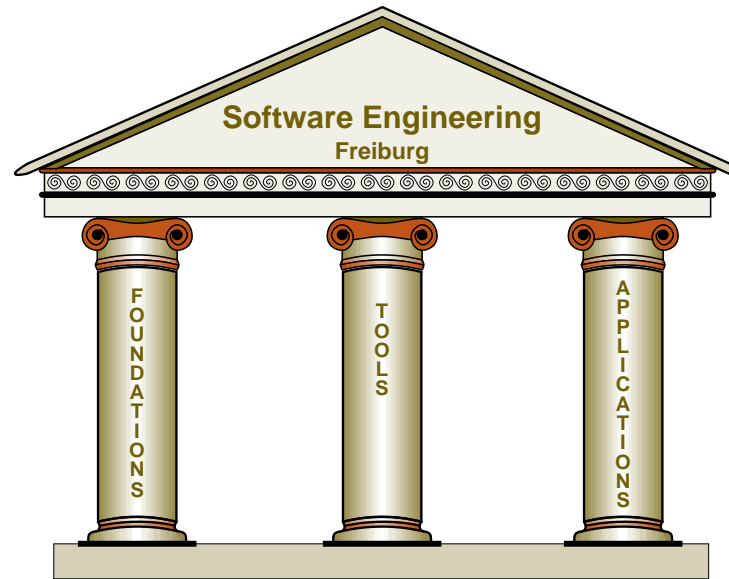


Introduction

David Basin

Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Spring 2002



- Our focus: **formal methods**
Techniques and tools based on mathematics and logic that support the description, construction and analysis of hardware and software systems.
- Classes in SS02
 - Automata Based System Analysis
 - Seminar on Tool and Methods for Automatic Program Generation
 - Oberseminar Softwaretechnik
- For other classes see www.informatik.uni-freiburg.de/~softech
- See also our project page **Hiwi-positions open!**

General administration

- Web site
`www.informatik.uni-freiburg.de/~softech/teaching/ss02/st`
- Class: Tuesday 9-11, Thursday 9-10 (ct)
Exercises: Thursday 10-11. Lead by Luca Viganò and Burkhardt Wolff
- Language: English (ACS)
- Grade based on exercises, project, and final exam.
Software engineering is not a spectator sport!
- Prerequisites
 - Experience programming in the small with Java
 - A semester logic is desirable although not mandatory
 - General computer science (theory, data bases, and algorithms).

Literature

- Sommerville: Software Engineering
- Pagel/Six: Software Engineering
- Ghezzi/Jazzayeri/Mandrioli: Fundamentals of Software Engineering
- Ian Hayes: Specification Case Studies
- See webpages for resources
 - Slides available **after** class!
 - As time permits, we may write supplemental notes.

Today — an overview

1. Why bother with software engineering?
2. What is software engineering?
3. Structuring and abstraction in modeling

Why bother with software engineering?

There are too many system failures! Some humorous. . .

Restaurant orders on-line, computer crash overcooks steaks.¹

Why bother with software engineering?

There are too many system failures! Some humorous. . .

Restaurant orders on-line, computer crash overcooks steaks.¹

A Budd Company assembly robot has apparently committed suicide. The robot was programmed to apply a complex bead of fluid adhesive, but the robot ignored the glue, picked up a fistful of highly-active solvent, and shot itself in its electronics-packed chest.²

Why bother with software engineering?

There are too many system failures! Some humorous. . .

Restaurant orders on-line, computer crash overcooks steaks.¹

A Budd Company assembly robot has apparently committed suicide. The robot was programmed to apply a complex bead of fluid adhesive, but the robot ignored the glue, picked up a fistful of highly-active solvent, and shot itself in its electronics-packed chest.²

One of the first things the Air Force test pilots tried on an early F-16 was to tell the computer to raise the landing while still standing on the runway. Guess what happened? Scratch one F-16.³

¹ACM SIGSOFT, Software Engineering Notes 12(2), 1987

²ACM SIGSOFT, Software Engineering Notes 13(3), 1988

³ACM SIGSOFT, Software Engineering Notes 11(5), 1986

Why bother with software engineering? (cont.)

Others are frustrating and expensive

A Norwegian Bank was embarrassed yesterday after a cashpoint computer applied its own form of 'fuzzy logic' and handed out thousands of pounds no one had asked for. A long queue formed at the Oslow cashpoint after news spread that customers were receiving 10 times what they requested.⁴

Why bother with software engineering? (cont.)

Others are frustrating and expensive

A Norwegian Bank was embarrassed yesterday after a cashpoint computer applied its own form of 'fuzzy logic' and handed out thousands of pounds no one had asked for. A long queue formed at the Oslow cashpoint after news spread that customers were receiving 10 times what they requested.⁴

In early 1997, after many years, \$4 billion spent, extensive criticism from the General Accounting Office [...] the IRS abandoned its tax systems modernization effort. The FBI abandoned development of a \$500-million new fingerprint-on-demand computer system. The State of California spent \$1 billion on a nonfunctional welfare database system. The Assembly Information Technology Committee was considering scrapping California's Automated Child Support System, which had already overrun its \$100 million budget by more than 200%.⁵

Why bother with software engineering? (cont.)

Others are frustrating and expensive

A Norwegian Bank was embarrassed yesterday after a cashpoint computer applied its own form of 'fuzzy logic' and handed out thousands of pounds no one had asked for. A long queue formed at the Oslow cashpoint after news spread that customers were receiving 10 times what they requested.⁴

In early 1997, after many years, \$4 billion spent, extensive criticism from the General Accounting Office [...] the IRS abandoned its tax systems modernization effort. The FBI abandoned development of a \$500-million new fingerprint-on-demand computer system. The State of California spent \$1 billion on a nonfunctional welfare database system. The Assembly Information Technology Committee was considering scrapping California's Automated Child Support System, which had already overrun its \$100 million budget by more than 200%.⁵

The costs of the year 2000 problem has been estimated to be over \$600 billion worldwide.⁶

⁴ACM SIGSOFT, Software Engineering Notes 15(3), 1990

⁵INSIDE RISKS, Communications of the ACM 40, 12, Dec 1997

⁶RISKS, 1998.

Why bother with software engineering? (cont.)

Others are absolutely not acceptable!

The automatic propulsion control in our Boeing 737 had the occasional habit during take-off at exactly 60 knots of cutting out. It was someone in our workshops who looked at our listings found the cause. The programmer had spelled out what the propulsion control should do under 60 knots and what it should do over 60 knots. But he had forgotten to say how it should react at exactly 60 knots. So if at exactly 60 knots the computer asked for the appropriate instruction it found nothing, got confused, and turned itself off.⁷

Why bother with software engineering? (cont.)

Others are absolutely not acceptable!

The automatic propulsion control in our Boeing 737 had the occasional habit during take-off at exactly 60 knots of cutting out. It was someone in our workshops who looked at our listings found the cause. The programmer had spelled out what the propulsion control should do under 60 knots and what it should do over 60 knots. But he had forgotten to say how it should react at exactly 60 knots. So if at exactly 60 knots the computer asked for the appropriate instruction it found nothing, got confused, and turned itself off.⁷

[With regard to the Lufthansa A320 accident in Warsaw] the spoilers, brakes and reverse thrust were disabled for up to 9 seconds after landing in a storm on a waterlogged runway, and the airplane ran off the end of the runway and into a conveniently placed earth bank, with resulting injuries and loss of life. On 10 Nov, Frankfurter Allgemeine reported that Lufthansa had concluded there was a problem with the logic, and was requiring their pilots to land in a different configuration and a different manner in such weather and runway conditions, to 'fool' the logic. This decision was supported by the Luftfahrtbundesamt.⁸

⁷Hasch, 1983

⁸RISKS-FORUM Digest, Weds 1 December 1993 Volume 15 : Issue 30.

The problem is enormous!

Software development statistics

- The typical software project requires 1-2 years and at least 500,000 lines of code
- Only between 70-80% of all projects are successfully completed
- Over the entire development cycle, each person produces on average less than 10 lines of code per day
- During development on average 50-60 errors are found per 1,000 lines of source code. Typically this drops to around 4 after system release.

How did we end up in this wretched state?



Historical context: 1960s – 1970s

- Batch software (punch-cards!) with highly restricted memory
- Simple complexity
- Low-level, machine-oriented programming languages: Cobol, Fortran, Algol
- Development problems spurred interest in semantics and verification questions
- The term **software crisis** was coined in 1965
- Initiated research in structured data types leading to ALGOL-W (-68), C, Pascal, Ada

Historical context: 1970s – 1980s

- Technology supports increasingly large projects
- Engineering **in the large** leads to new kinds of problems
- Challenge: programming in teams
 - Development must be split \Rightarrow **decomposition** in analysis, specification, coding
 - Unstructure programming methods don't scale: global data, non-modular construction, lack of well-define interfaces
- Gradual recognition that software development difficult!
- Evolution of concepts like software engineering, structured programming, stepwise refinement, modularization, abstract datatypes
- Ideas embodied in languages like Pascal, Modula-2, ML, C++, Java

Current situation — new complexities

- Large scale, distributed, heterogenous systems, e.g., internet centric computing
- cheap microsystems = massive distribution
 - Typical car has 100s of microprocessors
 - computerized control systems are increasingly used in security critical applications, e.g., controlling trains, planes, and nuclear reactors



Roll of software engineering

- Is there a 'silver bullet' ?

No!

Software engineering is less clear cut than, say, theoretical computer science.

- But there are **techniques**, **methods**, and **tools**, that can reduce the complexity of constructing systems
- There are also techniques for building specific kinds of systems with high degrees of reliability

Distribution systems, embedded systems, real-time systems, etc. all have specialized development/validation techniques

Role of a software engineering course

- Is it possible to present and practice the full spectrum of approaches to software engineering in one class? **No!**
 - The industrial setting is completely different from a University
 - Insufficient time for development in the large
 - Different problems demand different techniques
- We survey central concepts and experiment with selected approaches
- Specialized techniques presented in depth in advanced courses

Overview

1. Why bother with software engineering?

⇒ What is software engineering?

2. Structuring and abstraction in modeling

What is software engineering?

- No consensus. Includes:
 - Development process and business aspects, e.g., planning, cost and resource estimation, documentation, check-points, etc.
 - Informal heuristics or rules of thumb
 - (Semi-)Formal methods
- Our definition

Software engineering is the practical application of scientific methods to the specification, design, and implementation of programs and systems.
- Our focus: (semi-)formal methods

(Semi-)Formal Methods

- A **language** is **formal** when
 - it has formal language (syntax)
 - whose meaning (semantics) is described in a mathematically precise way
- A development method is **formal** when
 - it is based on a formal language and
 - there are semantically consistent transformation/proof rules

(Semi-)Formal Methods (Cont.)

- Semi-formal methods are widely used, for example UML
Often just syntax with a mere hint of semantics
- Formal methods offer numerous advantages
 - ++ Typically more concise
 - ++ Precise and unambiguous
 - ++ Precise transformation rules \Rightarrow machine support possible
 - ++ Uniform framework for specification, development, and testing
 - However they are more difficult for novices

Formal or Informal?

- Answer depends on task and available resources
- My sympathies lie with the following author:

This book is a discourse explaining how the task of programming (in the small as well as in the large) can be accomplished *by returning to mathematics*. By this, I first mean that the precise mathematical definition of what a program does must be present at the *origin* of its construction. If such a definition is lacking, or if it is too complicated, we might wonder whether our future program will mean anything at all. [...] At this point, I have no objection with people feeling more comfortable with the word 'English' replacing the word 'mathematics'. I just wonder whether such people are not assigning themselves a more difficult task.

(Jean-Raymond Abrial, the B-Book)

- we will examine both approaches

Also consider the integration of semi-formal with formal methods.

Thematic overview

- Structuring the software development process
 - Process models and support tools
- Modeling, specification, refinement, implementation, and verification
 - (Semi-)formal methods and their integration in development
- Foundations cover many aspects of theoretical computer science
 - Syntax and semantics from specification and programming languages
 - Specification, verification, logic
- Method-oriented themes
 - Problem analysis, modularization, OO-development, model building, . . .

Schedule

2 Weeks: Overview, requirements analysis, development models and version control.

4 Weeks: Specification and modeling

4 Weeks: Implementation and system construction

1 Weeks: Testing

Project is of central importance! **Theory without application is useless!**

Note on the role of languages?

- (Almost) all programming languages are equivalently expressive
- But some provide better support for development
 - Datatypes with abstraction/information-hiding
 - Parameterization (for reusability)
 - State/assignment?
 - Strong typing?
 - Inheritance?
- Wir will use Java
 - Supports OO-programming and the use of class libraries

Overview

1. Why bother with software engineering?
 2. What is software engineering?
- ⇒ Structuring and abstraction in modeling

Modeling



- The goal of development is to solve problems in the **real world**
- Structuring and abstraction are critical: the real world is complex!
- Modeling are built in the early development phases
Goal: specify the requirements clearly and precisely while avoiding a premature commitment to algorithms or datastructures
- Later one builds models of a possible implementation architecture
Model sketches components, interfaces, communication, etc.
- Modeling style depends on notion of “component”, e.g., function, procedure, class, module, . . .

Structuring and abstraction in modeling

- Important for overcoming the complexity of program development **in the large**:

Structuring serves to organize/decompose the problem/solution

Abstraction aims to eliminate insignificant details

- Classical approaches to structuring and simplification include:

Functional decomposition: decomposition in independent tasks

Parameterization and generic development: reusability

Model simplification: to improve understanding of tasks and possible solutions

Information-hiding: interfaces and property-oriented descriptions

Structuring (Cont.)

- In all cases, **interfaces** must be clearly described

Interface: imagined or actual boarder between two functional entities with fixed rules for the transfer of data

Syntactic properties: the available rules, the types of their arguments, etc.

Semantic properties: a description of the entities behavior; goal is to support the proper use of the entity.

