Modeling

Main issues:

- What do we want to build
- How do we write this down

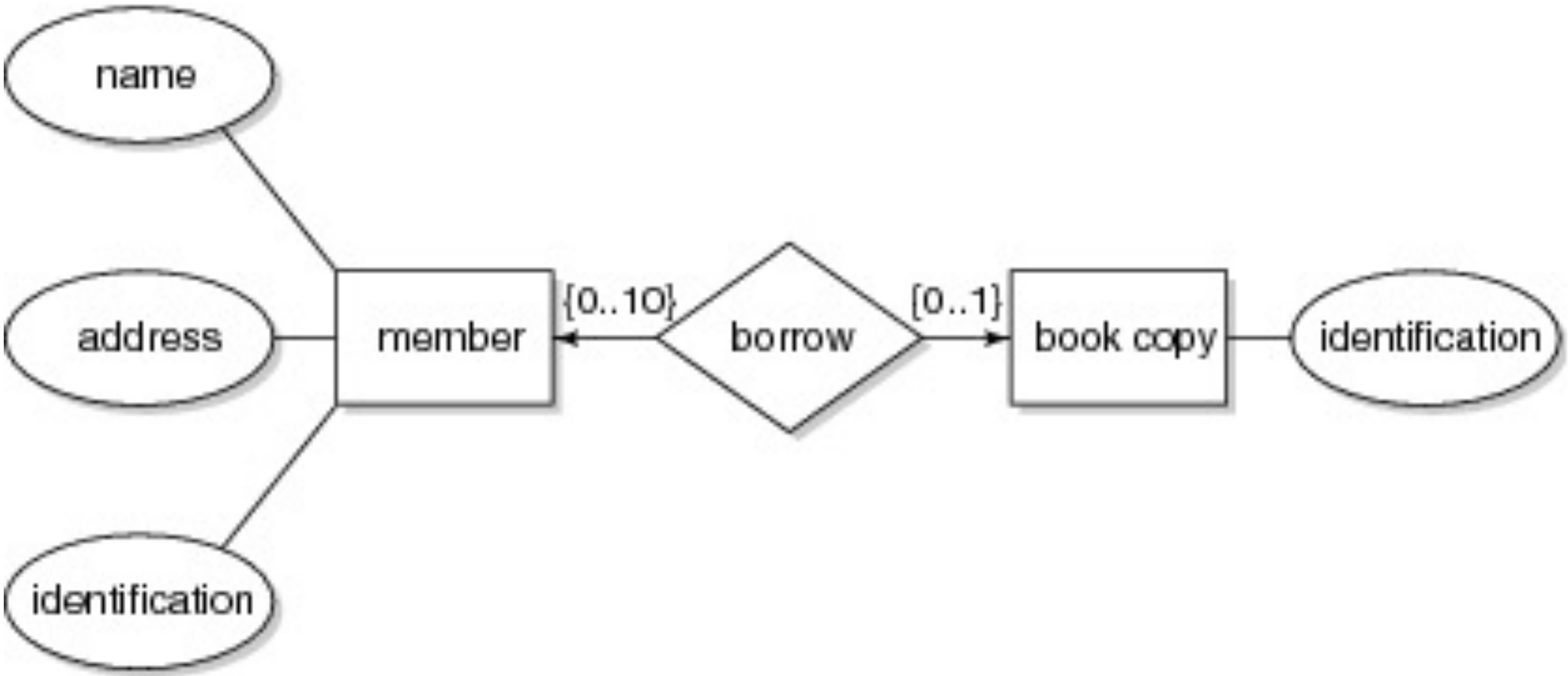
System Modeling Techniques

- Classic modeling techniques:
  - Entity-relationship modeling
  - Finite state machines
  - Data flow diagrams
  - CRC cards
- Object-oriented modeling: variety of UML diagrams

Entity-Relationship Modeling

- entity: distinguishable object of some type
- entity type: type of a set of entities
- attribute value: piece of information (partially) describing an entity
- attribute: type of a set of attribute values
- relationship: association between two or more entities

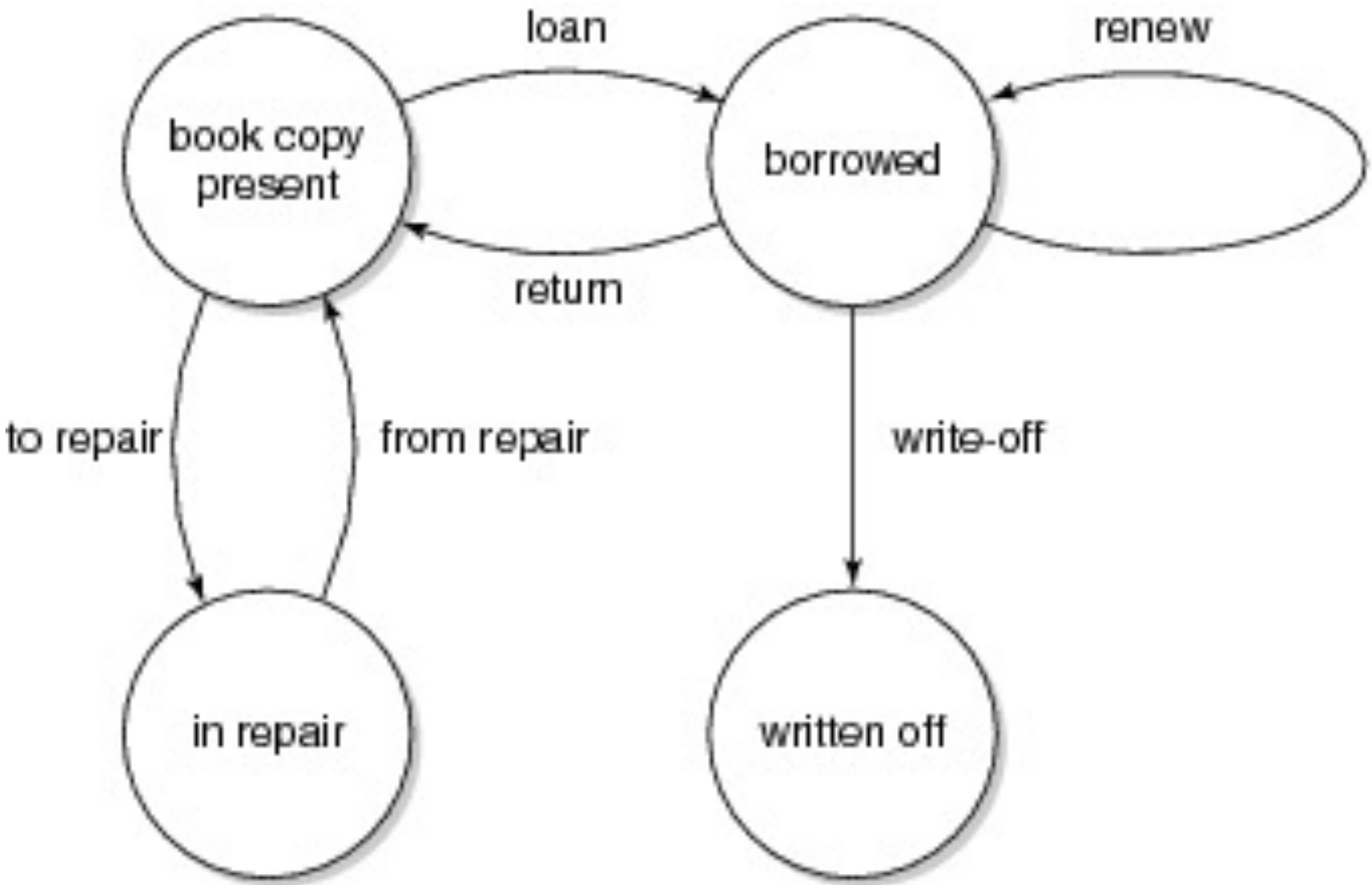
Example ER-diagram



Finite state machines

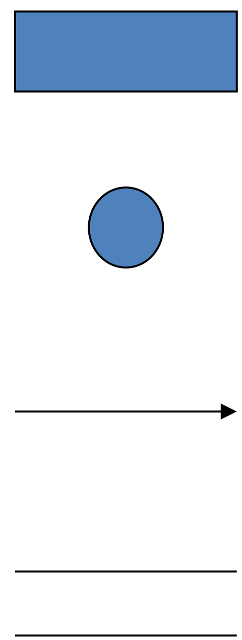
- Models a system in terms of (a finite number of) states, and transitions between those states
- Often depicted as state transition diagrams:
  - Each state is a bubble
  - Each transition is a labeled arc from one state to another
- Large system $\Rightarrow$ large diagram hierarchical diagrams: statecharts$\Rightarrow$

Example state transition diagram

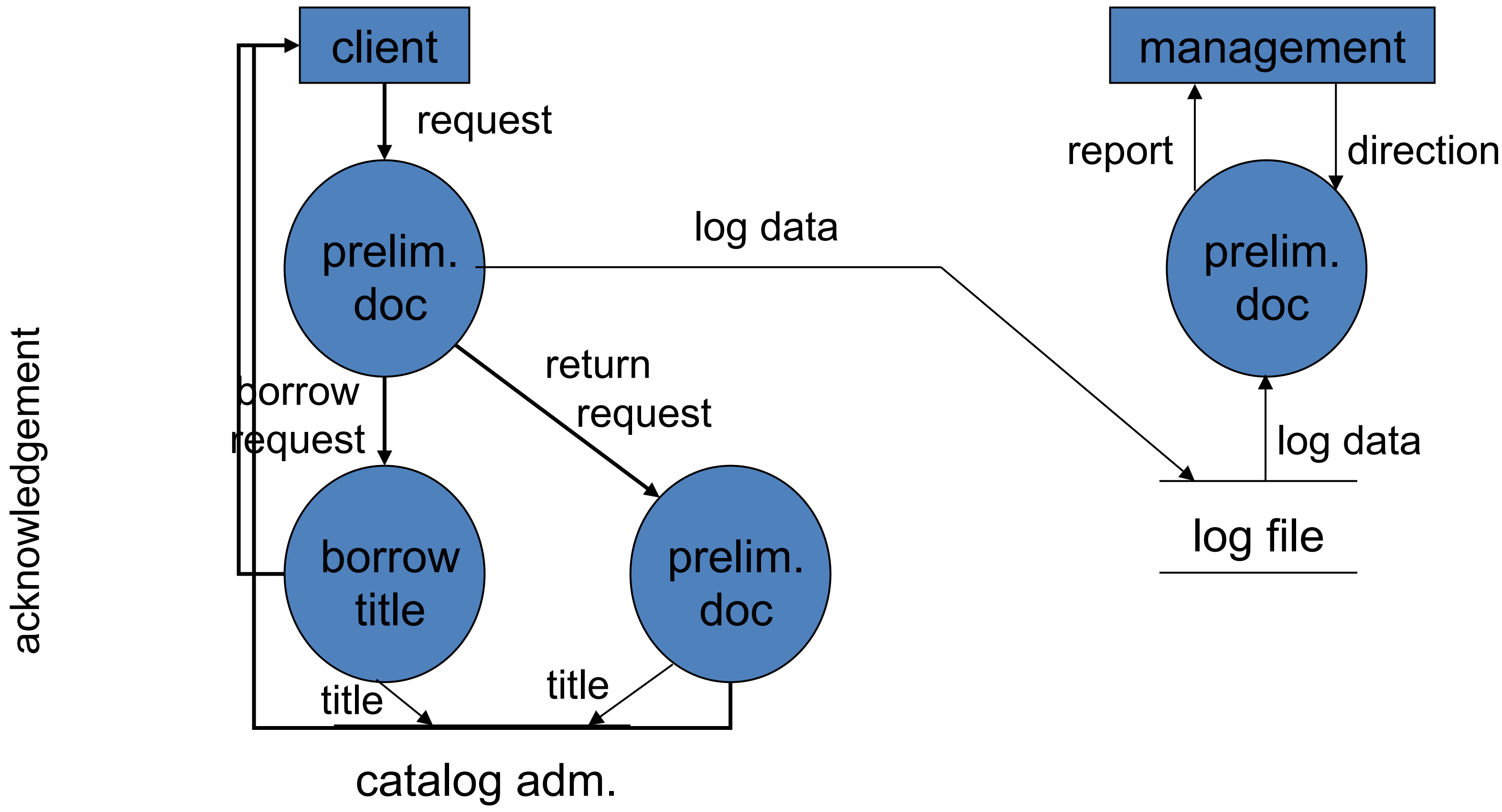


Data flow diagrams

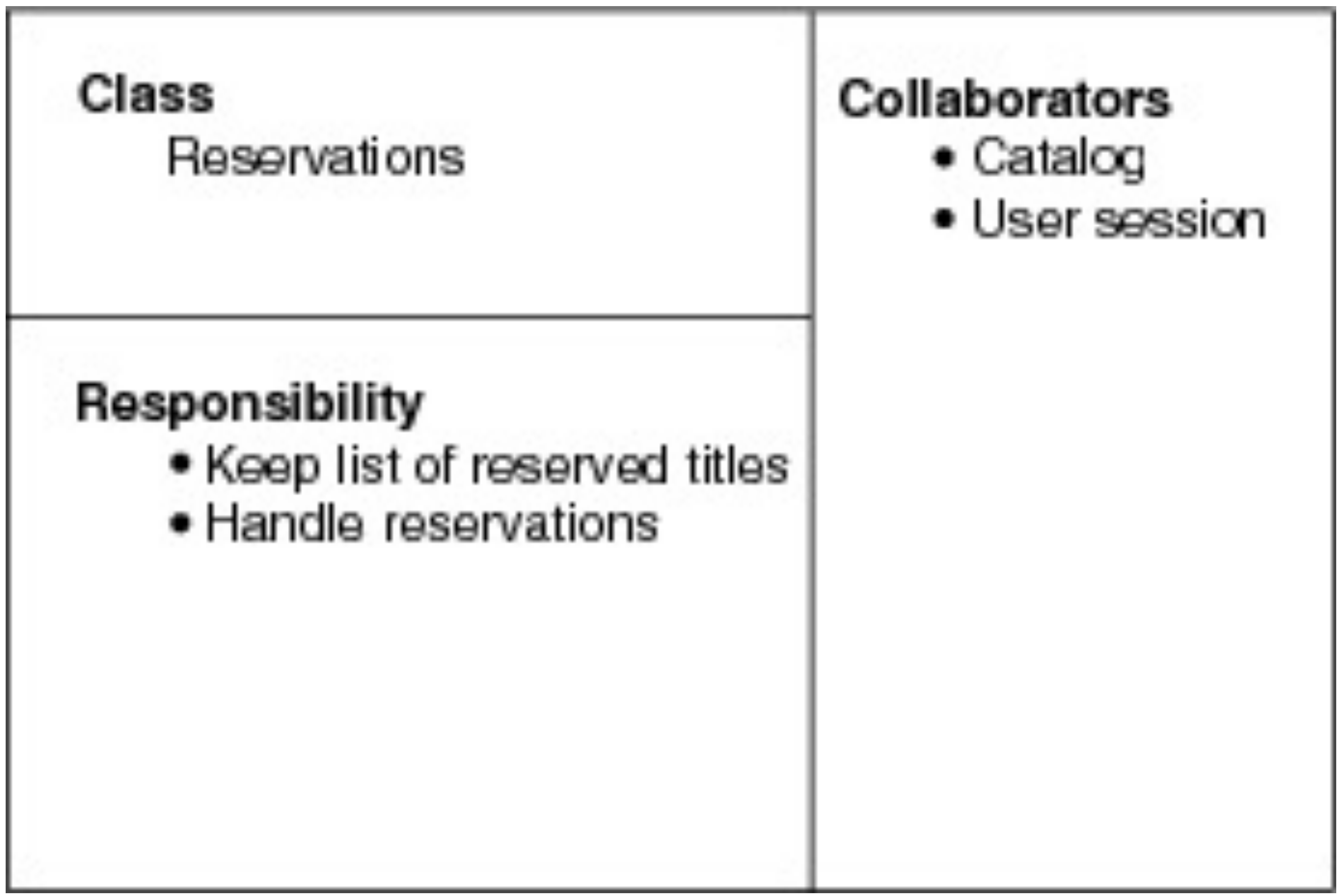
- external entities
- processes
- data flows
- data stores



Example data flow diagram



CRC: Class, Responsibility, Collaborators



Intermezzo: what is an object?

- Modeling viewpoint: model of part of the world
  - Identity+state+behavior
- Philosophical viewpoint: existential absractions
  - Everything is an object
- Software engineering viewpoint: data abstraction
- Implementation viewpoint: structure in memory
- Formal viewpoint: state machine

Objects and attributes

- Object is characterized by a set of attributes
  - A table *has* a top, legs, …
- In ERM, attributes denote *intrinsic* properties; they do not depend on each other, they are descriptive
- In ERM, relationships denote *mutual* properties, such as the membership of a person of some organization
- In UML, these relationships are called *associations*
- Formally, UML does not distinguish attributes and relationships; both are properties of a class

Objects, state, and behavior

- *State* = set of attributes of an object
- *Class* = set of objects with the same attributes
- Individual object: *instance*
- Behavior is described by *services*, a set of *responsibilities*
- Service is invoked by *sending a message*

Relations between objects

- Specialization-generalization, is-a
  - A dog *is an* animal
  - Expressed in hierarchy
- Whole-part, has
  - A dog *has* legs
  - Aggregation of parts into a whole
  - Distinction between ‘real-world’ part-of and ‘representational’ part of (e.g. ‘Publisher’ as part of ‘Publication’)
- Member-of, has
  - A soccer team *has* players
  - Relation between a set and its members (usually not transitive)

Specialization-generalization relations

- Usually expressed in hierarchical structure
  - If a tree: single inheritance
  - F a DAG: multiple inheritance

- Common attributes are defined at a higher level in the object hierarchy, and *inherited* by child nodes

- Alternative view: object hierarchy is a *type hierarchy*, with *types* and *subtypes*

Unified Modeling Language (UML)

- Controlled by OMG consortium: Object Management Group
- Latest version: UML 2
- UML 2 has 13 diagram types
  - Static diagrams depict static structure
  - Dynamic diagrams show what happens during execution
- Most often used diagrams:
  - class diagram: 75%
  - Use case diagram and communication diagram: 50%
  - Often loose semantics

UML diagram types

Static diagrams:

- Class
- Component
- Deployment
- Interaction overview
- Object
- Package

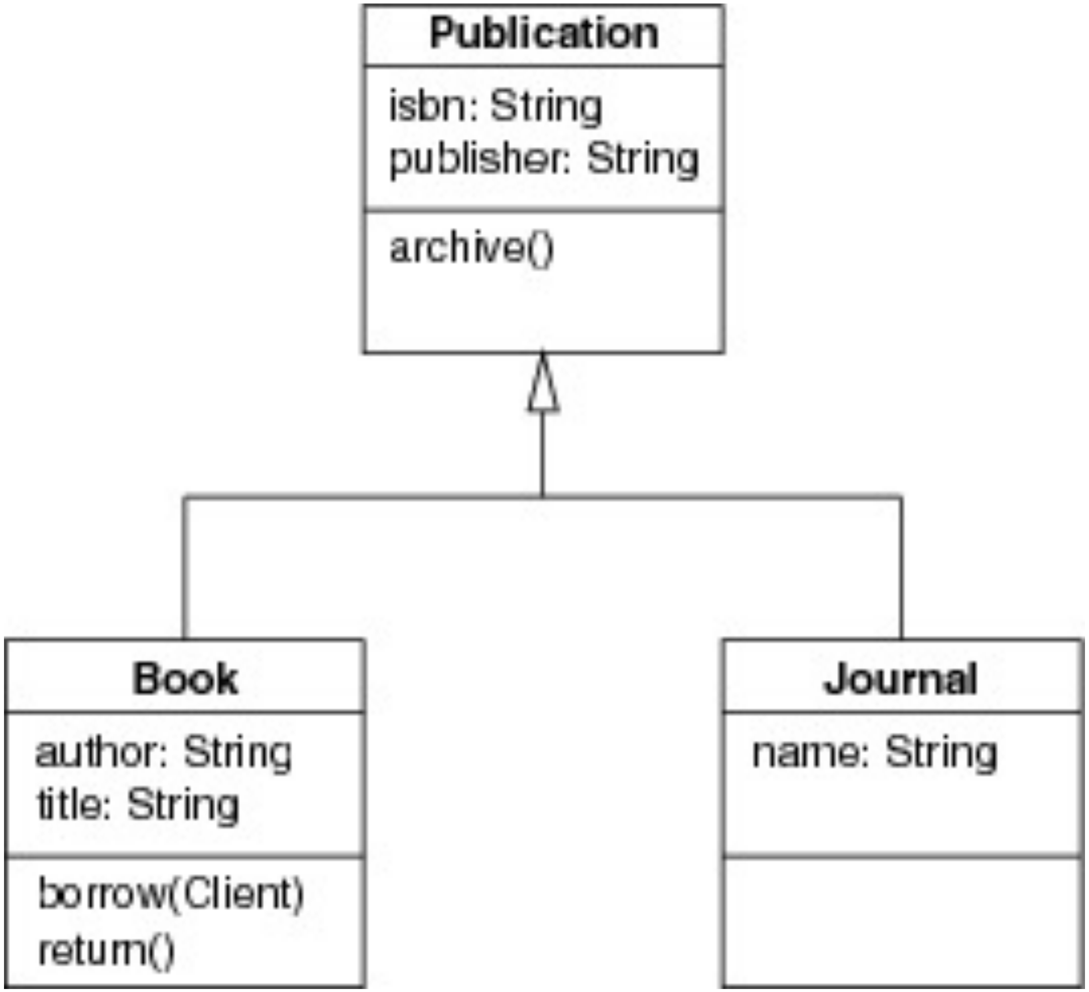
Dynamic diagrams:

- Activity
- Communication
- Composite structure
- Sequence
- State machine
- Timing
- Use case

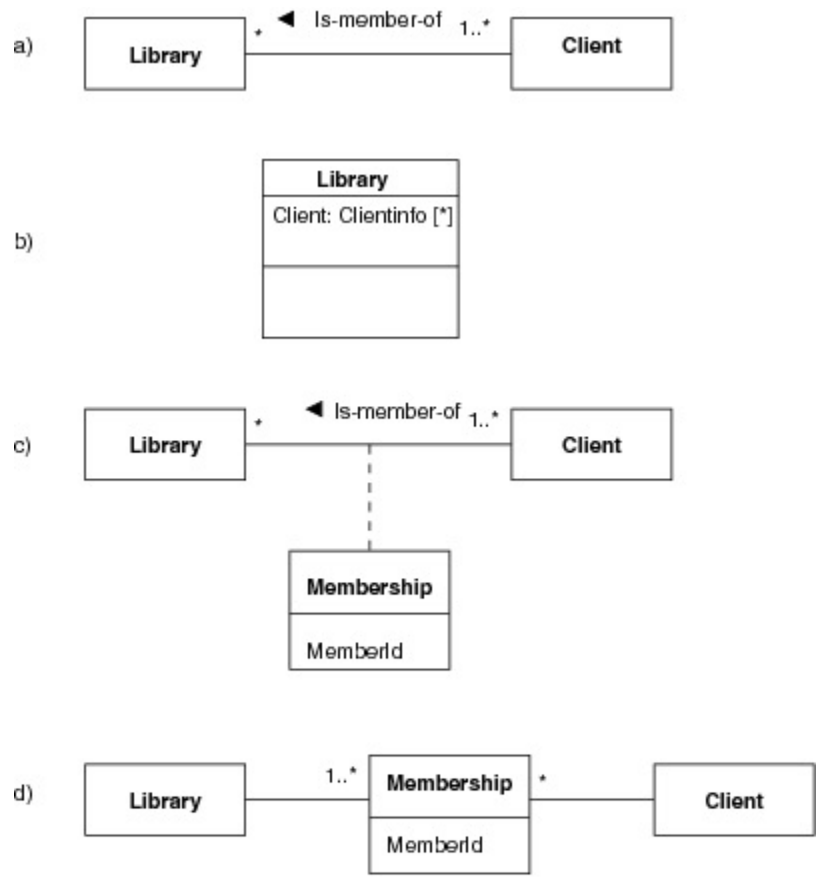
UML class diagram

- depicts the static structure of a system
- most common example: subclass/superclass hierarchy
- also mutual properties between two or more entities (ER relationships, often called associations in OO)

