Modeling

Security Engineering

David Basin
ETH Zurich

Goals for this module

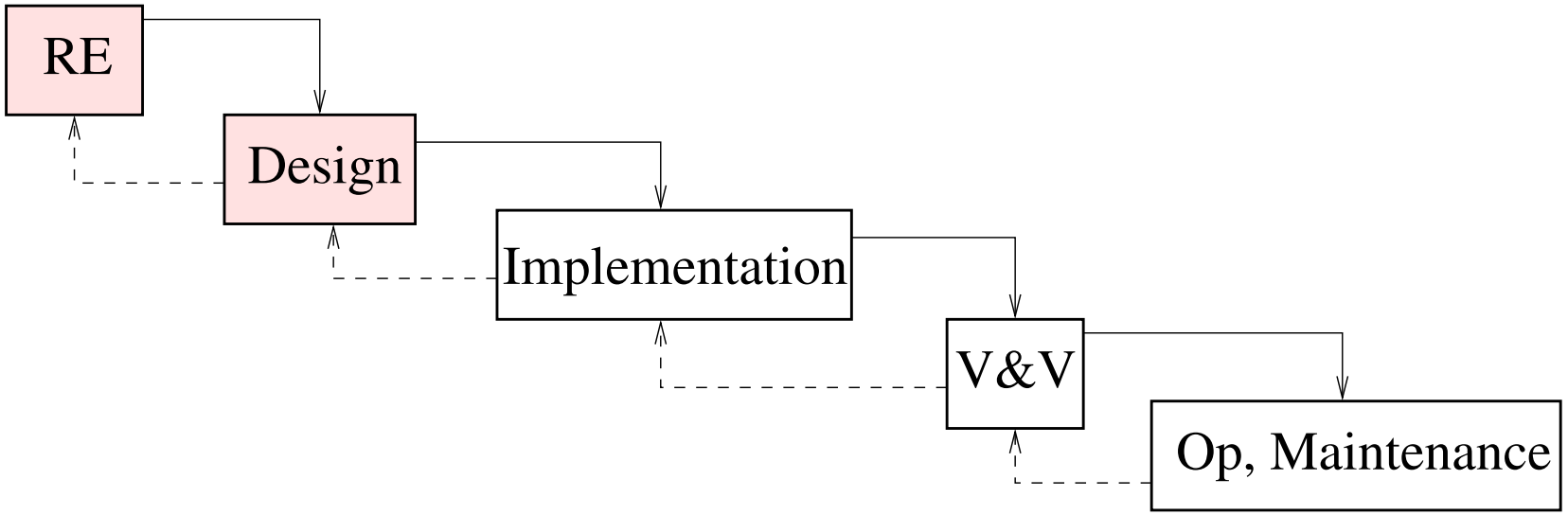
- Motivate use of models in Security Engineering
- Provide some historical perspective
- Introduce the UML modeling foundations
- Introduce a running example

Road map

☞ **Why model?**

- Two simple examples
- The Unified Modeling Language
- Summary

Context: modeling languages and methods

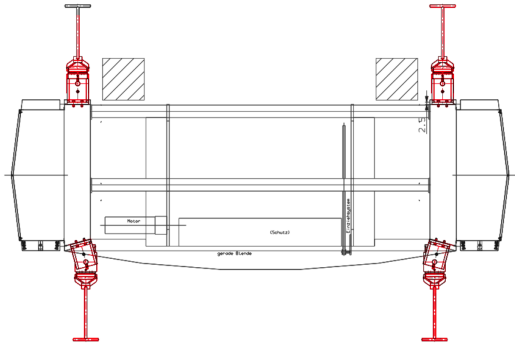


- The **overall goal:** **specify** requirements as **precisely** as possible

  What languages and methods are appropriate for this?

- Lecture emphasizes **Software Engineering** aspects, providing pointers to security

Modeling



- **Definition:** A **model** is a construction or mathematical object that describes a system or its properties

  Example: A construction engineer models buildings and employs static models (e.g., of stress and strains) for their analysis

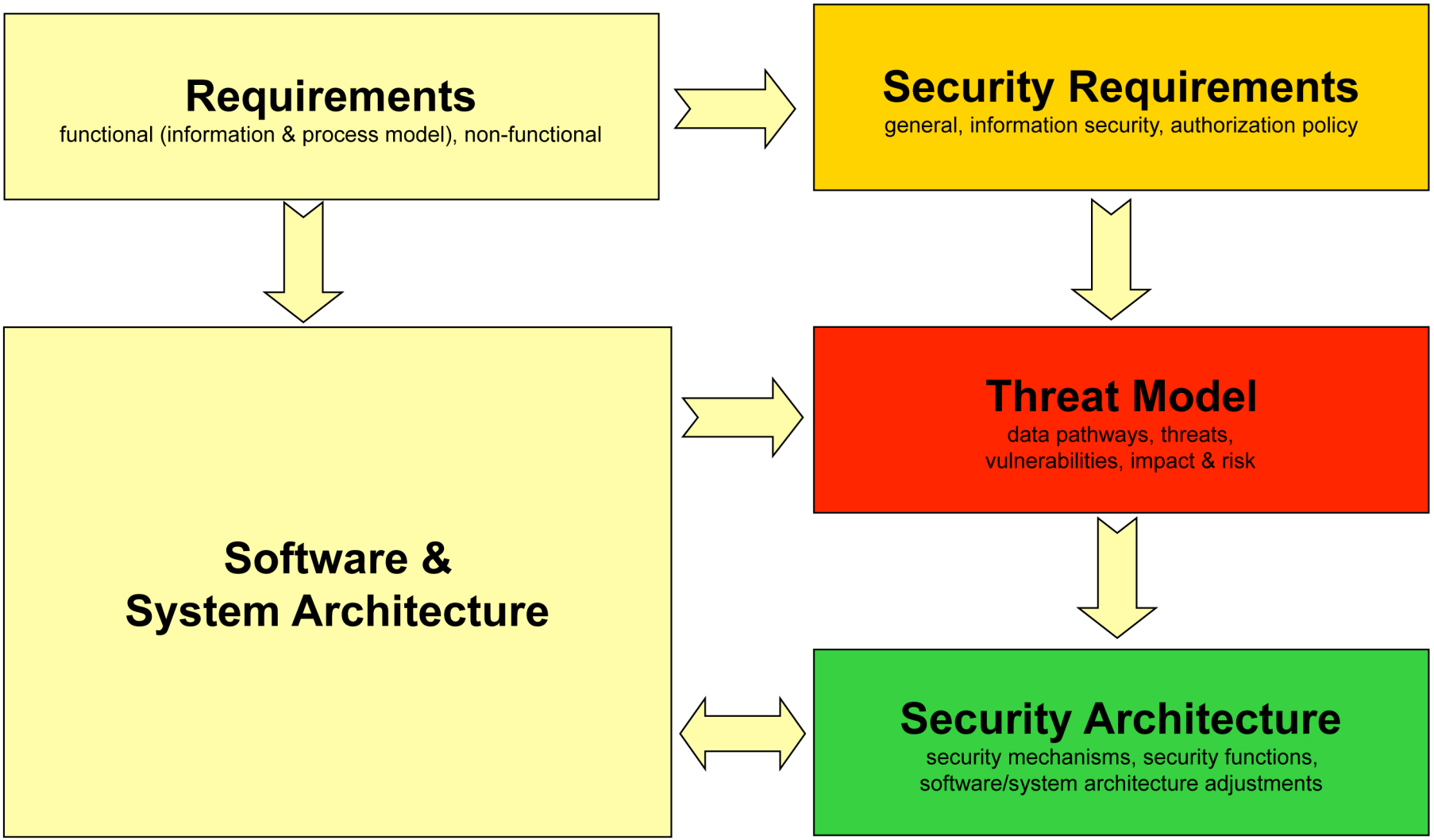
- In computer science, we model systems, their operating environment, and their intended properties

  Models aid requirements engineering, design, and system analysis

- The construction of models is the main focus of the design phase.

  Engineers build models, so should software engineers!

Models in Security Engineering



Which modeling language?
There are hundreds!

- Differences include:

  System view: static, dynamic, functional, object-oriented, . . .

  Abstraction-level: e.g., requirements versus system architecture

  Formality: Informal, semi-formal, formal

  Religion: OO-school (OOA/OOD, OMT, Fusion, UML), algebraic specification, Z/CSP, HOL, . . .

- Examples include:

  Function trees, data-flow diagrams, E/R diagrams, syntax diagrams, data dictionaries, pseudo-code, rules, decision tables, (variants of) automata, petri-nets, class diagrams, MSCs, . . .

- In practice, often combined to model different system views

Road map

- Why model?

☞ **Two simple examples**

  - **Entity-relationship diagrams**
  - **Data-flow diagrams**

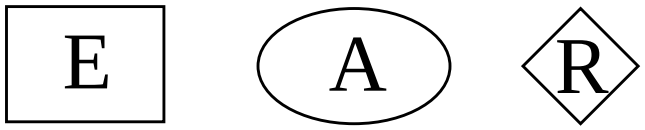
- The Unified Modeling Language

- Summary

Entity/Relationship modeling (E/R)

- Very simple language for data modeling
  - Specifies sets of (similar) data and their relationships
  - Relations are typically stored as tables in a data-base
  - Useful as many systems are data-centric

- Three kinds of objects are visually specified



Entities: sets of individual objects

Attributes: a common property of all objects in an entity set

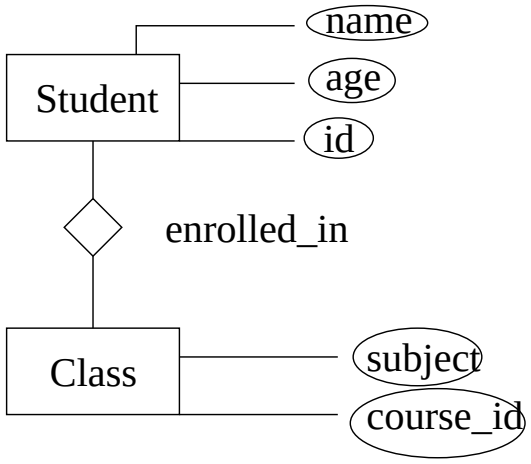
Relations: relationships between entities

E/R example: students & classes

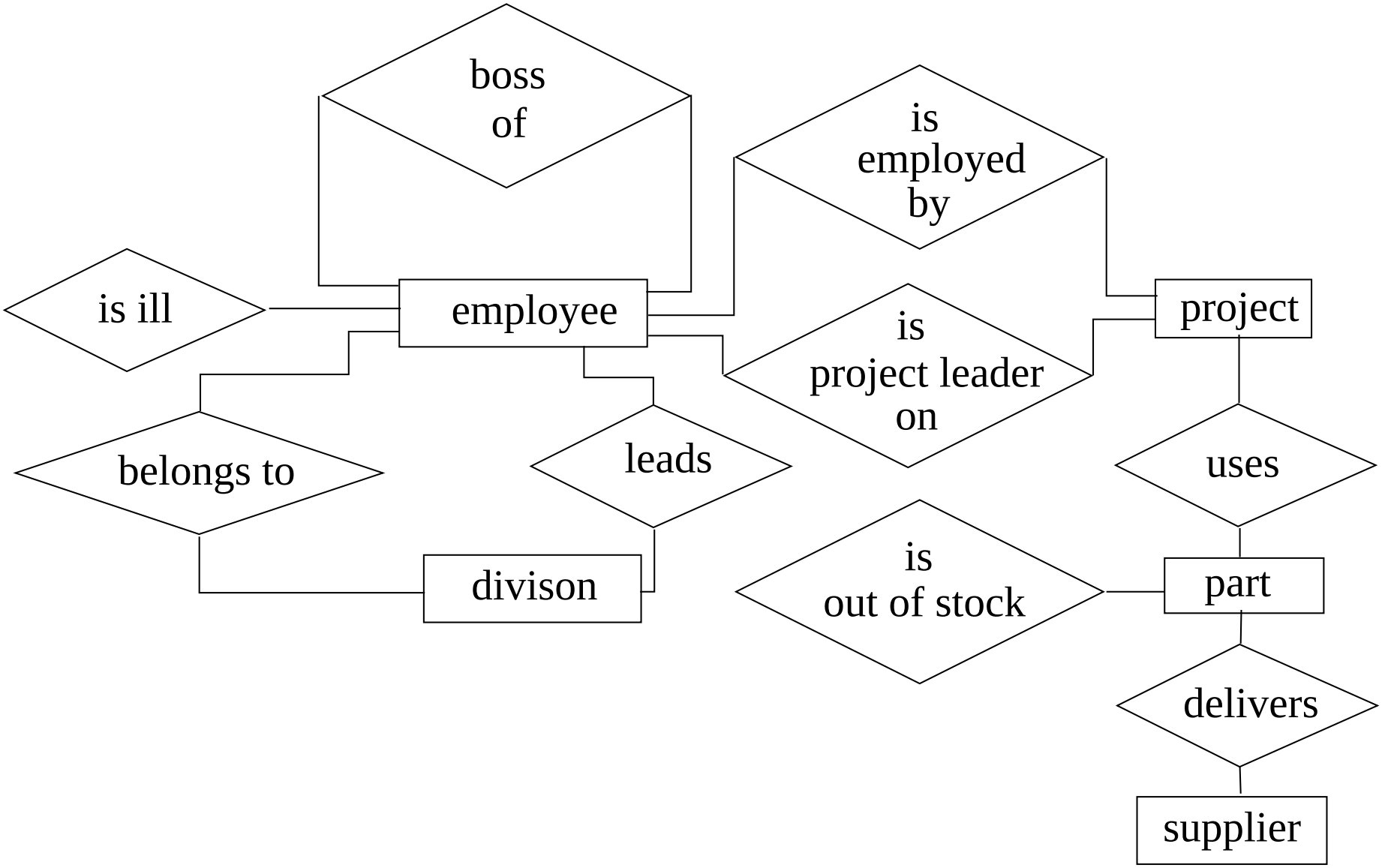
- A *Student* has a *name*, *age*, and *identity*
- A *Class* has a *subject* and *course_id*
- In programming languages: attributes $\mapsto$ basis types, entities $\mapsto$ record types, and relations $\mapsto$ references

$$\begin{aligned} type\ Student &= student(name : string, age : N, id : N) \\ type\ Class &= class(subject : string, course\_id : string) \end{aligned}$$

- Relations graphically represented, e.g., Student *enrolled_in* Class
- In a database, entities also represented by relations. So 3 relations in example.



E/R: a larger example (ignoring attributes)



E/R: pros and cons

++ 3 concepts and pictures $\Longrightarrow$ easy to understand

++ Tool supported and successful in practice

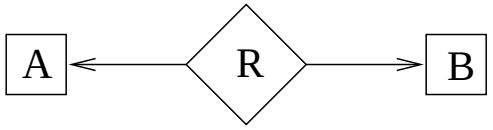
E/R diagrams mapped to relational database schemes

−− Not standardized

Are relations binary or n-ary? OO extensions (e.g. *is_a*)?

−− Weak semantics: only defines database schemas

- Cannot specify properties of relations

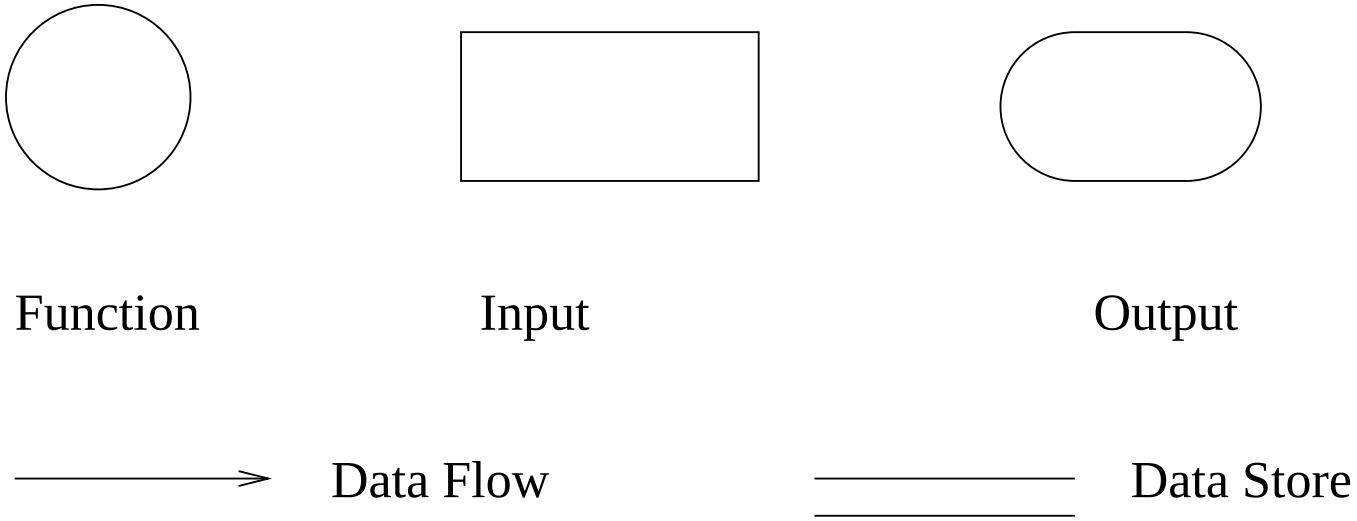


  E.g. R is injective: $\forall x\ x'\ y.(x\ R\ y \wedge x'\ R\ y) \Rightarrow x = x'$
- Says nothing about how data can be modified

For more, see a database course!