- Interfaces also provide the basis for communicating and explaining (sub)systems between the specifier, implementer, and user
- Correctly describing interfaces and ensuring their correct use is a central aspect of software engineering

Simplicity in modeling as a key to problem solving

In an American quiz-show, the candidate can win a car if he correctly guess behind which of the three doors A , B , and C , the car is hidden. To make things more interesting, the show proceeds as follows: First the candidate selects one of the three doors. Then the quizmaster opens one of the two remaining doors, choosing so as to not reveal the car. Afterwards, the candidate has the possibility of switching the door he initial selected for the remaining door.

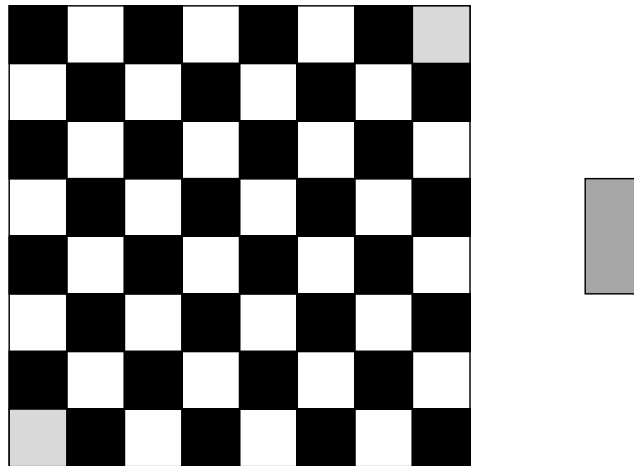
- After the initial selection, the car is behind one of the two doors. **How should the candidate proceed?**
- Strategies
 1. He selects a door and does not change his selection
 2. He selects a door and later changes his selection, i.e., choosing the remaining door
- A probabilistic analysis is nontrivial
- Solution is simple using an appropriate model!

Solution

Claim: Strategy 2 can be seen as allowing the candidate to choose **two** doors, and win when the auto is behind one of them.

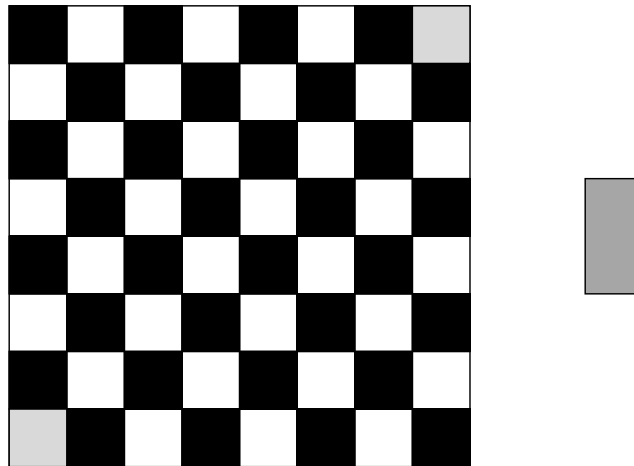
Explanation: Suppose he wishes to select A and B . Then he first chooses C . After the quizmaster opens one of A or B the candidate switches from C to the remaining door. Hence he wins the car if it is behind A or B .

Modeling example #2: Mutilated checkerboard



- Can the board with two missing corner pieces be tiled with the piece shown?
- Solution:

Modeling example #2: Mutilated checkerboard



- Can the board with two missing corner pieces be tiled with the piece shown?
- Solution: seek an invariant!

Requires understanding important aspects of problem

Conclusion

- Software engineering concerns development in the large and related problems
- Techniques for problem structuring and abstraction play an important role
- Formality can be helpful, in particular in
 - creating meaningful, unambiguous models of systems
 - rigorously analyzing and transforming these models
- We will focus on a software development process based on building models, analyzing models, and transforming models into robust, correct, and evolvable systems

The Development Process

David Basin

Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Spring 2002

Overview

- The software life cycle
- Software models
 - Waterfall and V-model
 - Evolutionary development
 - Rational unified Process
- Evaluating models

Software development — historically

- The **code-and-fix** development process
 1. Write program
 2. Improve it (debug, add functionality, improve efficiency, ...)
 3. GOTO 1
- Works well for 1-man-projects and Informatik I assignments
Unstructured development fine for small projects
- Larger projects \implies software crisis
 - Poor reliability
 - Disastrous when programmer quits
 - Inappropriate for multiple man-year projects
 - Expectations often differ when the developer isn't the end-user
 - Maintainability? What's that?

Requirements for development process

- A procedure to **guide** and **control** the entire development.
- Should support developing **high-quality** systems.

Adequacy: System satisfies desired requirements

Usability: Appropriate GUIs, documentation, etc.

Reliability: Robustness, security, etc.

Maintainability: System easy to modify and improve

Cost: Acceptable development costs (time, money, etc.)

Performance: Resource use is minimized (or not wasted)

Software development activities

- Development process: **activities** and **results** of software production
- Four basic activities:
 - Specification:** Definition of functionality and constraints
 - Development and Implementation:** Production of system
 - Validation:** Verification, testing, etc.
 - Maintenance:** Changes and improvements
- Subdivision of activities depends on the particular process employed
 - Differs depending on the kind of system built and the organizational context

Process models

- Numerous proposals for process models:

Descriptive: How **are** systems developed?

Prescriptive: How **should** systems be developed?

- Examples:

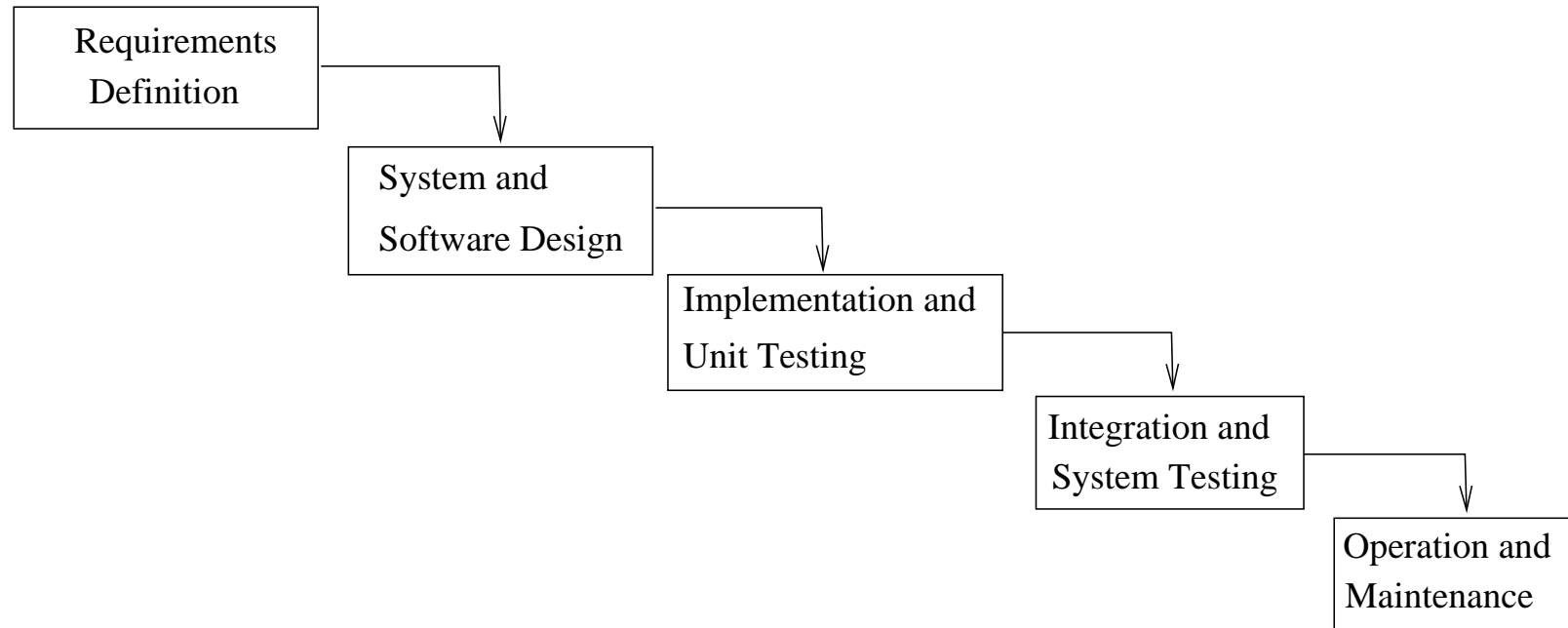
Waterfall: Activities staged in phases, each with a well-defined deliverable.

Evolutionary: Activities are interleaved. System quickly develops from rough specification.

Formal Transformation: Stepwise transformation of a formal specification into a program.

Construction from reusable components: Reuse of existing components.

Waterfall model (Royce 1970)



- First process model (also called phase model)
 - The development is decomposed in phases
 - Each phase is completed before the next starts
 - Each phase produces a product (document or program)
- Enthusiastically welcomed by managers!

Waterfall phases

Requirements analysis and definition:

- System requirements defined with customer.
- Result should be understandable by both customer and system developers

System development:

- Requirements are decomposed into software and hardware requirements.
- A system-architecture is fixed.

Implementation and testing of components:

- The system is realized as a set of components (objects, modules, units, ...).
- Units are individually tested.

Integration and system testing: Units are integrated, integration tests are performed, and the resulting system is delivered.

Operation and maintenance: Bug fixes, improvements, etc.

Waterfall process assumptions

- Requirements are known from the start, before design
- Requirements rarely change
- Design can be conducted in a purely abstract way
- Everything will all fit nicely together when the time comes

Advantage of waterfall: transparency

Activity	Result
Requirements analysis	Feasibility study, requirements sketch
Requirements definition	Requirements plan
System specification	Functional specification
	Test plan,
	Development of user documentation
Architecture development	Architecture specification
	System test plan
Interface development	Interface specification
	Integration test plan
Detailed development	Specification, unit test plan
Programming	Program
Unit test	Report
Module test	Report
Integration test	Report, final user documentation
System test	Report
Acceptance test	Report, documentation

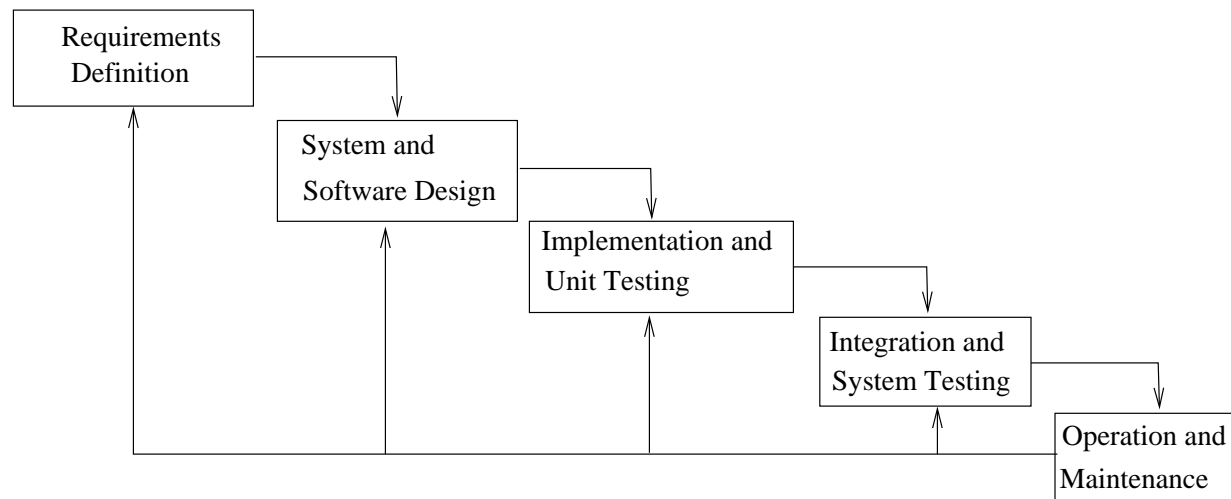
The output of one phase is the input of the next.

Problems with the waterfall

- The process assumptions typically don't apply! E.g., requirements typical imprecise and mature as development advances
 - The **Big Bang Delivery Theory** is risky: proof of concept only at the end!
- Too much documentation! (Paper flood \Rightarrow CASE TOOLS)
- Late deployment hides many risks
 - Technological (well, I **thought** they would work together...)
 - Conceptual (well, I **thought** that's what they wanted ...)
 - Personnel (took so long, half the team left)
 - User doesn't **see** anything real until the end, and they always **hate** it
 - Testing comes in too late in the process

Problems (cont.)