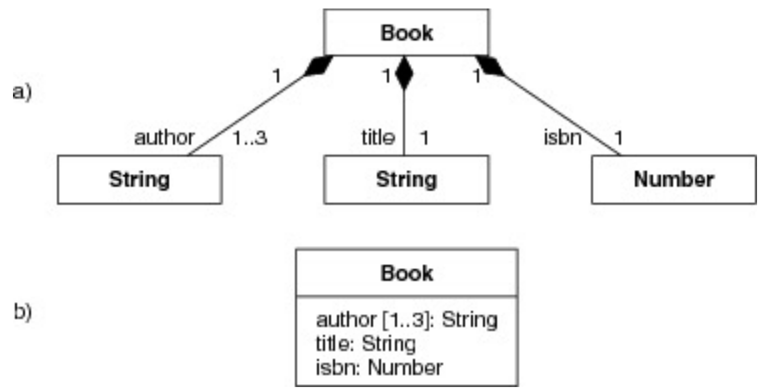
Example class diagram (1): generalization



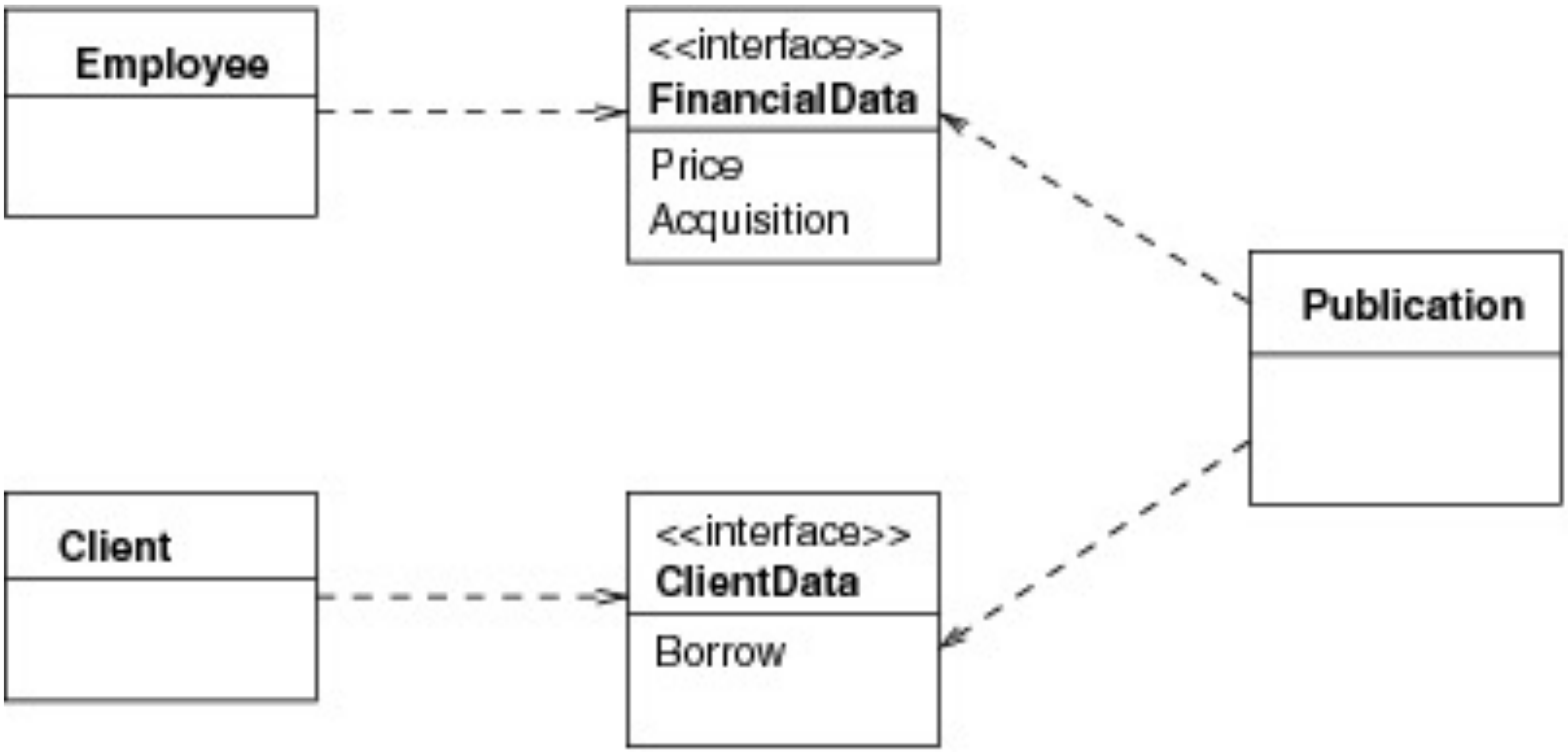
Example class diagram (2) association



Example class diagram (3): composition



Interface: class with abstract features



State machine diagram

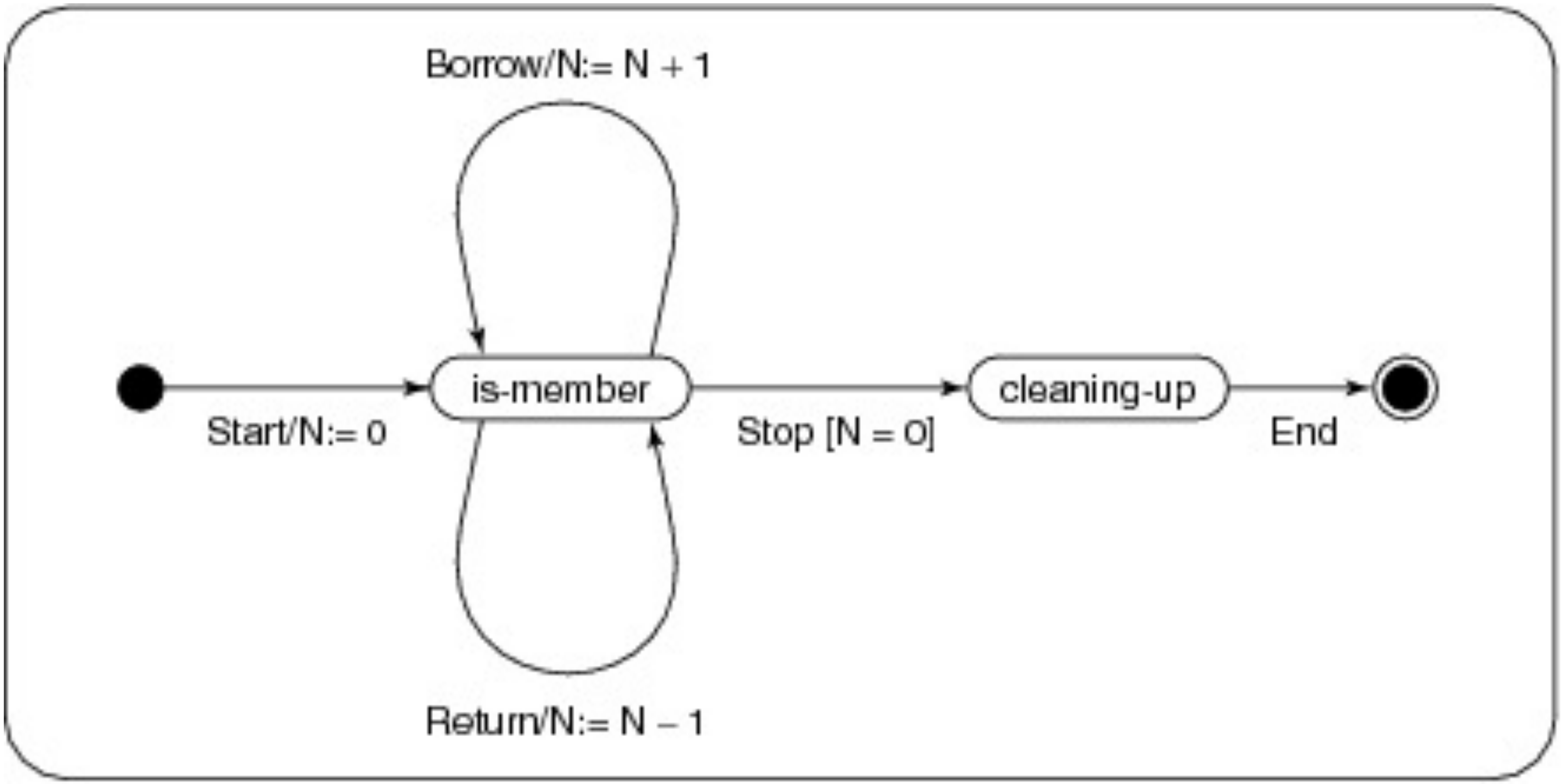
- Resembles finite state machines, but:

- Usually allows for local variables of states

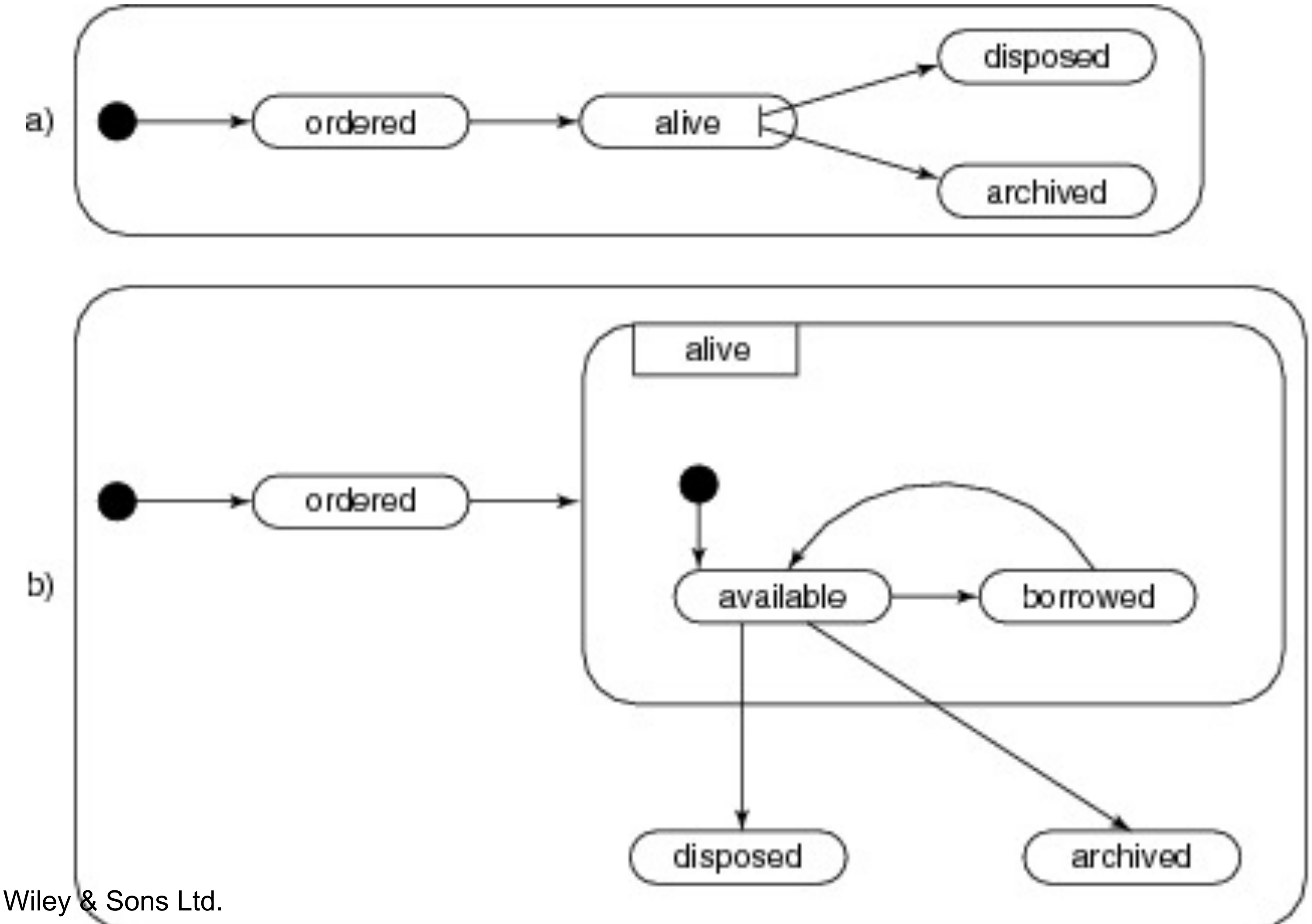
- Has external inputs and outputs

- Allows for hierarchical states

Example state machine diagram



Example state machine diagram: global and expanded view



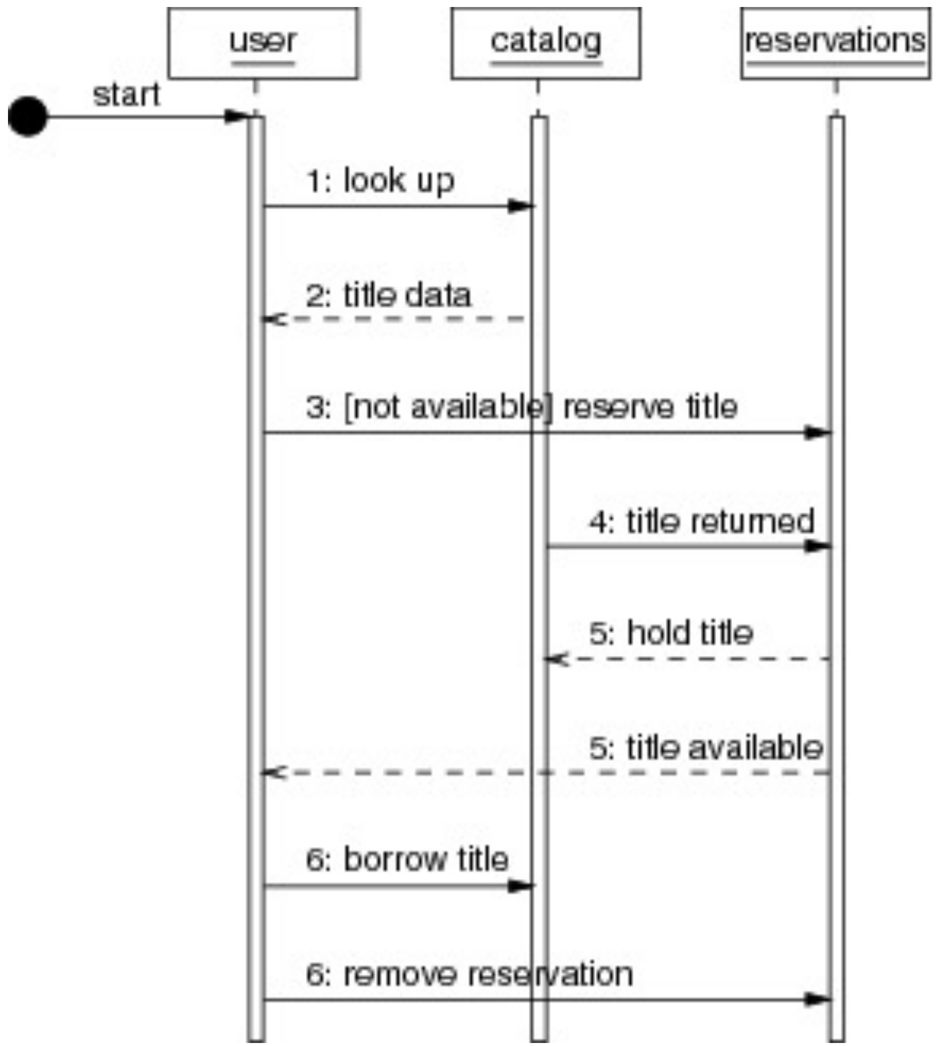
Interaction diagram

- Two types: sequence diagram and communication diagram

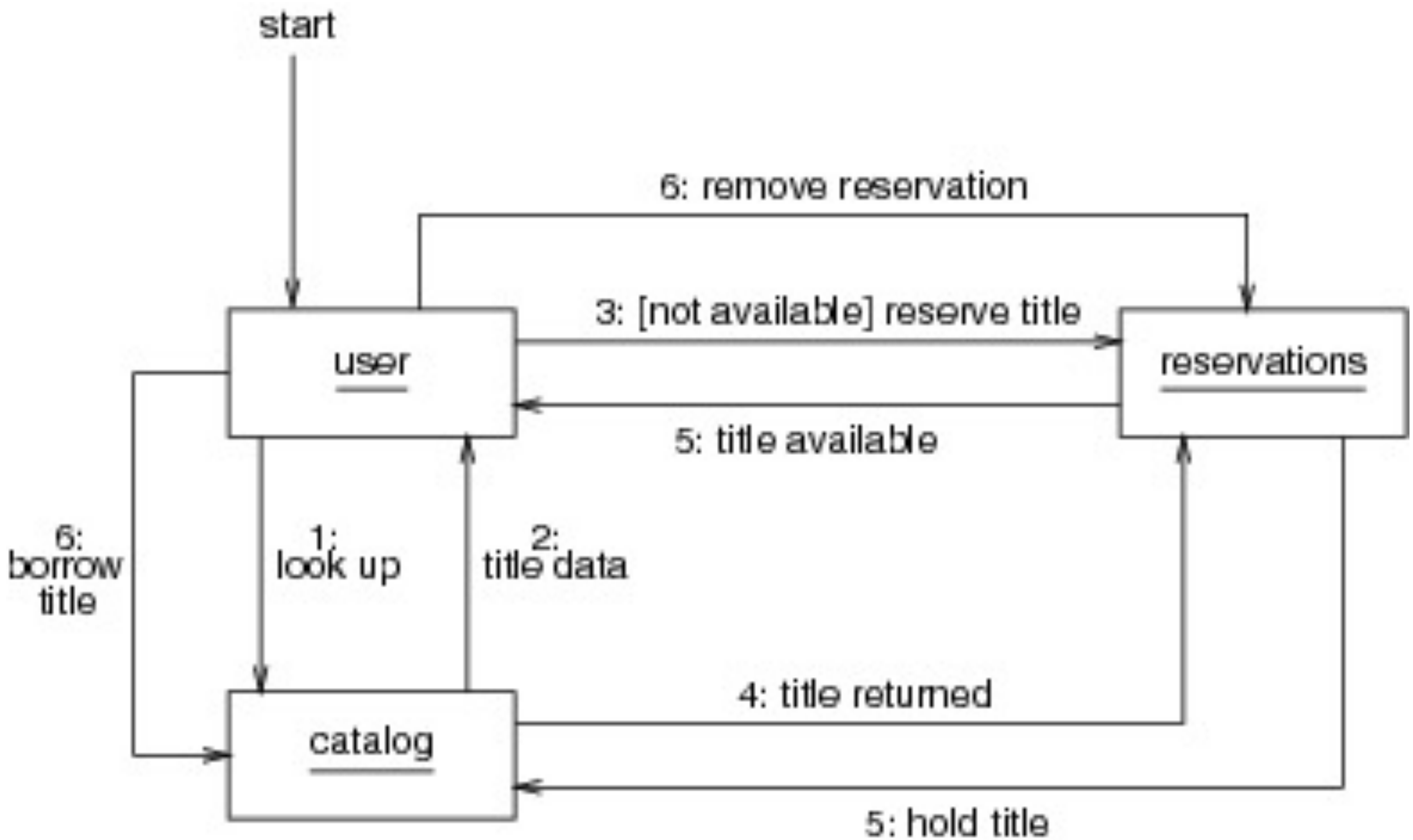
- Sequence diagram: emphasizes the ordering of events, using a *lifeline*

- Communication diagram emphasizes objects and their relationships

Example sequence diagram



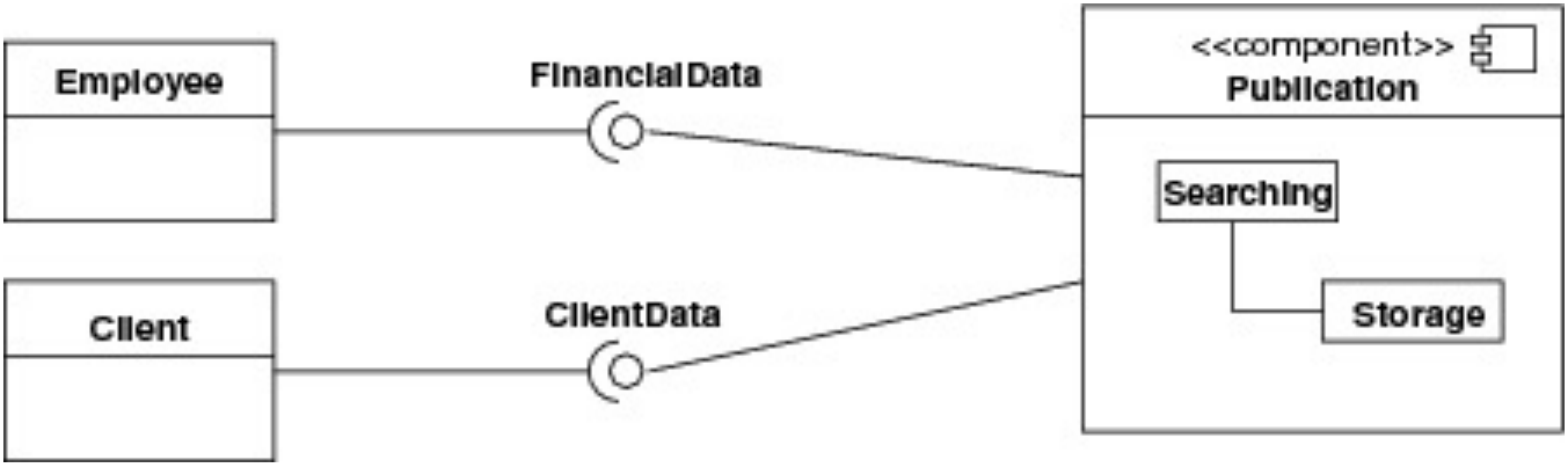
Example communication diagram



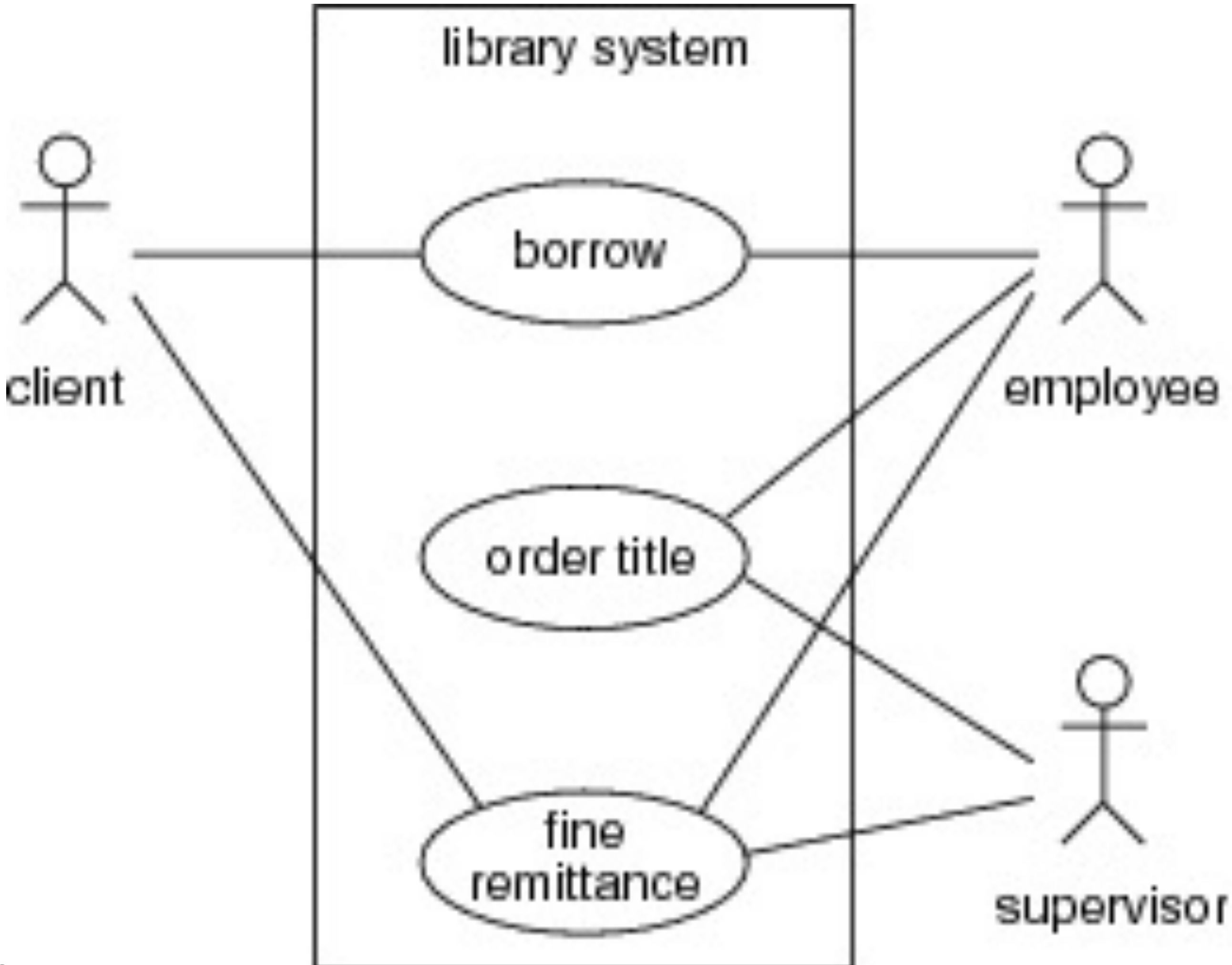
Component diagram

- Class diagram with stereotype <<component>>
- Way to identify larger entities
- One way to depict a module view (see Software Architecture chapter)
- Components are connected by interfaces

