9 Model-based requirements specification

Why do we model requirements?

- Gain an overview of a set of requirements
- Understand relationships and inter-connections between requirements
- Focus on some aspect of a system, abstracting from the rest

Primarily for functional requirements

Quality requirements and constraints are mostly specified in natural language

9.1 Models in RE

DEFINITION. Model – an abstract representation of an existing part of reality or a part of reality to be created.

The notion of reality includes any conceivable set of elements, phenomena or concepts, including other models.

With respect to a model, the modeled part of reality is called the original.

- Requirements models are problem-oriented models of the system to be built
- Architecture and design information is omitted

Requirements models can be used for

- **Specifying** requirements (as a means of replacing textually represented requirements)
- **Paraphrasing** textually represented requirements to improve understanding of complex structures and dependencies
- **Testing** textually represented requirements to uncover omissions, ambiguities and inconsistencies
- **Decomposing** a complex reality into comprehensible parts

Which aspects can be modeled?

- Structure and Data
  - Structural properties of a system, particularly of the static data
  - Structure of a system's domain
- Function and Flow

  Sequence of actions and control / data flow for
  - producing a required result
  - describing a (business) process
- State and Behavior

  Behavior of a system or a domain component
  - State-dependent reactions to events
  - Dynamics of component interaction

Which aspects can be modeled? – continued

- Context
  - Structural embedding of system in its environment
  - Interaction between system and actors in the context
- Goals

  Understanding the goals for a system
  - Goal decomposition
  - Goal-agent networks

9.2 Modeling structure and data

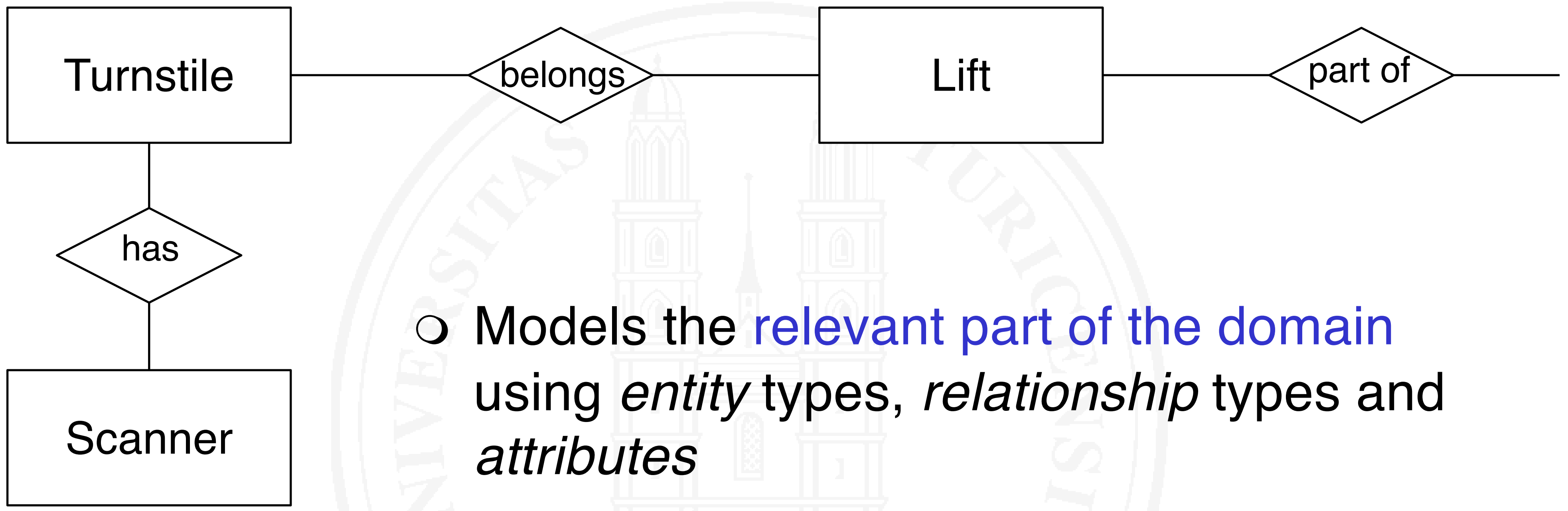
- Entity-relationship models
- Class and object models
- Component models

What to model

- Static system models: Information that a system needs to know and store persistently
- Static domain models: The (business) objects and their relationships in a domain of interest.

Data modeling (entity-relationship models)

[Chen 1976]



- ❍ Models the relevant part of the domain using *entity* types, *relationship* types and *attributes*
- + Rather easy to model
- + Straightforward mapping to relational database systems
- – Ignores functionality and behavior
- – No means for system decomposition

Object and class modeling

[Booch 1986, Booch 1994, Glinz et al. 2002]

Idea

- Identify those entities in the domain that the system has to store and process
- Map this information to objects/classes, attributes, relationships and operations
- Represent requirements in a static structural model
- Modeling individual objects does not work: too specific or unknown at time of specification
  - → *Classify* objects of the same kind to classes: Class models
  - → or select an abstract *representative*: Object models

Terminology

Object – an individual entity which has an identity and does not depend on another entity.

Examples: Turnstile no. 00231, The Plauna chairlift

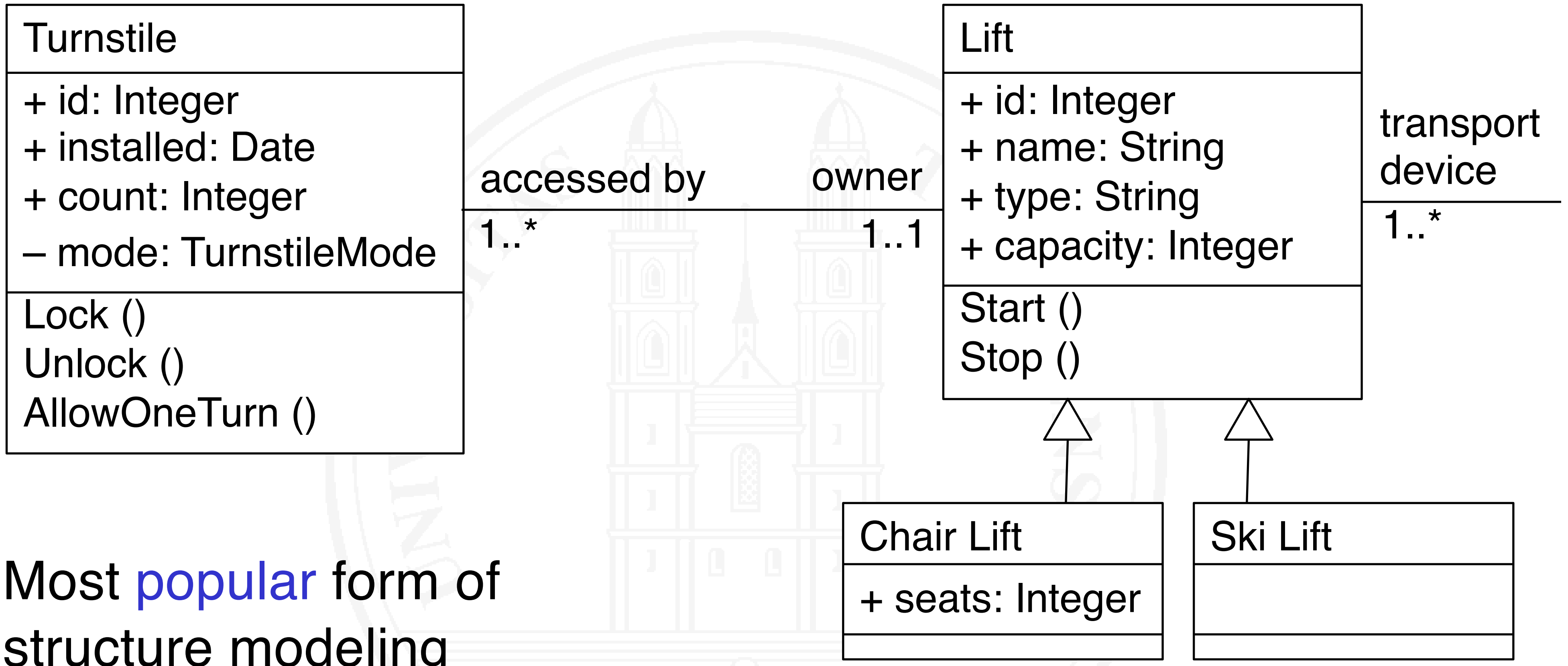
Class – Represents a set of objects of the same kind by describing the structure of the objects, the ways they can be manipulated and how they behave.

Examples: Turnstile, Lift

Abstract Object – an abstract representation of an individual object or of a set of objects having the same type

Example: A Turnstile

Class models / diagrams



Most popular form of structure modeling

Typically using UML class diagrams

Class diagram: a diagrammatic representation of a class model

Class models are sometimes inadequate

- Class models don’t work when different objects of the same class need to be distinguished
- Class models can’t be decomposed properly: different objects of the same class may belong to different subsystems
- Subclassing is a workaround, but no proper solution

In such situations, we need object models

Object models: a motivating example

Example: Treating incidents in an emergency command and control system

Emergency command and control systems manage incoming emergency calls and support human dispatchers in reacting to incidents (e.g., by sending police, fire fighters or ambulances) and monitoring action progress.

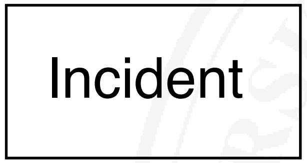
When specifying such a system, we need to model

- Incoming incidents awaiting treatment
- The incident currently managed by the dispatcher
- Incidents currently under treatment
- Closed incidents

Class models are inadequate here

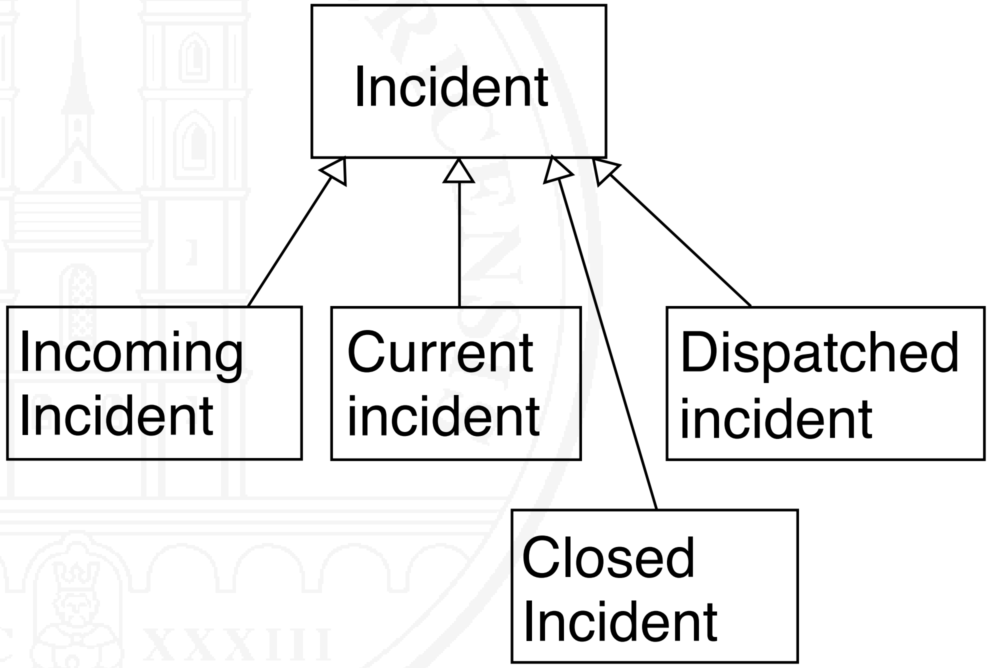
In a class model, incidents would have to be modeled as follows:

either



Bad: essential elements of the problem are not modeled

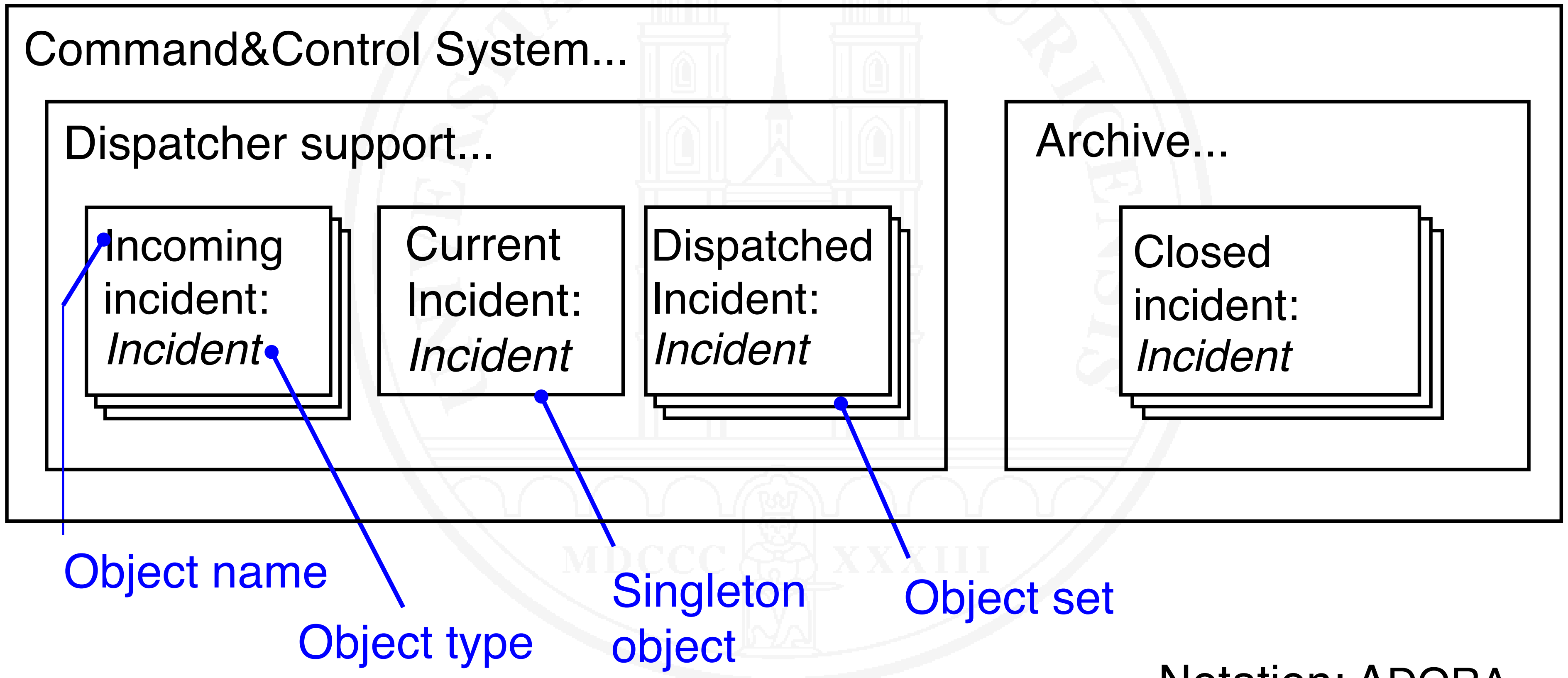
or



Unnatural: all subclasses are structurally identical

Object models work here

Modeling is based on a hierarchy of abstract objects



Command&Control System...
Dispatcher support...
Incoming incident: *Incident*
Current Incident: *Incident*
Dispatched Incident: *Incident*
Archive...
Closed incident: *Incident*
Object name
Object type
Singleton object
Object set

Notation: ADORA

ADORA [Glinz et al. 2002]

- ADORA is a language and tool for object-oriented specification of software-intensive systems
- Basic concepts
  - Modeling with abstract objects
  - Hierarchic decomposition of models
  - Integration of object, behavior and interaction modeling
  - Model visualization in context with generated views
  - Adaptable degree of formality
- Developed in the RERG research group at UZH

Modeling with abstract objects in UML

- Not possible in the original UML (version 1.x)
- Introduced 2004 as an option in UML 2
- Abstract objects are modeled as components in UML
- The component diagram is the corresponding diagram
- Lifelines in UML 2 sequence diagrams are also frequently modeled as abstract objects
- In UML 2, class diagrams still dominate

What can be modeled in class/object models?

- Objects as *classes* or *abstract objects*
- Local properties as *attributes*
- Relationships / non-local properties as *associations*
- Services offered by objects as *operations* on objects or classes (called *features* in UML)
- Object behavior
  - Must be modeled in separate *state machines* in UML
  - Is modeled as an *integral part* of an object hierarchy in ADORA
- System-context interfaces and functionality from a user's perspective *can't* be modeled *adequately*

Object-oriented modeling: pros and cons

+ Well-suited for describing the structure of a system
+ Supports locality of data and encapsulation of properties
+ Supports structure-preserving implementation
+ System decomposition can be modeled
– Ignores functionality and behavior from a user’s perspective
– UML class models don’t support decomposition
– UML: Behavior modeling weakly integrated

Mini-Exercise: Classes vs. abstract objects

Specify a distributed heating control system for an office building consisting of a central boiler control unit and a room control unit in every office and function room.

- The boiler control unit shall have a control panel consisting of a keyboard, a LCD display and on/off buttons.
- The room control unit shall have a control panel consisting of a LCD display and five buttons: on, off, plus, minus, and enter.

