Model-Based Requirements Engineering

Tutorial 2010-02-09

by

Kristian Sandahl

Planned topics

- What are requirements?
- Modelling requirements in UML
- Requirement model traceability
- Non-functional software requirements
- Short introduction to requirements in SysML
- Short introduction to formal methods

Requirements

- "Software requirements express the needs and constraints placed on a software product that contribute to the solution of some real-world problem."

(*Kotonya and Sommerville 2000*)

Process model:

- Elicitation
- Analysis
- Specification
- Validation

Elicitation

Purpose:

- Understand the **true** needs of the customer
- Trace future implementation to needs

Sources:

- Goals
- Domain knowledge
- Stakeholders
- Environment

Techniques:

- Interviews
- Scenarios
- Prototypes
- Facilitated meetings
- Observation

Analysis: Goal

- Detect and resolve conflicts btwn requirements
- Discover bounds of software
- Define interaction with the environment
- Elaborate high-level requirements to derive detailed requirements

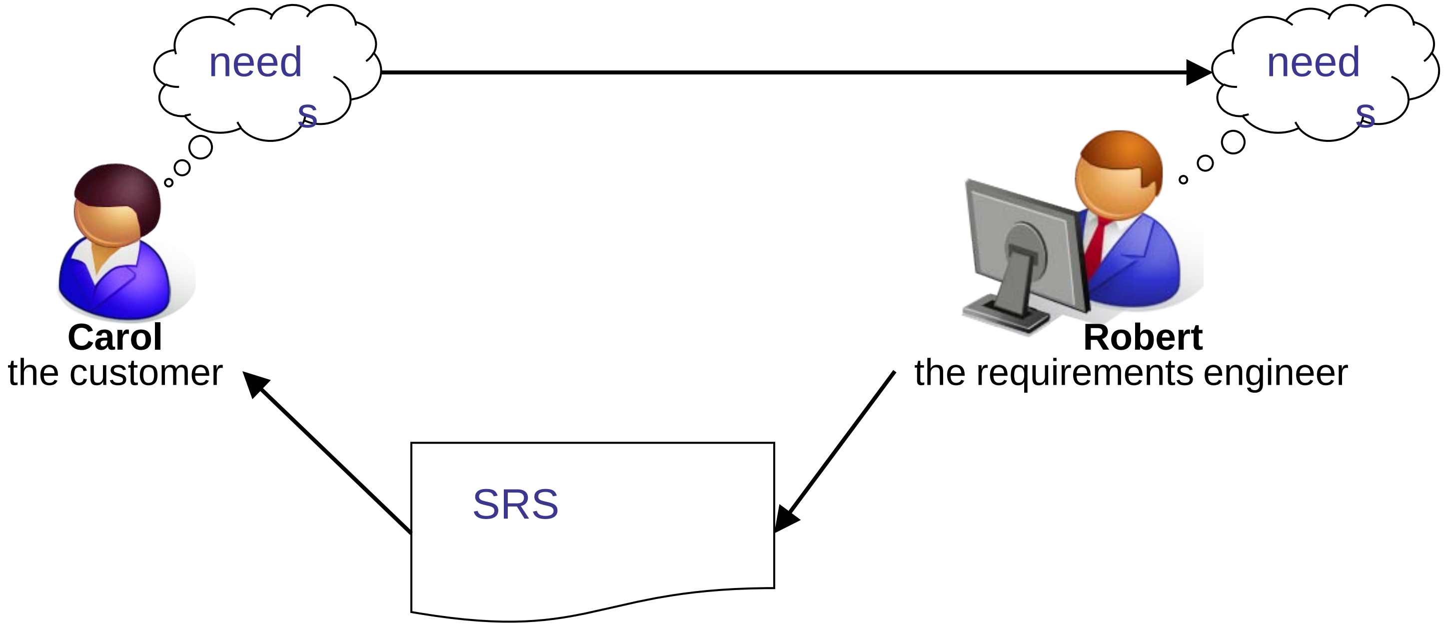
Analysis: Requirements classification

- Functional vs non-functional requirements
- Source
- Product or process requirements
- Priority
- Scope in terms of affected components
- Volatility vs stability

Analysis: Conceptual Modelling

- Representation in semi-formal notation
- Often diagrammatic representation
- Examples:
  - Object-orientation, use-cases, state-machines
  - Activity diagrams
  - Data flow diagrams
  - Entity-relationship models

Specification



- There is no perfect specification, but you can write a good one
- The RS, or SRS avoids many misunderstandings
- The RS is of special importance in outsourcing programming

SRS contents

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 General constraints
- 2.5 Assumptions and dependencies
- 2.6 Lower ambition levels

3 Specific requirements

3.1 Interface requirements
- 3.1.1 User interfaces
- 3.1.2 Hardware interfaces
- 3.1.3 Software interfaces
- 3.1.4 Communication interfaces

3.2 Functional requirements

3.3 Performance requirements

3.4 Design constraints

3.5 Software system attributes

3.6 Other requirements

4 Supporting information
- 4.1 Index
- 4.2 Appendices

Individual requirements

Requirement #: Requirement Type: Event/use case #:

Description:

Rationale:

Source:
Fit Criterion:

Customer Satisfaction: Customer Dissatisfaction:
Dependencies: Conflicts:
Supporting Materials:
History:

Volere

Requirements specification

Requirements are:

- Numbered
- Inspected
- Prioritised
- Unambiguous
- Testable
- Complete
- Consistent
- Traceable
- Feasible
- Modifiable
- Useful for:
  - operation
  - maintenance
  - customer
  - developer
  - ….

Validation of requirements

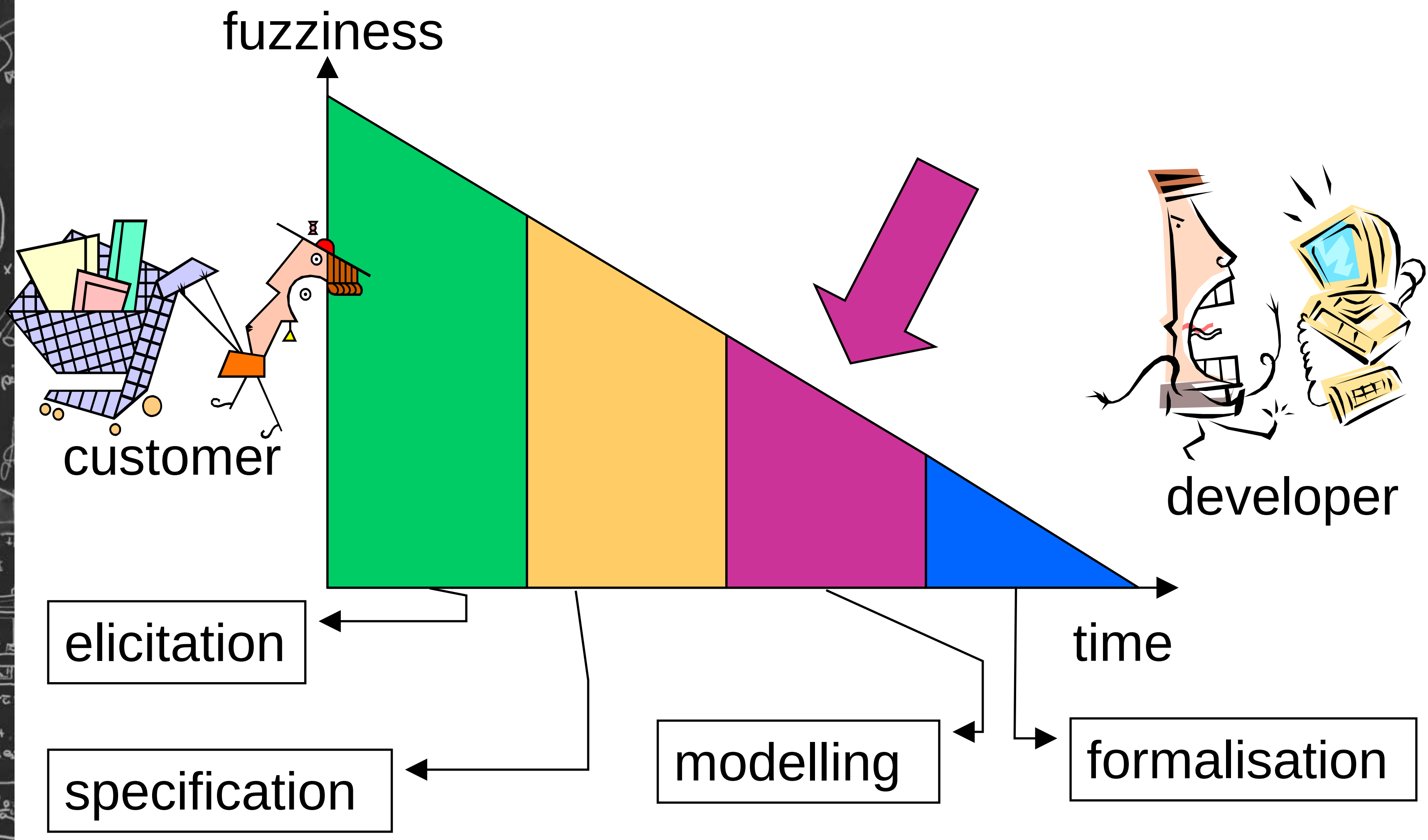
Before design and coding

- Inspections
- Cross-referencing
- Interviews
- Checklists
- Scenarios
- Proofs
- Model validation
- Simulation
- Prototyping

After (some) design and coding

- Prototyping
  - Overcomittment
  - Teach-back
- Alfa test
- Beta test
- Acceptance test

Requirement representation process



Introduction

- Models **supplement** natural language
- Models support both elicitation and design
- The boundaries between specification and design have to be decided
- There are high transition costs from functional to object-oriented models
- **UML** is becoming the standard notation

Develop complementary system models

Benefits:

- Forces analysis from different views
- Different readers take different views

Implementation:

- The UML 4+1 model
- Combination of other diagrams

Drawbacks:

- Different readers make different interpretation
- Normally weak exception handling
- Hard to model non-functional requirements

UML 4+1 Model

Views:

- Logical view: which parts belong together?
- Process view: what threads of control are there?
- Development view: what is developed by whom? reuse issues
- Physical view: which part will execute where?

+

- Use-case model: required system from the user's point of view. static and dynamic

Use-case modelling

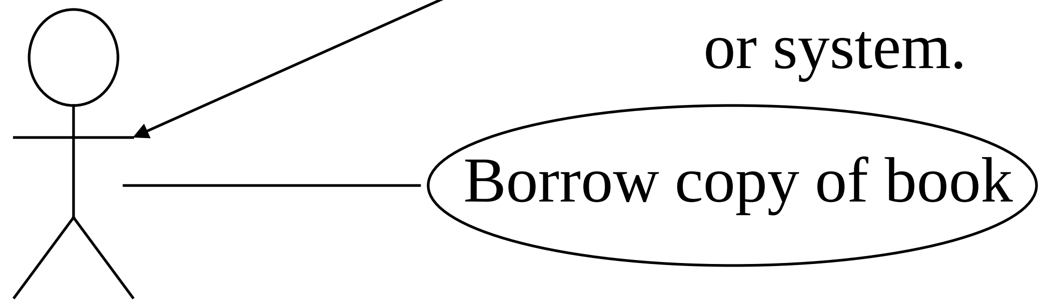
A use-case is:

"… a particular form or pattern or exemplar of usage, a scenario that begins with some user of the system initiating some transaction of sequence of interrelated events."

(*Jacobson, m fl 1992*)

Use-case diagram

Actor: a user of the system in a particular role. Can be human or system.

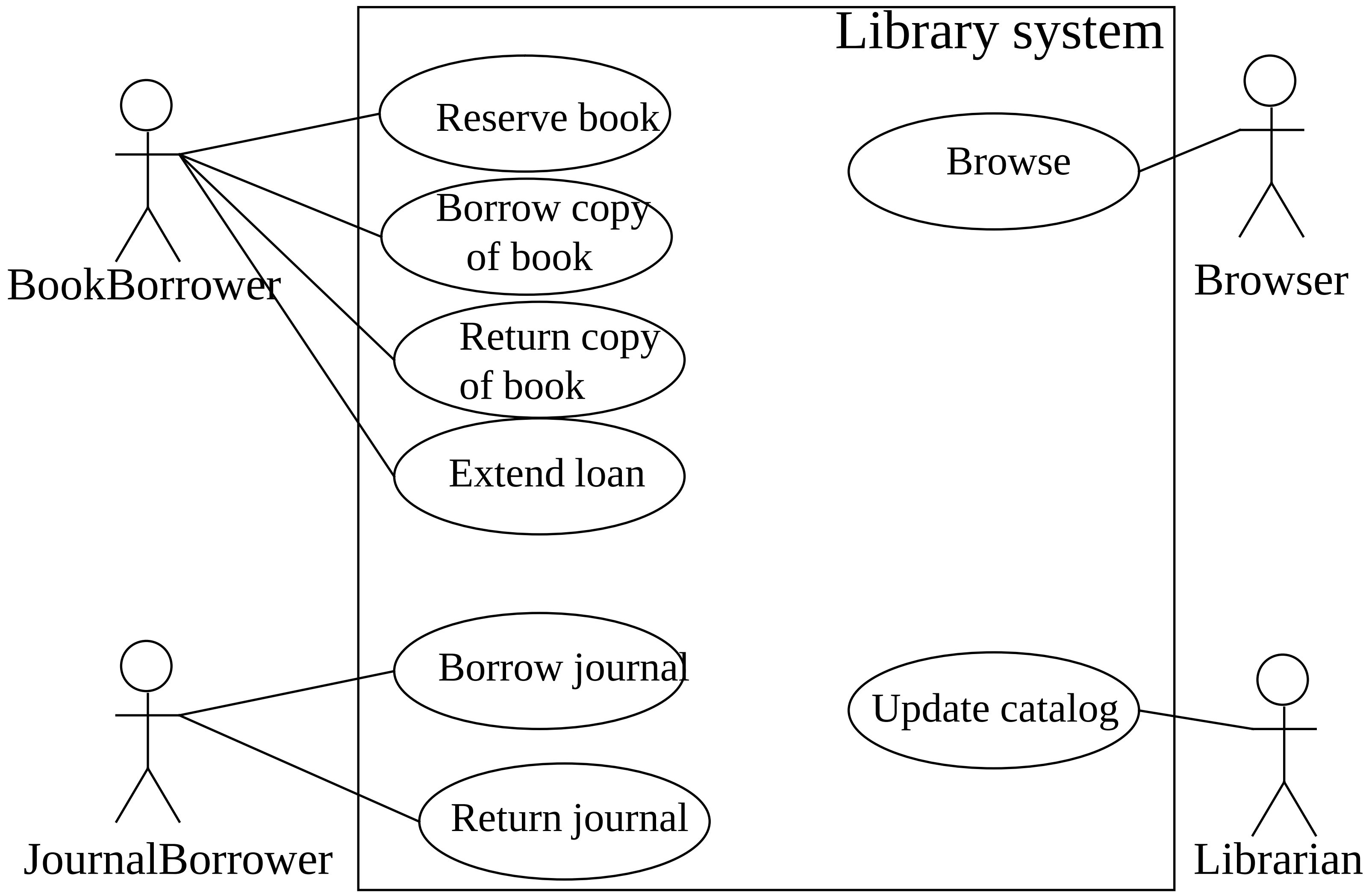


BookBorrower

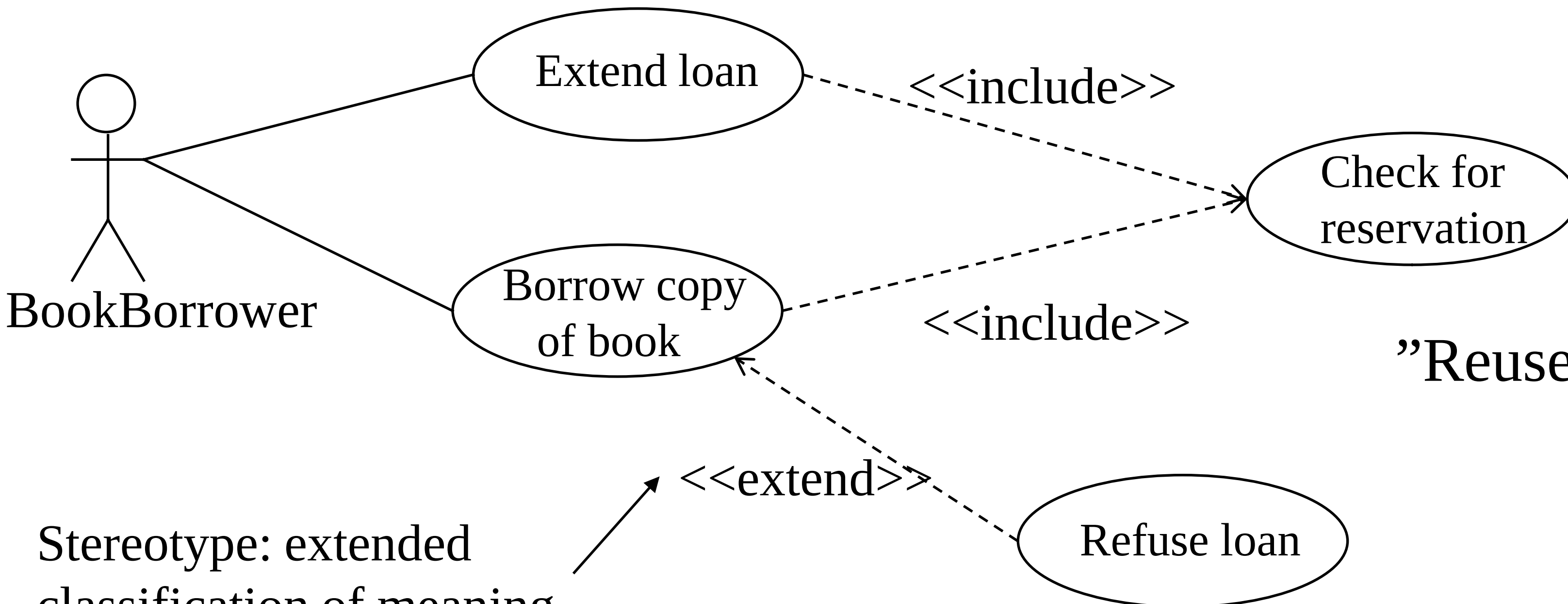
Detail of use-case →

A BookBorrower presents a book. The system checks that the potential borrower is a member of the library, and that he/she doesn’t already have the maximum permitted book on loan. This maximum is 6 unless the member is a staff member, in which case it is 12. If both checks succeed, the system records that this library member has this copy of the book on loan. Otherwise it refuses the loan.