Data-flow diagrams

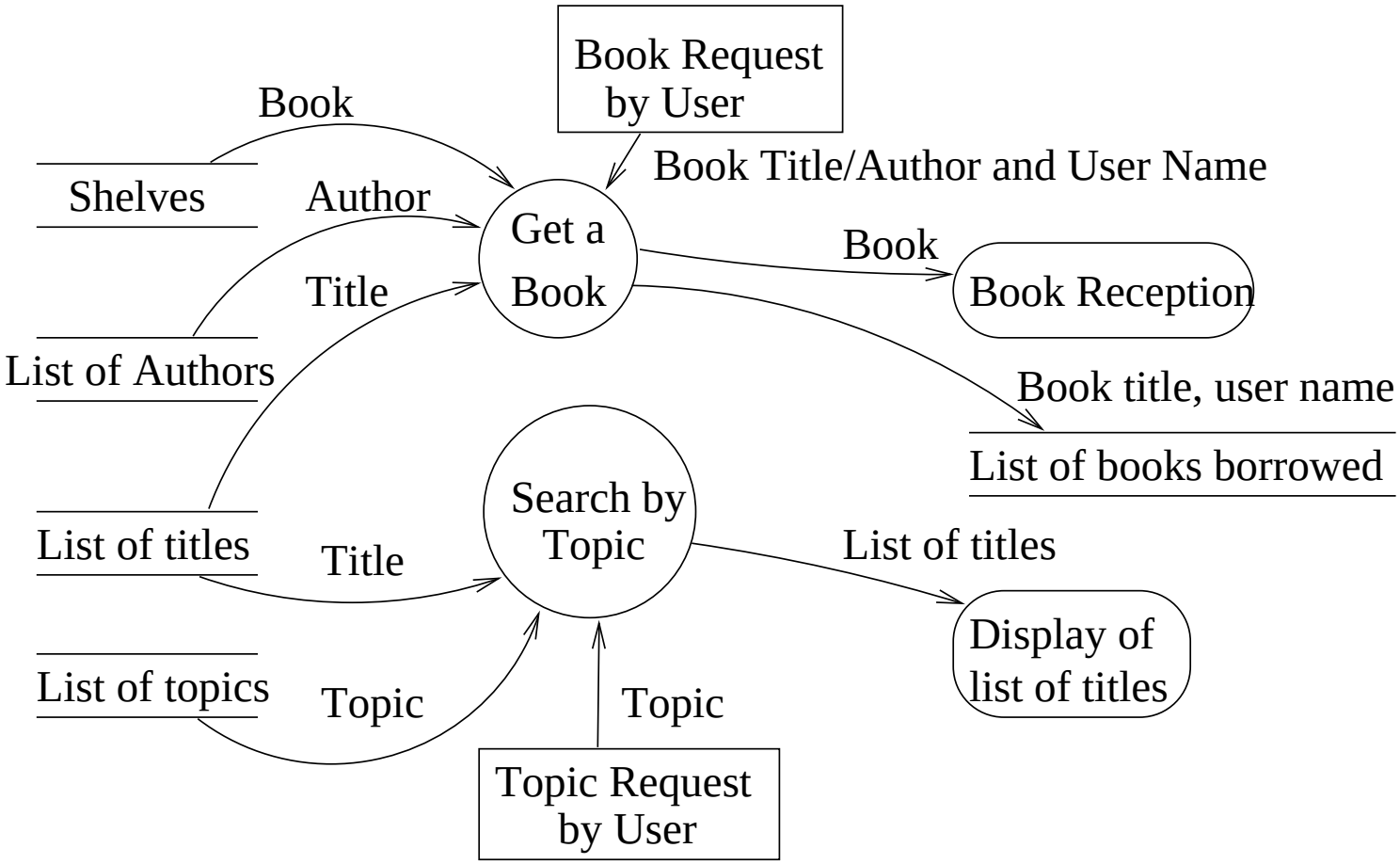
- Graphical specification language for functions and data-flow



- Useful for requirements plan and system definition

  Provides a high-level system description that can be refined later

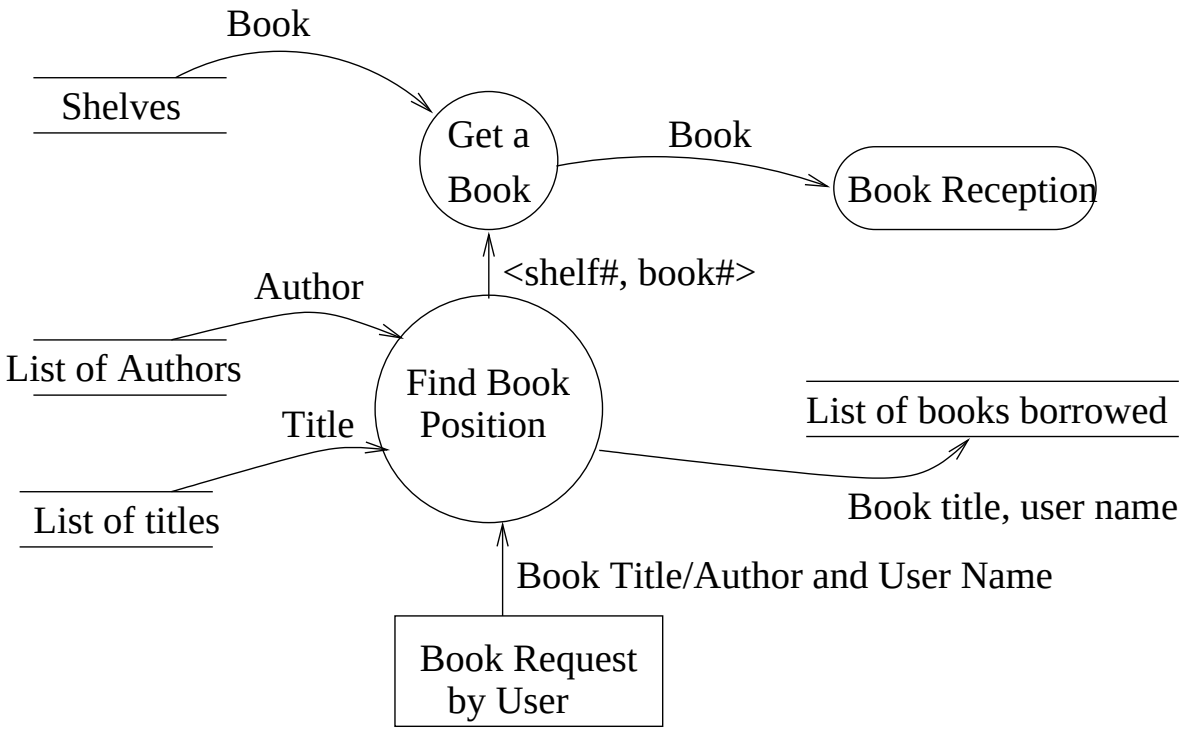
Example: library information system[1]



First approximation. Unspecified how books are found, etc.

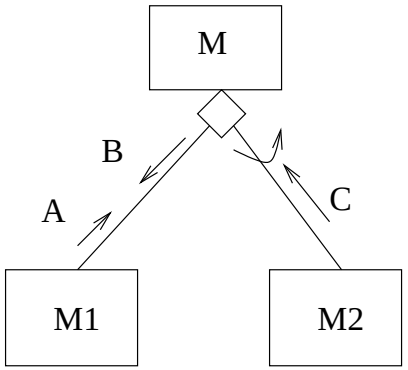
[1] Source: Ian Sommerville, Software Engineering, Addison Wesley.

DFD Refinement

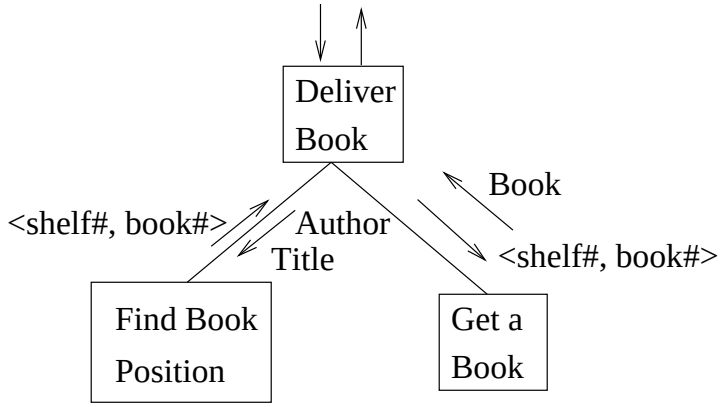


- Informally describes how a book is selected
  - Inexact. Are both title and name needed?
  - Semantics merely suggested by the function names
- Input/output view of system
  - State and transitions not explicitly modeled
  - Control open. Execution scheduling not specified

Hierarchical DFDs development methods



- Hierarchical DFDs yield module structure
  - Procedure M calls either M_1 once or M_2 multiple times
  - M passes B to M_1 and receives A back and M receives C from M_2



- Example: module **order book**.
- Partial description of system dynamics

  Can combine with E/R-diagrams to give a composite system view

- Can be supported by CASE-tools, e.g., for automatic generation of classes or module signatures

Road map

- Why model?
- Two simple examples
- ☞ **The Unified Modeling Language**
  - ☞ **Overview**
    - Diagram types (relevant subset)
  - Summary

A brief history of OO-modeling

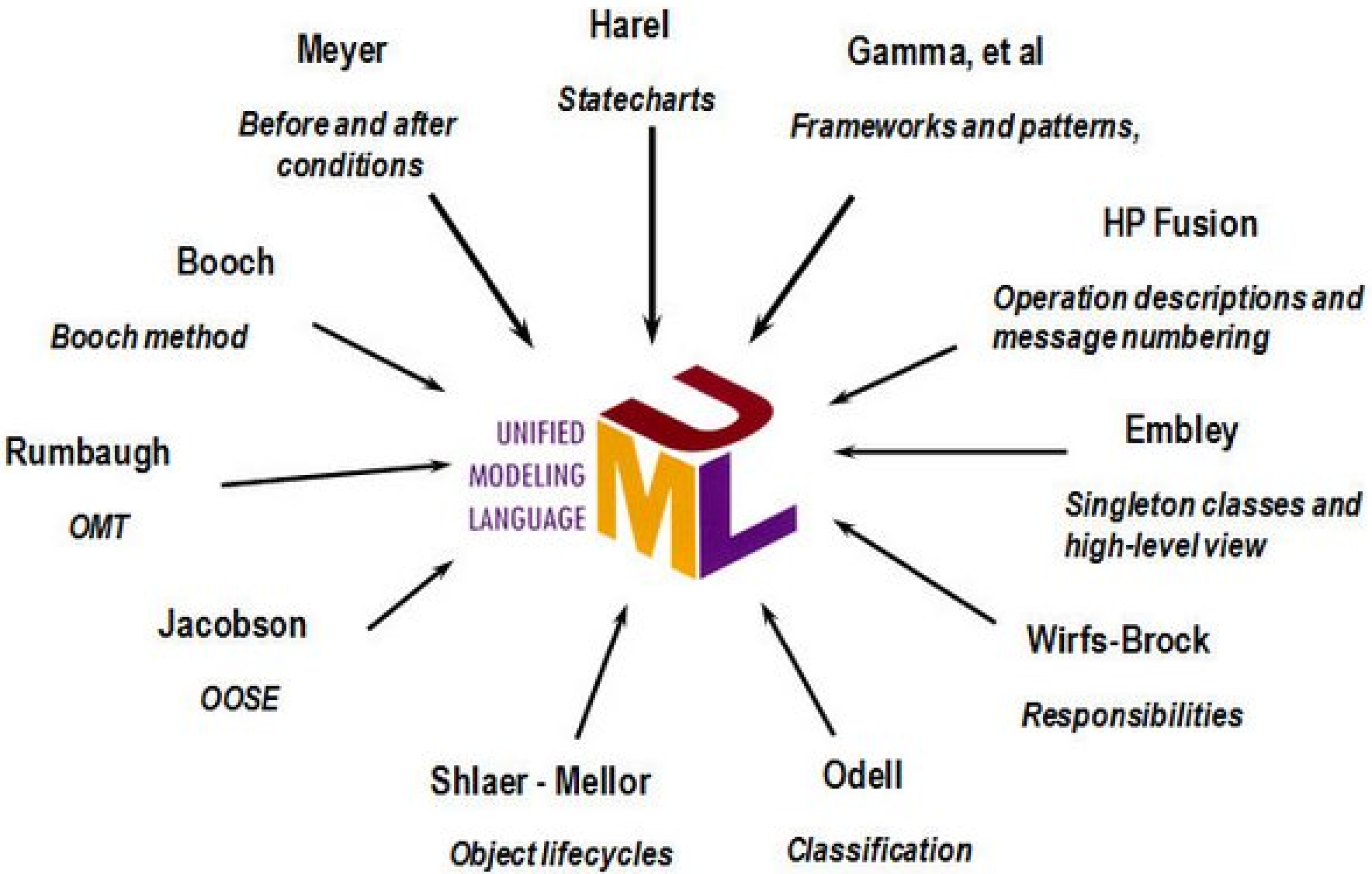
- Proliferation of methods from 1970–1990
- **Method war** between the different languages and methods
- **Campaign** of the 3 Amigos



  - Grady Booch's OOD
  - James Rumbaugh's OMT
  - Ivar Jacobson's OOSE (based on "Use Cases") and Objectory
- Standardization in the **Unified Modeling Language**, 1995–???

  Has an associated process and development method (RUP)
- UML has wide acceptance, but is certainly not the last word!