Model this problem using

a. A class model

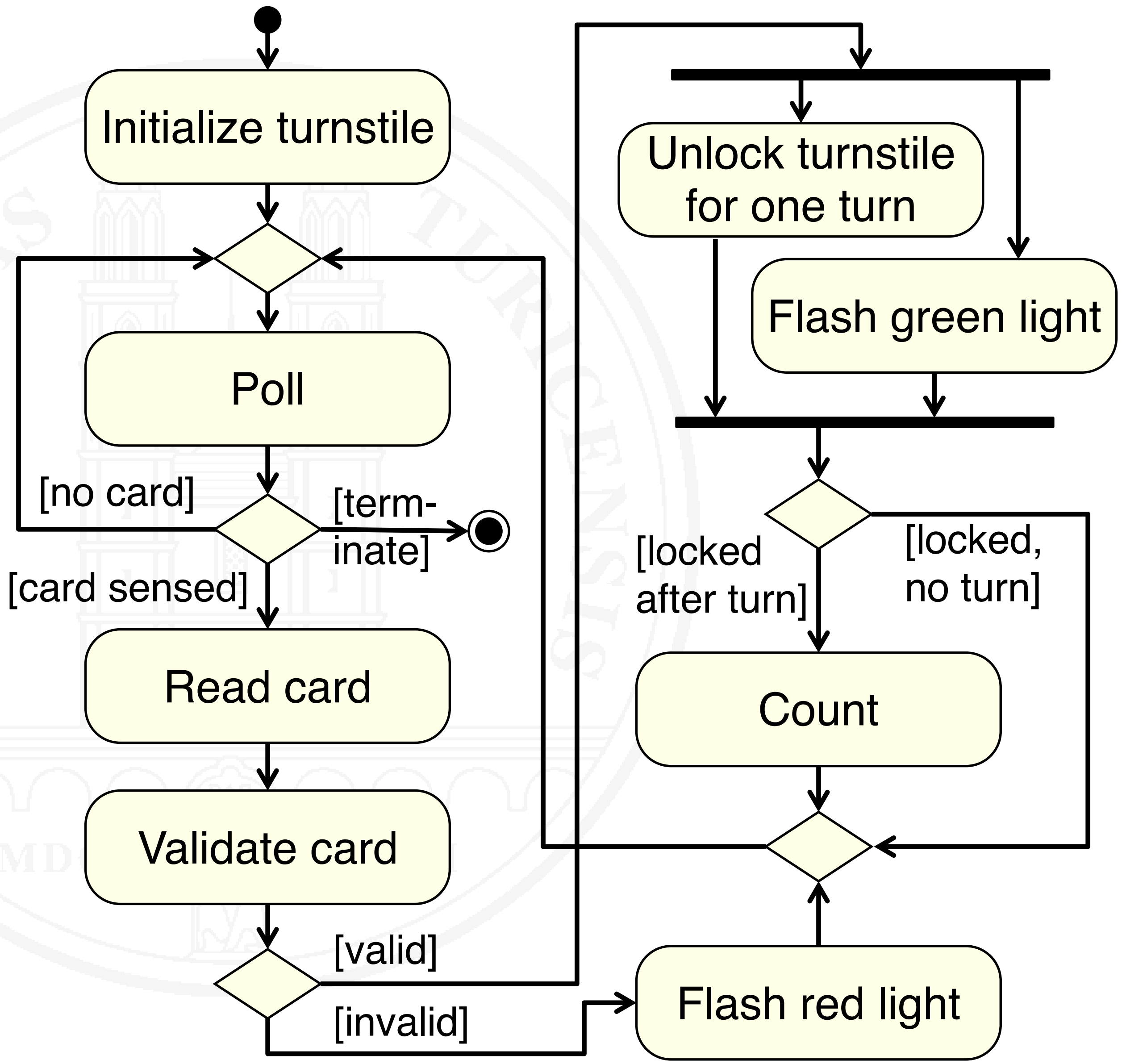
b. An abstract object model.

9.3 Modeling function and flow

- Activity models
- Data flow / information flow models
- Process and work flow models
- Domain story models

Activity modeling: UML activity diagram

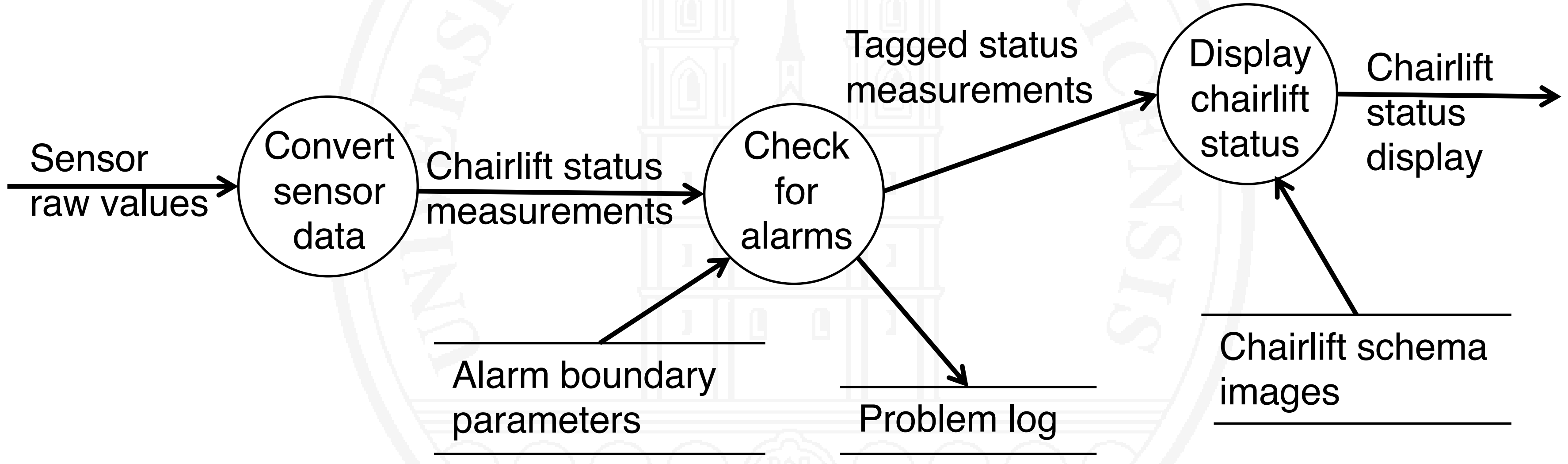
- Models process activities and control flow
- Can model data flow
- Model can be underpinned with execution semantics



Data and information flow

[DeMarco 1978]

- Models system functionality with data flow diagrams
- Once a dominating approach; rarely used today



+ Easy to understand
+ Supports system decomposition
– Treatment of data outdated: no types, no encapsulation

Process and workflow modeling

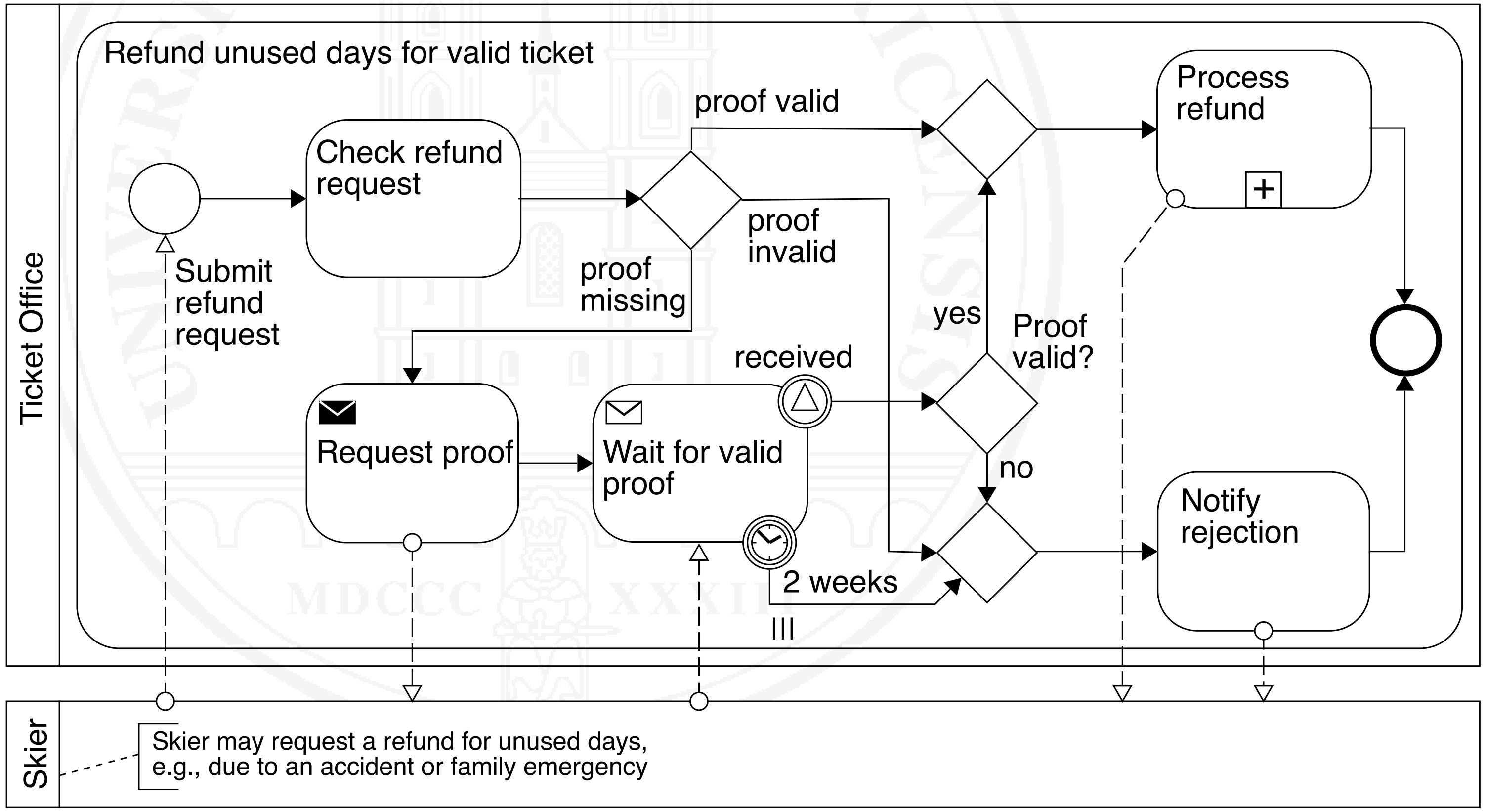
- Elements
  - Process steps / work steps
  - Events influencing the flow
  - Control flow
  - Maybe data / information access and responsibilities
- Typical languages
  - UML activity diagrams
  - BPMN
  - Event-driven process chains

Process modeling: BPMN

[Object Management Group 2013]

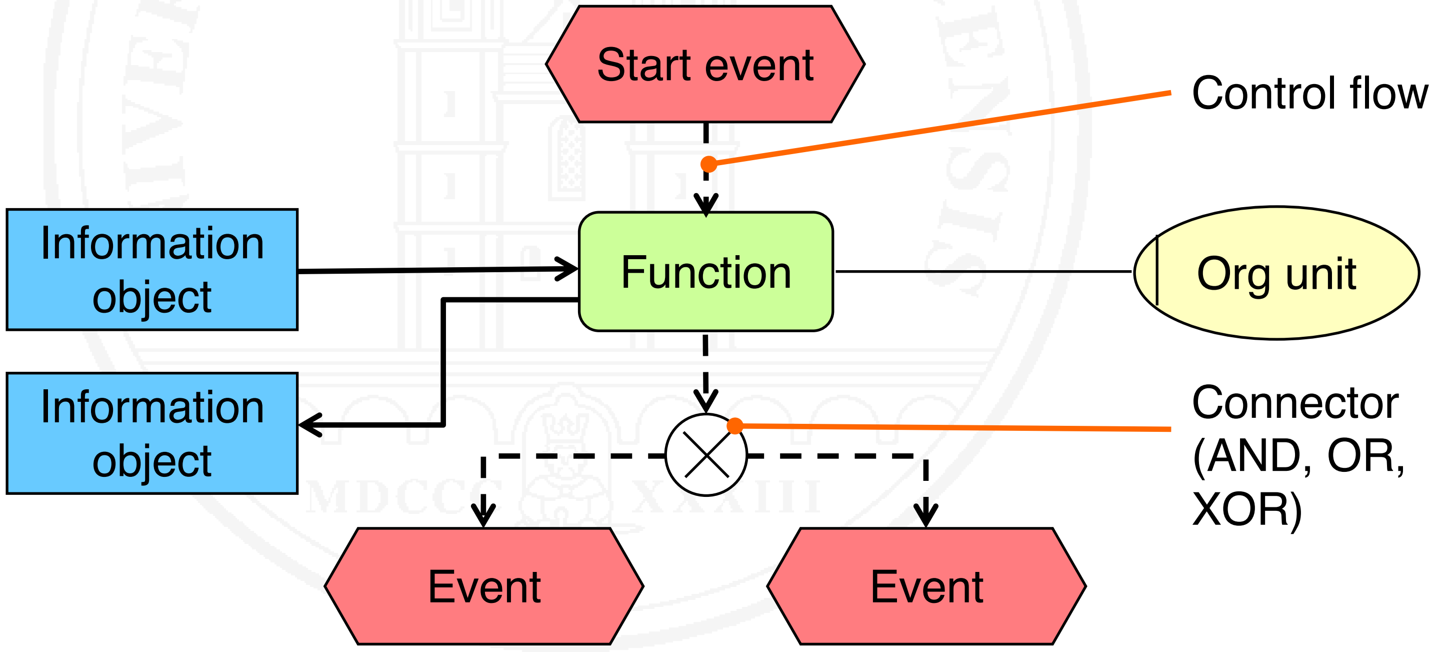
BPMN (Business Process Model and Notation)

- Rich language for describing business processes

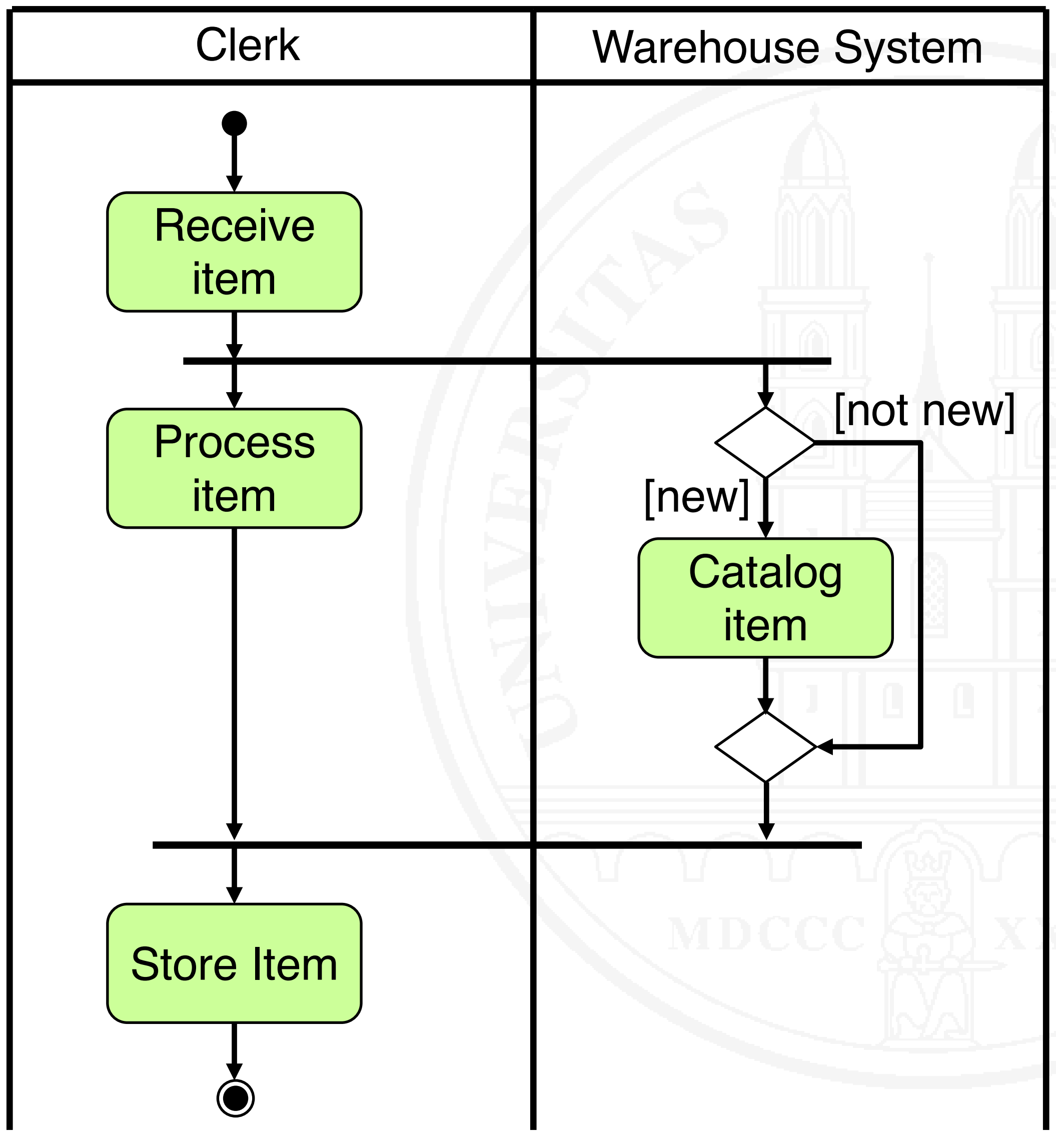


Process modeling: EPC

- Event-driven process chains (In German: ereignisgesteuerte Prozessketten, EPK)
- Adopted by SAP for modeling processes supported by SAP's ERP software