Example component diagram



Use case diagram



Summary

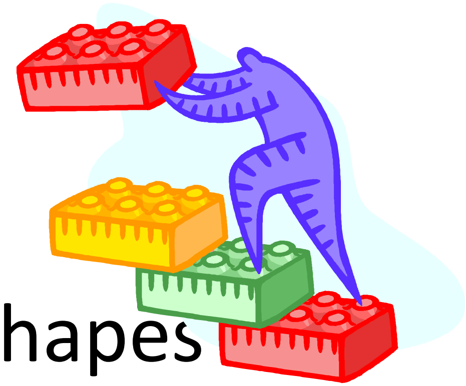
- Classic notations:
  - Entity-relationship diagrams
  - Finite state machines
  - Data flow diagrams
  - CRC cards
- Unified Modeling Language (UML)
  - evolved from earlier OO notations
  - 13 diagram types
  - widely used

Component-Based Software Engineering

Main issues:

- assemble systems out of (reusable) components
- compatibility of components

LEGO analogy

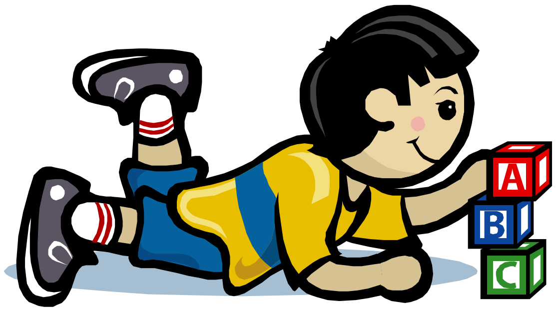


- Set of building blocks in different shapes colors
- Can be combined in different ways
- Composition through small stubs in one and corresponding holes in another building block

- ⇒ LEGO blocks are generic and easily composable

- LEGO can be combined with LEGO, not with

Why CBSE?

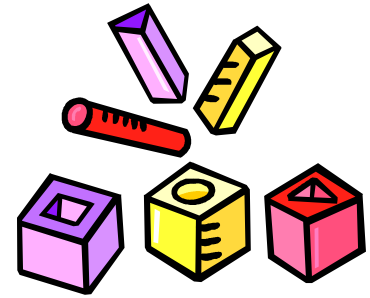


- CBSE increases quality, especially evolvability and maintainability

- CBSE increases productivity

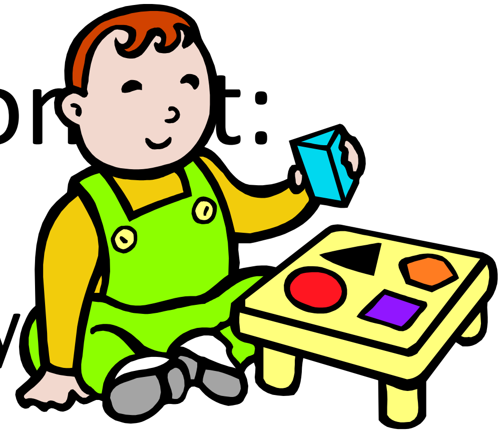
- CBSE shortens development time

Component model



- Defines the types of building block, and the recipe for putting them together
- More precisely, a component model defines standards for:
  - Properties individual components must satisfy
  - Methods and mechanisms for composing components
- Consequently, a component has to conform to

A software component:



- Implements some functionality
- Has explicit dependencies through provides and required interfaces
- Communicates through its interfaces only
- Has structure and behavior that conforms to a component model

A component technology

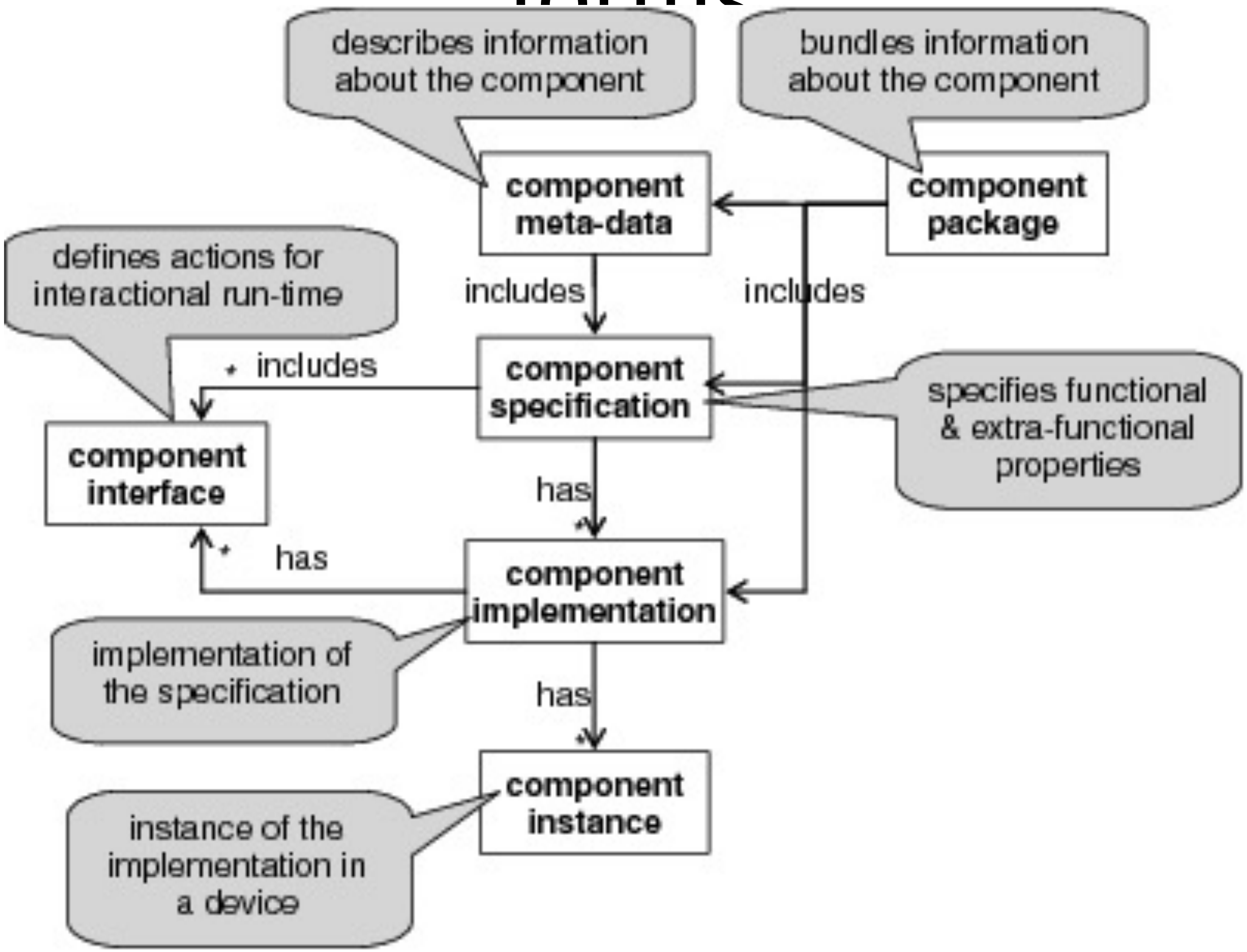
- Is the implementation of a component model, by means of:
  - Standards and guidelines for the implementation and execution of software components
  - Executable software that supports the implementation, assembly, deployment, execution of components

- Examples: EJB, COM+, .NET, CORBA

Component forms

- Component goes through different stages: development, packaging, distribution, deployment, execution
- Across these stages, components are represented in different forms:
  - During development: UML, e.g.
  - When packaging: in a .zip file, e.g.
  - In the execution stage: blocks of code and data

Characterization of component forms



Component specification vs component interface

- Interface describes how components interact: *usage contract*

- Specification is about the component as a whole, while an interface might be about part of a component only

Hiding of component internals

- Black box: only specification is known
- Glass box: internals may be inspected, but not changed
- Grey box: part of the internals may be inspected, limited modification is allowed
- While box: component is open to inspection and modification

Managing quality in CBSE

- *Who* manages the quality: the component, or the execution platform
- *Scope* of management: per-collaboration, or system-wide

Common features of component models

- Infrastructure mechanisms, for binding, execution, etc
- Instantiation
- Binding (design time, compile time, …)
- Mechanisms for communication between components
- Discovery of components
- Announcement of component capabilities (interfaces)
- Development support
- Language independence
- Platform independence
- Analysis support
- Support for upgrading and extension
- Support for quality properties

Development process in CBSE

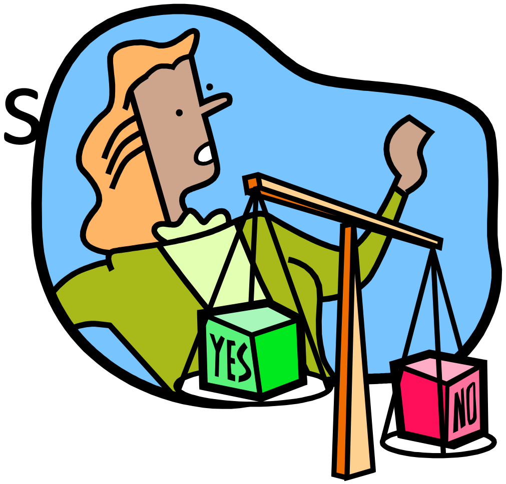
- Two separate development processes:
  - Development of components
  - Development of systems out of components

- Separate process to *assess* components

CBSE system development process

- Requirements: also considers *availability* of components (like in COTS)
- Analysis and design: very similar to what we normally do
- Implementation: less coding, focus on selection of components, provision of glue code
- Integration: largely automated
- Testing: verification of components is necessary

Component assess



- Find components
- Verify components
- Store components in repository

Component development process

- Components are intended for reuse

$$\Rightarrow$$

- Managing requirements is more difficult
- More effort required to develop reusable components
- More effort in documentation for consumers

Component development process

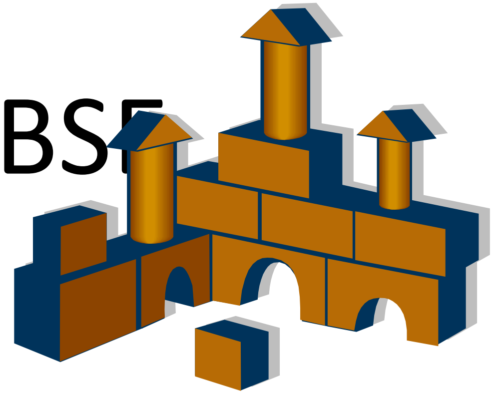
- Requirements: combination of top-down (from system) and bottom-up (generality)
- Analysis and design: generality is an issue, assumptions about system (use) must be made
- Implementation: largely determined by component technology
- Testing: extensive (no assumptions of usage!), and well-documented

- Release: not only executables, also metadata

Component maintenance

- Who is responsible: producer or consumer?
- Blame analysis: relation between manifestation of a fault and its cause, e.g.
  - Component A requires more CPU time
  - As a consequence, B does not complete in time
  - As required by C, so
  - C issues a time-out error to its user
  - Analysis: goes from C to B to A to input of A
  - Who does the analysis, if producers of A,B,C are different?

Architecture and CBSE



- Architecture-driven: top-down: components are identified as part of an architectural design
- Product line: family of similar products, with 1 architecture
- COTS-based: bottom-up, architecture is secondary to components found

Summary

- To enable composition, components must be compatible: achieved by component model
- Separation of development process for components from that of assembling systems out of components
- Architectural plan organizes how components fit together and meet quality requirements

Requirements Engineering

Main issues:

- What do we want to build
- How do we write this down

Requirements Engineering

- the first step in finding a solution for a data processing problem
- the results of requirements engineering is a requirements specification
- requirements specification
  - contract for the customer
  - starting point for design

Natural language specs are dangerous

"All users have the same control field"

- the same value in the control field?
- the same format of the control field?
- there is one (1) control field for all users?

Requirements engineering, main steps

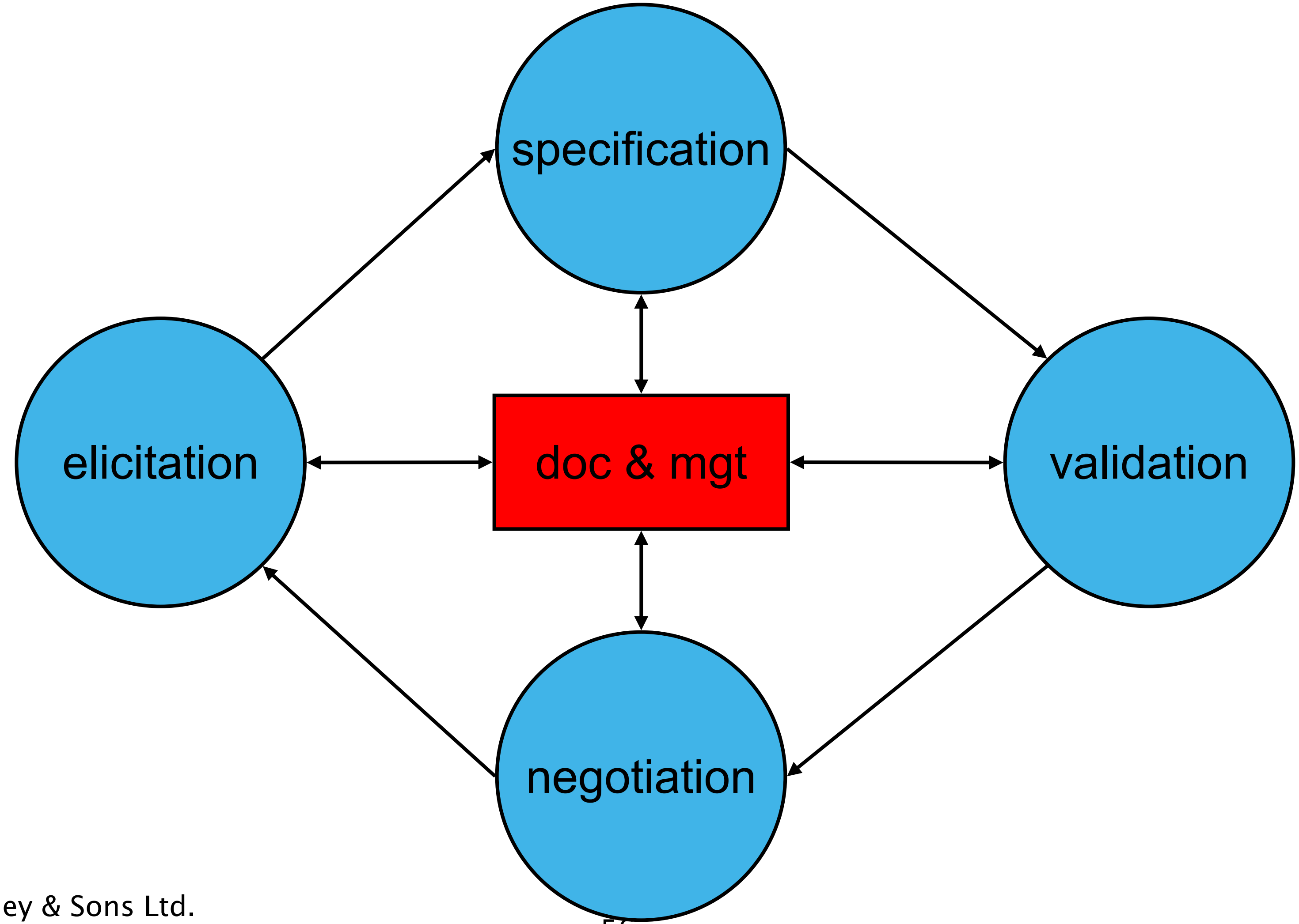
1. understanding the problem: elicitation

2. describing the problem: specification

3. agreeing upon the nature of the problem: validation

4. agreeing upon the boundaries of the problem: negotiation

Framework for RE process



Conceptual modeling

- you model part of reality: the Universe of Discourse (UoD)
- this model is an explicit conceptual model
- people in the UoD have an implicit conceptual model of that UoD
- making this implicit model explicit poses problems:

Requirements engineering is difficult

Success depends on the degree with which we manage to properly describe the system desired

Software is not continuous!

Tsjechow vs Chekhov vs ЦЕХОВ

Beware of subtle mismatche

- a library employee may also be a client
- there is a difference between `a book` and `a copy of a book`
- status info `present` / `not present` is not sufficient; a (copy of a) book may be lost, stolen, in repair, ...

Humans as information sourc

- different backgrounds
- short-term vs long-term memory
- human prejudices
- limited capability for rational thinking

Negotiation problems

- existing methods are “Taylorian”
- they may work in a “technical” environment, but many UoDs contain people as well, and their models may be irrational, incomplete, inconsistent, contradictory
- as an analyst, you cannot neglect these aspects; you participate in <u>shaping</u> the UoD

Point to ponder #1

- how do you handle conflicts during requirements engineering?

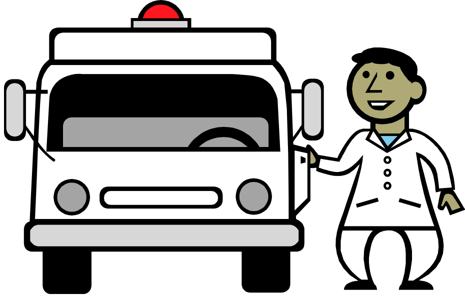
How we study the world around us

- people have a set of assumptions about a topic they study (paradigm)
- this set of assumptions concerns:
  – how knowledge is gathered
  – how the world is organized
- this in turn results in two dimensions:
  – subjective-objective (wrt knowledge)
  – conflict-order (wrt the world)
- which results in 4 archetypical approaches to requirements engineering

Four approaches to RE

- functional (objective+order): the analyst is the expert who empirically seeks the truth
- social-relativism (subjective+order): the analyst is a `change agent'. RE is a learning process guided by the analyst
- radical-structuralism (objective+ conflict): there is a struggle between classes; the analyst chooses for either party
- neohumanism (subjective+conflict): the analyst is kind of a social therapist, bringing

Point to ponder #2



- how does the London Ambulance System example from chapter 1 relate to the different possible approaches to requirements engineering?

Elicitation techniques

- interview
- Delphi technique
- brainstorming session
- task analysis
- scenario analysis
- ethnography
- form analysis
- analysis of natural language descriptions
- synthesis from existing system
- domain analysis
- Business Process Redesign (BPR)
- prototyping

Task Analysis

- Task analysis is the process of analyzing the way people perform their jobs: the things they do, the things they act on and the things they need to know.
- The relation between tasks and goals: a task is performed in order to achieve a goal.
- Task analysis has a broad scope.

Task Analysis (cntd)

- Task analysis concentrates on the current situation. However, it can be used as a starting point for a new system:
  - users will refer to new elements of a system and its functionality
  - scenario-based analysis can be used to exploit new possibilities
- See also the role of task analysis as discussed in the context of user interface design (chapter 16)

Scenario-Based Analysis

- Provides a more user-oriented view perspective on the design and development of an interactive system.
- The defining property of a scenario is that it projects a concrete description of an activity that the user engages in when performing a specific task, a description sufficiently detailed so that the design implications can be inferred and reasoned about.

Scenario-Based Analysis (example)

- first shot:
  - check due back date
  - if overdue, collect fine
  - record book as being available again
  - put book back

- as a result of discussion with library employee:
  - what if person returning the book is not registered as a client?
  - what if the book is damaged?

