Part I: The Fundamentals

Part II: Requirements Engineering Practices

Part III: Enablers and Stumbling Blocks

Conclusions

References

5 Documenting requirements

Bridging the gap:

Stakeholders

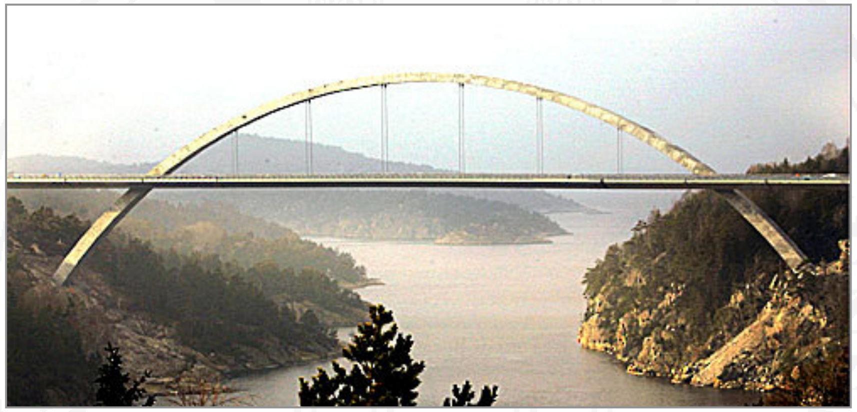


Photo © Lise Aserud / DPA

System builders

The need:

- Communicating requirements
- Having a basis for contracts and acceptance decisions

The means: A requirements specification document

5.1 Document types

Various document types, depending on RE process and specification purpose

- Stakeholder requirements specification (also called customer requirements specification)
  What the stakeholders want (independent of any system providing it)
- System requirements specification
  The system to be developed and its context
- Software requirements specification
  If the system is a pure software system

Document types – 2

- Business requirements specification
  High-level specification of business needs or goals
- Collection of user stories and/or task descriptions
  Used in agile software development

Stakeholder requirements specification

- Written when stakeholder needs shall be documented before any system development considerations are made
- Typically written by domain experts on the customer side (maybe with help of RE consultants)
- If a stakeholder requirements specification is written, it precedes and informs system or software requirements specifications

System/software requirements specification

- The classic form of a requirements specification
- No methodological difference between system requirements specification and software requirements specification
- Typically written by requirements engineers on the supplier side

5.2 Aspects to be documented

Independently of any language and method,
four aspects need to be documented:

- Functionality
  - Data: Usage and structure
  - Functions: Results, preconditions, processing
  - Behavior: Dynamic system behavior as observable by users
  - Both normal and abnormal cases must be specified

Aspects to be documented – 2

- Performance
  - Data volume
  - Reaction time
  - Processing speed
  - Specify measurable values if possible
  - Specify more than just average values

- Specific qualities
  - “-ilities” such as
    - Usability
    - Reliability
    - etc.

Aspects to be documented – 3

- Constraints

  Restrictions that must be obeyed / satisfied

  - Technical: given interfaces or protocols, etc.
  - Legal: laws, standards, regulations
  - Cultural
  - Environmental
  - Physical
  - Solutions / restrictions demanded by important stakeholders

5.3 How to document

Sample documentation standards

IEEE Std 830-1998 (outdated, but still in use)

- Three parts
- System requirements only
- Representation of specific requirements tailorable

VOLERE

- 27 chapters
- System and project requirements

Enterprise-specific standards

- Imposed by customer or given by supplier

IEEE Std 830-1998

1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions, acronyms, and abbreviations
   - 1.4 References
   - 1.5 Overview
2. Overall description
   - 2.1 Product perspective
   - 2.2 Product functions
   - 2.3 User characteristics
   - 2.4 Constraints
   - 2.5 Assumptions and dependencies
3. Specific requirements

Appendixes

Index

Variants:
Organize by
- Mode
- User class
- Object
- Feature
- Stimulus
- Function

VOLERE

[Robertson and Robertson 2006]
[www.volere.co.uk]

Project Drivers
1. The Purpose of the Project
2. Client, Customer & other Stakeholders
3. Users of the Product

Project Constraints

4. Mandated Constraints
5. Naming Conventions and Definitions
6. Relevant Facts and Assumptions

Functional Requirements

7. The Scope of the Work
8. The Scope of the Product
9. Functional and Data Requirements

Non-Functional Requirements

10. Look and Feel Requirements
11. Usability and Humanity Requirements
12. Performance Requirements
13. Operational Requirements
14. Maintainability and Support Requirements
15. Security Requirements
16. Cultural and Political Requirements
17. Legal Requirements

Project Issues

18. Open Issues
19. Off-the-Shelf Solutions
20. New Problems

22. Tasks
22. Cutover
23. Risks
24. Costs
25. User Documentation and Training
26. Waiting Room
27. Ideas for Solutions

How to document – language options

Informally

- Natural language (narrative text)

Semi-formally

- Structural models
- Interaction models

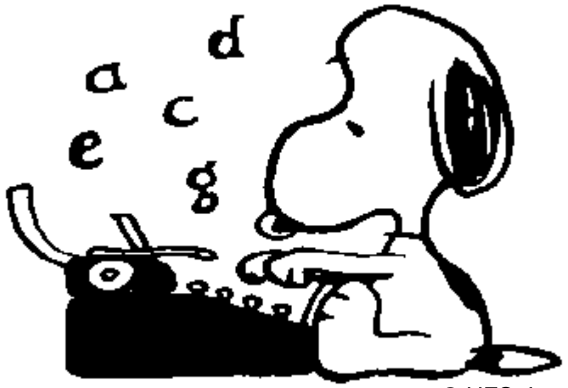
} Typically as diagrams which are enriched with natural langue texts

Formally

- Formal models, typically based on mathematical logic and set theory

General rules for requirements documentation

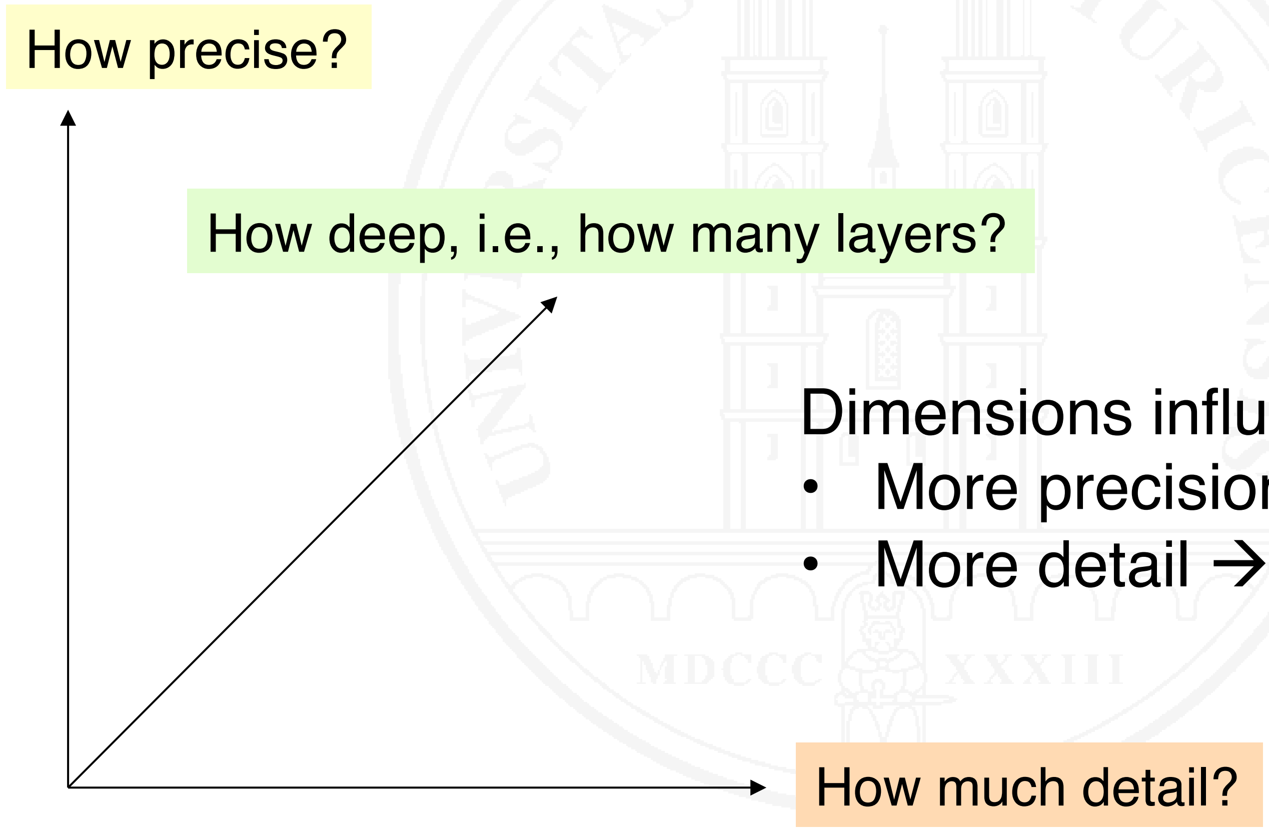
- Specify every requirement as a small, identifiable unit
- Add metadata such as source, author, date, status
- Build the requirements document according to some structure template
- Adapt the degree of detail to the risk associated with a requirement
- Specify normal and exceptional cases
- Don’t forget quality requirements and constraints



© UFS, Inc.

Precision – Detail – Depth

Three dimensions:



Dimensions influence each other:

- More precision → more detail
- More detail → more depth

Precision: reduce ambiguity

Restrict your language

Use a glossary

Define acceptance test cases

Quantify where appropriate

Formalize

Meine Liebe, Du fehlst mir morgens, mittags und abends.
DAS IST ZU UNGENAU...
5-8
WENN DU EINEM MÄDCHEN SCHREIBST, MUSST DU DICH PRÄZISE AUSDRÜCKEN.
Du fehlst mir um 8 Uhr 15, 11 Uhr 45 und 21 Uhr 36...
© 1985 United Feature Syndicate, Inc.

Snoopy quantifies ... unfortunately, I have it only in German

Detail

What's better?

"The participant entry form has fields for name, first name, sex, ..."

"The participant entry form has the following fields (in this order): Name (40 characters, required), First Name (40 characters, required), Sex (two radio buttons labeled male and female, selections exclude each other, no default, required),..."

It depends.

- Degree of implicit shared understanding of problem
- Degree of freedom left to designers and programmers
- Cost vs. value of detailed specification
- The risk you are willing to take

Depth

The more precise, the more information is needed

➜ Preserve readability with a hierarchical structure

"...

4.3 Administration of participants

- 4.3.1 Entering a new participant
  - 4.3.1.1 New participant entry form
  - 4.3.1.2 New participant confirmation
- 4.3.2 Updating a participant record

..."

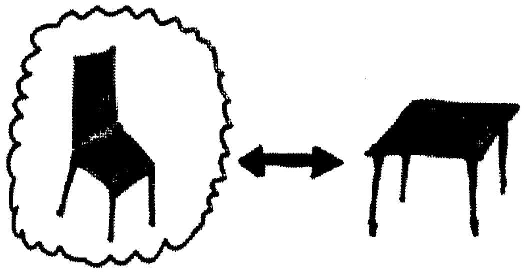
5.4 Quality of documented requirements

Two aspects of requirements quality

- ❍ Quality of individual requirements
- ❍ Quality of requirements specification documents

Hint: Don't confuse quality of requirements with quality requirements!

Quality of individual requirements



For individual requirements, strive for requirements that are...

● Adequate	True and agreed stakeholder needs
● Necessary	Part of the relevant system scope
● Unambiguous	True shared understanding
● Consistent	No contradictions
● Complete	No missing parts
● Understandable	Prerequisite for shared understanding
● Verifiable	Conformance of implementation can be checked
● Feasible	Non-feasible requirements are a waste of effort
● Traceable	Linked to other requirements-related items

Quality of requirements documents

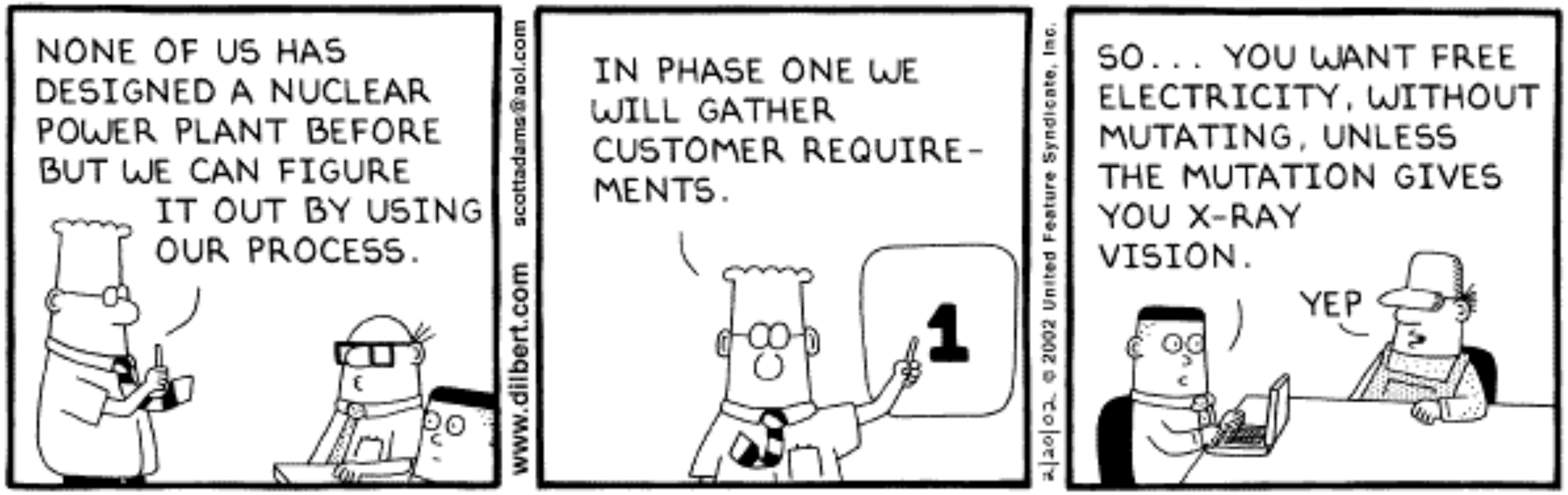
When creating a requirements specification, strive for a document that is

● Consistent	No contradictions
● Unambiguous	True shared understanding
● Structured	Improves readability of document
● Modifiable & extensible	Because change will happen
● Traceable	Linked to related artifacts
● Complete	Contains all relevant requirements
● Conformant	Conforms to prescribed document structure, format or style

Quality criteria are in the eye of the beholder

- No general consensus
- Different, overlapping sets of quality criteria used in
  - this course
  - RE textbooks
  - RE standards
  - Quasi-standards such as the IREB Certified Professional for Requirements Engineering (see http://www.ireb.org)

6 Requirements Engineering processes


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

The principal tasks

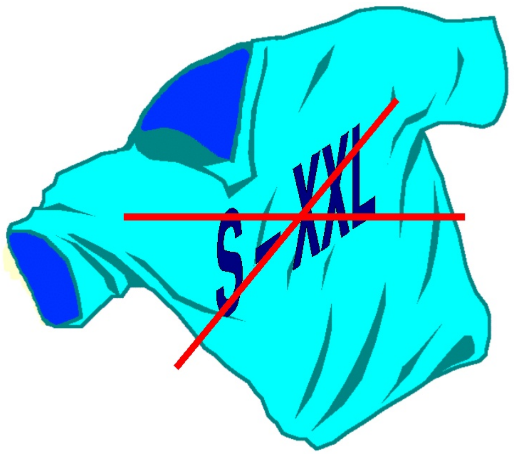
Requirements Specification

- Elicitation
- Analysis
- Documentation
- Validation

Requirements Management

- Identification and metadata
- Requirements prioritization
- Change and release management
- Traceability