Process modeling: UML Activity Diagram



Initial node

Action

Fork node (fork into parallel flows)

Decision node (with decision conditions)

Merge node

Join node (join&synchronize flows)

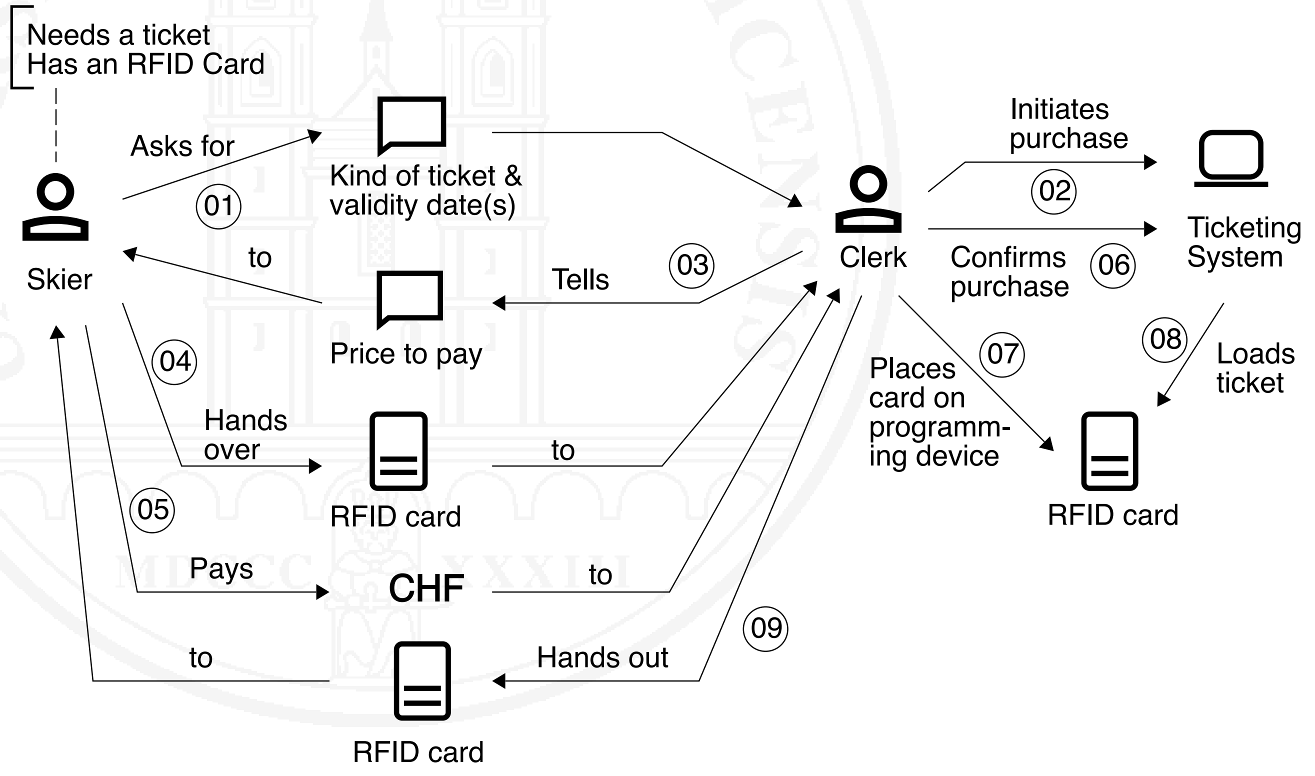
Final node

Activity partitions (“swimlanes”) model who is responsible for what

Domain story models

[Hofer&Schwendtner 2020]

- Visual stories about what stakeholders want to achieve
- Includes information about processes, system, people and organizations



9.4 Modeling behavior

Goal: describe dynamic system behavior

- How the system reacts to a sequence of external events
- How independent system components coordinate their work

Means:

- Finite state machines (FSMs) – not discussed here
- Statecharts / State machines
  - Easier to use than FSMs (although theoretically equivalent)
  - State machines are the UML variant of statecharts
- Sequence diagrams (primarily for behavioral scenarios)
- Petri nets – not discussed here

Statecharts

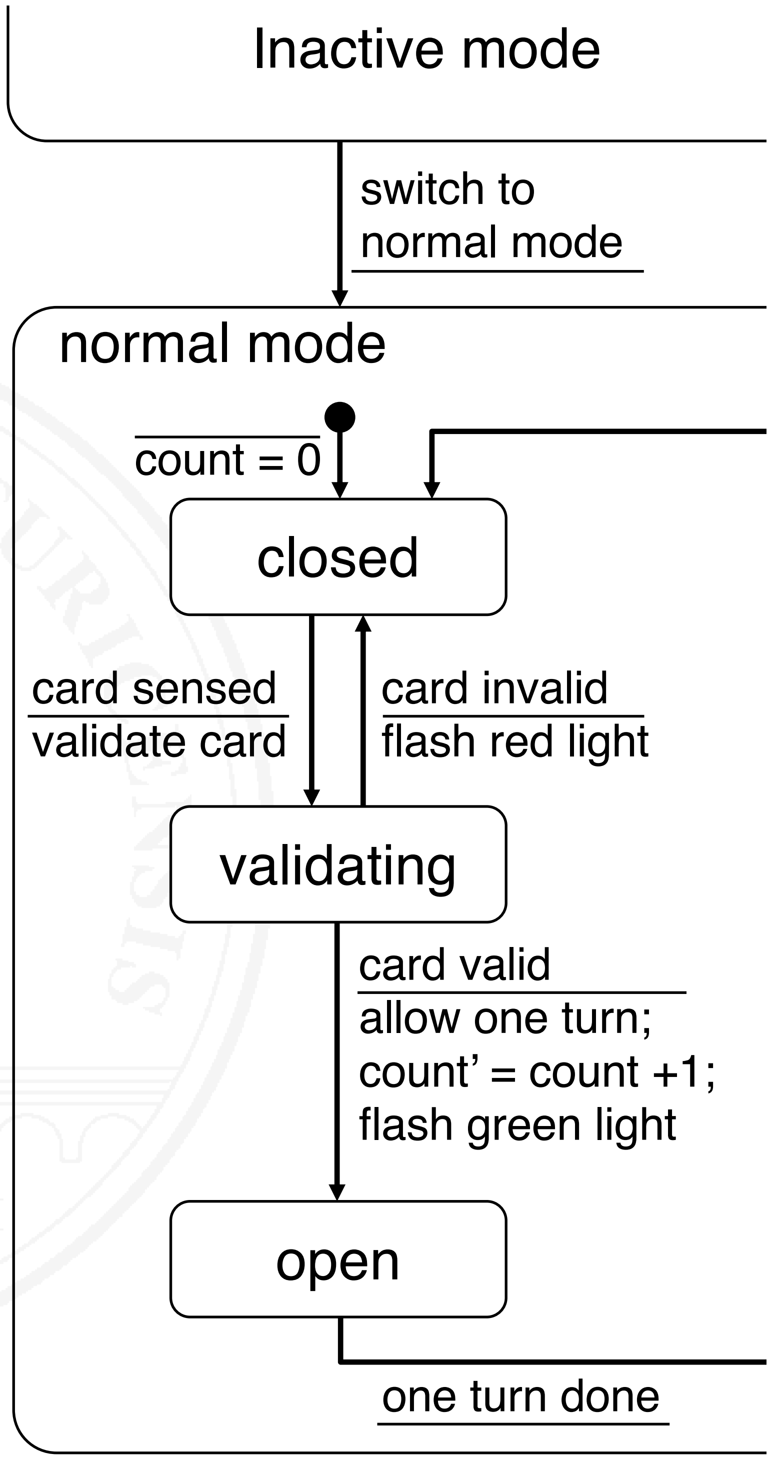
[Harel 1988]

- Models the *dynamic* behavior:
  - How the system reacts to external events in a given state
  - Reaction depends on actual state
  - States may be hierarchically nested and/or orthogonal (parallel)
- In UML: state machine diagrams

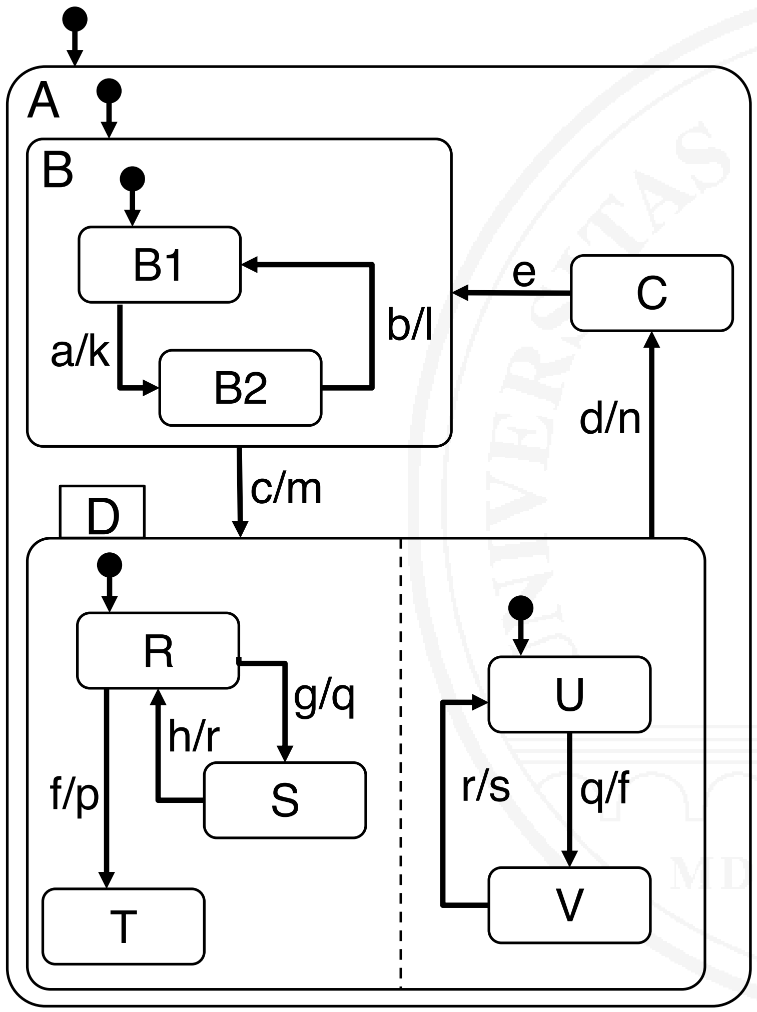
+ Global view of system behavior

+ Precise, but still readable

– Weak for modeling functionality and data

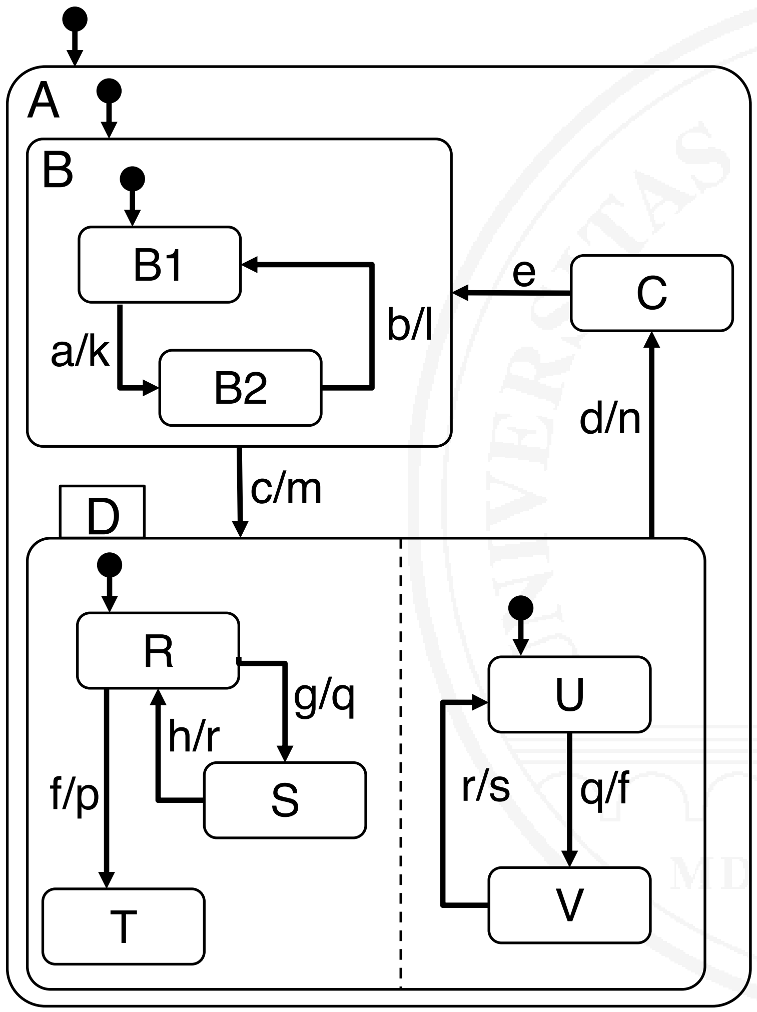


Interpretation of Statecharts



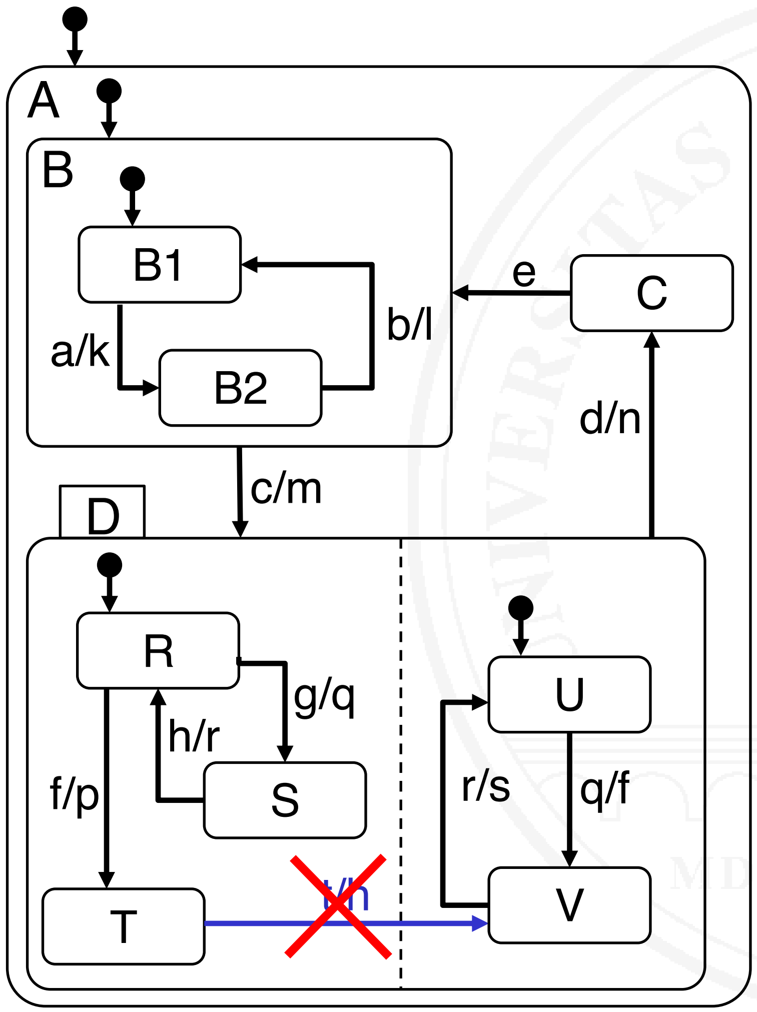
- Statecharts may have *composite states* with substates and *parallel regions, e.g.*:
  – B is a composite state, consisting of substates B1 and B2
  – D is a composite state with two parallel regions
- *Events* trigger *state transitions* and can trigger *actions* or *new events*, e.g.: The occurrence of c triggers the transition from B to D, provided the system currently is in state B. The transition triggers m, which may be an action or an event.

