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*
#string ≤ 64

Declaration

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

Counter

Value, Limit: IN

$Value \leq Limit \leq 65535$

Name

Declaration part:
Declaration of state variables

Predicate part:
- Restrictions
- Invariants
- Relationships
- State change

Relations, functions und operations

- Relations and functions are ordered set of tuples:

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

  Birthday: Person $\rightarrow$ *Date*

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

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

Mathematical equality, no assignment!

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

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

$\mathbf{dom}\ lent \subseteq Stock$
$\mathbf{ran}\ lent \subseteq User$

⇸	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? $\in$ *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? $\in$ *Stock*
isAvailable! = **if** *InquiredBook?* $\notin$ **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.

.

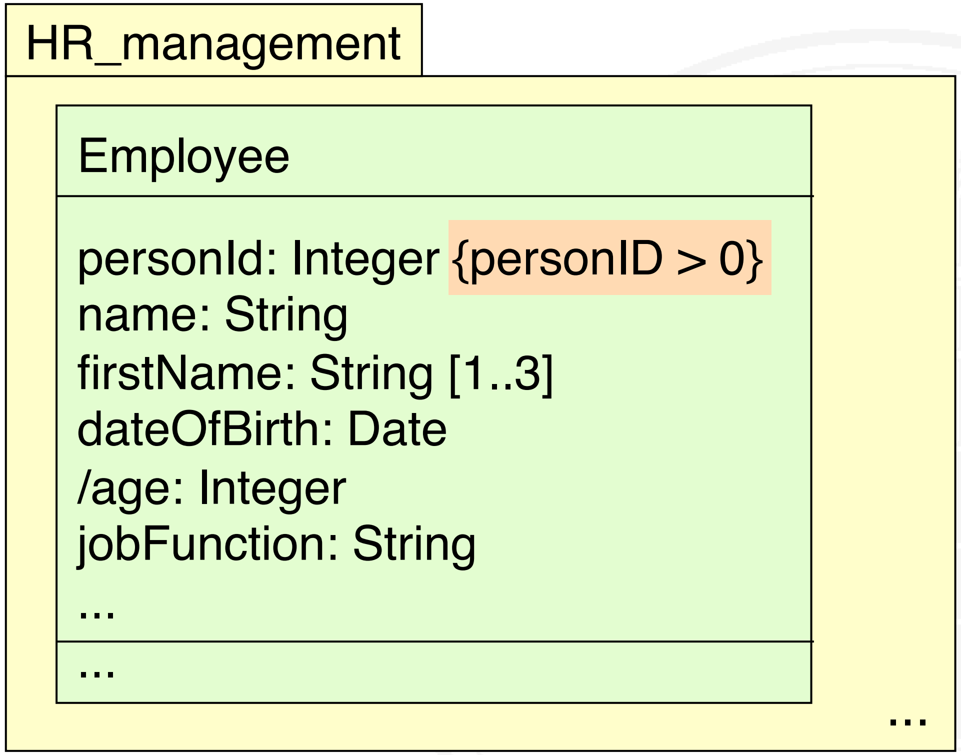
10.4 OCL (Object Constraint Language)

- 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 2.3.1
  - 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