Use-case diagram for the library



Relations between use-cases

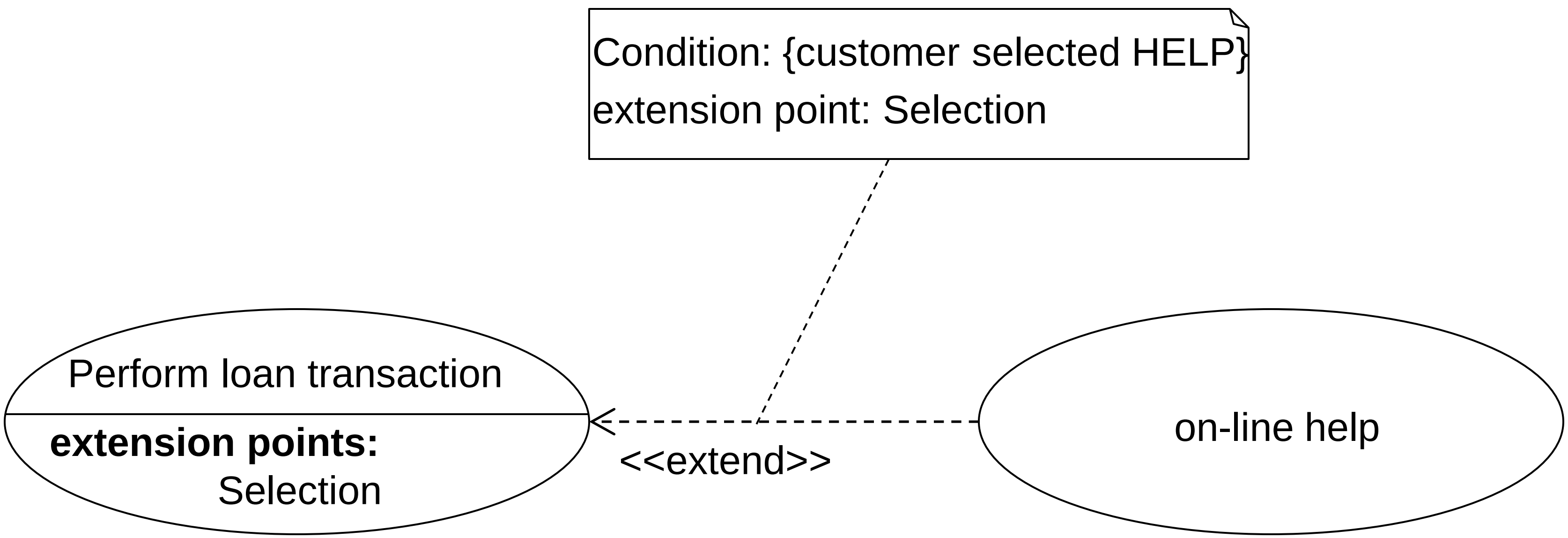


”Reuse”

Stereotype: extended classification of meaning

”Separating scenarious”

Extension points



Identifying classes: noun analysis

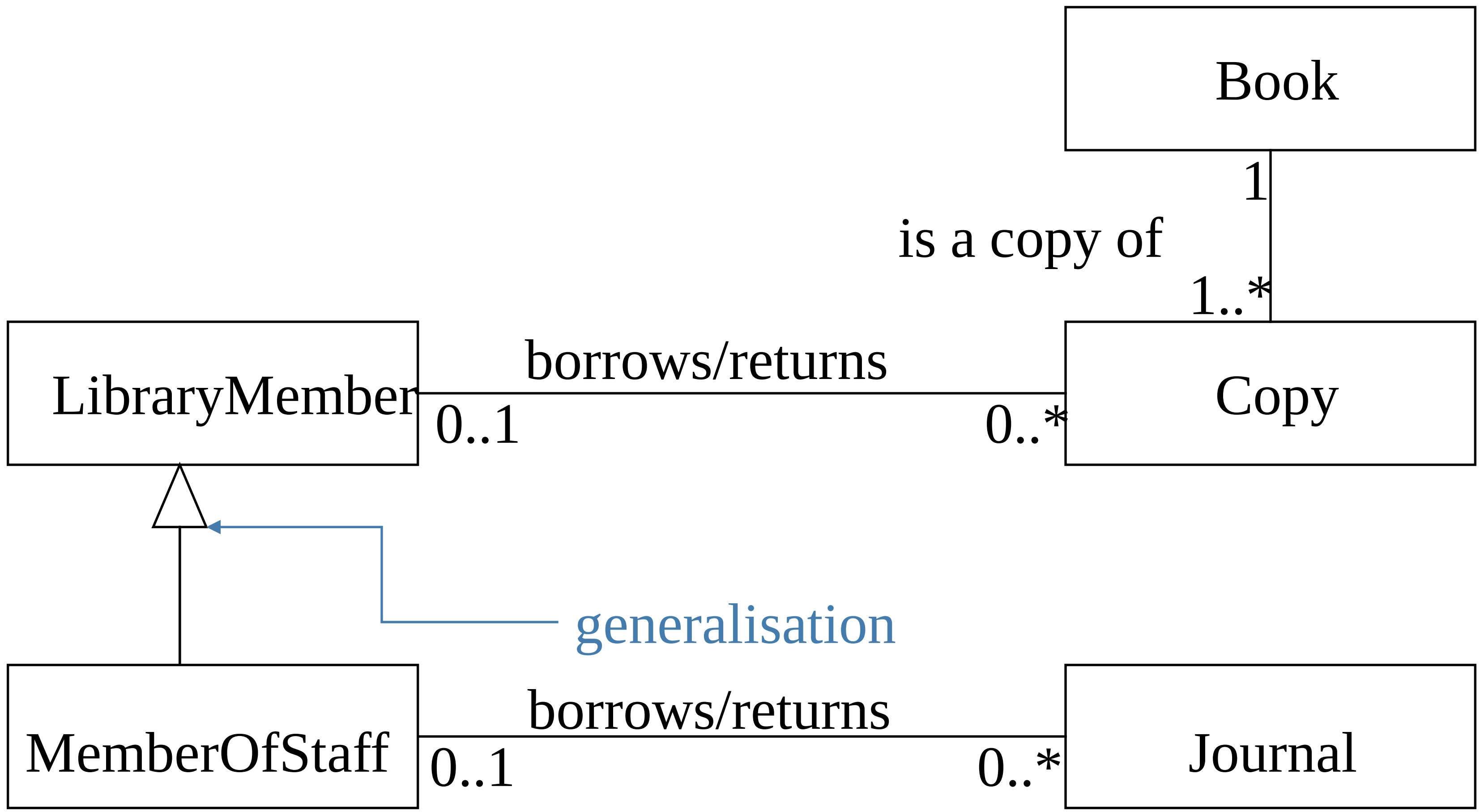
A BookBorrower presents a book. The system checks that the potential borrower is a member of the library, and that he/she doesn't already have the maximum permitted book on loan. This maximum is six unless the member is a staff member, in which case it is 12. If both checks succeed, the system records that this library member has this copy of the book on loan. Otherwise it refuses the loan.

- **book – real noun handled by the system**
- system – meta-language
- borrower – already actor
- **library member – handled by the system**
- **staff member – handled by the system**
- checks – event
- **copy of book – handled by the system**

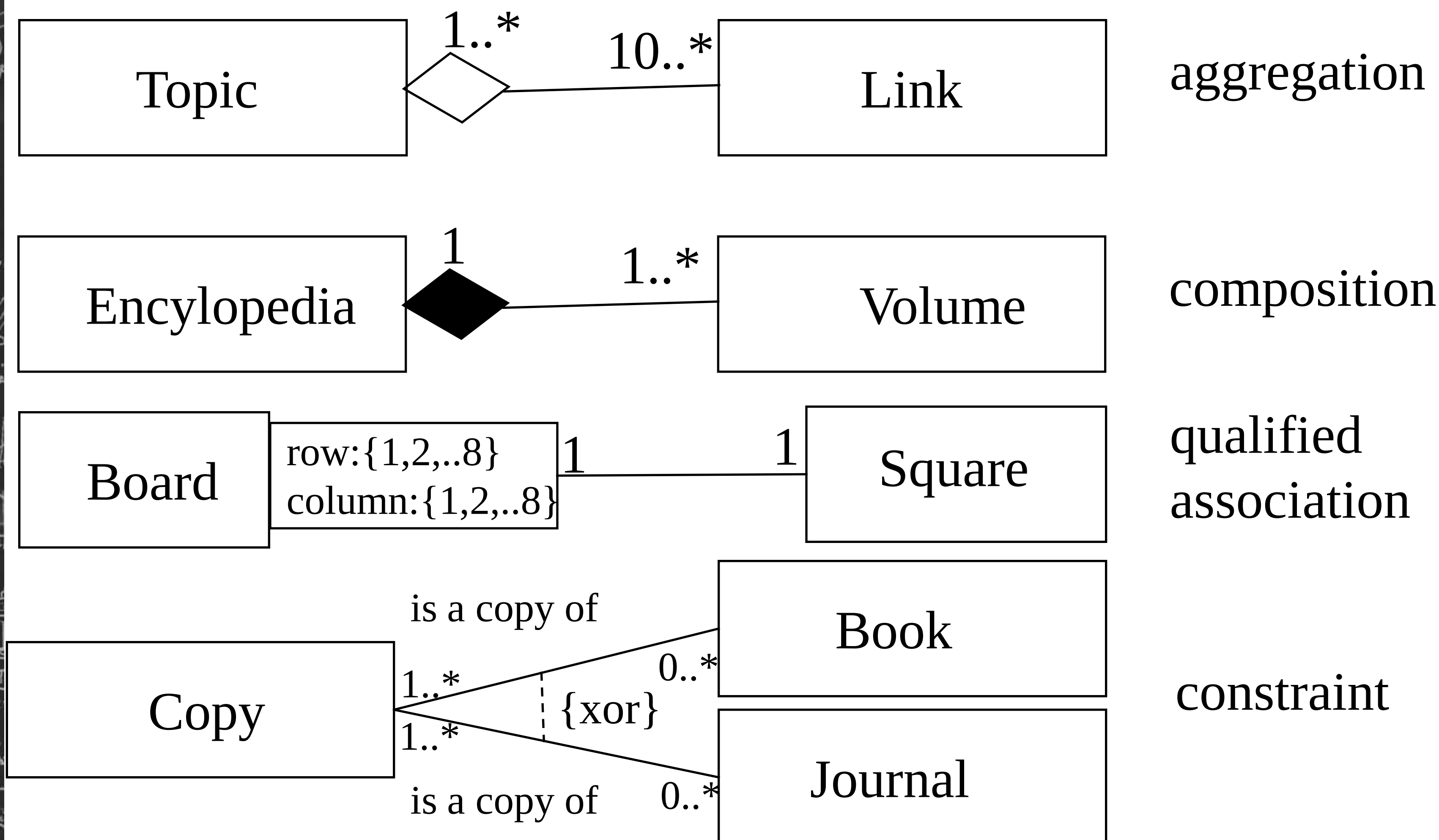
The single class model

Book	name
title: String	attribute
copiesOnShelf() : Integer borrow(c:Copy)	operations

The library class model



More relations between classes



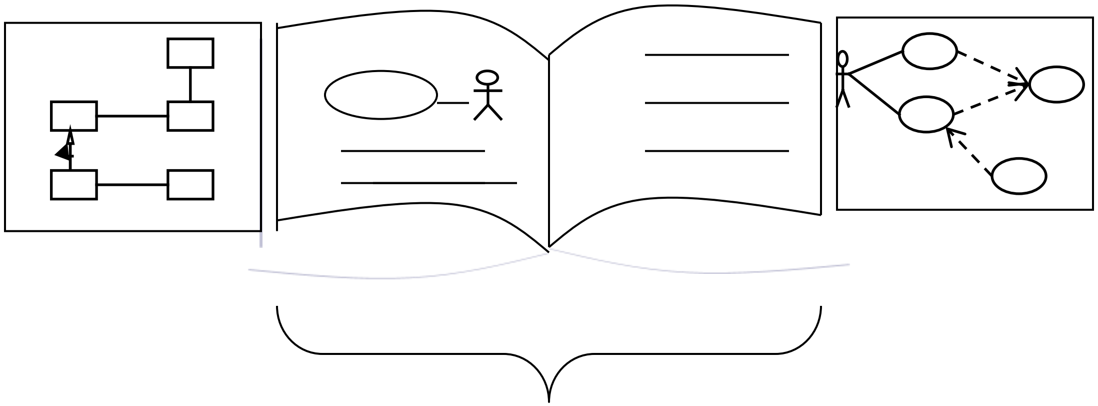
Where to go now?

1. Continue with a traditional specification
2. Writing a detailed use-case specification
3. Continue modelling

Writing a detailed use-case specification

- Name
- Brief Description
- Flow of Events: Write the description so that the customer can understand it. The flows can include a basic flow, alternative flows, and sub flows.
- (Key scenarios)
- Special Requirements
- Preconditions
- Post-conditions
- Extension points

“Classical” use-case specification



max 40 pages

Use-cases need System-wide requirements

1. Introduction
2. System-Wide Functional Requirements
3. System Qualities
   - 3.1 Usability
   - 3.2 Reliability
   - 3.3 Performance
   - 3.4 Supportability
4. System Interfaces

4.1 User Interfaces
   - 4.1.1 Look & Feel
   - 4.1.2 Layout and Navigation Requirements
   - 4.1.3 Consistency
   - 4.1.4 User Personalization & Customization Requirements