Interpretation of Statecharts – 2



- The system is always in exactly one combination of states and nested substates, e.g.:
  – Statechart A initially is in state B and its substate B1
  – After the occurrence of c, A is in state D and substates (R, U)
  – After the occurrence of f, A still is in state D, but now in substates (T, U)
- Events are ignored when there is no transition for it in the current state: e.g., in state B2, event f is ignored

Interpretation of Statecharts – 3

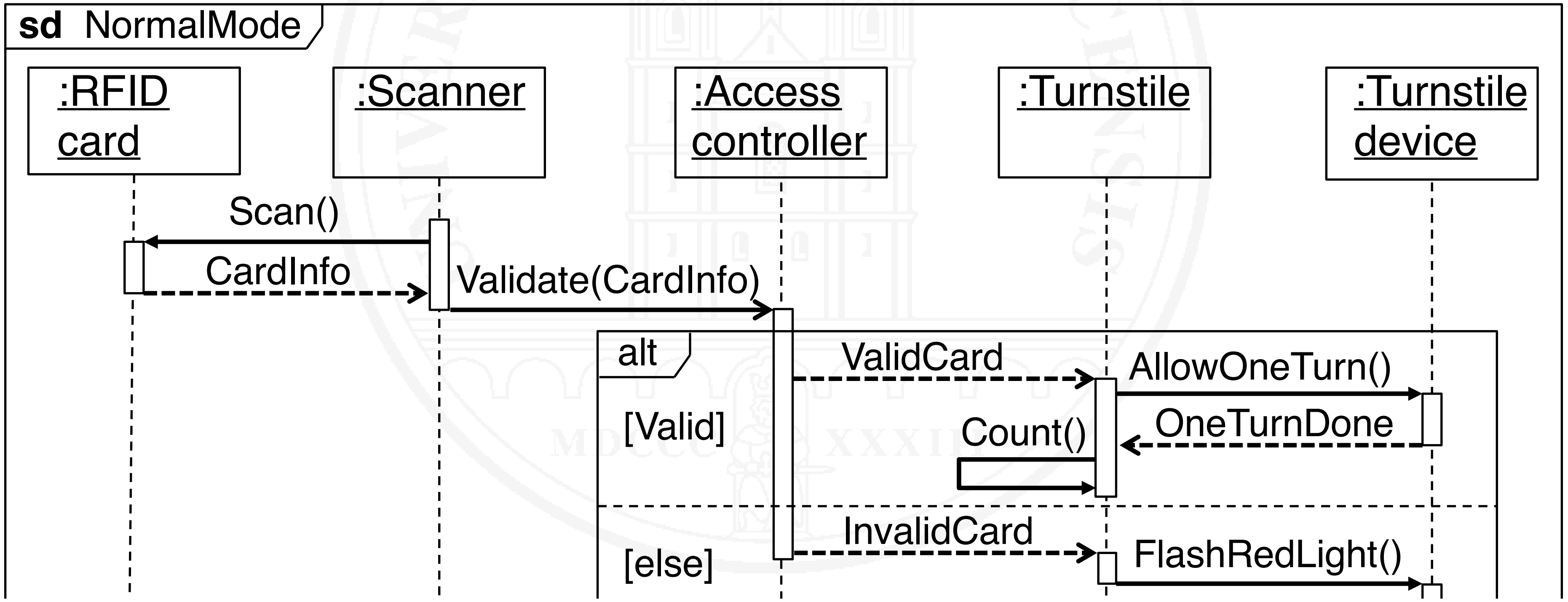


- State transitions into a composite state also enter its substates
- Leaving a state implies leaving all its substates
- Regions can influence each other via events, e.g.:
  If the system is in R and U, the event g triggers a transition from R to S, producing q. Event q in turn triggers a transition from U to V.
- Transitions between regions are forbidden

Sequence diagrams / MSCs

Object Management Group (2011b)

- Models ...
  - ... lifelines of system components or objects
  - ... messages that the components exchange



- Notation/terminology:
  - UML: Sequence diagram
  - Otherwise: Message sequence chart (MSC)

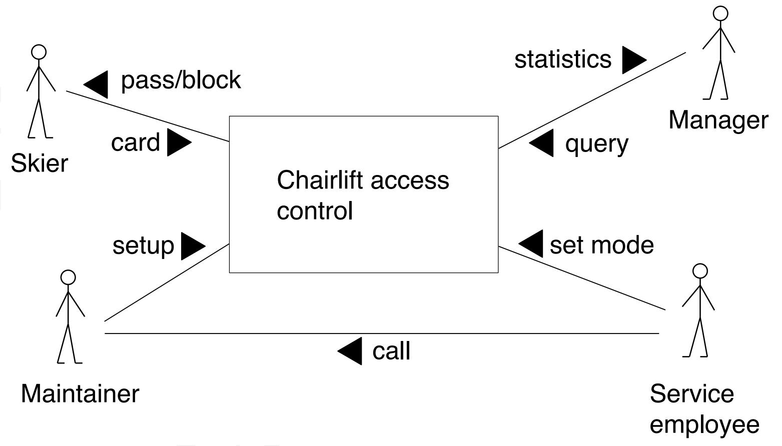
+ Visualizes component collaboration on a timeline

– In practice confined to the description of required scenarios

– Design-oriented, can detract from modeling requirements

9.5 Modeling context

Structural embedding



- Context diagrams, modeling
  - The system
  - The actors in the system's context
  - Information interfaces between actors and system
  - Information interfaces among actors

→ Chapter 2.4

Dynamic interaction between system and context

- Scenarios
- Use cases

Dynamic interaction: modeling the users' view

Describing the functionality of a system from a user's perspective: How can a user interact with the system?

Two key terms:

- Use case
- Scenario

[Carroll 1995,
Glinz 1995,
Glinz 2000a,
Jacobson et al. 1992,
Sutcliffe 1998,
Weidenhaupt et al. 1998]

Use case

DEFINITION. Use case – A set of possible interactions between external actors and a system that provide a benefit for the actor(s) involved.

Use cases specify a system from a user's (or other external actor's) perspective: every use case describes some functionality that the system must provide for the actors involved in the use case.

- Use case diagrams provide an overview
- Use case descriptions provide the details

[Jacobson et al. 1992
Glinz 2013]

Scenario

DEFINITION. Scenario – 1. In general: A description of a potential sequence of events that lead to a desired (or unwanted) result.
2. In RE: An ordered sequence of interactions between partners, in particular between a system and external actors. May be a concrete sequence (instance scenario) or a set of potential sequences (type scenario, use case).

[Carroll 1995
Sutcliffe 1998
Glinz 1995]

Use case / scenario descriptions

Various representation options

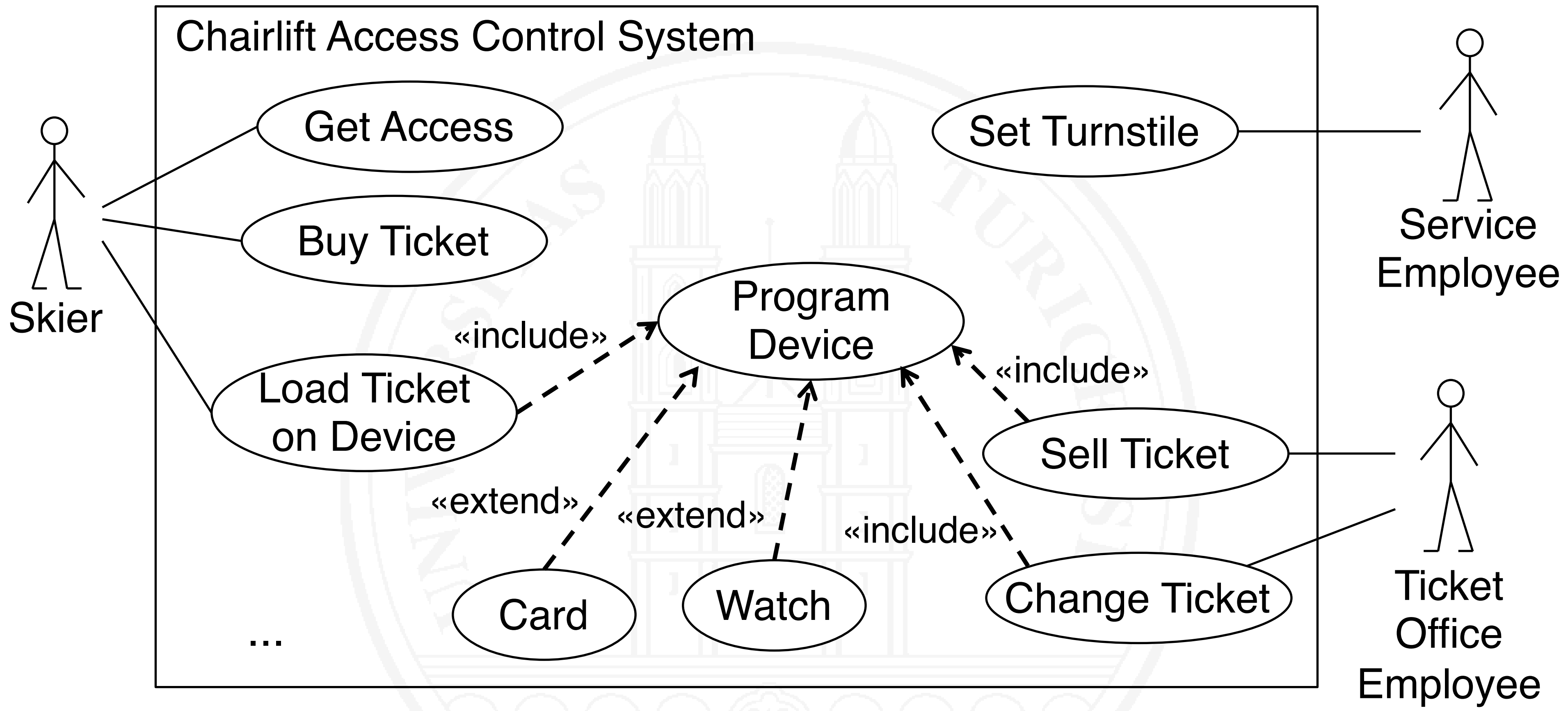
- Free text in natural language
- Structured text in natural language
- Statecharts / UML state machines
- UML activity diagrams
- Sequence diagrams / MSCs

Structured text is most frequently used in practice

A use case description with structured text

USE CASE SetTurnstiles
Actor: Service Employee
Precondition: none
Normal flow:
1 Service Employee chooses turnstile setup.
System displays controllable turnstiles: locked in red, normal in green, open in yellow.
2 Service Employee selects turnstiles s/he wants to modify.
System highlights selected turnstiles.
3 Service Employee selects Locked, Normal, or Open.
System changes the mode of the selected turnstiles to the selected one, displays all turnstiles in the color of the current mode.
...
Alternative flows:
3a Mode change fails: System flashes the failed turnstile in the color of its current mode.
...

UML Use case diagram

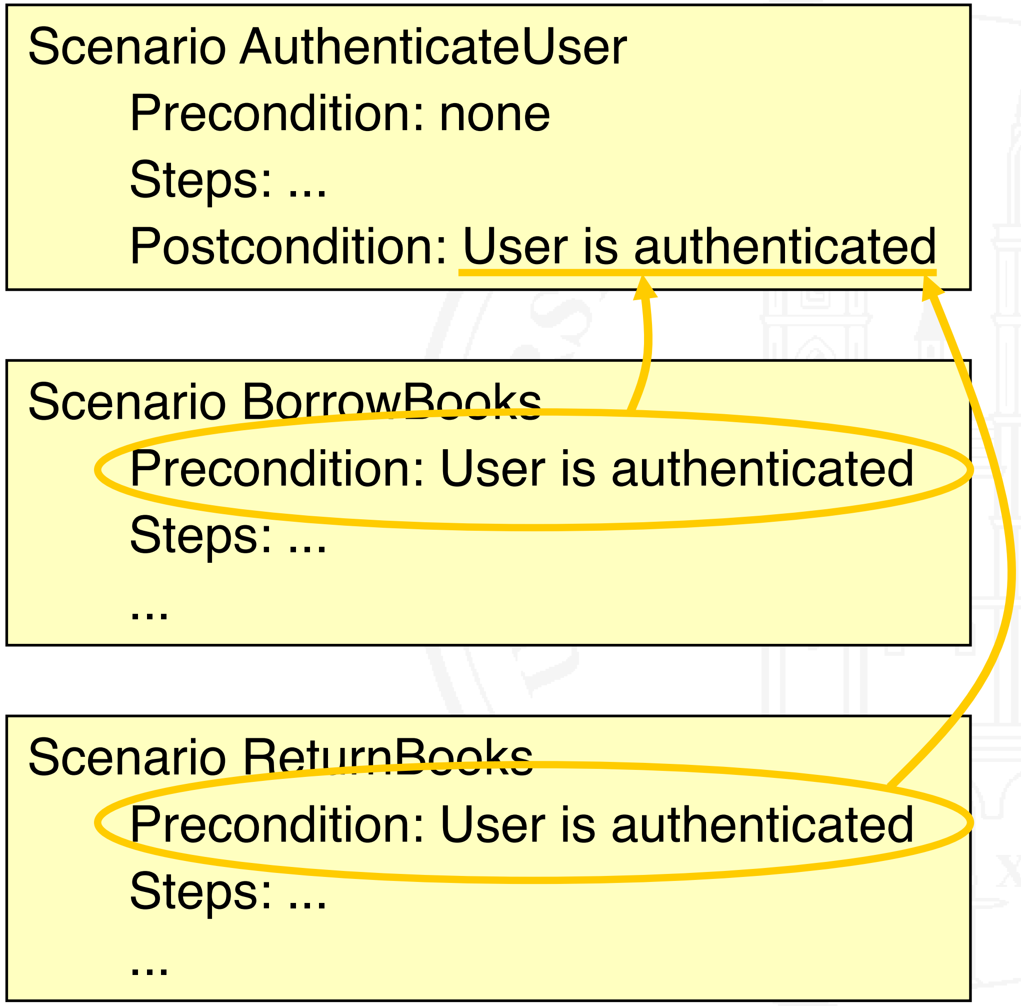


+ Provides abstract overview from actors' perspectives
– Ignores functions and data required to provide interaction
– Can't properly model hierarchies and dependencies

Dependencies between scenarios / use cases

- UML can only model inclusion, extension and generalization
- However, we need to model
  - Control flow dependencies (sequence, alternative, iteration)
  - Hierarchical decomposition
- Largely ignored in UML (Glinz 2000b)
- Options
  - Pre- and postconditions
  - Statecharts
  - Extended Jackson diagrams (in ADORA, Glinz et al. 2002)
  - Specific dependency charts (Ryser and Glinz 2001)

Dependencies with pre- and postconditions



Scenario AuthenticateUser
    Precondition: none
    Steps: ...
    Postcondition: User is authenticated

Scenario BorrowBooks
    Precondition: User is authenticated
    Steps: ...
    ...

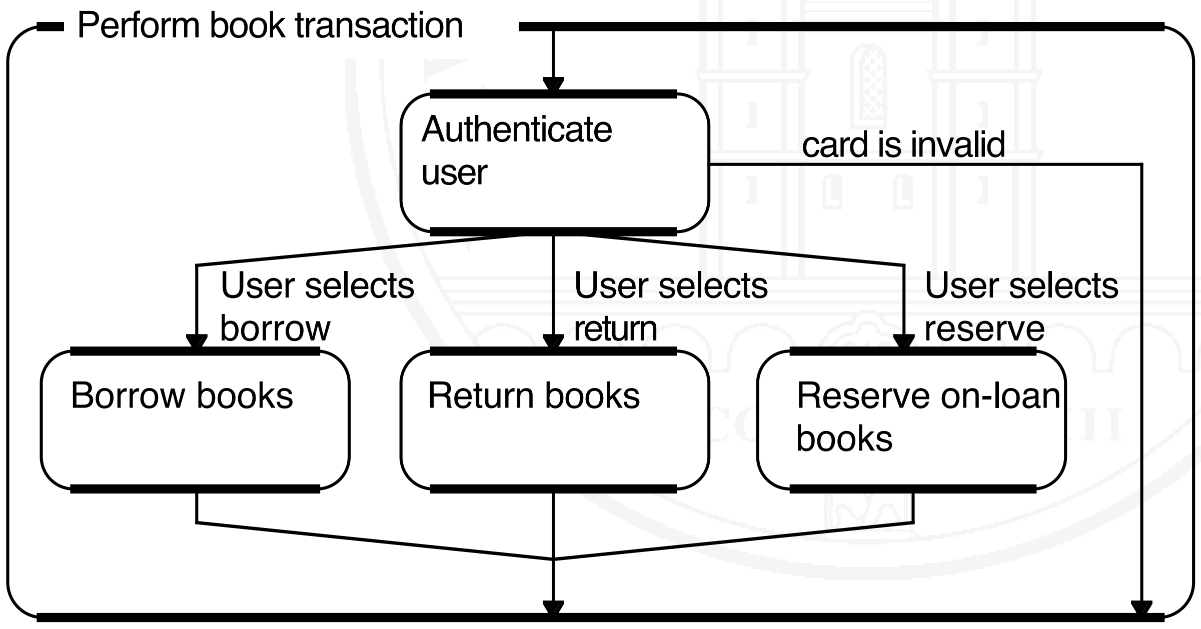
Scenario ReturnBooks
    Precondition: User is authenticated
    Steps: ...
    ...

