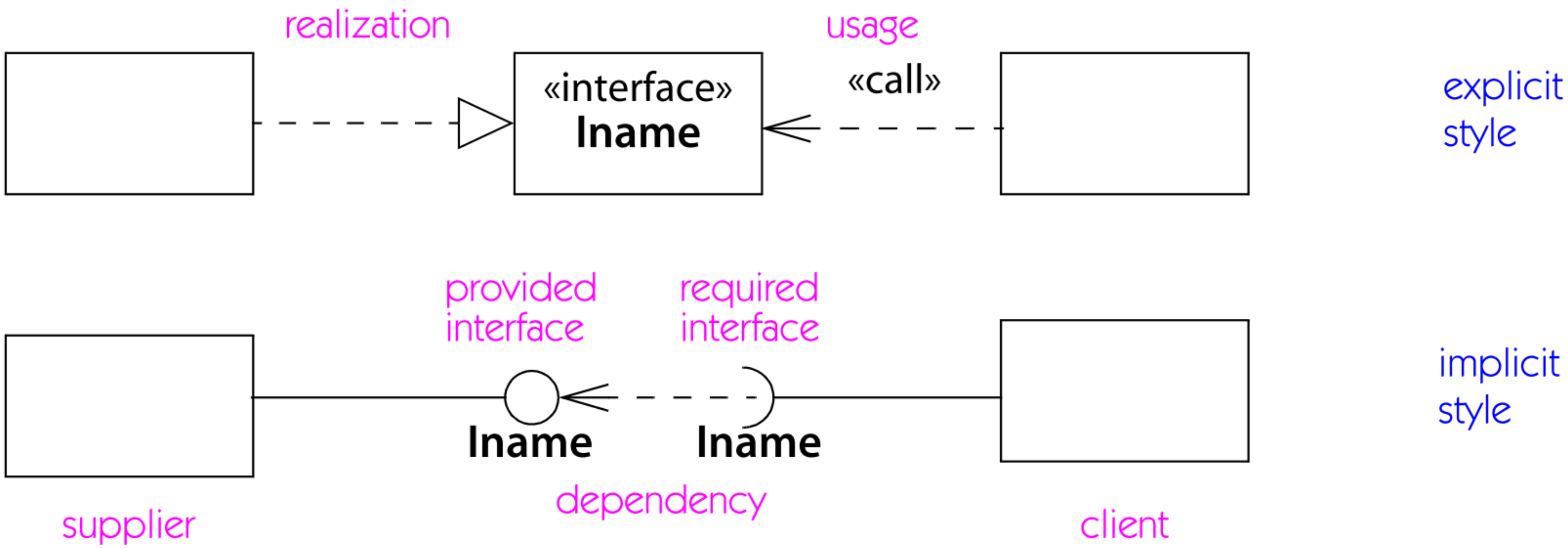
Details: Class



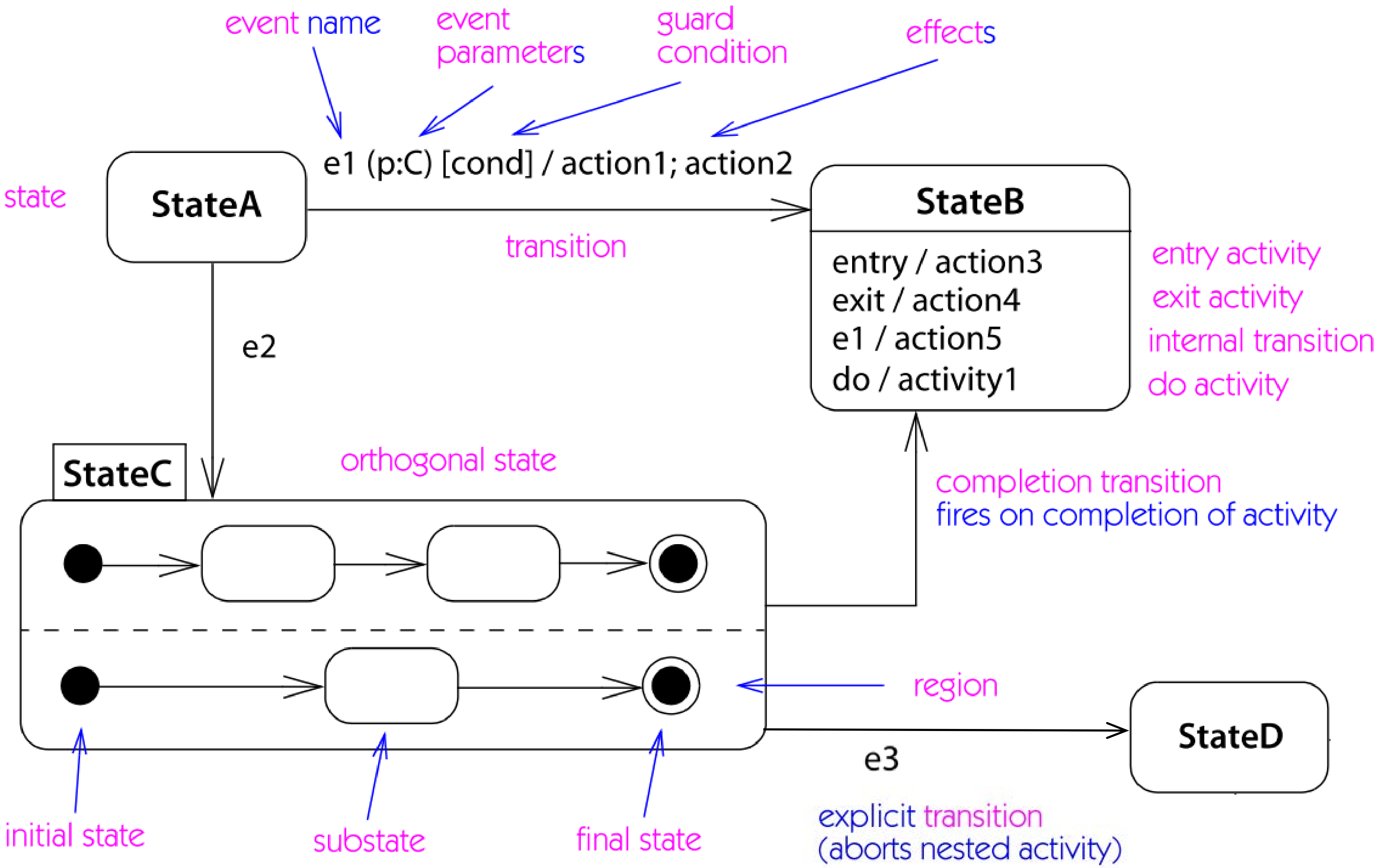
Details: Association



Details: Interfaces



Details: States


event name
event parameters
guard condition
effects
e1 (p:C) [cond] / action1; action2
state
StateA
transition
StateB
entry / action3
exit / action4
e1 / action5
do / activity1
entry activity
exit activity
internal transition
do activity
e2
StateC
orthogonal state
completion transition
fires on completion of activity
region
StateD
e3
initial state
substate
final state
explicit transition
(aborts nested activity)