4.2 Interfaces to External Systems or Devices

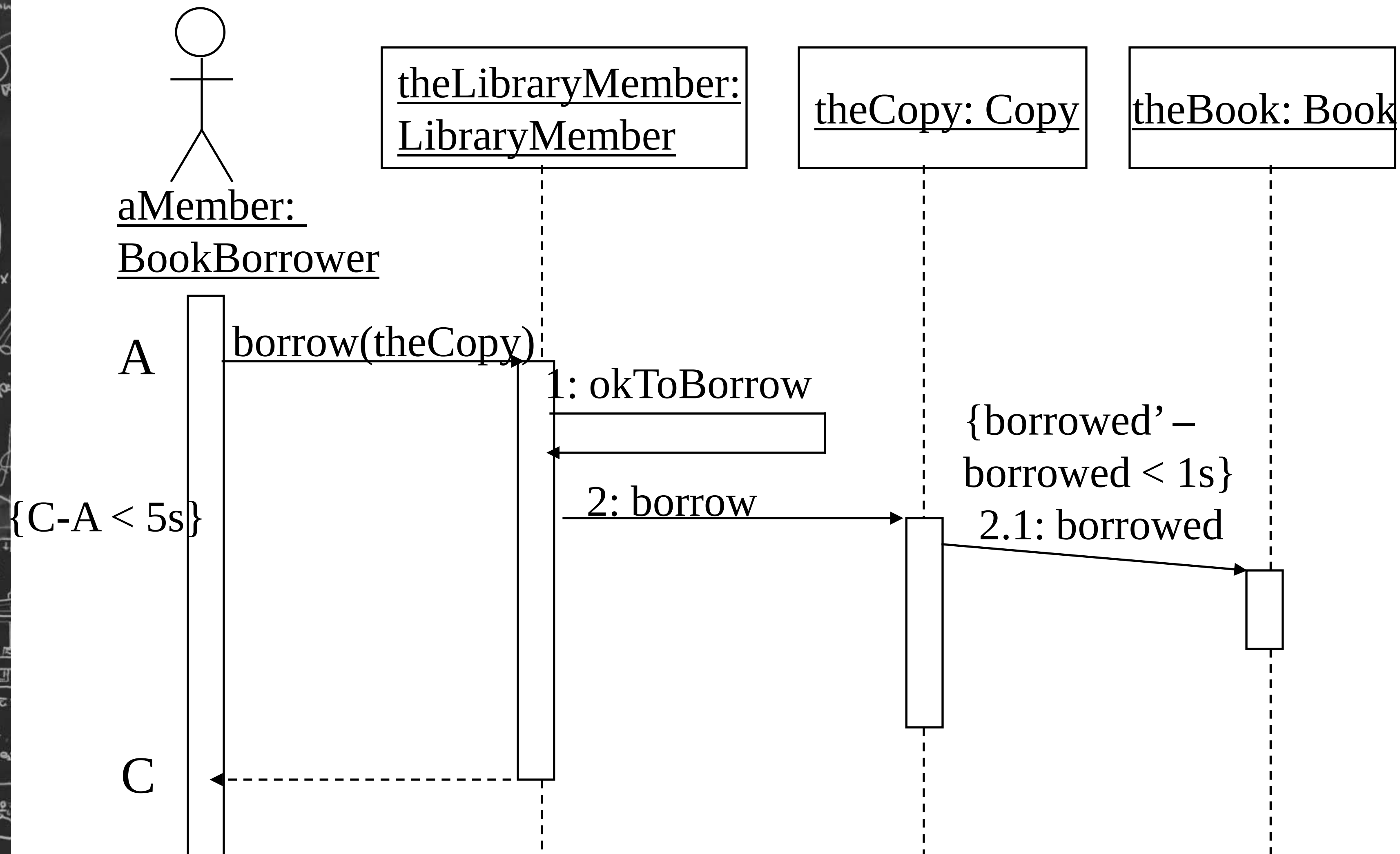
4.2.1 Software Interfaces

4.2.2 Hardware Interfaces

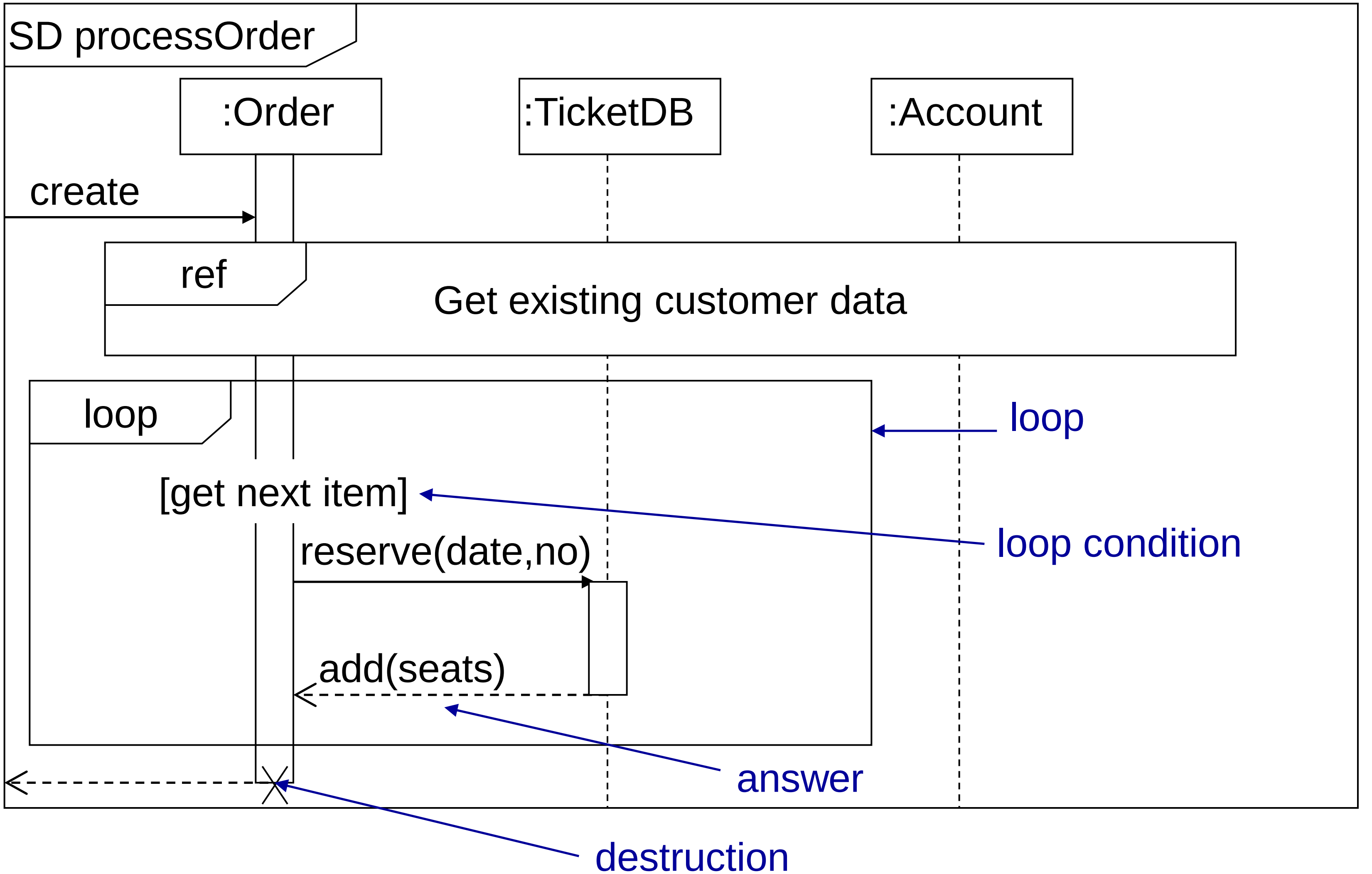
4.2.3 Communications Interfaces

5. Business Rules
6. System Constraints
7. System Compliance
   - 7.1 Licensing Requirements
   - 7.2 Legal, Copyright, and Other Notices
   - 7.3 Applicable Standards
8. System Documentation

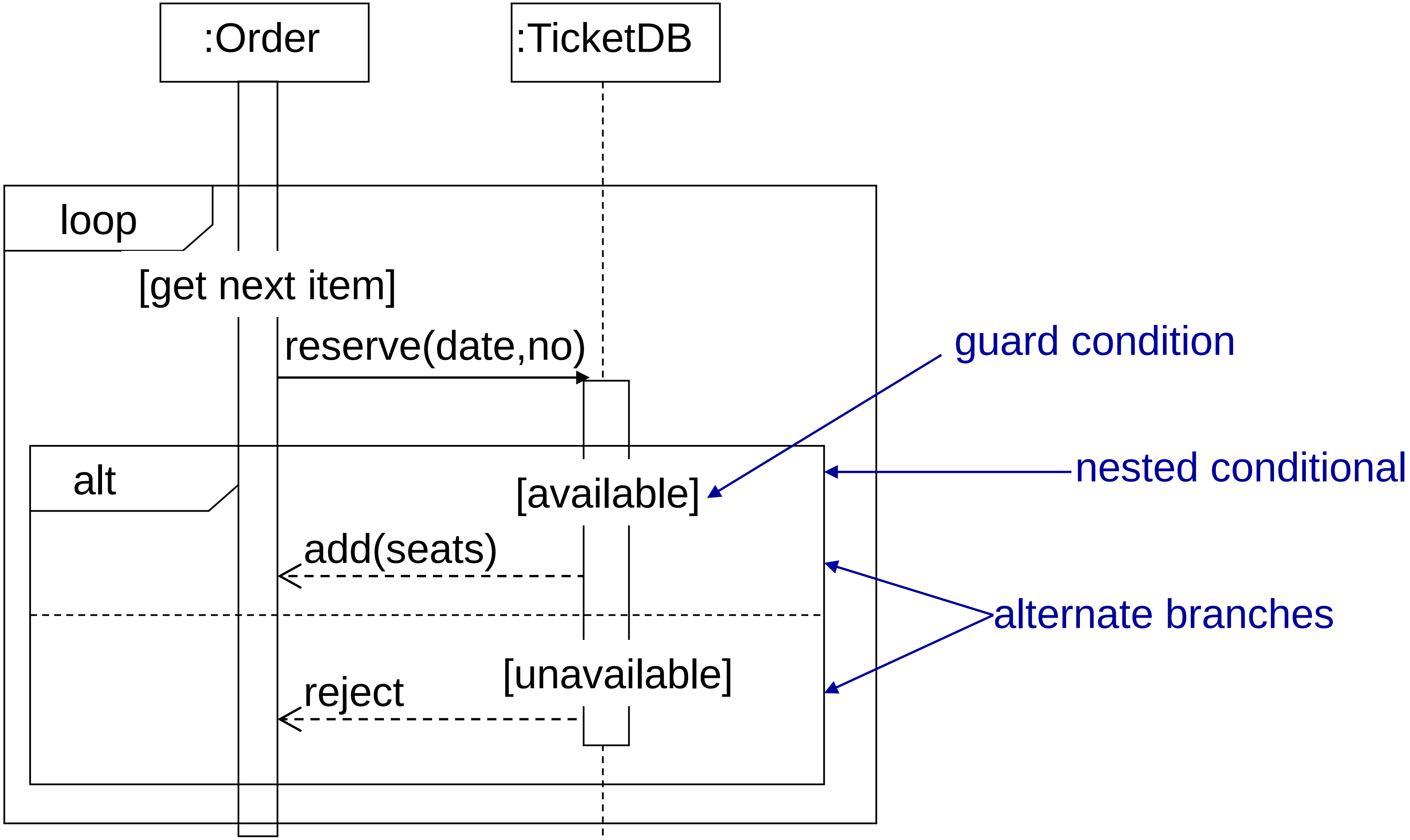
Continue modelling :Sequence diagram



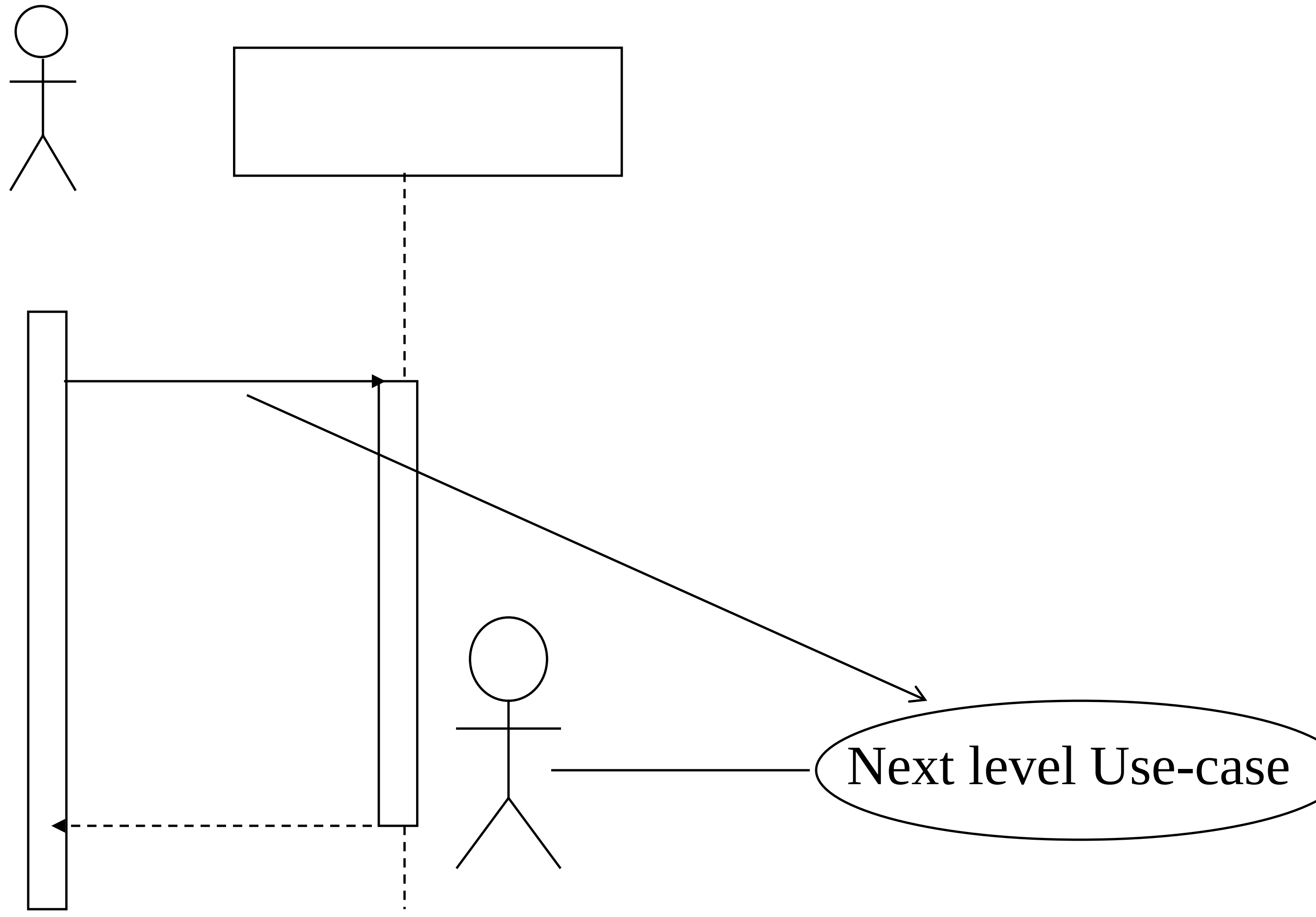
Combining fragments of sequence diagrams



More fragments of sequence diagrams

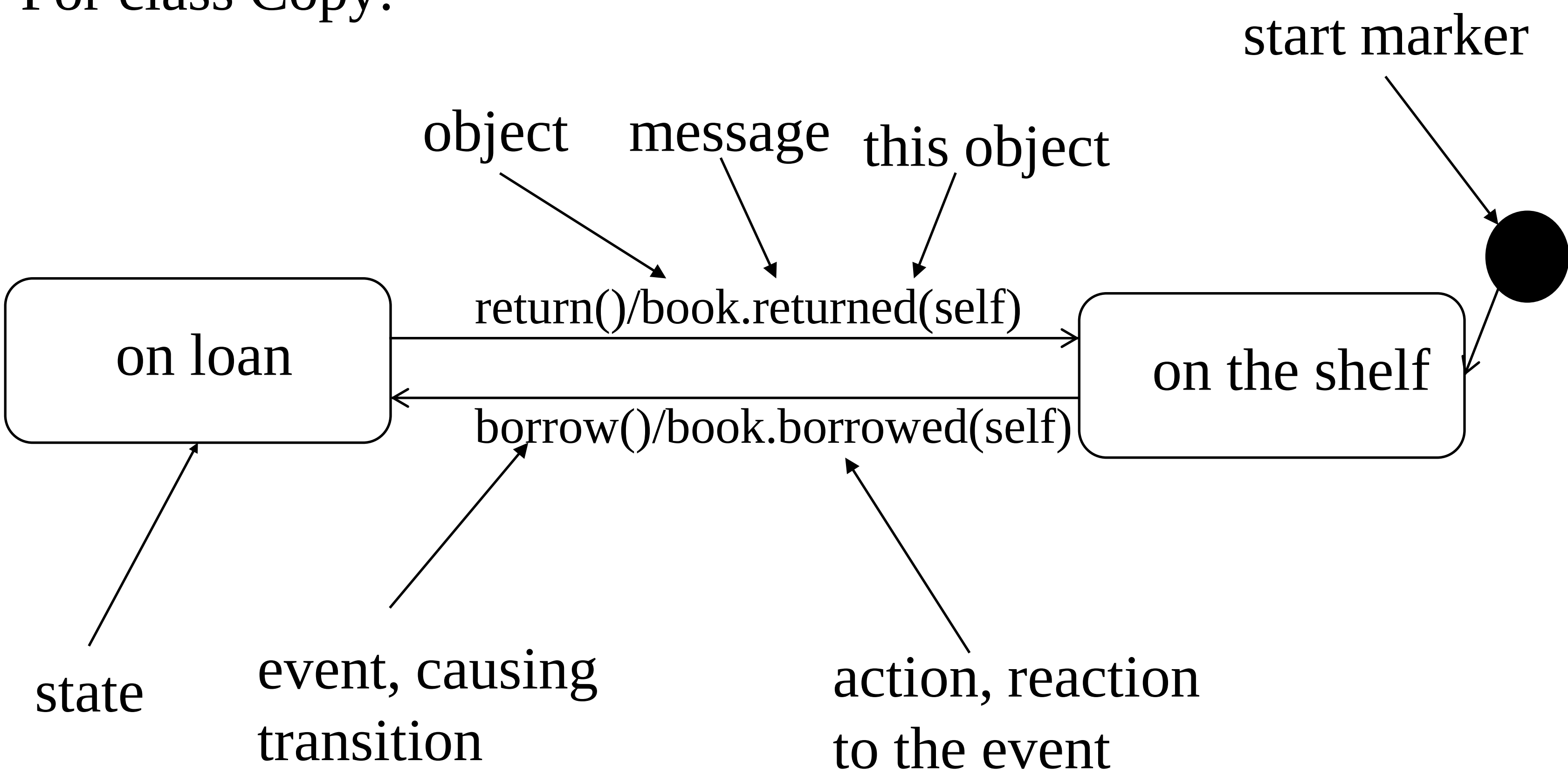


Continue modelling: next level



State diagram

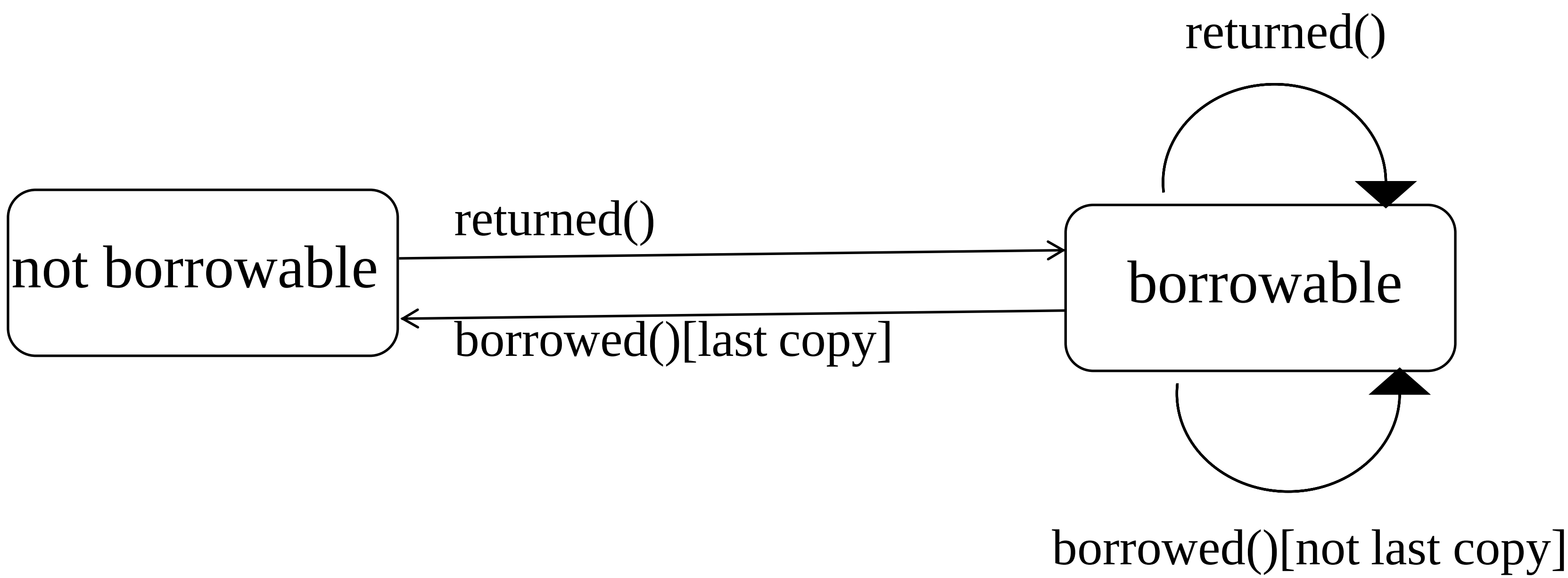
For class Copy:



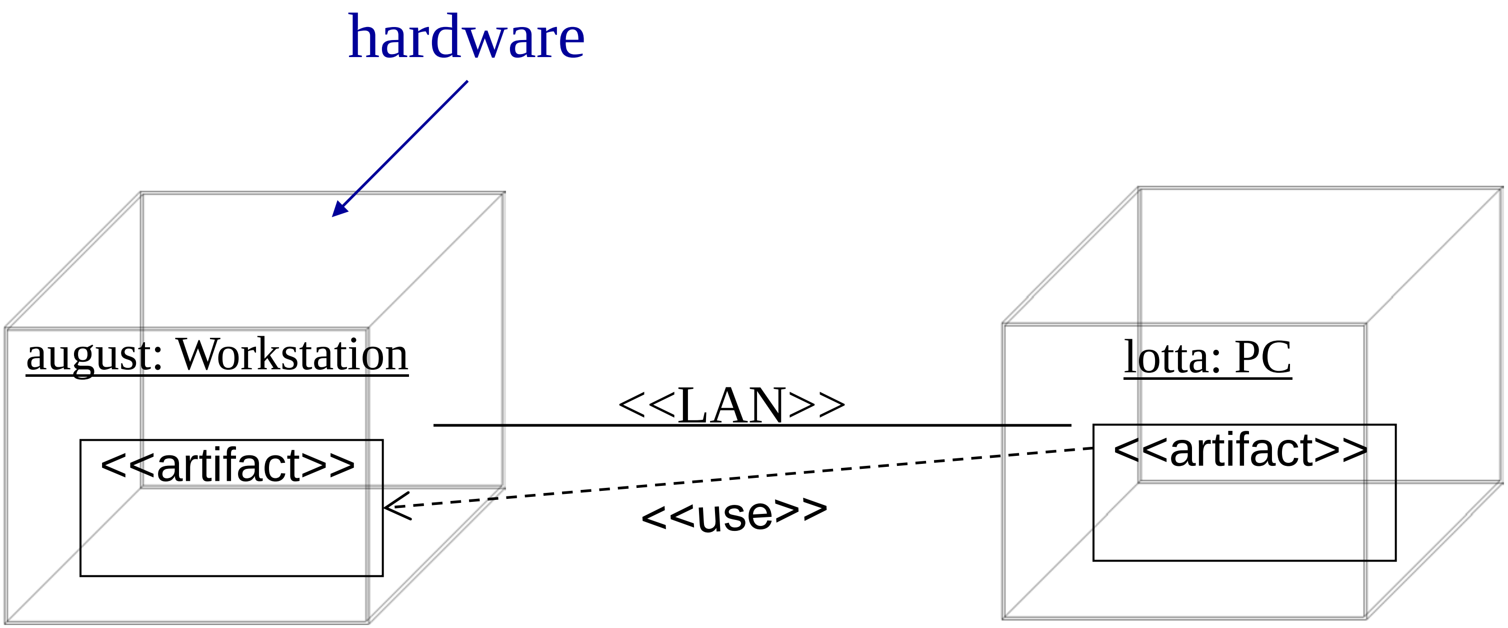
State diagram with guards

Wiht OCL, Object Constraint Language, this becomes very powerful

For class Book:

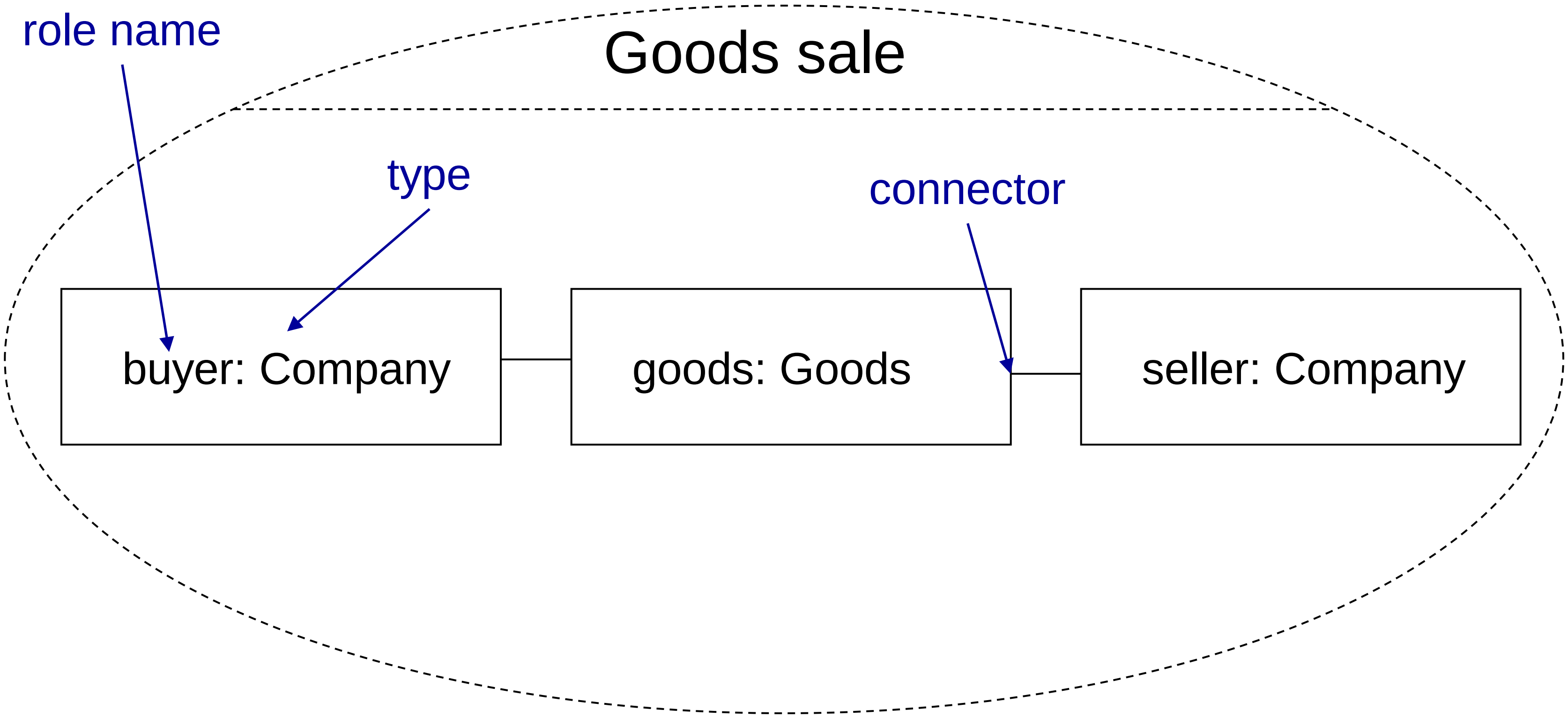


Deployment diagram

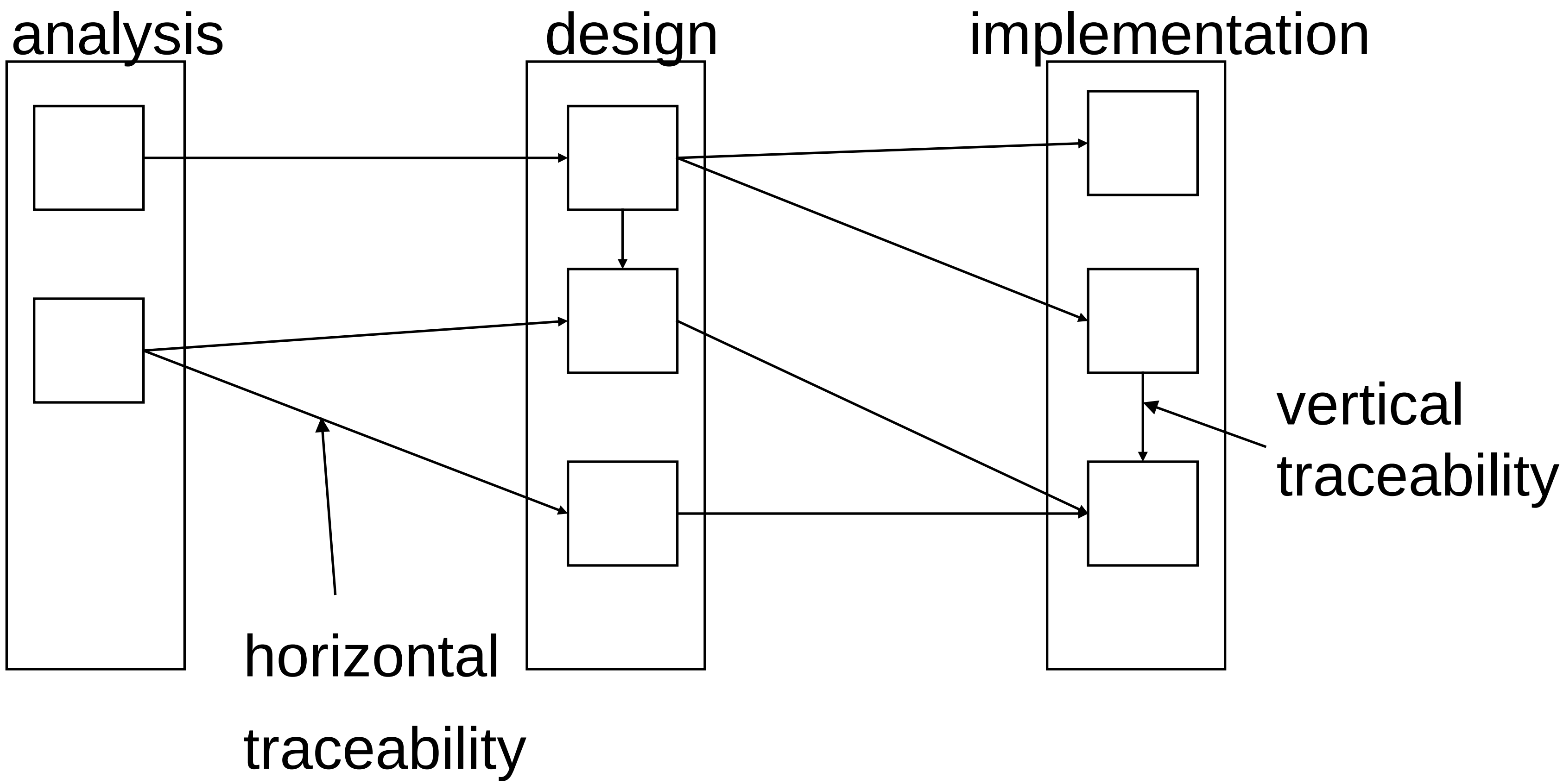


Collaboration

- Provides a focused view of how instances of classes may collaborate to achieve something, for example, a use-case



Traceability



Traceability methods

- Explicit links provided by a tool
- Textual references
- Name tracing using a pre-defined convention
- System knowledge and domain knowledge used by experienced people

Cross-referencing traceability

- R1: The system shall print all invoices at the department. (D1, D2, ...)
- D1: The system takes data from the customer data base and template A to print external invoices. (R1)
- D2: The system prompts the user for input and use template B for internal invoices. (R1)

The traceability matrix

	D1	D2	D3	D4	D5	D6	D7
R1	x			x			
R2		x		x			
R3			x				
R4					x	x	x
R5			x		x		
R6	x	x					x
R7							

Oops!

Benefits from good traceability

- Fulfilment of requirements can be assured
- Design rationale can be sought in affected requirements
- Change impact analysis forwards and backwards
- Cost estimations are made possible
- System understanding becomes easier
- Maintenance and testing are facilitated

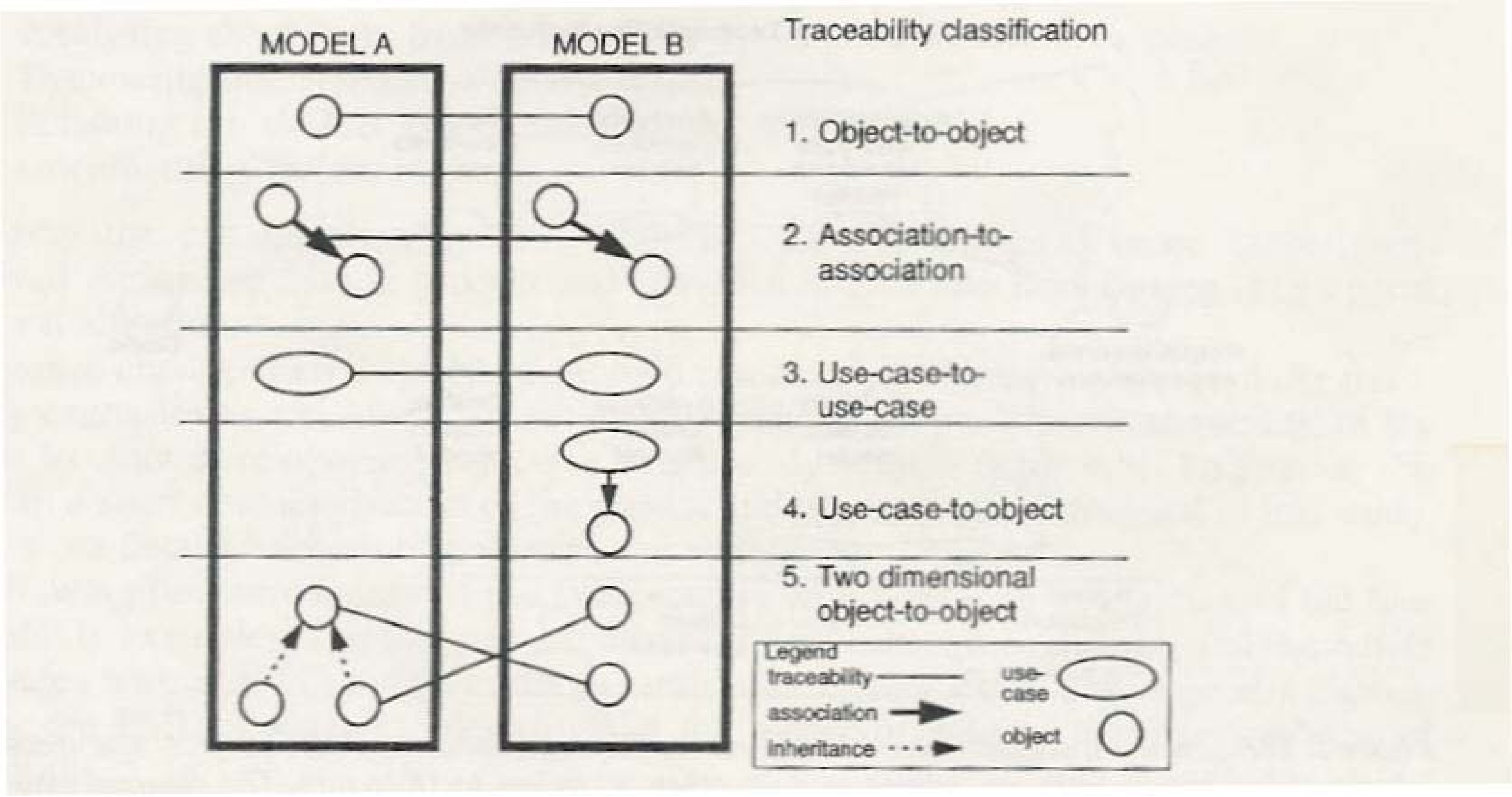
Troubles with traceability

- Hard to know what to trace
- Hard to maintain tracing information
- People don’t trust tracing information
- Hard to visualize traces
- It is thought of as an internal quality factor
- Is traceability item-wise even possible?