No 'one size fits all' process

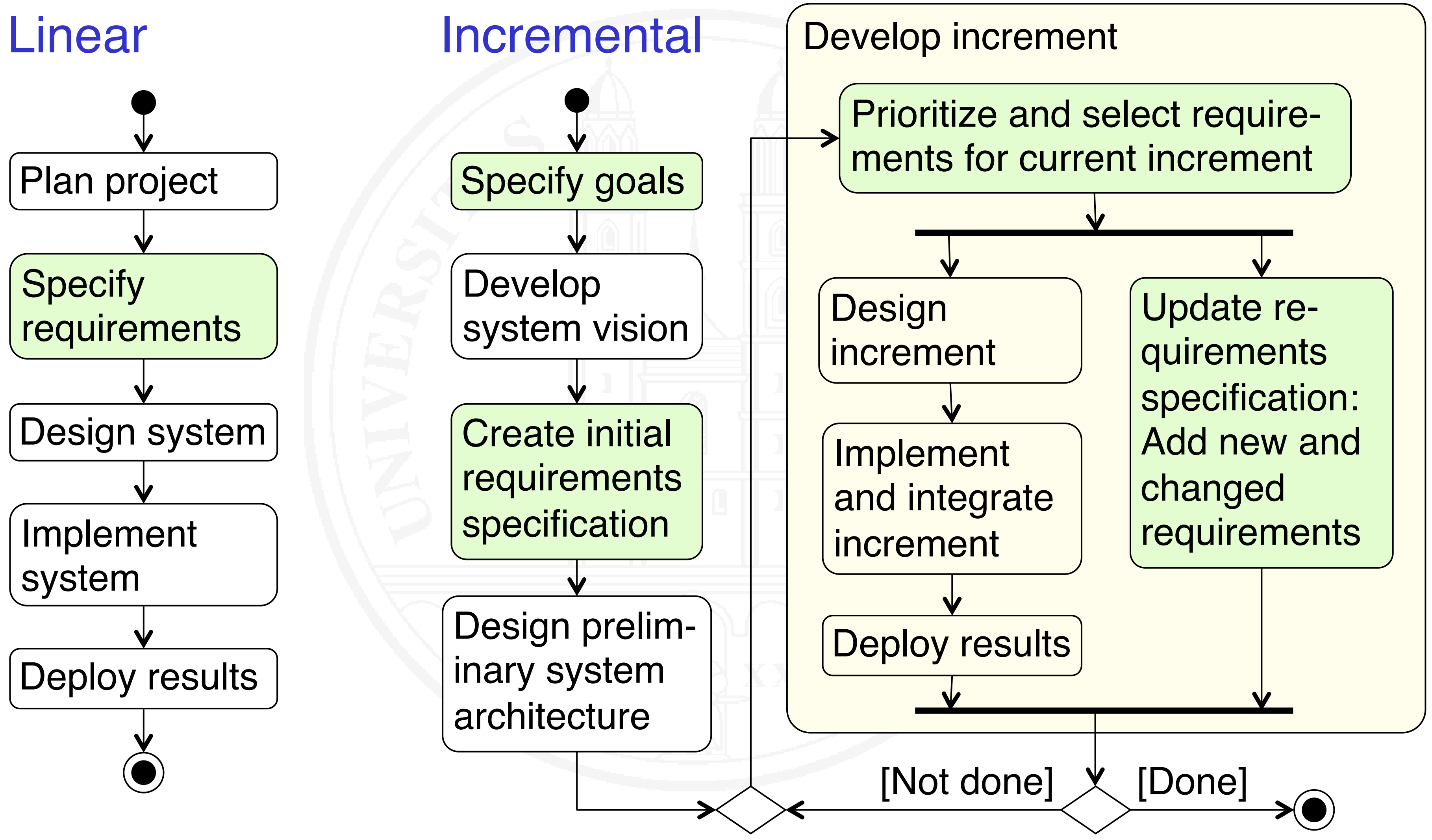


Some determining factors

- The embedding process:
  linear or incremental?
- Contract (prescriptive) or collaboration (explorative)?
- Can you talk with your stakeholders?
- Customer order or development for a market?
- Using COTS?

➪ Tailor the process from some principal configuration options and a rich set of RE practices

Linear vs. incremental processes


Linear
Plan project
Specify requirements
Design system
Implement system
Deploy results
Incremental
Specify goals
Develop system vision
Create initial requirements specification
Design prelim-inary system architecture
Develop increment
Prioritize and select require-ments for current increment
Design increment
Implement and integrate increment
Deploy results
Update re-quirements specification: Add new and changed requirements
[Not done]
[Done]

Linear vs. incremental processes – 2

Decision criteria

- Linear
  - Clear, stable, a priori known requirements
  - Low risk (of developing the wrong system)
  - Relatively short duration of project
  - Complex requirements change process is acceptable
- Incremental
  - Evolving requirements
  - High risk (of developing the wrong system)
  - Long duration of project
  - Ability to change requirements easily is important

Prescriptive – Explorative – COTS-driven

Prescriptive process

- Requirements specification is a contract: All requirements are binding and must be implemented
- Functionality determines cost and deadlines
- Needed when tendering design and implementation
- Development of specified system may be outsourced
- Frequently combined with linear processes

Explorative process

- Only goals known, concrete requirements have to be explored
- Stakeholders strongly involved, continuous feedback
- Prioritizing and negotiating requirements to be implemented
- Deadlines and cost constrain functionality
- Typically works only with incremental processes

Prescriptive – Explorative – COTS-driven – 2

COTS-driven process

- System will be implemented with COTS software
- Requirements must reflect functionality of chosen COTS solution
- Requirements need to be prioritized according to importance
- Frequently, only require-ments not covered by the COTS solution are specified

COTS (Commercial Off The Shelf) – A system or component that is not developed, but bought as a standard product from an external supplier

Customer-specific vs. Market-oriented

Customer-specific process

- System is ordered by a customer and developed by a supplier for this customer
- Individual persons can be identified for all stakeholder roles
- Stakeholders on customer side are main source for requirements

Market-oriented process

- System is developed as a product for the market
- Prospective users typically not individually identifiable
- Requirements are specified by supplier
- Marketing and system architects are primary stakeholders
- Supplier has to guess/estimate/elicit the needs of the envisaged customers

Typical requirements process configurations

- Participatory: incremental & exploratory & customer-specific
  - Main application case: Supplier and customer closely collaborate; customer stakeholders strongly involved both in specification and development processes

- Contractual: typically linear (sometimes explorative) & prescriptive & customer-specific
  - Main application case: Specification constitutes contractual basis for development of a system by people not involved in the specification and with little stakeholder interaction after the requirements phase

Typical requirements process configuration

- Product-oriented: Incremental & mostly explorative & market-oriented
  - Main application case: An organization specifies and develops software in order to sell/distribute it as a product

- COTS-aware: [Incremental I linear] & COTS-driven & customer-specific
  - Main application case: The requirements specification is part of a project where the solution is mainly implemented by buying and configuring COTS

Agile requirements process

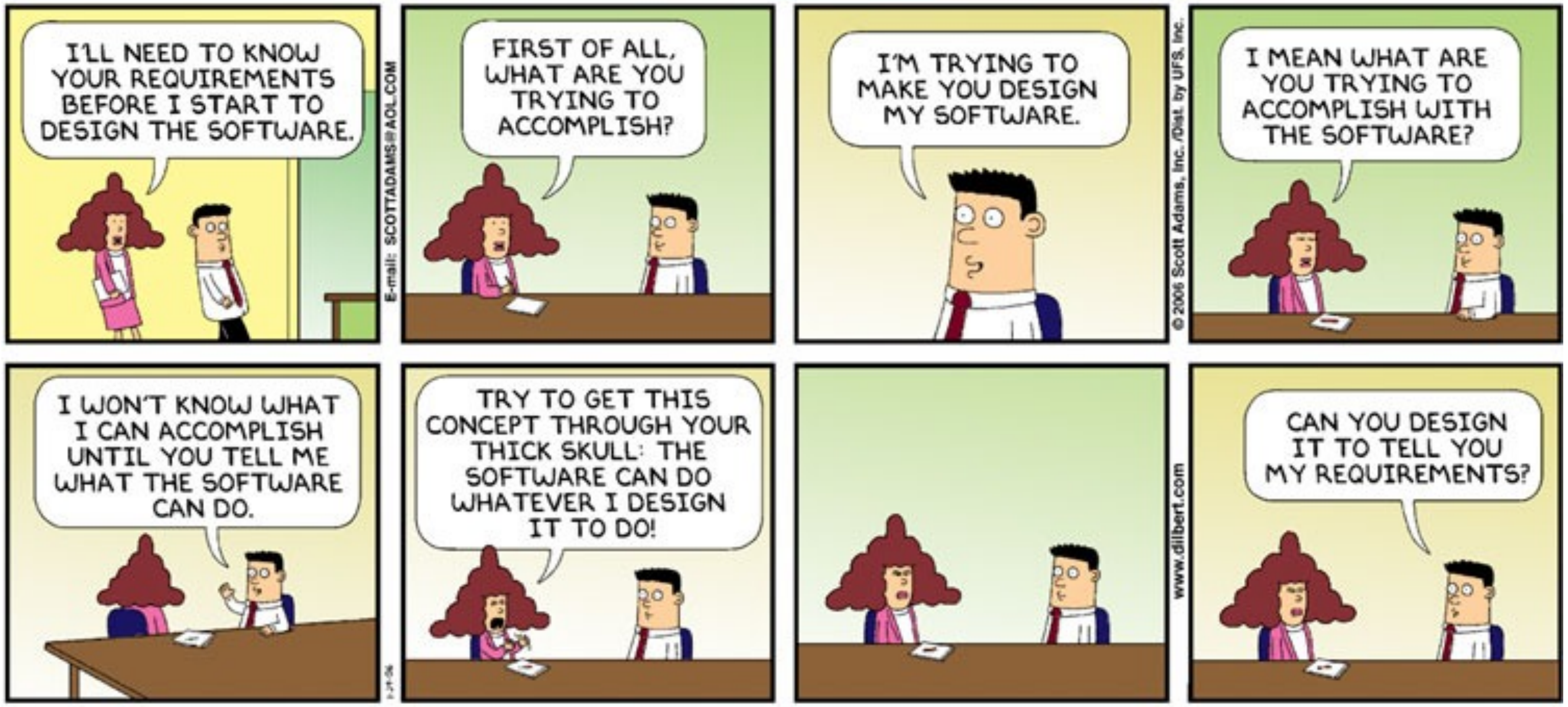
Pushes incrementality and exploration to the extreme

- Fixed-length increments of 1-6 weeks
- Product owner or customer representative always available and has power to make immediate decisions
- Only goals and vision established upfront
- Requirements loosely specified as stories
- Details captured in test cases
- At the beginning of each increment
  - Customer prioritizes requirements
  - Developers select requirements to be implemented in that increment
- Short feedback cycle from requirements to deployed system

Characteristics of an “ideal” RE process

- Strongly interactive
- Close and intensive collaboration between
  - Stakeholders (know the domain and the problem)
  - Requirements engineers (know how to specify)
- Very short feedback cycles
- Risk-aware and feasibility-aware
  - Technical risks/feasibility
  - Deadline risks/feasibility
- Careful negotiation / resolution of conflicting requirements
- Focus on establishing shared understanding
- Strives for innovation

7 Requirements elicitation


I'LL NEED TO KNOW YOUR REQUIREMENTS BEFORE I START TO DESIGN THE SOFTWARE.
E-mail: SCOTTADAMS@AOL.COM
FIRST OF ALL, WHAT ARE YOU TRYING TO ACCOMPLISH?
I'M TRYING TO MAKE YOU DESIGN MY SOFTWARE.
© 2006 Scott Adams, Inc. /Dist. by UFS, Inc.
I MEAN WHAT ARE YOU TRYING TO ACCOMPLISH WITH THE SOFTWARE?
I WON'T KNOW WHAT I CAN ACCOMPLISH UNTIL YOU TELL ME WHAT THE SOFTWARE CAN DO.
TRY TO GET THIS CONCEPT THROUGH YOUR THICK SKULL: THE SOFTWARE CAN DO WHATEVER I DESIGN IT TO DO!
www.dilbert.com
CAN YOU DESIGN IT TO TELL YOU MY REQUIREMENTS?

Definition and principles

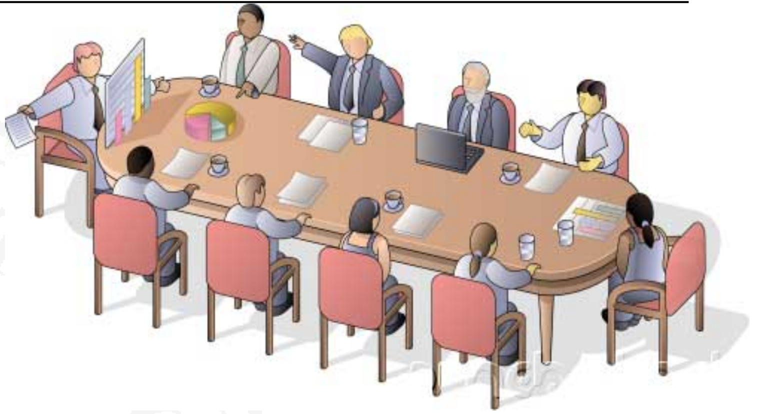
DEFINITION. Requirements elicitation – The process of seeking, capturing and consolidating requirements from available sources. May include the re-construction or creation of requirements.

- ❍ Determine the stakeholders' desires and needs
- ❍ Elicit information from all available sources and consolidate it into well-documented requirements
- ❍ Make stakeholders happy, not just satisfy them
- ❍ Every elicited and documented requirement must be validated and managed
- ❍ Work value-oriented and risk-driven

Information sources

- ❍ Stakeholders
- ❍ Context
- ❍ Observation
- ❍ Documents
- ❍ Existing systems

Stakeholder analysis



Identify stakeholder roles

End user, customer, operator, project manager, regulator,...

In complex cases: Build model of stake-holder goals, dependencies and rationale

[Yu 1997]
[van Lamsweerde 2001]

Classify stakeholders

- Critical
- Major
- Minor

[Glinz and Wieringa 2007]

Identify/determine concrete persons for each stakeholder role

Context analysis

Photo © Universitätsklinikum Halle (Saale)

Determine the system's context and the context boundary

Identify context constraints

- Physical, legal, cultural, environmental
- Embedding, interfaces

Identify assumptions about the context of your system and make them explicit

Map real world phenomena adequately...

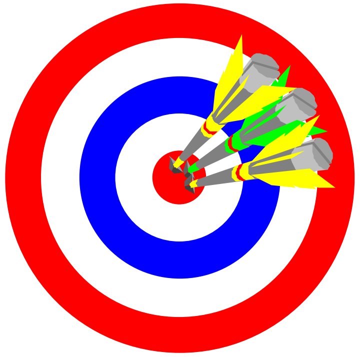
- ... on the required system properties and capabilities
- ... and vice-versa

Goal analysis

Knowing your destination is more important than the details of the timetable.

Before eliciting detailed requirements, the general goals and vision for the system to be built must be clear

- What are the main goals?
- How do they relate to each other?
- Are there goal conflicts?



Mini-Exercise

Consider the chairlift access control case study.

(a) Perform a stakeholder analysis.

(b) How can you map the context property that a skier passes an unlocked turnstile to a system property which can be sensed and controlled by the system?

(c) Identify some business goals.