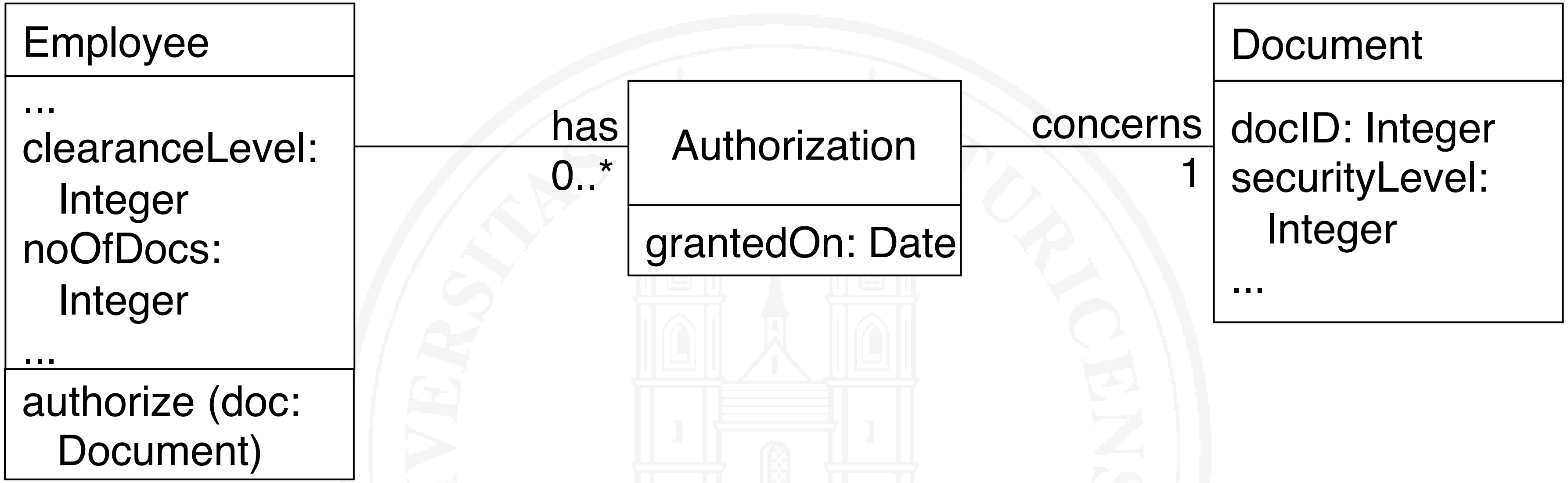
OCL expressions: invariants



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 |
  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

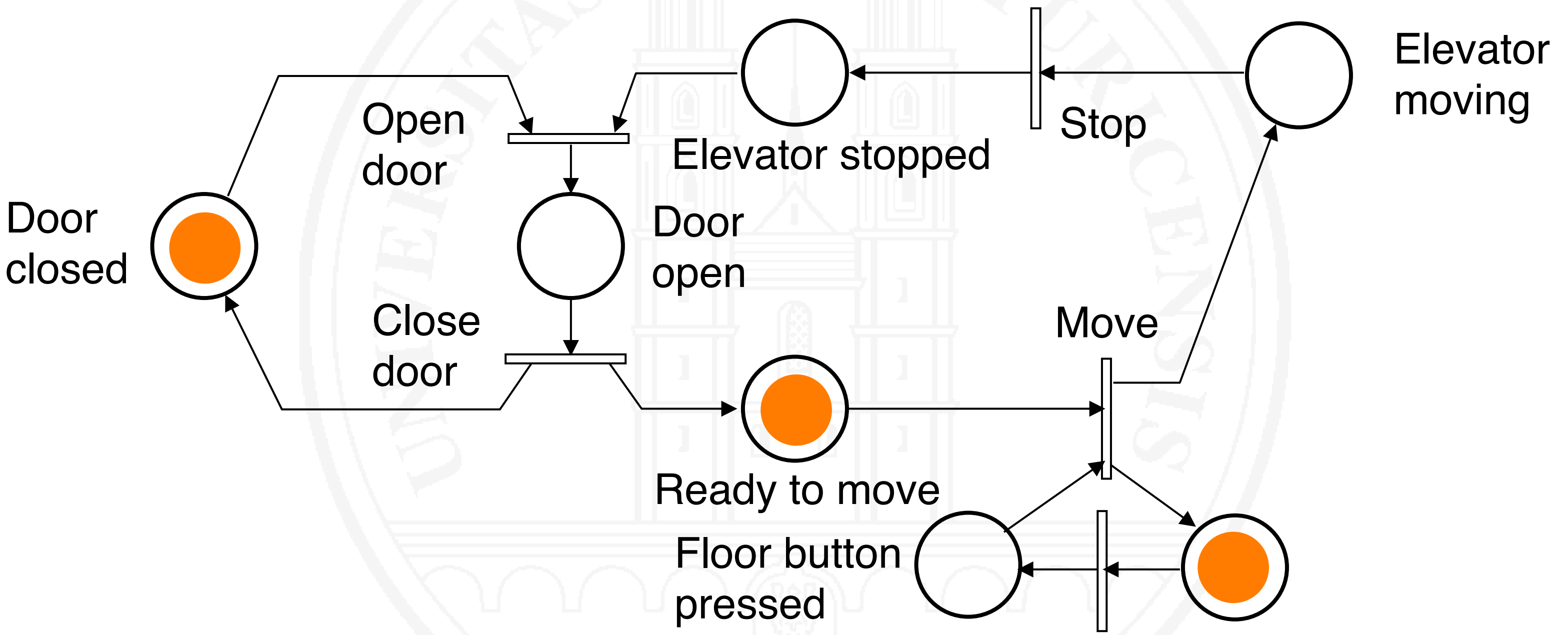
+ Model checking is fully automatic and produces counter-examples in case of failure