- Simple dependencies of kind «B follows A» can be modeled
- Relationships buried in use case descriptions, no overview
- No hierarchical decomposition
- Modeling of complex relationships very complicated

Dependencies with Statecharts

[Glinz 2000a]

- Model scenarios as states*
- Classic dependencies (sequence, alternative, iteration, parallelism) can be modeled easily
- Hierarchical decomposition is easy

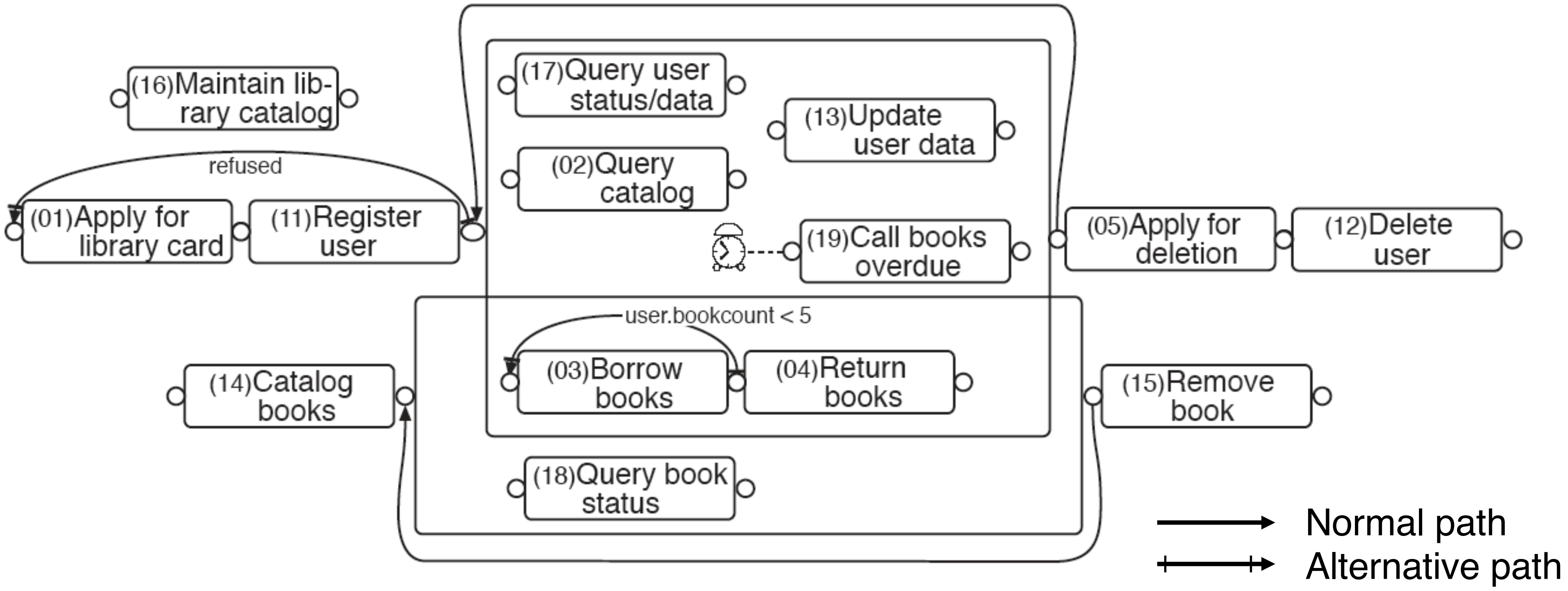


Research result, not used in today's practice

* With one main entry and exit point each; symbolized by top and bottom bars in the diagram

Dependency charts

- **Specific notation** for modeling of scenario dependencies (Ryser und Glinz 2001)
- **Research result**; not used in today's practice



Mini-Exercise: Writing a use case

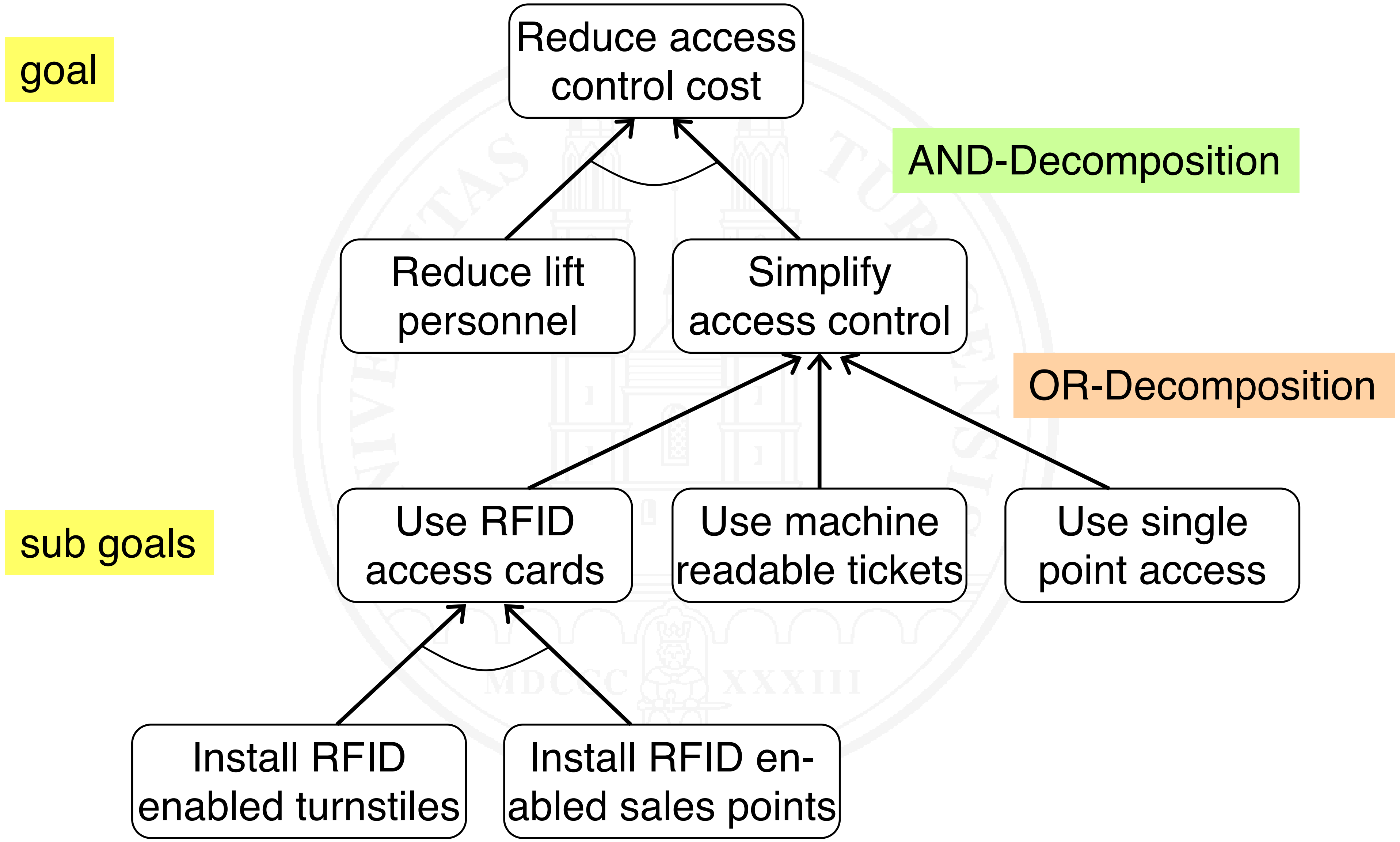
For the Chairlift access control system, write the use case “Get Access”, describing how a skier gets access to a chairlift using his or her RFID ticket.

9.6 Modeling goals

- Knowing the goals of an organization (or for a product) is essential when specifying a system to be used in that organization (or product)
- Goals can be decomposed into sub goals
- Goal decomposition can be modeled with AND/OR trees
- Considering multiple goals results in a directed goal graph

[van Lamsweerde 2001, 2004
Mylopoulos 2006
Yu 1997]

AND/OR trees for goal modeling

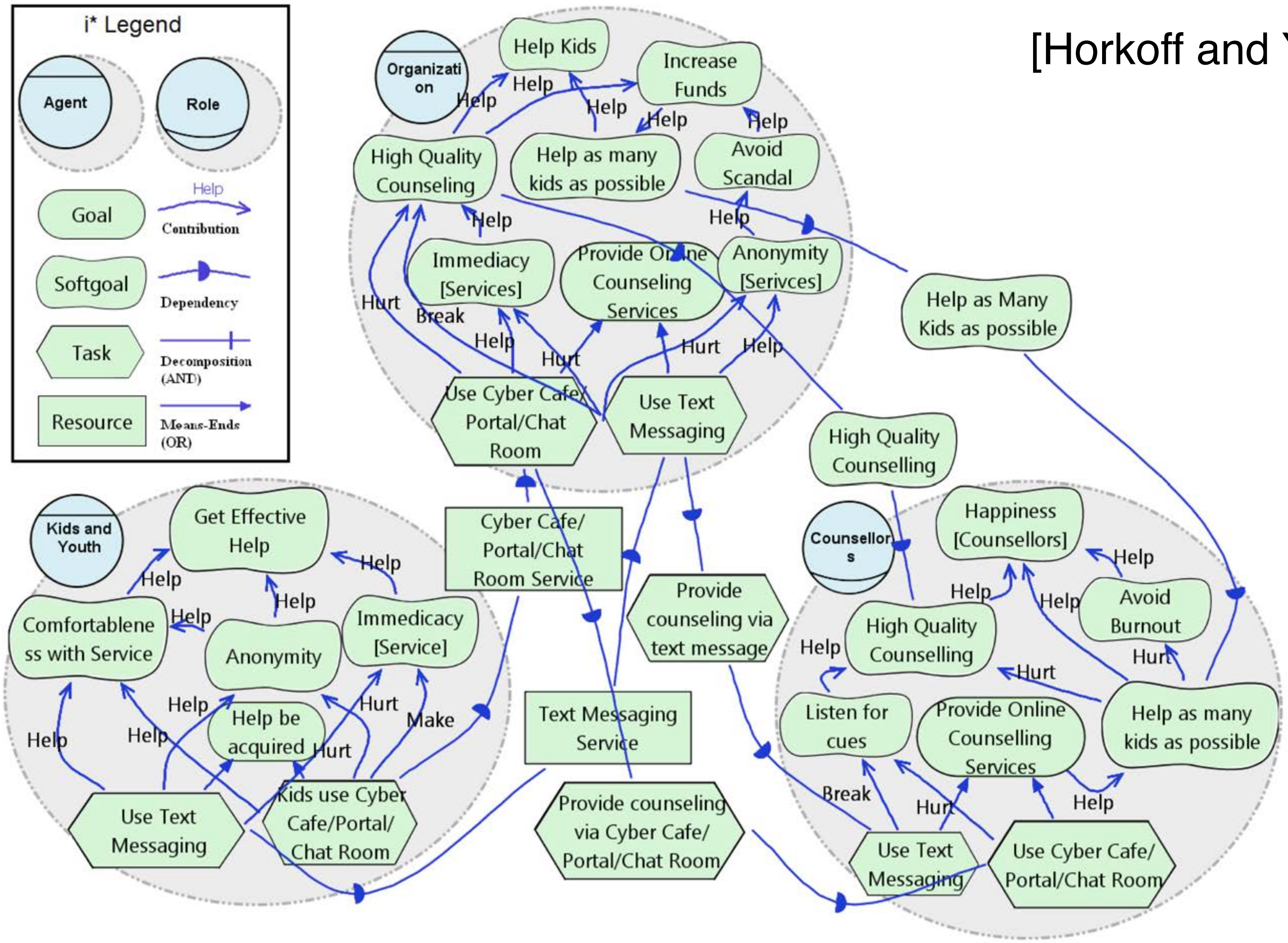


Goal-agent networks

- Explicitly models agents (stakeholders), their goals, tasks that achieve goals, resources, and dependencies between these items
- Many approaches in the RE literature
- i* is the most popular approach
- Rather infrequently used in practice

A real world i* example: Youth counseling

[Horkoff and Yu 2010]



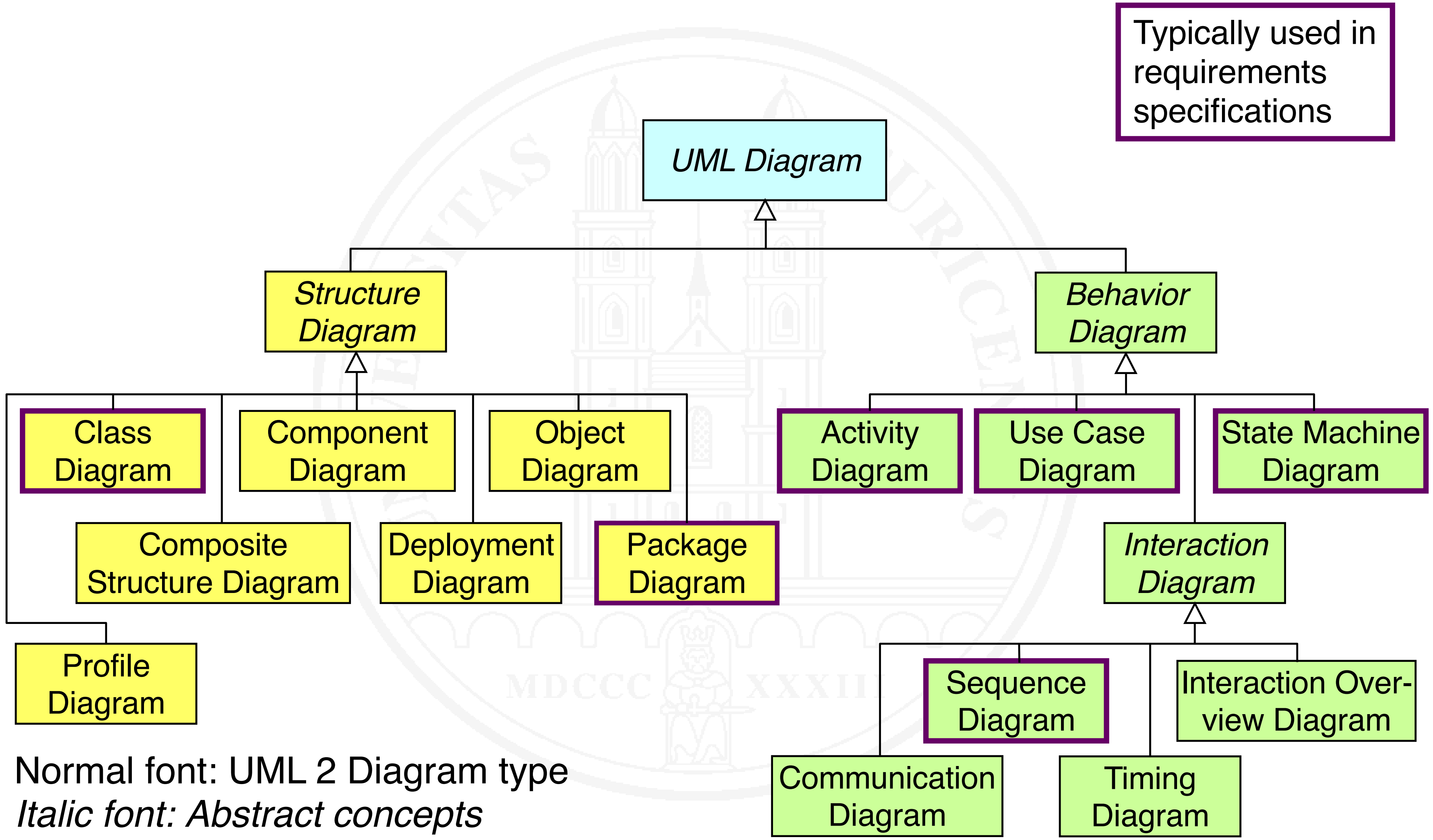
9.7 UML (Unified Modeling Language)

[Object Management Group 2017]

- UML is a collection of primarily graphic languages for expressing requirements models, design models, and deployment models from various perspectives
- A UML specification typically consists of a collection of loosely connected diagrams of various types
- Additional restrictions can be specified with the formal textual language OCL (Object Constraint Language)

[Object Management Group 2014]