Elicitation techniques

Ask

- ❍ Interview stakeholders
- ❍ Use questionnaires and polls

Collaborate

- ❍ Hold requirements workshops

Build and play

- ❍ Build, explore and discuss prototypes and mock-ups
- ❍ Perform role playing

[Zowghi and Coulin 2005]
[Dieste, Juristo, Shull 2008]
[Gottesdiener 2002]
[Hickey and Davis 2003]
[Goguen and Linde 1993]

Elicitation techniques – 2

Observe

- Observe stakeholders in their work context

Analyze

- Analyze work artifacts
- Analyze problem/bug reports
- Conduct market studies
- Perform benchmarking

Which technique for what?

Technique	Suitability for			
	Express needs	Demonstrate opportunities	Analyze system as is	Explore market potential
Interviews	+	–	+	o
Questionnaires and polls	o	–	+	+
Workshops	+	o	o	–
Prototypes and mock-ups	o	+	–	o
Role play	+	o	o	–
Stakeholder observation	o	–	+	o
Artifact analysis	o	–	+	–
Problem/bug report analysis	+	–	–	o
Market studies	–	–	o	+
Benchmarking	o	+	–	+

Typical problems

Inconsistencies among stakeholders in

- needs and expectations
- terminology

Stakeholders who know their needs, but can't express them

Stakeholders who don't know their needs

Stakeholders with a hidden agenda

Stakeholders thinking in solutions instead of problems

Stakeholders frequently neglect attributes and constraints

➥ Elicit them explicitly

Who should elicit requirements?

- Stakeholders must be involved
- Domain knowledge is essential
  - Stakeholders need to have it (of course)
  - Requirements engineers need to know the main domain concepts
  - A “smart ignoramus” can be helpful [Berry 2002, Sect. 7]
- Don’t let stakeholders specify themselves without professional support
- Best results are achieved when stakeholders and requirements engineers collaborate

Eliciting functional requirements

- Who wants to achieve what with the system?
- For every identified function
  - What's the desired result and who needs it?
  - Which transformations and which inputs are needed?
  - In which state(s) shall this function be available?
  - Is this function dependent on other functions?
- For every identified behavior
  - In which state(s) shall the system have this behavior?
  - Which event(s) lead(s) to this behavior?
  - Which event(s) terminate(s) this behavior?
  - Which functions are involved?

Eliciting functional requirements – 2

- For every identified data item
  - What are the required structure and the properties of this item?
  - Is it static data or a data flow?
  - If it's static, must the system keep it persistently?
- Analyze mappings
  - How do real world functions/behavior/data map to system functions/behavior/data and vice-versa?
- Specify normal and exceptional cases

Eliciting quality requirements

Stakeholders frequently state quality requirements in qualitative form:

"The system shall be fast."

"We need a secure system."

Problem: Such requirements are

- Ambiguous
- Difficult to achieve and verify

❍ Classic approach:

- Quantification ➔ ⊕ measurable ⊖ maybe too expensive
- Operationalization ➔ ⊕ testable ⊖ implies premature design decisions

New approach to eliciting quality requirements

[Glinz 2008]

Represent quality requirements such that they deliver optimum value

Value of a requirement = benefit of development risk reduction minus cost for its specification

- Assess the criticality of a quality requirement
- Represent it accordingly
- Broad range of possible representations

The range of adequate representations

Situation	Representation	Verification
1. Implicit shared understanding	Omission	Implicit
2. Need to state general direction Customer trusts supplier	Qualitative	Inspection
3. Sufficient shared understanding to generalize from examples	By example	Inspection, (Measurement)
4. High risk of not meeting stake-holders' desires and needs	Quantitative in full	Measurement
5. Somewhere between 2 and 4	Qualitative with partial quantification	Inspection, partial measurement

Eliciting performance requirements

Things to elicit

- Time for performing a task or producing a reaction
- Volume of data
- Throughput (data transmission rates, transaction rates)
- Frequency of usage of a function
- Resource consumption (CPU, storage, bandwidth, battery)
- Accuracy (of computation)

Eliciting performance requirements – 2

- What's the meaning of a performance value:
  - Minimum?
  - Maximum?
  - On average?
  - Within a given interval?
  - According to some probability distribution?
- How much deviation can be tolerated?

Eliciting specific quality requirements

- Ask stakeholders explicitly
- A quality model such as ISO/IEC 25010:2011(formerly ISO/IEC 9126) can be used as a checklist
- Quality models also help when a specific quality requirement needs to be quantified

Eliciting constraints

- Ask about restrictions of the potential solution space
  - Technical, e.g., given interfaces to neighboring systems
  - Legal, e.g., restrictions imposed by law, standards or regulations
  - Organizational, e.g. organizational structures or processes that must not be changed by the system
  - Cultural, environmental, ...
- Check if a requirement is concealed behind a constraint
  - Constraint stated by a stakeholder: “When in exploration mode, the print button must be grey.”
  - Actual requirement: “When the system is used without a valid license, the system shall disable printing.”

Mini-Exercise

Consider the chairlift access control case study.

(a) Which technique(s) would you select to elicit requirements from the chairlift ticket office clerks?

(b) How, for example, can you achieve consensus among the ski resort management, the technical director of chairlifts, the ticket office clerks, and the service employees?

(c) Identify some constraints for the chairlift access control system.

Analysis of elicited information

Structure-oriented

Process-oriented

Analyze terminology / domain properties
Build glossary

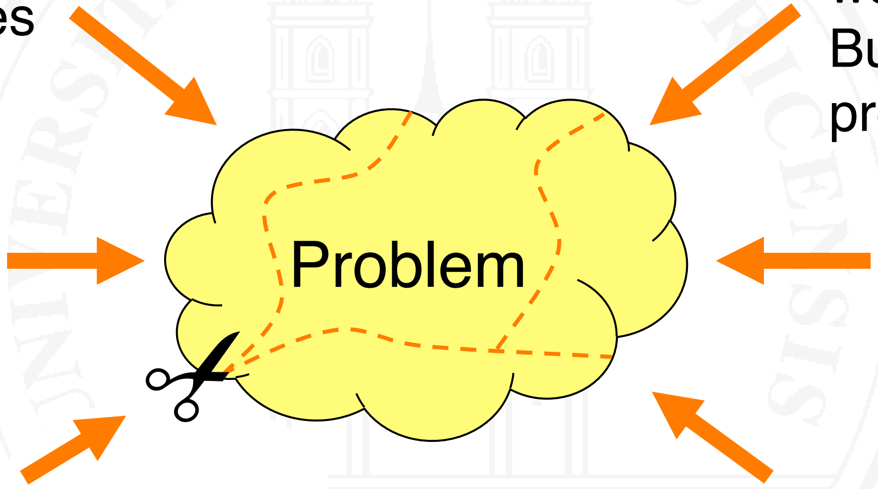
Analyze business and data objects
Build object and class models

Decompose problem
Build hierarchical structure

Analyze processes / workflows
Build activity / process models

Analyze dynamic system behavior
Build behavior model

Analyze actor-system interaction
Build scenarios / use cases



Note: requirements are about a future state of affairs; analyze the current state only when necessary

Documenting elicited requirements

Build specification incrementally and continuously

Document requirements in small units

End over means: Result → Function → Input

Consider the unexpected: specify non-normal cases

Quantify critical attributes

Document critical assumptions explicitly

Avoid redundancy

Build a glossary and stick to terminology defined in the glossary

8 Specifying with natural language

The system shall ...

The oldest...

...and most widely used way

- taught at school
- extremely expressive

But not the best

- Ambiguous
- Imprecise
- Error-prone
- Verification only by careful reading

Michelangelo's Moses (San Pietro in Vincoli, Rome)
Moses holds the Ten Commandments in his hand: written in natural language

Problems with natural language requirements

Read the subsequent requirements. Any findings?

“For every turnstile, the total number of turns shall be read and archived once per day.”

“The system shall produce lift usage statistics.”

“Never shall an unauthorized skier pass a turnstile.”

“By using RFID technology, ticket validation shall become faster.”

“In the sales transaction, the system shall record the buyer’s data and timestamp the sold access card.”

Some rules for specifying in natural language

[Rupp et al. 2009]

- Use active voice and defined subjects
- Build phrases with complete verbal structure
- Use terms as defined in the glossary
- Define precise meanings for auxiliary verbs (shall, should, must, may,...) as well as for process verbs (for example, "produce", "generate", "create")
- Check for nouns with unspecific semantics ("the data", "the customer", "the display",...) and replace where appropriate
- When using adjectives in comparative form, specify a reference point: "better" ➜ "better than"

More rules

- ❍ Scrutinize all-quantifications: “every”, “always”, “never”, etc. seldom hold without any exceptions
- ❍ Scrutinize nominalizations (“authentication”, “termination”...): they may conceal incomplete process specifications
- ❍ State every requirement in a main clause. Use subordinate clauses only for making the requirement more precise
- ❍ Attach a unique identifier to every requirement
- ❍ Structure natural language requirements by ordering them in sections and sub-sections
- ❍ Avoid redundancy where possible: “never ever” ➜ “never”

Phrase templates

[Rupp et al. 2009
ISO/IEC/IEEE 29148:2011]

Use templates for creating well-formed natural language requirements

Typical template:

[<Condition>] <Subject> <Action> <Objects> [<Restriction>]

Example:

When a valid card is sensed, the system shall send the command ‘unlock_for_a_single_turn’ to the turnstile within 100 ms.

Agile stories

[Cohn 2004]

- A single sentence about a requirement
- Written from a stakeholder’s perspective
- Optionally including the expected benefit
- Accompanied by acceptance criteria for requirement
- Acceptance criteria make the story more precise

Standard template:

As a <role> I want to <my requirement> [so that <benefit>]

A sample story

As a skier, I want to pass the chairlift gate so that I get access without presenting, scanning or inserting a ticket at the gate.

Author: Dan Downhill Date: 2013-09-20 ID: S-18

Sample acceptance criteria

Acceptance criteria:

- Recognizes cards worn anywhere in a pocket on the left side of the body in the range of 50 cm to 150 cm above ground
- If card is valid: unlocks turnstile and flashes a green light for five seconds or until the turnstile is moved
- If card is invalid: doesn’t unlock gate and flashes a red light for five seconds
- Time from card entering the sensor range until unlock and flash red or green is less than 1.5 s (avg) & 3 s (max)
- The same card is not accepted twice within an interval of 200 s

Mini-Exercise: Writing a user story

Consider the chairlift access control case study.

Write a story from a skier’s perspective about buying a day card.

All-quantification and exclusion

- ❍ Specifications in natural language frequently use all-quantifying or excluding statements without much reflection:

“When operating the coffee vending machine, the user shall always be able to terminate the running transaction by pressing the cancel key.”

Also when the coffee is already being brewed or dispensed?

➪ Scrutinize all-quantifications (“every”, “all”, “always”...) and exclusions (“never”, “nobody”, “either – or”,...) for potential exceptions

➪ Specify found exceptions as requirements

Dealing with redundancy

- Natural language is frequently (and deliberately) redundant

  → Secures communication success in case of some information loss

- In requirements specifications, redundancy is a problem
  - Requirements are specified more than once
  - In case of modifications, all redundant information must be changed consistently
- Make redundant statements only when needed for abstraction purposes
- Avoid local redundancy: “never ever” → “never”

9 Model-based requirements specification

A guided tour through ...

- ❍ Data and object modeling
- ❍ Behavior modeling
- ❍ Function and process modeling
- ❍ User interaction modeling
- ❍ Goal modeling
- ❍ UML

Primarily for functional requirements

Quality requirements and constraints are mostly specified in natural language

9.1 Characteristics and options

- ❍ Requirements are described as a problem-oriented model of the system to be built
- ❍ Architecture and design information is omitted
- ❍ Mostly graphically represented
- ❍ Semi-formal or formal representation

What can be modeled?

System view: modeling a system's static structure, behavior and functions

Static structure perspective
- (Entity-Relationship) data models
- Class and object models
- Sometimes component models

Behavior perspective
- Finite state machines
- Statecharts / state machines
- Petri nets

Function and flow perspective
- Activity models
- Data flow / information flow models
- Process and work flow models

What can be modeled? – continued

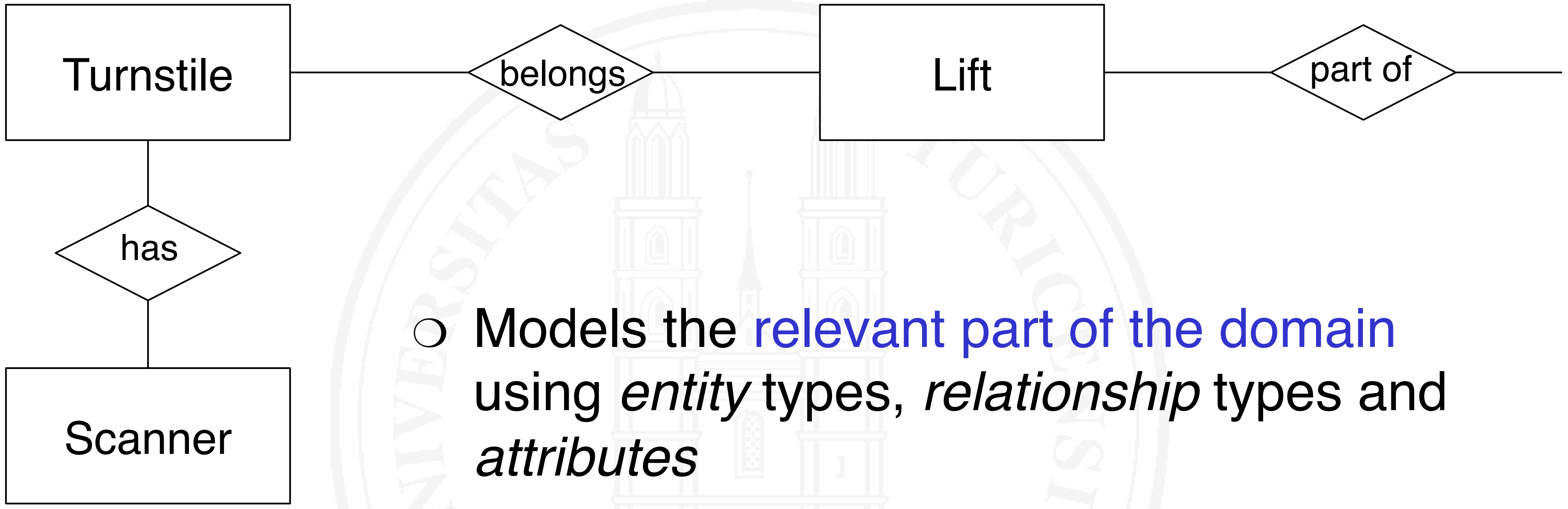
- User-system interaction view: modeling the interaction between a system and its external actors
  - Use cases, scenarios
  - Sequence diagrams
  - Context models
- Goal view: modeling goals and their dependencies
  - Goal trees
  - Goal-agent networks, e.g., i*