Scenario-Based Analysis (cntd)

The scenario view	The standard view
• concrete descriptions	• abstract descriptions
• focus on particular instances	• focus on generic types
• work-driven	• technology-driven
• open-ended, fragmentary	• complete, exhaustive
• informal, rough, colloquial	• formal, rigorous
• envisioned outcomes	• specified outcomes

Scenario-Based Analysis (cntd)

- Application areas:
  - requirements analysis
  - user-designer communication
  - design rationale
  - sofware architecture (& its analysis)
  - software design
  - implementation
  - verification & validation
  - documentation and training
  - evaluation

Form analysis (example

Proceedings request form:

Client name	……………
Title	……………
Editor	……………
Place	……………
Publisher	……………
Year	……………

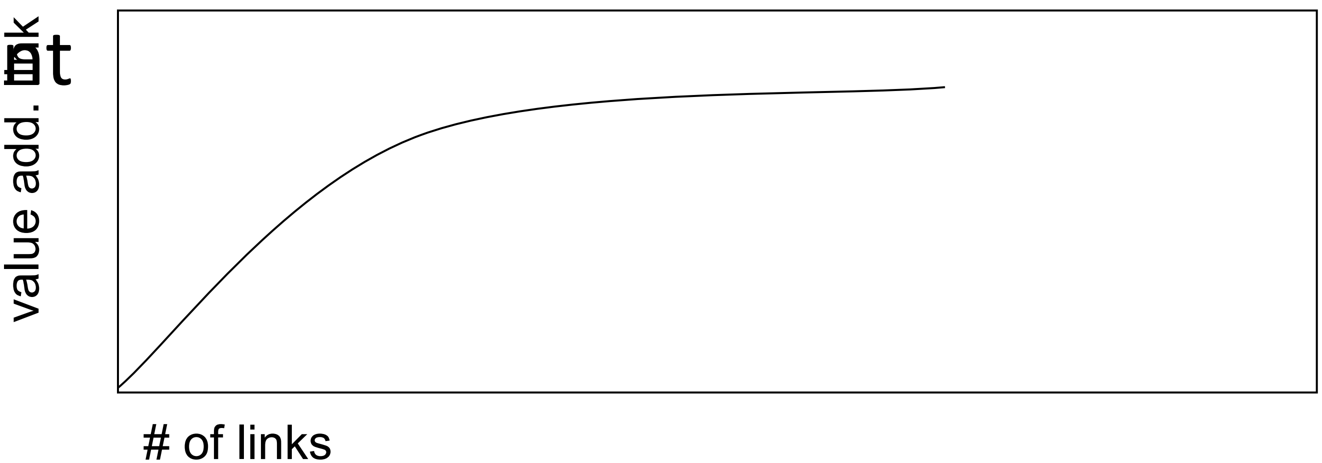
Certainty vs uncertainty

Types of links between customer and developer

- facilitated teams
- intermediary
- support line/help desk
- survey
- user interface prototyping
- requirements prototyping
- interview
- usability lab
- observational study
- user group
- trade show
- marketing & sales

Direct versus indirect links

- lesson 1: don't rely too much on indirect links (intermediaries, surrogate users)
- lesson 2: the more links, the better - up to a point



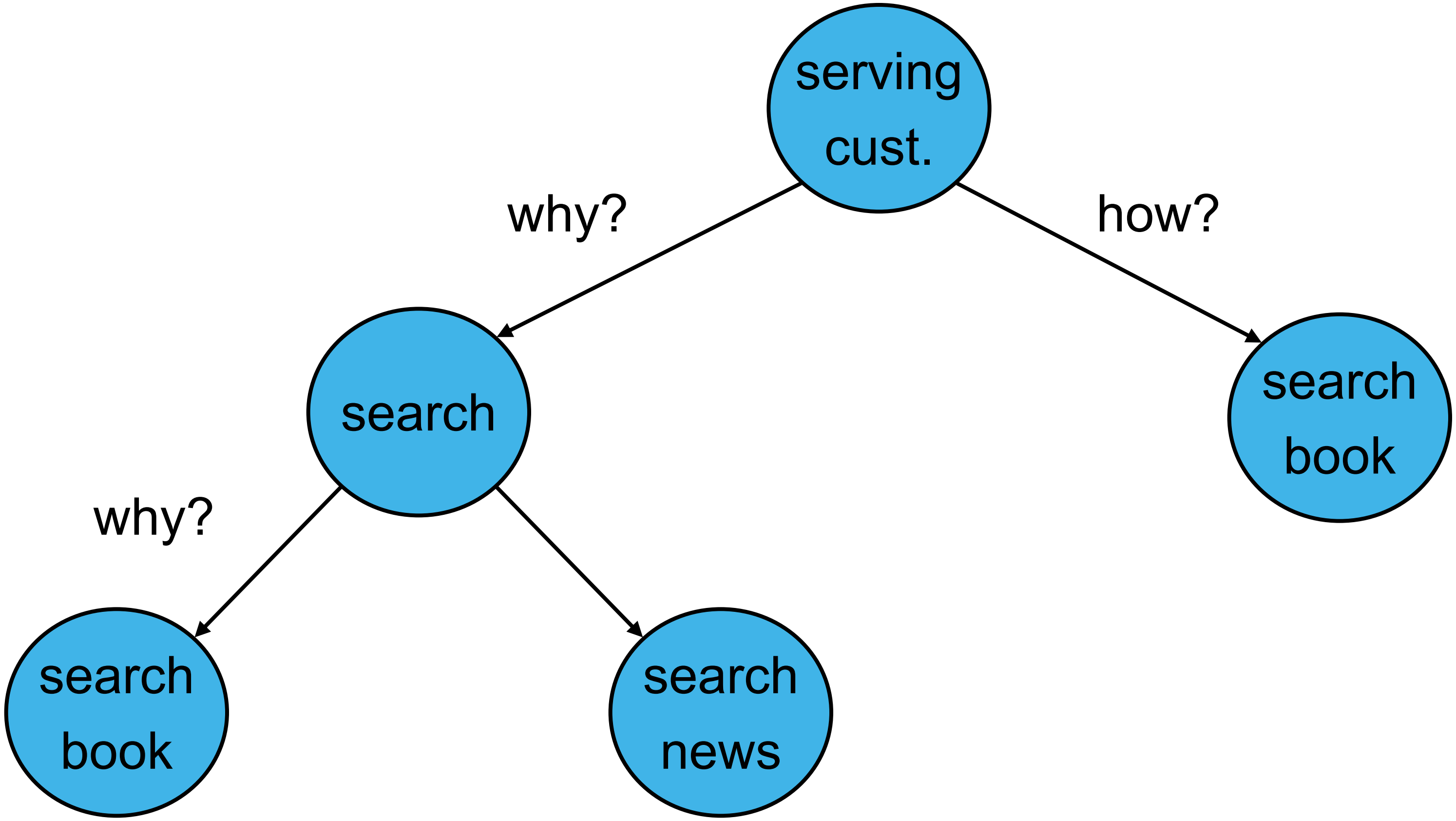
Structuring a set of requirements

1. Hierarchical structure: higher-level reqs are decomposed into lower-level reqs

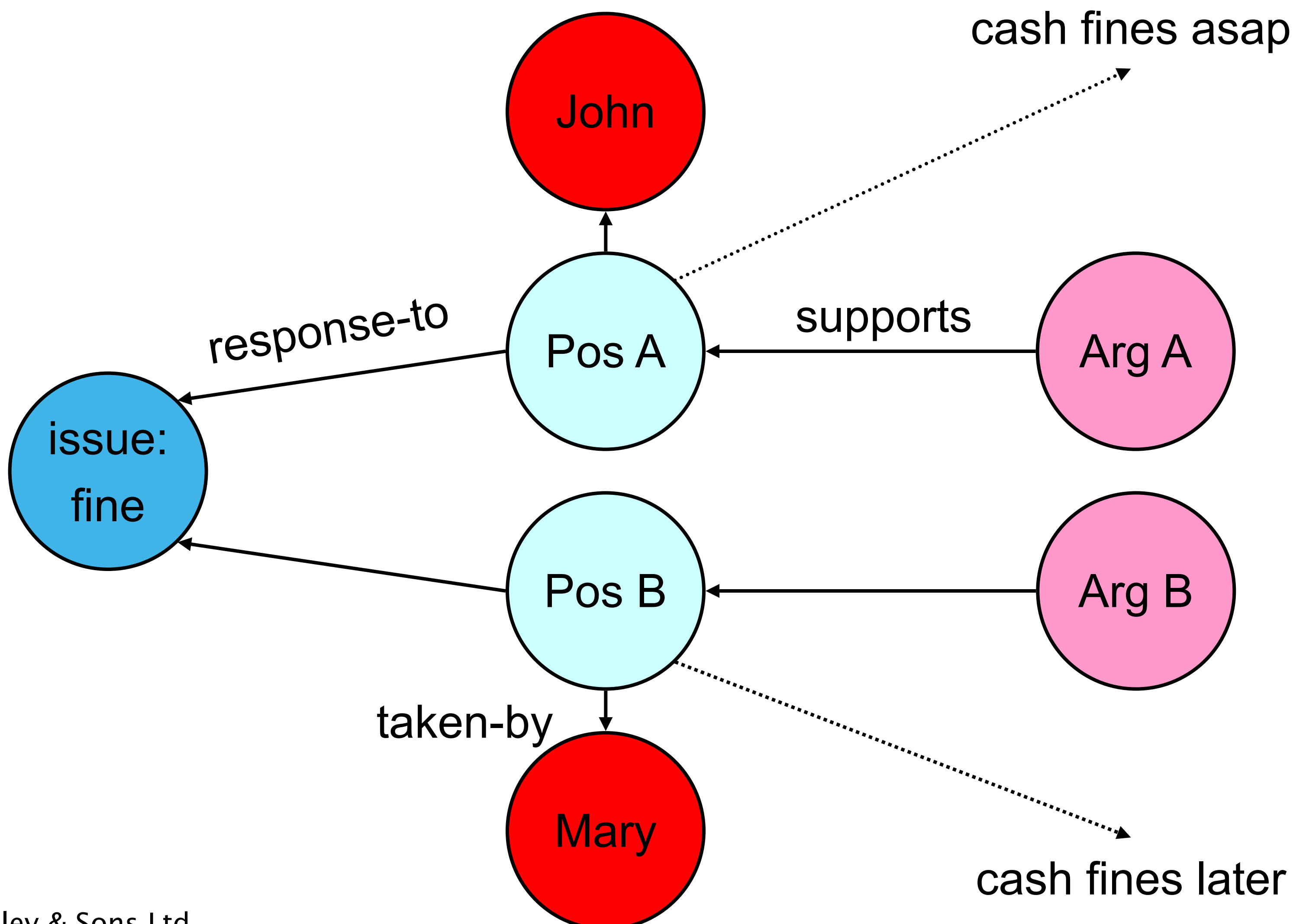
2. Link requirements to specific stakeholders (e.g. management and end users each have their own set)

In both cases, elicitation and structuring go hand in hand

Goal-driven requirements engineering



Conflicting viewpoints



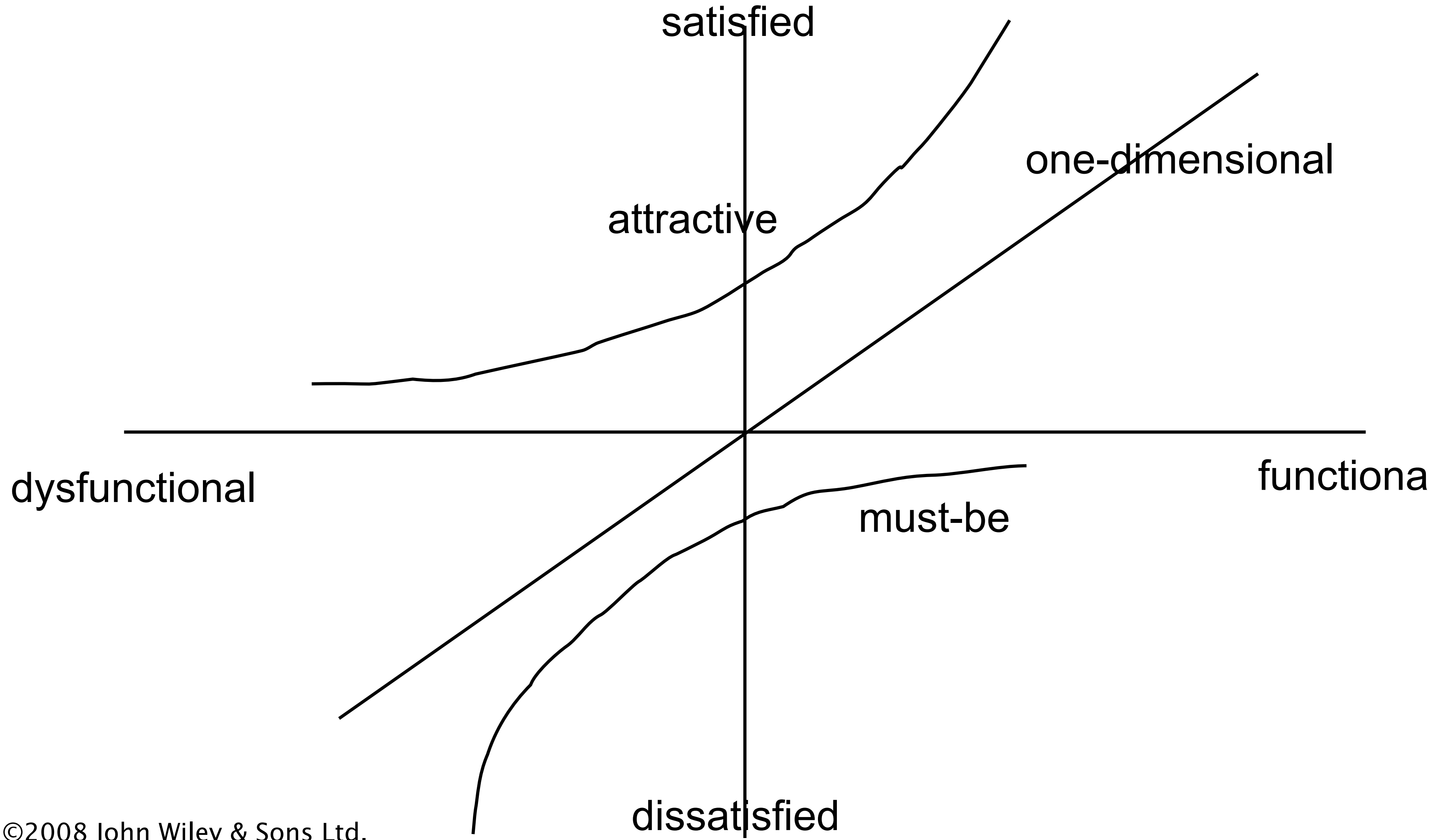
Prioritizing requirements (MoSCoW)

- Must haves: top priority requirements
- Should haves: highly desirable
- Could haves: if time allows
- Won’t haves: not today

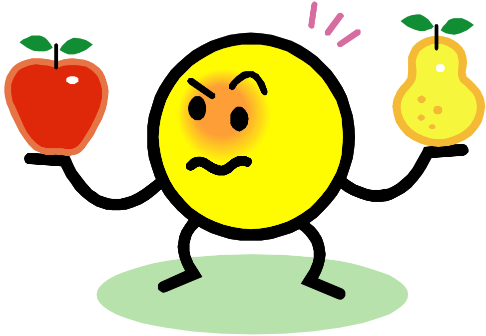
Prioritizing requirements (Kano model)

- Attractive: more satisfied if +, not less satisfied if –
- Must-be: dissatisfied when -, at most neutral
- One-dimensional: satisfaction proportional to number
- Indifferent: don’t care
- Reverse: opposite of what analist thought
- Questionable: preferences not clear

Kano diagram



COTS selection



- COTS: Commercial-Off-The-Shelf
- Iterative process:
  – Define requirements
  – Select components
  – Rank components
  – Select most appropriate component, or iterate
- Simple ranking: weight * score (WSM – Weighted Scoring Method)

Crowdsourcing

1. Go to LEGO site
2. Use CAD tool to design your favorite castle
3. Generate bill of materials
4. Pieces are collected, packaged, and sent to you

5. Leave your model in LEGO's gallery
6. Most downloaded designs are prepackaged

- No requirements engineers needed!

Requirements specification

- readable
- understandable
- non-ambiguous
- complete
- verifiable
- consistent
- modifiable
- traceable
- usable
- …
- ...

IEEE Standard 830

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

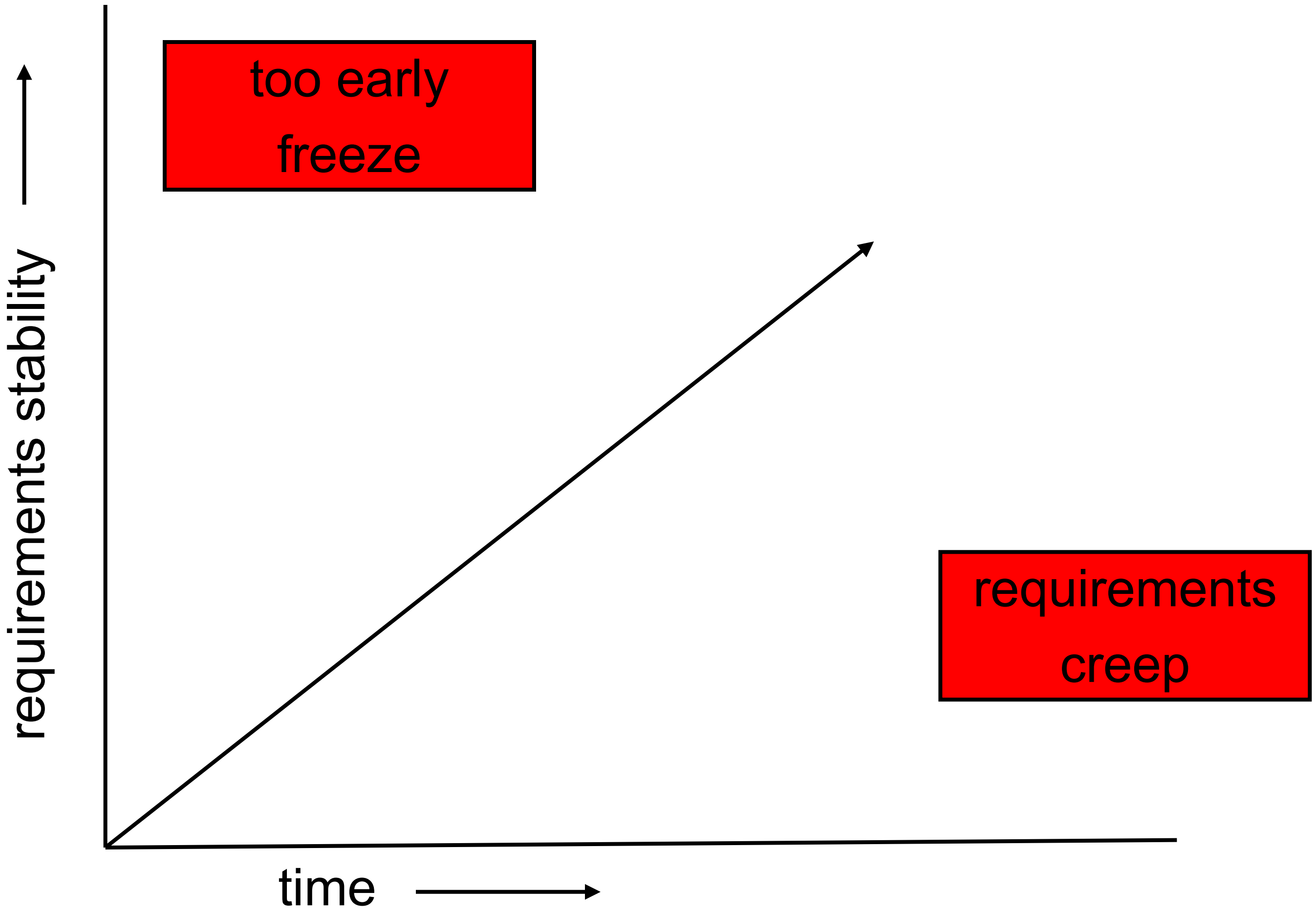
IEEE Standard 830 (cntd)

3. Specific requirements
    - 3.1. External interface requirements
        - 3.1.1. User interfaces
        - 3.1.2. Hardware interfaces
        - 3.1.3. Software interfaces
        - 3.1.4. Comm. interfaces
    - 3.2. Functional requirements
        - 3.2.1. User class 1
            - 3.2.1.1. Functional req. 1.1
            - 3.2.1.2. Functional req. 1.2

              ...
        - 3.2.2. User class 2

          …
    - 3.3. Performance requirements
    - 3.4. Design constraints
    - 3.5. Software system attributes
    - 3.6. Other requirements

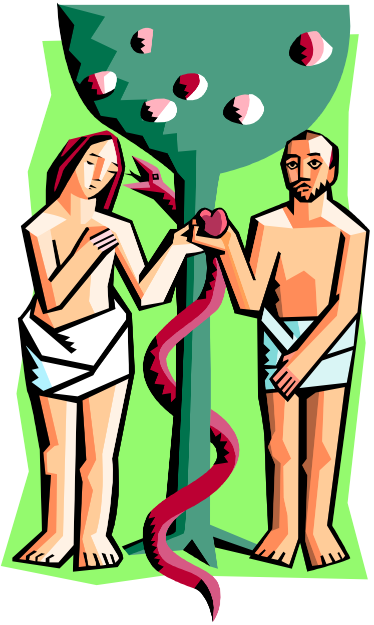
Requirements management



Requirements management

- Requirements identification (number, goal-hierarchy numbering, version information, attributes)
- Requirements change management (CM)
- Requirements traceability:
  - Where is requirement implemented?
  - Do we need this requirement?
  - Are all requirements linked to solution elements?
  - What is the impact of this requirement?
  - Which requirement does this test case cover?

The 7 sins of the analyst



- noise
- silence
- overspecification
- contradictions
- ambiguity
- forward references
- wishful thinking

Functional vs. Non-Functional Requirements

- functional requirements: the system services which are expected by the users of the system.
- non-functional (quality) requirements: the set of constraints the system must satisfy and the standards which must be met by the delivered system.
  - speed
  - size
  - ease-of-use

Validation of requirements

- inspection of the requirement specification w.r.t. correctness, completeness, consistency, accuracy, readability, and testability.