UML – Overview of diagram types



Normal font: UML 2 Diagram type
Italic font: Abstract concepts

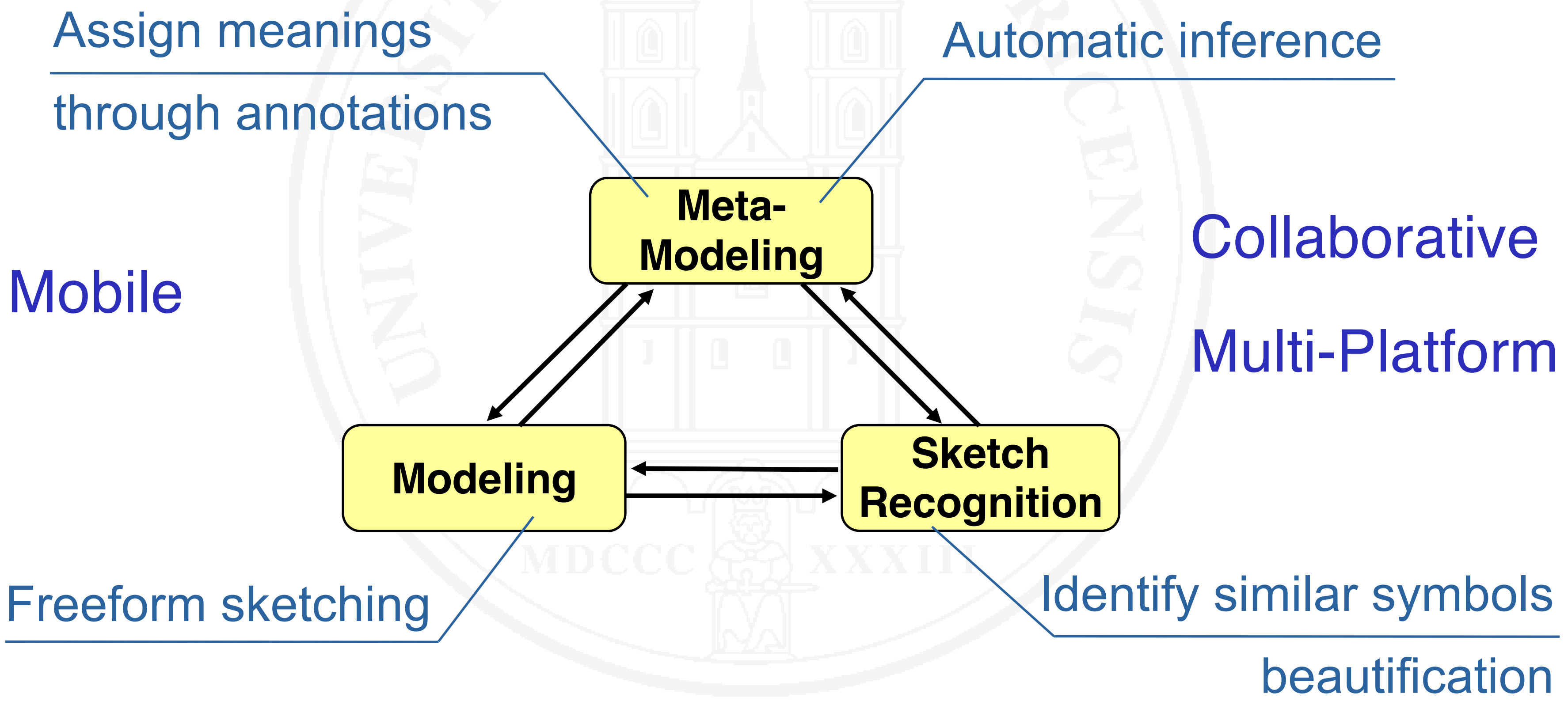
9.8 Lightweight, flexible modeling

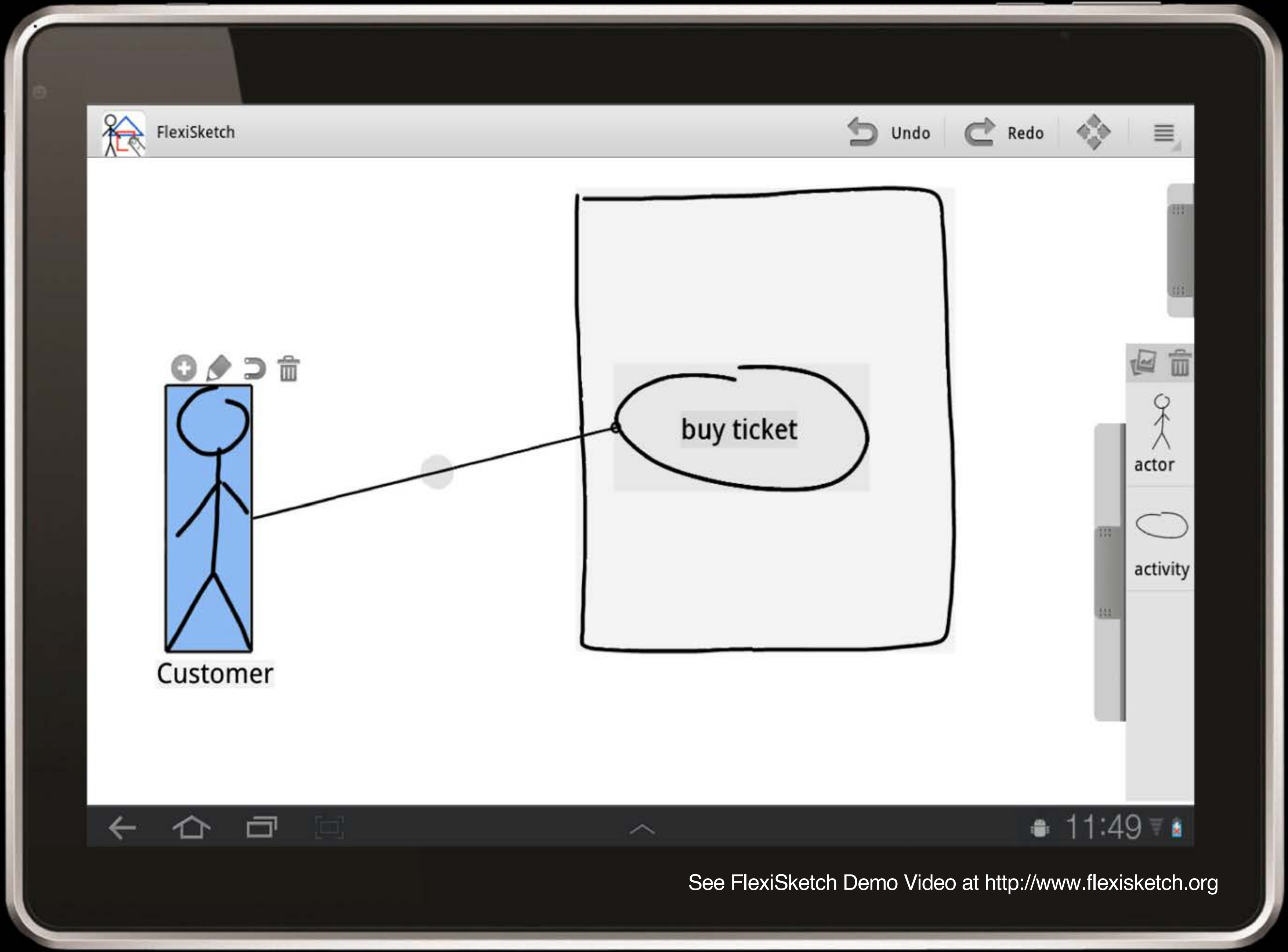
- Modeling languages – Have a predetermined syntax
  - Limited expressibility and flexibility
  - ➔ Too restrictive for sketching ideas or initial requirements
- Free-form sketching – Is fully flexible
  - Resulting sketches do not carry any structure or meanings
  - ➔ Too vague when sketches serve as a basis for further RE tasks
- Need for a middle-ground approach
  - High flexibility; no fixed set of language constructs
  - Co-evolution of models and model syntax & meanings
  - ➔ FlexiSketch

[Wüest, Seyff, Glinz 2019]
www.flexisketch.org

FlexiSketch – supporting flexible modeling

- Allow users to define their own notations & languages on the fly
- Co-evolve models and their metamodels





See FlexiSketch Demo Video at http://www.flexisketch.org

D. Wüest, N. Seyff, M. Glinz (2015) FlexiSketch Team: Collaborative Sketching and Notation Creation on the Fly. 37th International Conference on Software Engineering

10 Formal specification languages

Requirements models with formal syntax and semantics

The vision

- Analyze the problem
- Specify requirements formally
- Implement by correctness-preserving transformations
- Maintain the specification, no longer the code

Typical languages

- “Pure” Automata / Petri nets
- Algebraic specification
- Temporal logic: LTL, CTL
- Set&predicate-based models: Z, OCL, Alloy, B

What does “formal” mean?

- Formal calculus, i.e., a specification language with
  - formally defined syntax

    and
  - formally defined semantics
- Primarily for specifying functional requirements

Potential forms

- Purely descriptive, e.g., algebraic specification
- Purely constructive, e.g., Petri nets
- Model-based hybrid forms, e.g. Alloy, B, OCL, VDM, Z

10.1 Algebraic specification

- Developed mid 1970ies for specifying complex data types
- Signatures of operations define the syntax
- Axioms (expressions being always true) define semantics
- Axioms describe properties that are invariant

```
TYPE Stack
...
push:   (Stack, elem)  →   Stack;
...
¬  full(s) → empty(push(s,e)) = false
...
```

+ Purely descriptive and mathematically elegant
– Hard to read
– Over- and underspecification difficult to spot
– Has never made it from research into industrial practice

10.2 Model-based formal specification

- Mathematical model of system state and state change
- Based on sets, relations and logic expressions
- Typical language elements
  - Base sets
  - Relationships (relations, functions)
  - Invariants (predicates)
  - State changes (by relations or functions)
  - Assertions for states

The formal specification language landscape

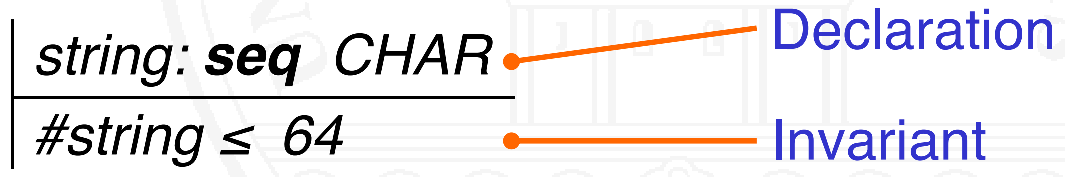
- VDM – Vienna Development Method (Björner and Jones 1978)
- Z (Spivey 1992)
- OCL (from 1997; OMG 2014)
- Alloy (Jackson 2002)
- B (Abrial 2009)

10.3 An overview of Z

- A typical model-based formal language
- Only basic concepts covered here
- More detail in the literature, e.g., Jacky (1997)

The basic elements of Z

- Z is set-based
- Specification consists of sets, types, axioms and schemata
- Types are elementary sets: *[Name]* *[Date]* *IN*
- Sets have a type: *Person:* $\mathbb{P}$ *Name* *Counter: IN*
- Axioms define global variables and their (invariant) properties



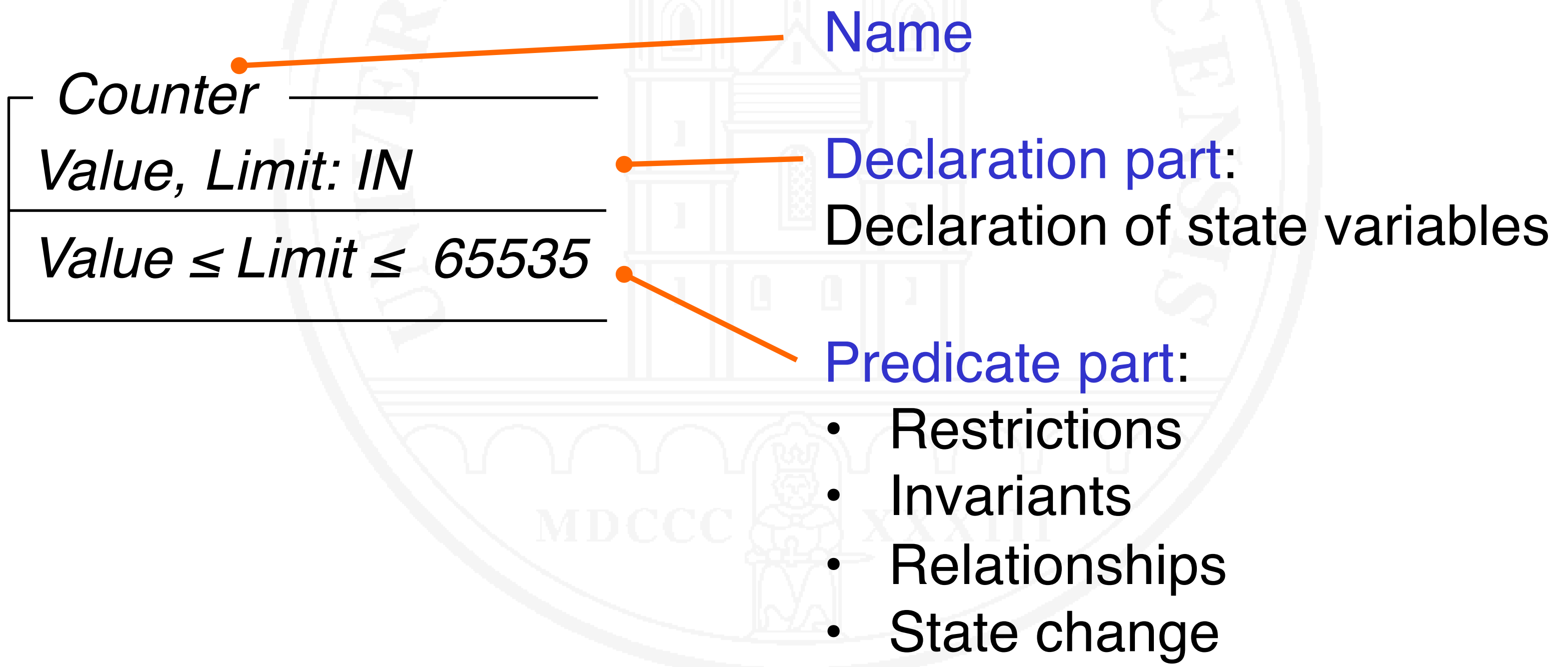
$string: \mathbf{seq}\ CHAR$

$\#string \leq 64$

IN	Set of natural numbers
$\mathbb{P}\,M$	Power set (set of all subsets) of *M*
seq	Sequence of elements
#M	Number of elements of set *M*

The basic elements of Z – 2

- Schemata
  - organize a Z-specification
  - constitute a name space



Relations, functions und operations

- Relations and functions are ordered set of tuples:

  Order: $\mathbb{P}$ *(Part x Supplier x Date)*

  A subset of all ordered triples (p, s, d) with p $\in$ *Part*, s $\in$ *supplier,* and d $\in$ *Date*

  Birthday: Person $\rightarrow$ *Date*

  A function assigning a date to a person, representing the person's birthday

State change through operations:

```
┌ Increment counter ──
│ ∆ Counter
├──────────────────────
│ Value < Limit
│ Value' = Value + 1
│ Limit' = Limit
└──────────────────────
```

Mathematical equality, no assignment!

Δ *S*	The sets defined in schema S will be changed
M'	State of set M after executing the operation

Example: specification of a library system

The library has a stock of books and a set of persons who are library users.

Books in stock may be borrowed.

```
┌ Library ──────────────────────────
│ Stock: ℙ Book
│ User: ℙ Person
│ lent: Book ⇸ Person
├───────────────────────────────────
│ dom lent ⊆ Stock
│ ran lent ⊆ User
└───────────────────────────────────
```

⇸	Partial function
dom	Domain ...
ran	Range...
	...of a relation

Example: specification of a library system – 2

Books in stock which currently are not lent to somebody may be borrowed

Borrow

Δ *Library*
BookToBeBorrowed?: Book
Borrower?: Person

BookToBeBorrowed? $\in$ *Stock\* **dom** *lent*
Borrower? $\in$ *User*
lent' = lent $\cup$ *{(BookToBeBorrowed?, Borrower?)}*
Stock' = Stock
User' = User

x?	*x* is an input variable
a $\in$ *X*	*a* is an element of set *X*
\\	Set difference operator
$\cup$	Set union operator

Example: specification of a library system – 3

It shall be possible to inquire whether a given book is available

```
┌─ InquireAvailability ──────────────────────────────
│ Ξ Library
│ InquiredBook?: Book
│ isAvailable!: {yes, no}
├────────────────────────────────────────────────────
│ InquiredBook? ∈ Stock
│ isAvailable! = if InquiredBook? ∉ dom lent
│                    then yes else no
│
└────────────────────────────────────────────────────
```

Ξ *S*	The sets defined in schema S can be referenced, but not changed
x!	x is an output variable

Mini-Exercise: Specifying in Z

Specify a system for granting and managing authorizations for a set of individual documents.

The following sets are given:

Authorization

Stock $\mathbb{P}$ *Document*
Employee: $\mathbb{P}$ *Person*
authorized: $\mathbb{P}$ *(Document x Person)*
prohibited: $\mathbb{P}$ *(Document x Date)*

Specify an operation for granting an employee access to a document as long as access to this document is not prohibited. Use a Z-schema.

.

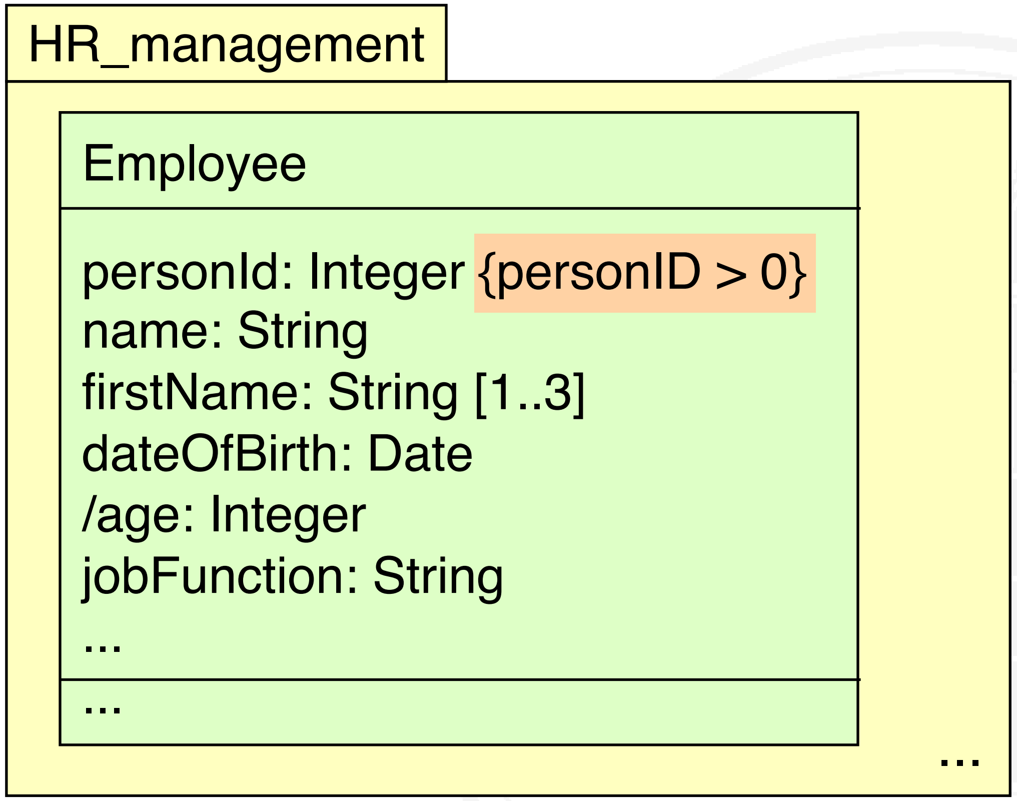
10.4 OCL (Object Constraint Language)

- What is OCL?
  - A textual formal language
  - Serves for making UML models more precise
  - Every OCL expression is attached to an UML model element, giving the context for that expression
  - Originally developed by IBM as a formal language for expressing integrity constraints (called ICL)
  - In 1997 integrated into UML 1.1
  - Current standardized version is Version 2.4 of 2014

Why OCL?