UML Contributions (in more detail)



UML overview

What? 14 languages for modeling different views of systems

- **Static models** describe system parts and their relationships
- **Dynamic models** describe the system's (temporal) behavior

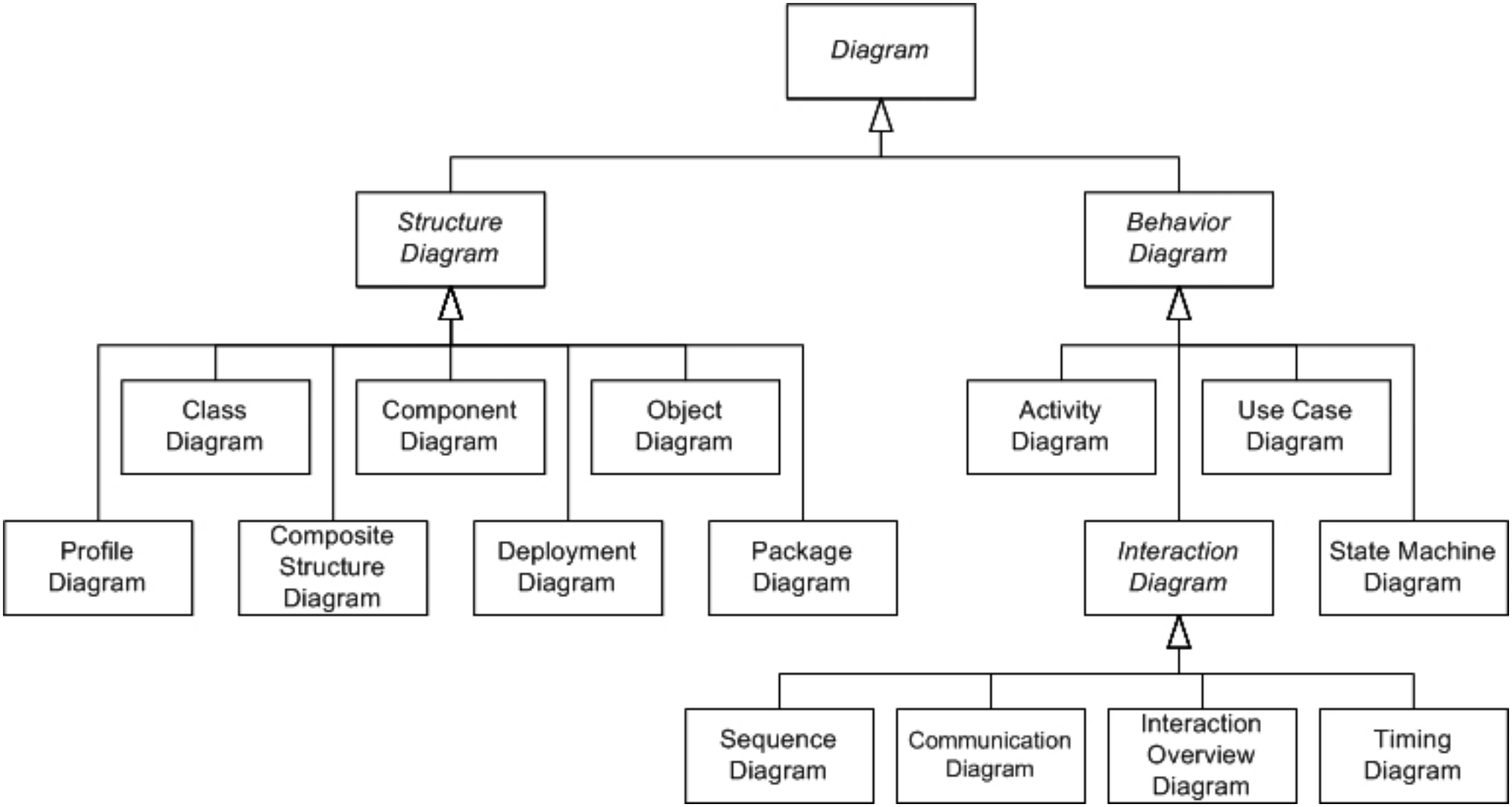
Why?

- **De facto** standard
- Combines various good ideas and is more-or-less intuitive.
- Tool support

Why not?

- Semantics? What do UML models actually **mean**?
- Support for analysis is (currently) weak
- Complicated/cryptic bits due to committee-driven development
- It keeps changing! (Most slides based on v2.2)

UML diagram types



Road map

- Why model?
- Two simple examples
- ☞ **The Unified Modeling Language**
  - Overview
  - ☞ **Diagram types (relevant subset)**
- Summary

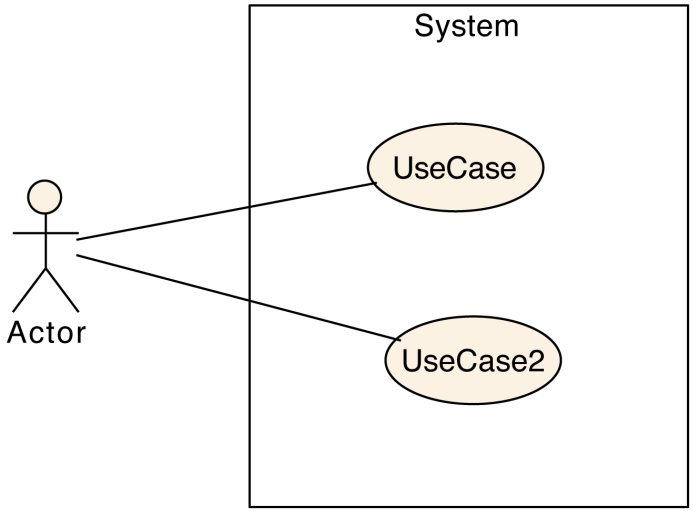
Use Cases
Model scenarios for using system

- Specifies who can do what with the system
- Key concepts

  System: the system under construction

  Actor: users (roles) and other systems that may interact with the system

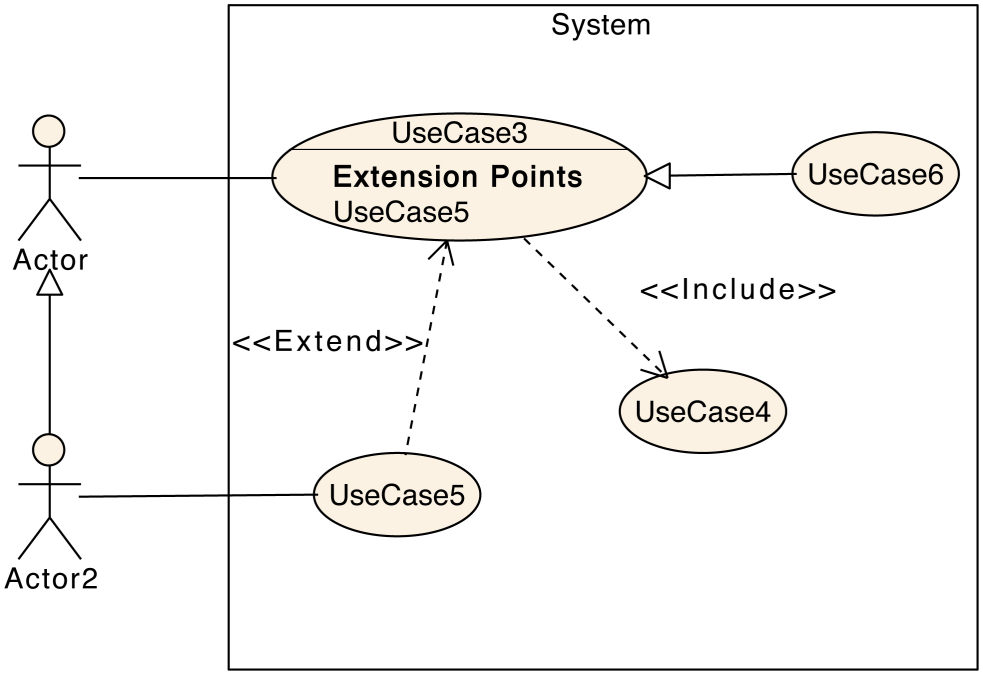
  Use case: specifies a required system behavior according to actors' needs



- Can specify associated behavior in different ways
  - **textually**, e.g., as a sequence of steps, or
  - using an **activity diagram**

Use Cases: more details

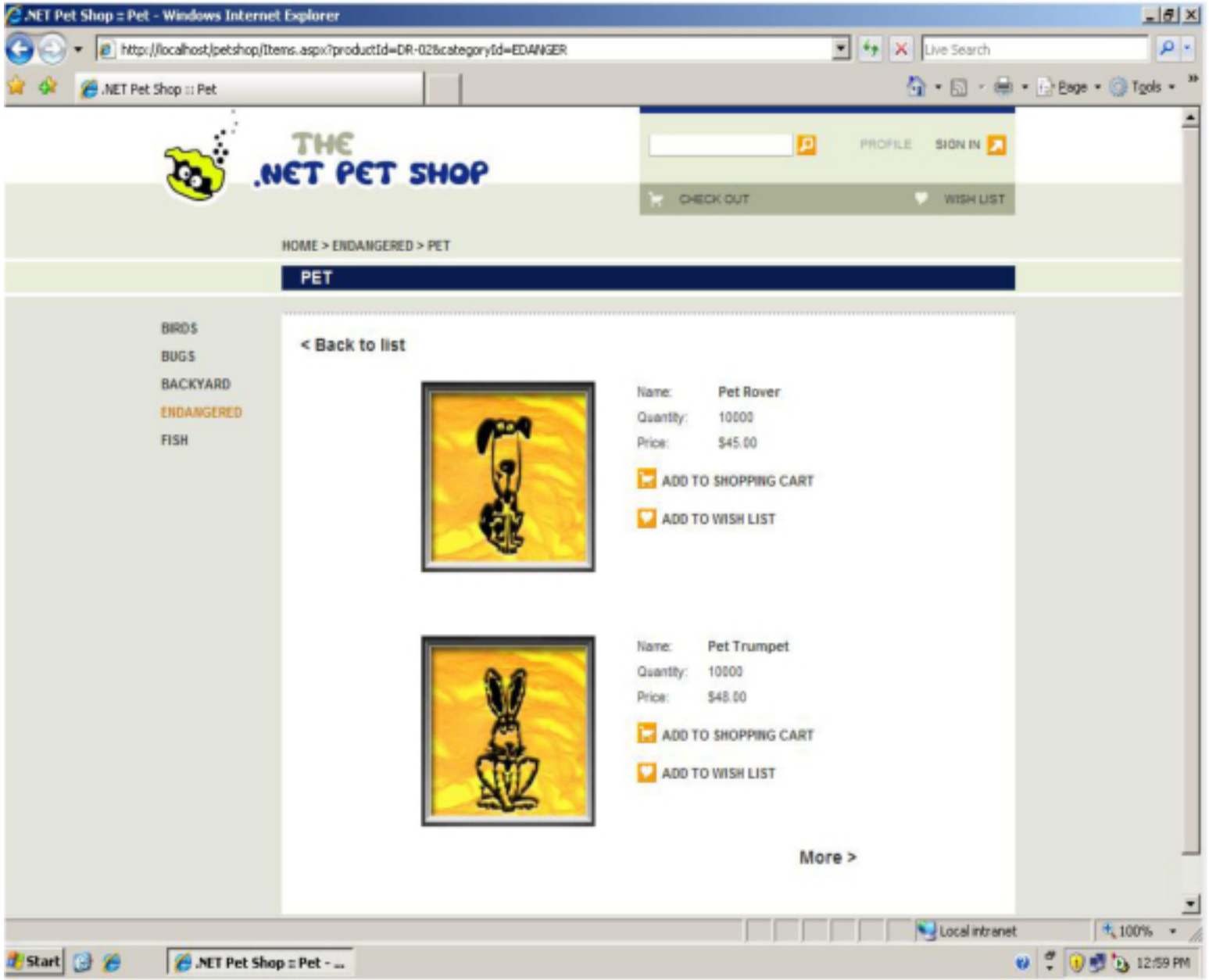
- Relations between actors
  - Generalization/specialization
- Relations between use cases
  - Generalization/specialization
  - Extend (one use case extends functionality of another)
  - Include



- Relations flexible. Can add your own using new stereotypes.

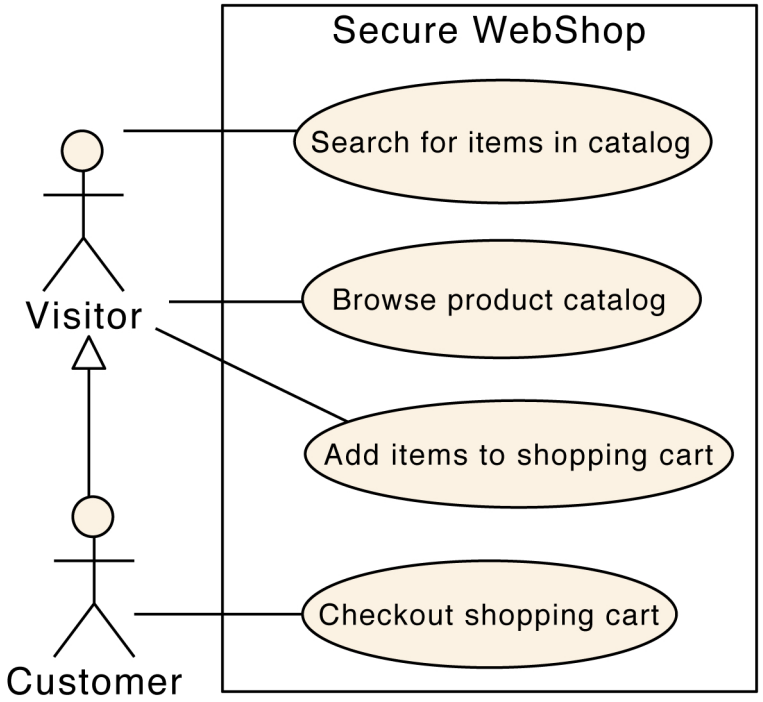
  Is this good or bad?

Running example



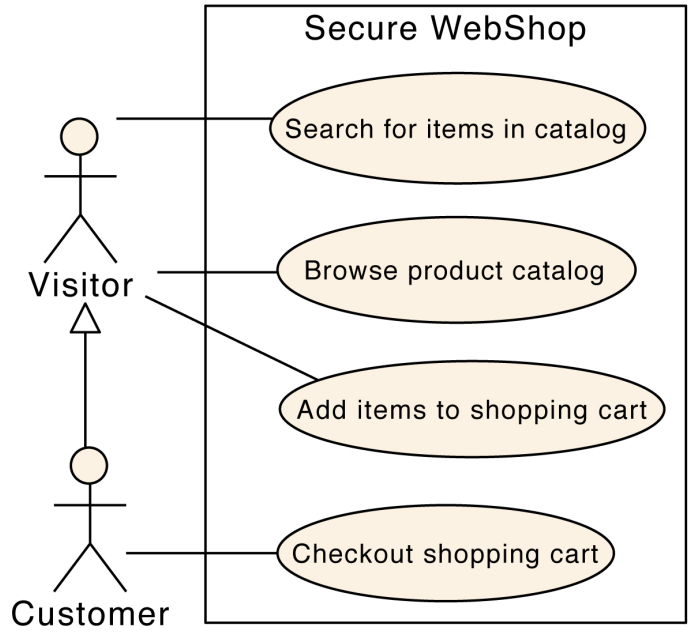
Use Cases for Web Shop

- **Scope:** Implementation of a single system, the **secure webshop**
- **Visitors** may search and browse the site and add items to shopping cart
- **Customers** may checkout



Use Cases — relevance for security engineering

- Authorization policy: Actors are candidate roles, contributing to authorization policy (roles)
- System boundary represents system interface i.e, something to be protected!
  - Interface details should be visible in deployment diagrams (coming up)
- Foundation to derive misuses cases



Activity diagrams

Depict the sequence and conditions for coordinating activities

Key concepts:

Action: a single step, not further decomposed

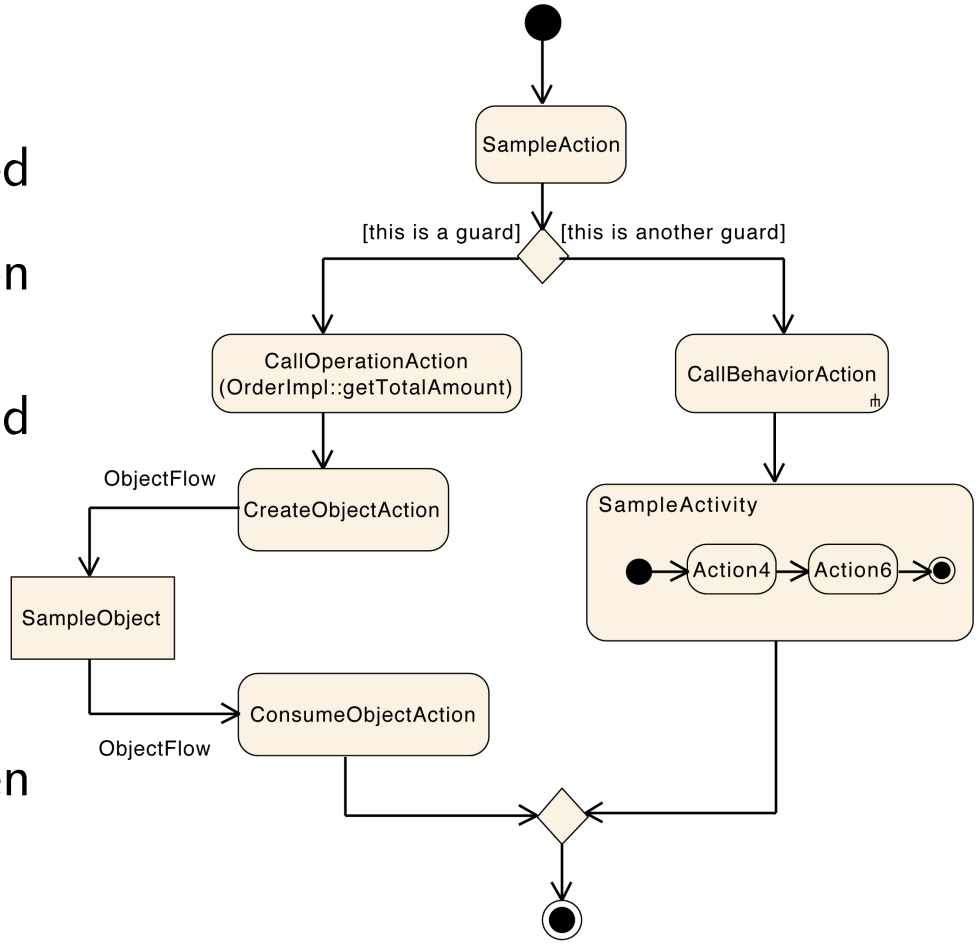
E.g., send a message or trigger an operation

Activity: encapsulates a flow of activities and actions; may be hierarchically structured.