- some aids:
  - structured walkthroughs
  - prototypes
  - develop a test plan
  - tool support for formal specifications

Summary

- goal: a maximally clear, and maximally complete, description of WHAT is wanted
- RE involves elicitation, specification, validation and negotiation
- modeling the UoD poses both analysis and negotiation problems
- you must realize that, as an analyst, you are more than an outside observer
- a lot is still done in natural language, with all its inherent problems

Software Reusability

Main issues:

- Why is reuse so difficult
- How to realize reuse

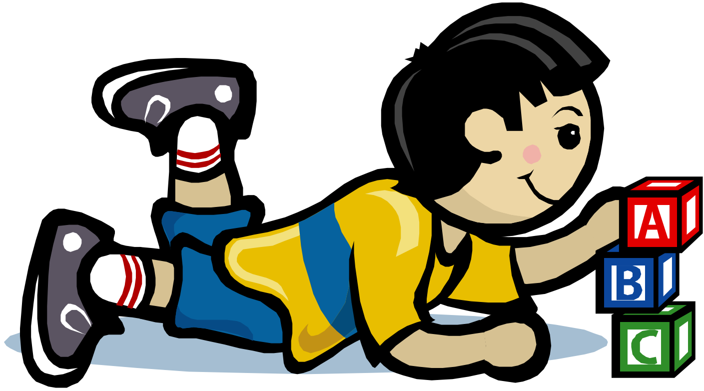
Reuse dimensions

- Things being reused: components, concepts, …
- Scope: horizontal vs vertical
- Approach: systematic or opportunistic
- Technique: compositional or generative
- Use: black-box or white-box
- Product being reused: source code, design, …

Success criteria for component libraries

- Well-developed field, standard terminology
- Small interfaces
- Standardized data formats

Requirements for component libraries



- Searching for components
- Understanding/evaluating components found
- Adapt components if necessary
- Compose systems from components

Component evaluation, useful information

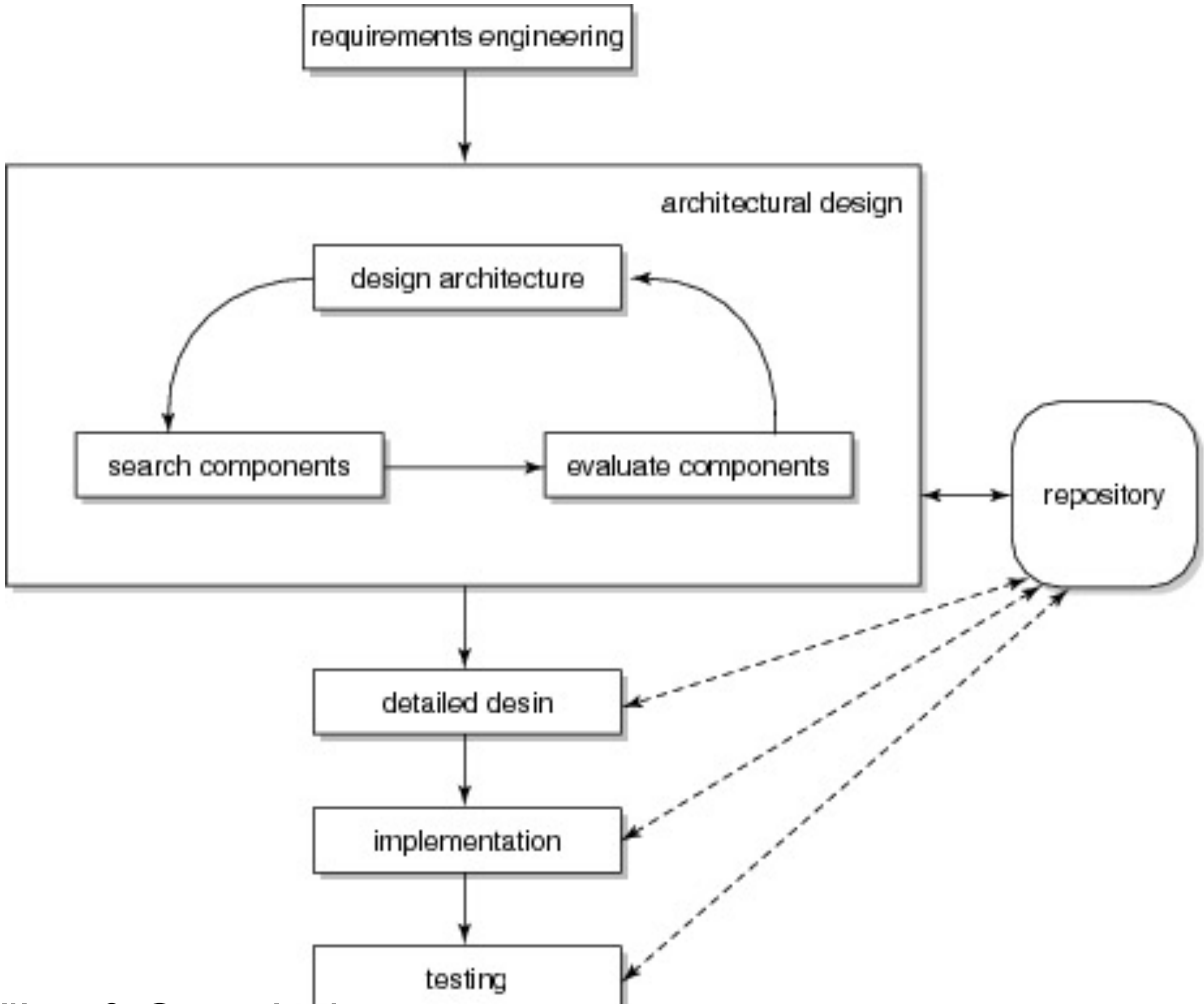
- Quality information
- Administrative information (name developer, modification history, etc)
- Documentation
- Interface information
- Test information

Reuse process models

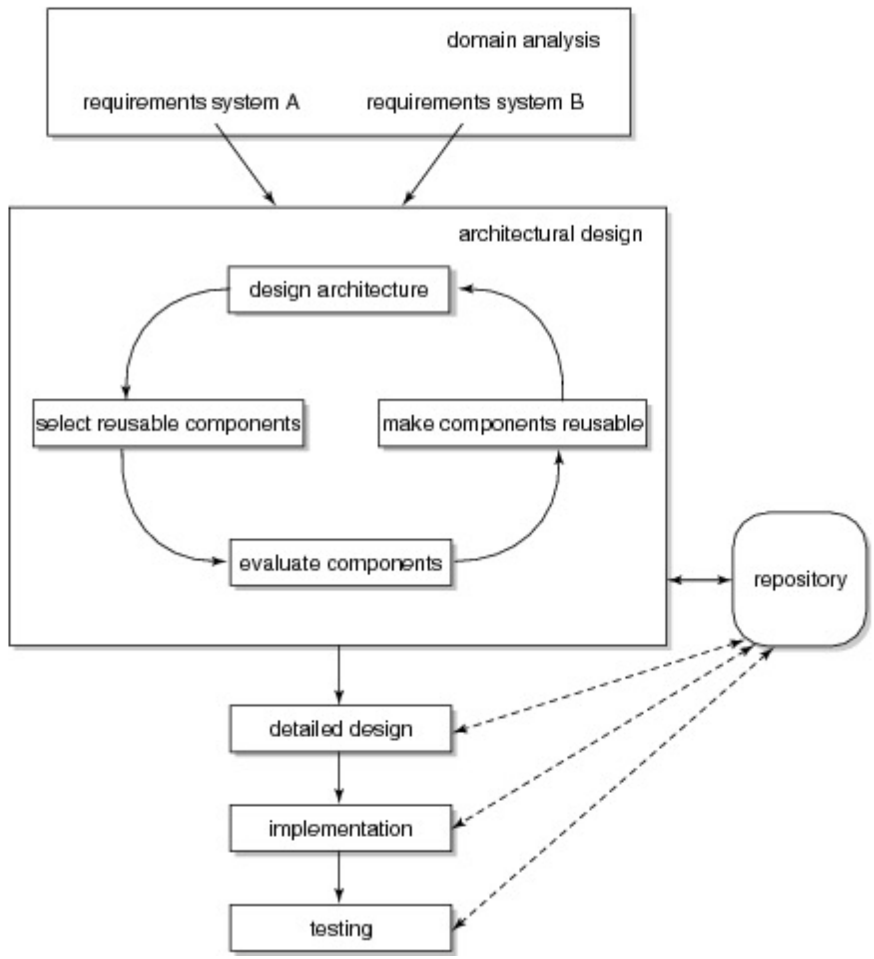
- Software development *with* reuse
  - Passive
  - Component library evolves haphazardly

- Software development *for* reuse
  - Active
  - Reusable assets are *developed*, rather than found by accident

Software development with reuse



Software development for reuse



Software development for reuse

- Often two separate development processes:
  - Development of components (involving domain analysis)
  - Development of applications, using the available components

- Specific forms hereof:
  - Component-based software development
  - Software factory
  - Software product lines

Reuse tools and techniques

- Languages to describe compositions
  – Module Interconnection Language (MIL)
  – Architecture Description Language (ADL)

- Middleware (CORBA, JavaBeans, .NET)

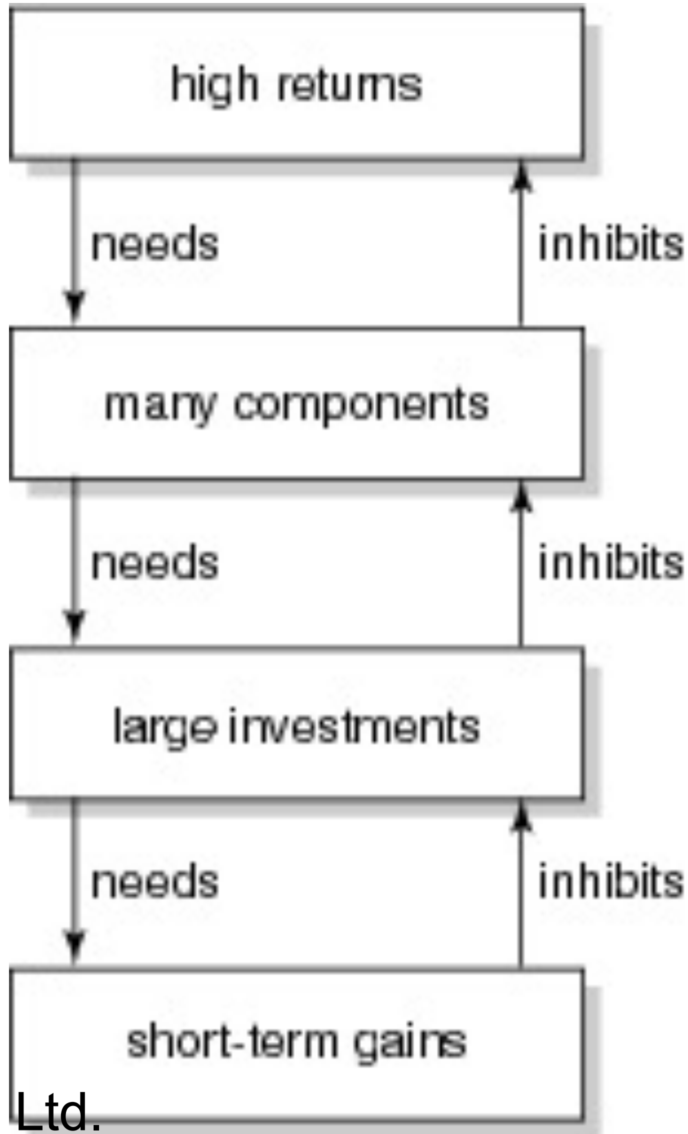
Characteristics of successful reuse programs

- Extensive management support
- Organizational support structure
- Incremental implementation
- Significant success
- High incentives
- Domain analysis done
- Attention to architectural issues

Non-technical aspects of software reuse

- Economics: it is a long term investment
- Management: it does not happen spontaneously
- Psychology: people do not want to reuse someone else's code

Reuse devil's loop



Summary

- We can reuse different things: code, design, …
- Reuse can be *systematic* (software development *for* reuse), or *opportunistic* (software development *with* reuse)
- Reuse does not just happen; it needs to be planned

One final lesson

Walking on water

and

developing software from a specification

are easy

if they are frozen

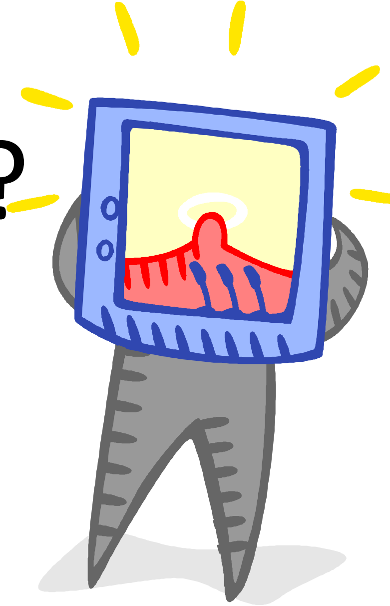
(E.V. Berard, Essays on object-oriented software engineering)

User Interface Design

Main issues:

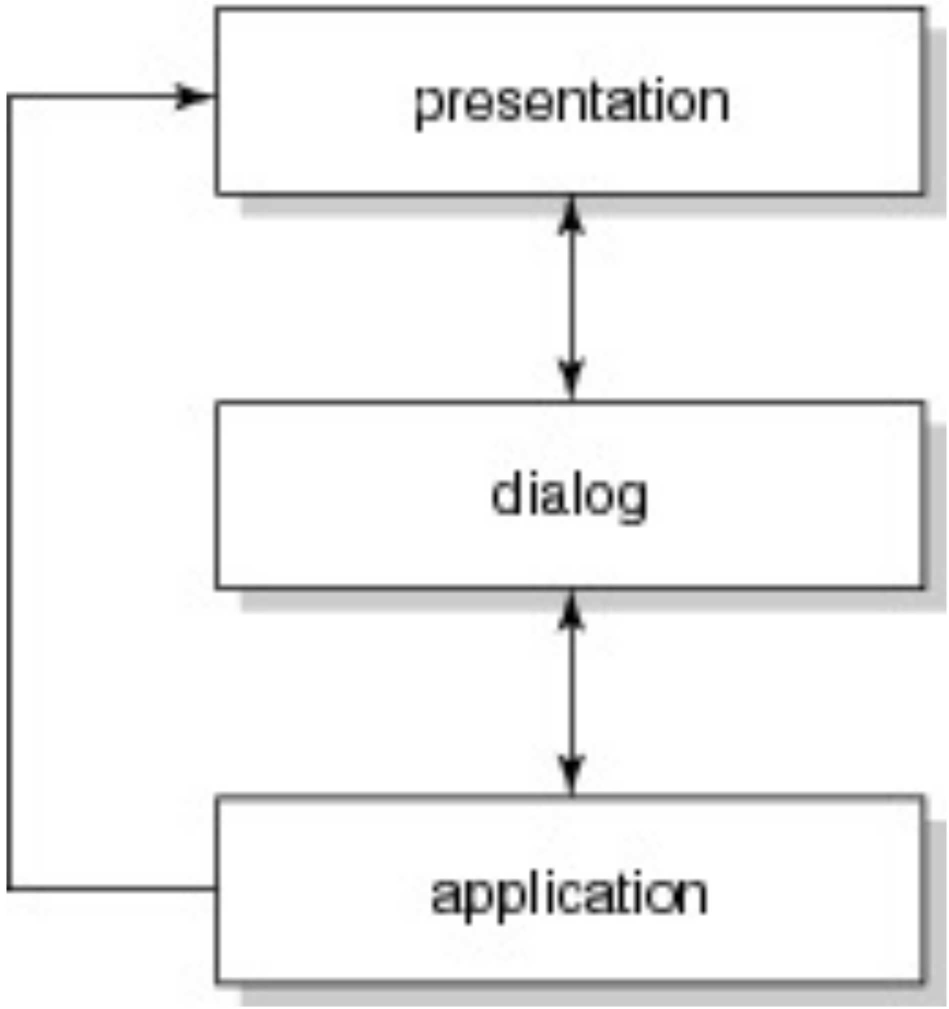
- What *is* the user interface
- How to design a user interface

Where is the user interface?

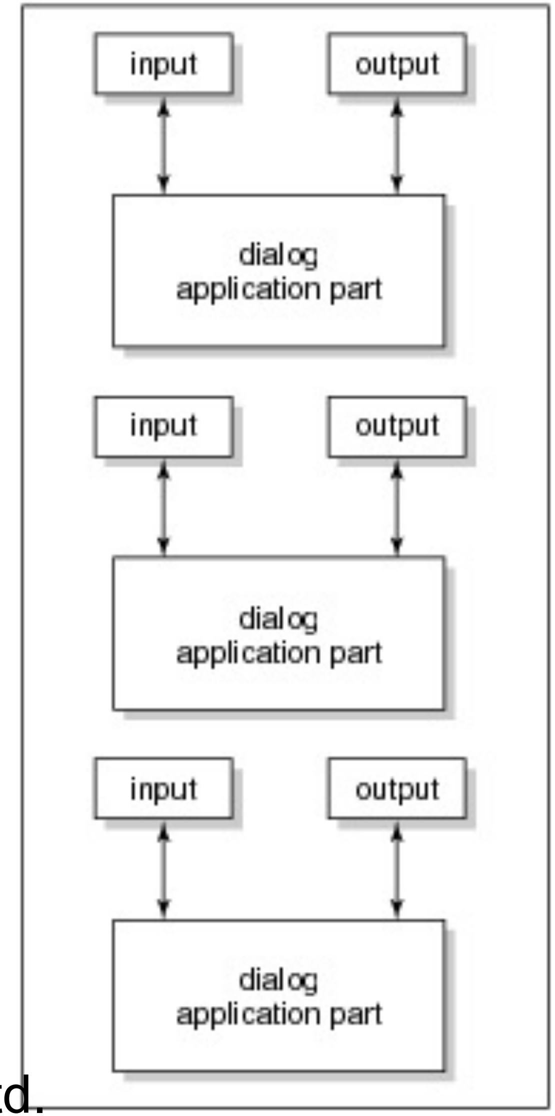


- Seeheim model: separate presentation and dialog from application
- More recently: MVC – Model-View-Controller

Seeheim model



Model-View-Controller (MVC)



What is the user interface?

- User interface: *all* aspects of a system that are relevant to the user
- Also called: *User Virtual Machine* (UVM)

- A system can have more than one UVM, one for each set of tasks or roles
- An individual may also have more than one user interface to the same application, e.g. on a mobile phone and a laptop

Two ways to look at a user interface

- *Design aspect*: how to design everything relevant to the user?
- *Human aspect*: what does the user need to understand?

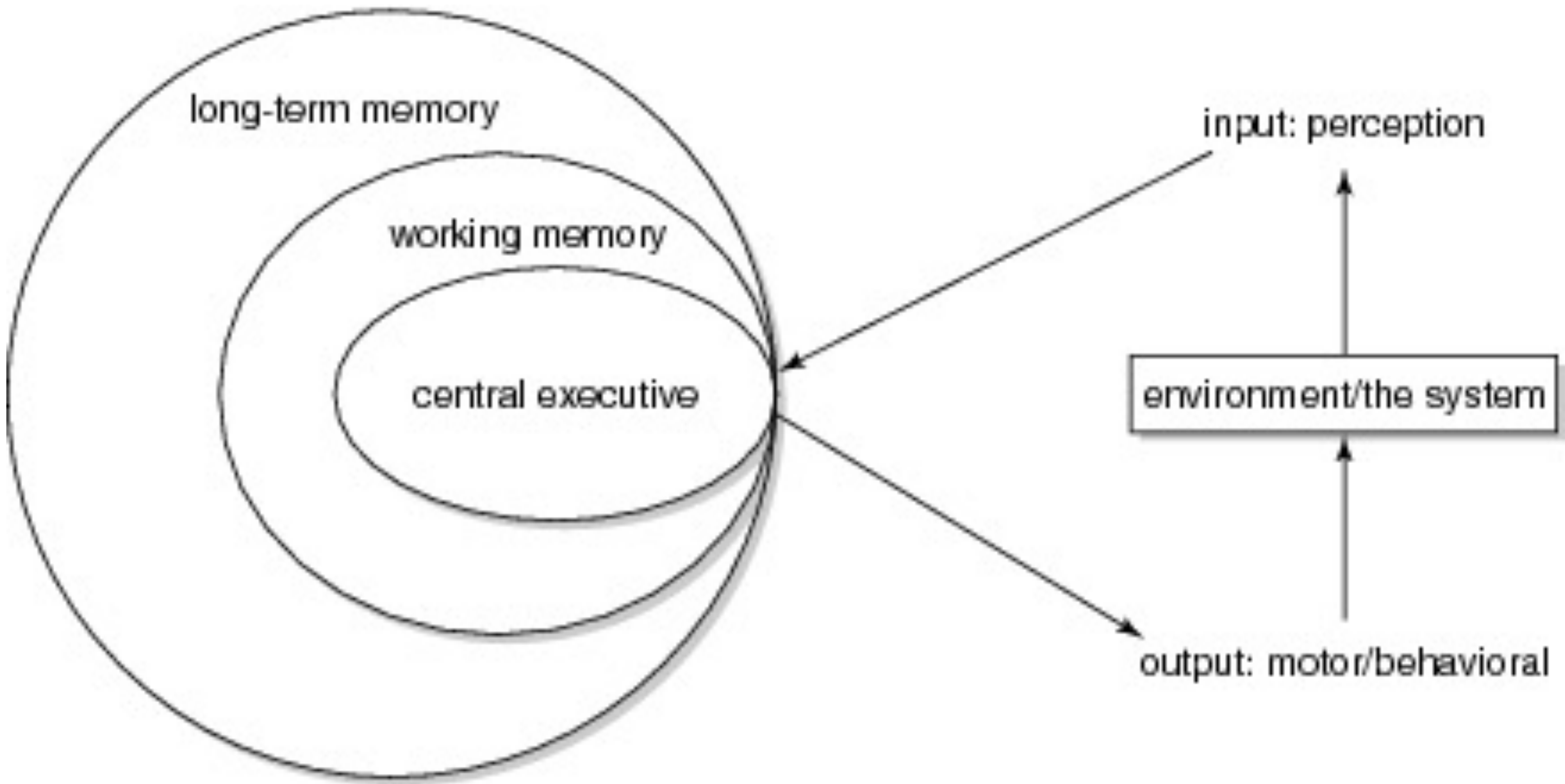
Human factors

- Humanities
  - Psychology: how does one perceive, learn, remember, …
  - Organization and culture: how do people work together, …
- Artistic design
  - Graphical arts: how doe shapes, color, etc affect the viewer
  - Cinematography: which movements induce certain reactions
  - Getting attractive solutions
- Ergonomics
  - Relation between human characteristics and artifacts
  - Especially cognitive ergonomics

Models in HCI

- Internal models ('models for execution')
  - Mental model (model of a system held by a user)
  - User model (model of user held by a system)
- External models ('for communication')
  - Model of human information processing
  - Conceptual models (such as Task Action Grammar)

Model of human information processing



Use of mental models

- Planning the use of technology
  - First search by author name
- Finetuning user actions while executing a task
  - Refine search in case of too many hits
- Evaluate results
  - Keep the titles on software engineering
- Cope with events while using the system
  - Accept slow response time in the morning

Characteristics of mental models (Norman)

- They are incomplete
- They can only partly be ‘run’
- They are unstable
- They have vague boundaries
- They are parsimonious
- They have characteristics of superstition

Conceptual model

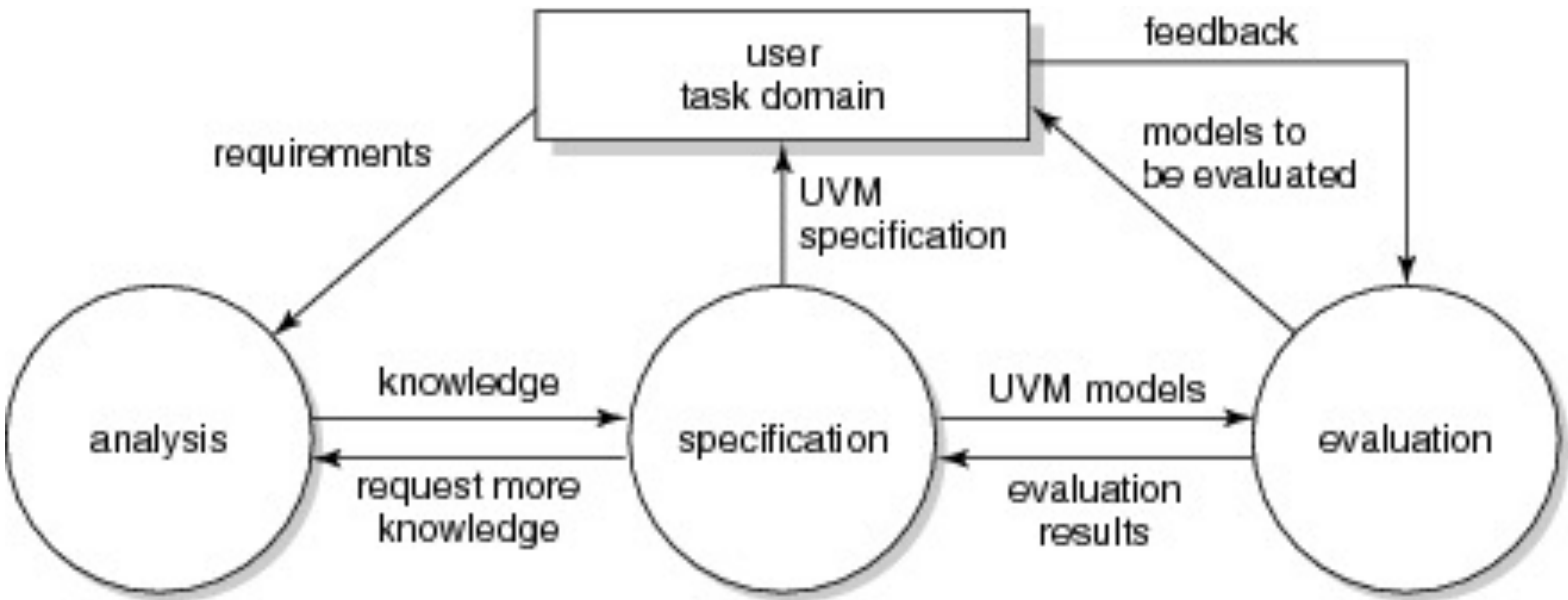
- All that is modeled as far as it is relevant to the user

- Formal models
  - Some model the user's knowledge (competence model)
  - Others focus on the interaction process
  - Others do both

Viewpoints of conceptual models

- Psychological view: definition of all the user should know and understand about the system
- Linguistic view: definition of the dialog between the user and the system
- Design view: all that needs to be decided upon from the point of view of user interface design

Design of the user interface



Dimensions of task knowledge

		Sources of knowledge	
		individual	group
Levels of communicability	explicit	A	C
	implicit	B	D

Gathering task knowledge (cnt'd)

- Cell A (individual, explicit): interviews, questionnaires, etc
- Cell B (individual, implicit): observations, interpretation of mental representations
- Cell C (group, explicit): study artifacts: documents, archives, etc
- Cell D (group, implicit): ethnography

Guidelines for user interface design

- Use a simple and natural dialog
- Speak the user's language
- Minimize memory load
- Be consistent
- Provide feedback
- Provide clearly marked exits
- Provide shortcut
- Give good error messages

Summary

- Central issue: tune user's mental model (model in memory) with the conceptual model (model created by designers)

- User interface design requires input from different disciplines: cognitive psychology, ethnography, arts, …

Global Software Development

Main issue:

- distance matters

Collocated versus global/multisite

- Collocated: housed within walking distance
  - People reinvent the wheel if they have to walk more than 30 meters, or climb the stairs
- Main question: how to overcome distance in global projects:
  - Communication
  - Coordination
  - Control

Arguments for global software development

- Cost savings
- Faster delivery (“follow the sun”)
- Larger pool of developers
- Better modularization
- Little proof that these advantages materialize

Challenges

	distance		
	temporal	geographical	sociocultural
communication	X	X	X
coordination	X	X	
control	X		X

Temporal distance challenges

- Communication:
  - Being effective (asynchronous is less effective, misunderstandings, …)
- Coordination:
  - Cost is larger (travels, infrastructure cost, …)
- Control:
  - Delays (wait for next teleconference meeting, send email and wait, search for contact, …)

Geographical distance challenges

- Communication:
  - Effective information exchange (less informal exchange, different languages, different domain knowledge, …)
  - Build a team (cohesiveness, “them and us” feelings, trust, …)
- Coordination:
  - Task awareness (shared mental model, …)
  - Sense of urgency (perception, …)
- Control:
  - Accurate status information (tracking, blaming, …)
  - Uniform process (different tools and techniques, …)

Geographical distance: awareness

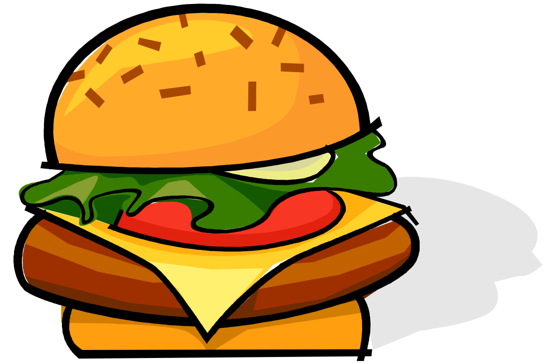
- Activity awareness:
  - What are the others doing?
- Availability awareness:
  - When can I reach them?
- Process awareness:
  - What are they doing?
- Perspective awareness:
  - What are the others thinking, and why?