- Unidirectional flow often too stiff: feedback is needed between phases
- Unidirectional flow means problems are pushed to others or programmed around
- Alternative: weakening through feedback



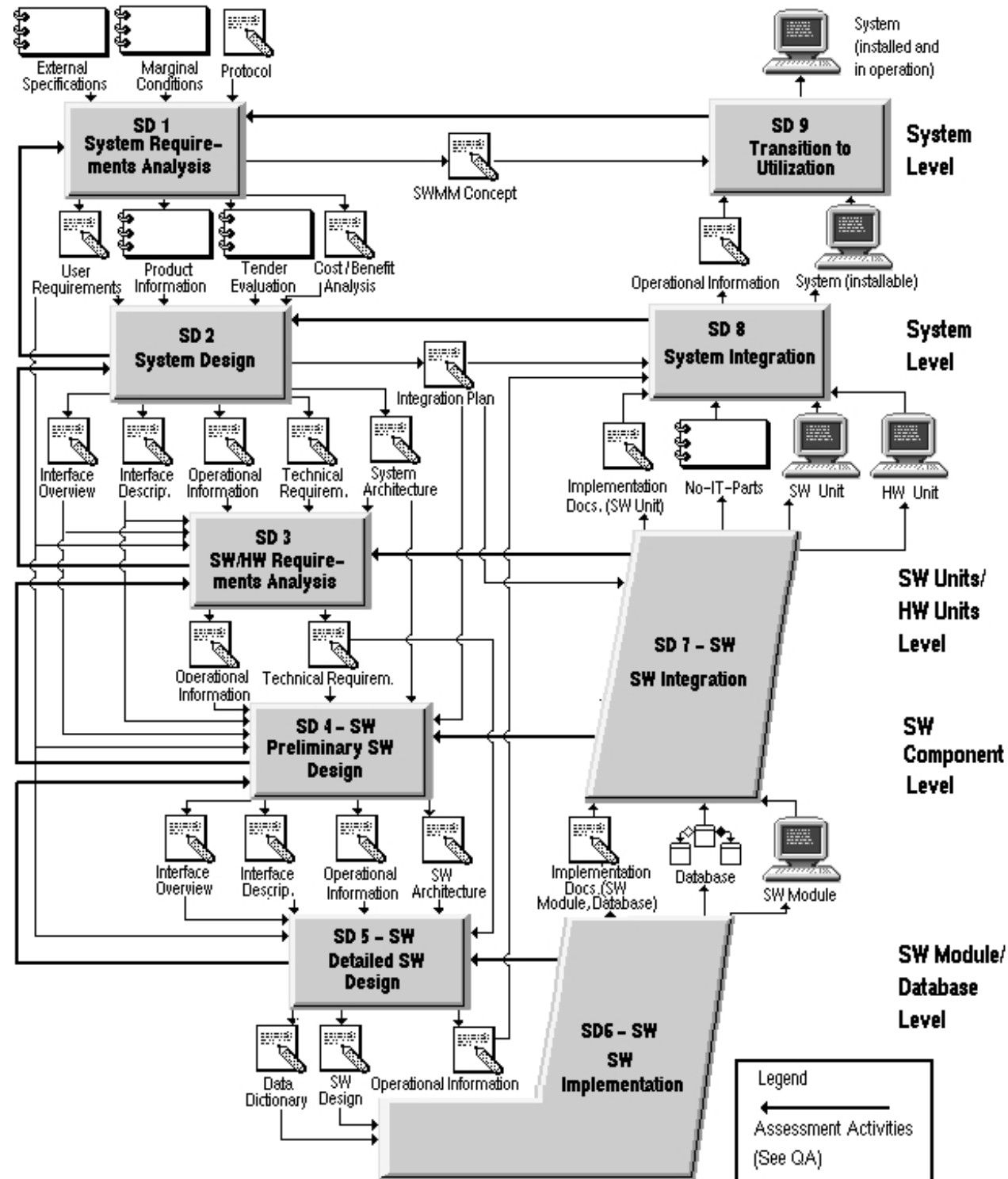
New problem: frequent iteration makes checkpoints difficult

Variations on a waterfall

- Many variants, depending on organization, country, and software product
- The **V-Model** is an (ISO-Standard) Model for military and administrative projects in Germany

Regulates all **activities** and **products** as well as **product states** and **relationships** during IT-development and maintenance

- Built from different submodels. Describes system development, configuration management, project management (purchasing, planning, ...), etc.
- System development model can be seen as a V-formed variant of the waterfall.



Evolutionary development

- **Idea:** build a prototype and continually improve it.

Build in customer feedback at each iteration

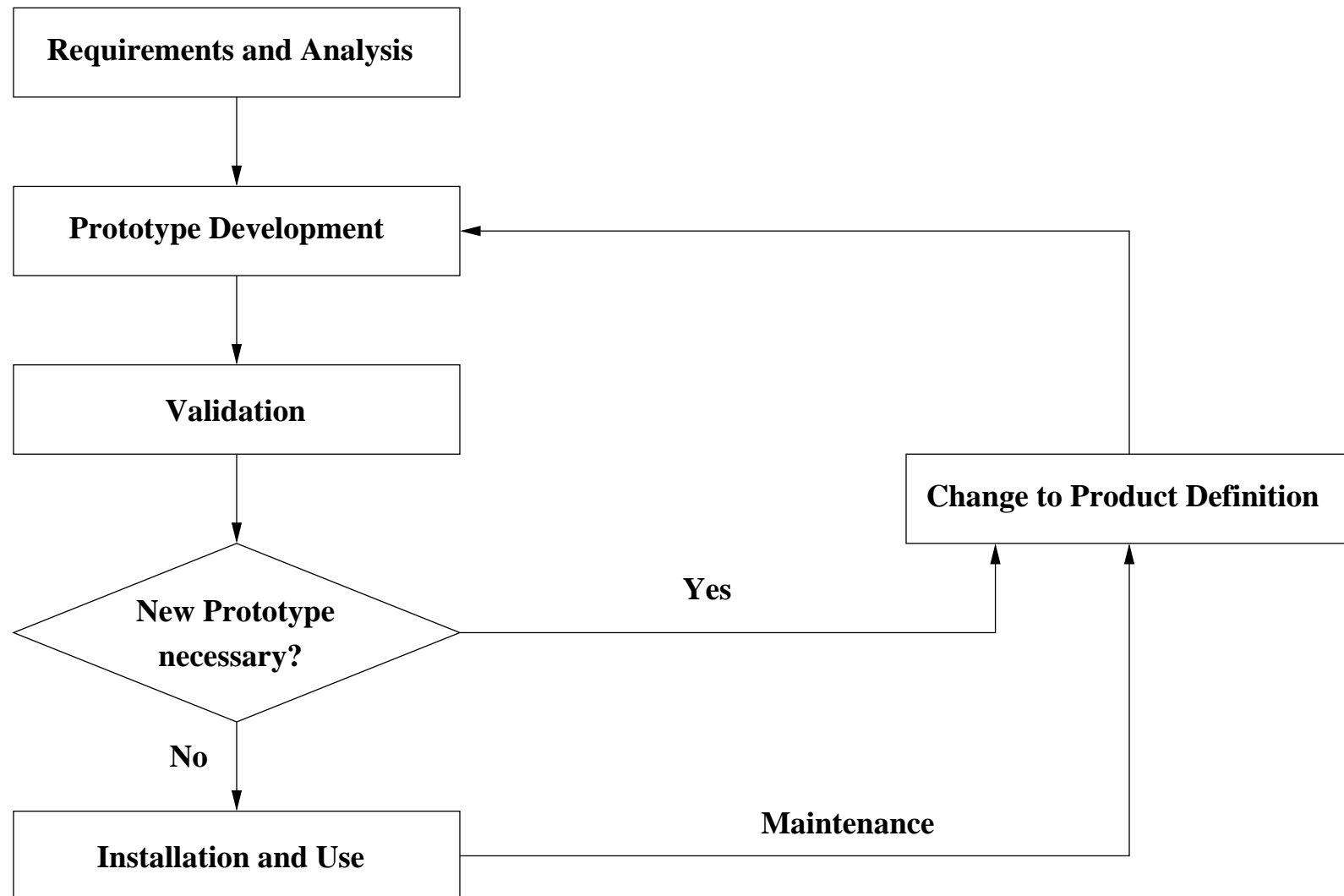
- All process activities occur simultaneously

- Two kinds of development:

Exploratory Programming: Query customer requirements. First implement basic requirement and later add more complex ones.

Throw-away-Prototyping: Build prototype to **understand** the customer's requirements. The prototype and experiments with the customer help to **define** the requirements.

Evolutionary development (cont.)



Evolutionary development (cont.)

- **Theoretically, wide applicability**; many systems built this way
- Problematic — **look ma, no hands!**

Not transparent: Difficult to judge progress. Managers have no checkpoints

Poorly structured code: due to frequent modification

Requires a skilled team: Small, skilled, and motivated group

- **Practically, narrow applicability**: small systems with limited life times
- Typical example: AI projects where it is difficult to specify human capabilities and activities

Rational Unified Process

- Wide spread methodology championed by Rational Corporation
- Combines water-fall and evolutionary development
Plan a little, design a little, code a little.
- Aims to minimize risk of failure
Breaks system into mini-projects, focusing on riskier elements first
- Other (claimed) advantages
 - Encourages all participants, including testers, integrators, and documenters to be involved earlier on
 - Mini-waterfalls centered around UML, a particular OO-methodology
 - CASE-TOOL support (of course, from Rational)
- Does it work?
Many positive case studies although benefits difficult to quantify

Evaluating process models: attributes

Clarity: How precise is **process** defined? How easy is it to understand?

Transparency: Do activities produce clearly defined results?

Support: Can activities be machine supported?

Acceptability: Will developers accept the process?

Reliability: Are errors avoided or quickly discovered?

Robustness: Can the process be continued, despite problems?

Maintenance: Can changes be easily integrated?

Time factor: How quickly can a system be built?

A comparison: software development versus ship building

- Software lacks form and body

The manager of a ship yard can see components and measure progress

- The software development is not standardized

Ship construction has a long history and standard construction procedures. One knows how to specify and test ships.

- Software products are usually not standardized

- Many ships are similar.

- In contrast, many software projects involve new challenges/technologies

- Some kinds of systems are fairly standard \implies component-based systems

- Software errors are acceptable

A company that builds sinking ships will be sued!

Conclusion

- The software development process needs structuring
 - Various models prescribe structure with different advantages and disadvantages:
 - Strong structuring results in inflexibility
 - Weak structuring degenerates to code-and-fix
 - Improvements and, in particular, tool-support are active research areas
 - We will use the water-fall model in the class
- We will support feedback by using version management

Requirements Analysis

David Basin

Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Spring 2002

A welcome problem. . .

You have inherited 100 Million Euro and want to start your own airline company!

What system should your future IT-department design?

Well, there is some data to administer

- Flights:

Routes, times, seats, crew, food (first class, vegetarian, . . .)

- Crew and support staff

Personal data, tasks, time plan, pay-roll

- Customers

Reservations, accounts, Miles & More, meat-eater, . . .

- Inventory

Airplanes, fuel, vegetarian food, deliveries (carrots, . . .)

And don't forget

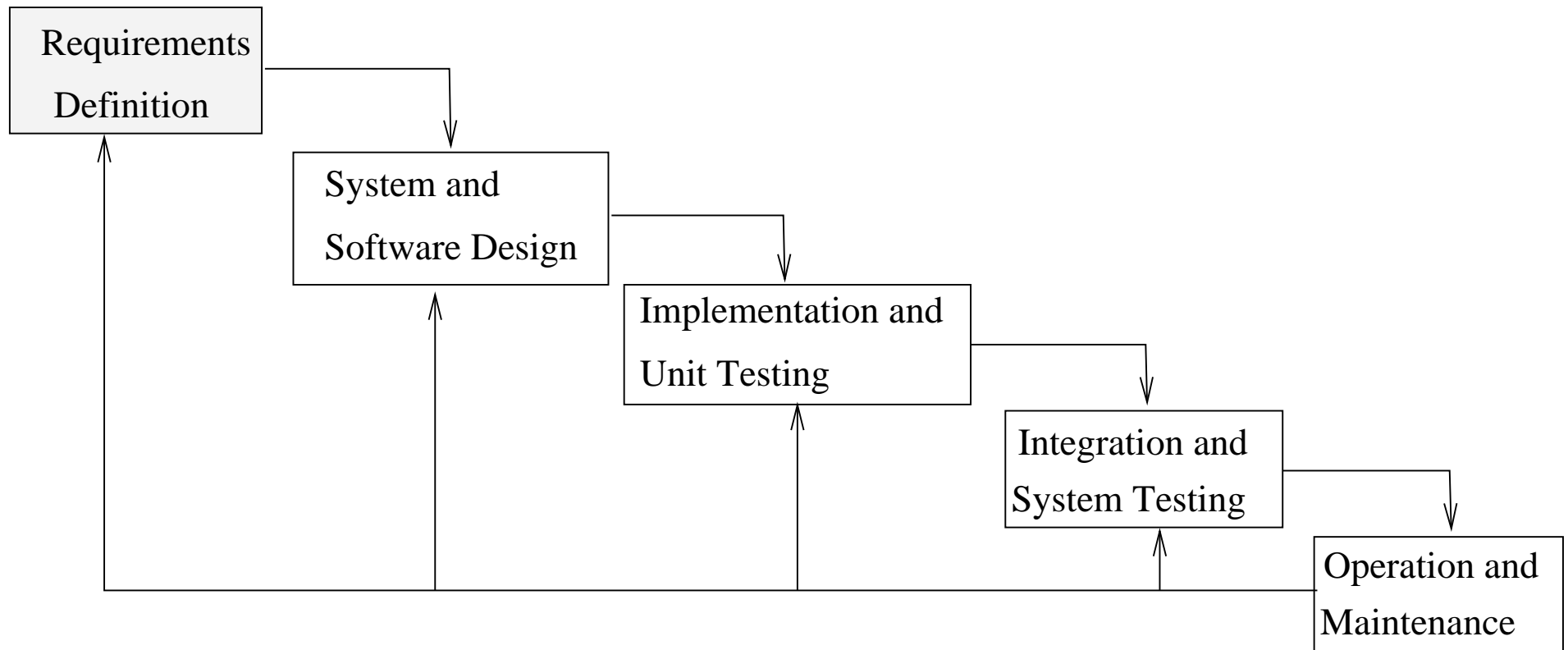
- Planning programs (time and cost plans)
- Support for E-commerce
 - Customer-to-business, e.g., buying tickets, checking status
 - Business-to-business, e.g., supply-chain management
 - Web pages (internal/external)
- Building a system that is secure, redundant (fault tolerant), evolvable, . . .

and this is only the beginning!