Control Flow: edges ordering activities

Decision: a control node choosing between outgoing flows based on guards

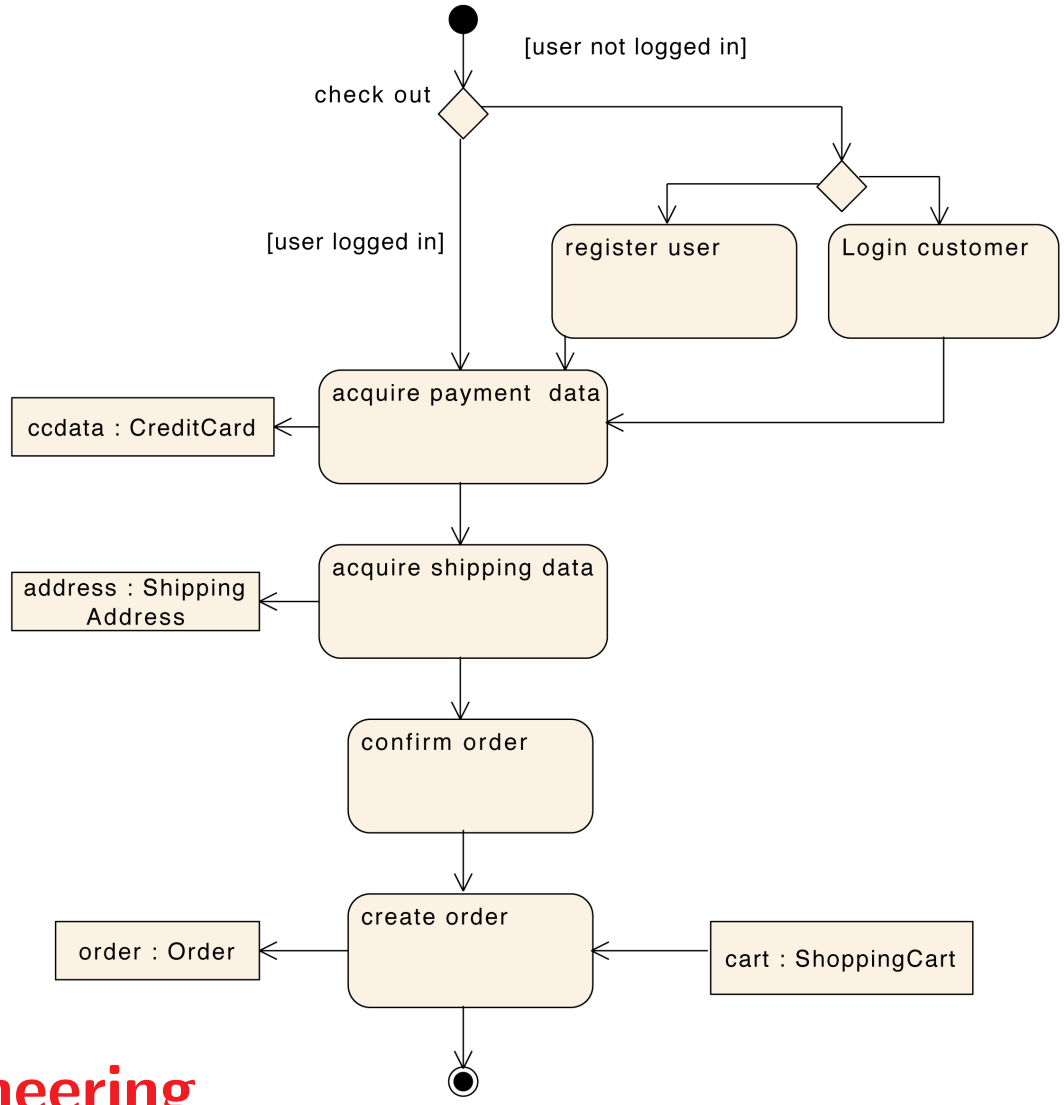
Object Flow: an edge that has objects or data passing along it



Activities: webshop checkout process

- User must login before checkout and non-customers must register
- Data is acquired or produced during process:

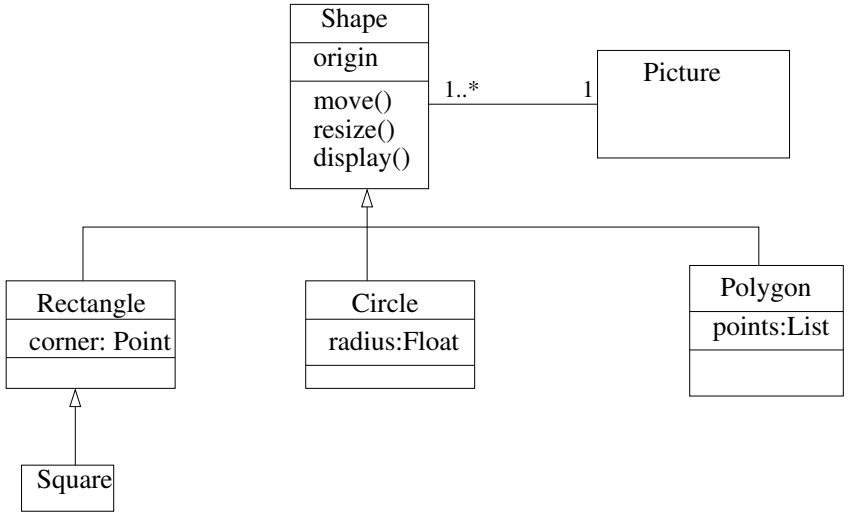
  Credit Card Data, Address, Order, Shopping Cart



- **Relevance for Security Engineering**
  - Understand application flows, participants, and system parts
  - Understand data flows and system dependencies
  - Contributes to authorization policy (process and service access)

Class diagrams

Shows a system's classes, their attributes, and their relationships



Key concepts:

Class: A class describes a set of objects that share the same specifications of features, constraints, and semantics

Attribute: a structural feature of a class

Operation: a behavioral feature of a class that specifies the name, type, parameters, and any constraints for invocation

Association: specifies a semantic relationship between typed instances

Generalization: relates a specific classifier to a more general classifier

Class diagrams in more detail

- Models the **static view** of a system

  A **class diagram** describes the **kind of objects** in a system and their different **static relationships**.

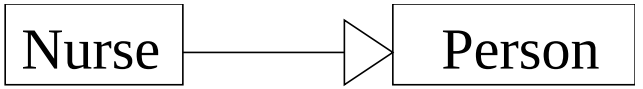
- Kinds of relationships include

  Associations: between objects of a class

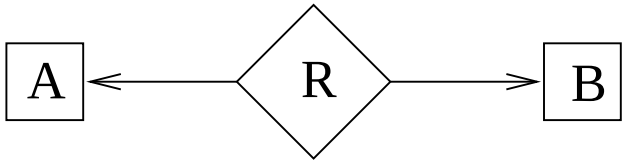
  Example: Customers can rent videos

  Inheritance: Between classes themselves

  Example: Nurses are people

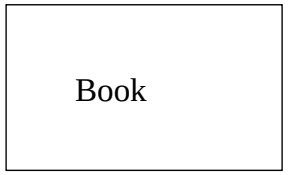


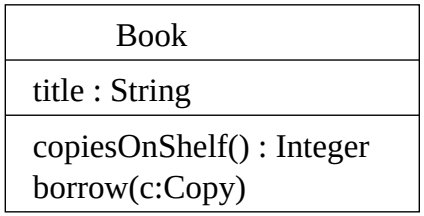
- Generalize E/R diagrams



Representing classes

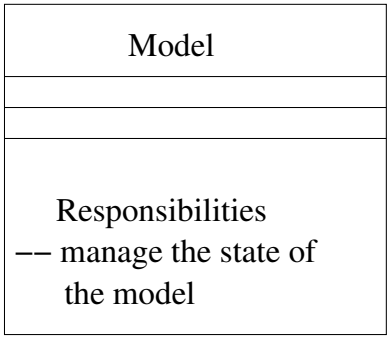
- Icon: a square, with optional attributes and operations

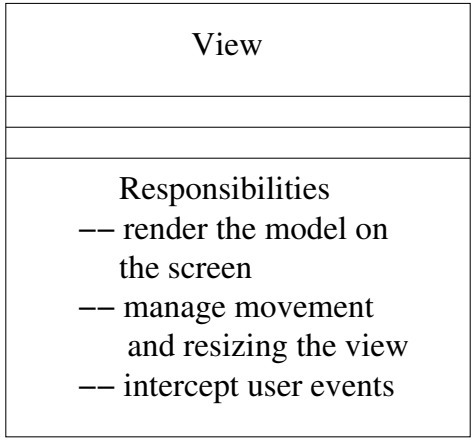


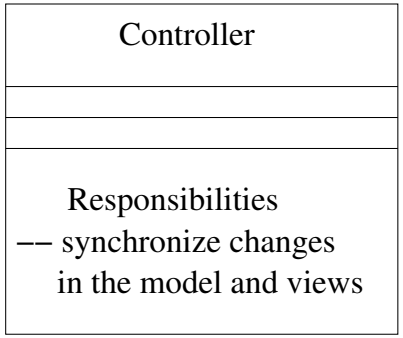


  - **Attributes** define the state (data values) of the object
  - **Operations** (or **methods**) define how objects affect each other

- One can also specify **responsibilities**

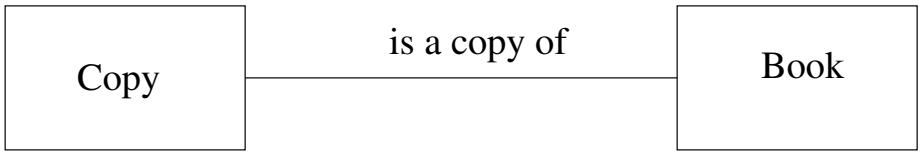






Associations

- Relates objects and other instances of a system
- Semantics: a relation like E/R models

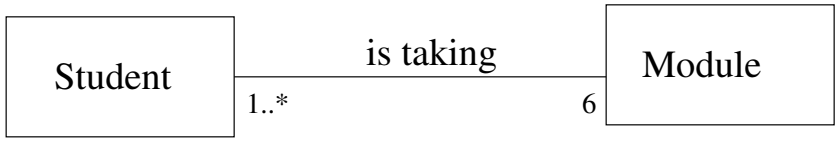


- Examples:
  - X **is a copy of** the book Y
  - Fred **borrows** copy 17 of Book XYZ
  - An object of class A **sends a message** to an object of class B
  - An object of class A **creates** an object of class B

Associations can be annotated

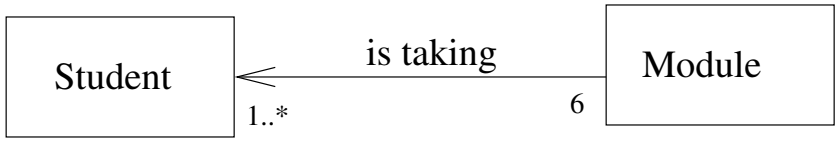
Name: e.g. **is a copy of**

Multiplicity: a number n, or a range $n..m$, or an arbitrary number $*$



- Each student takes 6 courses and each course has ≥ 1 student
- **Semantics**: constrains the relation

Navigation: a line can be directed with an arrow

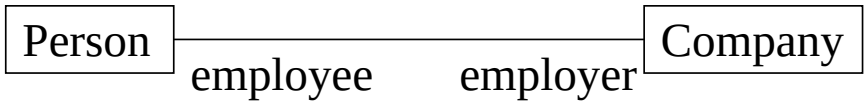


- Module objects can send students messages, but not vice versa.
- **Semantics**: relation can be queried in only one direction

Can omit details in the early modeling phases. Result is ambiguous.

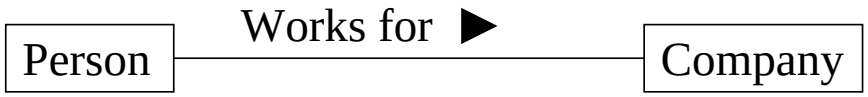
Annotation (cont.)

Role: describes the roles played by objects in the association



- Makes relationships easier to read and understand
- No semantic consequences

Directed names: describes the direction that a name should be read



- Independent of navigation (direction)
- Also no semantic consequences

Associations: aggregation and composition

Notation for certain kinds of frequently occurring associations

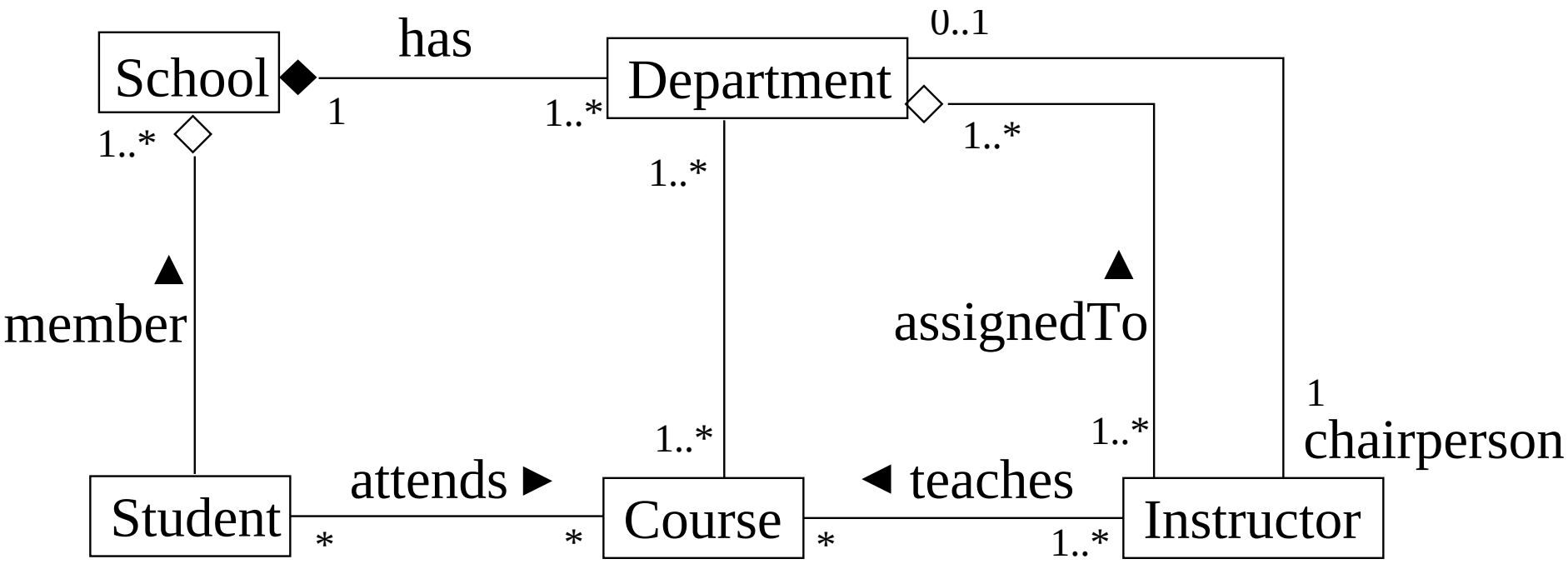
Aggregation: whole-parts relation with independent existence

- Diamond marks the "whole"
- Typically no name. Implicitly "is a part of"

Composition: aggregation where the "part" has no independent existence and is part of only one whole

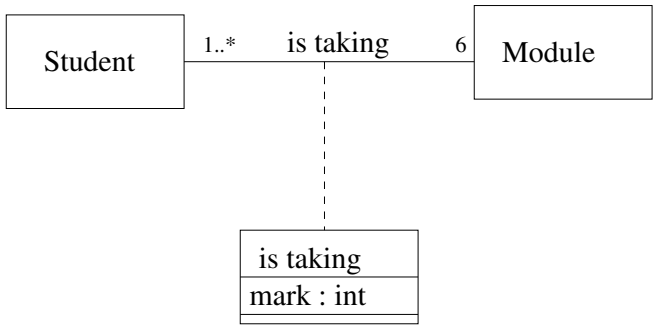
When the whole is deleted, so are its parts