- Improving awareness and familiarity with other members helps!

Sociocultural distance challenges

- Communication:
  - Cultural misunderstandings (corporate, technical, national, …)
- Coordination:
  - Effectiveness (vocabulary, communication style, …)
- Control:
  - Quality and expertise (CMM level 5 does not guarantee quality)

National culture



- American managers have a hamburger style of management. They start with sweet talk – the top of the bun. Then the criticism is slipped in – the meat. Finally, some encouraging words – the bottom bun.

- With the Germans, all one gets is the meat.

- With the Japanese, all one gets is the bun; one has to smell the meat.

Hofstede’s dimensions

- Power distance
  - status is important versus individuals are equal
- Collectivism versus individualism
  - Individuals are part of a group, or everyone looks after himself
- Femininity versus masculinity
  - Earnings, challenges, recognition (masculine) versus good relationships, cooperation, security (feminine)
- Uncertainty avoidance
  - Strict rules that mitigate uncertainty versus more flexible
- Long-term versus short-term orientation
  - Persistence in pursuing goals, order (LT) versus protecting one’s face, tradition (ST)

Power distance

- North America, Europe: managers have to convince their team members
- Asia: people respect authority

Collectivism versus individualism

- Asia: personal relationships are more important than the task at hand
- North-America, Europe: very task-oriented
- IDV (Individualism Index) differs

Uncertainty avoidance

- Low uncertainty avoidance (UAI): can better cope with uncertainty: they can deal with agile approaches, il-defined requirements, etc.
- High uncertainty avoidance: favor waterfall, contracts, etc.

- Latin America, Japan: high UAI
- North America, India: low UAI

How to overcome distance?

- Common ground
- Coupling of work
- Collaboration readiness
- Technology readiness

Common ground

- How much common knowledge members have, and are aware of
- Common ground has to be established:
  - Traveling, especially at start of project
  - Socialization (kick-off meetings)
- Intense interaction is more important for success than CMM level

Coupling of work

- Tasks that require much collaboration: at same site
- Little interaction required: different sites
  - E.g., testing or implementing relatively independent subsystems

Collaboration readiness

- Transition to global development organization:
  - Requires changing work habits
  - Learning new tools
  - Needs incentives for individuals to cooperate

Technology readiness

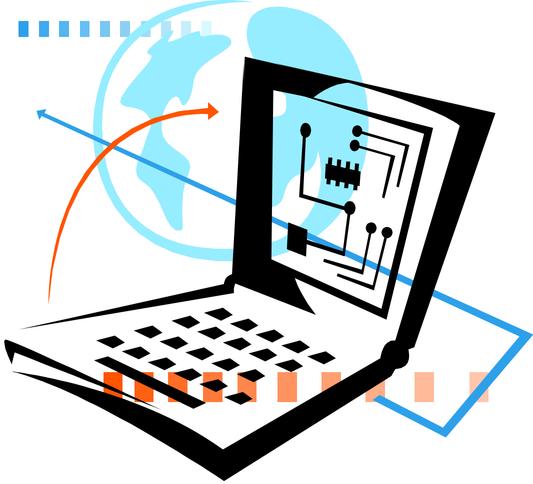
- Project management tools (workflow management)
- Web-enabled versions of tools
- Remote control of builds and tests
- Web-based project repositories
- Real-time collaboration tools (simple media for simple messages, rich media for complex ones)
- Knowledge management technology (codification AND personalization)

Organizing work in global software development

- Reduce the need for informal communication
  - Usually through organizational means, e.g.:
    - Put user interface people together
    - Use gross structure (architecture) to divide work (Conway's Law)
    - Split according to life cycle phases

- Provide technologies that ease informal communication

Summary



- Distance matters
- Main challenges:
  - Deal with lack of informal communication
  - Handle cultural differences

Service Orientation

Main issues:

- What’s special about services?
- Essentials of service-oriented SE

Overview

- Services, service description, service communication
- Service-Oriented Architecture (SOA)
- Web services
- SOSE: Service-Oriented Software Engineering

Italian restaurant analogy

- Restaurant provides food: a service
- After the order is taken, food is produced, served, …: service may consist of other services
- The menu indicates the service provided: a service description
- The order is written down, or yelled at, the cook: services communicate through messages

Main ingredients

- Services
- Service descriptions
- Messages

- Implementation: through web services

Other example

- Citizen looking for a house:
  - Check personal data $\Rightarrow$ System X
  - Check tax history $\Rightarrow$ System Y
  - Check credit history $\Rightarrow$ System Z
  - Search rental agencies $\Rightarrow$ System A,B
  - …

What’s a service

- Platform-independent computational entity that can be used in a platform-independent way
- Callable entities or application functionalities accessed via exchange of messages
- Component capable of performing a task
- Often just used in connection with something else: SOA, Web services, …

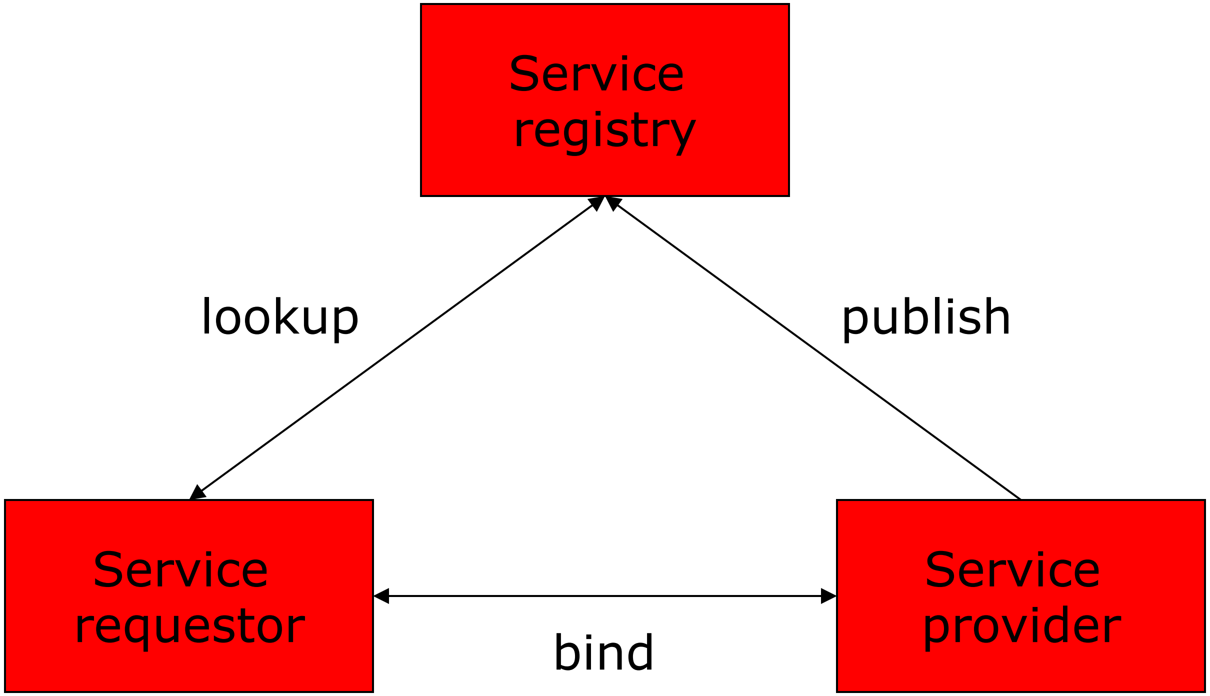
What’s a service, cnt’d

- Shift from producing software to using software
  - You need not host the software
  - Or keep track of versions, releases
  - Need not make sure it evolves
  - Etc
- Software is “somewhere”, deployed on as-needed basis
- SaaS: Software as a Service

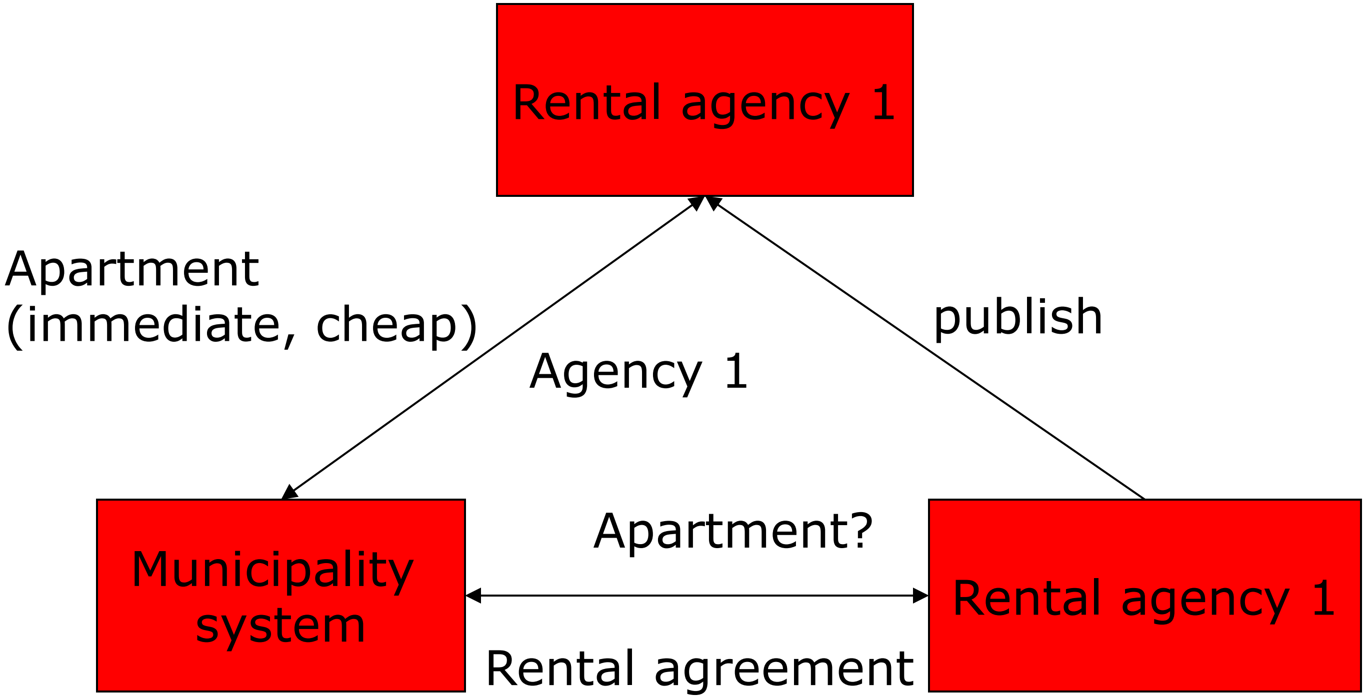
Key aspects

- Services can be discovered
- Services can be composed to form larger services
- Services adhere to a service contract
- Services are loosely coupled
- Services are stateless
- Services are autonomous
- Services hide their logic
- Services are reusable
- Services use open standards
- Services facilitate interoperability

Service discovery



Service discovery



Service discovery

- Discovery is dynamic, each invocation may select a different one
- Primary criterion in selection: contract
- Selection may be based on workload, complexity of the question, etc $\Rightarrow$ optimize compute resources
- If answer fails, or takes too long $\Rightarrow$ select another service $\Rightarrow$ more fault-tolerance

Is discovery really new?

- Many design patterns loosen coupling between classes
- Factory pattern: creates object without specifying the exact class of the object.

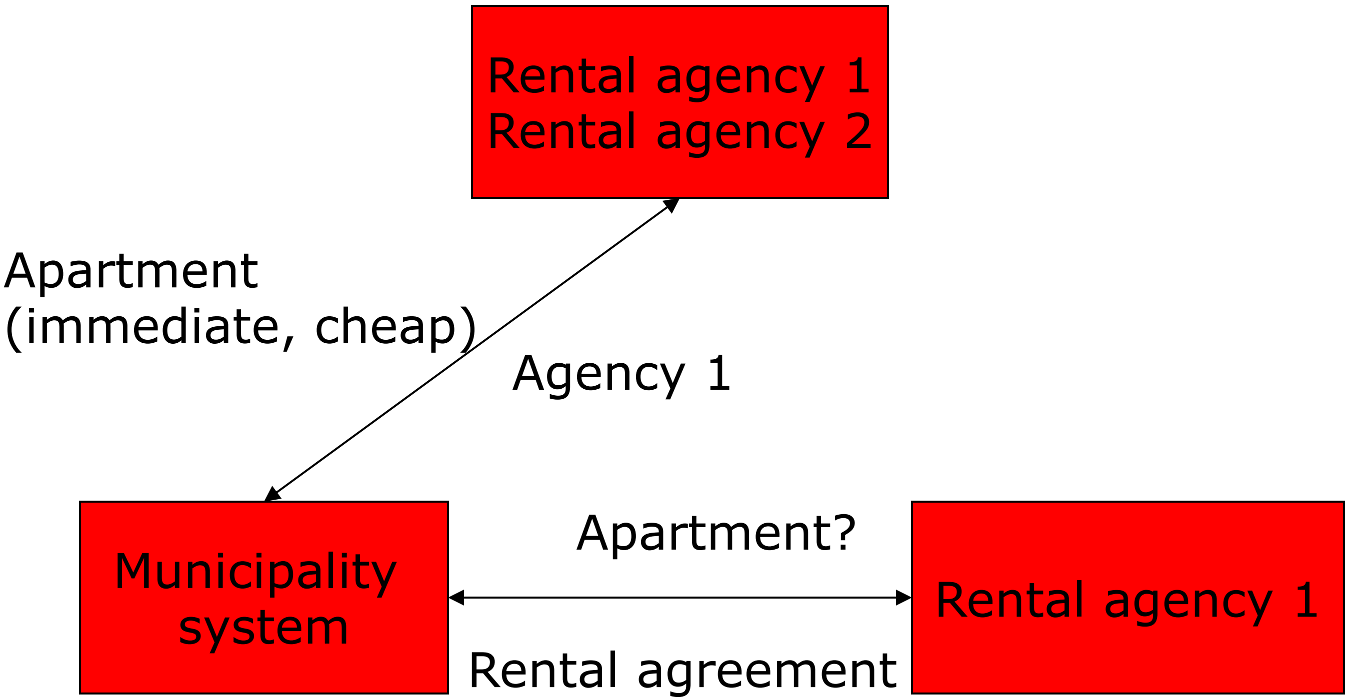
Services can be composed

- Service can be a building block for larger services
- Not different from CBSE and other approaches

Services adhere to a contract

- Request to registry should contain everything needed, not just functionality
- For “normal” components, much is implicit:
  - Platform characteristics
  - Quality information
  - Tacit design decisions
- Trust promises?
- Quality of Services (QoC), levels thereof
- Service Level Agreement (SLA)

Service discovery



Services are loosely coupled

- Rental agencies come and go
- No assumptions possible
- Stronger than CBSE loose coupling

Services are stateless

- Rental agency cannot retain information: it doesn't know if and when it will be invoked again, and by whom

Services are autonomous, hide their logic

- Rental agency has its own rules on how to structure its process
- Its logic does not depend on the municipality service it is invoked by
- This works two ways: outside doesn’t know the inside, and vice versa

Services are reusable

- Service models a business process:
  - Not very fine grained
  - Collecting debt status from one credit company is not a service, checking credit status is
- Deciding on proper granularity raises lots of debate

Service use open standards

- Proprietary standards $\Rightarrow$ vendor lockin
- There are lots of open standards:
  – How services are described
  – How services communicate
  – How services exchange data
  – etc

Services facilitate interoperability

- Because of open standards, explicit contracts and loose coupling

- Classical CBSE solutions pose problems:
  – Proprietary formats
  – Platform differences
  – Etc

- Interoperability within an organization (EAI) and between (B2B)

Overview

- Services, service description, service communication
- Service-Oriented Architecture (SOA)
- Web services
- SOSE: Service-Oriented Software Engineering

Overview

- Services, service description, service communication
- Service-Oriented Architecture (SOA)

- Web services

- SOSE: Service-Oriented Software Engineering

Web services

- Implementation means to realize services
- Based on open standards:
  - XML
  - SOAP: Simple Object Access Protocol
  - WSDL: Web Services Description Language
  - UDDI: Universal Description, Discovery and Integration
  - BPEL4WS: Business Process Execution Language for Web Services
- Main standardization bodies: OASIS, W3C

XML

- Looks like HTML
- Language/vocabulary defined in schema: collection of trees
- Only syntax
- Semantic Web, Web 2.0: semantics as well: OWL and descendants

SOAP

- Message inside an envelope
- Envelop has optional header (~address), and mandatory body: actual container of data
- SOAP message is unidirectional: it's NOT a conversation

WSDL

- Four parts:
  - Web service interfaces
  - Message definitions
  - Bindings: transport, format details
  - Services: endpoints for accessing service. Endpoint = (binding, network address)

UDDI

- Three (main) parts:
  - Info about organization that publishes the services
  - Descriptive info about each service
  - Technical info to link services to implementation

UDDI (cnt'd)

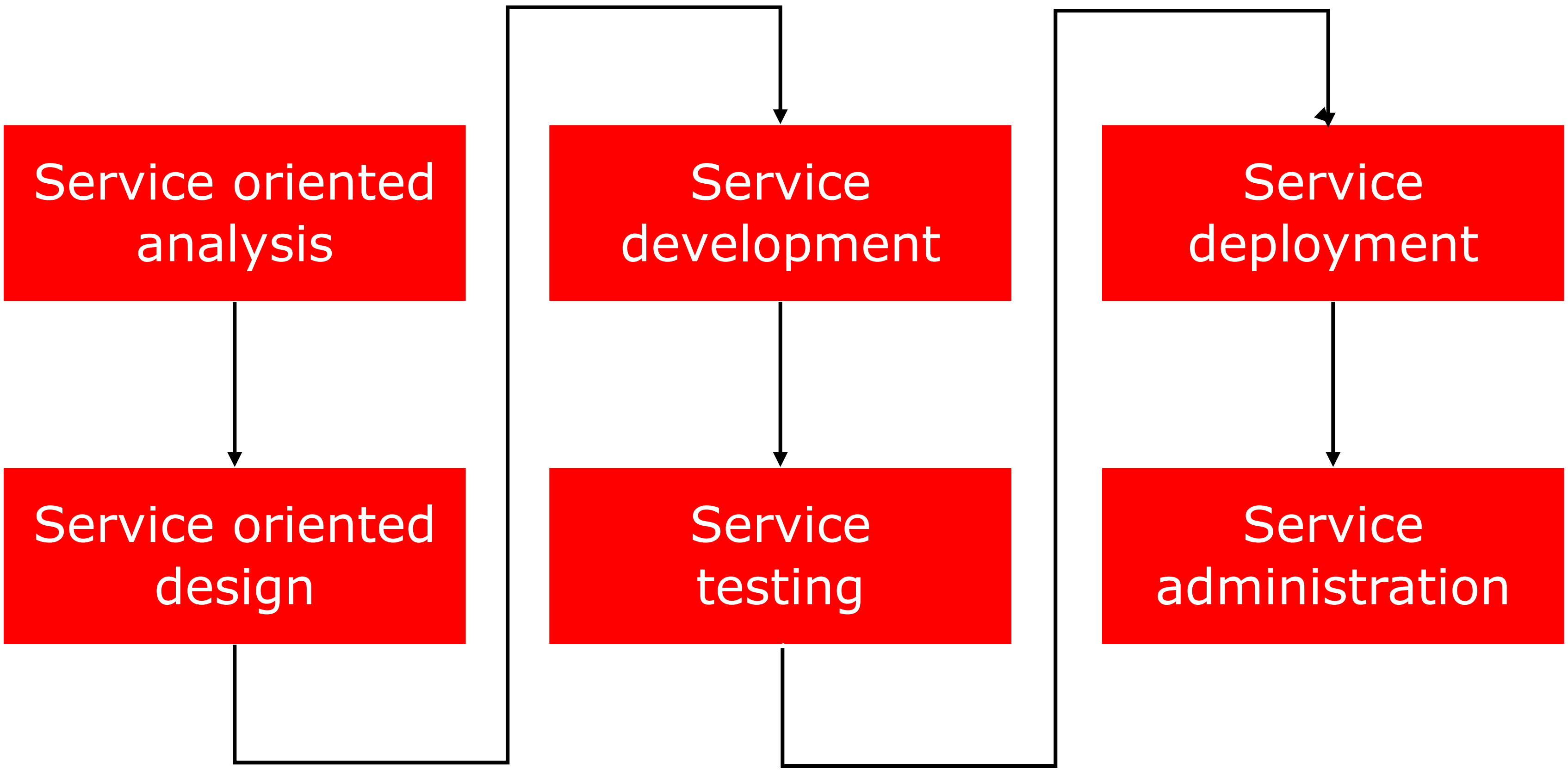
- Original dream: one global registry
- Reality: many registries, with different levels of visibility
  - Mapping problems