Overview

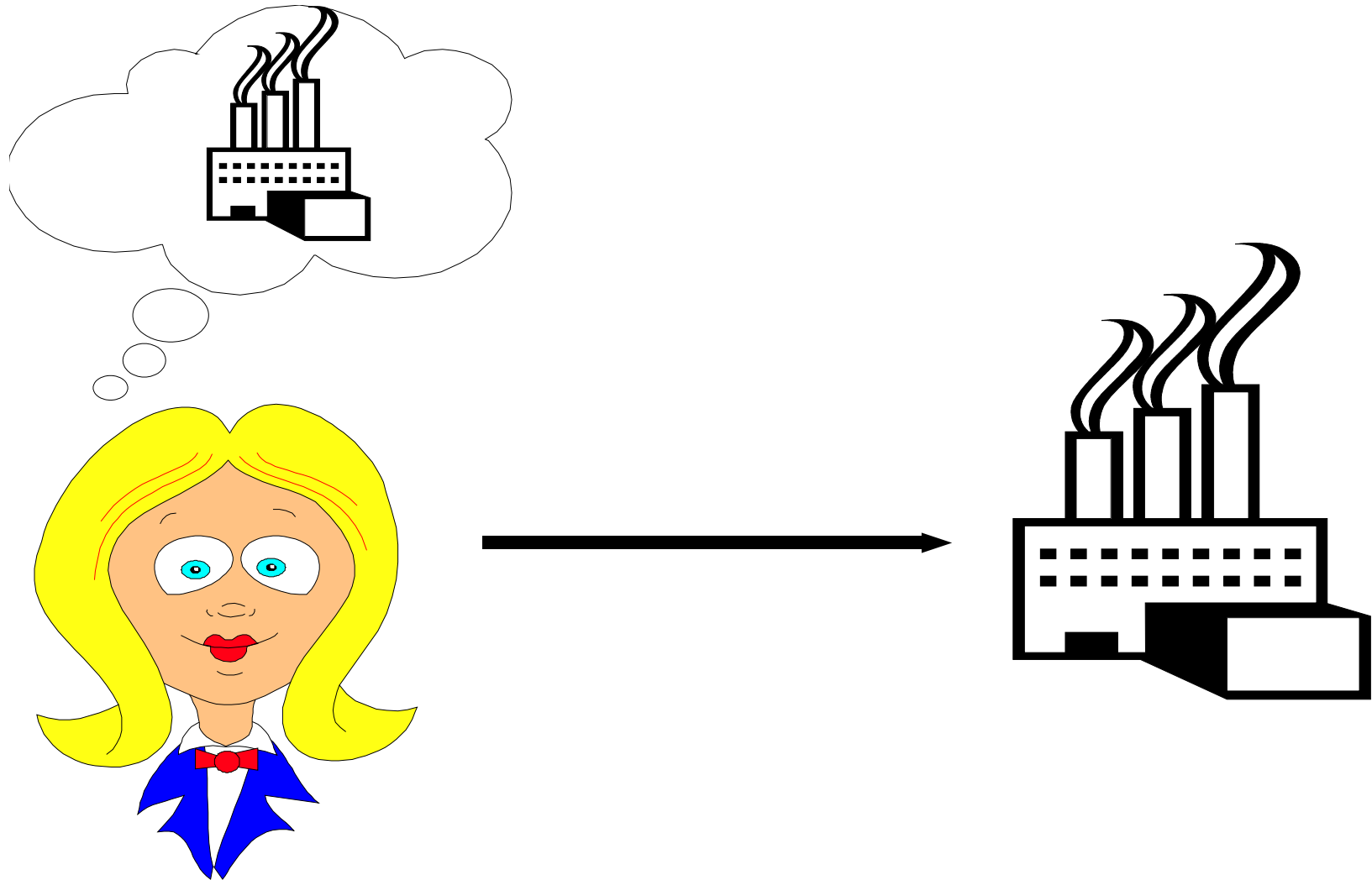
- Background/motivation
- The analysis and definition phase
- Structuring requirements
- An example

The context



First phase: requirements analysis & definition
Sometimes called requirements planning

Goal



Goal (cont.)

The goal: Specify the requirements as **detailed** as possible, but as **abstract** as possible.

- To understand what the product should do
 - Otherwise programming is pointless!
 - Exception: rapid prototyping to aid analysis itself
- As a contract with the customer
- To plan the development
- Many projects require a concrete product definition
 - Companies, governments, most large companies . . .
 - The German, term **Pflichtenheft** is sometimes used to stress the legal aspects.

Planning is difficult

- What does one want?

The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later. (Brooks 1987)

- Compromises are necessary between different, contradictory user requirements

Example: Bank customers want security. Bank employees want minimal restrictions and overhead.

- Delayed effects are difficult to predict

Example: Influence of the invention of the auto on the sexual behavior of American teenagers

Possible structuring of requirements analysis

Name	For whom	Question addressed
Requirements analysis	Customer (Managers) Users Contract manager System architects	What is the problem (rough)? Answer: Product sketch in natural language (+figures)
Requirements specification	Customer System architects Programmers	What is the problem (detailed)? Answer: Product definition Precise structured document Serves as contract
Software specification	System architect Programmers	What is the solution (rough)? Answer: System architecture

Structuring requirements analysis — an example

Product sketch:

1. The program should read files over the Internet

Product definition:

- 1.1 The user can access and view files over the Internet
 - 1.2 A file is given through a URL, typed in from the keyboard
 - 1.3 Various types of URLs are supported
 - 1.3.1 'html' means that the file is hyper-text . . .
- (Explanation of how are files loaded, displayed, browsed, . . .)

Software specification:

- (Abstract description of the main browser routines, . . .)

Structuring requirements

- Unstructured text is poor. Example: Editor for a CASE Tool)

2.6 Grid facilities To assist in positioning entities on a diagram, the user may turn on a grid in either centimeters or inches, via an option on the control panel. Initially, the grid is off. The grid may be turned on and off at any time during an editing session and can be toggled between inches and centimeters. A grid option will be provided on the reduce-to-fit view but the number of grid lines shown will be reduced to avoid filling the smaller diagram with grid lines.

- Problems
 - The first sentence mixes different requirements
 - Incomplete description (e.g., units used when turned on)
 - Justification partially mixed with specification

Structuring requirements (cont.)

Better is to describe fundamental properties. . .

2.6 Grid Facilities

2.6.1 The editor shall provide a grid facility where a matrix of horizontal and verticle lines provides a background to the editor window. This grid shall be a passive grid where the alignment of entities is the user's responsibility.

Rationale: A grid helps the user to create a tidy diagram with well space entities. Although an active grid, where entities "snap-to" grid lines can be useful, the positioning is imprecise. The user is the best person to decide where entities should be positioned.

2.6.2 When used in "reduce-to-fit" mode (see 2.1), the number of units separating grid lines must be increased.

Rationale: If line spacing is not increased, the background will be cluttered with grid lines.

Specification: ECLIPSE/WS/TOOLS/DE/FS Section 2.6

Structuring requirements (cont.)

. . . and to describe basic operations

3.5.1 Adding nodes to a design

3.5.1.1 The editor shall provide a facility where users can add nodes of a specified type to a design. Nodes are selected (see 3.4) when they are added to the design

3.5.1.2 The sequence of actions should be as follows:

- (1) The user should select the type of node to be added
- (2) The user moves the cursor to the appropriate node position in the diagram and indicates that the node symbol should be added at that point.
- (3) The symbol may then be dragged to its final position.

Rationale: The user is the best person to decide where to position a node on the diagram. This approach gives the user direct control over node type selection and positioning.

Specification: ECLIPSE/WS/TOOLS/DE/FS Section 3.5.1

Structuring requirements documents

- No standard structure
 - Some organizations have their own standards
 - Can also depends on process model and the kind of system developed
- Some common principles
 - Describe goals, functionality, environment/use
 - Description should be abstract and avoid unnecessary commitments

Example: ANSI/IEEE-Standard STD-830-1984 (simplified)

1. Introduction

- (a) Product objectives
- (b) Product scope
- (c) Definitions, acronyms, abbreviations
- (d) References
- (e) Overview of requirements description

2. General description

- (a) Product function
- (b) User description
- (c) General restrictions
- (d) Acceptance criteria and dependencies

3. Specific requirements

4. Appendix

NASA-Standard SMAP-DID-P200-SW (simplified)

1. Introduction
2. Related documentation
3. Requirements on external interfaces
4. Requirements specification
 - (a) Processes and data requirements
 - (b) Behavior and quality requirements
 - (c) Security requirements
 - (d) Implementation constraints and installation requirements
 - (e) Development goals
5. Plan for incremental delivery of subsystems
6. Definitions, acronyms, abbreviations
7. Notes
8. Appendix

V-Model-Standard: customer requirements (simplified)

1. General
2. Analysis of status quo (current situation)
3. IT security goals
4. Threat and risk analysis
5. System requirements
 - (a) General system description and intended use
 - (b) Organizational context for system deployment
 - (c) Description of external interfaces
 - (d) Functionality requirements
 - (e) Quality requirements
6. Constraints
 - (a) Technical constraints
 - (b) Organizational constraints
 - (c) Miscellaneous constraints

Let's consider a concrete example
of a simplified product sketch.

Project sketch — organization

Section I Problem description and goals

Section II Functionality

Section III User profile

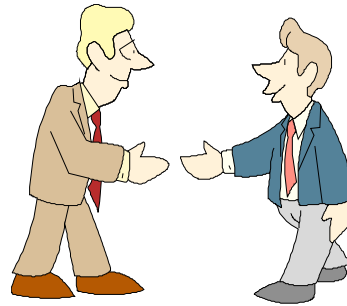
Section IV Acceptance criteria

Section V Development, deployment, and maintenance environments, interfaces, and other considerations

Section VI General architecture (solution sketch)

Section VII Information sources (e.g., contact persons, manuals, glossary)

Project sketch — an example (adapted from Pagel/Six)



Customer (bank manager or head of IT division) to IT consultant:

The entire account processing including cash payments, checks, and electronic transactions (e.g., transfers and e-banking) for our bank branch are currently supported by an antiquated accounting system. The support of our bank employees and, especially, our customer representatives is inadequate: information about customers must be individually extracted from individual accounts. Simple queries such as computing balances over multiple customer accounts cannot be carried out directly.

We would like a new branch information system that extends the functionality of the current system and provides better support for the customer representatives and other bank employees. Moreover, we expect an improvement in overall productivity.

Typical: imprecisely defined improvement of the Status Quo.

Bank information system (BIS)

Delivered on . . .

by . . .

Section I Problem description and goals

Section II Functionality

Section III User profile

Section IV Acceptance criteria

Section V Development, deployment, and maintenance environments, interfaces, and other considerations

Section VI General architecture (solution sketch)

Section VII Information sources (e.g., contact persons, manuals, glossary)

Project sketch

Section I. Project description and goals

The information system should help administer data and accounts for customers of branch banks. **Primary goals** are:

- Fast retrieval of up-to-date information about customers and their accounts, to aid customer support
- Cost-effective and secure administration of payments
- To automate the processing of standing orders

The system should allow for a printer that prints account statement so that . . .

Employee training is planned. Maximally 2 employees per branch . . .

Project sketch

BIS

Section II: Functionality

Develop a comprehensive Bank Information System (BIS) for the customer's different branch banks. Functionality can be loosely categorized as follows:

a) Account administration

The analysis of the current customer and account data indicates that the following functionality is required:

- Create new accounts
- Delete and change accounts
- Create, change, and delete standing orders
- . . .

b) Payments

Payments require the following functionality: . . .

Project sketch

BIS

Section III: User profile

Users of BIS are, exclusively, employees of the branch bank. They are

- **Bank tellers** who process cash payments
- **Customer representatives** who use the information system to advise customers and administer their data
- **Administrators** who maintain user data, fix technical problems, and are responsible for backups.

Section IV: Acceptance criteria

The primary criteria are the **execution time** of the various functions and the **correctness** of account data and the automatic payment processing.

The time for manual activities (payments, information processing, account administration) should be substantially reduced with respect to the current system.

The automatic functions . . .

As far as possible, all data should be administered in an object-oriented database management system. This is mandatory for customer and account data.

The **security** of the data must be guaranteed using a mechanism for access control.

Testing should be employed to verify all procedures that manipulate account data. Deposits, withdrawals, and transfers must function absolutely correctly.

Project sketch

BIS

Section V. Development, deployment, and maintenance environments, interfaces, and other considerations

Each branch should have its own software copy as well as a mainframe computer with multiple graphic terminals. The system operates in multi-user mode.

The system should be linked to an external central system that coordinates the work of the individual branches and maintains central data.

The development and maintenance environment consist of Unix workstations and the operating environment consists of a SUN-compute server running Unix and supporting an Oracle database system. The data transfer between the central system and the BIS is over Datex-P cables. . . . Graphical interfaces in the current standard (e.g., OSF-Motif) will be used.

The system includes a one year guarantee. A maintenance contract can later be arranged.

Subsequent system extensions possibly include a subsystem to aid investment planning and support for managing life-insurance policies.

Project sketch

BIS

Section VI. General architecture (solution sketch)

The system is based on a Client-Server-Architecture within a local-area network.

All static customer and account data are maintained in a relational database on the server side. . . .