– 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:

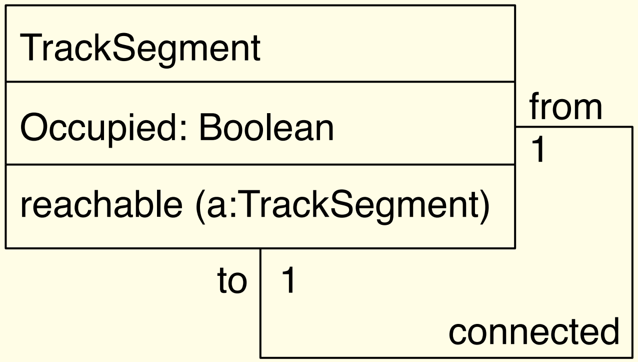
(P) The places *Door open* and *Elevator moving* never hold tokens at the same time

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 | 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)

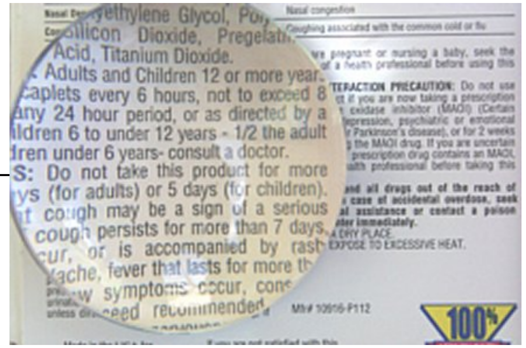
Limitations / problems

- 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 6 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:

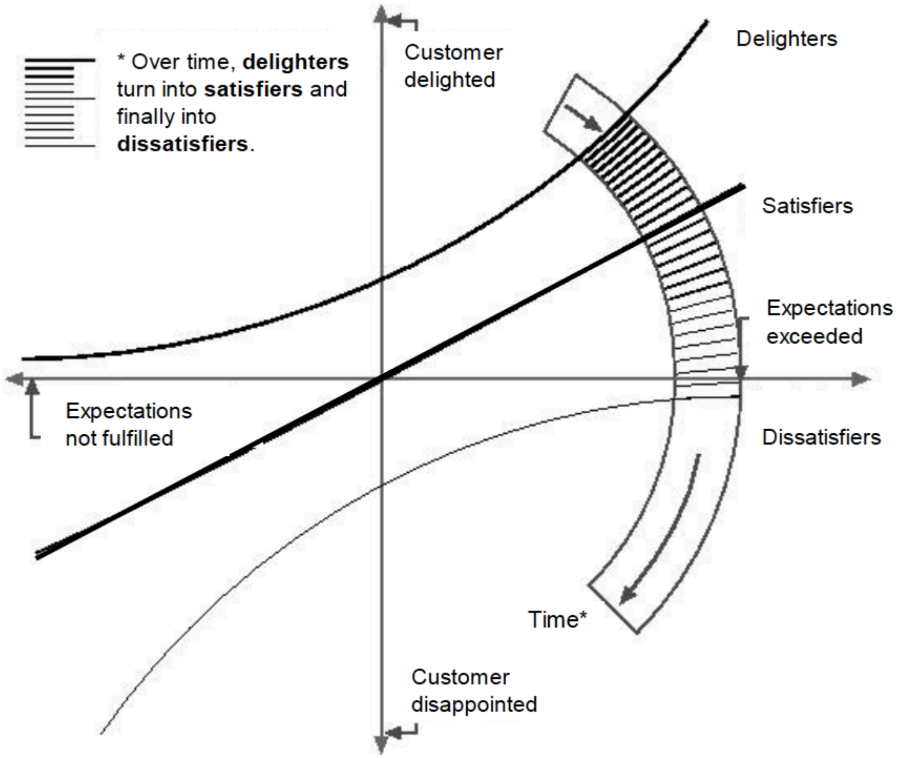
- Requirements coverage: At least one case per requirement
- Function coverage: At least one case per function
- Scenario coverage: For every type scenario / use case
  - All actions covered
  - All branches covered
- Consider the usage profile: not all functions/scenarios are equally frequent / important