Overview

- Services, service description, service communication
- Service-Oriented Architecture (SOA)
- Web services

- SOSE: Service-Oriented Software Engineering

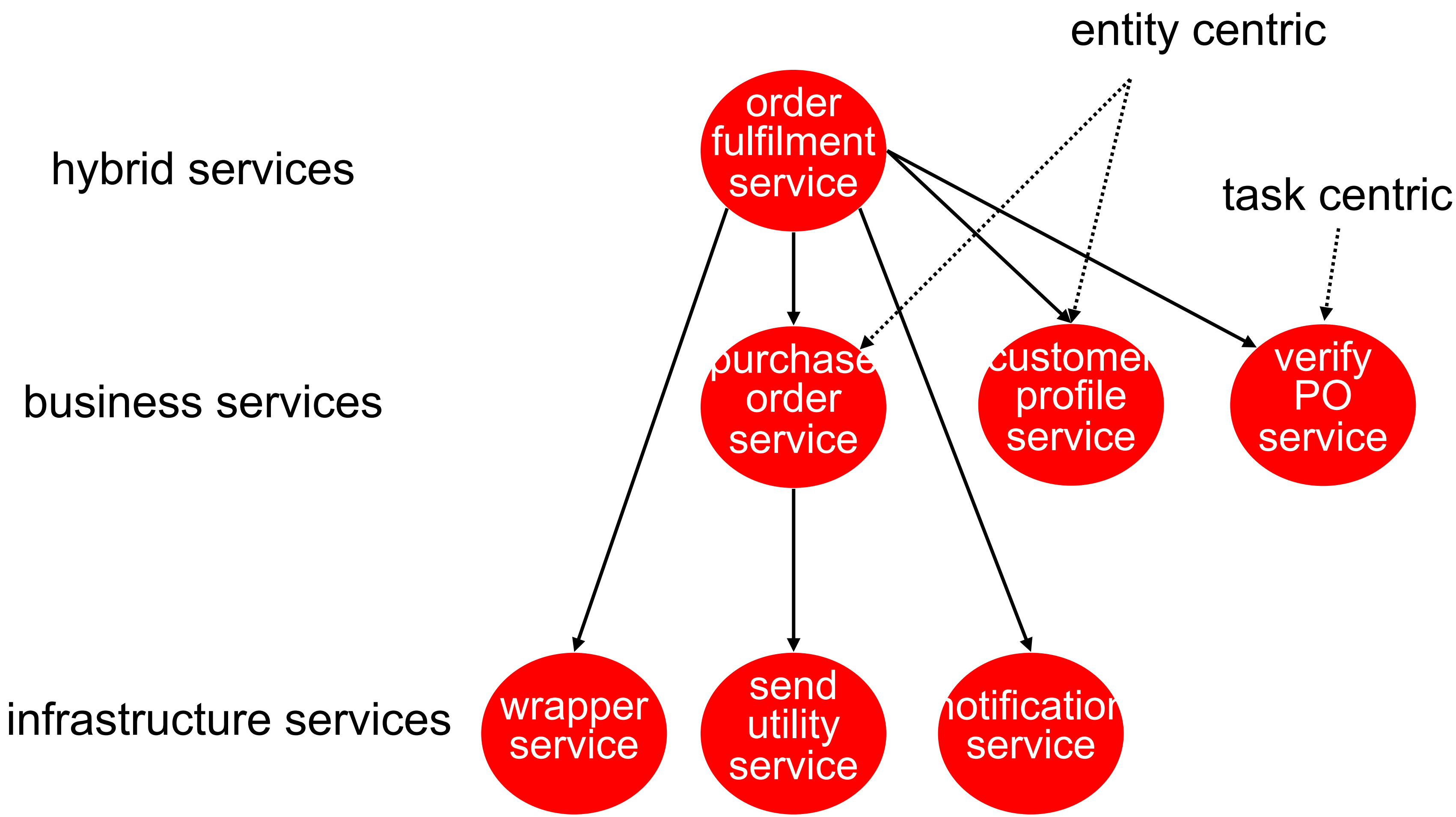
SOSE life cycle



Terminology

- service oriented environment (or service oriented *ecosystem)*
- *business process + supporting services*
  - *application (infrastructure) service*
  - *business service*
    - *Task-centric business service*
    - *Entity-centric business service*
  - *hybrid service*

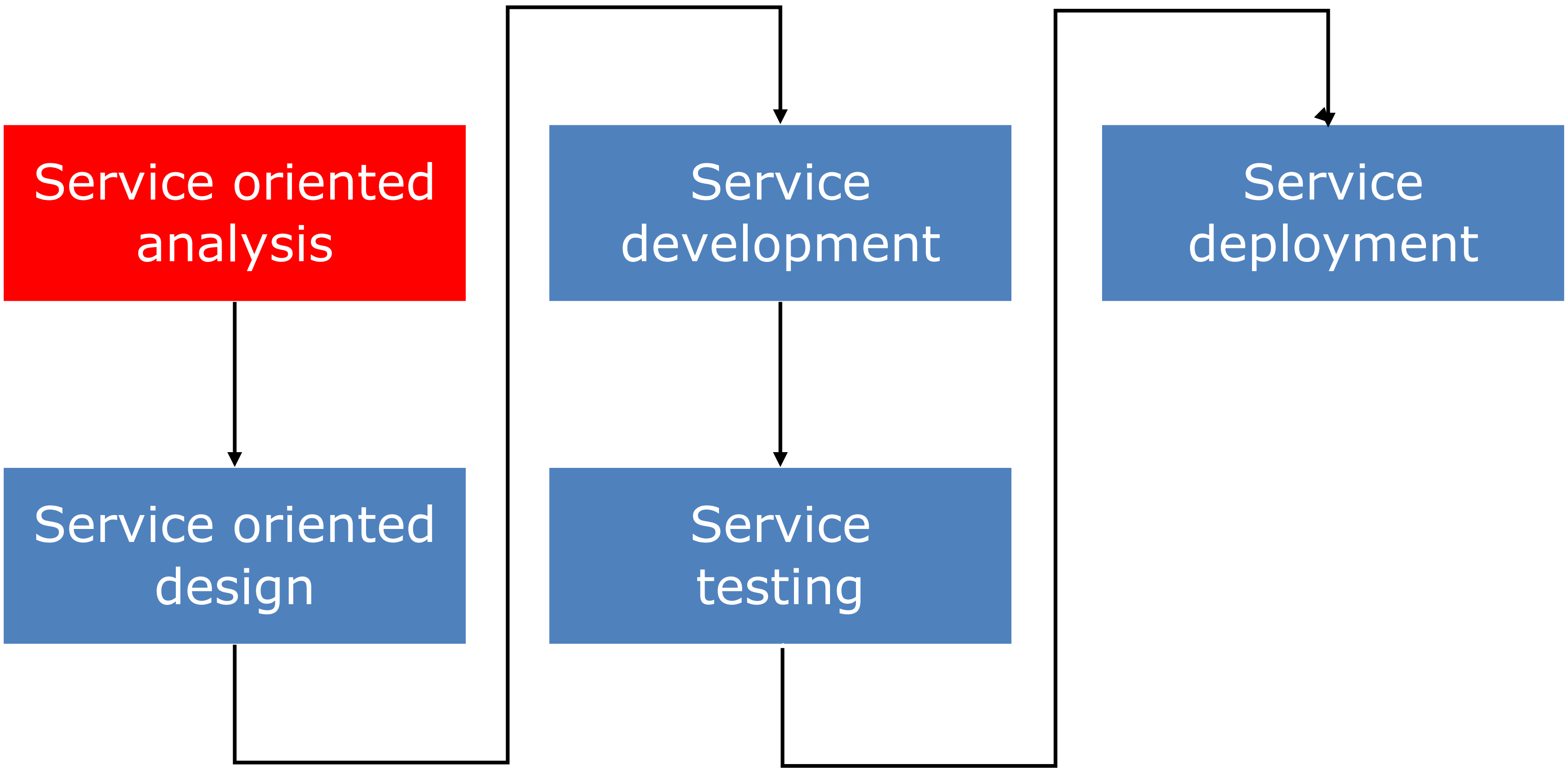
Terminology



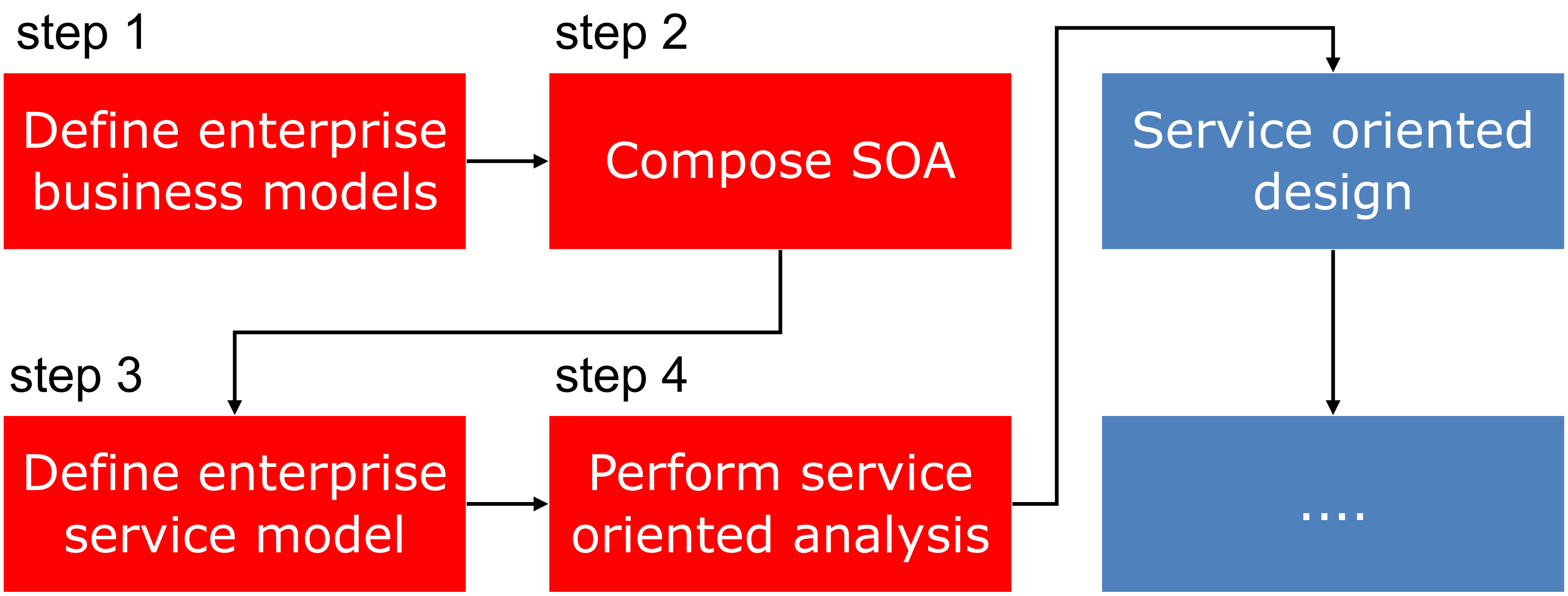
Strategies for life cycle organization

- Top-down strategy
- Bottom-up strategy
- Agile strategy

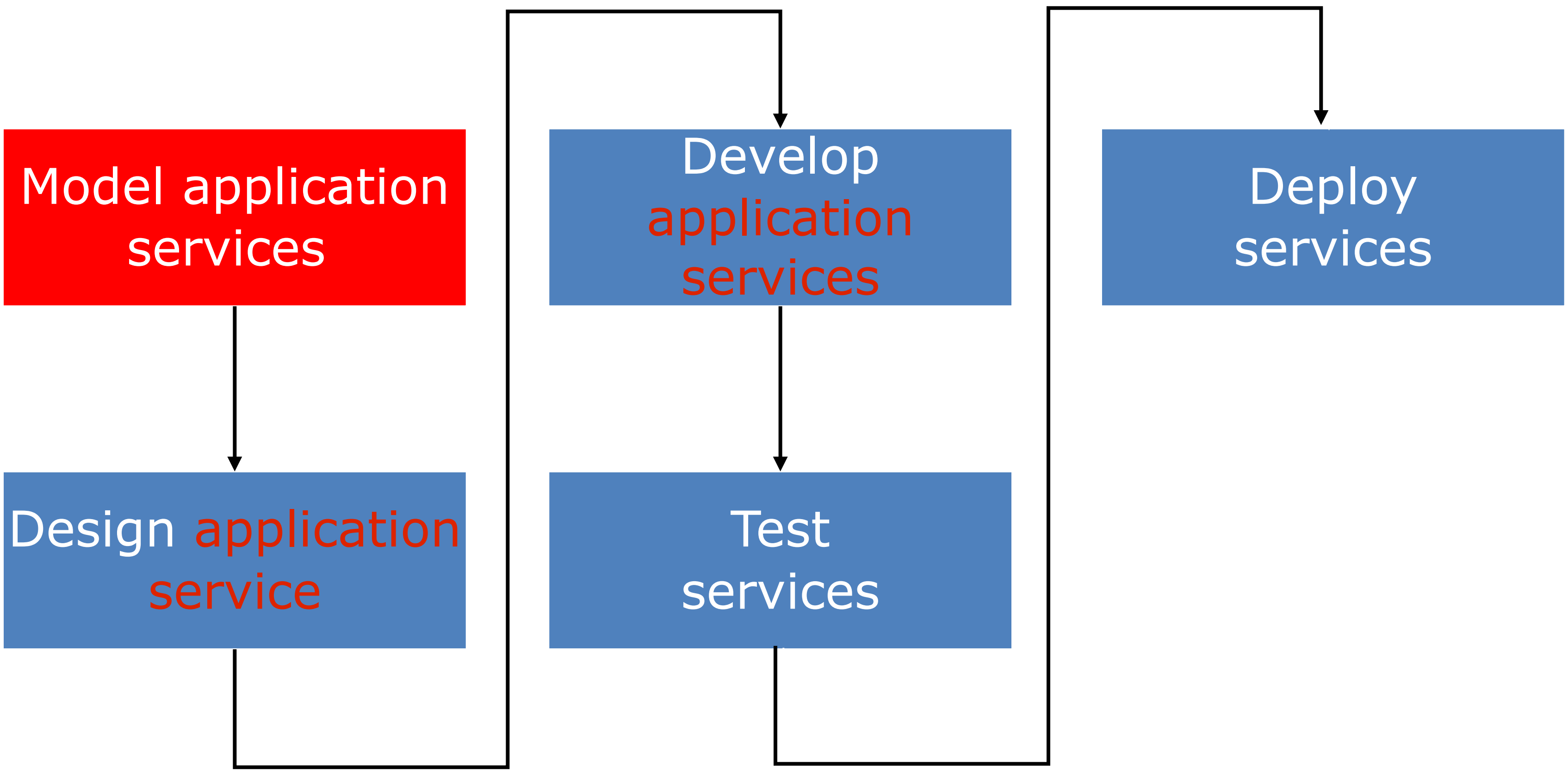
Top-down strategy



Top-down SO analysis



Bottom-up strategy



application service = infrastructure service

Agile strategy

Top-down analysis

on-going

align with current state business models

SO analysis

SO design

Develop services

Test service operations

Deploy services

Revisit business (and process) services

align with current state business models

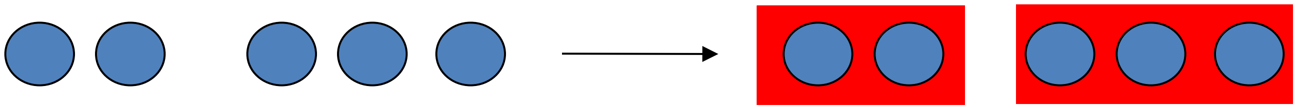
Service oriented analysis

- The process of determining how business automation requirements can be represented through service orientation

Goals of SO analysis

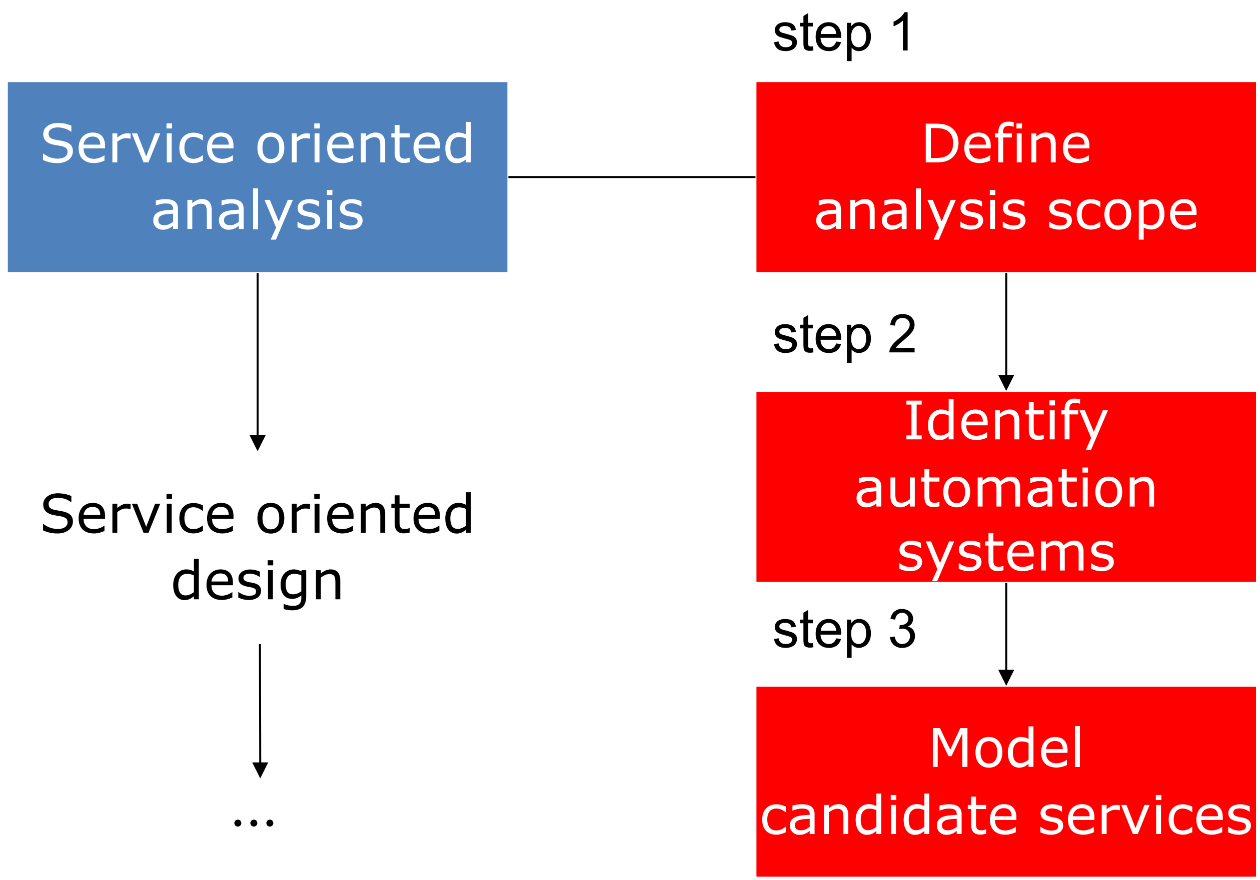
Service operation candidates

Service candidates (logical contexts)



- Appropriateness for intended use
- Identify preliminary issues that may challenge required service autonomy
- Define known preliminary composition models

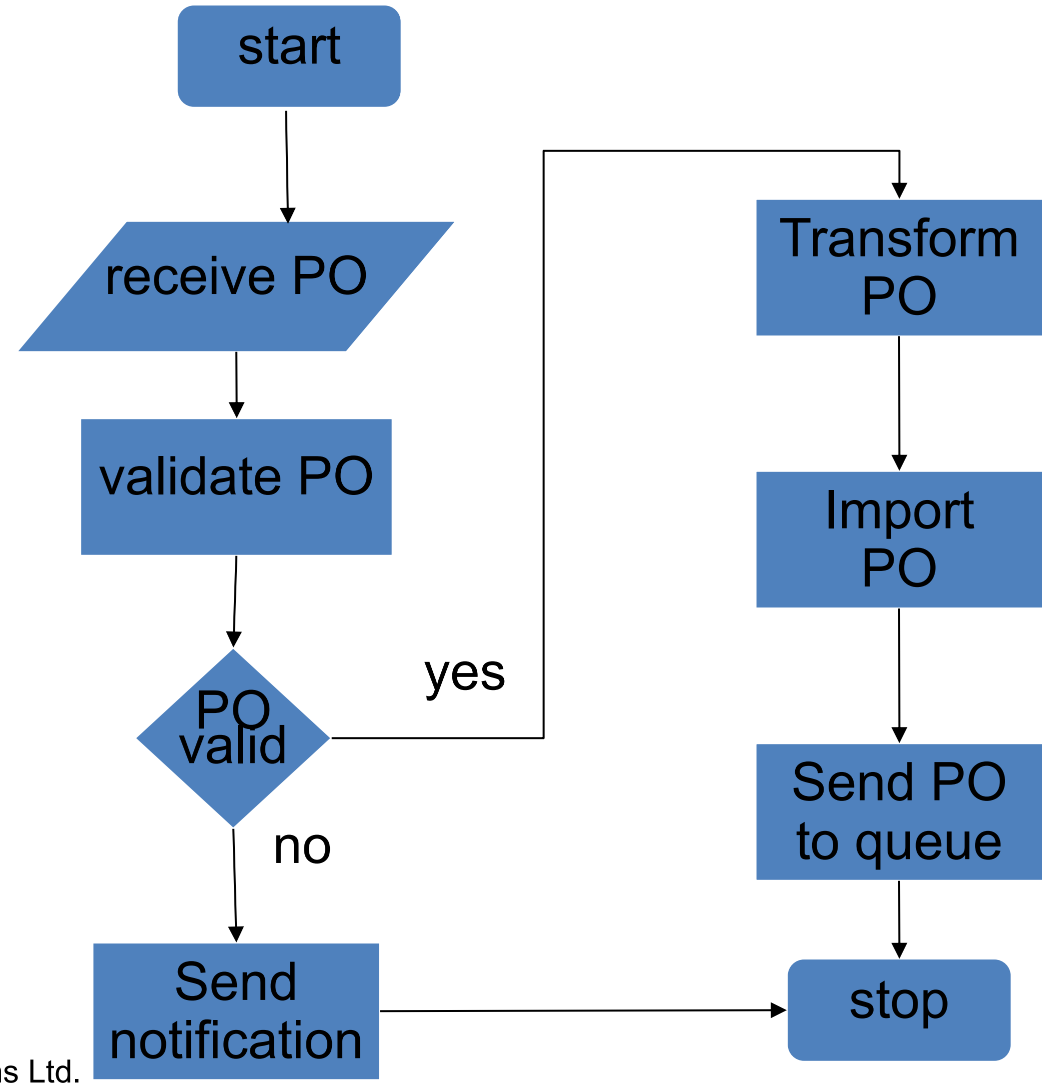
3 Analysis sub-steps



Step 1: Define analysis scope

- Mature and understood business requirements
  - $S = \sum_i S_i$, where smaller services may still be quite complex
- Can lead to
  - process-agnostic services/service operations (generic service portfolio)
  - services delivering business-specific tasks
- Models: UML use case or activity diagrams