9.2 Models of static system structure

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

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

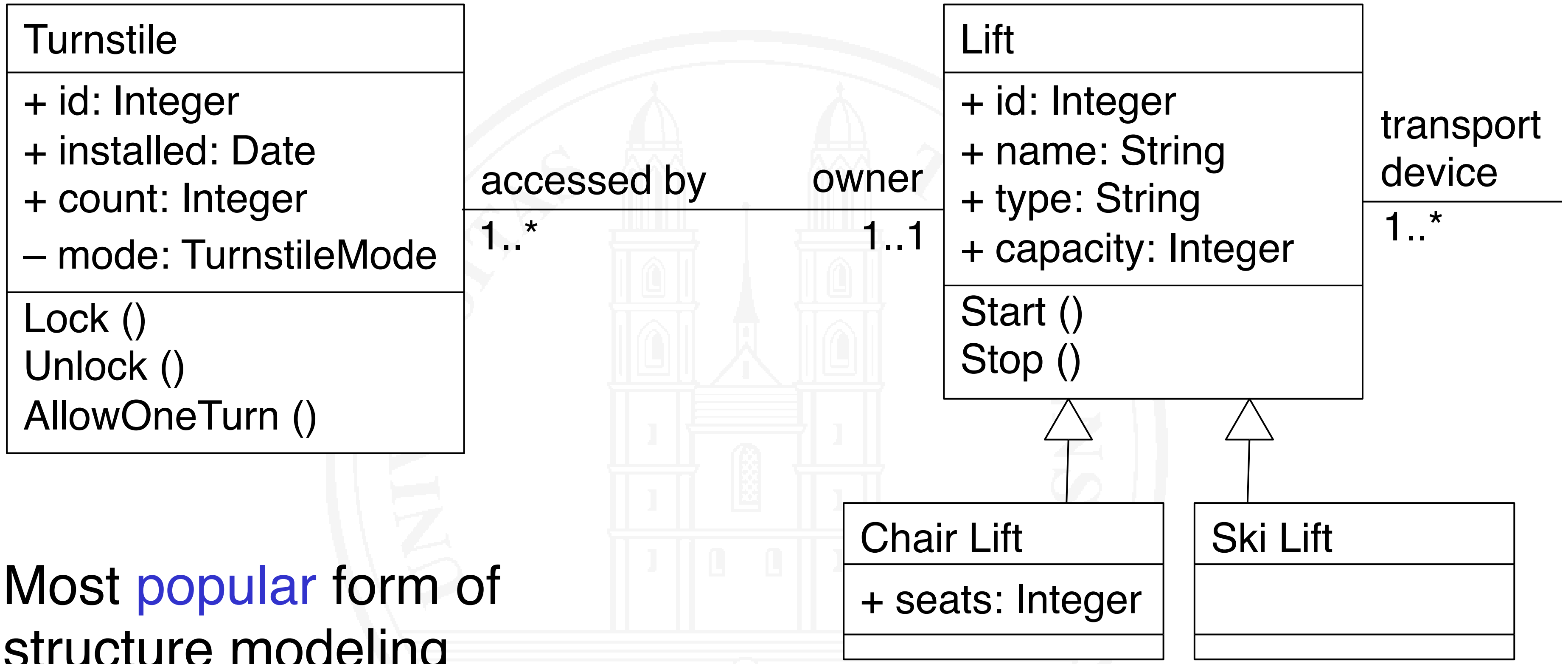
Class – Represents a set of objects of the same kind by describing the structure of the objects, the ways they can be manipulated and how they behave.

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.

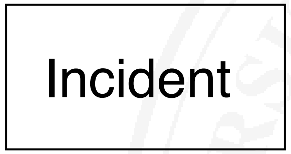
When specifying such a system, we need to model

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

Class models are inadequate here

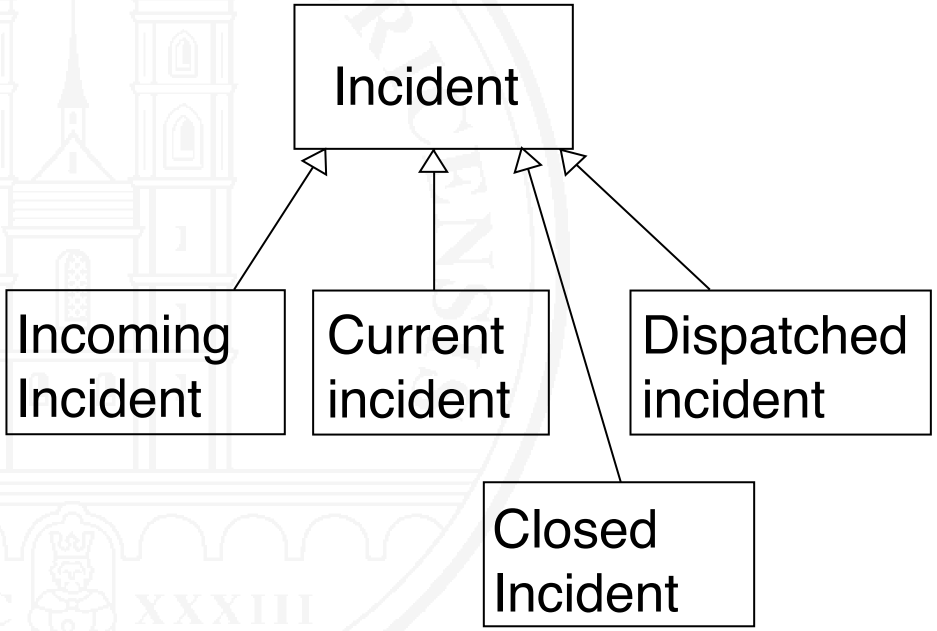
In a class model, incidents would have to be modeled as follows:

either



Bad: essential elements of the problem are not modeled

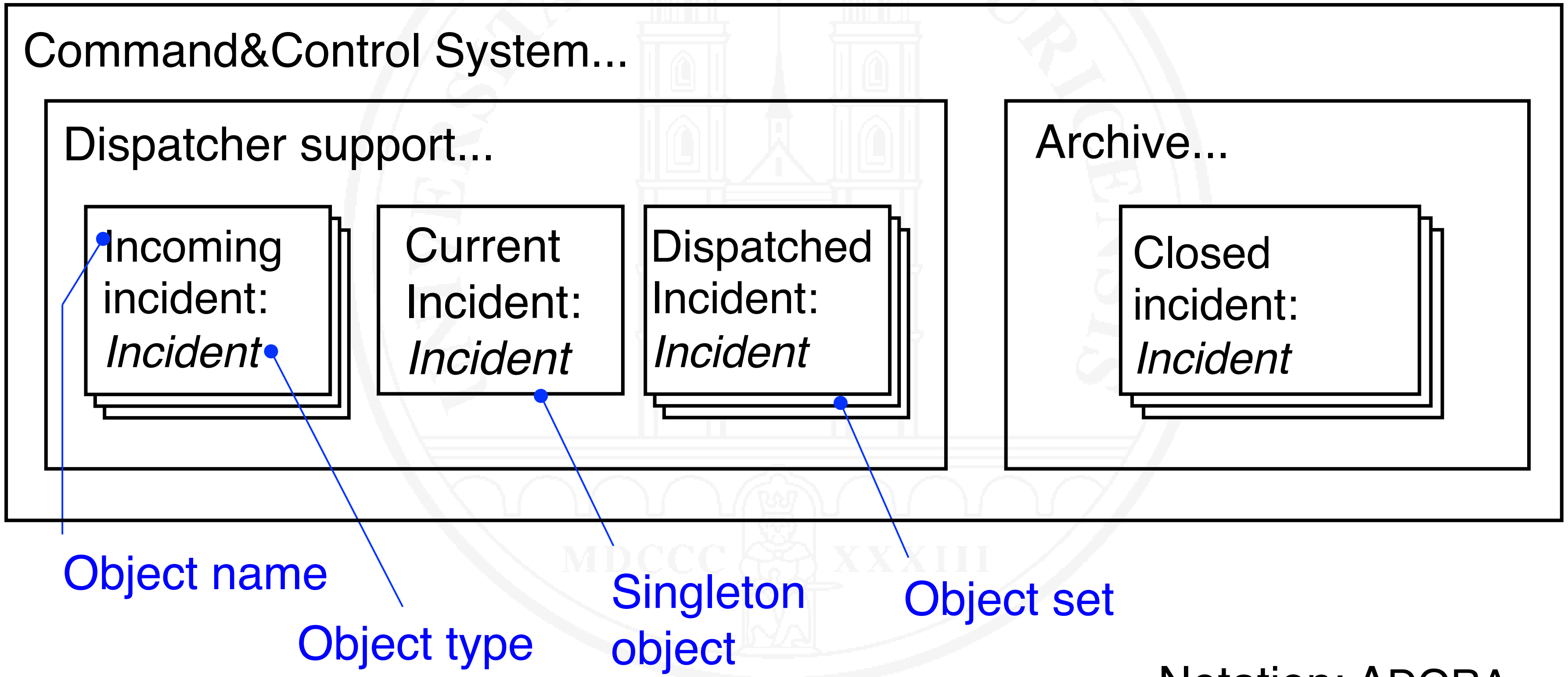
or



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 Behavior modeling

Goal: describe dynamic system behavior

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

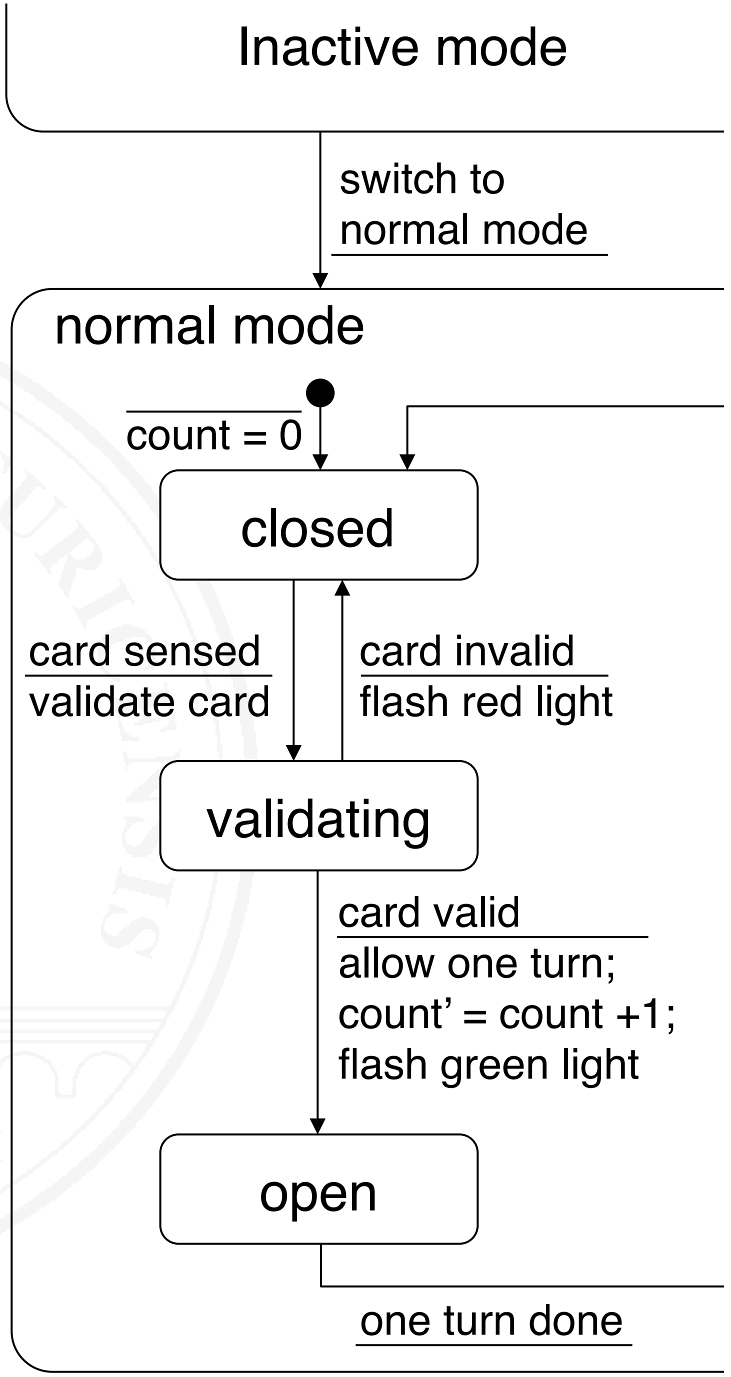
Means:

- ❍ 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]

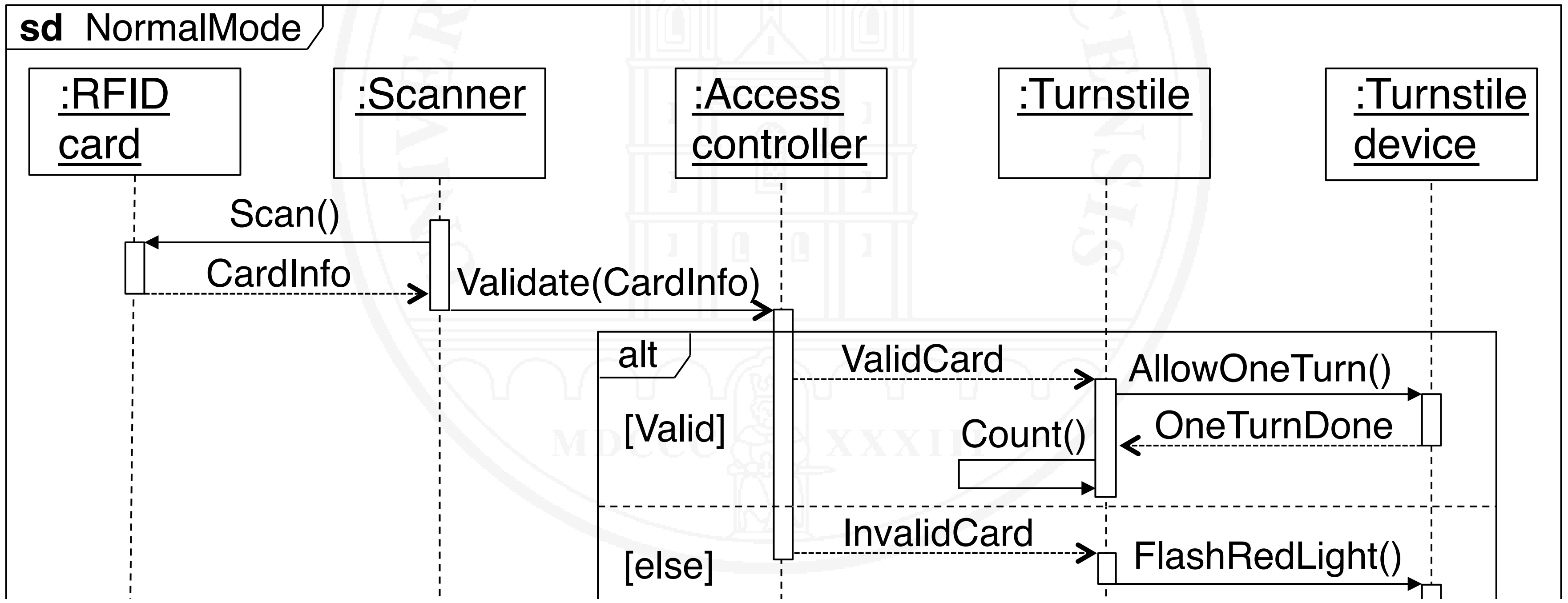
- ❍ Models the *dynamic* behavior:
  - ● How the system reacts to external events in a given state
  - ● Reaction depends on actual state
  - ● States may be hierarchically nested and/or orthogonal (parallel)
- ❍ In UML: state machine diagrams
- + Global view of system behavior
- + Precise, but still readable
- – Weak for modeling functionality and data