An example

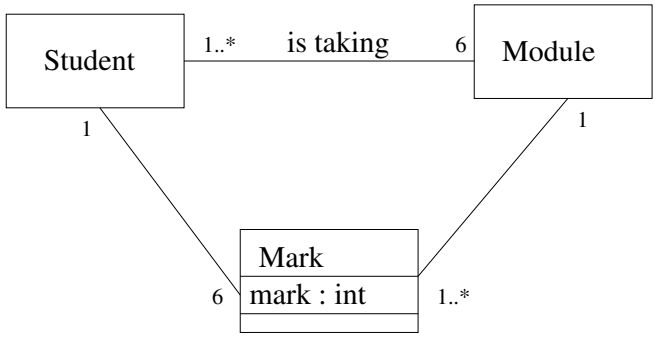

School
has
Department
0..1
1
1..*
1..*
1..*
member
assignedTo
1
chairperson
1..*
1..*
attends ▶
◀ teaches
Student
Course
Instructor
*
*
*
1..*

Association classes

- An association can itself have properties
- An **association class** models things with both association and class properties



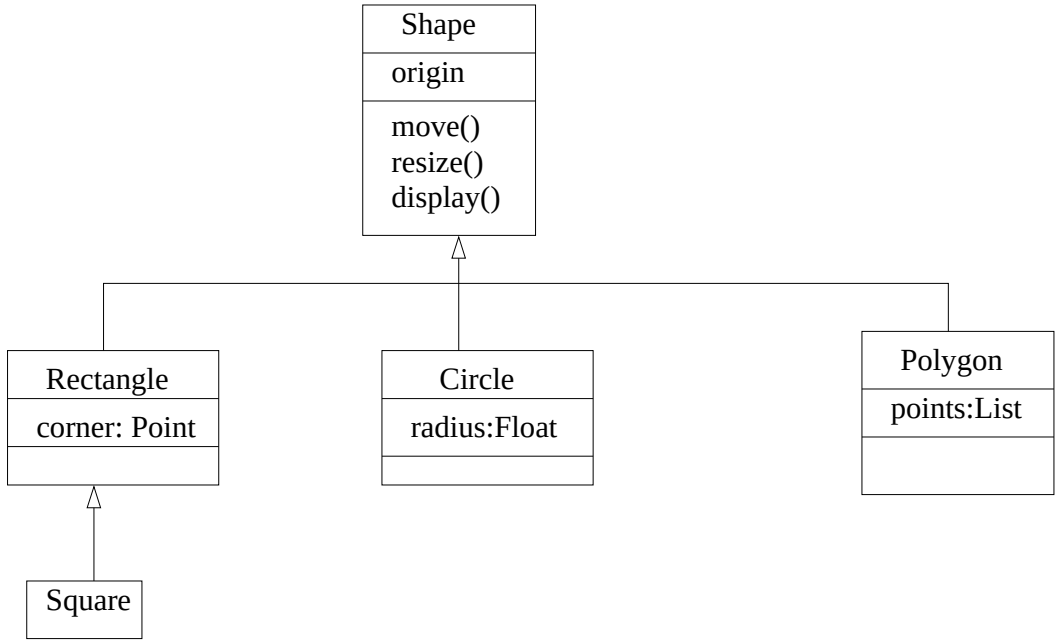
- An association class can be replaced with multiple associations



Question: What semantics do these have? Are they equivalent?

Generalization

- Relation between a general thing (**superclass**) and a specific thing (**subclass**). Sometimes called **is-a**.
- Example:

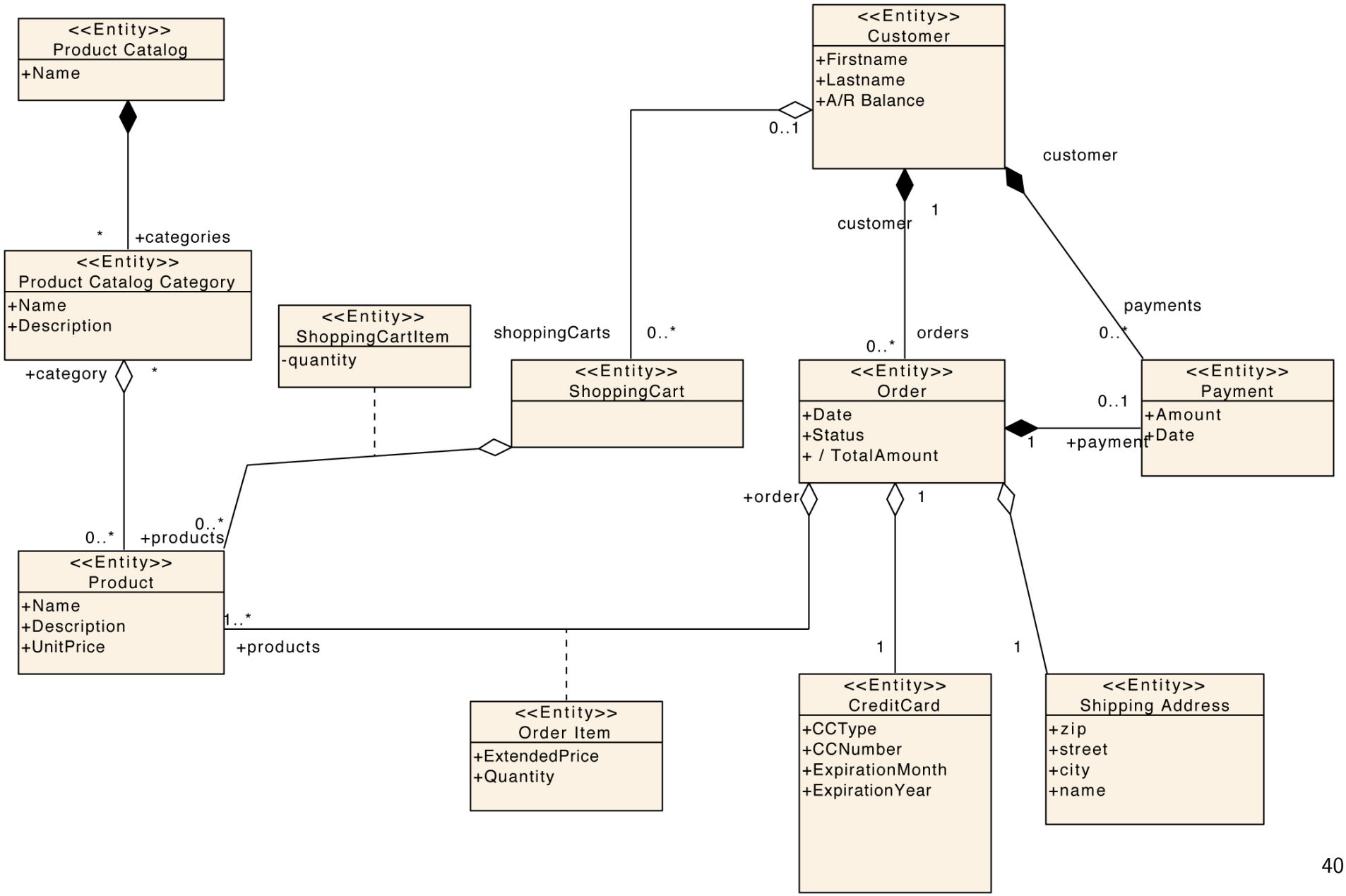


- Semantically: an object of a subclass can be substituted for a superclass object

Usage scenarios

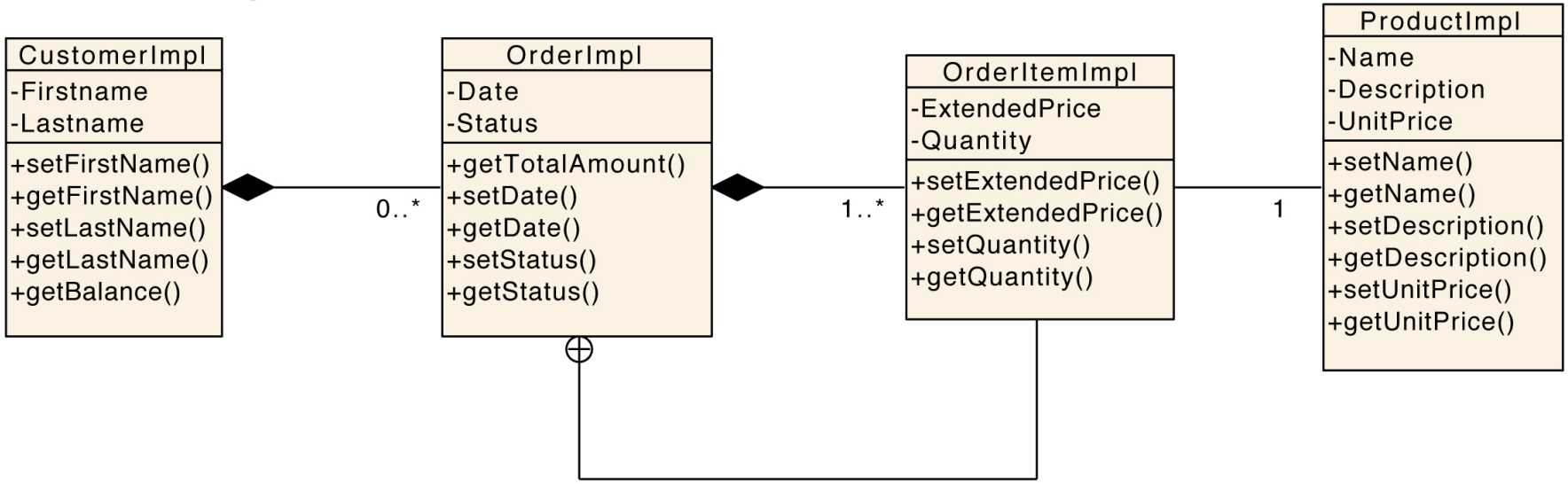
- In requirements analysis: entity/domain model
- In software design: for class design and database design
- Relationship between both abstraction levels specified by "refinement relations"

Webshop — domain model

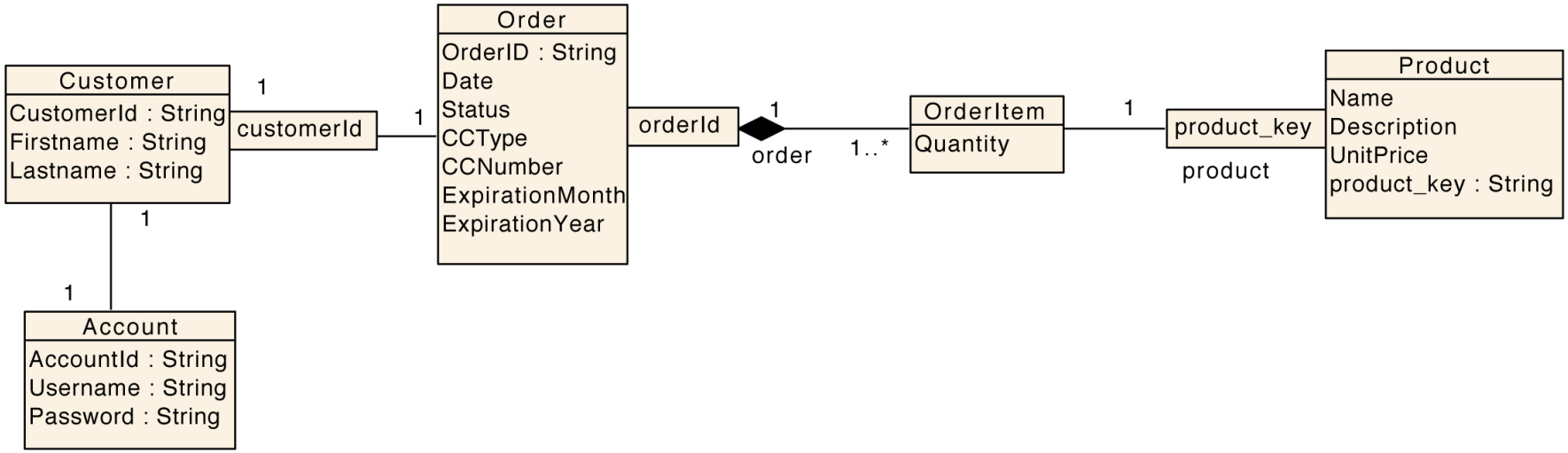


Webshop — design models

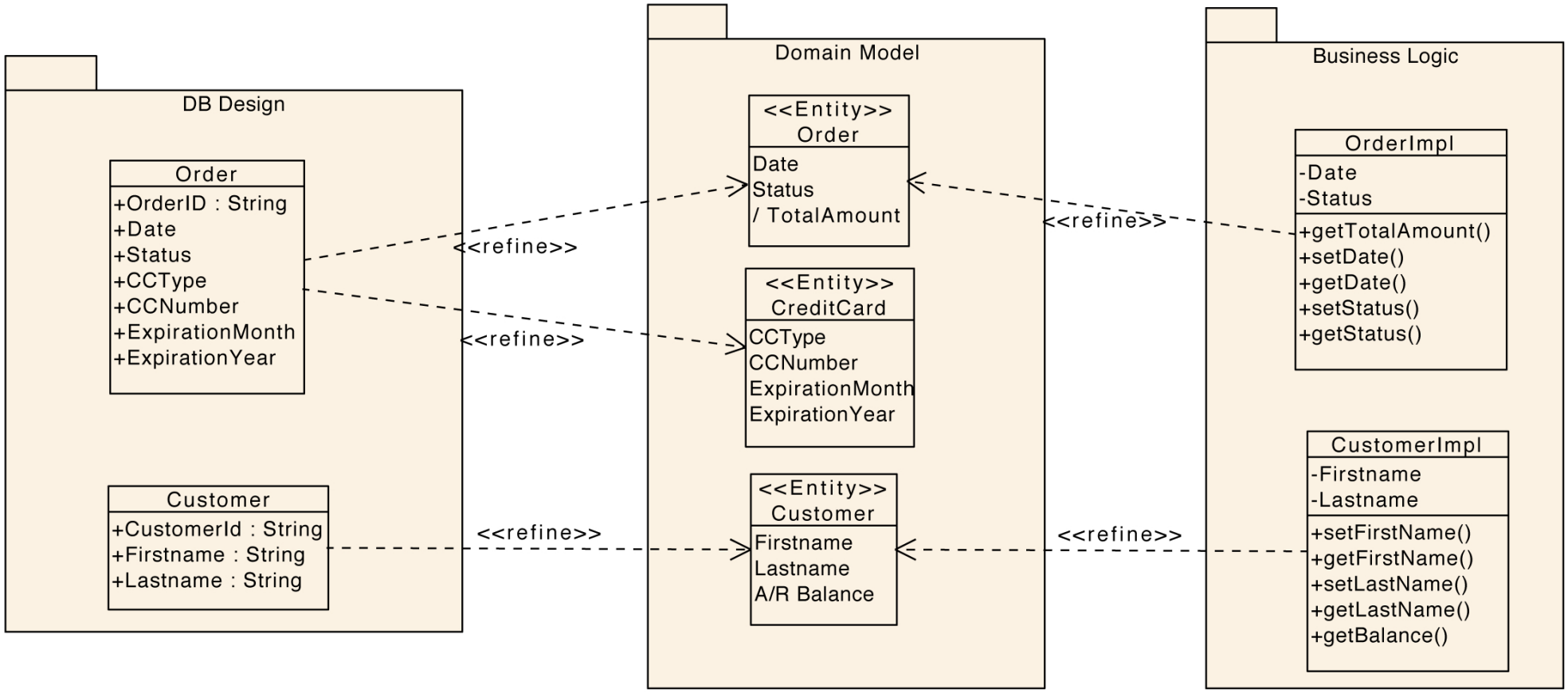
Class design



Database design



Webshop — class refinements



- Tracing of the refinement of (security) requirements through all phases and models of the development process

Relevance for Security Engineering

- Class diagrams specify the assets to be protected at the level of objects, attributes, and methods

  These assets (and respectively requirements) are refined to components and systems using other diagrams