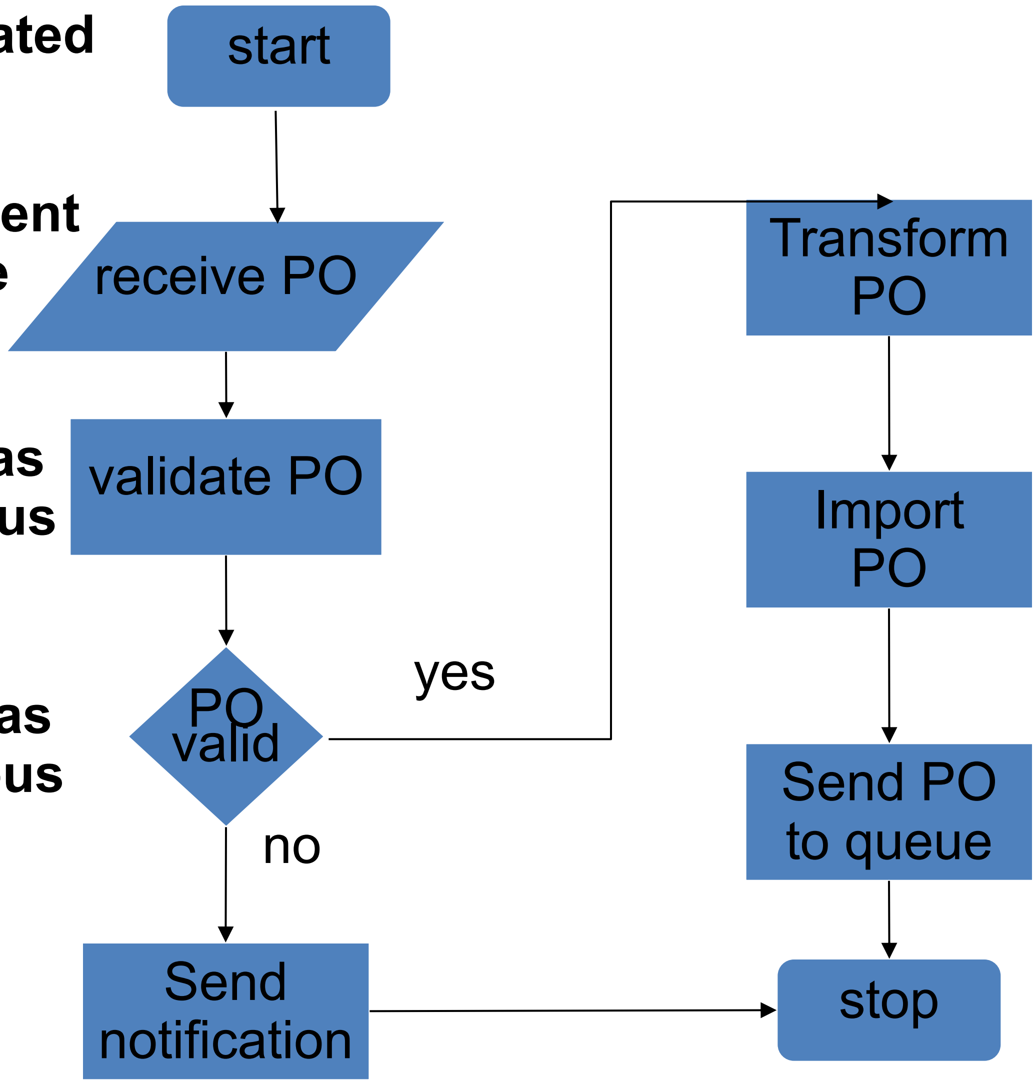
Order Fulfillment Process



Step 2: Identify automation systems

- What is already implemented?
  - encapsulate
  - replace
- Models: UML deployment diagram, mapping tables

Order Fulfillment Process



already automated by Order fulfillment service

same as previous

same as previous

(XML -> native format) (currently custom component)
service candidate

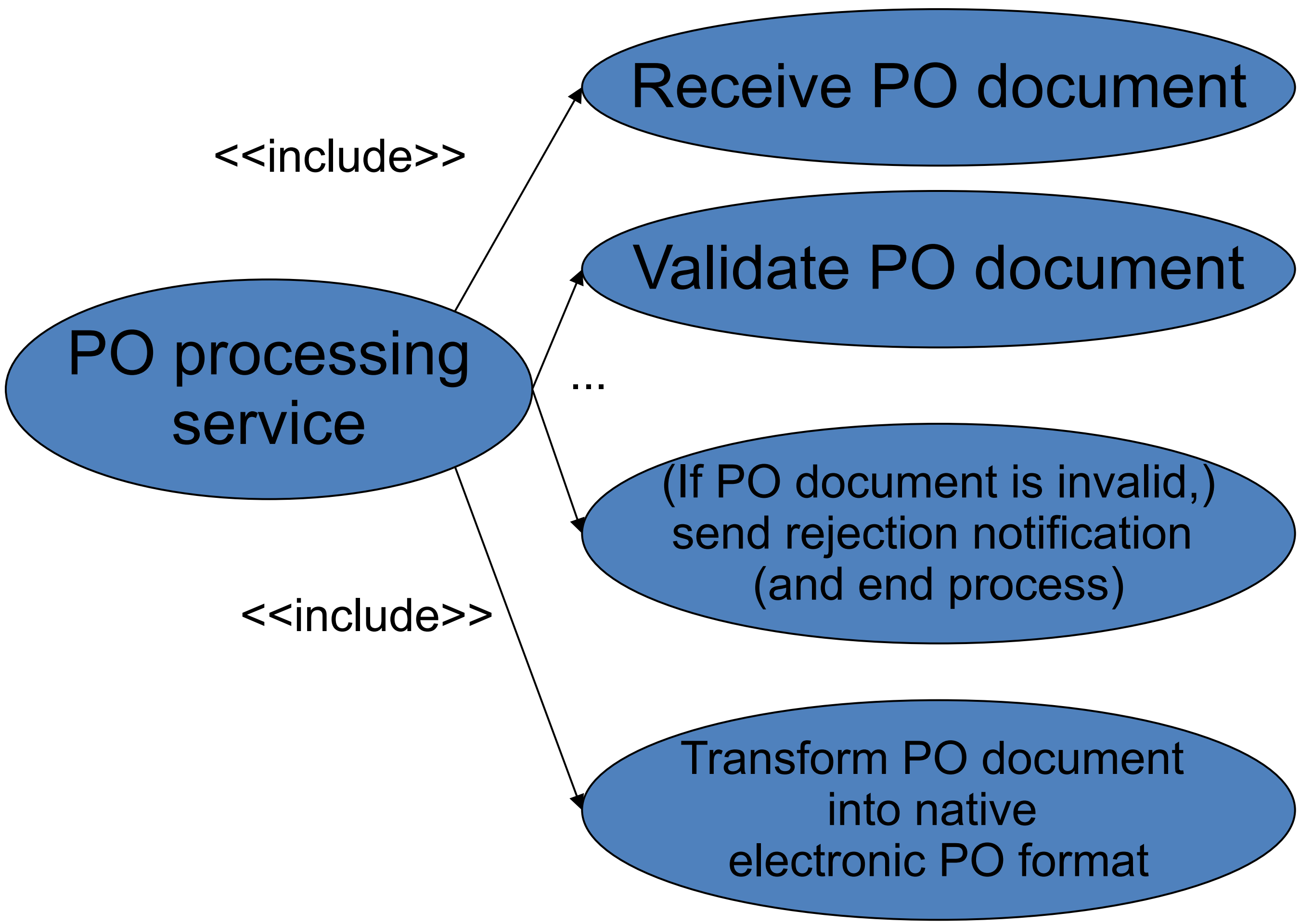
(into accounting sys.) service candidate (currently custom legacy)
service candidate

(to accounting clerk's work queue)
same as previous

Step 3: Model candidate services

- How to compose services?
- Service (candidates) conceptual model
  - operations + service contexts
  - SO principles
- Focus on task- and entity-centred services
- Models: BPM, UML use case or class diag.

Example service operation candidates



Example business process logic

- Not service operation candidates
  - if PO document is valid, proceed with the transform PO document step
  - if the PO document is invalid, end process

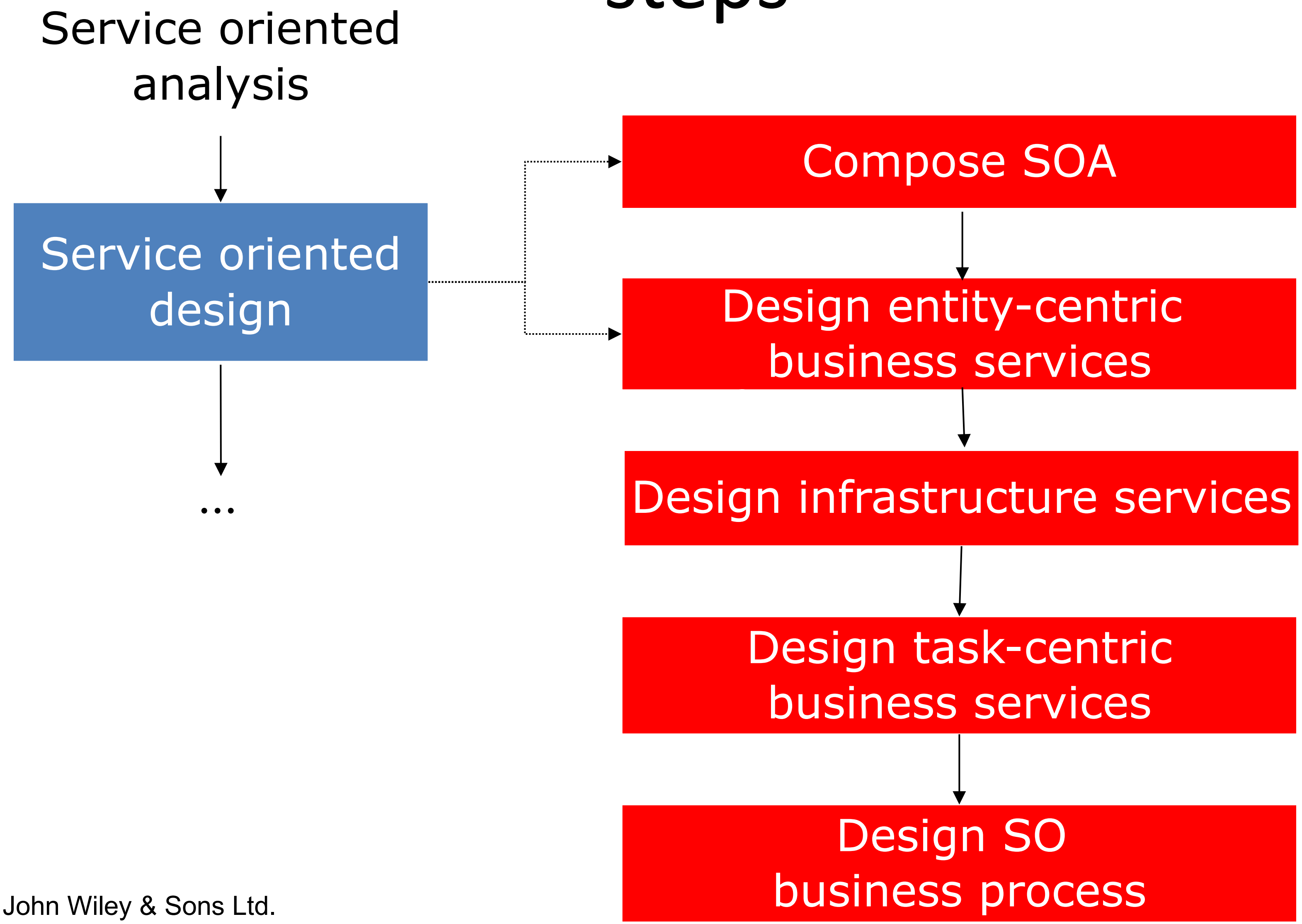
Task- versus entity-centred services

- Task-centred
  - (+) direct mapping of business requirements
  - (-) dependent on specific process
- Entity-centred
  - (+) agility
  - (-) upfront analysis
  - (-) dependent on controllers

Benefits of business-centric SOA

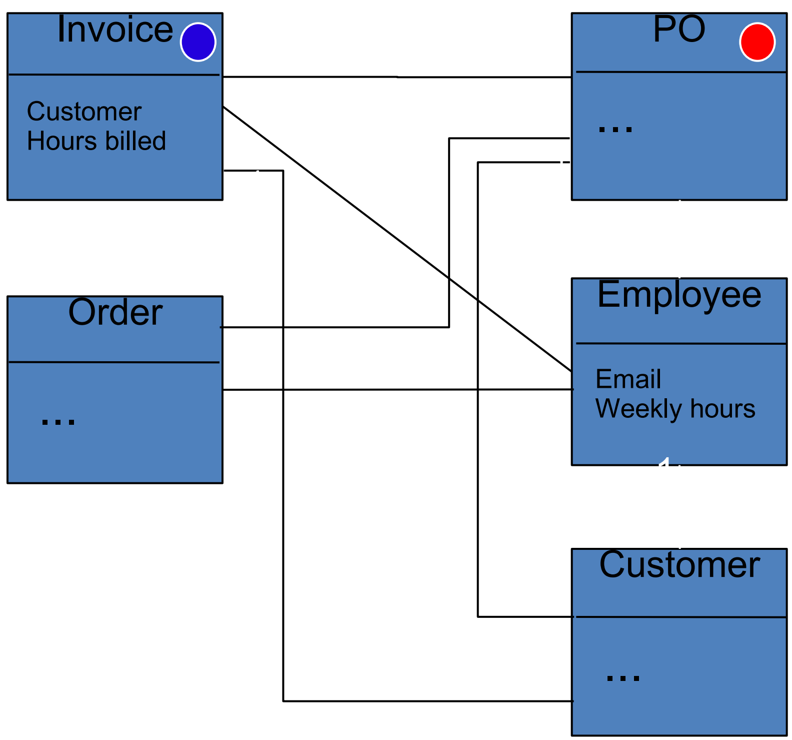
- introduce agility
- prepare for orchestration
- enable reuse

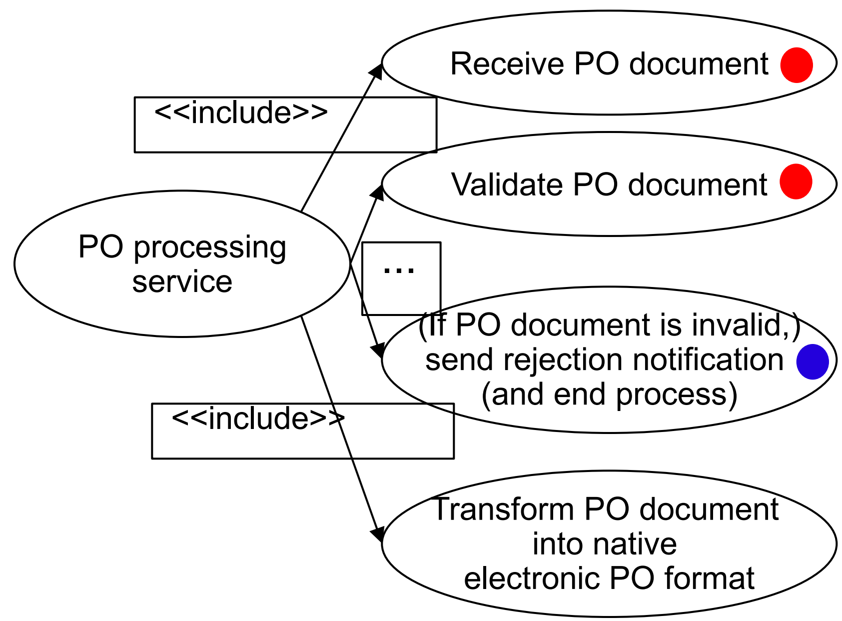
Service-oriented design: design sub-steps



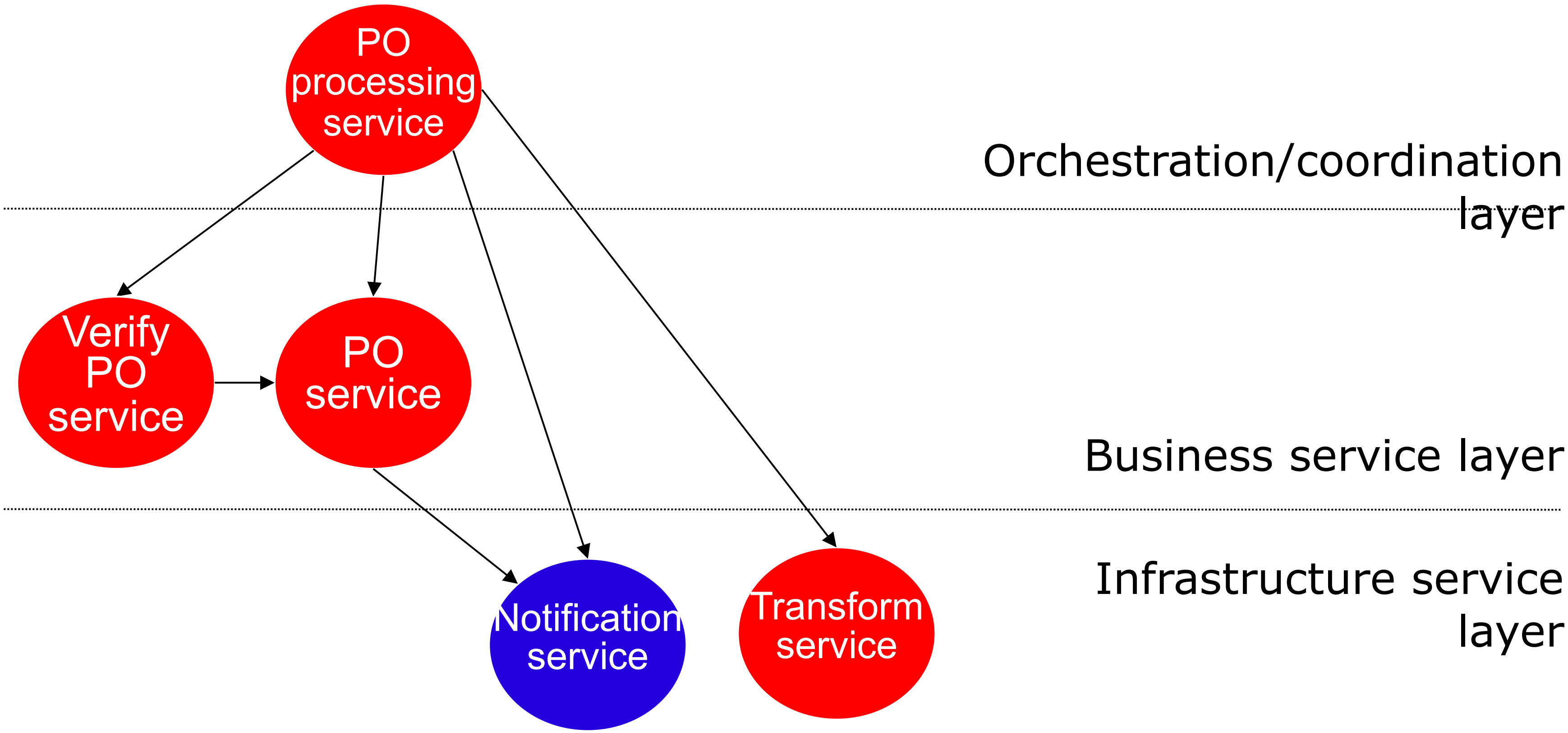
Entity-centric business services

- Goal: entity-centric business service layer + parent orchestration layer



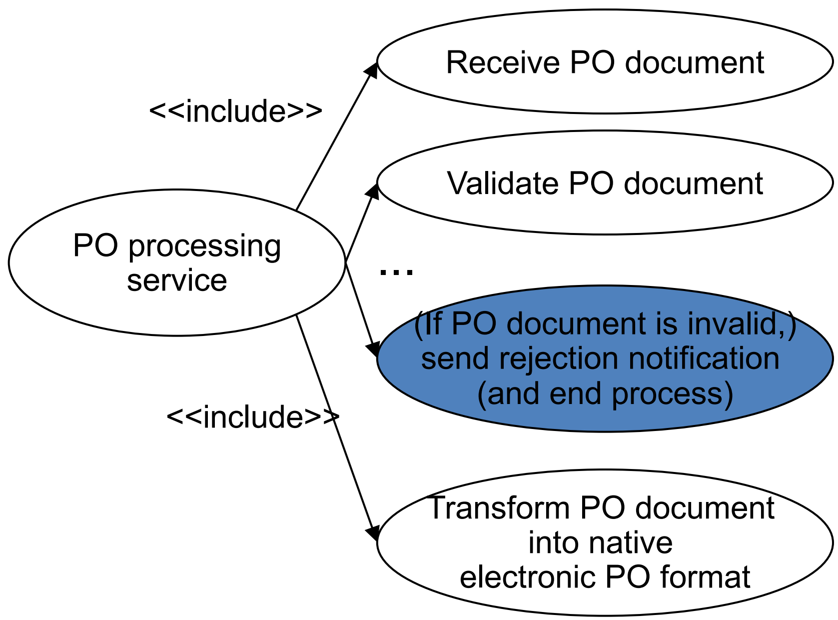


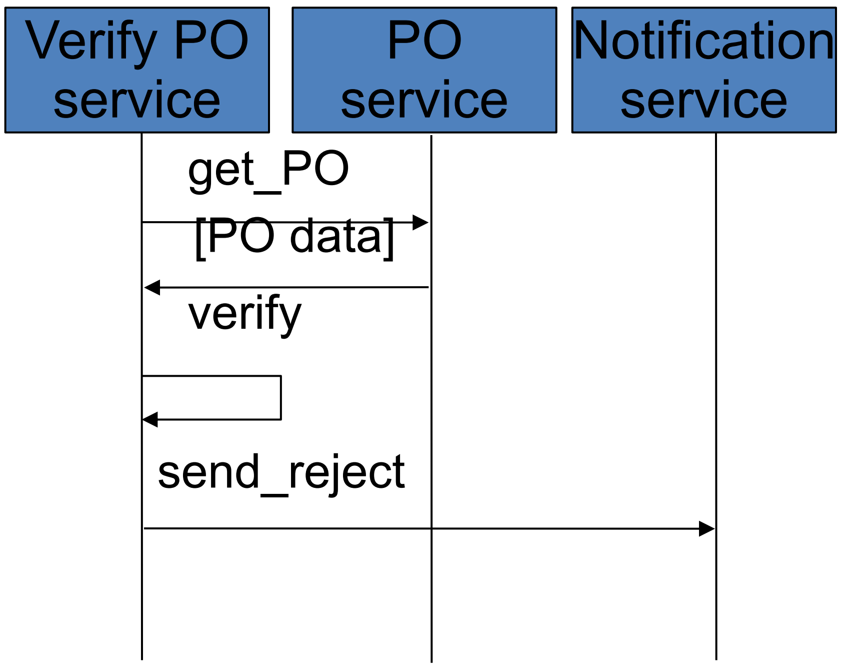
Infrastructure services



Task-centric business services

- UML sequence diagram
  - express and refine order of invocations implicit in the UML use case diagram





Summary

- Services have a long history (telephony)
- Most important characteristic: dynamic discovery of services
- SOA as architectural style
- Today's Web services mostly syntax-based
- Key design decisions in SOSE concern service layering, industry standards, and relevant SO principles
- SOSE differentiates from traditional life cycles mainly in the analysis and design phases