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

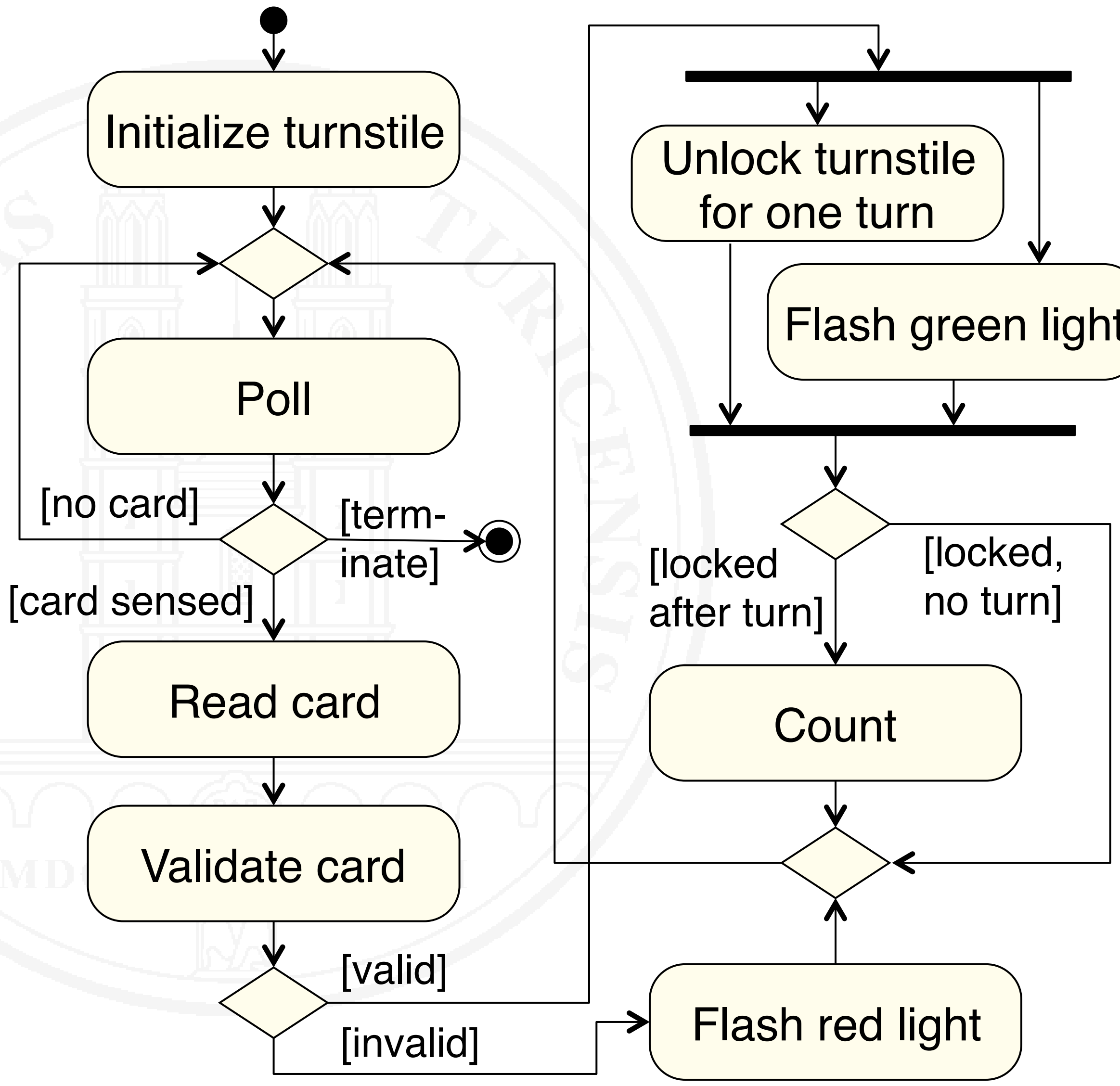
– Design-oriented, can detract from modeling requirements

9.4 Function and flow modeling

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

Activity modeling: UML activity diagram

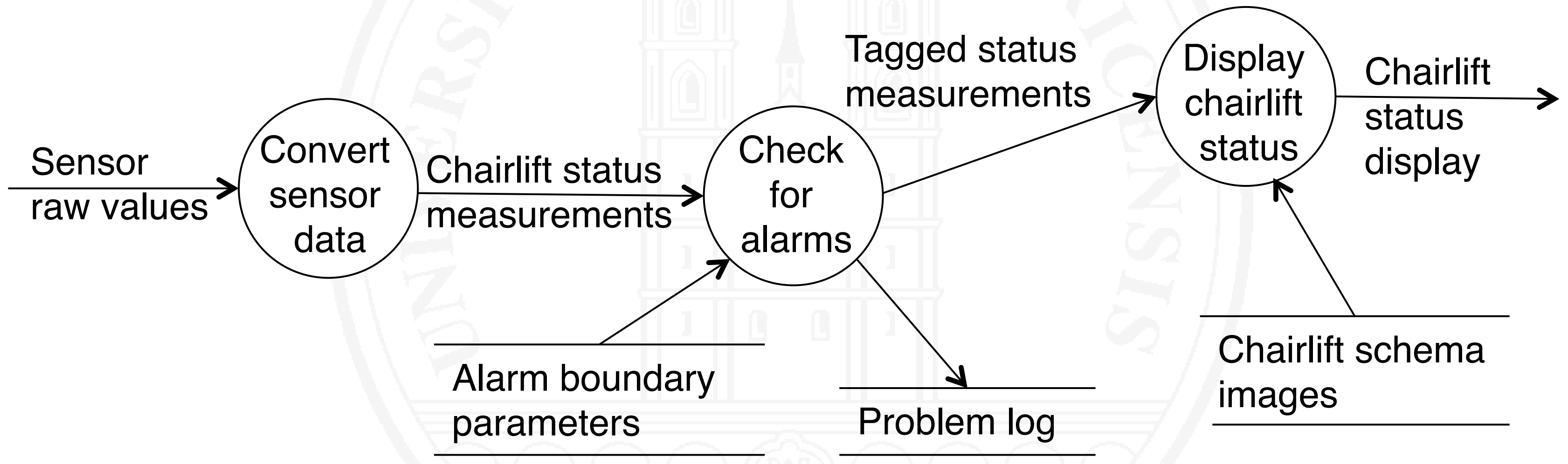
- ❍ Models process activities and control flow
- ❍ Can model data flow
- ❍ Model can be underpinned with execution semantics



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

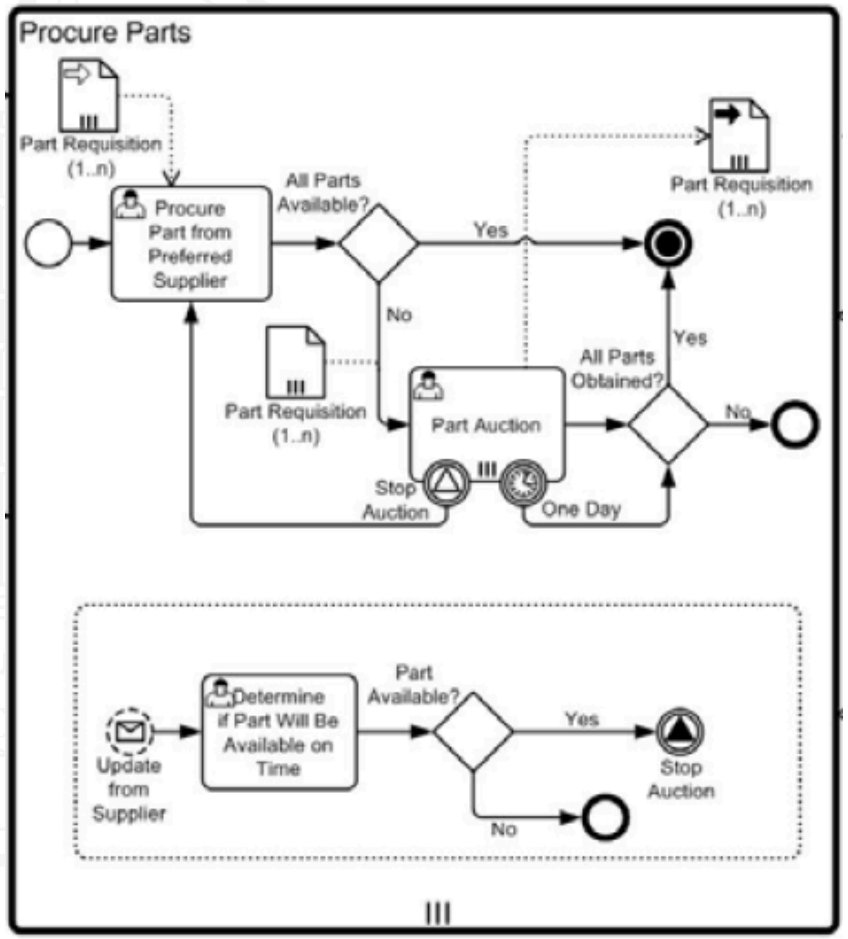
– Treatment of data outdated: no types, no encapsulation

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

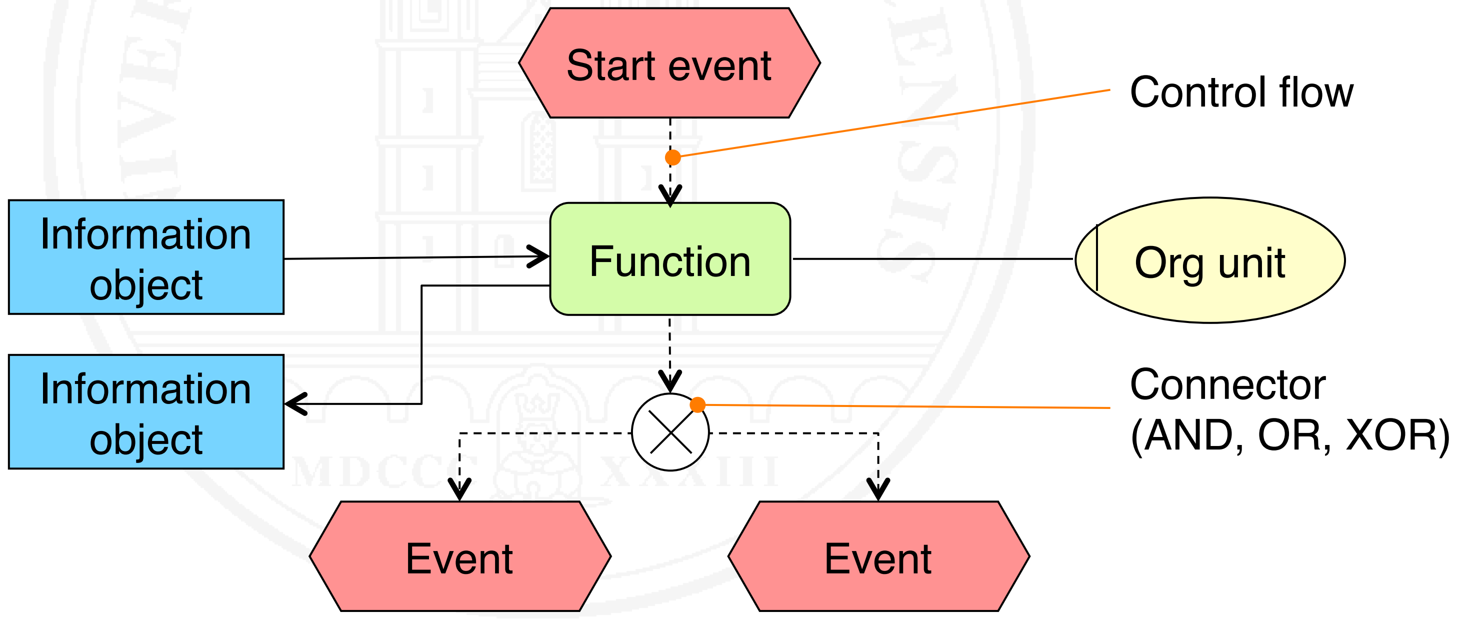
- BPMN (Business Process Model and Notation)
- Rich language for describing business processes



[Object Management Group 2011]

Process modeling: EPC

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



9.5 User-system interaction modeling

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 description of the interactions possible between actors and a system that, when executed, provide added value.

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. A description of a potential sequence of events that lead to a desired (or unwanted) result. 2. 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). 3. In UML: An execution trace of a use case.

[Carroll 1995
Sutcliffe 1998
Glinz 1995]

Use case / scenario descriptions

Various representation options

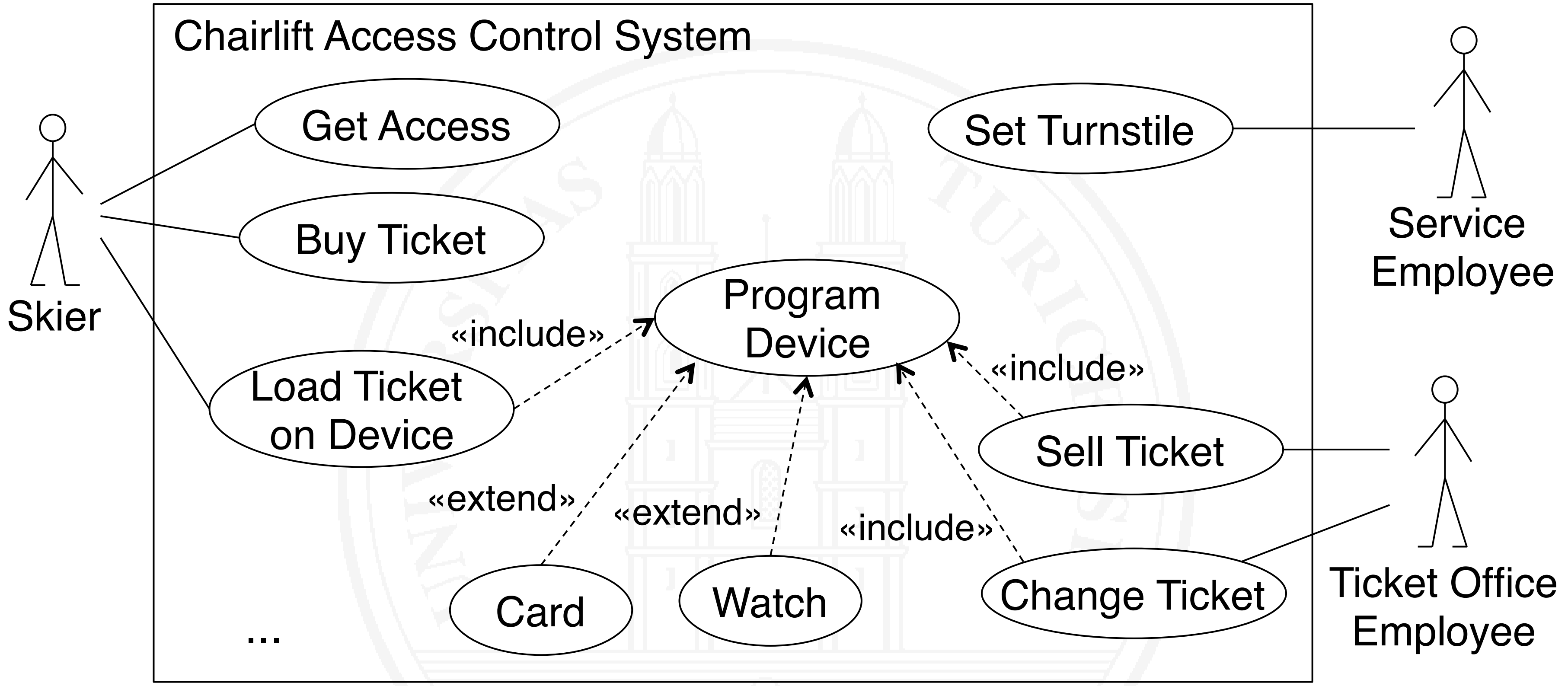
- Free text in natural language
- Structured text in natural language
- Statecharts / UML state machines
- UML activity diagrams
- Sequence diagrams / MSCs