Cash transactions (deposits/withdrawals), for branch accounts, are carried out within the BIS. Non-cash transactions (e.g., checks, transfers, etc.), are first processed by the BIS using a special transfer-account. From this special account

VII. Information sources (e.g., contact persons, manuals, glossary) . . .

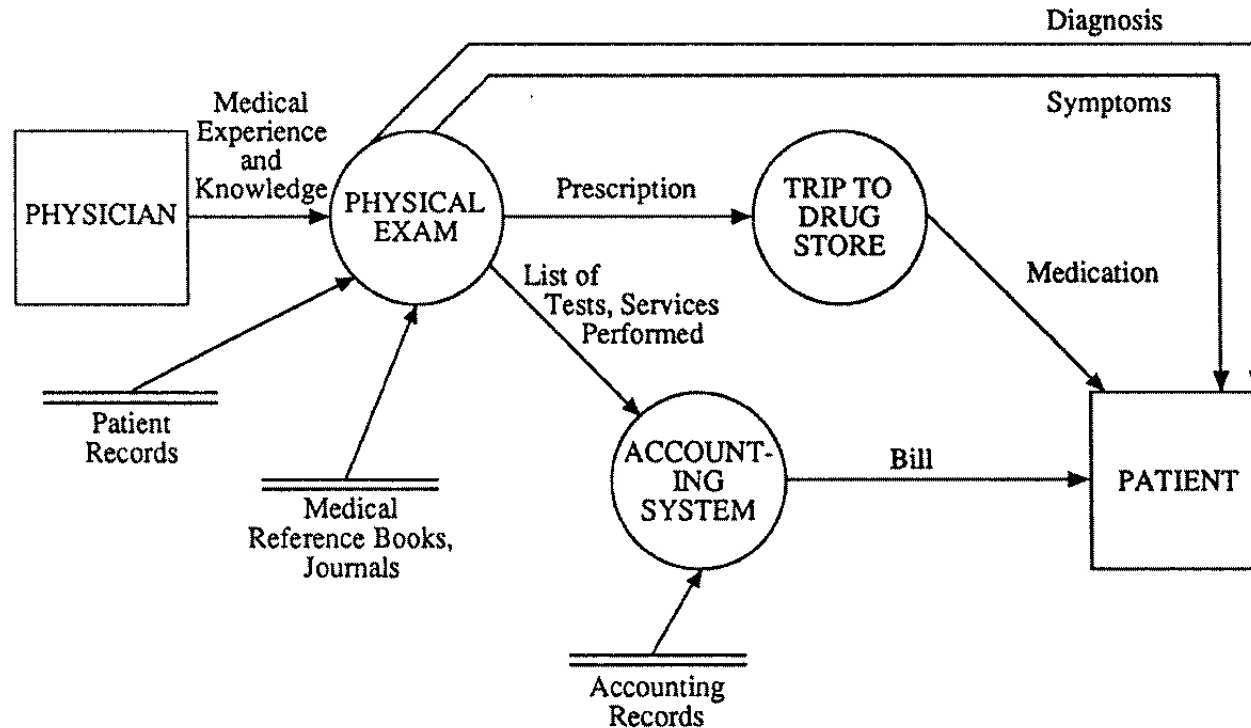
Requirements validation

- Goal: check whether the sketch is:
 - Valid:** Is the right functionality specified? For whom?
 - Consistency:** There should be no conflicts between requirements
 - Completeness:** The entire functionality should be specified
 - Realistic:** It must be possible to implement the system under the given resource (time and financial) constraints
- How? Different approaches with different advantages/disadvantages:
 - ‘Walk-through’ and ‘requirements review’
 - Construction of a prototype
 - Use of logic-based tools
- Process models sometimes prescribe form of analysis

The requirements plan

- A more detailed product sketch
 - Precise description of external system behavior: functional and non-functional requirements
 - Uses detailed system model (coming shortly!) and views
- Typically combines
 - informal languages:** natural language
 - semi-formal languages:** graphical notation, Design description languages (e.g., UML)
- But seldomly
 - formal:** automata, petri-nets, first-order logic, . . .
 - as formal languages are considered less customer friendly

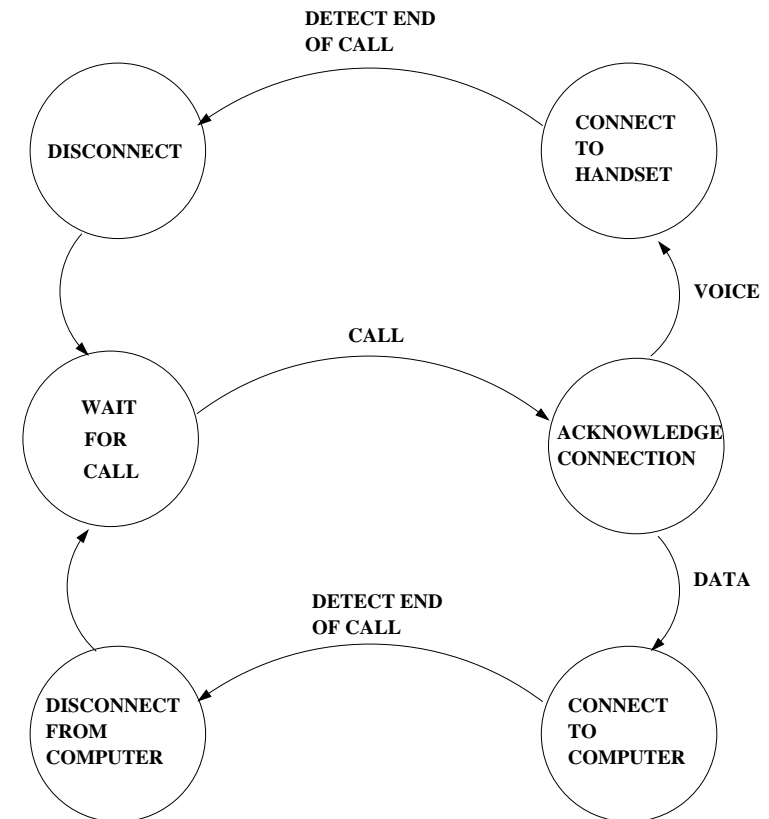
Example: data-flow modeling



- System modeled as data-transformer
- Describes data as well as function input and output
- Control flow not fixed
- Expresses only (very weak!) functional requirements

Example: event-oriented modeling

- System modeled as reactive system
 - System consists of states and transitions
 - Transitions describe reactions to events
- Models often more powerful than automata
 - Hierarchical or with parallelism
 - Sometimes fully formal
 - Basis for, e.g., developing real-time systems
- Functional requirements are not considered!



Coming up next. . .

- We will study modeling in detail
 - Emphasis: (Semi-)formal modeling languages and their application
 - Modeling data, objects, functions, processes, . . .
 - Semi-formal languages like UML and formal languages like Z
- But next: version control and its role in the software process

softech

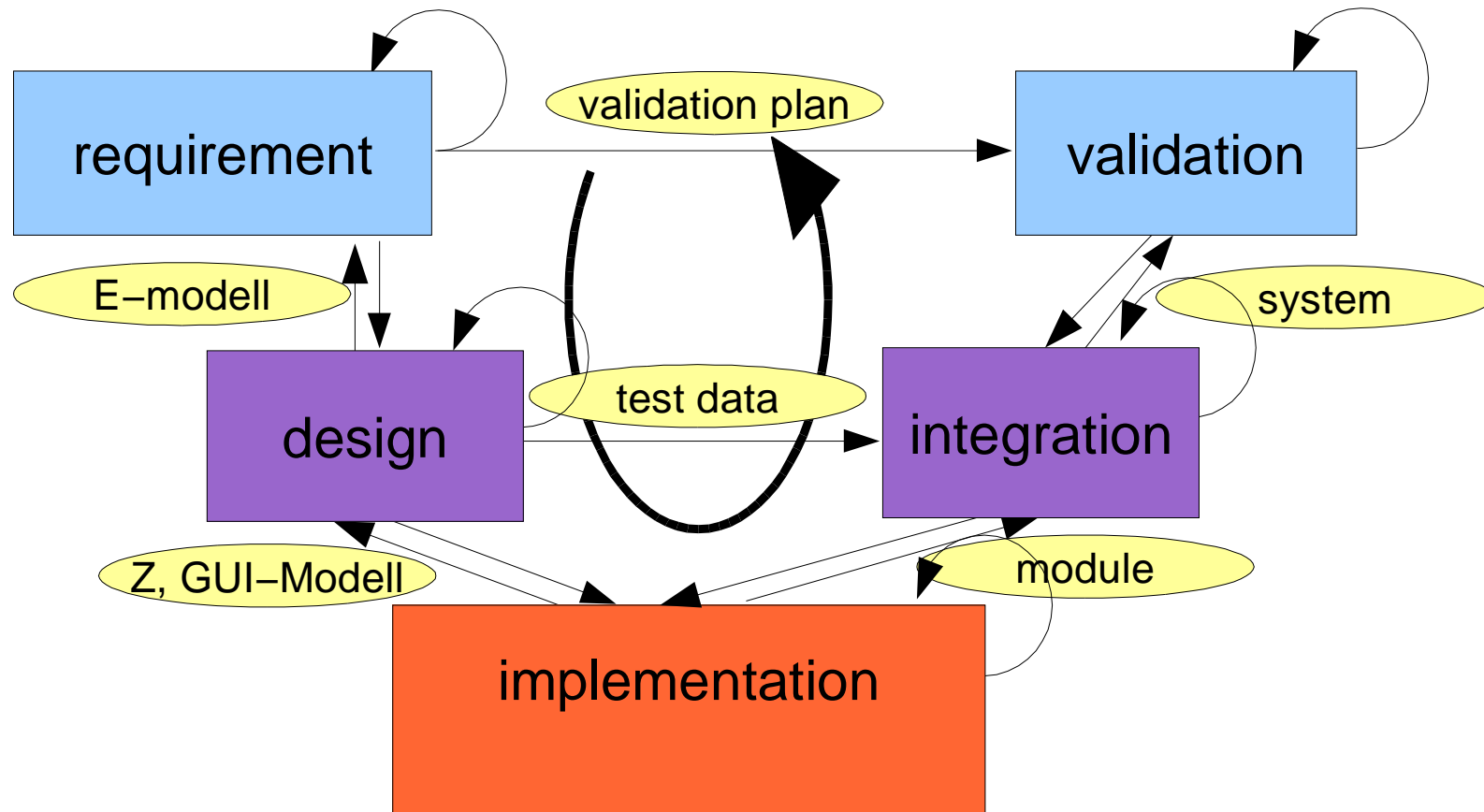


Version Management in the Software Cycle (with CVS as an Example)

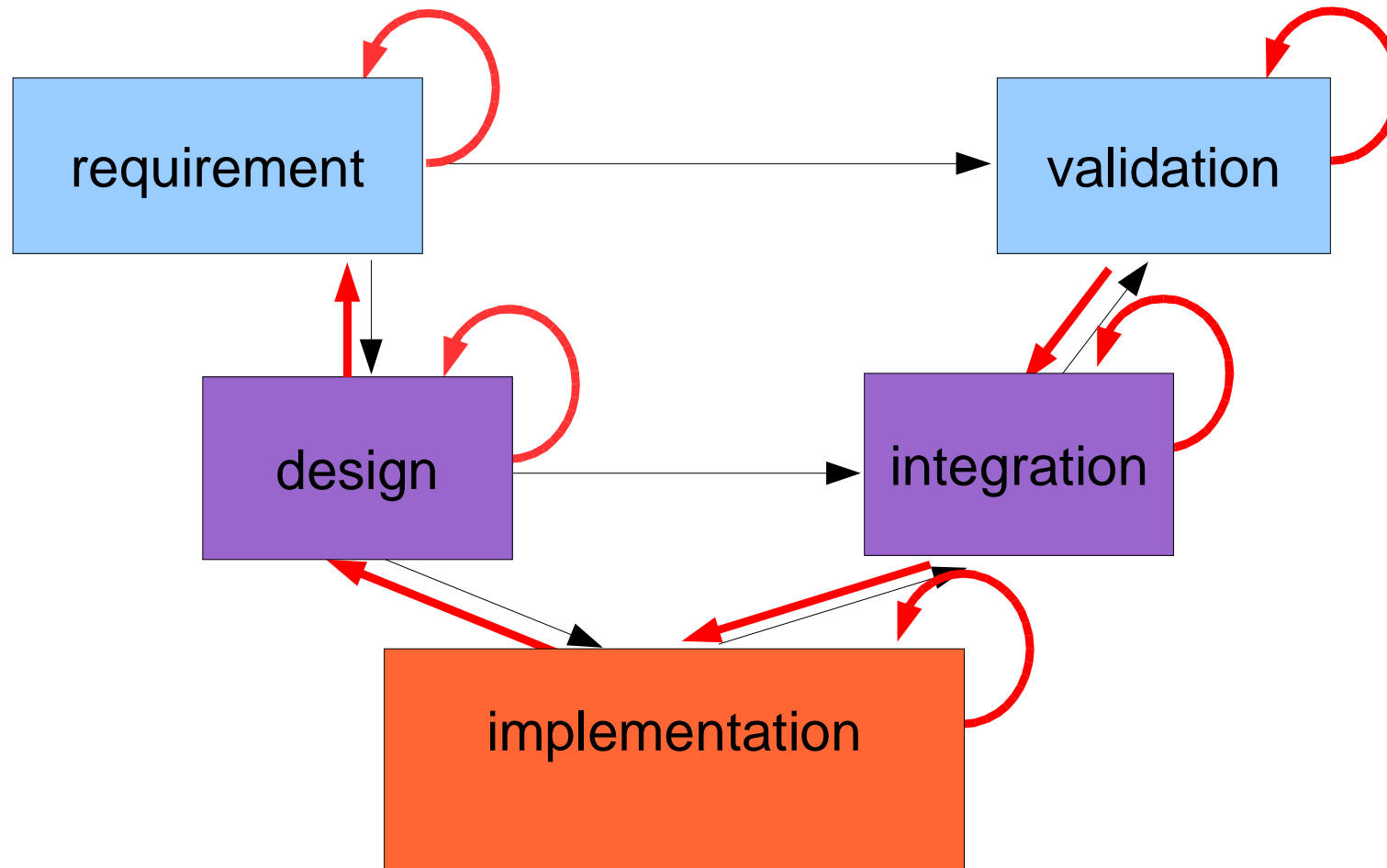
Burkhart Wolff

Institut für Informatik
Albert-Ludwigs-Universität Freiburg

The Role of Version Management: An Introduction



The Role of Version Management: An Introduction



Overview

- What's Version Management?
- Concepts: Branches (Zweige), Modules, Repository, Distribution, Security
- Outlook: Configuration management
- A concrete example: CVS
 - pragmatics of CVS
 - weaknesses of CVS
- Outlook

What's Version Management?