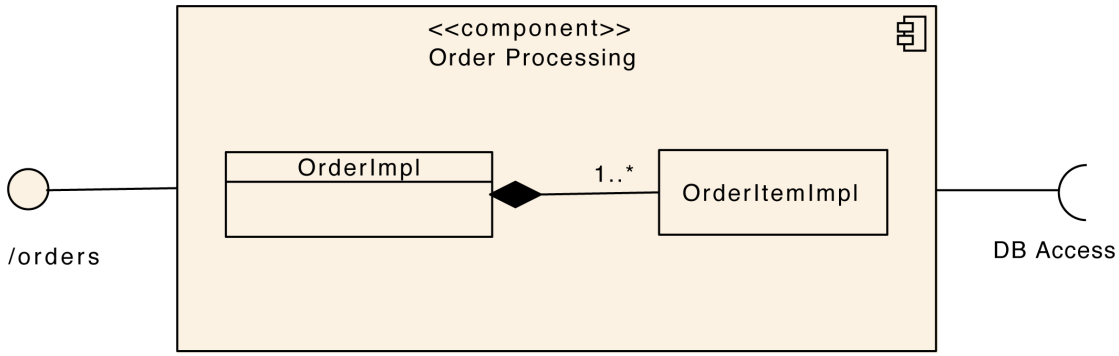
- Foundation for specifying information security policy

  Assets have CIA requirements

- Contribute to authorization policy (data access)

Component Diagrams

Model system's components



Key concepts:

Component:

- Modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment
- Behavior typically implemented by one or more classes or sub-components

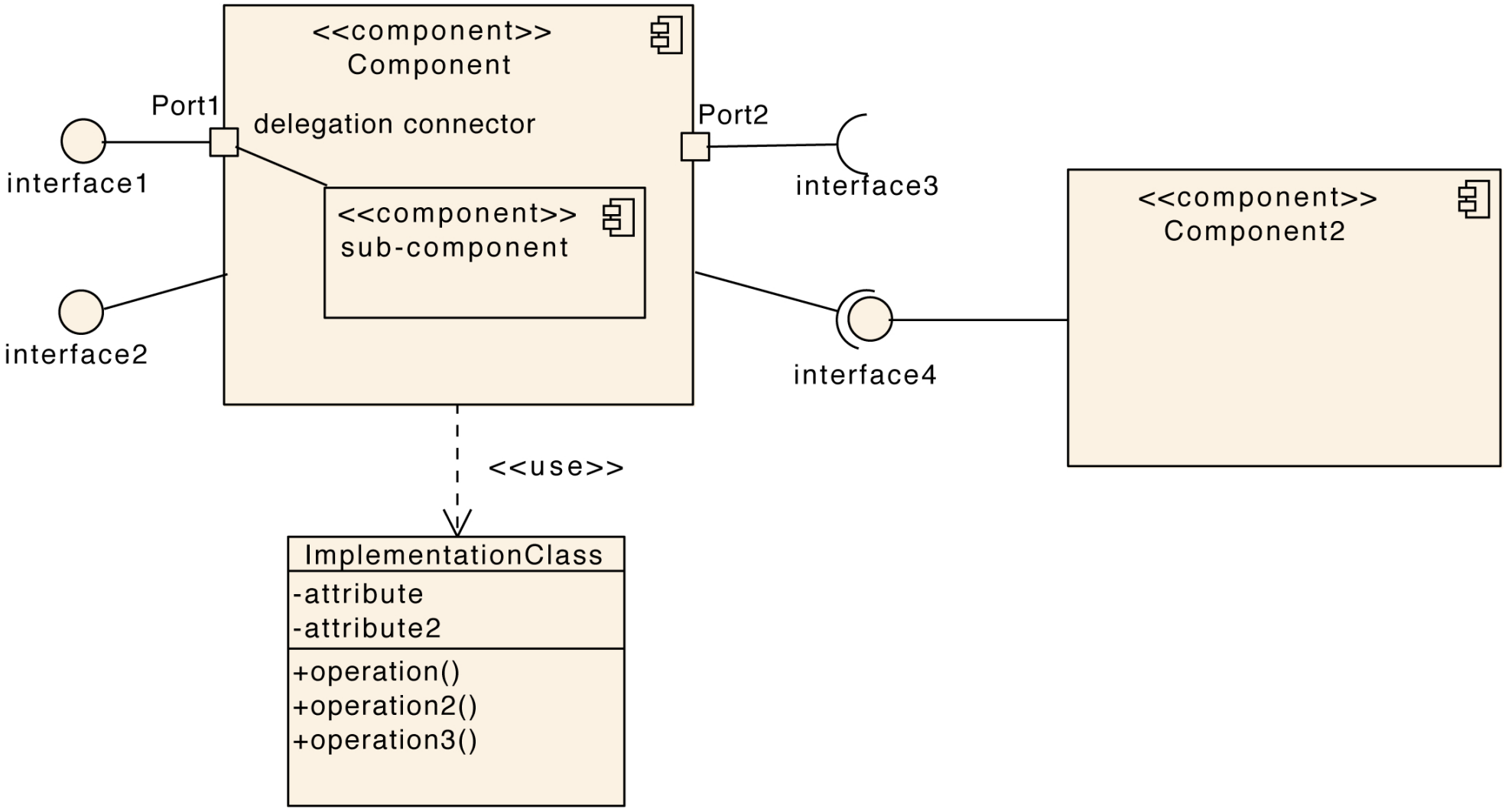
Provided interfaces: interfaces implemented and exposed by a component

Required interfaces: interfaces required to implement component's behavior

An assembly connector links an interface provided by one component to an interface required by another component

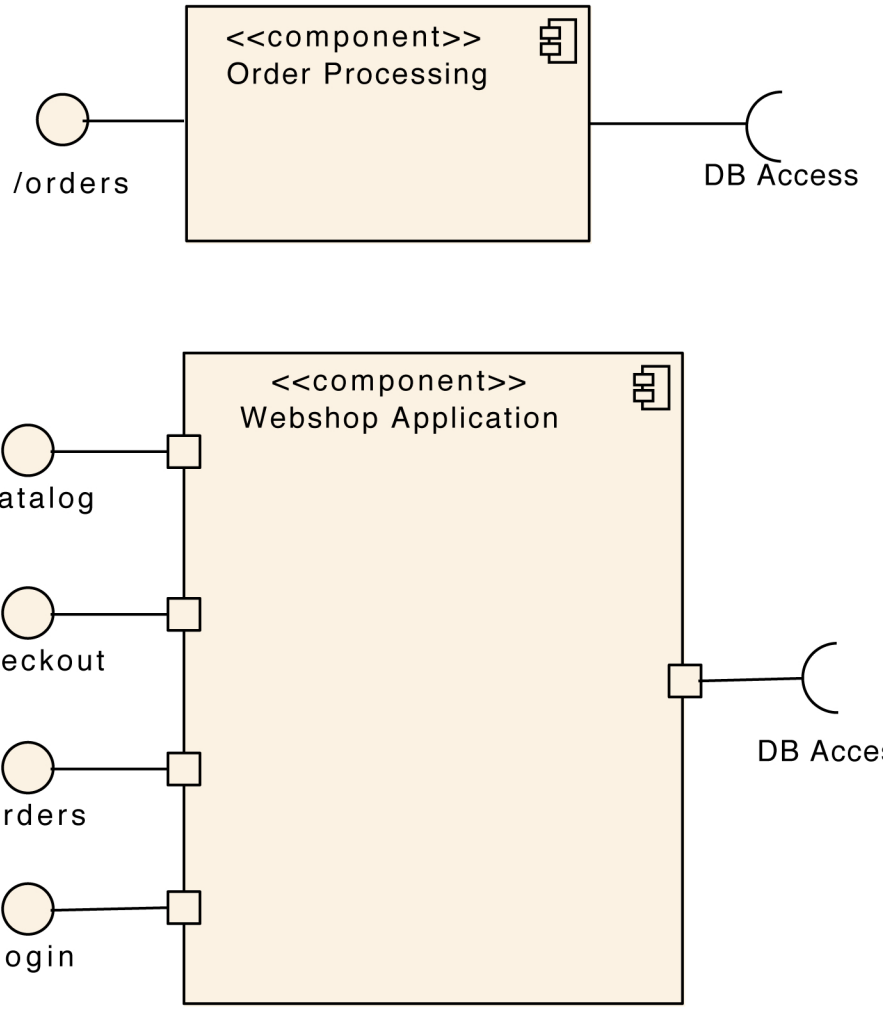
Ports: named sets of provided and required interfaces. Models how interfaces relate to internal parts.

Component diagram example



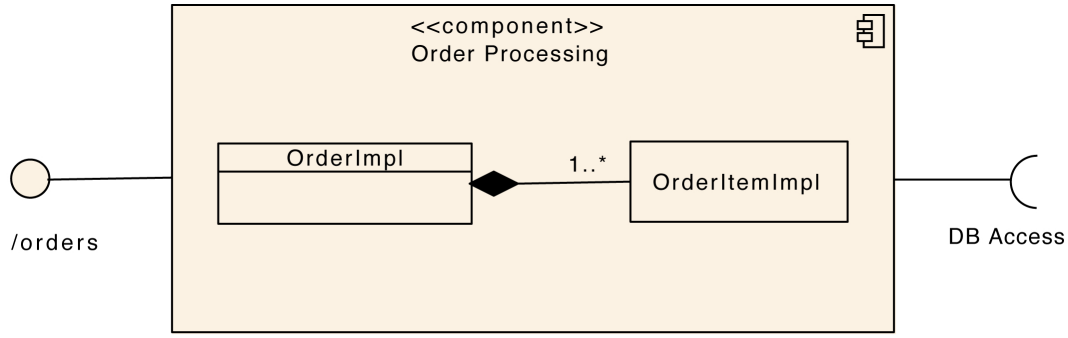
Webshop component diagram (I)

- Order Processing is a Servlet implementing requests
- Webshop Application is the web application offering all user interaction endpoints
- Both components depend on the DB access interface

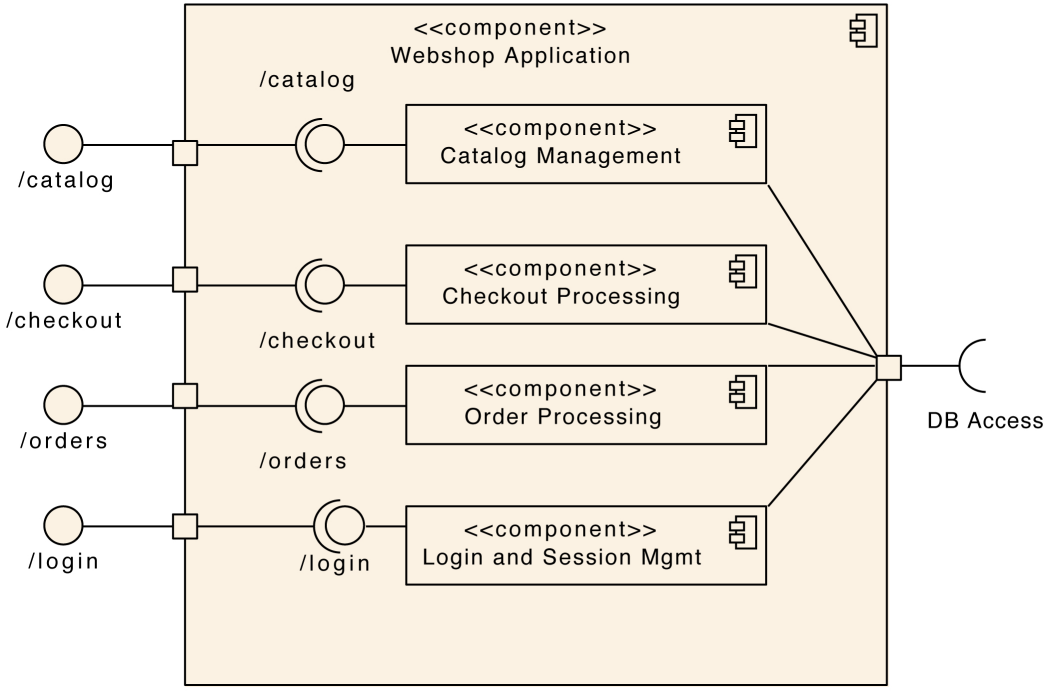


Webshop component diagram (II)

- Order processing is realized by several classes



- Webshop Application is composed from several components

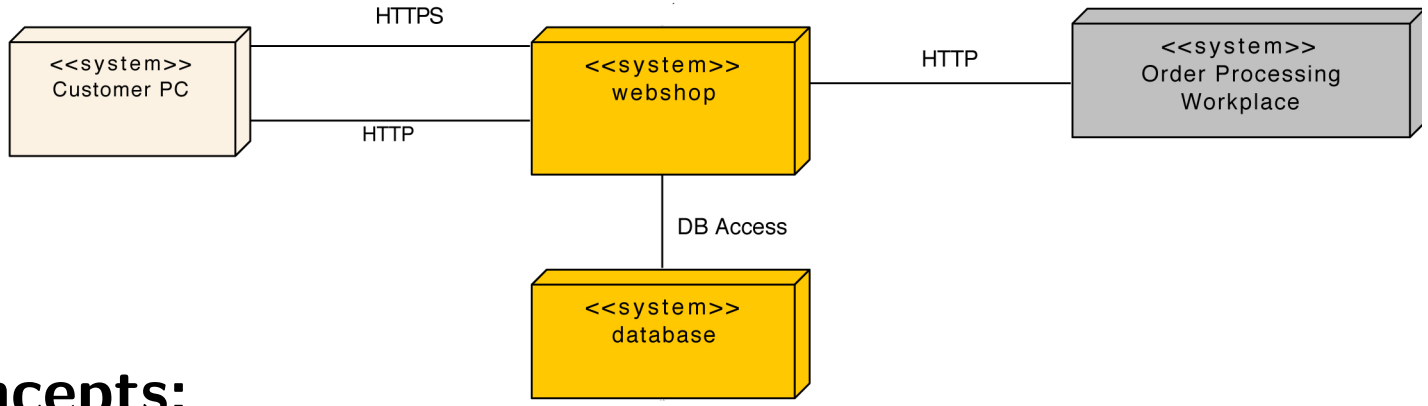


Relevance for Security Engineering

- Component diagrams link design and deployment models
- Which data (assets) are processed in which components?
- What are the security sensitive applications and why?
- Contribute to risk analysis

Deployment Diagrams

Model the execution architecture of a system



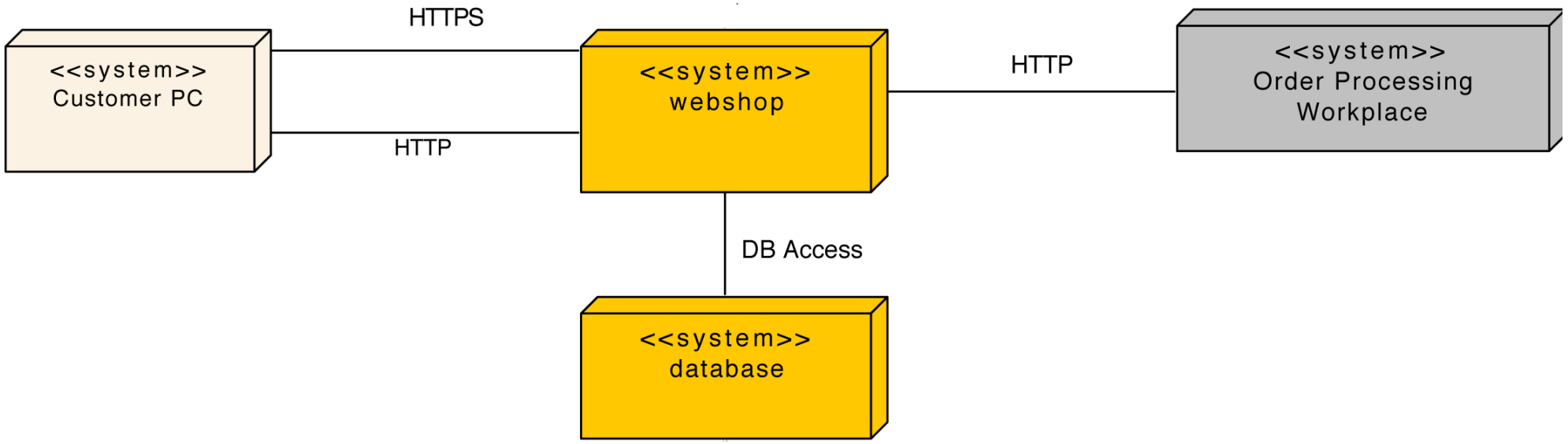
Key concepts:

A node is a computational resource where components are deployed for execution by way of artifacts

A communication path is an interconnection between nodes to exchange messages, typically used to represent network connections