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

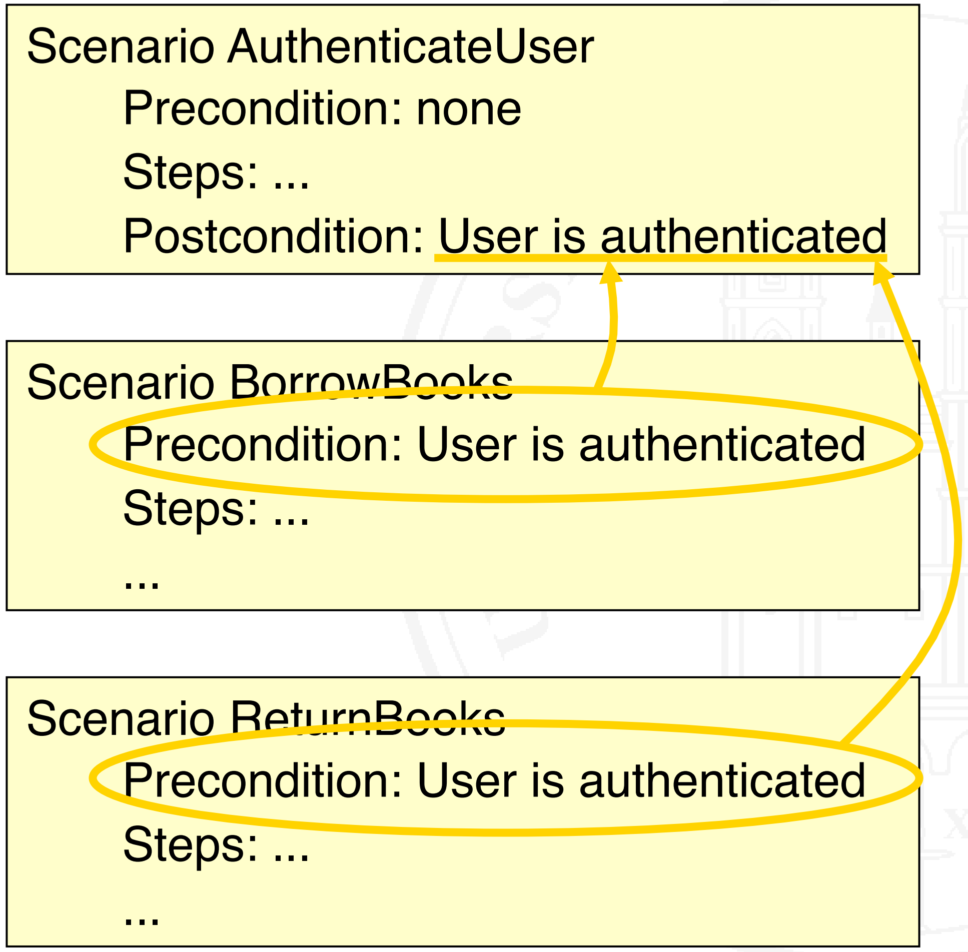


+ Provides abstract overview from actors’ perspectives
– Ignores functions and data required to provide interaction
– Can’t properly model hierarchies and dependencies

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: ...
...

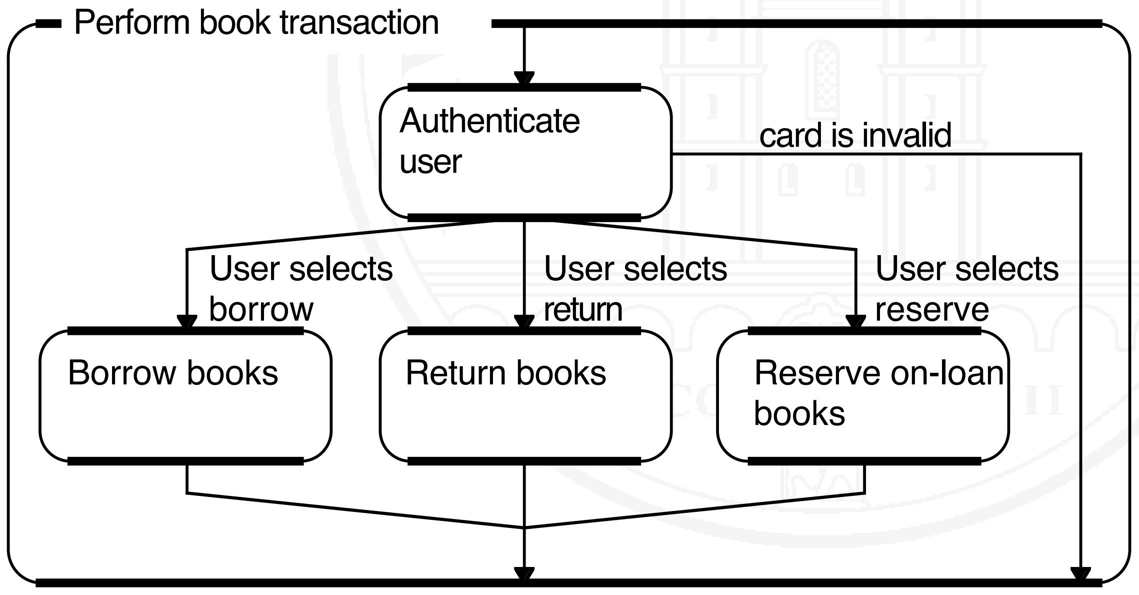
Scenario ReturnBooks
Precondition: User is authenticated
Steps: ...
...

- 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



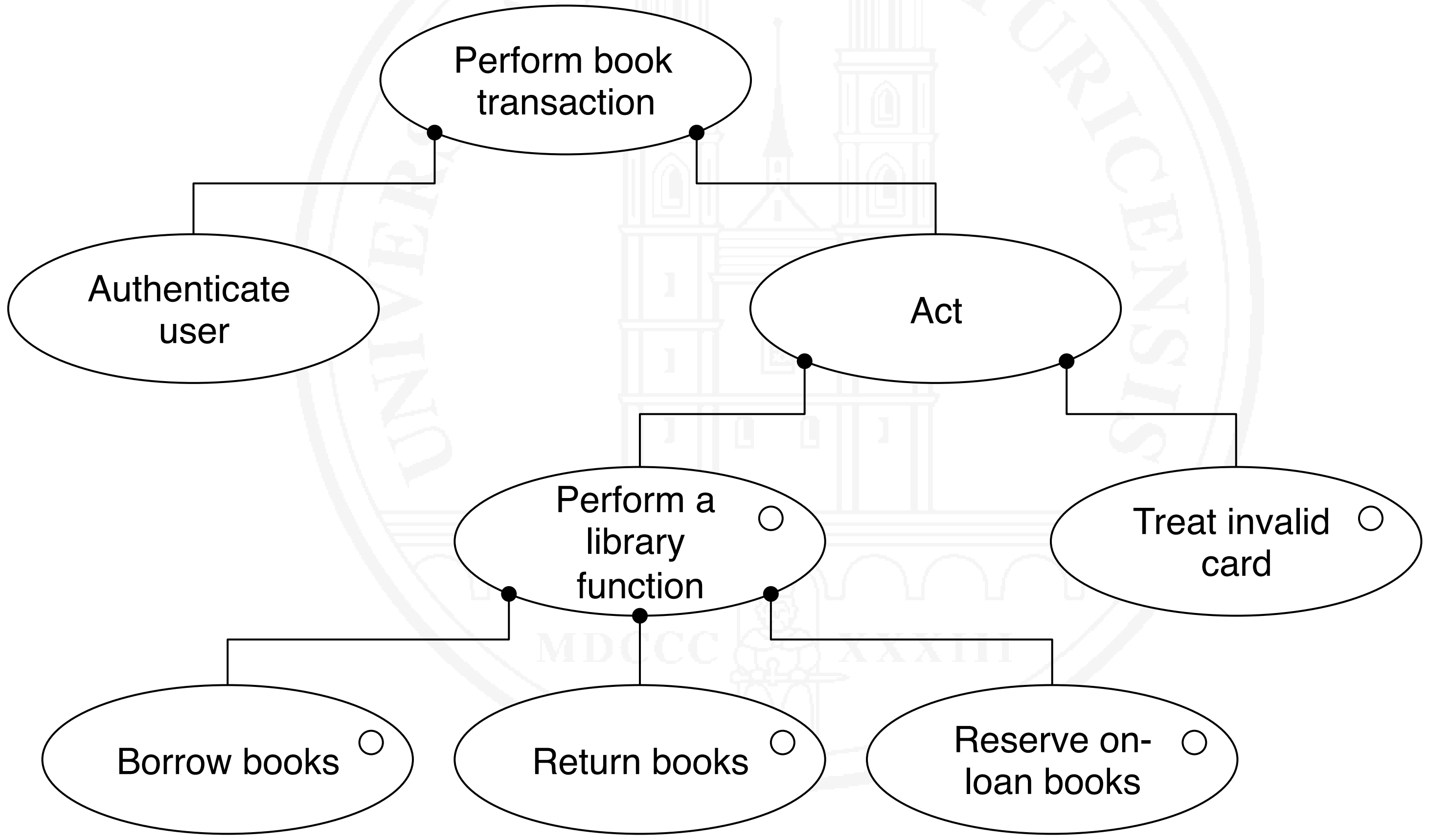
Research result, not used in today's practice

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

Dependencies with extended Jackson-diagrams

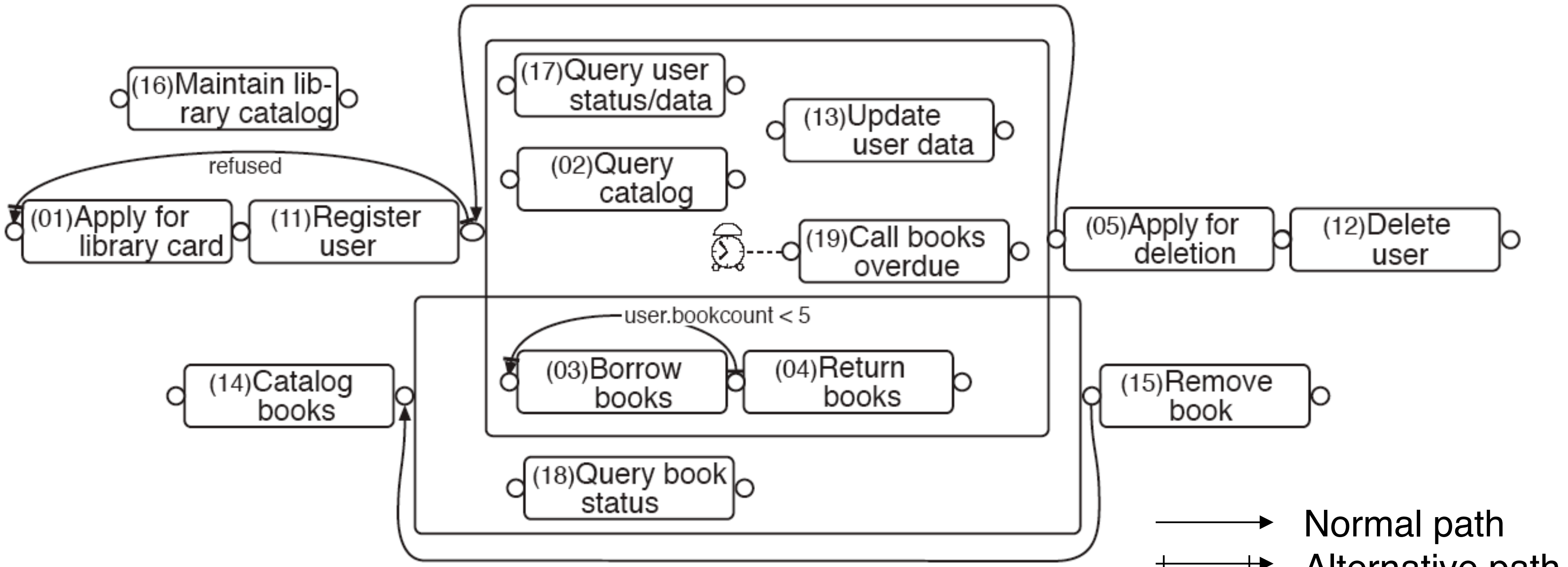
[Glinz et al. 2002]

- Used in ADORA for modeling scenario dependencies



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

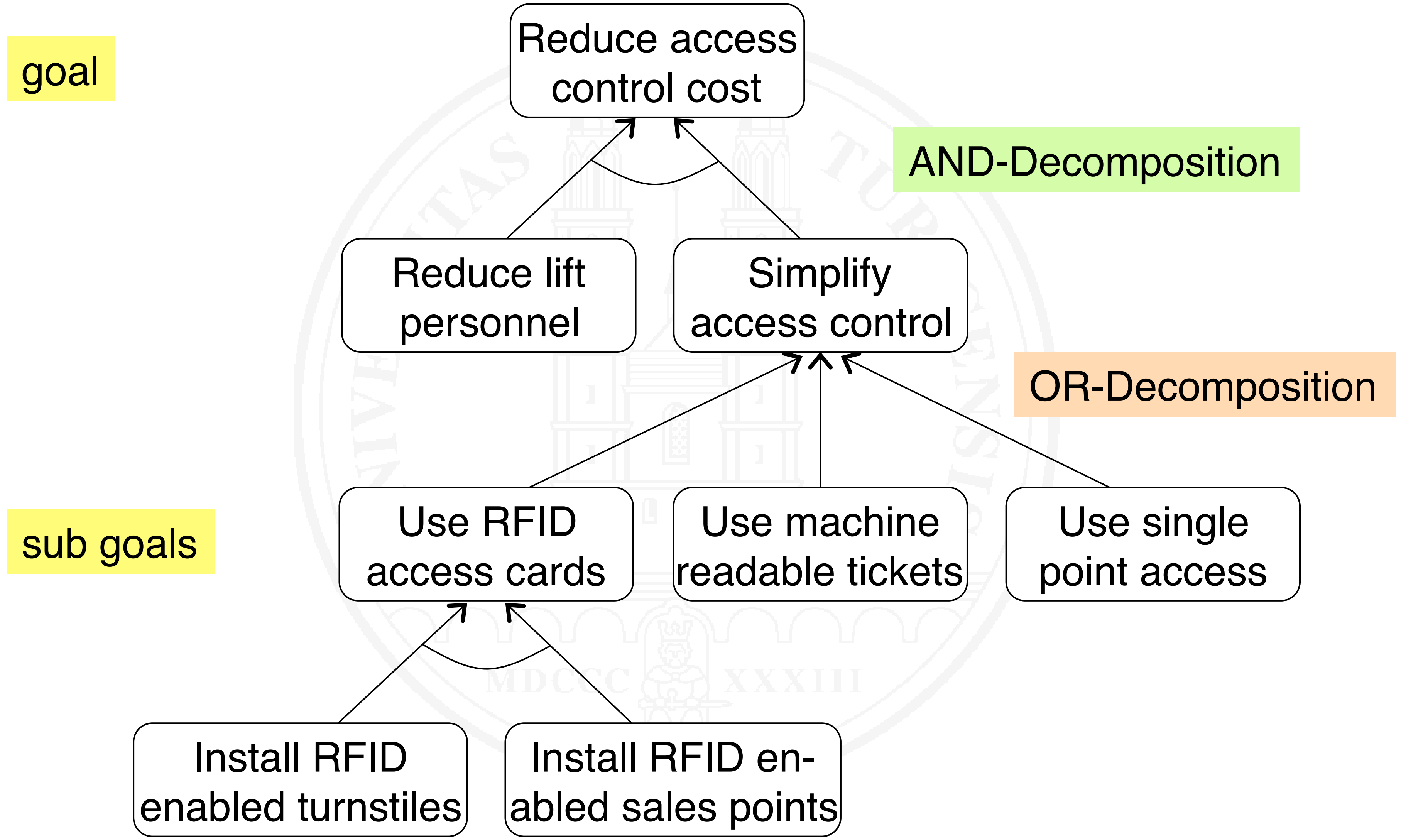
For the Chairlift access control system, write the use case "Get Access", describing how a skier gets access to a chairlift using his or her RFID ticket.