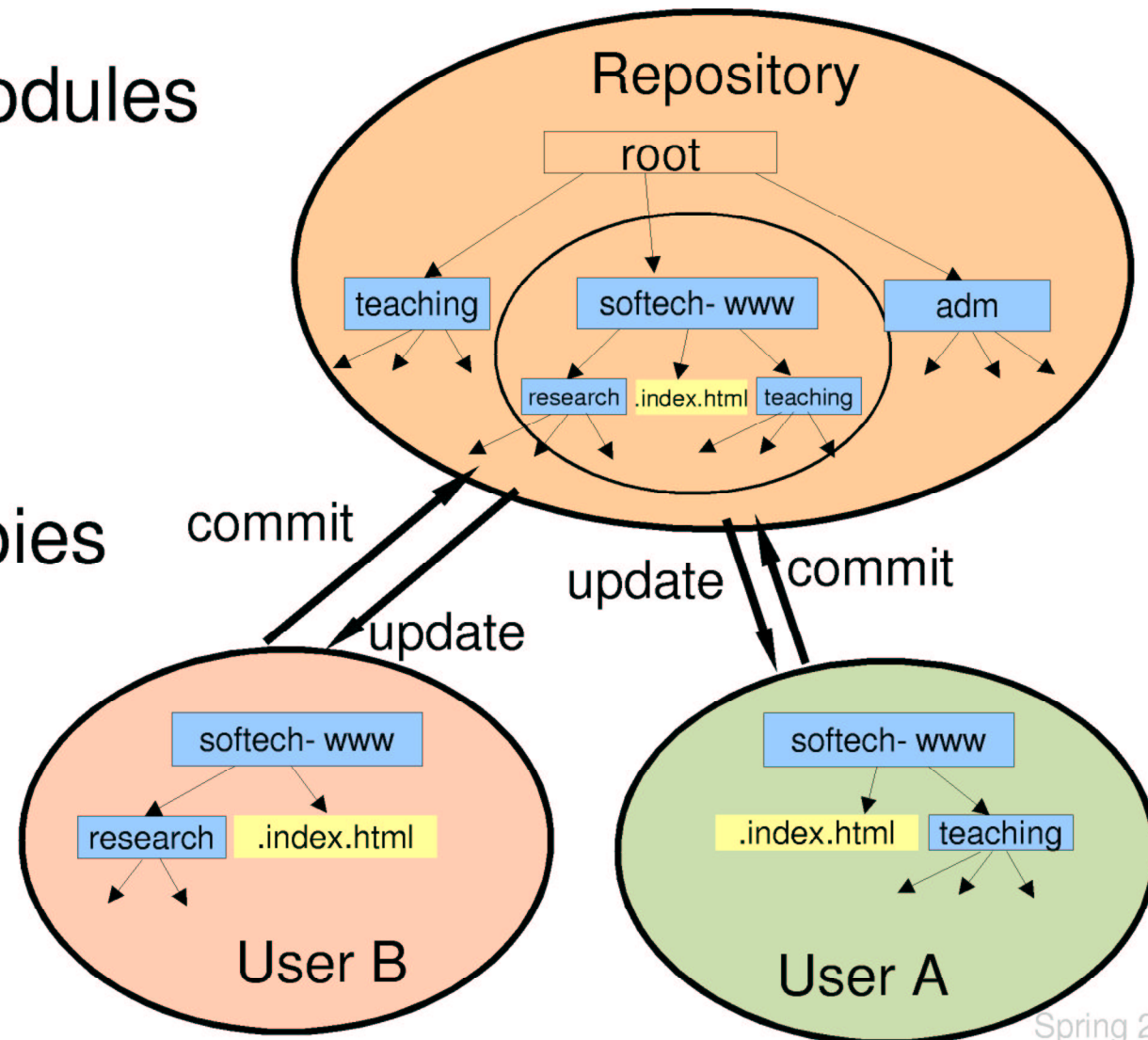
- administration of sequences of documents
(sources, *.c, *.ML, *.tex, ...
documentation, *.o, *.ps, *.gz)

and their **reconstruction**

- optimisation and compression
- administration of distributed development
(**and** synchronisation ("Merge"))
- generation of releases

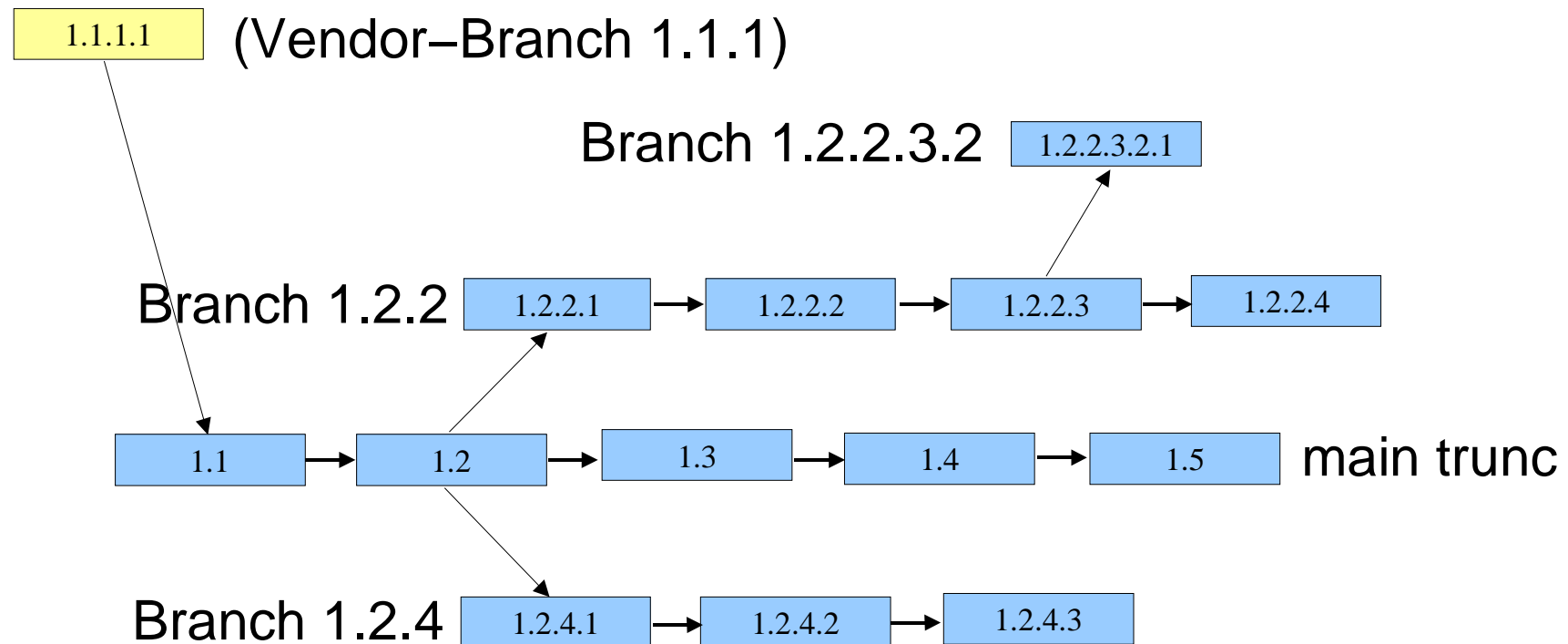
Basic Notions

- projects, modules
- user, working copies



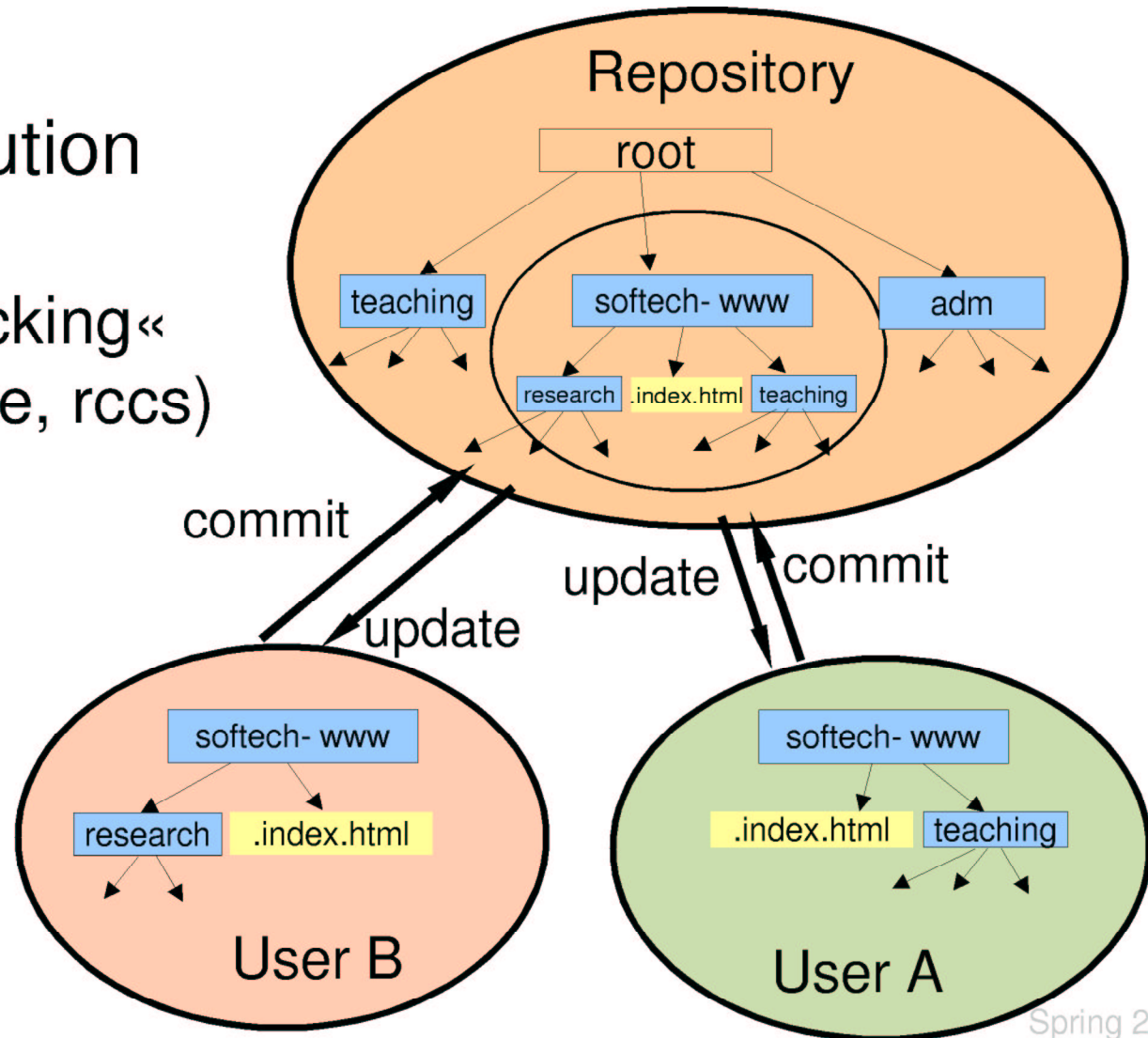
Basic Notions (2)

- Branches (Entwicklungszweige)



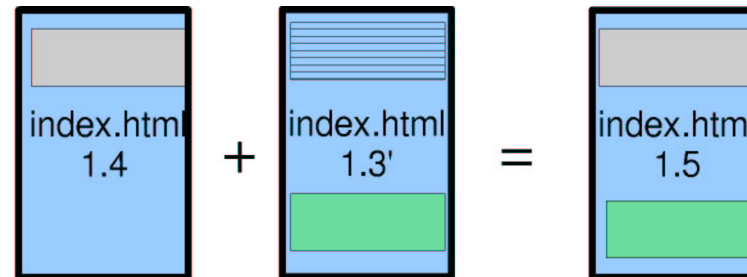
Basic Notions (3)

- conflict resolution
- »resource locking«
(e.g.. Perforce, rccs)
- »merge«
(e.g. CVS)

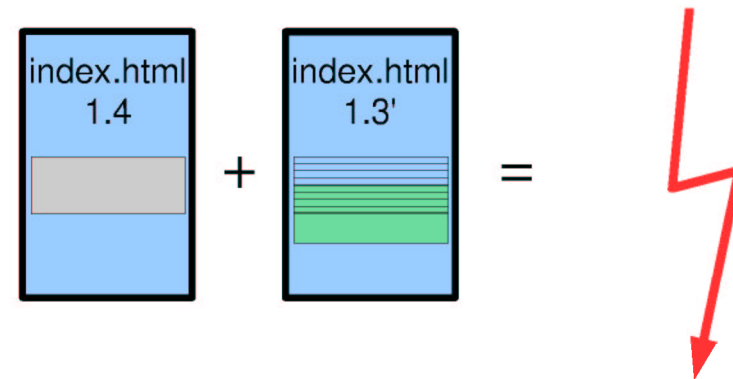


Basic Notions (4)