- Making UML models more precise
  - Specification of Invariants (i.e., additional restrictions) on UML models
  - Specification of the semantics of operations in UML models
- Also usable as a language to query UML models

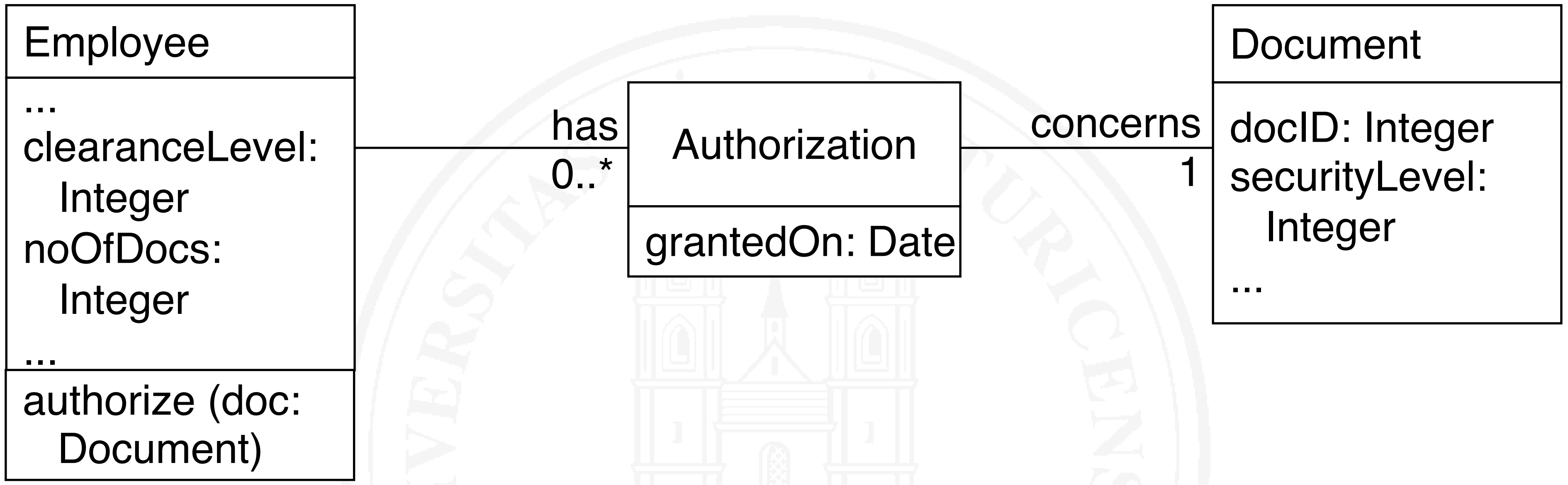
OCL expressions: invariants



context HR_manangement::Employee **inv**:
self.jobFunction = "driver" **implies** self.age ≥ 18

- OCL expression may be part of a UML model element
- Context for OCL expression is given implicitly
- OCL expression may be written separately
- Context must be specified explicitly

OCL expressions: Semantics of operations



context Employee::authorize (doc: Document)
  pre: self.clearanceLevel ≥ doc.securityLevel
  post: noOfDocs = noOfDocs@pre + 1
           and
           self.has->**exists** (a: Authorization | a.concerns = doc)

Navigation, statements about sets in OCL

- Persons having Clearance level 0 can't be authorized for any document:

context Employee **inv**: self.clearanceLevel = 0 **implies**
self.has->isEmpty()

Navigation from current object to a set of associated objects

Application of a function to a set of objects

Navigation, statements about sets in OCL – 2

More examples:

- The number of documents listed for an employee must be equal to the number of associated authorizations:

  context Employee **inv**: self.has->size() = self.noOfDocs

- The documents authorized for an employee are different from each other

  context Employee **inv**: self.has->**forAll** (a1, a2: Authorization I
    a1 <> a2 **implies** a1.concerns.docID <> a2.concerns.docID)

- There are no more than 1000 documents:

  context Document **inv**: Document.allInstances()->size() ≤ 1000

Summary of important OCL constructs

- Kind and context: **context**, **inv**, **pre**, **post**
- Boolean logic expressions: **and**, **or**, **not**, **implies**
- Predicates: **exists**, **forAll**
- Alternative: **if then else**
- Set operations: size(), isEmpty(), notEmpty(), sum(), ...
- Model reflection, e.g., *self.oclIsTypeOf (Employee)* is true in the context of Employee
- Statements about all instances of a class: allInstances()
- Navigation: dot notation self.has.date = ...
- Operations on sets: arrow notation self.has->size()
- State change: @pre notation noOfDocs = noOfDocs@pre + 1

10.5 Proving properties

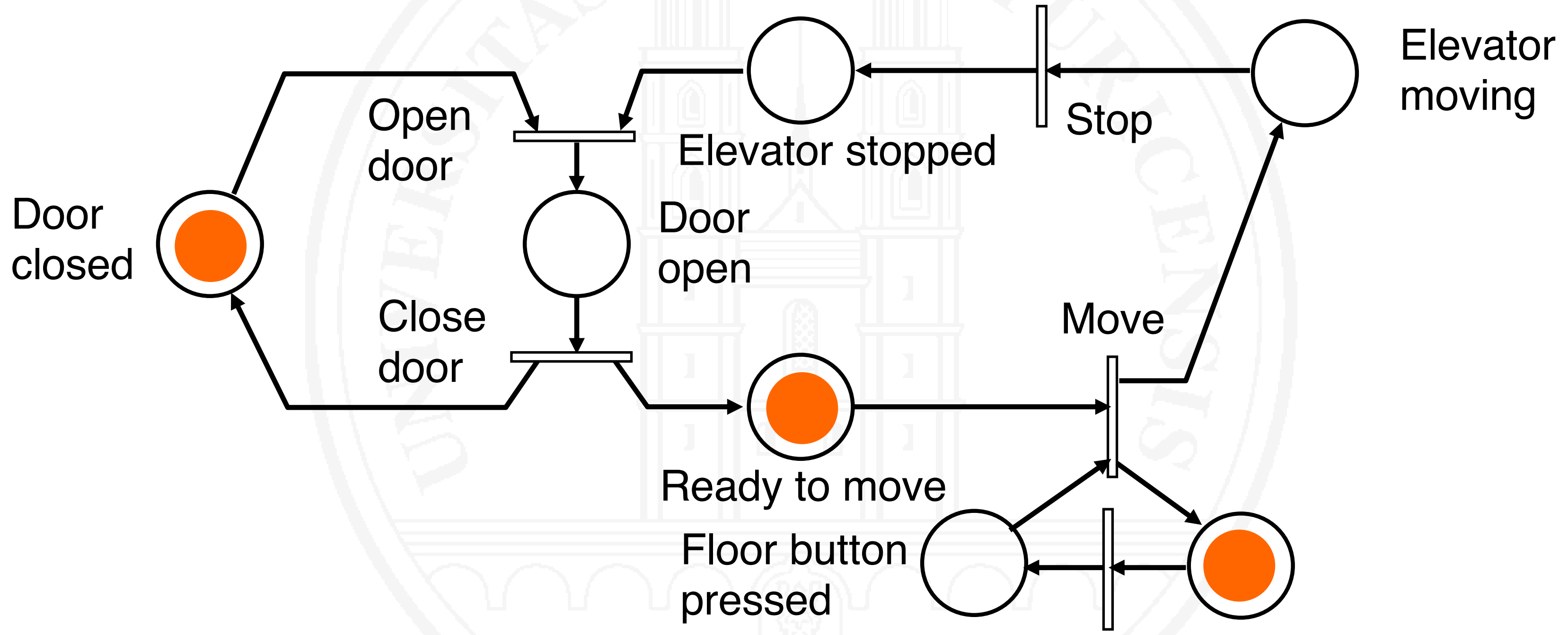
With formal specifications, we can prove if a model has some required properties (e.g., safety-critical invariants)

- Classic proofs (usually supported by theorem proving software) establish that a property can be inferred from a set of given logical statements
- Model checking explores the full state space of a model, demonstrating that a property holds in every possible state

– Classic proofs are still hard and labor-intensive
+ Model checking is fully automatic and produces counter-examples in case of failure
– Exploring the full state state space is frequently infeasible
+ Exploring feasible subsets is a systematic, automated test

Example: Proving a safety property

A (strongly simplified) elevator control system has been modeled with a Petri net as follows:



The property that an elevator never moves with doors open shall be proved

Example: Proving a safety property – 2

The property to be proven can be restated as:

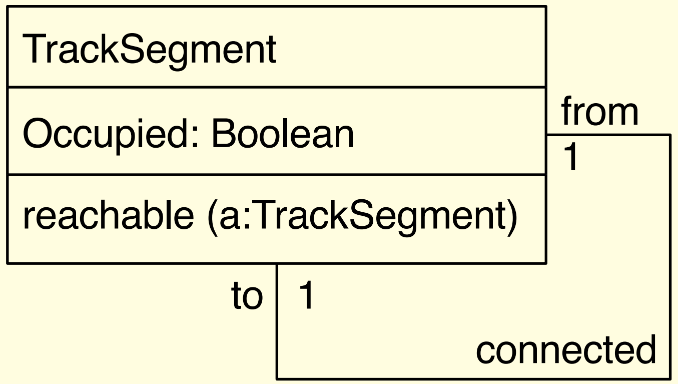
(P) The places *Door open* and *Elevator moving* never hold tokens at the same time

Due to the definition of elementary Petri Nets we have

- The transition *Move* can only fire if *Ready to move* has a token (1)
- There is at most one token in the cycle *Ready to move – Elevator moving – Elevator stopped – Door open* (2)
- (2) $\Rightarrow$ If *Ready to move* or *Elevator moving* have a token, *Door open* hasn't one (3)
- If *Door open* has no token, *Door closed* must have one (4)
- (1) & (3) & (4) $\Rightarrow$ (P) ▯

Mini-Exercise: A circular metro line

A circular metro line with 10 track segments has been modeled in UML and OCL as follows:



Context TrackSegment::
  reachable (a: TrackSegment): Boolean
  post:
  result = (self.to = a) **or** (self.to.reachable (a))

context TrackSegment **inv**:
  TrackSegment.allInstances->size = 10

In a circle, every track segment must be reachable from every other track segment (including itself). So we must have:

context TrackSegment **inv** (1)
  TrackSegment.allInstances->forAll (x, y l x.reachable (y))

a) Falsify this invariant by finding a counter-example

Mini-Exercise: A circular metro line – 2

Only the following trivial invariant can be proved:

> **context** TrackSegment **inv**:
> TrackSegment.allInstances->forAll (x I x.reachable (x))

b) Prove this invariant using the definition of *reachable*

Obviously, this model of a circular metro line is wrong. The property of being circular is not mapped correctly to the model.

c) How can you modify the model such that the original invariant (1) holds?

10.6 Benefits and limitations, practical use

Benefits

- Unambiguous by definition
- Fully verifiable
- Important properties can be
  - proven
  - or tested automatically (model checking)

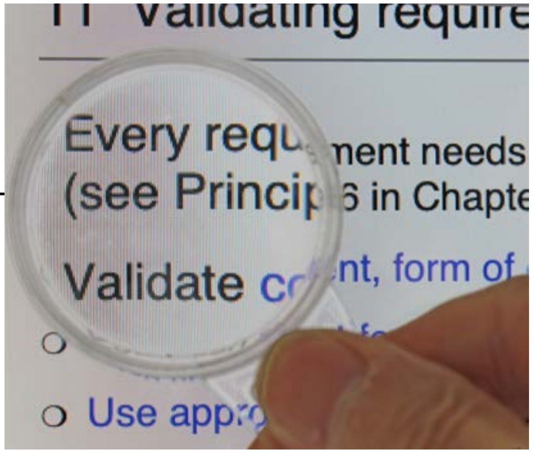
Limitations / problems

- Cost vs. value
- Stakeholders can’t read the specification: how to validate?
- Primarily for functional requirements

Role of formal specifications in practice

- Marginally used in practice
  - Despite its advantages
  - Despite intensive research (research on algebraic specifications dates back to 1977)
- Actual situation today
  - Punctual use possible and reasonable
  - In particular for safety-critical components
  - However, broad usage
    - not possible (due to validation problems)
    - not reasonable (cost exceeds benefit)
- Another option: semi-formal models where critical parts are fully formalized

11 Validating requirements



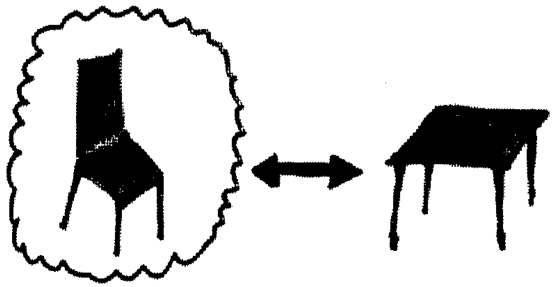
Every requirement needs to be validated
(see Principle 6 in Chapter 2)

- Content:
  - Stakeholders' desires and needs adequately covered?
  - Requirements agreed?
- Work products: Requirements documented well?
- Context: Assumptions reasonable?

Important validation aspects

- Involvement of the right stakeholders
- Separating the identification and the correction of defects
- Validation from different views
- Repeated / continuous validation
- Use appropriate techniques

Validation of content



Identify requirements that are

- Inadequate or wrong
- Incomprehensible
- Incomplete or missing
- Ambiguous

Also look for requirements with these quality defects:

- Not verifiable
- Unnecessary
- Infeasible
- Premature design decisions

Validation of content: Agreement

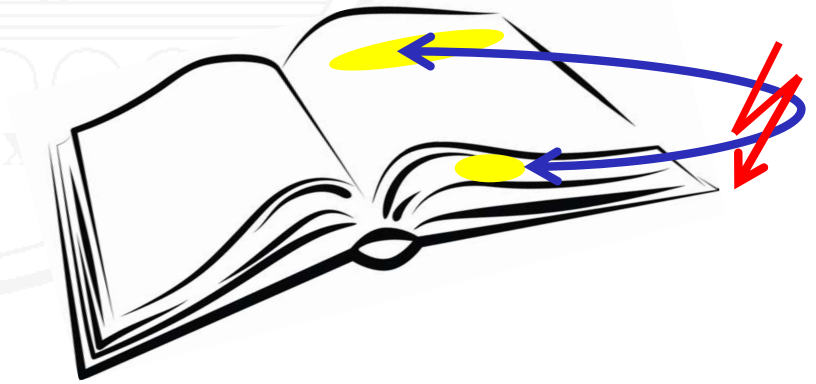
- Requirements elicitation involves achieving consensus among stakeholders having divergent needs
- When validating requirements, we have to check whether agreement has actually been achieved
  - All known conflicts resolved?
  - For all requirements: have all relevant stakeholders for a requirement agreed to this requirement in its documented form?
  - For every changed requirement, have all relevant stakeholders agreed to this change?

Validation of requirements work products

Scope: checking the requirements work products (e.g., a systems requirements specification or a collection of user stories) for formal problems

Identify requirements that are

- Inconsistent with each other
- Missing
- Non-conforming to documentation rules, structure or format
- Redundant
- Badly structured
- Hard to modify
- Not traceable



Context validation

- Context assumptions reasonable?
- Mappings from context phenomena to system inputs / outputs adequate?
- Can we reasonably argue that the domain requirements will be met when the system will be built and deployed as specifiend in the requirements?

Requirements validation techniques

Review

- Main means for requirements validation
- Walkthrough: author guides experts through the specification
- Inspection: Experts check the specification
- Author-reviewer-cycle: Requirements engineer continuously feeds back requirements to stakeholder(s) for review and receives feedback

Construction of other work products

- Acceptance criteria / test cases help disambiguate / clarify requirements
- Writing user manuals or creating models for textual requirements may help identify missing or wrong requirements

Requirements validation techniques – 2

Prototyping

- Lets stakeholders judge the practical usefulness of the specified system in its real application context
- Prototype constitutes a sample model for the system-to-be
- Most powerful, but also most expensive means of requirements validation

Simulation/Animation

- Means for investigating dynamic system behavior
- Simulator executes specification and may visualize it by animated models

Requirements validation techniques – 3

Testing (when evolving an existing system)

- A/B testing
- Classic alpha and beta testing of source code

Requirements Engineering tools

- Help find gaps and contradictions

Formal Verification / Model Checking / Model Analysis

- Formal proof of critical properties
- Automated, systematic and comprehensive test of critical properties (when proofs are not tractable)

Reviewing practices

- Paraphrasing
  - Explaining the requirements in the reviewer’s own words
- Perspective-based reading
  - Analyzing requirements from different perspectives, e.g., end-user, tester, architect, maintainer,...
- Playing and executing
  - Playing scenarios
  - Mentally executing acceptance test cases
- Checklists
  - Using checklists for guiding and structuring the review process

Requirements negotiation

- Requirements negotiation implies
  - Identification of conflicts
  - Conflict analysis
  - Conflict resolution
  - Documentation of resolution

- Requirements negotiation can happen
  - While eliciting requirements
  - When validating requirements

Conflict analysis

Identifying the underlying reasons of a conflict helps select appropriate resolution techniques

Typical underlying reasons are

- Subject matter conflict (divergent factual needs)
- Data conflict (different interpretation of data, inconsistent data)
- Interest conflict (divergent interests, e.g., cost vs. function)
- Value conflict (divergent values and preferences)
- Relationship conflict (emotional problems in personal relationships between stakeholders)
- Organizational conflict (between stakeholders on different hierarchy and decision power levels in an organization)

Conflict resolution

- Various strategies / techniques
- Conflicting stakeholders must be involved in resolution
- Win-win techniques
  - Agreement
  - Compromise
  - Build variants
- Win-lose techniques
  - Overruling
  - Voting
  - Prioritizing stakeholders (important stakeholders override less important ones)

Conflict resolution – 2

- Decision support techniques
  - PMI (Plus-Minus-Interesting) categorization of potential conflict resolution decisions
  - Decision matrix (Matrix with a row per interesting criterion and a column per potential resolution alternative. The cells contain relative weights which can be summarized per column and then compared)

Acceptance testing

DEFINITION. Acceptance – The process of assessing whether a system satisfies all its requirements.