Practical investigation in traceability

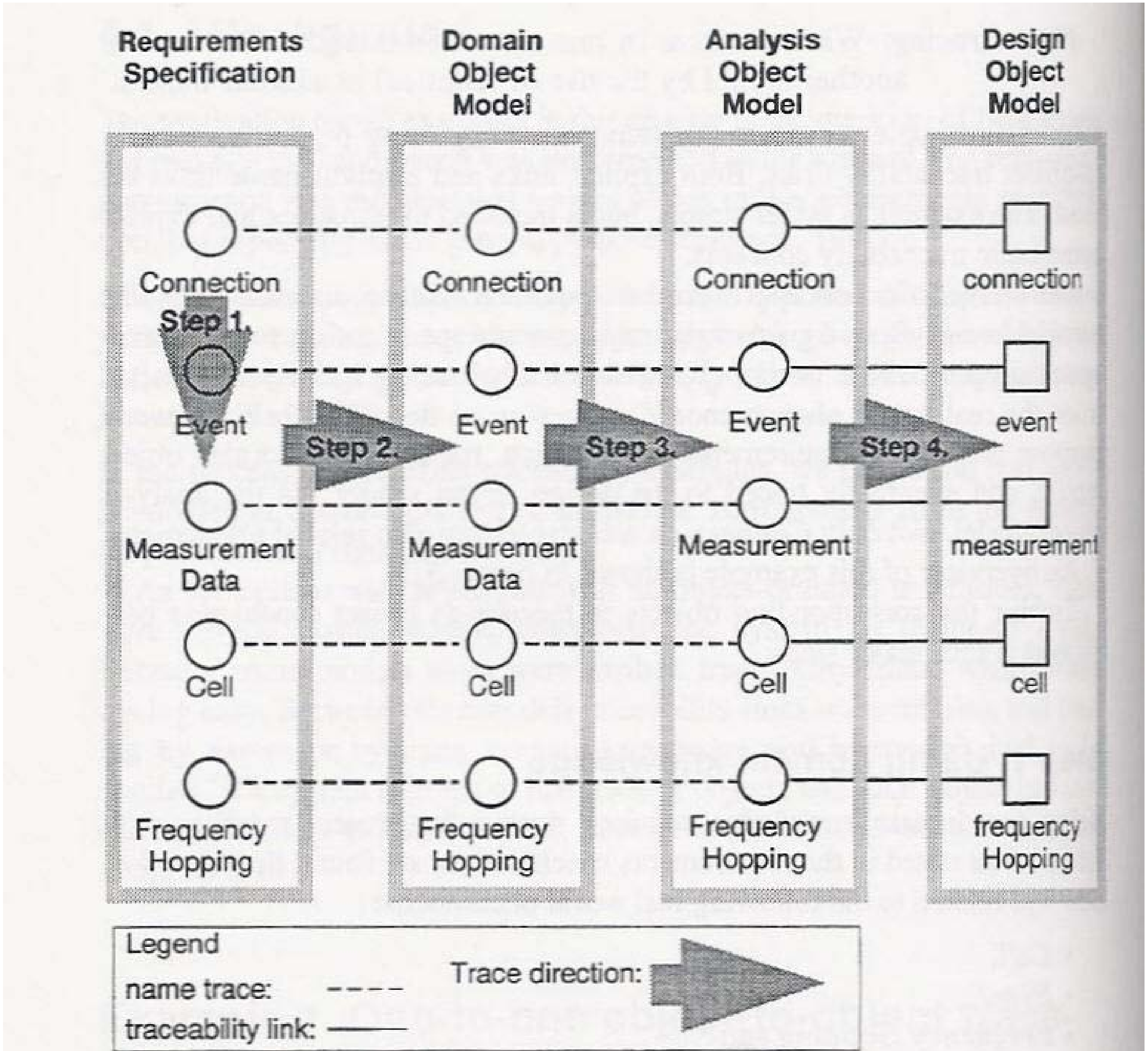
- From Lindvall and Sandahl: Practical Implications of Traceability, *Software – Practice and Experience*, 26(10), 1161-1180.
- Conducted at Ericsson's PMR project
- Example of successful project
- Method and tool: Forward engineering, Objectory SE (forerunner of UML and IBM Rational
- Updating of models was emphasised by the project leader

Types of traceability



Object-to-object traceability

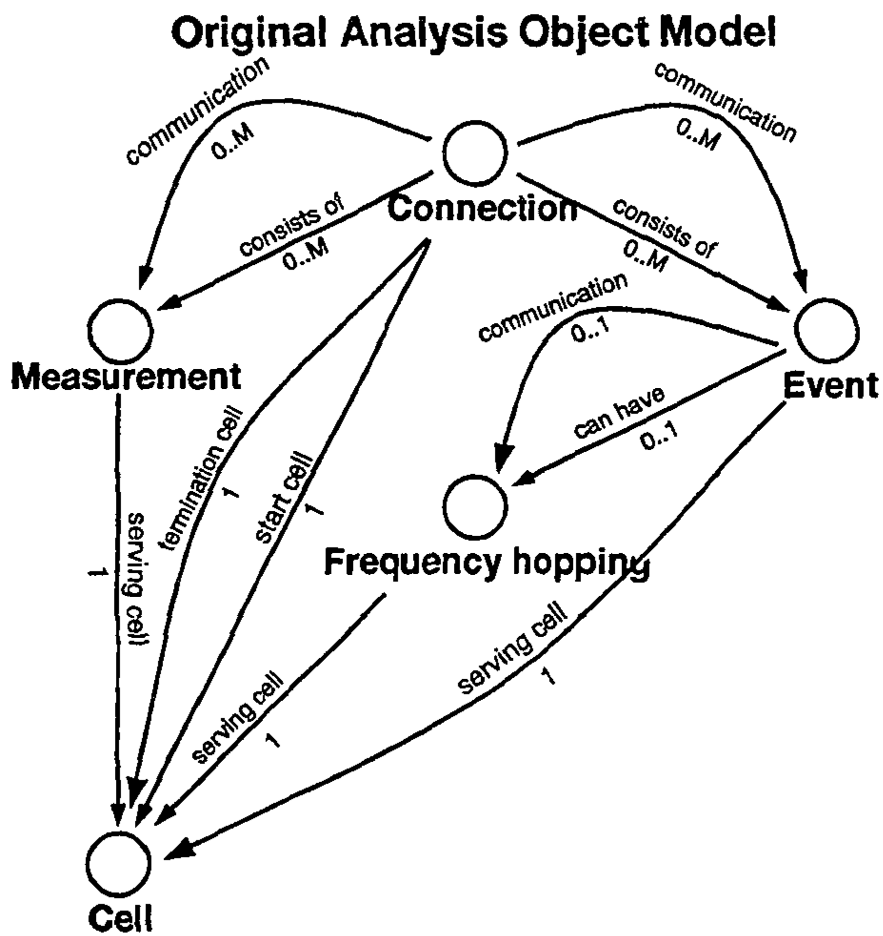
- Task: trace the concept *Connection* as described in the RS:
- ”The purpose is to provide a PMR operator with a presentation of the output from the recording in such a way that support is given for troubleshooting, verification of the radio network during one or several *Connections* for a specified MS”


Requirements Specification
Domain Object Model
Analysis Object Model
Design Object Model
Connection
Connection
Connection
connection
Step 1.
Event
Event
Event
event
Step 2.
Step 3.
Step 4.
Measurement Data
Measurement Data
Measurement
measurement
Cell
Cell
Cell
cell
Frequency Hopping
Frequency Hopping
Frequency Hopping
frequency Hopping
Legend
name trace: ----
traceability link: ——
Trace direction:

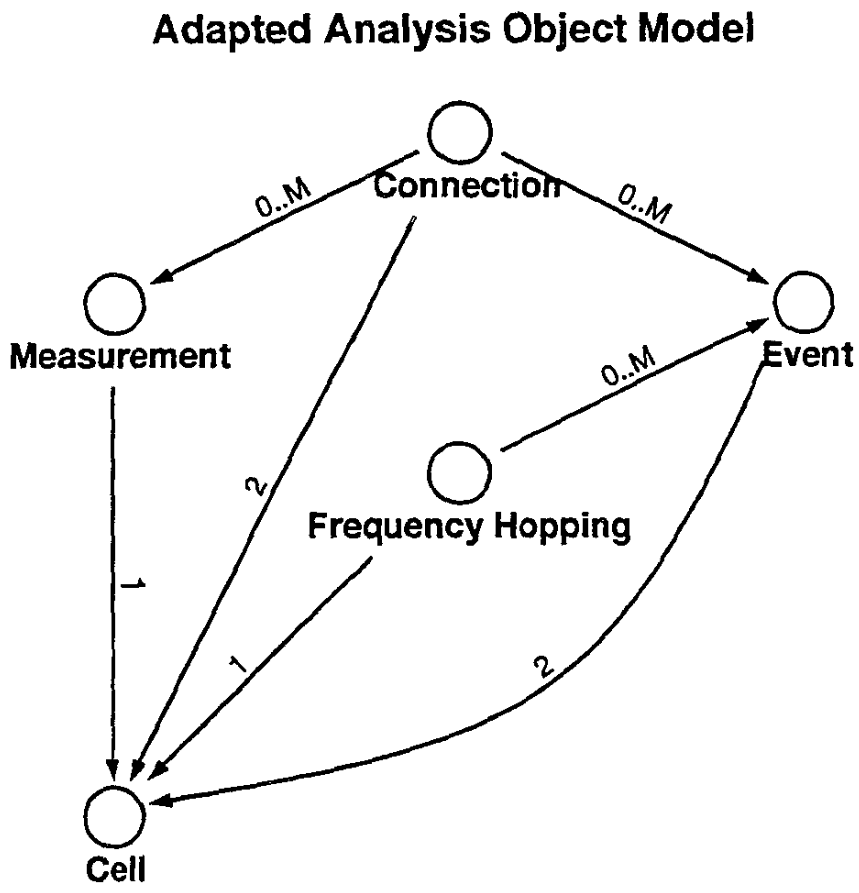
Association-to-association traceability

- Task: determine if there is a correspondence between associations of the objects

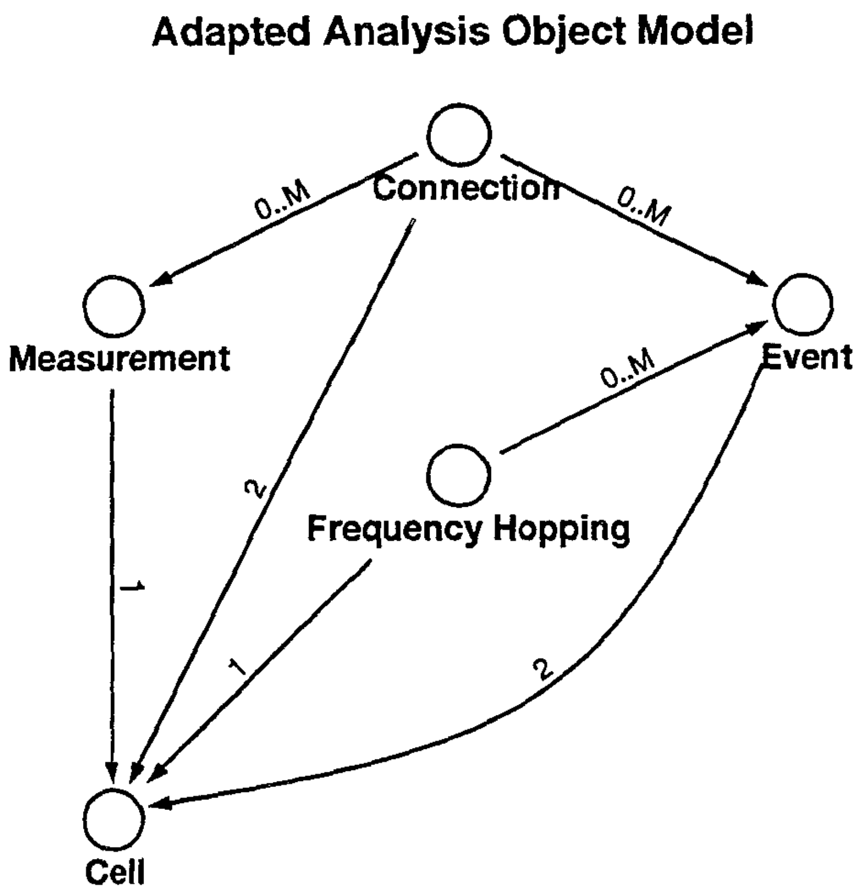
Original model

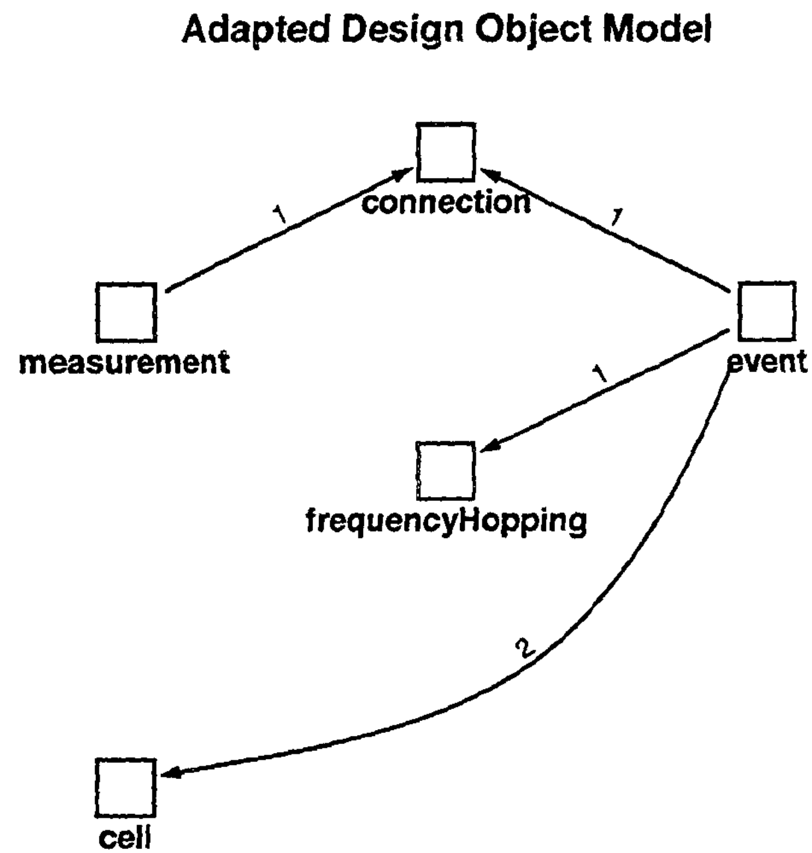


Correct and simplified model

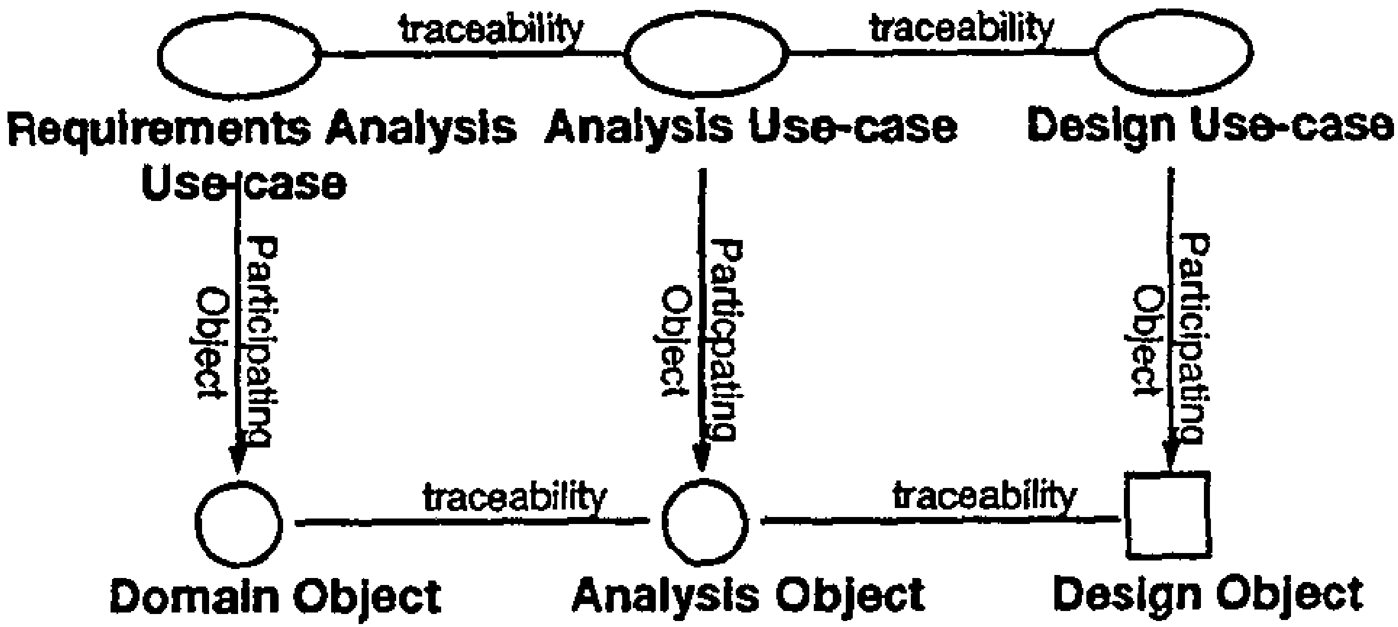


Are these the same models?