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

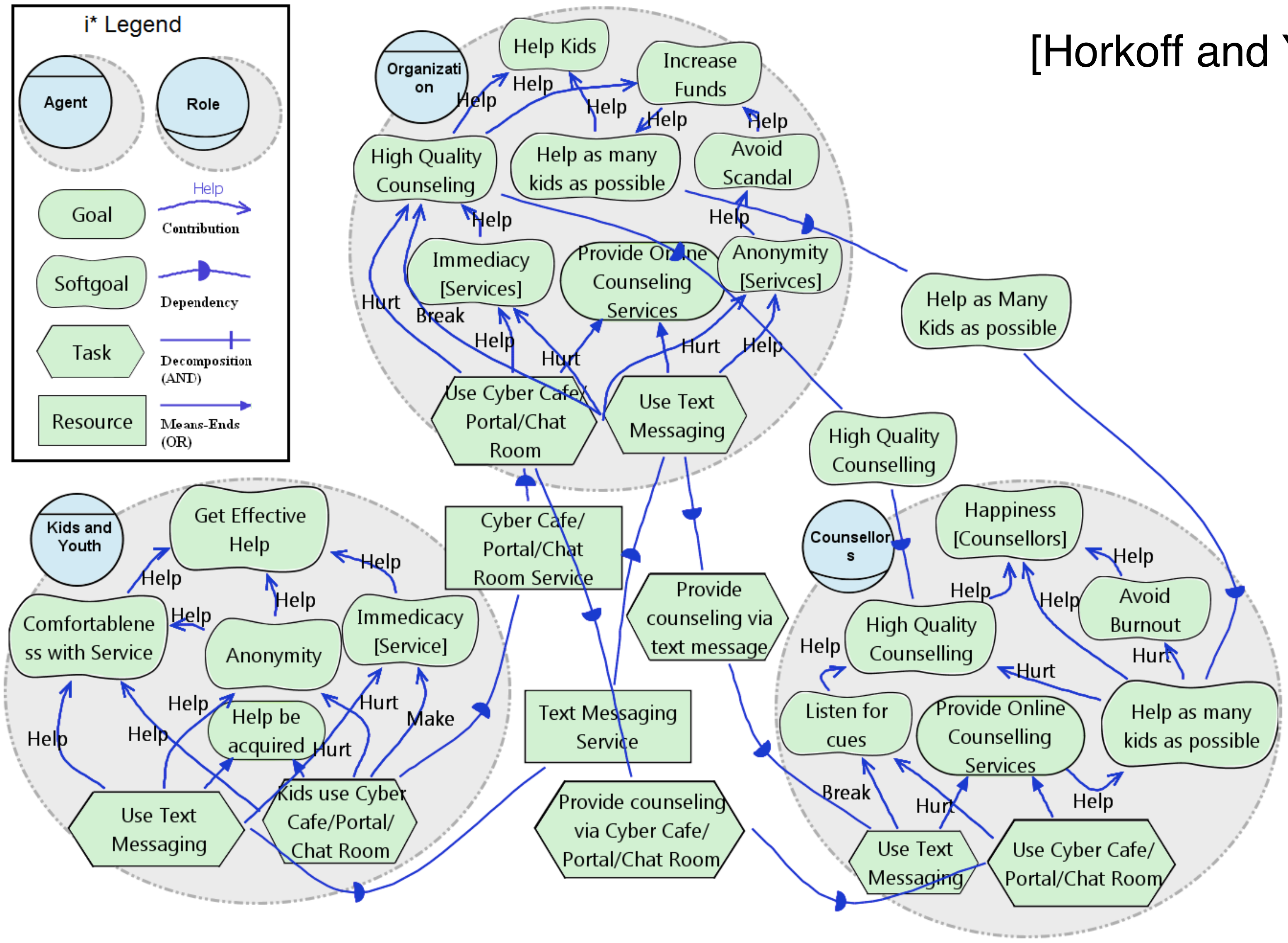


Goal-agent networks

- 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]



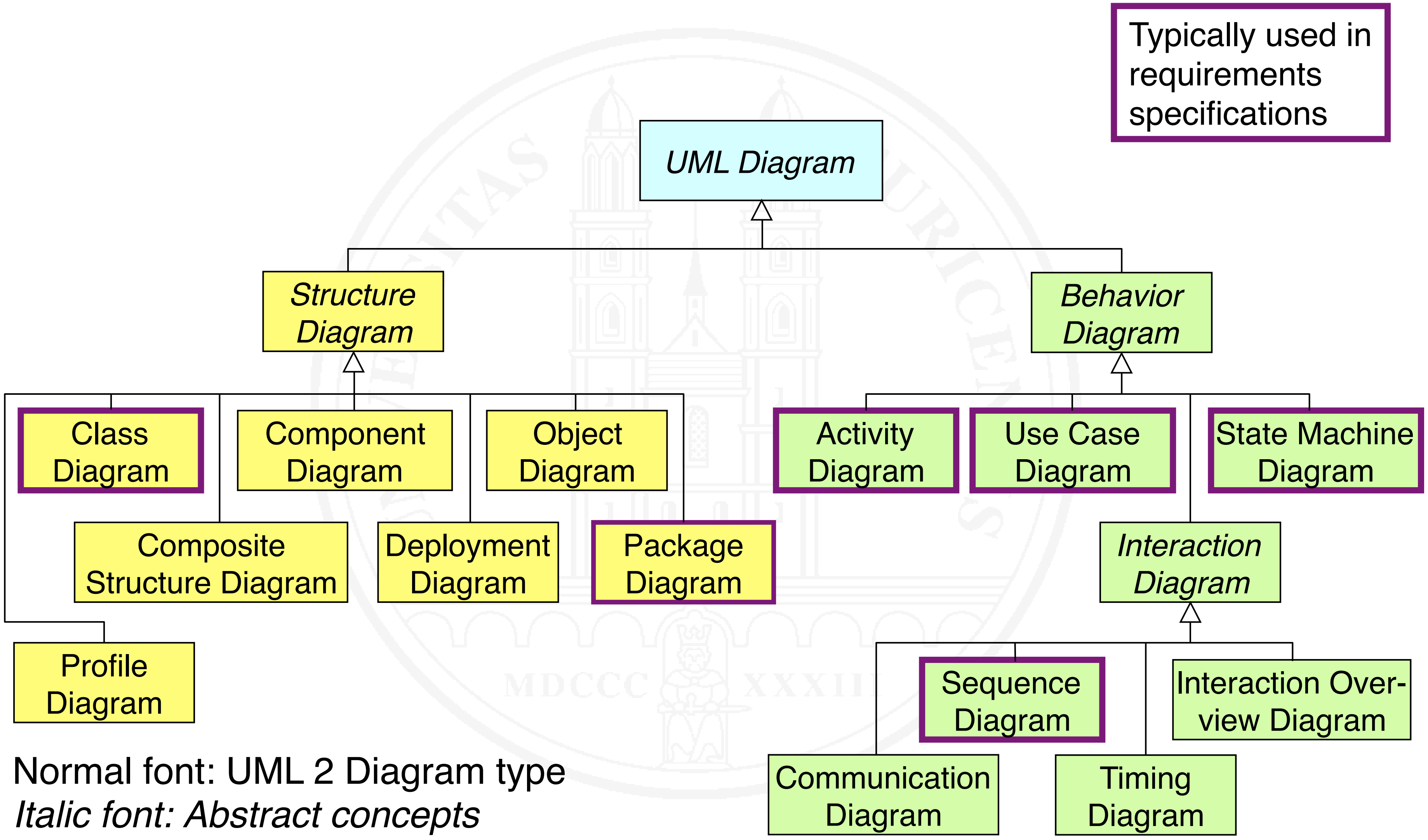
9.7 UML (Unified Modeling Language)

[Object Management Group 2015]

- 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 2012]

UML – Overview of diagram types



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

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, 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

- Originally developed for specifying complex data from 1977
- Signatures of operations define the syntax
- Axioms (expressions being always true) define semantics
- Axioms primarily describe properties that are invariant under execution of operations

+ Purely descriptive and mathematically elegant

– Hard to read

– Over- and underspecification difficult to spot

– Has never made it from research into industrial practice

Algebraic specification: a simple example

Specifying a stack (last-in-first-out) data structure

Let bool be a data type with a range of {false, true} and boolean algebra as operations. Further, let elem be the data type of the elements to be stored.

```
TYPE Stack
FUNCTIONS
new:   ()            →  Stack;  -- Create new (empty) stack
push:  (Stack, elem) →  Stack;  -- add an element
pop:   Stack         →  Stack;  -- remove most recent element from stack
top:   Stack         →  elem;   -- returns most recent element
empty: Stack         →  bool;   -- true if stack is empty
full:  Stack         →  bool;   -- true if stack is full
```

Algebraic specification: a simple example – 2

AXIOMS

$\forall$ s $\in$ Stack, e $\in$ elem

(1) $\neg$ full(s) $\rightarrow$ pop(push(s,e)) = s	-- *pop* reverses the effect of push
(2) $\neg$ full(s) $\rightarrow$ top(push(s,e)) = e	-- *top* retrieves the most recently stored element
(3) empty(new) = true	-- a *new* stack is always empty
(4) $\neg$ full(s) $\rightarrow$ empty(push(s,e)) = false	-- after *push*, a stack is not empty
(5) full(new) = false	-- a *new* stack is not full
(6) $\neg$ emtpy(s) $\rightarrow$ full(pop(s)) = false	-- after *pop*, a stack is not full

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 2012)
- 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

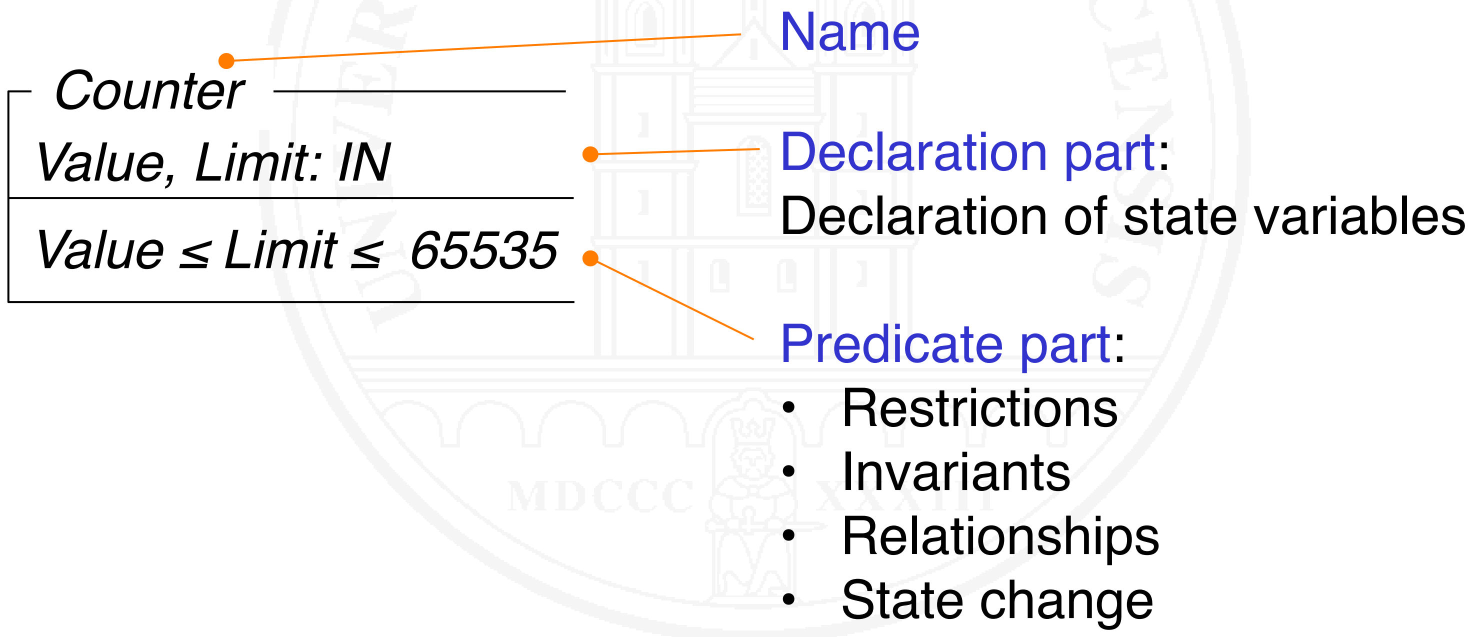
- Z is set-based
- Specification consists of sets, types, axioms and schemata
- Types are elementary sets: *[Name]* *[Date]* *IN*
- Sets have a type: *Person: ℙ Name* *Counter: IN*
- Axioms define global variables and their (invariant) properties

string: ***seq*** *CHAR*	Declaration
#string ≤ 64	Invariant

IN	Set of natural numbers
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: ℙ (Part x Supplier x Date)

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

  Birthday: Person → Date

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

State change through operations:

Increment counter

Δ 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: $\mathbb{P}$ *Book*
User: $\mathbb{P}$ *Person*
lent: *Book* ⇸ *Person*