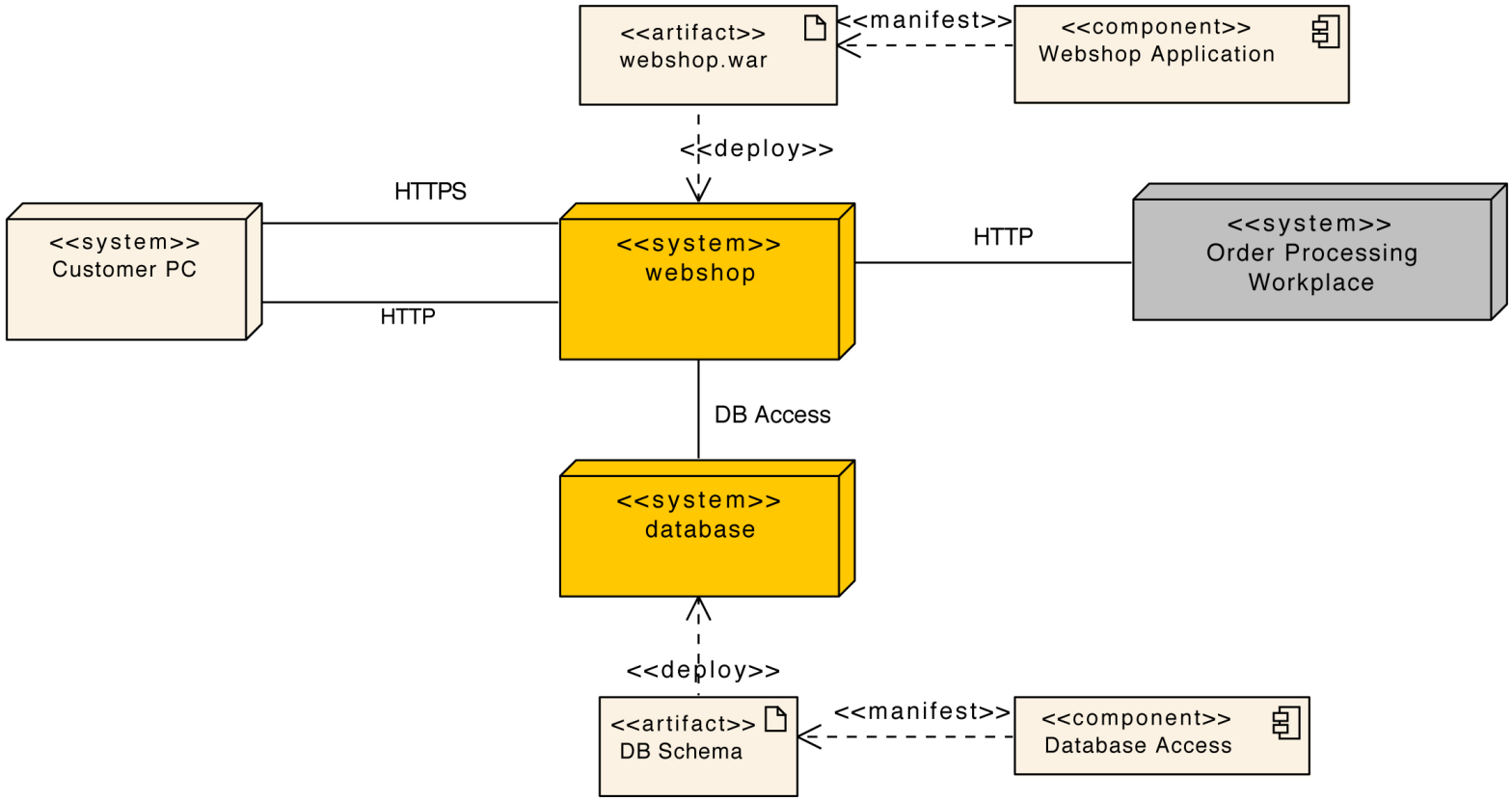
An artifact is a physical piece of information used in deployment and operation of a system. E.g., binaries or web application archives

Webshop deployment diagram (I)



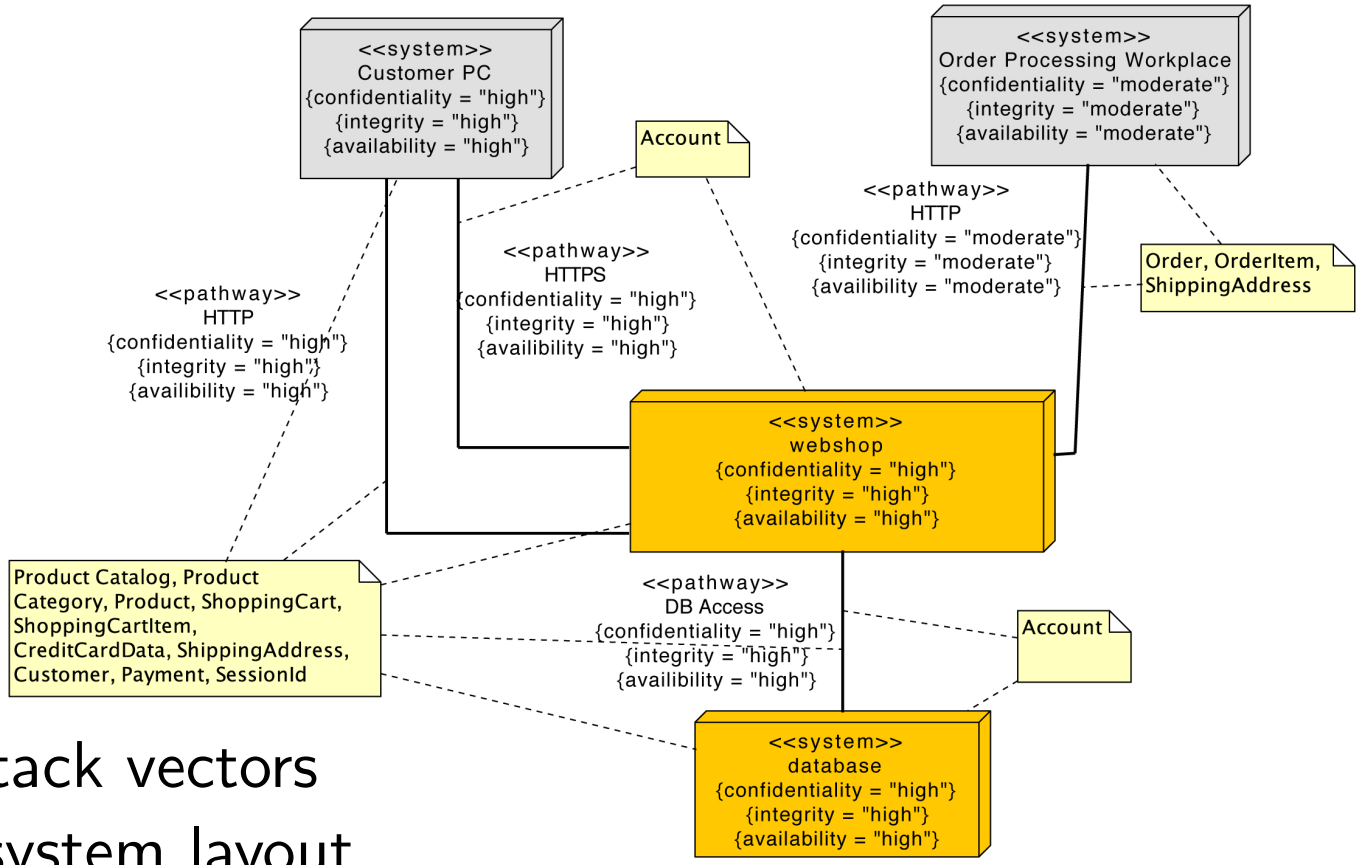
- Web shop runs on webshop and database nodes
- Employees use the Order Processing Workplace for system access
- Customers access the system from home PC
- Communication over HTTP and HTTPS, respectively

Webshop deployment diagram (II)



- Webshop Application is manifested in the artifact webshop.war (a Servlet Web Archive), which is deployed on the webshop node
- Database Access component is manifested in DB schema, which is deployed on the database node

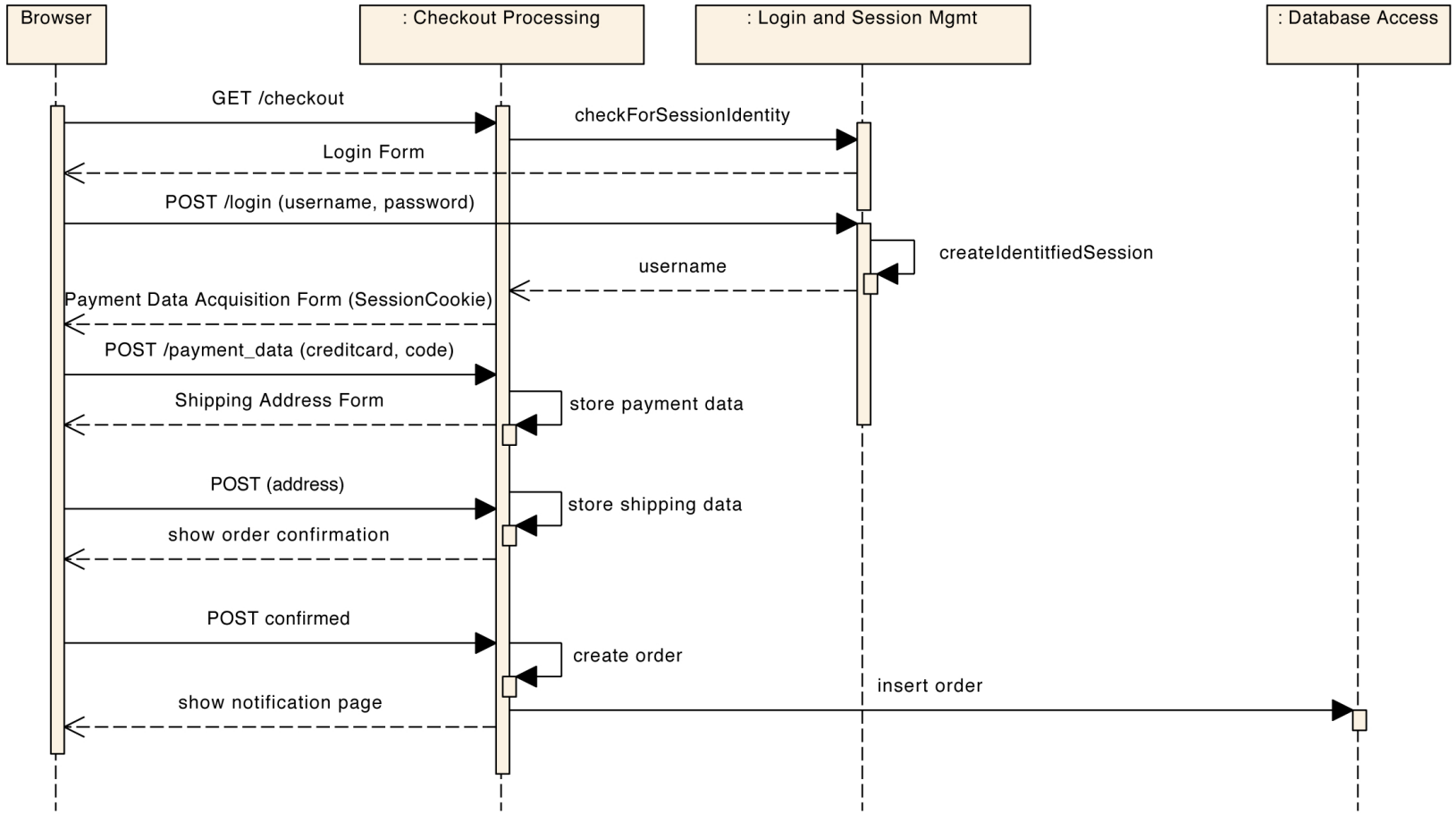
Depl. diagrams relevance for SE



- Identification of attack vectors based on physical system layout
  - Nodes reachable from internal and external networks
  - Communication channels
- Foundation for building the data pathways (explained later)
  - Aggregated sytem view showing where sensitive data are processed, transmitted, and stored
  - Useful for impact analysis and setting focus of work

Sequence Diagrams

Describe interaction, focusing on messages exchanged



Lifeline: represents an individual participant in the interaction.

Message: communication, e.g., raising a signal, invoking an operation, creating or destroying an instance.

Relevance for Security Engineering

- Sequence diagrams contribute to data pathway construction
  - Who communicates with whom over which channels?
  - What data travel over which channels?

Dynamic modeling — state oriented

- Model dynamic aspects of systems: **control** and **synchronization** within an object (or between objects)
  - What are the **states** of the system?
  - Which **events** does the system react to?
  - Which **transitions** are possible?
  - When are **activities** (functions) started and stopped?
- **Example:** In view mode, when the left mouse button is pressed, an option menu appears and user can subsequently enter input
- Such models correspond to **transition systems**

  Also called **state machines** or (variants of) **automata**

Statecharts

- Statecharts (Harel 1987) are a UML variant of state machines
- They extend standard state machines in various ways

  Hierarchy: nested states, used for iterated refinement

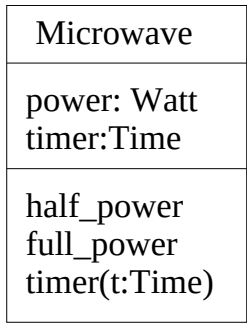
  Parallelism: machines are combined via product construction

  Time and reactivity: for modeling reactive systems

- See literature for syntax and semantics of these extensions
  - State charts are a formal method.
  - Unfortunately no canonical semantics.
    E.g., see von der Beeck, for over 20.[2]

[2] Michael von der Beeck, "A comparison of Staechart variants", Springer-Verlag LNCS, volume 863, 1994.

An example — a microwave



- Microwave object state stores time, power, ...
- Statechart states (also called "modes") represent sets of object states and are characterized by values of certain state variables

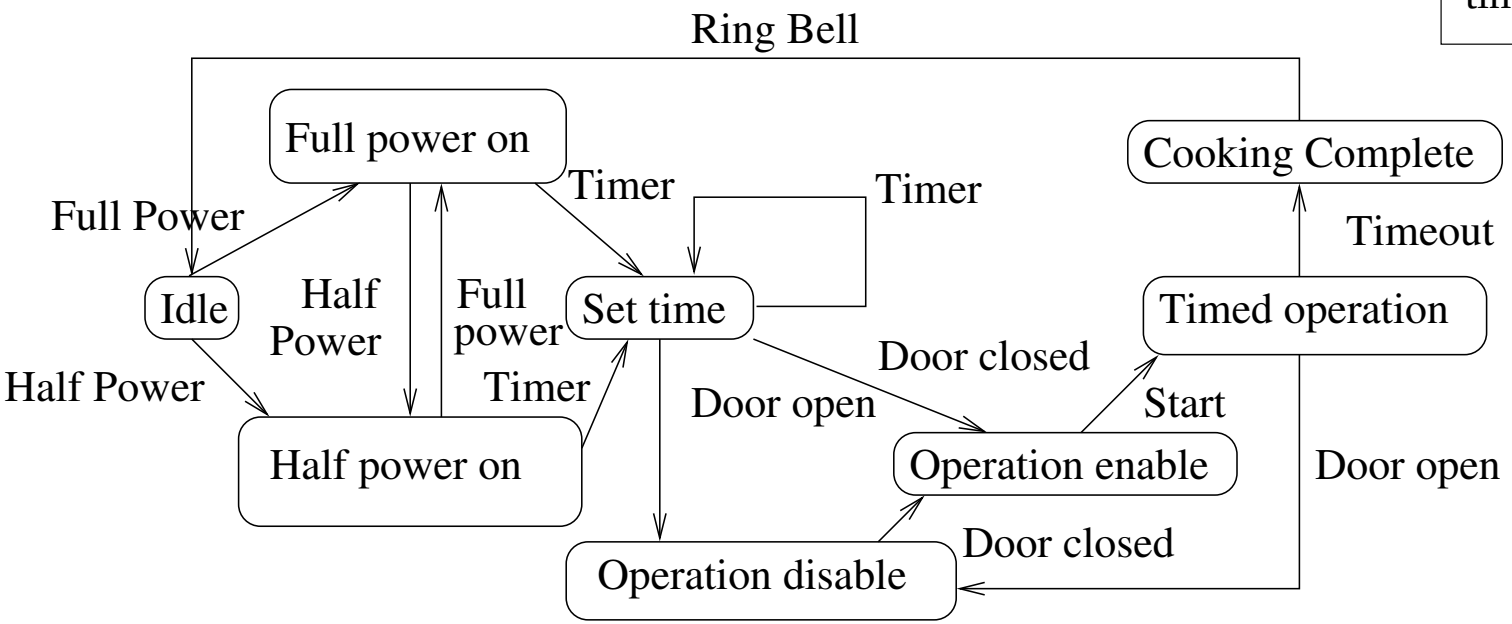
State	**Description**
Half Power On	Power is 300 Watt
Full Power On	Power is 600 Watt
Set Time	Timer turned on (>0)
⋮	⋮

- Transitions/events:

Event	**Description**
Half Power	'Half Power' button pressed
Full Power	'Full Power' button pressed
Timer	'Timer'-knob turned
⋮	⋮

A microwave model

Microwave
power: Watt timer:Time
half_power full_power timer(t:Time)



Questions

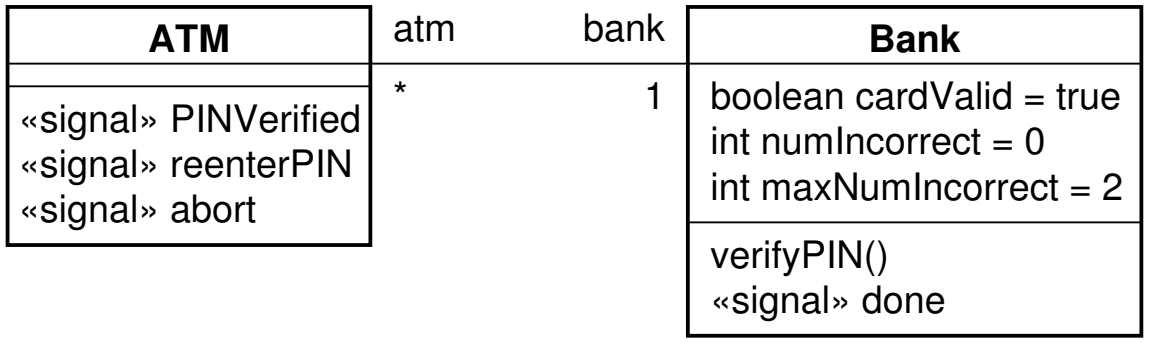
- Does class diagram offer all events depicted in the state chart?
- Does this statechart distinguish "input" versus "output" events?
- How would outputs be produced?