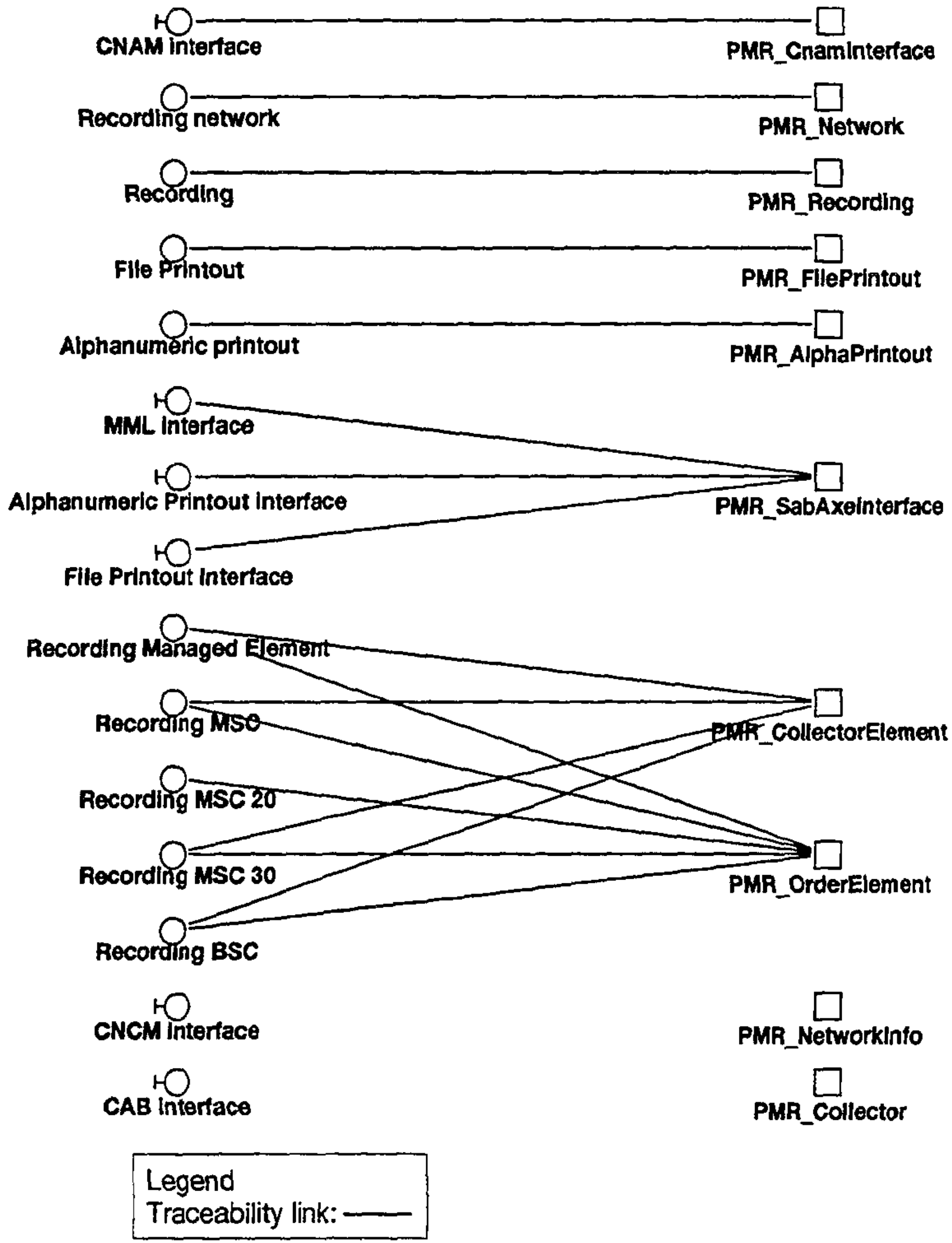


Use-case to object traceability

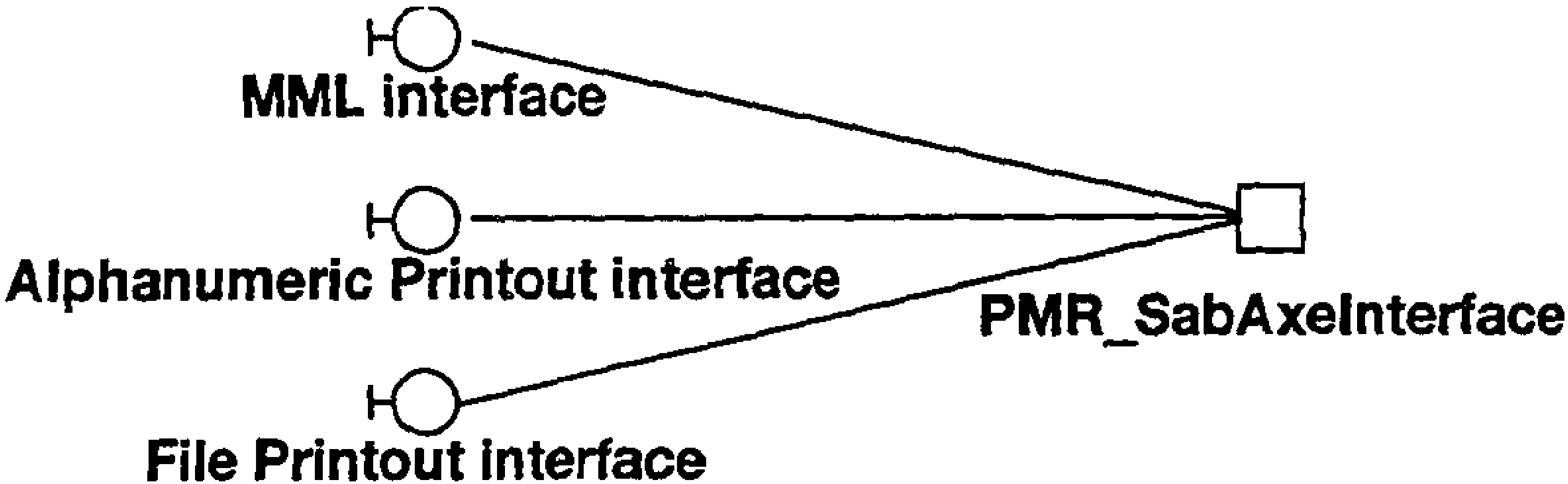


Use-case to object traceability

- Task: trace the requirement Recording Collection.
- Step 1: Find the use-case with name tracing
- Step 2: Trace to analysis objects
- Step 3: Trace to design objects via use-case
- Finally: Compare the object models


CNAM interface
PMR_CnamInterface
Recording network
PMR_Network
Recording
PMR_Recording
File Printout
PMR_FilePrintout
Alphanumeric printout
PMR_AlphaPrintout
MML interface
Alphanumeric Printout interface
PMR_SabAxeInterface
File Printout interface
Recording Managed Element
Recording MSC
PMR_CollectorElement
Recording MSC 20
Recording MSC 30
PMR_OrderElement
Recording BSC
CNCM interface
PMR_NetworkInfo
CAB interface
PMR_Collector
Legend
Traceability link:

Three-to-one traceability



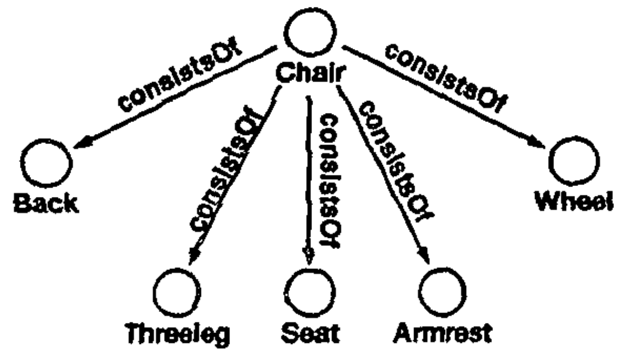


Figure 6.11: *A chair modeled as a direct correspondence to its physical realization in the real world.*

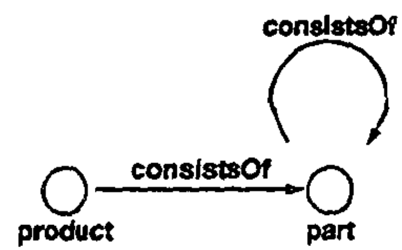
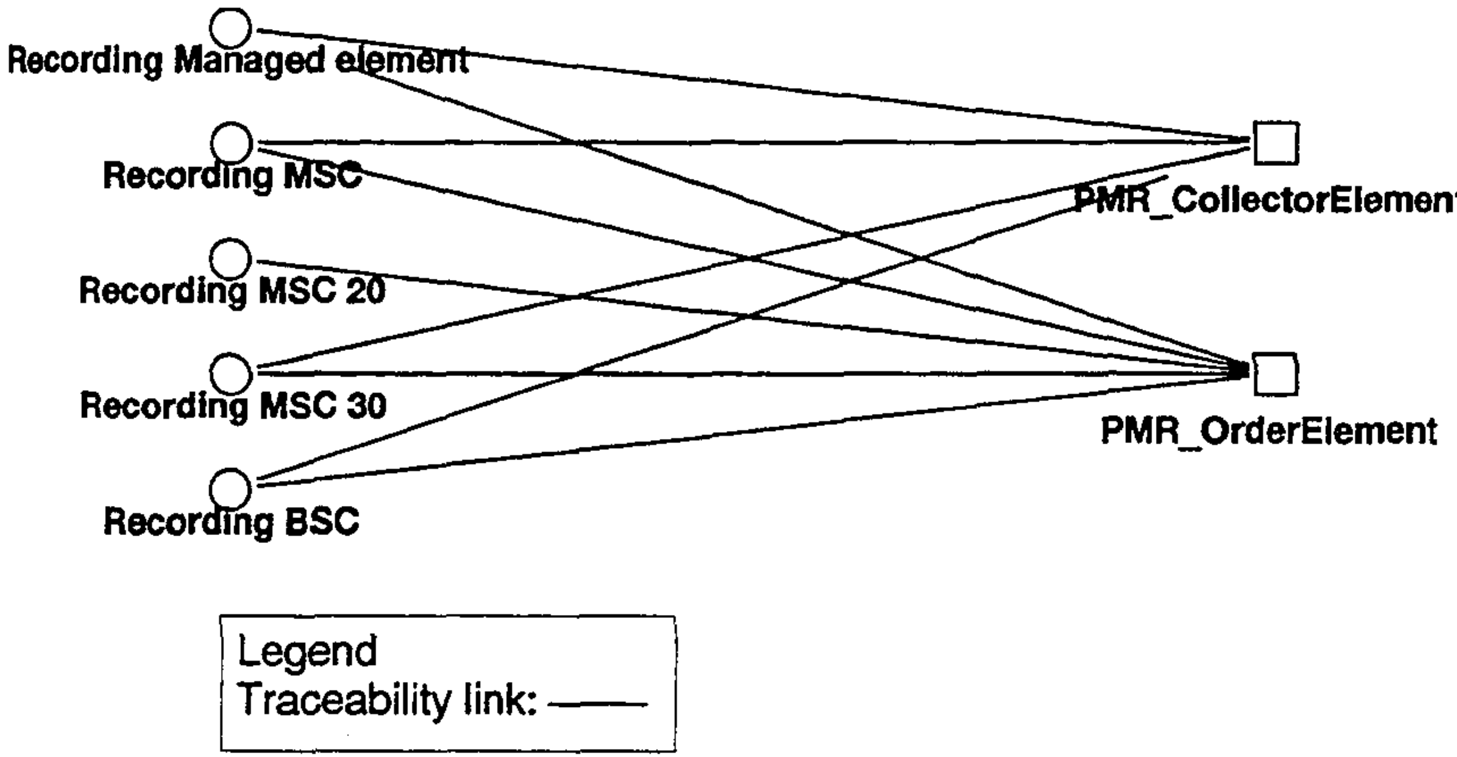
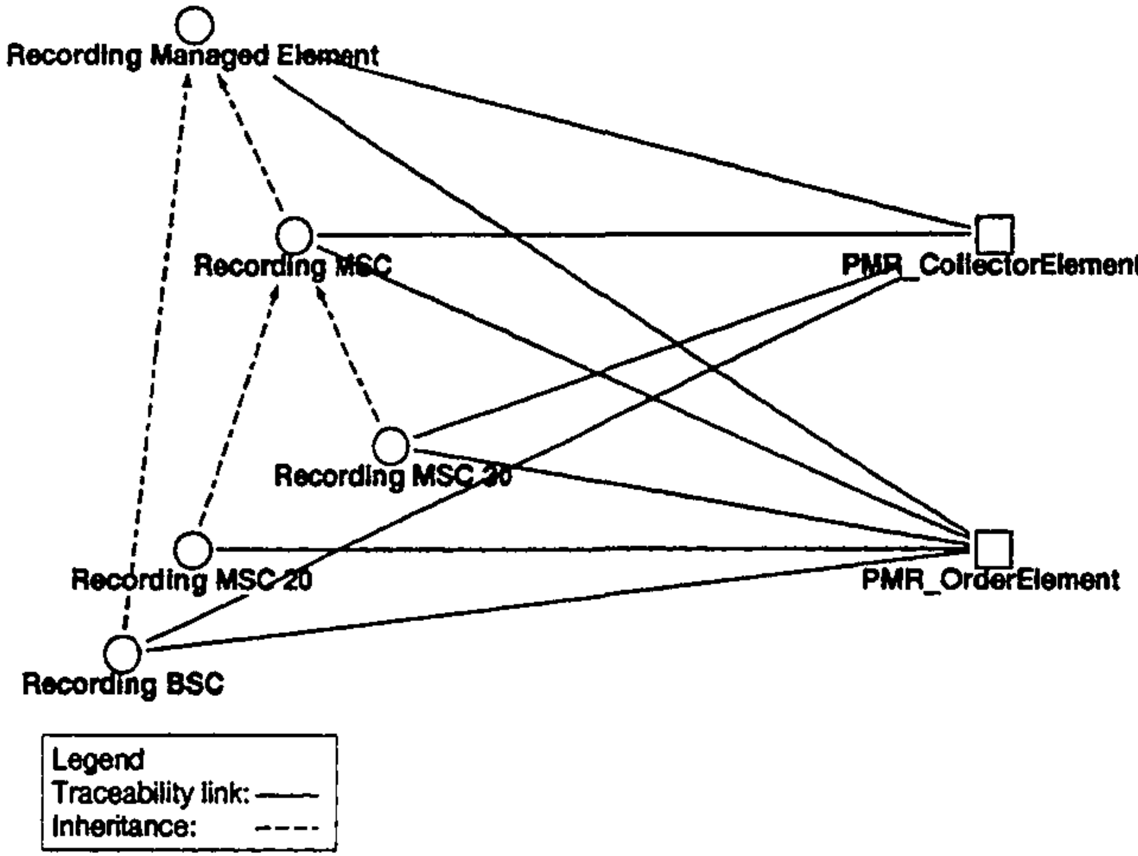


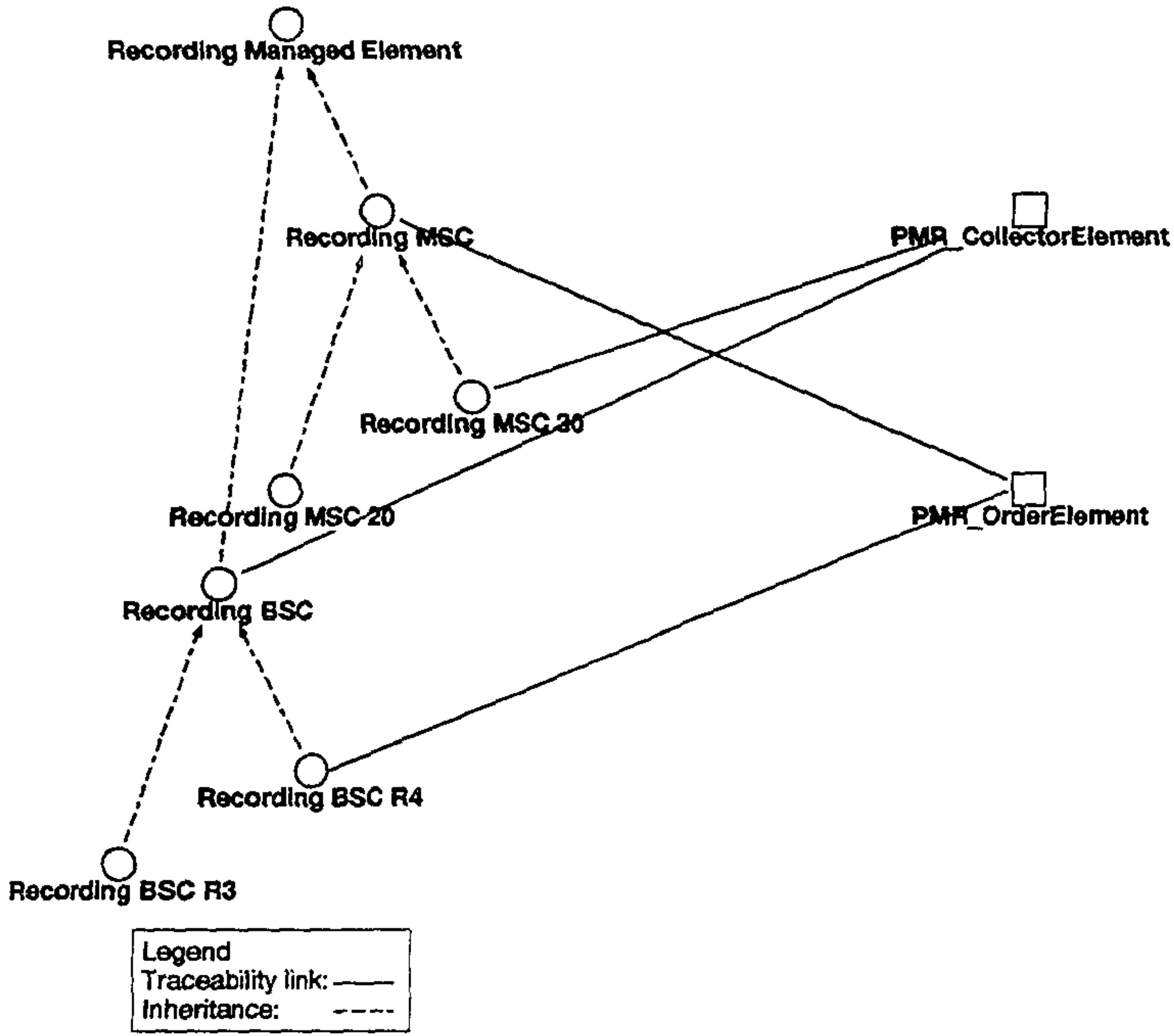
Figure 6.12: *A chair modeled as the role it plays in an information system:.a product consisting of parts.*

Many-to-many traceability

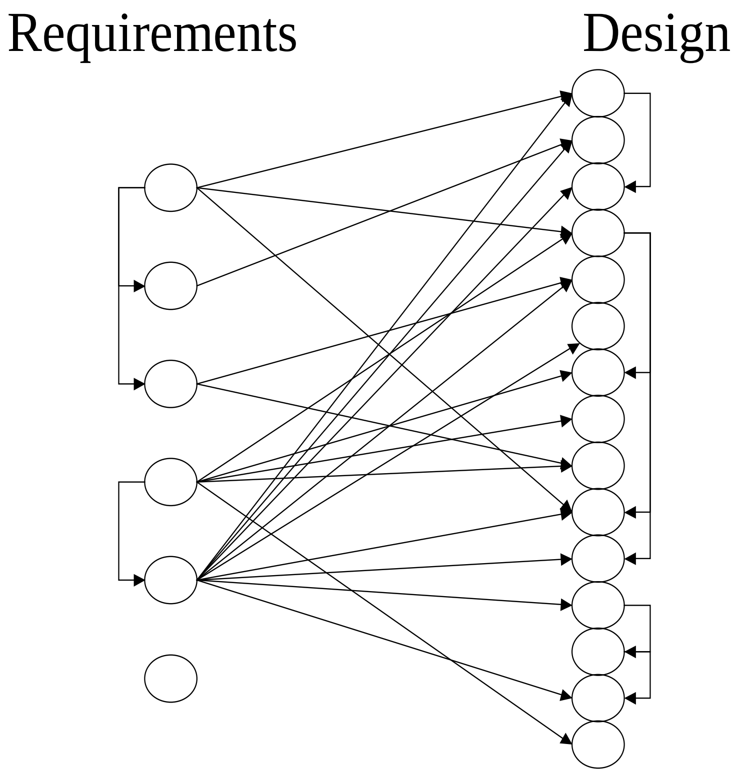


Two-dimensional traceability




Recording Managed Element
Recording MSC
PMR_CollectorElement
Recording MSC 30
Recording MSC 20
PMR_OrderElement
Recording BSC
Recording BSC R4
Recording BSC R3
Legend
Traceability link:
Inheritance:

A wicked visualisation problem



Matrix browser

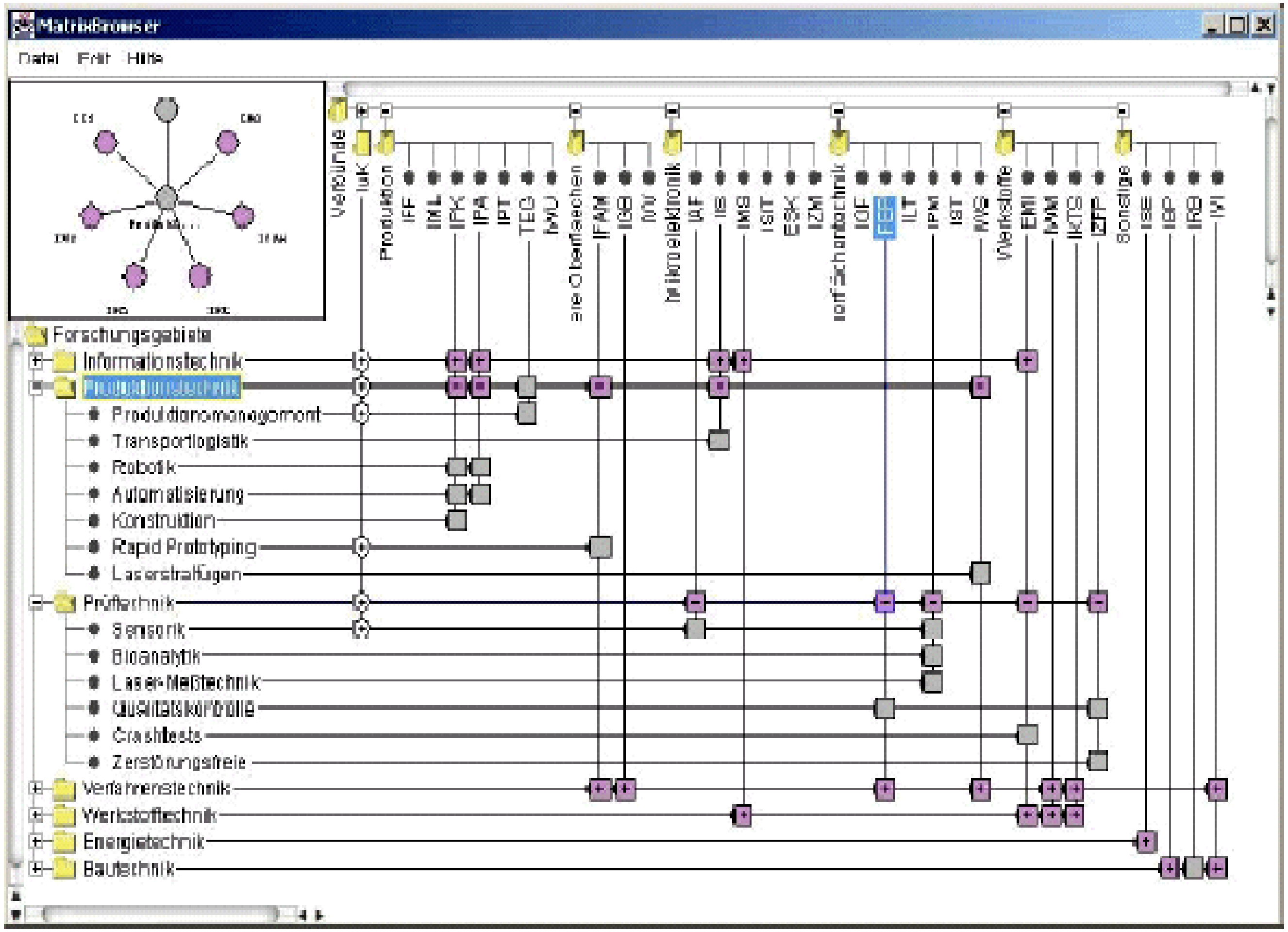
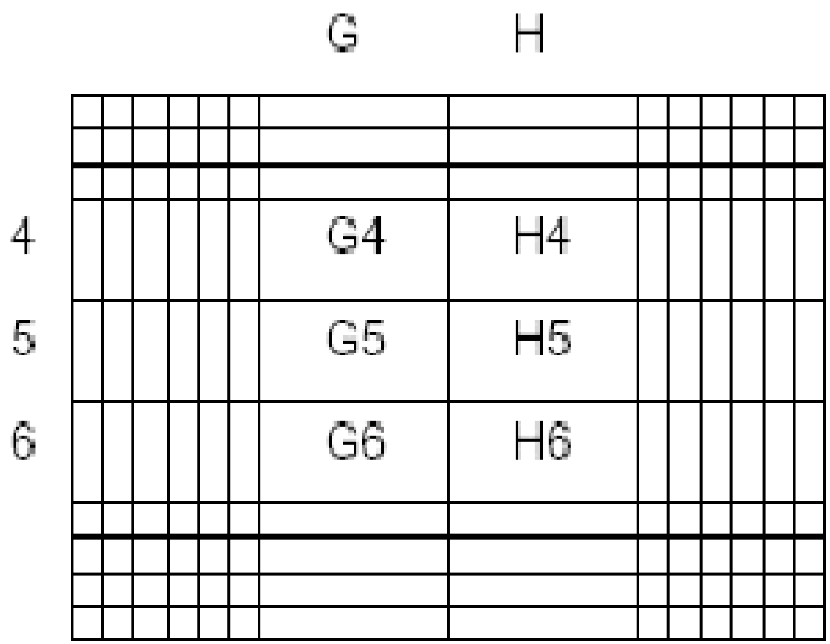


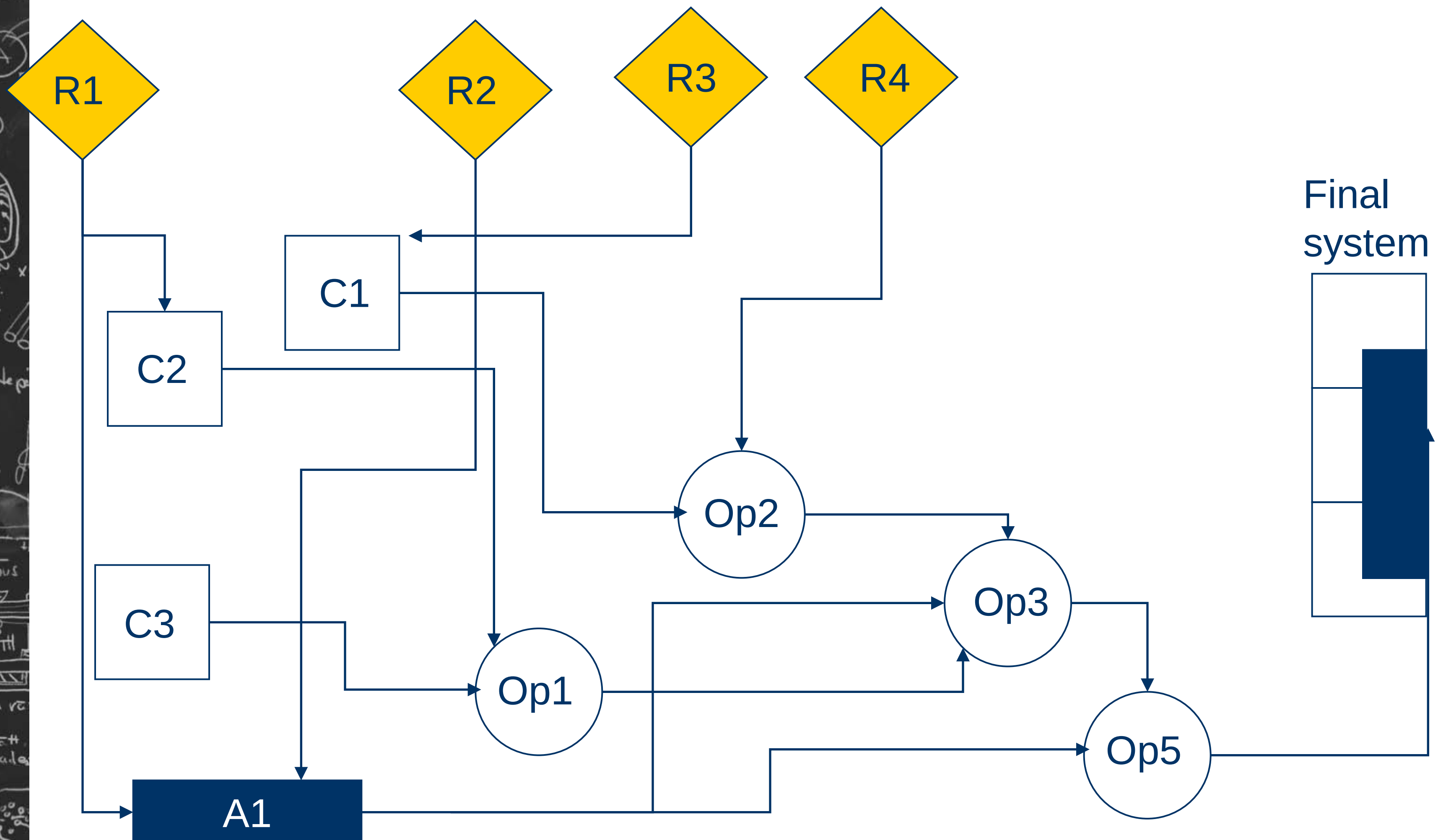
Table lens



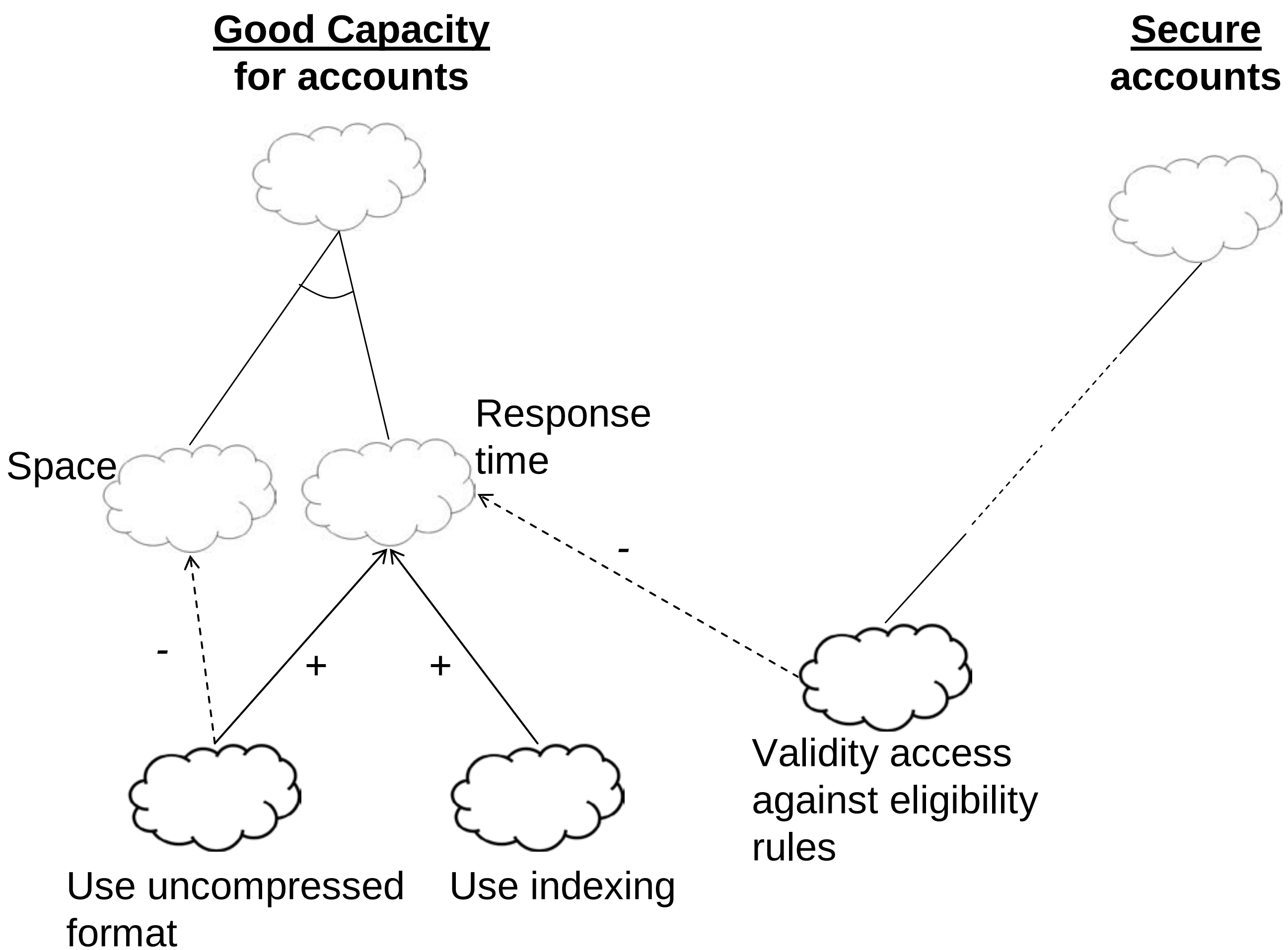
Conclusions

- Traceability in model-based development is possible and boosts system understanding and correctness
- In practice many different methods are used simultaneously
- You need to determine what is important to trace
- Sometimes you can get traceability for free
- To take full advantage you need to invest and handle the attitudes

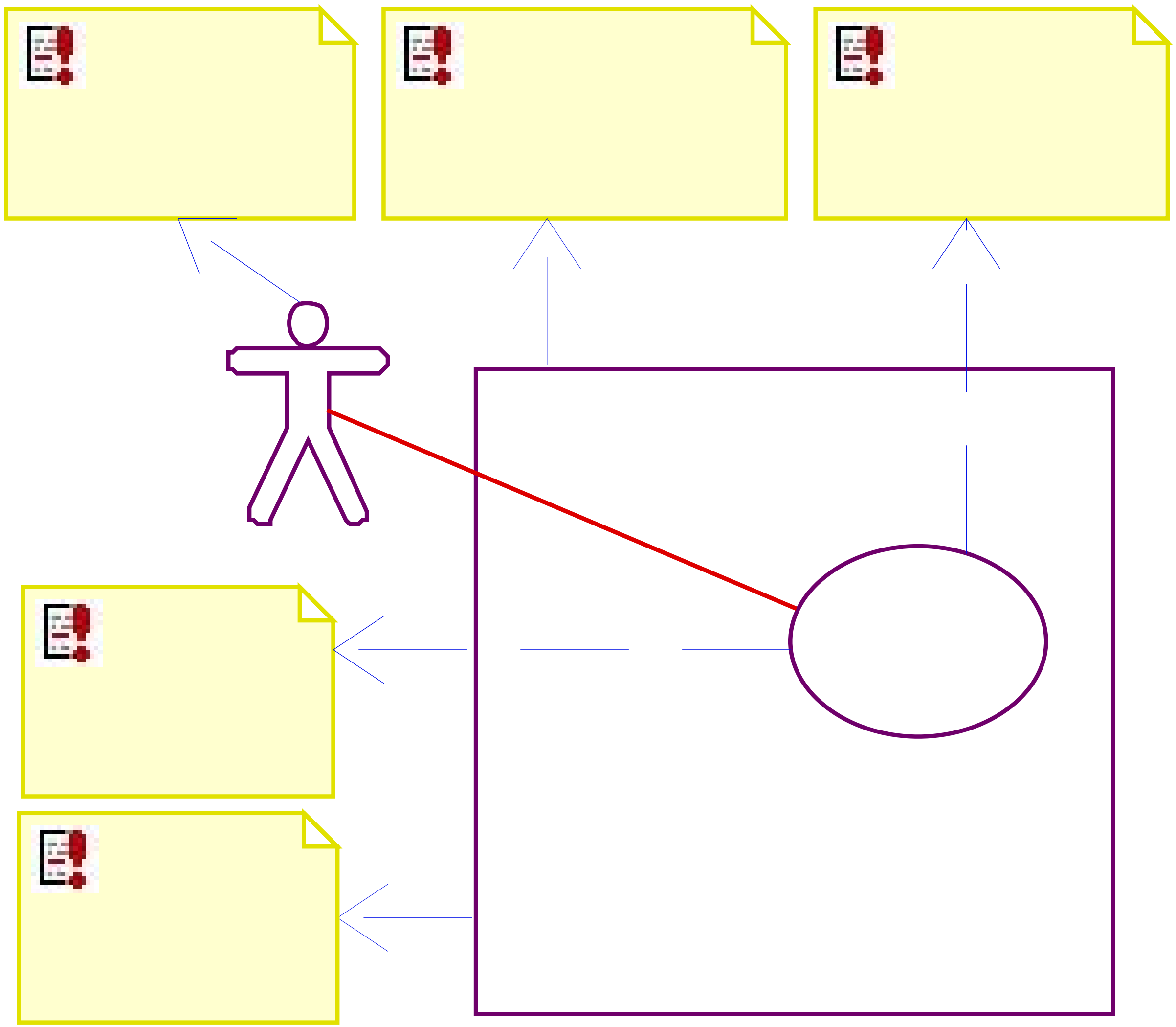
Future: Integrational Software Engineering