---

dom *lent* $\subseteq$ *Stock*
ran *lent* $\subseteq$ *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? ∈ Stock\ dom lent
Borrower? ∈ User
lent' = lent ∪ {(BookToBeBorrowed?, Borrower?)}
Stock' = Stock
User' = User
─────────────────────────────────────────────
```

x?	*x* is an input variable
a ∈ *X*	*a* is an element of set *X*
\	Set difference operator
∪	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}

$\mathit{InquiredBook?} \in \mathit{Stock}$
$\mathit{isAvailable!} = \textbf{if } \mathit{InquiredBook?} \notin \textbf{dom } \mathit{lent}$
$\qquad\qquad \textbf{then } \mathit{yes} \textbf{ else } \mathit{no}$

ΞS	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 ℙ Document
│ Employee: ℙ Person
│ authorized: ℙ (Document x Person)
│ prohibited: 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.

.

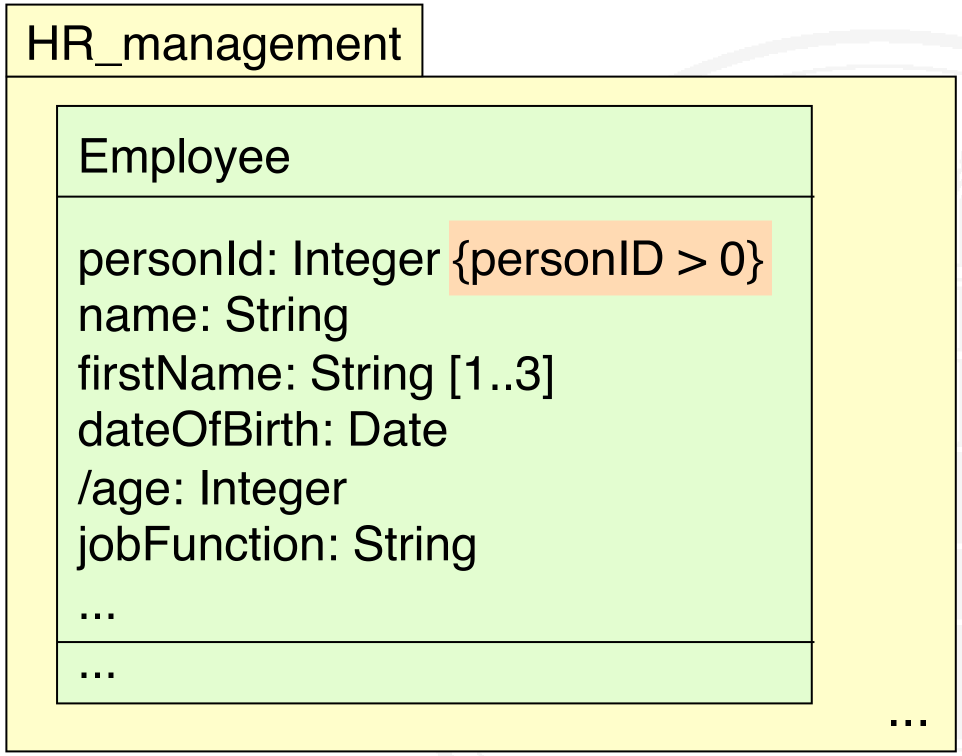
10.4 OCL (Object Constraint Language)

- What is OCL?
  - A textual formal language
  - Served 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 2.3.1
  - Also published as an ISO standard: ISO/IEC 19507:2012

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

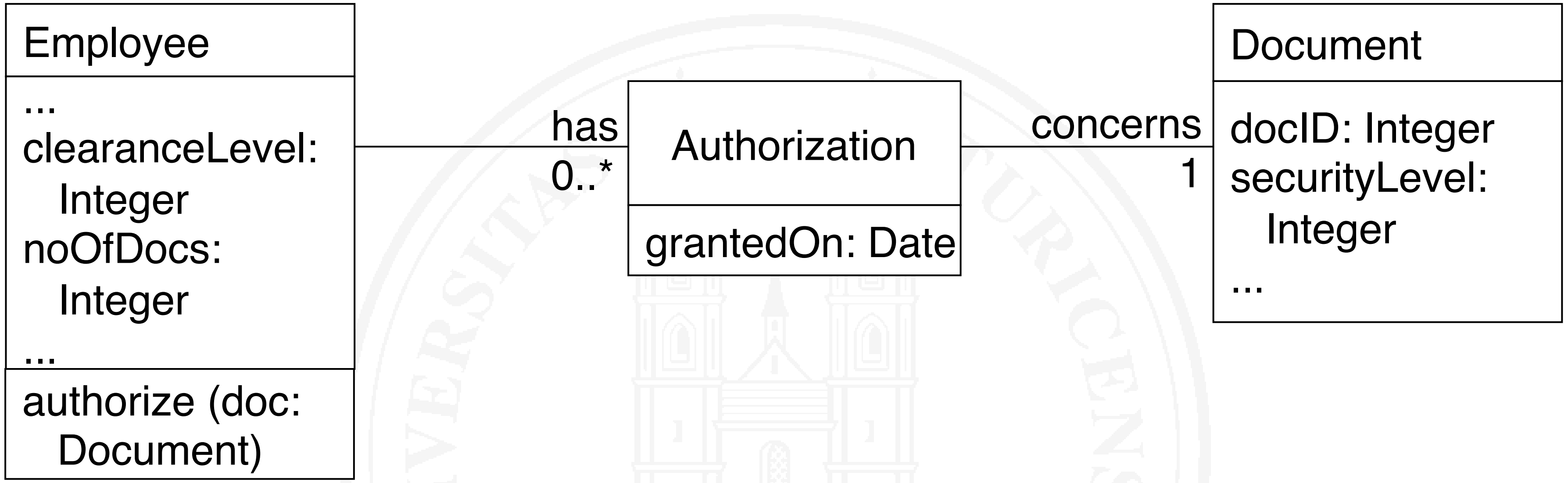


- OCL expression may be part of a UML model element
- Context for OCL expression is given implicitly

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

- 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 |
  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

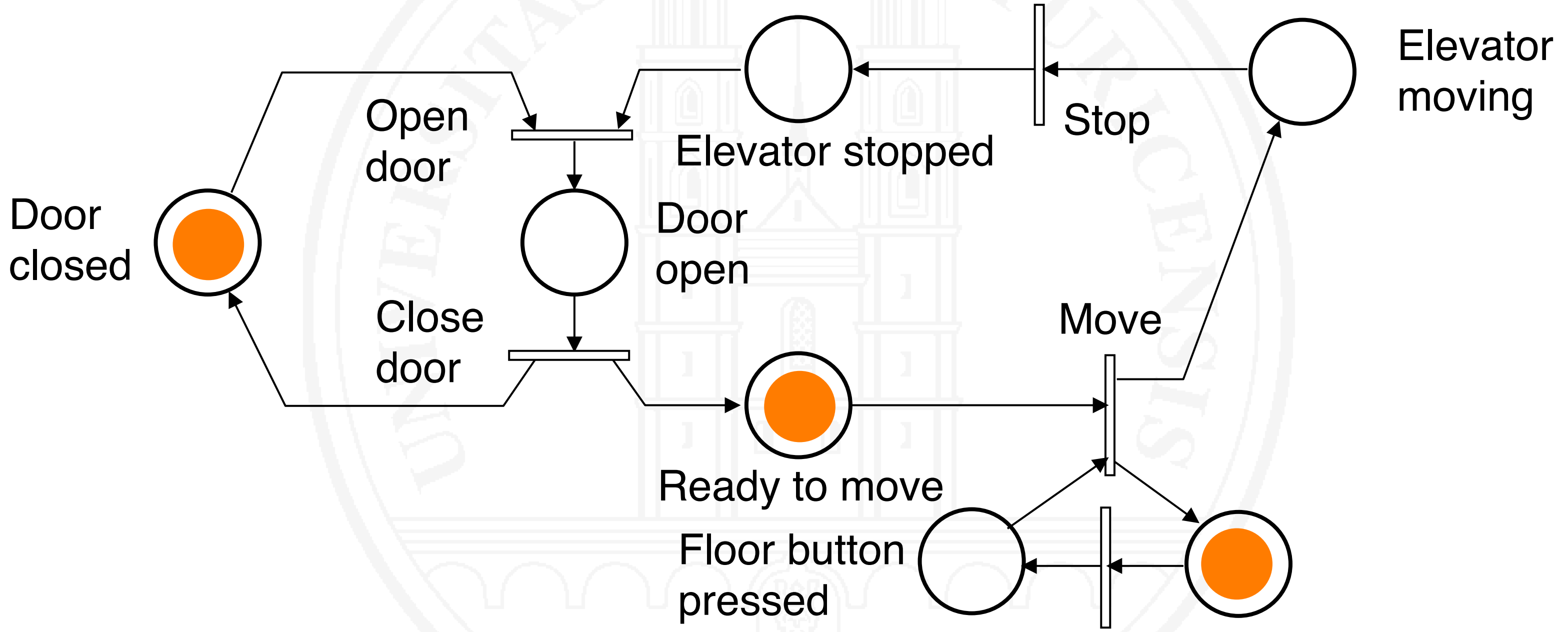
+ Model checking is fully automatic and produces counter-examples in case of failure

– 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:

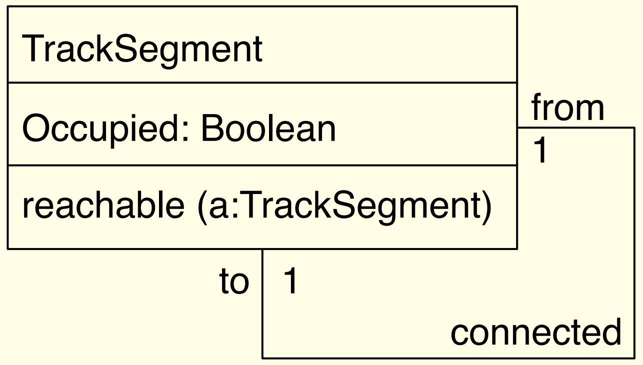
(P) The places *Door open* and *Elevator moving* never hold tokens at the same time

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) ⇒ If *Ready to move* has a token, *Door open* hasn't one (3)
- (2) ⇒ If *Elevator moving* has a token, *Door open* hasn't one (4)
- If *Door open* has no token, *Door closed* must have one (5)
- (1) & (3) & (4) & (5)⇒ (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 I 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 | 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)

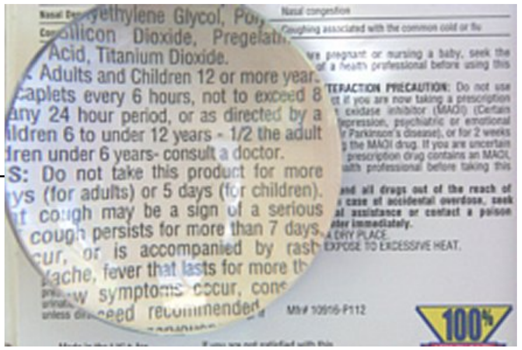
Limitations / problems

- 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 5 in Chapter 2)
- Validate content, form of documentation and agreement
- Establish short feedback cycles
- Use appropriate techniques
- Exemplify and disambiguate with acceptance test cases

Validation of content

Identify requirements that are

- Inadequate
- Incomplete or missing
- Inconsistent
- Wrong

Also look for requirements with these quality defects:

- Not verifiable
- Unnecessary
- Not traceable
- Premature design decisions

Validation of documentation

Scope: checking the requirements documentation (e.g., a systems requirements specification) for formal problems

Identify requirements that are

- Ambiguous
- Incomprehensible
- Non-conforming to documentation rules, structure or format

Validation of 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?

Some validation principles

General principles

- Work with the right people (i.e., stakeholders for requirements)
- Separate the processes of problem finding and correction
- Validate from different views and perspectives
- Validate repeatedly / continuously

Additional principles for requirements [Pohl and Rupp 2011]

- Validate by change of documentation type
  e.g., identify problems in a natural language specification by constructing a model
- Validate by construction of artifacts
  e.g., identify problems in requirements by writing the user manual, test cases or other development artifacts

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

Requirements Engineering tools

- Help find gaps and contradictions

Acceptance test cases

- Help disambiguate / clarify requirements

Requirements validation techniques – 2

Simulation/Animation

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

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

Formal Verification / Model Checking

- Formal proof of critical properties

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)
- Conflict of interest (divergent interests, e.g. cost vs. function)
- Conflict of value (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 all 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 must be written such that acceptance tests can be written to validate them
- Acceptance test cases can serve
  - for disambiguating requirements
  - as detailed specifications by example

Choosing acceptance test cases

Potential coverage criteria:

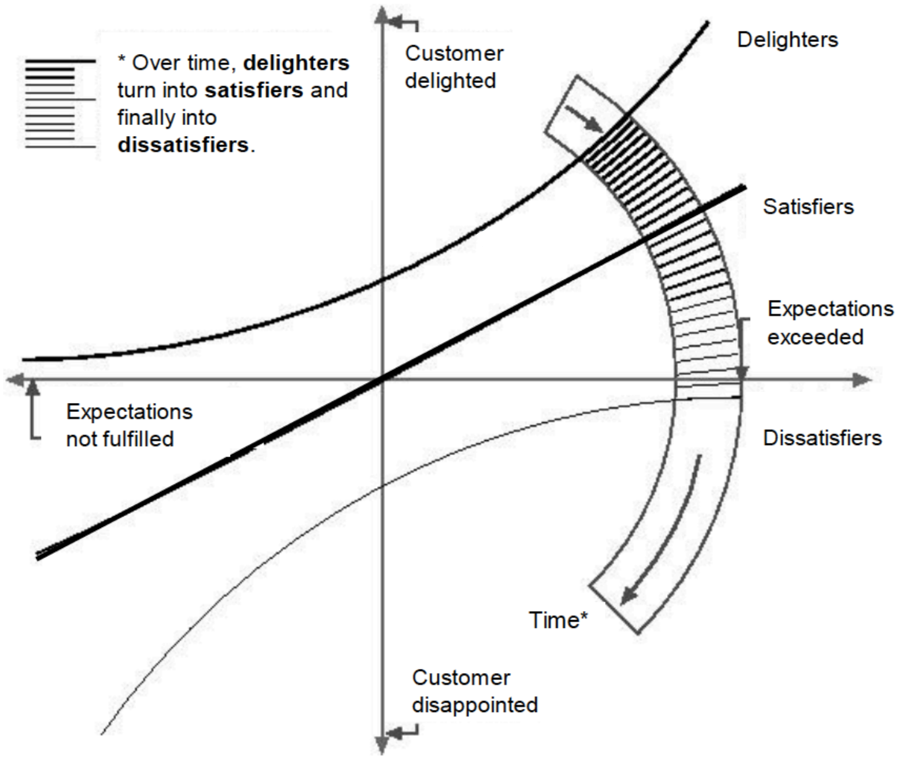
- Requirements coverage: At least one case per requirement
- Function coverage: At least one case per function
- Scenario coverage: For every type scenario / use case
  - All actions covered
  - All branches covered
- Consider the usage profile: not all functions/scenarios are equally frequent / important

12 Innovative requirements

Satisfying stakeholders is not enough (see Principle 7 in Chapter 2)

- 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

[Kano et al. 1984]



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

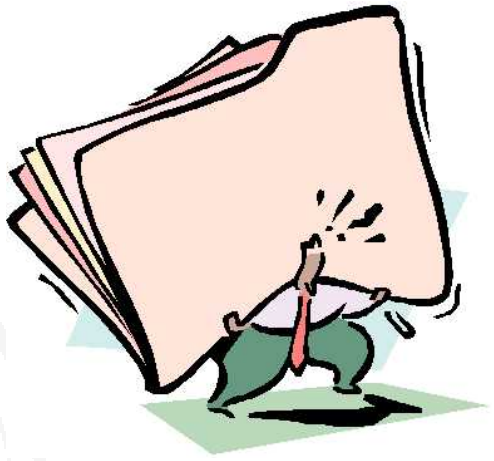
How to create innovative requirements?

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]

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]

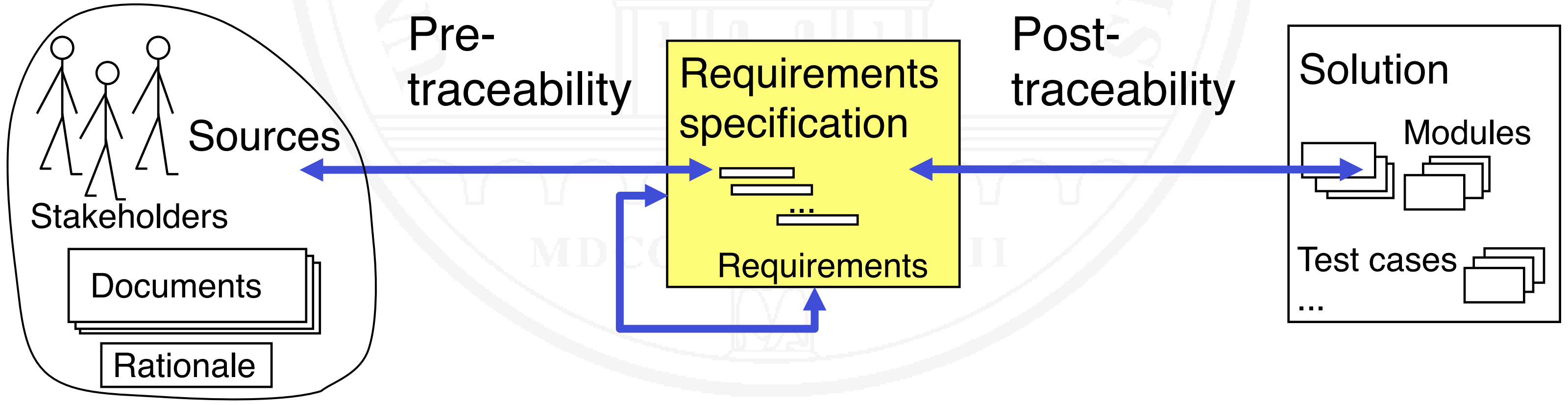
DEFINITION. Traceability – The ability to trace a requirement

(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

Every solution to this problem needs

- ❍ Requirements configuration management
- ❍ With long development cycles, also requirements change management is required

Requirements change management

- Configuration management for requirements
  - Versioning of requirements
  - Ability to create configurations, baselines and releases
- Strict change process
  - (1) Submit change request
  - (2) Triage. Result: [OK I NO I Later (add to backlog)]
  - (3) If OK: Impact analysis
  - (4) Submit result and recommendation to Change Control Board
  - (4) Decision by Change Control Board
  - (5) If positive: make the change, create new baseline/release, (maybe) adapt the contract between customer and supplier