- Merge : No Conflict

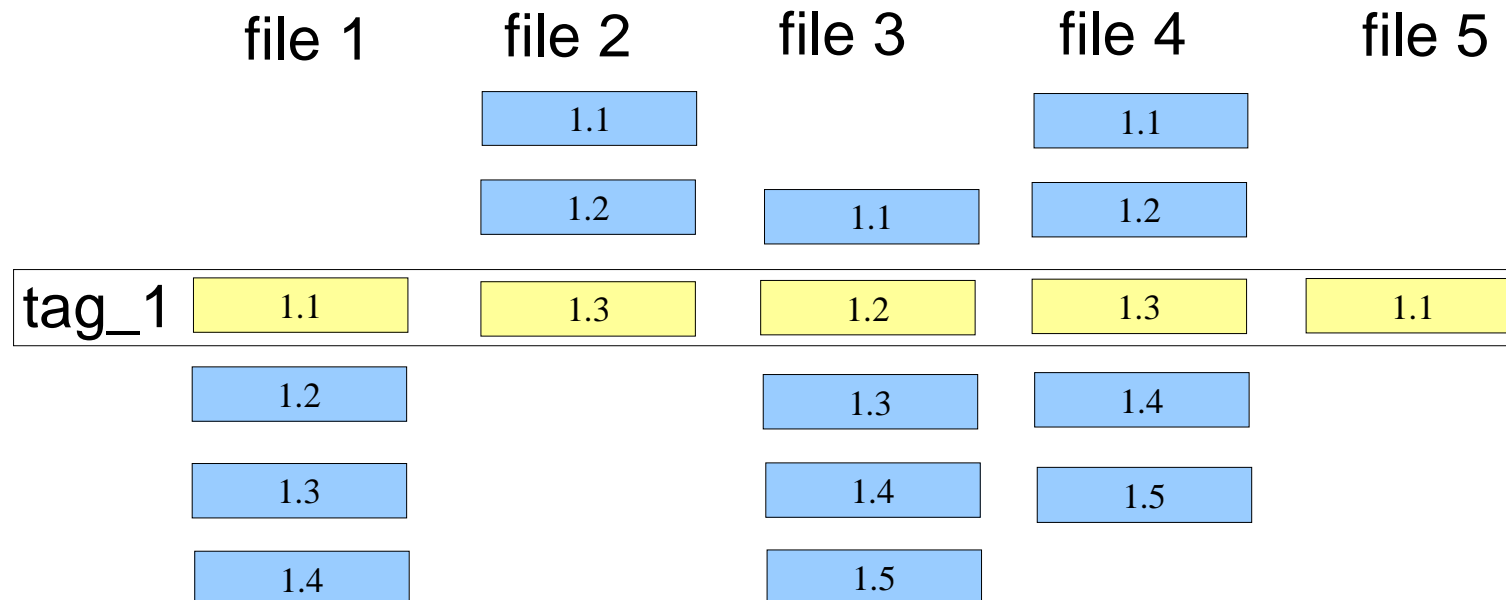


- Merge: Conflict
⇒ re- edit



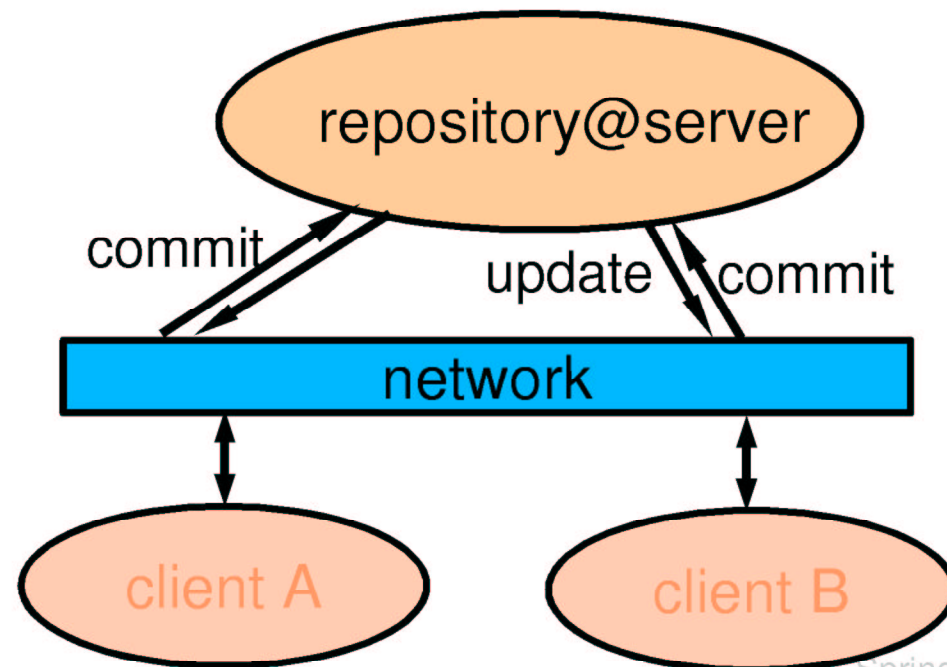
Vision Magement

- Tagging = symbolic names for
(collections of) versions
(=configurations)



Distributed Version Management

- Access to the Repository via Internet
 - Via Serverprocesses (e.g. CVS: demons, Shell)
 - Via NFS (z.B. ClearCase; mouted "working copies")
- Problem:
Access Control
Security
(e.g. CVS: kerberos)
- ClearCase:
Distributed
Repositories



Configuration Management (1)

- Towards the Generation of Configurations:
produce compiler with coder X on platform Y
 - requires dependency model,
change propagation,
control of the build process
(like make, but also event–controlled (ClearCase))
 - work flow management
- Systems: Aegis, ClearCase, StarTeam, . . .

Configuration Management (2)

- Pro's:
 - best "reconstructibility"
 - effektive management of many configurations possible
 - file-systemstructure can be versioned too (rename/move of directories)
 - data dependency model can be versioned, which can be exploited semantically (change propagation)

An Example: CVS

- Concurrent Version System
 - repository remote
 - multiple working copies without lock
 - terminology: "Version" \Rightarrow "Revision"
 - open source standard, Unix-add-on,
 - registration of configurations (tags)

CVS – Basic Operations

`cv`s <*cv*s-*options*> <*cv*s-*commands*> <*cmd*-*options*> <*args*>

	<code>add</code>	<code>-m msg</code>	<i>file</i>
	<code>commit</code>	<code>-m msg -r branch</code>	<i>file</i>
	<code>checkout</code>	<code>-j tag -r tag -A</code>	<i>module</i>
	<code>update</code>	<code>-d -A -r tag -j rev</code>	<i>file</i>
	<code>diff</code>	<code>-r rev -D date</code>	<i>file</i>
	<code>tag</code>	<code>-d tag -F -R</code>	<i>file</i>
	<code>rtag</code>	<code>-r tag -b</code>	<i>module</i>
	<code>status</code>	<code>-R -v</code>	<i>file</i>
	<code>admin</code>	<code>-m rev:msg -o rev</code>	<i>file</i>
	<code>init</code>		
<code>cv</code> s	<code>-d dir</code>		
	<code>-e edit</code>		
	<code>-H</code>		

CVS – Pragmatics: Overview

- Create
 - working copy (initial)
 - file/directory
 - project/module
 - repository
- Synchronyse
 - working copy (normal)
 - modules, branches
- Manage
 - difference
 - modifications, delete
 - register configurationen
 - status
- Diversities
 - notifications
 - wrappers
 - logs

CVS – Pragmatics: Create

- Creation of a working copy

```
mkdir work; cd work;  
cvs -d path checkout .  
rm project1 project2 project3
```

or

```
mkdir work; cd work;  
cvs -d path checkout softech-www  
cvs -d path checkout CVSROOT
```

CVS – Pragmatics: Create

- *Creation of Files/Directories*

```
cd work/softech
vi bla                (* create and edit bla *)
cvs add bla
cvs commit bla
```

resp.

```
cd work/softech
mkdir blubdir
cvs add blubdir
ls blubdir
> CVS . . .
```

- *Creation of Subsystems (for extern src only!)*

```
cvs import bla bla fdg
```

CVS – Pragmatics: Synchronisation

- of files:

```
vi bla                (* modify bla *)
cvs ci bla
>cvs commit: Up-to-date check failed for 'bla'
>cvs [commit aborted]: correct above errors first
cvs update bla
> U bla
vi bla
cvs ci bla
```

- of branches:

```
cvs update -j R1fix:yesterday -j 1.2 bla
cvs co -j R1fix mod
```

CVS – Pragmatics: Manage

- Status, Logs

```
cv$ status -v  
cv$ log bla
```

- Differences

```
cv$ diff -r 1.14 -r 1.16 bla  
cv$ diff -r RELEASE1 -r RELEASE2 > diffs
```

- Delete (for administrators only)

```
cv$ admin -o 1.1:1.5 bla
```

CVS – Pragmatics: Diverse

- Notifications (Loginfo)
- Wrappers
- Comment Headers
- Shell-Variablen

CVS–Weaknesses

- non–elementary use complex, some default–options is tricky.
- roles und perms: UNIX–Groups, Password
- architectures (filetrees) of configurationen and dependencies were not covered (filetree must grow monotonically)

Version Management: Conclusion & Outlook

- version management is a key technology in SE
- VM useful, but introduces also some problems and some bureaucracy
- a step to configuration management

Version Management: Conclusion & Outlook

- study of merge –operations of specific formats, which are of interest:
 - code
 - text from word processors
 - proof objects

Semi-formal Modeling Languages

David Basin

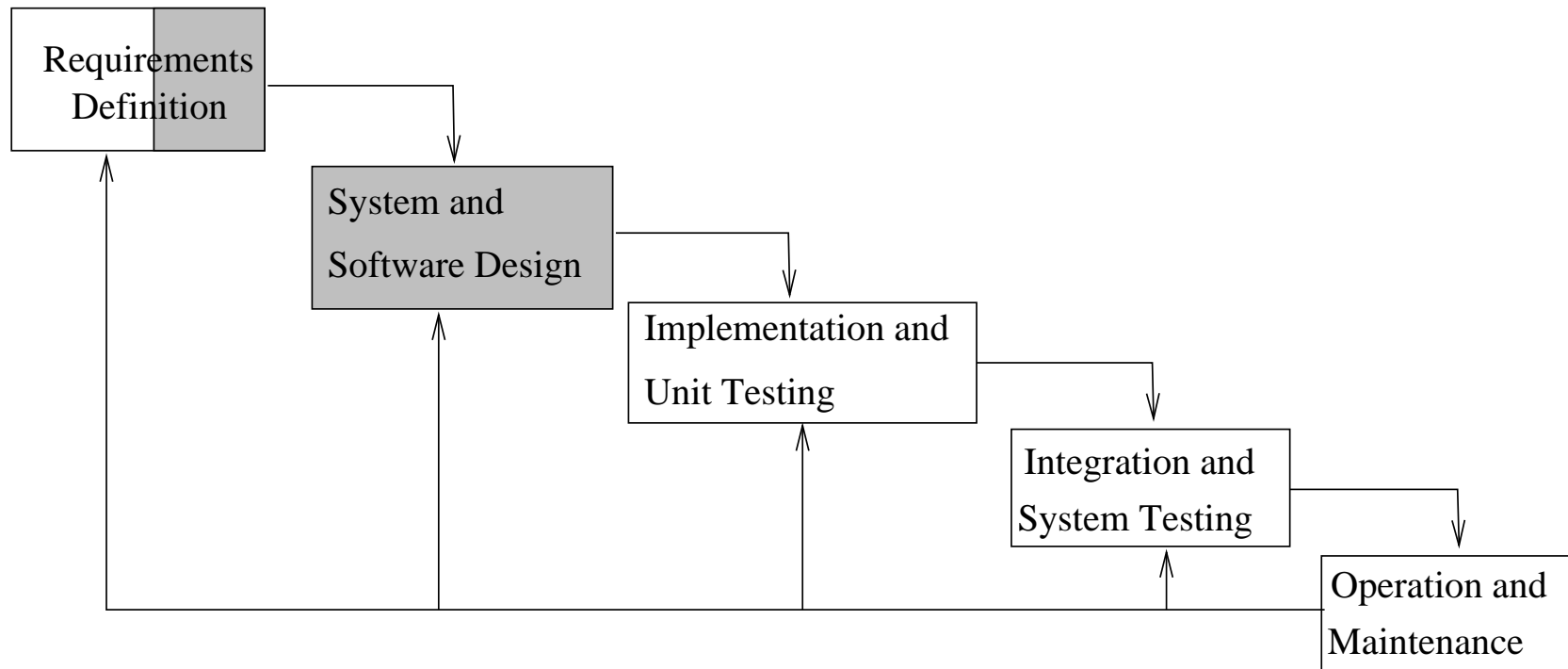
Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Spring 2002

Overview

- Motivation
- A simple example
- OO-modeling languages and methods (2 weeks)
 - Ideas
 - Some languages/methods
 - Use, advantages, and limitations of semi-formal approaches.

Context



Goal: Specify the requirements as **far** as possible, but as **abstract** as possible.

Modeling languages and methods are used here!

Modeling

- **Definition:** A **model** is a construction or mathematical object that describes a system or its properties.
 - Example from physics: $\text{distance} = \text{speed} \times \text{time}$
 - A construction engineer models buildings and employs static models (e.g., of stress and strains) for their analysis
- In computer science we model systems, their operating environment, and their intended properties

These models aid requirements analysis, design, and analysis of systems
- The construction of models is the Raison-d'être of planning

⇒ research on modeling languages and modeling methodologies

Motto: Engineers build models, so should software engineers!

Which Modeling Language?

- There are hundreds! Differences include:

System view: static, dynamic, functional, object-oriented, . . .

Degree of abstraction: e.g., requirements versus system architecture

Formality: Informal, semi-formal, formal

Religion: OO-school (OOA/OOD, OMT, Fusion, UML), algebraic specification, Oxford Z/CSP-Sect, Church of HOL, . . .

- Examples:

Function trees, data-flow diagrams, E/R diagrams, syntax diagrams, data dictionaries, pseudocode, rules, decision tables, (variants of) automata, petri-nets, class diagrams, CRC-cards, message sequence charts, . . .

Methodologies often mix languages

Z/CSP:

- 2 models: functional and dynamic/event oriented
- 2 languages

OMT: Object Modeling Technique, Rumbaugh et. al.

- 3 models: object, dynamic, functional
- 3 languages: class diagrams, statecharts, data-flow diagrams

Unified Modeling Language: – 9 languages: class, object, use cases, sequence, collaboration, statecharts, activities, component, and deployment

We will start with three simple examples: E/R-Diagrams, data-flow diagrams, and class models.