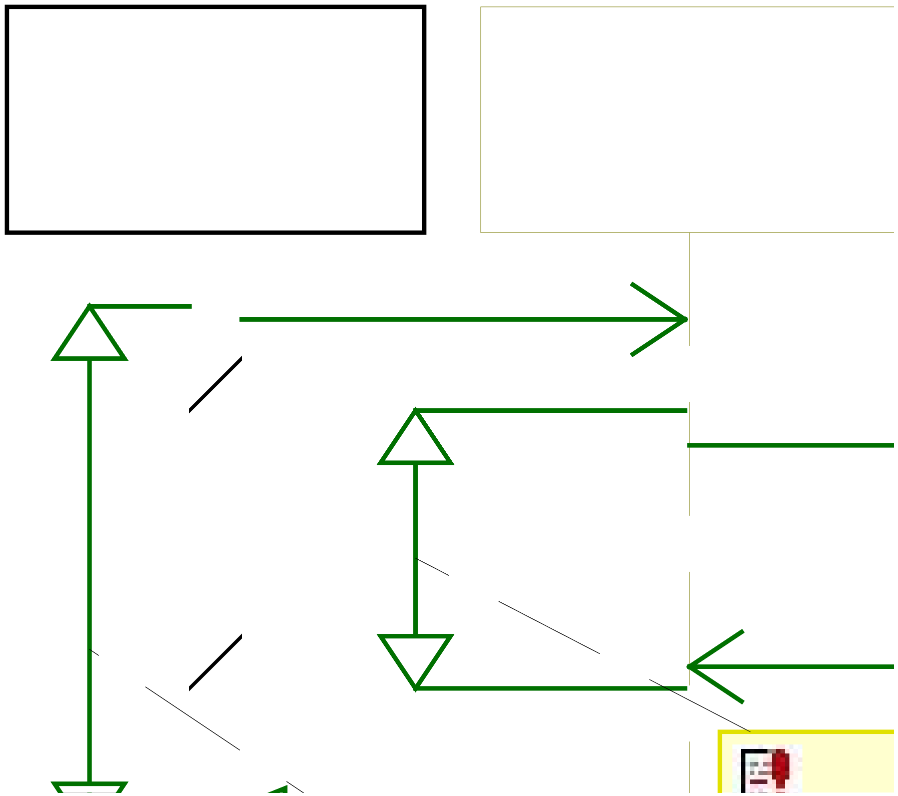
The NFR Framework



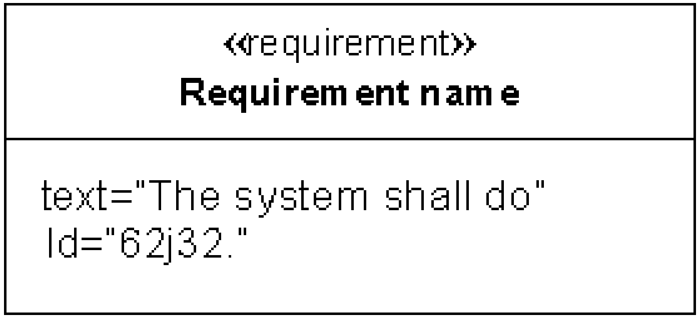
Annotating UML models



Time constraints in a sequence diagram



Requirements in SysML



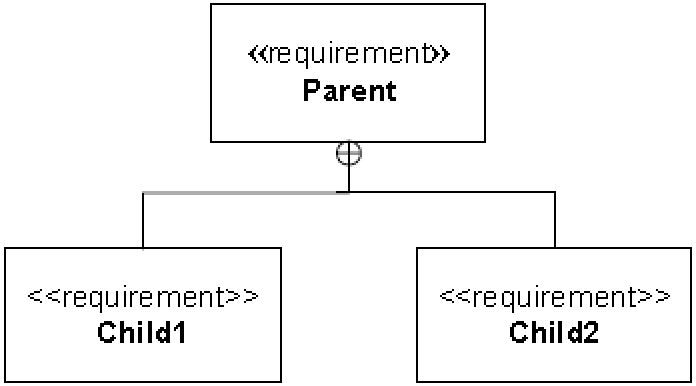
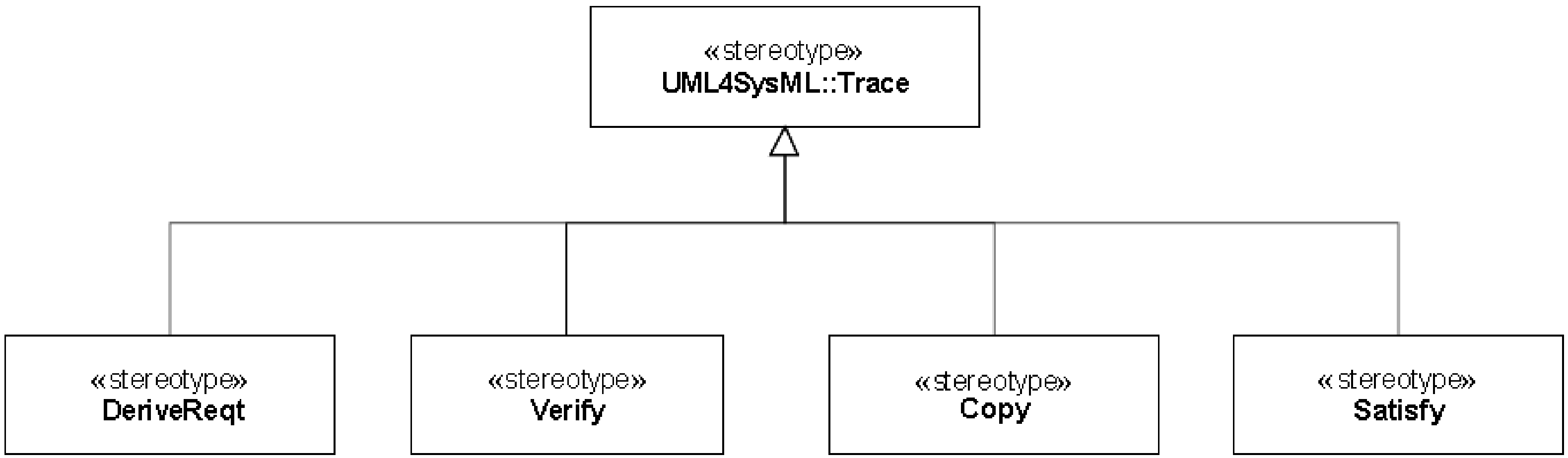


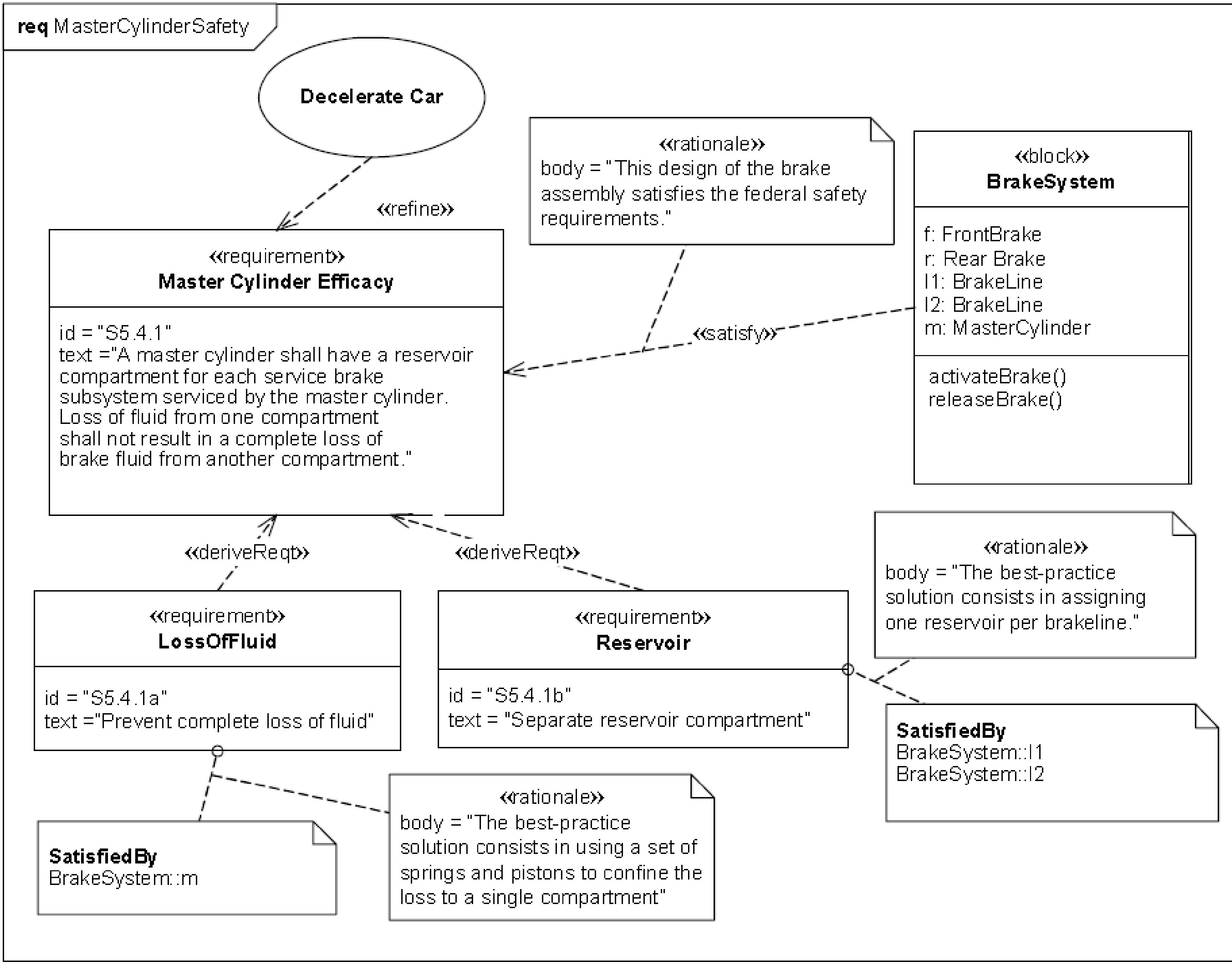
Table representation

table [requirement] Performance [Decomposition of Performance Requirement]

id	name	text
2	Performance	The Hybrid SUV shall have the braking, acceleration, and off-road capability of a typical SUV, but have dramatically better fuel economy.
2.1	Braking	The Hybrid SUV shall have the braking capability of a typical SUV.
2.2	FuelEconomy	The Hybrid SUV shall have dramatically better fuel economy than a typical SUV.
2.3	OffRoadCapability	The Hybrid SUV shall have the off-road capability of a typical SUV.
2.4	Acceleration	The Hybrid SUV shall have the acceleration of a typical SUV.

Relations


«stereotype»
UML4SysML::Trace
«stereotype»
DeriveReqt
«stereotype»
Verify
«stereotype»
Copy
«stereotype»
Satisfy


req MasterCylinderSafety
Decelerate Car
«refine»
«rationale»
body = "This design of the brake assembly satisfies the federal safety requirements."
«block»
BrakeSystem
f: FrontBrake
r: Rear Brake
l1: BrakeLine
l2: BrakeLine
m: MasterCylinder
activateBrake()
releaseBrake()
«requirement»
Master Cylinder Efficacy
id = "S5.4.1"
text ="A master cylinder shall have a reservoir compartment for each service brake subsystem serviced by the master cylinder. Loss of fluid from one compartment shall not result in a complete loss of brake fluid from another compartment."
«satisfy»
«deriveReqt»
«deriveReqt»
«rationale»
body = "The best-practice solution consists in assigning one reservoir per brakeline."
«requirement»
LossOfFluid
id = "S5.4.1a"
text ="Prevent complete loss of fluid"
«requirement»
Reservoir
id = "S5.4.1b"
text = "Separate reservoir compartment"
SatisfiedBy
BrakeSystem::l1
BrakeSystem::l2
SatisfiedBy
BrakeSystem::m
«rationale»
body = "The best-practice solution consists in using a set of springs and pistons to confine the loss to a single compartment"

Formal methods

- Just as models, formal methods is a **complement** to other specification methods.
- Standard is model-based methods, specified mathematically and interpreted with logic.
- Benefits: Non-ambiguous specification, all issues are discovered, proof of properties, simulation, code generation.
- Costs: Time, tools, training and inherent complexity of algorithms.
- High costs ⇒ use only for critical applications

The three Cs - definition

- Consistency – no internal contradictions
- Completeness – everything is there
- Correctness – satisfaction of business goals

Potential problems:

- adding requirements make the specification more complete, but there is a risk of introducing contradiction.
- correctness is vaguely defined,
  formally: consistent + complete?
  pragmatically: satisfaction of customer needs?

Single specification model

Requirements

Specification

states relationships between elements of

Domain

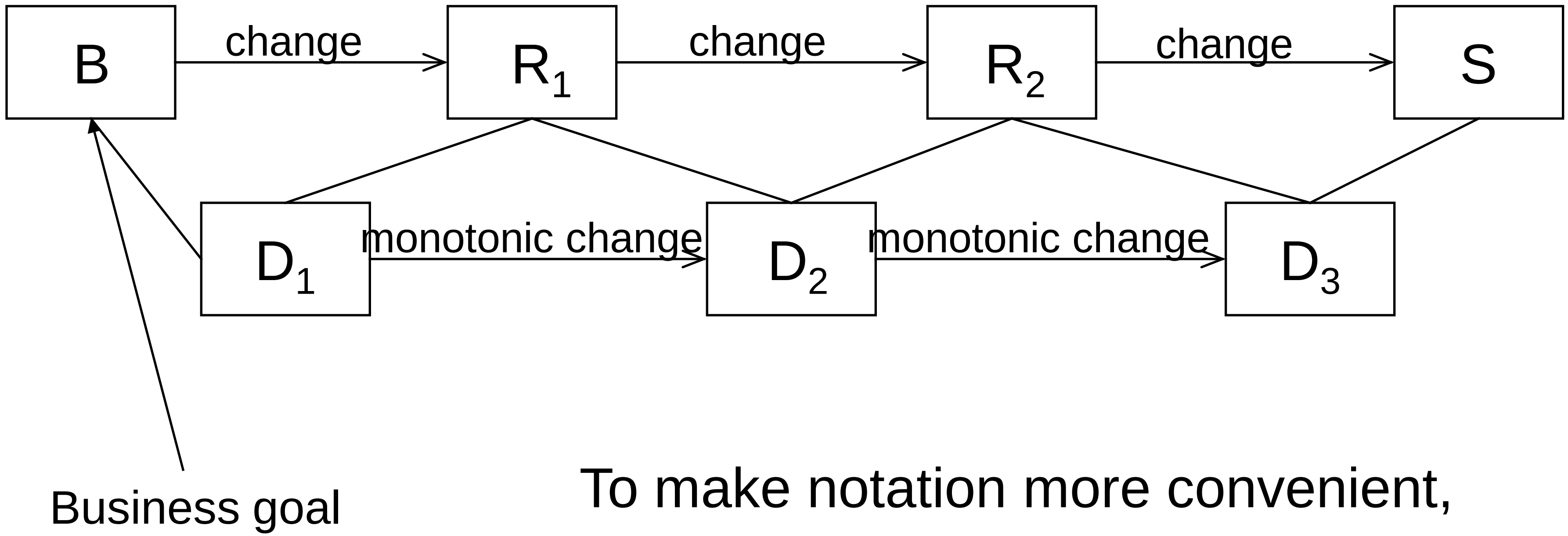
provides an interface to

$S \cup D \models R$

What we know about the domain, system and interfaces makes R true. Nothing in R is missing in S and D

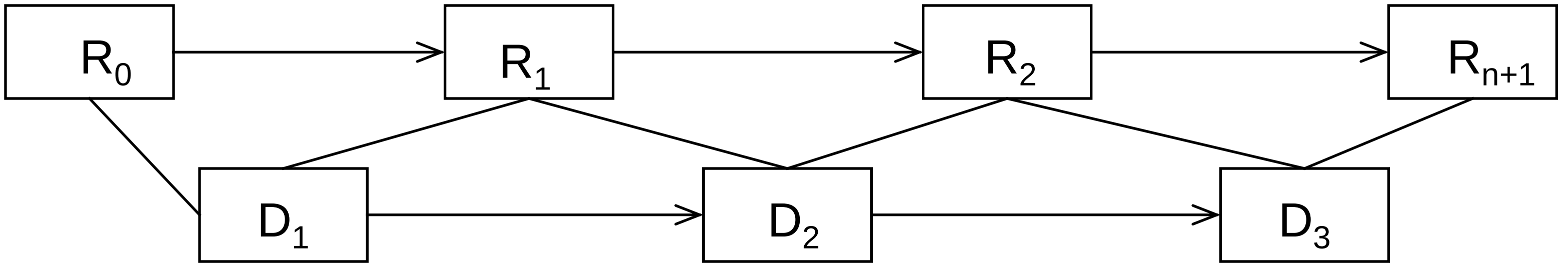
$S \cup D$ is consistent $\Rightarrow$ mission of S is possible $\wedge$ Tells if S is complete with respect to R $\Rightarrow$ Proof obligation towards correctness of S,or formal proof of correctness?

Evolutionary model



To make notation more convenient,
let $B = R_0$
and $S = R_{n+1}$

The three Cs



$R_i \cup D_i \vDash R_{i-1}$
(completeness)

$R_i \cup D_i \nvDash \perp$ (consistency)

$D_i \vDash D_{i-1}$ (monotonicity) $\Rightarrow$
$R_i \cup D_i \vDash R_{i-1} \cup D_{i-1}$

Induction gives:

$R_{n+1} \cup D_{n+1} \vDash R_0 \cup \{\ \}$

Replace back and have:

$S \cup D_{n+1} \vDash B$

Specification deployed in final domain satisfies customer needs = correctness

Example: shop owner(1)

- B = {when a customer comes near the entrance, the door shall open}

First attempt:

- D_1 = {when a person comes near the entrance door, a presence sensor gets activated}
- R_1 = {when the sensor gets activated, the door shall open}
- Prove $R_1 \cup D_1 \models B$, and fail, since B talks about customers, D_1 talks about persons
- Two choices: Improve D_1 with biometry and recognition or weaken B:
- B = {when a person comes near the entrance, the door shall open}
- Prove $R_1 \cup D_1 \models B$ and succeed (consistent, complete)

Example: shop owner (2)

Second iteration:

- $D_2 = D_1 \cup$ {when a sliding door's motor is turned on, the door opens}
- R_2 = {when the sensor gets activated, the door's motor shall be turned on}
- $R_2 \cup D_2$ is consistent and complete w.r.t R_1
- $D_2 \vDash D_1$ (containment)
- $R_2 \nvDash R_1$ (knowledge about whether motor(on) $\Rightarrow$ door(opened) is the the domain theory, not in Rs)

Continued development:

- S = {when a signal is detected on the input line associated with the door's presence sensor, establish +5V on the output line associated with the door's motor}
- If we have proved consistency and completeness in all iterations, S is correct w.r.t B

Z example

$ST = Key \mapsto VAL$

INIT

$st' : ST$

$st' = \{\}$

INSERT

$st, st' : ST$

$k : KEY$

$v : VAL$

$k \notin dom(st) \land$

$st' = st \cup \{k \mapsto v\}$

LOOKUP

$st, st' : ST$

$k : KEY$

$v : VAL$

$k \in dom(st) \land$

$v' = st(k) \land$

$st' = st$

DELETE

$st, st' : ST$

$k : KEY$

$k \in dom(st) \land$

$st' = \{k\} ⩤ st$