12 Innovative requirements

Satisfying stakeholders is not enough
(see Principle 8 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]



How to create innovative requirements?

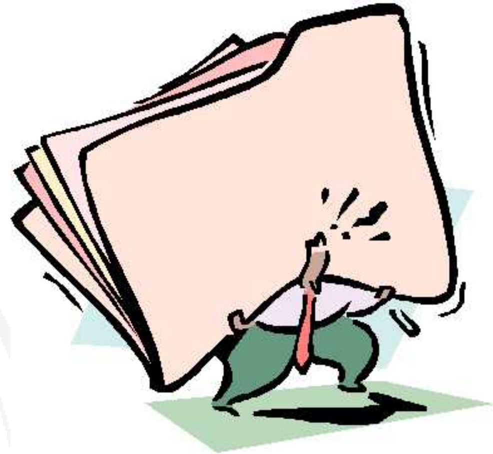
Encourage out-of-the-box thinking

- 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

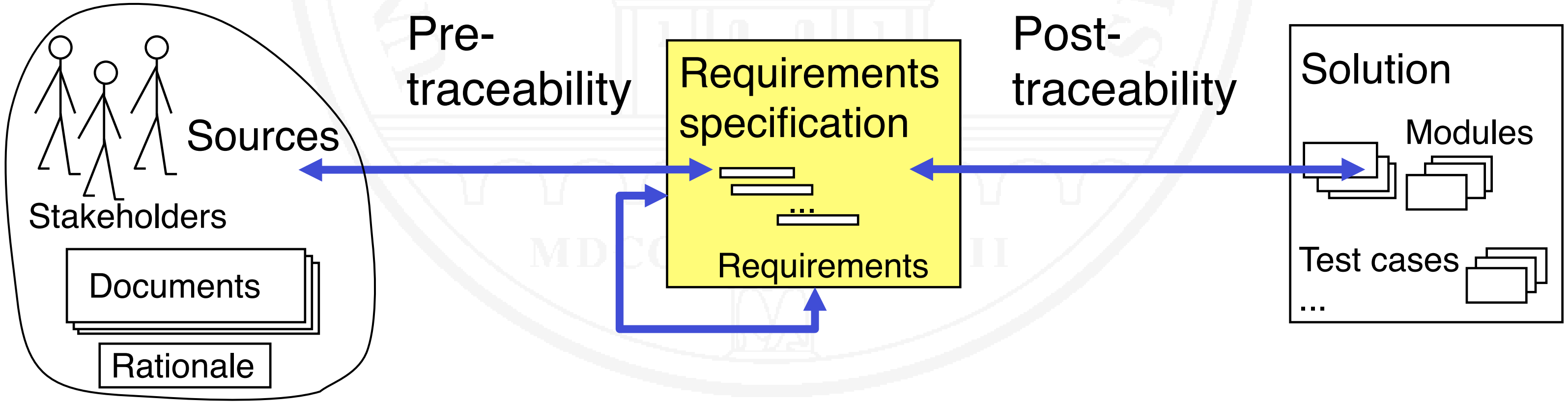
- Wiegers' matrix [Wiegers 1999]
  - Estimates relative benefit, detriment, cost, and risk for each requirement
  - Uses these values to calculate a weighted priority
  - Ranks according to calculated priority values
- AHP (Analytic Hierarchy Process) [Saaty 1980]
  - An algorithmic multi-criterion decision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

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

Requirements change management

Adhering to a strict change process

(1) Submit change request

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

(3) If OK: Perform 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 client and supplier

Change control board – A committee of client 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