Syntax and semantics

- In general, transitions can include guards and output actions

  - Output actions A are methods
  - Guards C are Boolean expressions
  - Other constructs (actions on state entry/exit, ...) ignored here
- Semantics similar to Mealy machines: transition from S_1 to S_2 calling action A when
  - System is in S_1
  - Event E occurs and condition C holds
  - Complications arise in full language, e.g., parallel state machines with circular actions/reactions

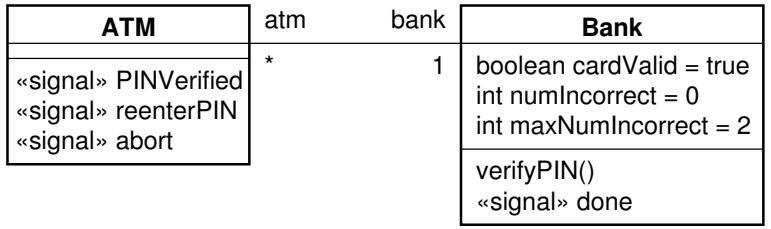
An ATM model[3]



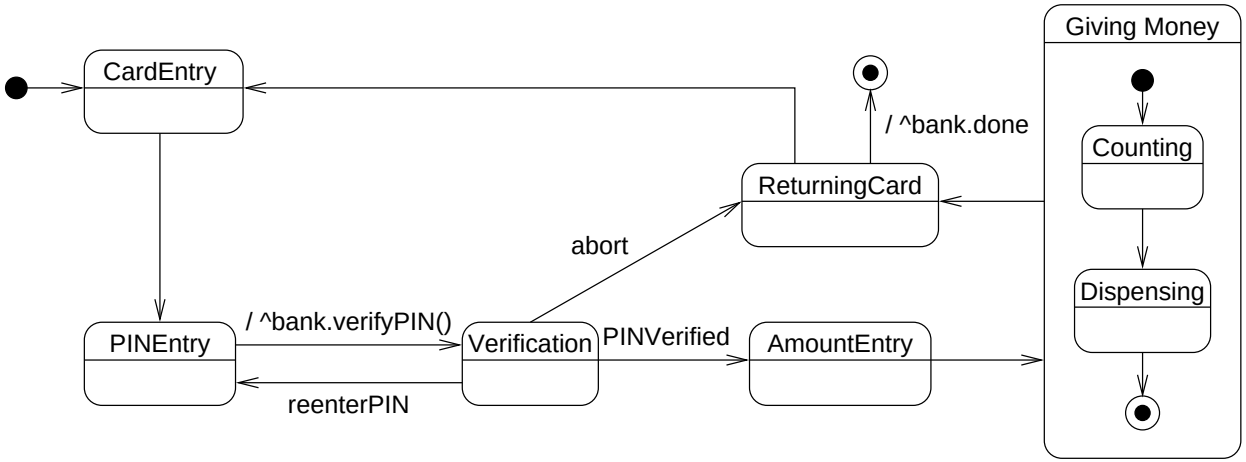
- Describes a world of objects populated by banks and ATMs.
  - Model only part of ATM usage dealing with PIN verification
  - Many details omitted or massively simplified
    E.g., only 1 ATM user, 1 smart card, population static, ...
- ⟨⟨signal⟩⟩ stereotype denotes signals that can be sent to objects

[3] See paper *Modeling Checking UML State Machines and Collaborations*, Schäfer, Knapp, and Merz. Graphics courtesy of Stephan Merz. For a formal model (including security requirements) see paper cited at end.

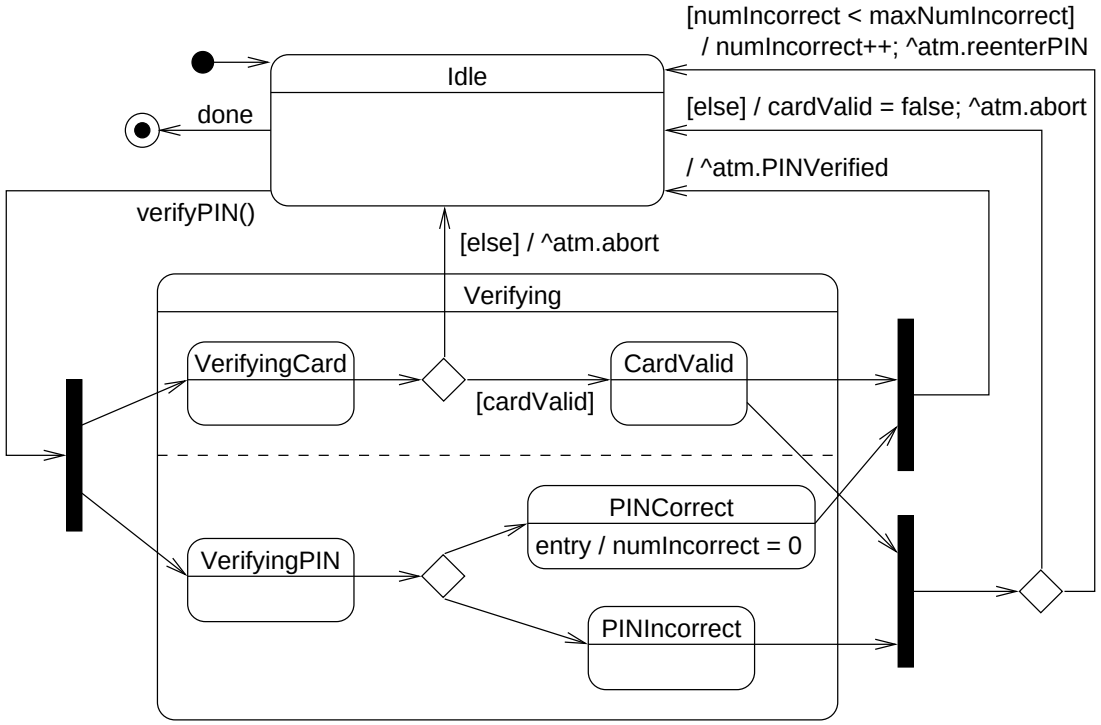
Corresponding state machines



ATM



Bank

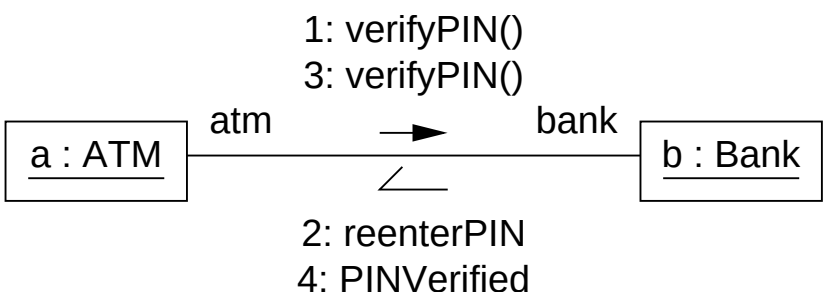


Notes on example

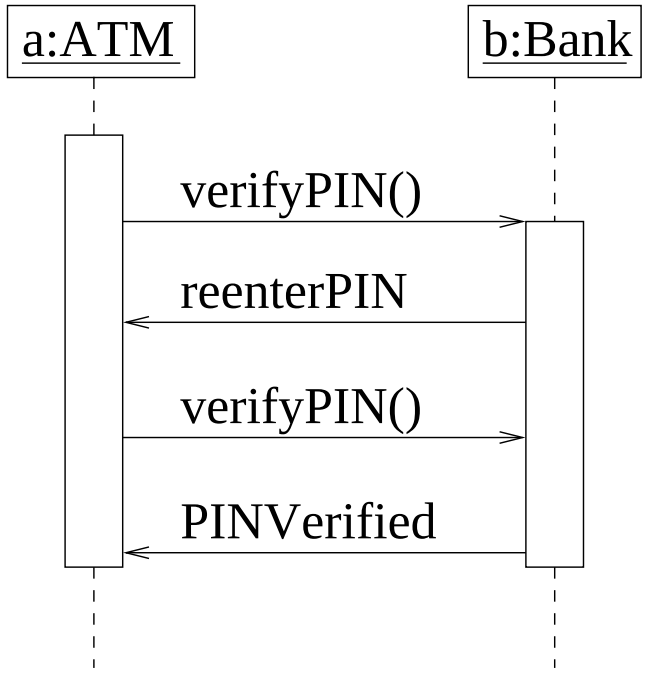
- Illustrates more advanced statechart features
  - Composite states (e.g., Giving Money)
  - Concurrent states (Card and Pin Verification)
  - Fork and Join (pseudo-)states, which synchronize transitions
  - Choice-points with conditions (representing multiple transitions)
- Illustrates multiple views and their conformance
- Motivates too the need for analysis

  E.g., can the system deadlock?

Behavior consistent with state machines



Communication Diagram



Sequence Diagram

- Equivalent descriptions of same interaction scenario where second PIN-entry attempt succeeds
- **Question:** How does this relate to automata theory studied in theoretical computer science?

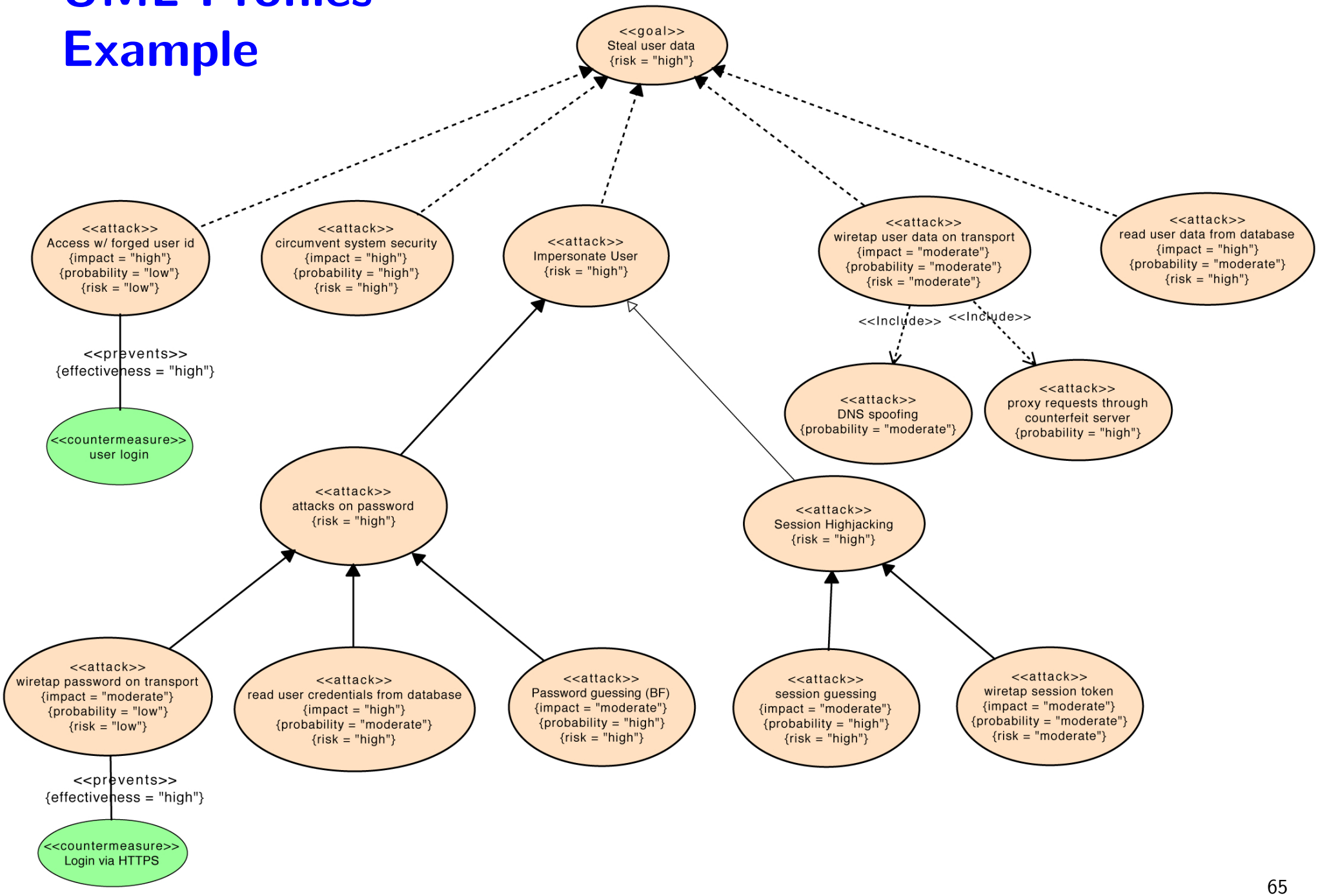
UML Extensibility

Foundation for building domain-specific languages using UML

Key concepts:

- Stereotype
  - represents a sub-type of a core UML type or another stereotype
  - Characterized by a Name (+ icon or color), Base type, and 0-n Tagged values
- Tagged value
  - represents additional model element attributes (name-value pairs)
  - Used in conjunction with stereotypes or independently

UML Profiles Example



Road map

- Why model?
- Two simple examples
- The Unified Modeling Language

☞ **Summary**

Conclusions and lessons learned

- Modeling languages used to capture different system views

  Static: e.g., classes and their relationships
  Dynamic: state-oriented behavioral description
  Functional: behavior described by function composition
  Traces/collaboration: showing different interaction scenarios

- Models are starting point for further phases. But their value is proportional to their prescriptive and analytic properties!

- Foundation of security analysis and bearer for additional security-related information

Coming up: applications in requirements and (security) design

Literature

- OMG Unified Modeling LanguageTM (OMG UML), Superstructure, www.omg.org/cgi-bin/doc?formal/09-02-02
- James Rumbaugh and Ivar Jacobson and Grady Booch, The Unified Modeling Language reference manual, Addison-Wesley, 1999.
- Hassan Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with UML Addison Wesley, 2000.
- Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli, Fundamentals of Software Engineering, Prentice Hall, 2003.
- Alexander Knapp, Stephan Merz, and Christopher Rauh, Model Checking Timed UML State Machines and Collaborations, FTRTFT '02, Springer-Verlag, 2002.
- David Harel, Statecharts in the Making: A Personal Account, Communications of the ACM, March 2009.