DEFINITION. Acceptance test – A test that assesses whether a system satisfies its requirements.

Requirements and acceptance testing

Requirements engineering and acceptance testing are naturally intertwined

- For every requirement, there should be at least one acceptance test case
- Requirements should be written such that acceptance tests can be written to verify them (→ verifiability)
- Acceptance test cases can serve
  - for disambiguating requirements
  - as detailed specifications by example → acceptance criteria for user stories

Choosing acceptance test cases

Potential coverage criteria:

- Requirements coverage: At least one case per requirement
- Function coverage: At least one case per function
- Scenario coverage: For every type scenario / use case
  - All actions covered
  - All branches covered

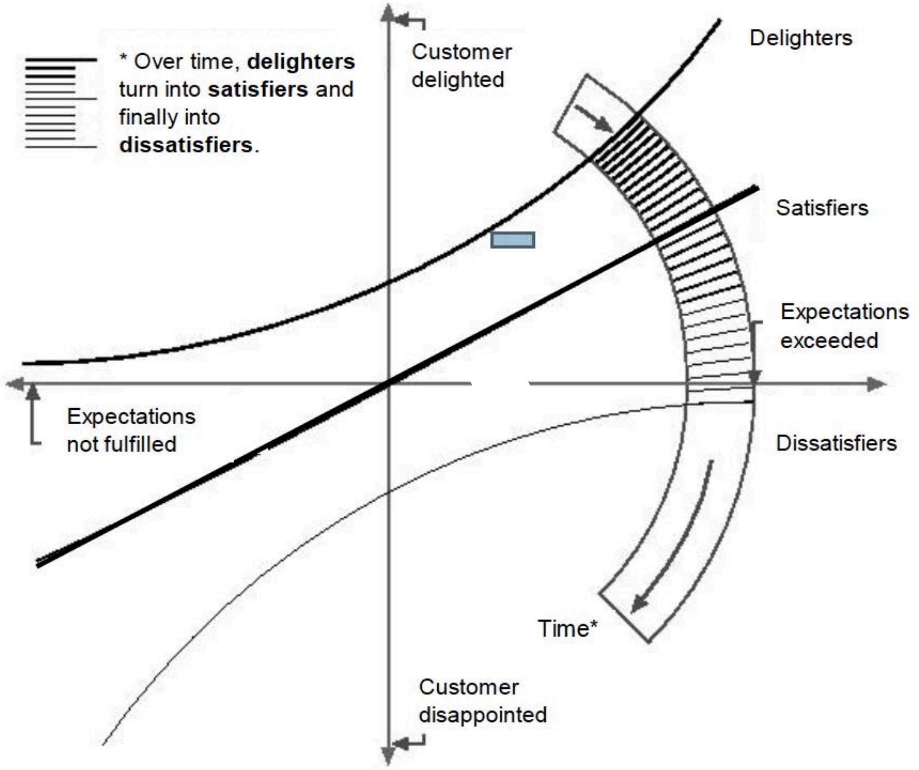
- Consider the usage profile: not all functions/scenarios are equally frequent / important

12 Innovative requirements

Satisfying stakeholders is not enough (see Principle 8 in Chapter 2)

[Kano et al. 1984]

- Kano's model helps identify...
  - what is implicitly expected (dissatisfiers)
  - what is explicitly required (satisfiers)
  - what the stakeholders don't know, but would delight them if they get it: innovative requirements



* Over time, **delighters** turn into **satisfiers** and finally into **dissatisfiers**.
Customer delighted
Delighters
Satisfiers
Expectations exceeded
Expectations not fulfilled
Dissatisfiers
Time*
Customer disappointed

- Over time, delighters degrade toward plain expectations

How to create innovative requirements?

Encourage out-of-the-box thinking

- Stimulate the stakeholders' creativity
  - Imagine/ make up scenarios for possible futures
  - Imagine a world without constraints and regulators
  - Find and explore metaphors
  - Study other domains
- Involve solution experts and explore what's possible with available and future technology
- Involve smart people without domain knowledge

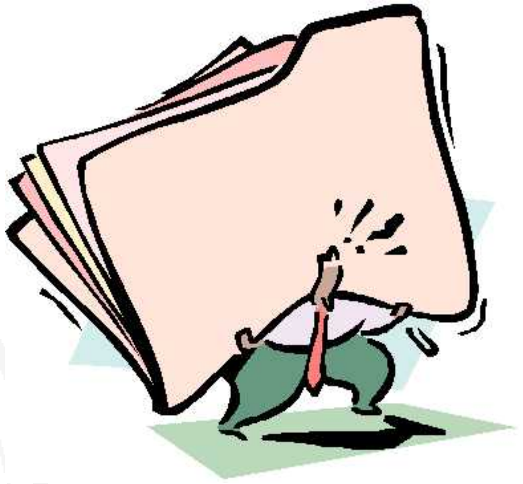
[Maiden, Gitzikis and Robertson 2004]
[Maiden and Robertson 2005]

Where to innovate

- Functionality – new exciting features
- Performance – not just a bit more, but significantly more powerful than previous or competing systems
- Usability – making usage an exciting experience

13 Requirements management

- Organize
  - Store and retrieve
  - Record metadata (author, status,...)
- Prioritize
- Keep track: dependencies, traceability
- Manage change



13.1 Organizing requirements

Every requirement needs

- a unique identifier as a reference in acceptance tests, review findings, change requests, traces to other artifacts, etc.
- some metadata, e.g.
  - Author
  - Date created
  - Date last modified
  - Source (stakeholder(s), document, minutes, observation...)
  - Status (created, ready, released, rejected, postponed...)
  - Necessity (critical, major, minor)

Storing, retrieving and querying

Storage

- Paper and folders
- Files and electronic folders
- A requirements management tool

Retrieving support

- Keywords
- Cross referencing
- Search machine technology

Querying

- Selective views (all requirements matching the query)
- Condensed views (for example, statistics)

13.2 Prioritizing requirements

- Requirements may be prioritized with respect to various criteria, for example
  - Necessity
  - Cost of implementation
  - Time to implement
  - Risk
  - Volatility

- Prioritization is done by the stakeholders
- Only a subset of all requirements may be prioritized
- Requirements to be prioritized should be on the same level of abstraction

Simple prioritization (by necessity)

Ranks all requirements in three categories with respect to necessity, i.e., their importance for the success of the system

- Critical (also called essential, or mandatory)

  The system will not be accepted if such a requirement is not met

- Major (also called conditional, desirable, important, or optional)

  The system should meet these requirements, but not meeting them is no showstopper

- Minor (also called nice-to-have, or optional)

  Implementing these requirements is nice, but not needed

Selected prioritization techniques

Single criterion prioritization

- Simple ranking

  Stakeholders rank a set of requirements according to a given criterion

- Assigning points

  Stakeholders receive a total of n points that they distribute among m requirements

- Prioritization by multiple stakeholders may be consolidated using weighted averages. The weight of a stakeholder depends on his/her importance

Selected prioritization techniques – 2

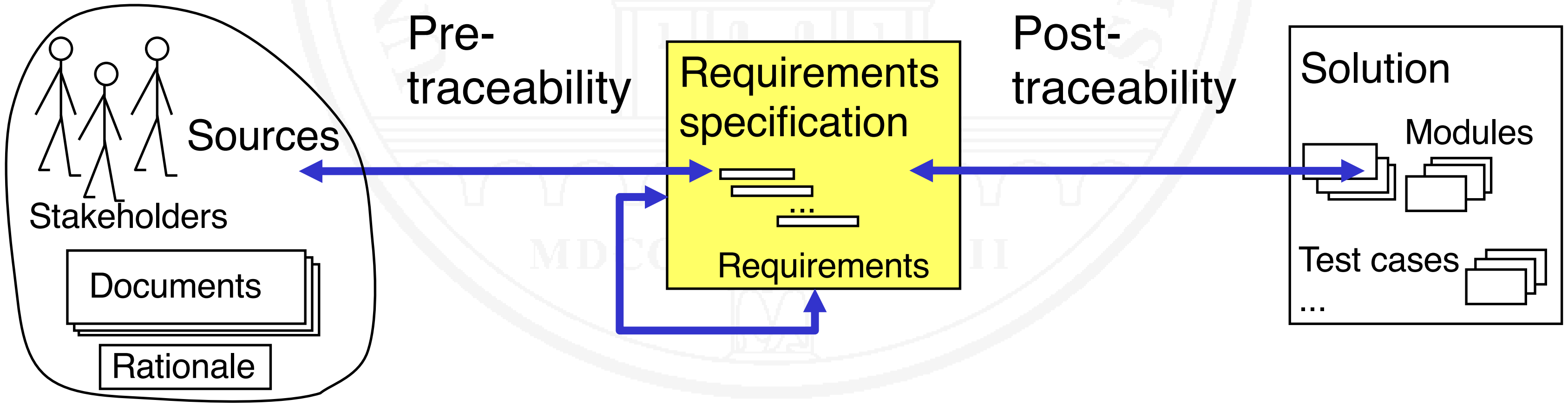
Multiple criterion prioritization

- Wiegers’ matrix [Wiegers 1999]
  - Estimates relative benefit, detriment, cost, and risk for each requirement
  - Uses these values to calculate a weighted priority
  - Ranks according to calculated priority values
- AHP (Analytic Hierarchy Process) [Saaty 1980]
  - An algorithmic multi-criterion decision making process
  - Applicable for prioritization by a group of stakeholders

13.3 Traceability

[Gotel and Finkelstein 1994]

DEFINITION. Traceability – The ability to trace a requirement
(1) back to its origins,
(2) forward to its implementation in design and code,
(3) to requirements it depends on (and vice-versa).
Origins may be stakeholders, documents, rationale, etc.



Establishing and maintaining traces

- Manually
  - Requirements engineers explicitly create traces when creating artifacts to be traced
  - Tool support required for maintaining and exploring traces
  - Every requirements change requires updating the traces
  - High manual effort; cost and benefit need to be balanced
- Automatic
  - Automatically create candidate trace links between two artifacts (for example, a requirements specification and a set of acceptance test cases)
  - Uses information retrieval technology
  - Requires manual post processing of candidate links

13.4 Requirements evolution

The problem (see Principle 7 in Chapter 2):

Keeping requirements stable...

... while permitting requirements to change

Potential solutions

- Agile / iterative development with short development cycles (1-6 weeks)
- Explicit requirements change management

Every solution to this problem further needs requirements configuration management

Requirements configuration management

Keeping track of changed requirements

- Versioning of requirements
- Ability to create requirements configurations, baselines and releases
- Tracing the reasons for a change, for example
  - Stakeholder demand
  - Bug reports / improvement suggestions
  - Market demand
  - Changed regulations

Classic requirements change management

Adhering to a strict change process

(1) Submit change request

(2) Triage. Result: [OK I NO I Later (add to backlog)]

(3) If OK: Perform impact analysis

(4) Submit result and recommendation to Change Control Board

(5) Decision by Change Control Board

(6) If positive: make the change, create new baseline/release, (maybe) adapt the contract between client and supplier

Change control board – A committee of customer and supplier representatives that decides on change requests.

Requirements change in agile development

In agile and iterative development processes, a requirements change request ...

- ... never affects the current sprint / iteration, thus ensuring stability
- ... is added to the product backlog

Decisions about change requests are made when prioritizing and selecting the requirements for the subsequent sprints / iterations

14 Requirements and design

A traditional belief:

- Requirements are about what a system ought to do
- Design deals with the problem of how to realize what has been stated in the requirements
- Requirements Engineering and System Design should be kept separate, with requirements preceding design

- Sounds good and is popular, but does not work

Design has two facets

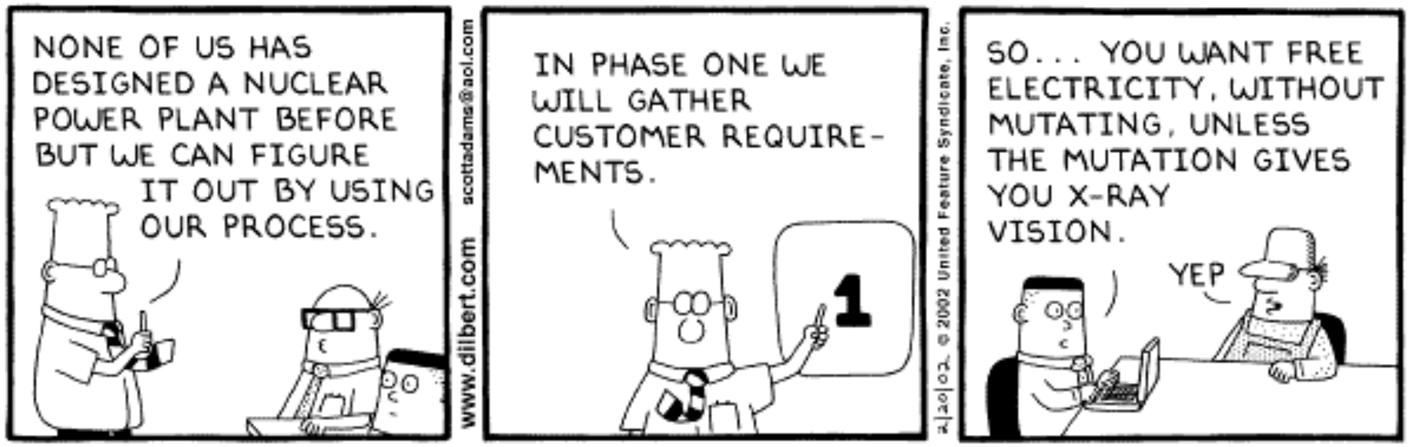
- Technical Design: Creating the architectural structure of a system and designing its components in detail
- Product Design: Shaping a product (or a system) with respect to its capabilities, behavior, outer form, and usage

Traditional RE: Product Design comes after RE

Modern RE: Product design shapes the essence of a product
→ crucial for meeting the stakeholders' desires and needs
→ Product Design and RE are strongly intertwined

Product design for digital products is also called "Digital Design"

Why care about both RE and product design?


NONE OF US HAS DESIGNED A NUCLEAR POWER PLANT BEFORE BUT WE CAN FIGURE IT OUT BY USING OUR PROCESS.
IN PHASE ONE WE WILL GATHER CUSTOMER REQUIRE-MENTS.
1
SO... YOU WANT FREE ELECTRICITY, WITHOUT MUTATING, UNLESS THE MUTATION GIVES YOU X-RAY VISION.
YEP
www.dilbert.com scottadams@aol.com
2/20/02 © 2002 United Feature Syndicate, Inc.

→ We need RE competencies

I DESIGNED THE USER INTERFACE MYSELF. HOW DO YOU LIKE THE COLORS?
PUKE
FLU?
INTERFACE DESIGN.
www.dilbert.com scottadams@aol.com
9/23/02 © 2002 United Feature Syndicate, Inc.

→ and product design competencies

Complementary contributions

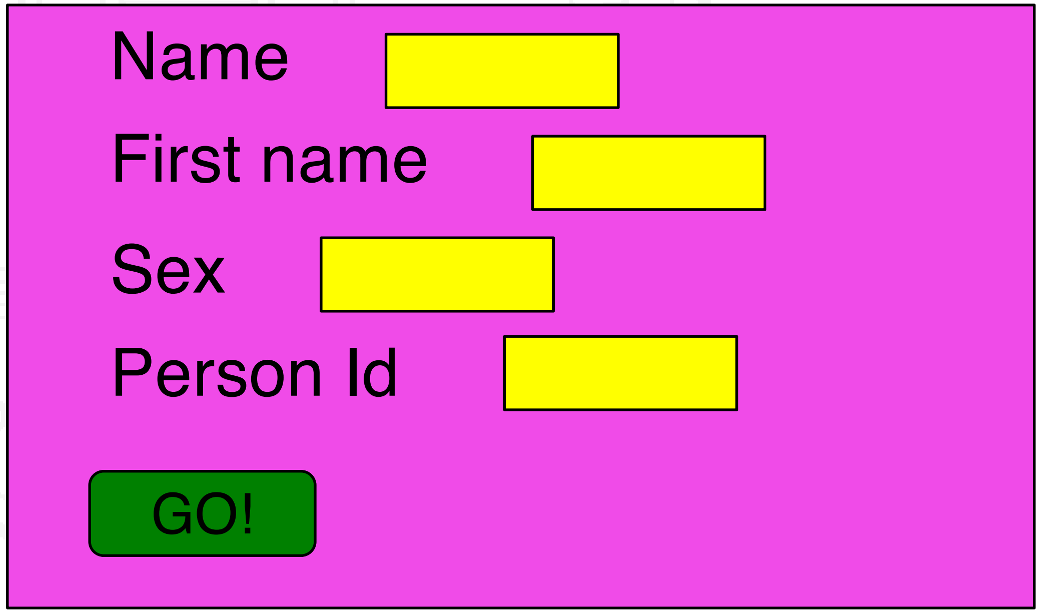
- RE contributes competencies about
  - Stakeholder identification
  - Elicitation of wishes and needs
  - Documentation of non-touchable things
  - Requirements negotiation, prioritization, and validation
- Product Design contributes competencies about
  - Usability
  - User experience design
  - Materials for physical & cyber-physical products, "digital materials" for digital products
  - Empirical product validation

Meeting requirements may not suffice to satisfy stakeholders

A requirement

The participant entry form shall have fields for the participant data *name*, *first name*, *sex*, and *person ID* and a *submit button*.

can be ruined by bad product design


Name
First name
Sex
Person Id
GO!