Entity/Relationship Modeling (E/R)

- Specifies sets of (similar) data and their relationships

Relationships are typically stored as tables in a data-base

- Three kinds of 'objects' are visually specified



Entities: sets of individual objects, differing in their properties

Attributes: a common property of all objects in an entity set

Relations: ('semantic') relationships between entities

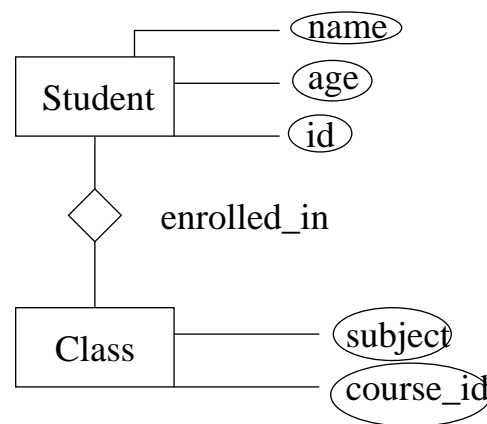
E/R example

- Student & Class entities
 - A *Student* has a *name*, *age*, and *identity*
 - A *Class* has a *subject* and *course_id*
 - In programming languages: attributes \mapsto basis types, entities \mapsto record types

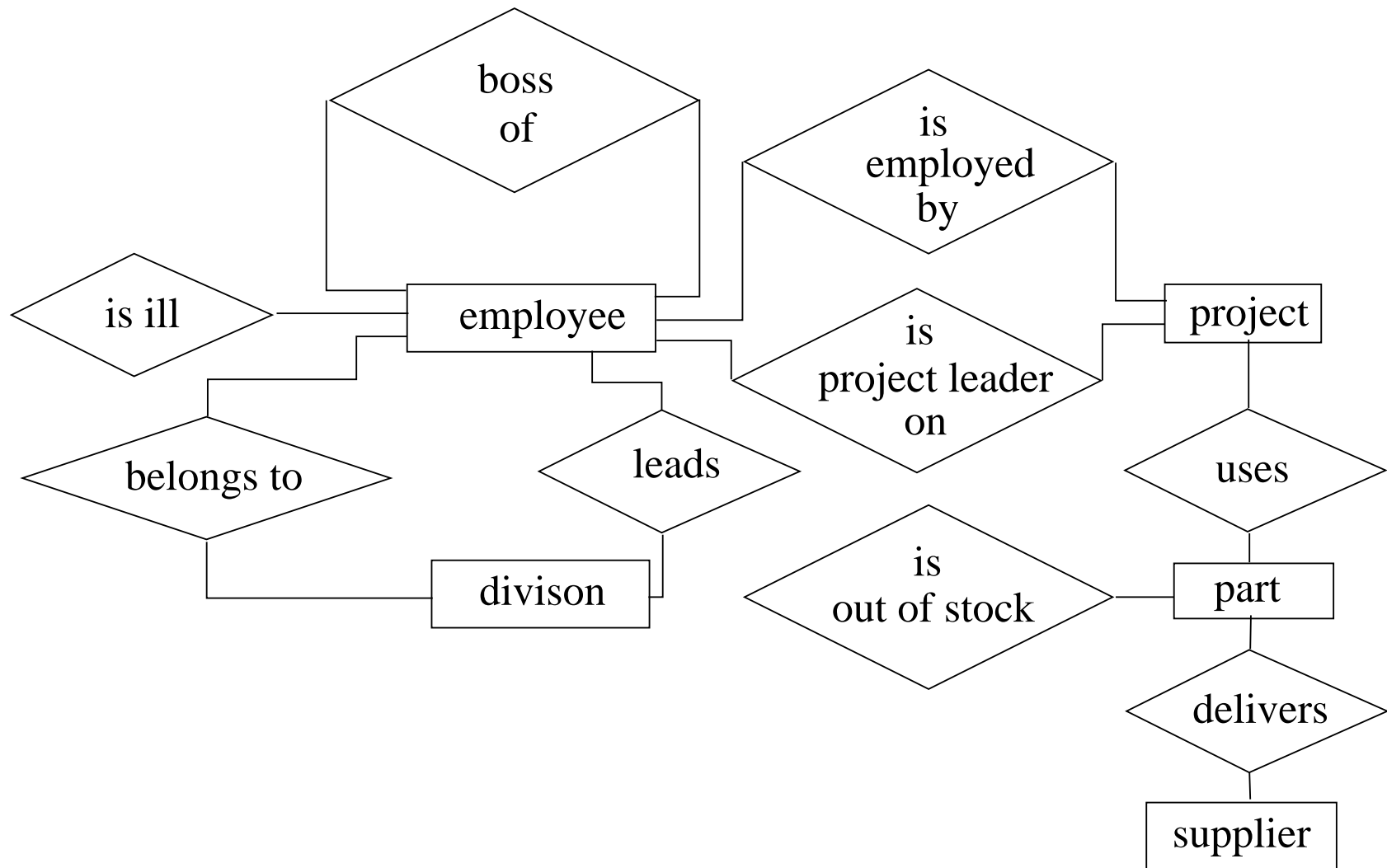
type *Student* = *student*(*name* : *string*, *age* : \mathbb{N} , *id* : \mathbb{N})

type *Class* = *class*(*subject* : *string*, *course_id* : *string*)

- Relations are graphically represented, e.g., Student *enrolled_in* Class



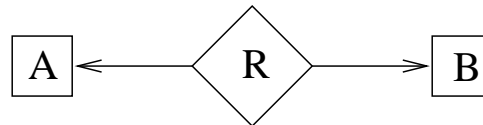
E/R: a larger example (without attributes)



E/R: advantages and disadvantages

- ++ 3 concepts and pictures \implies easy to understand
- ++ Tool supported and successful in practice.
- Not standardized
 - Are relations binary or n -ary? OO extensions (e.g. *is_a*)?
 - Notation for semantic conditions? E.g., R is injective:

$$\forall x \ x' \ y. (x \ R \ y \wedge x' \ R \ y) \Rightarrow x = x'$$

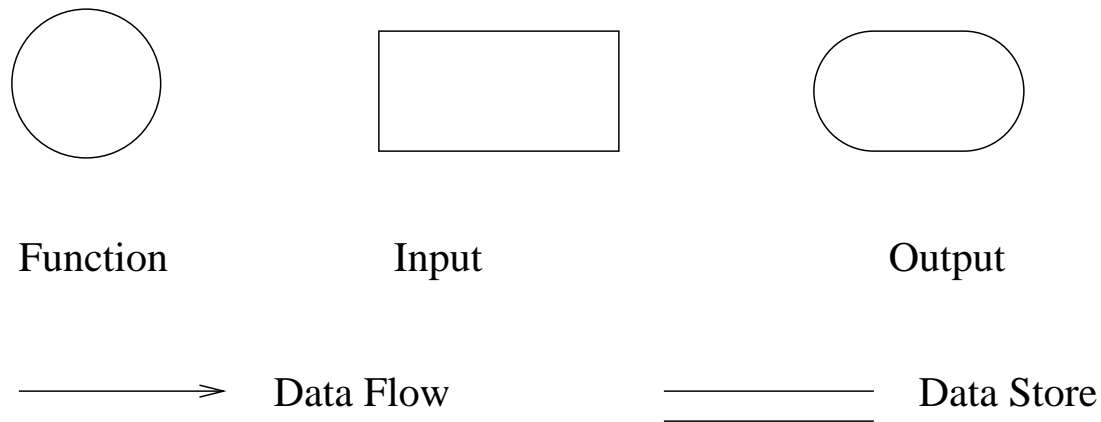


- Many relations cannot be specified
 - Every n -ary function corresponds to an $n+1$ -ary relation

For more, see data-base classes!

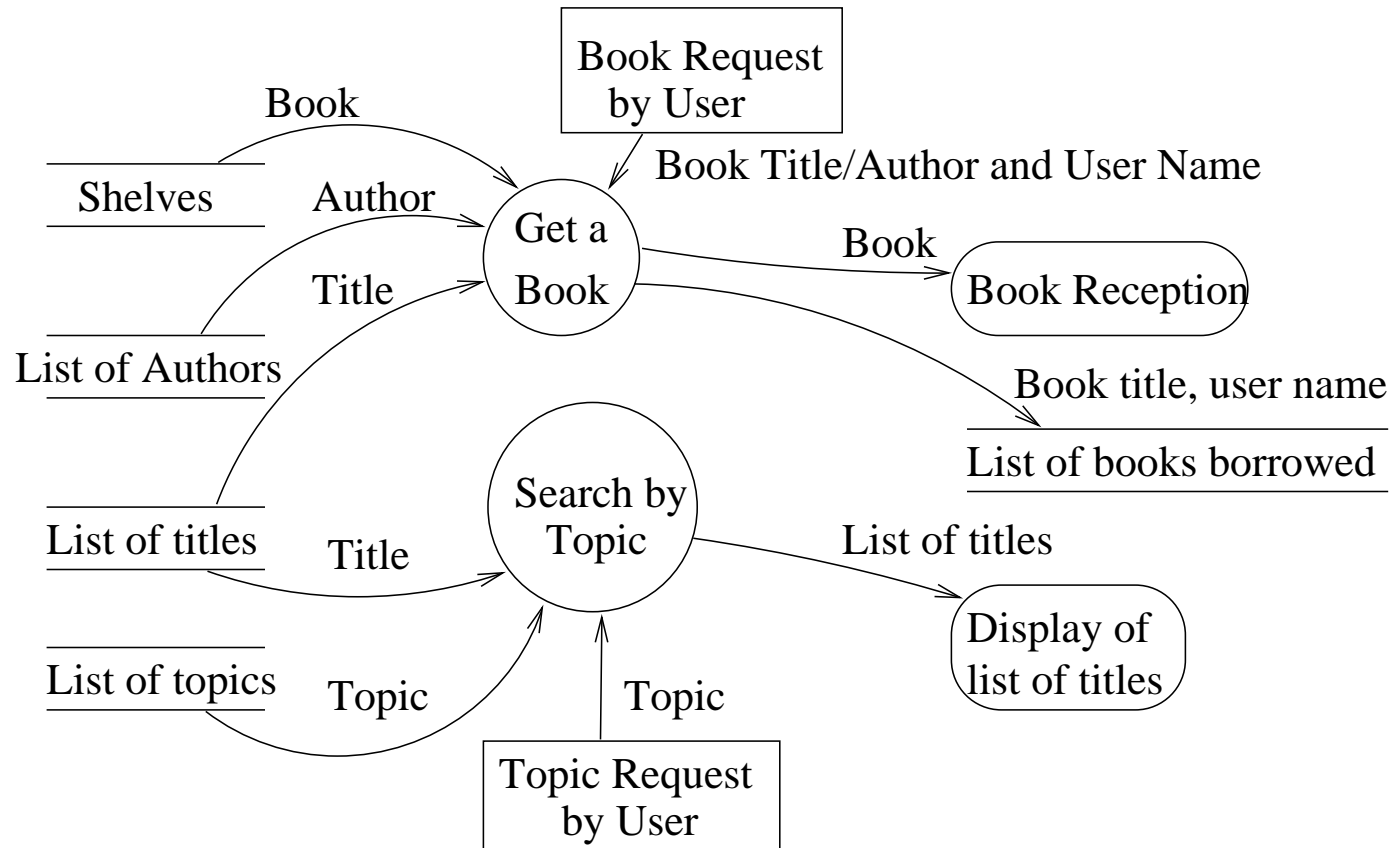
Data-flow diagrams

- Graphical specification language for functions and data-flow
- Based on symbols (not standardized):



- Useful for requirements plan and system definition
- Provides a high-level system description that can later be refined

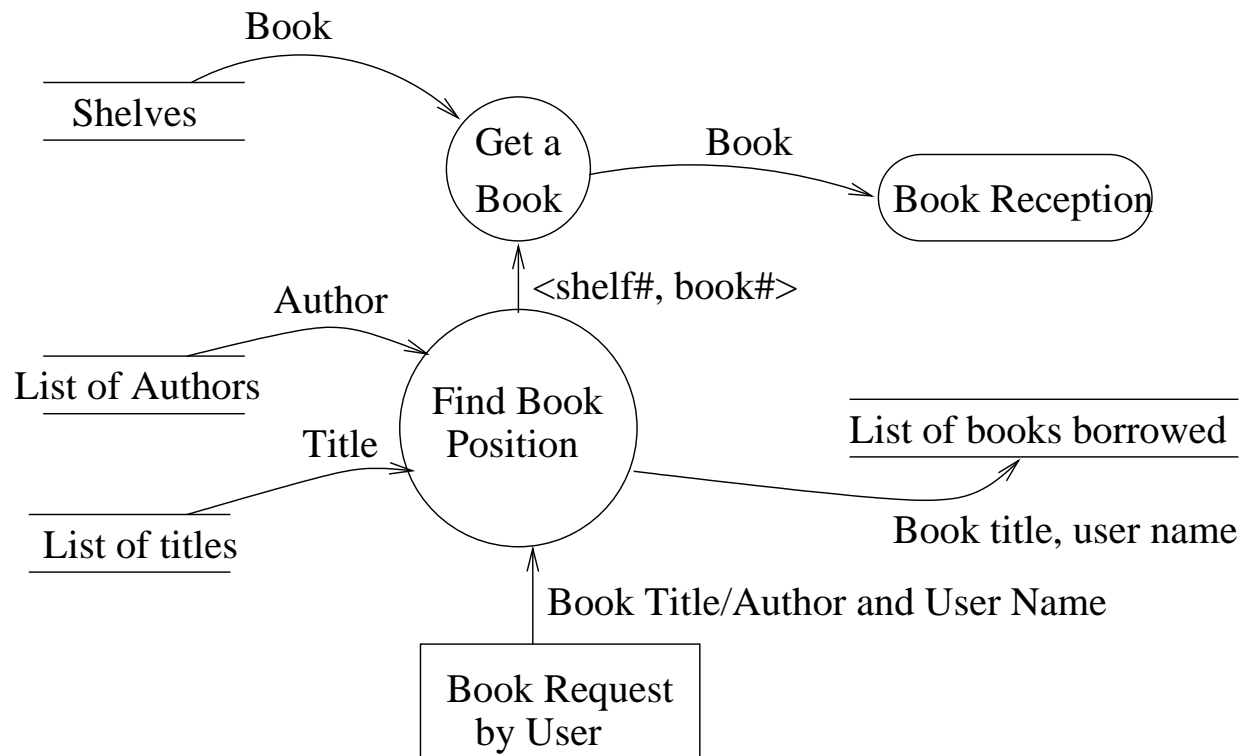
Example: library information system¹



First approximation. Unspecified how books are found, etc.

¹Source: Sommerville

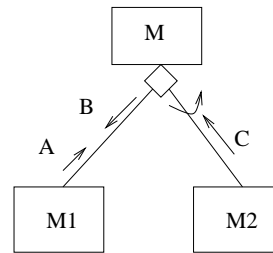
Partial DFD-Refinement



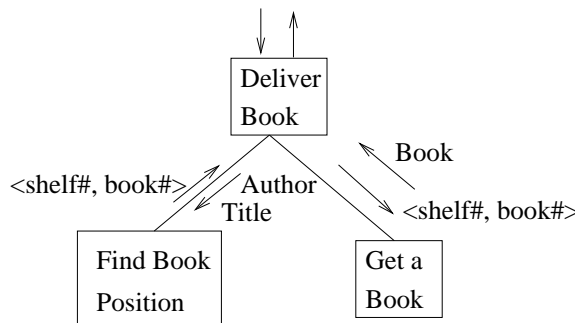
- Describes how a book is selected
 - Still inexact. Are both title and name needed?
 - Semantics suggested by the function names
 - Control open. Execution scheduling is not specified

Hierarchical DFDs & development methods

- Hierarchical DFDs yield a module structure



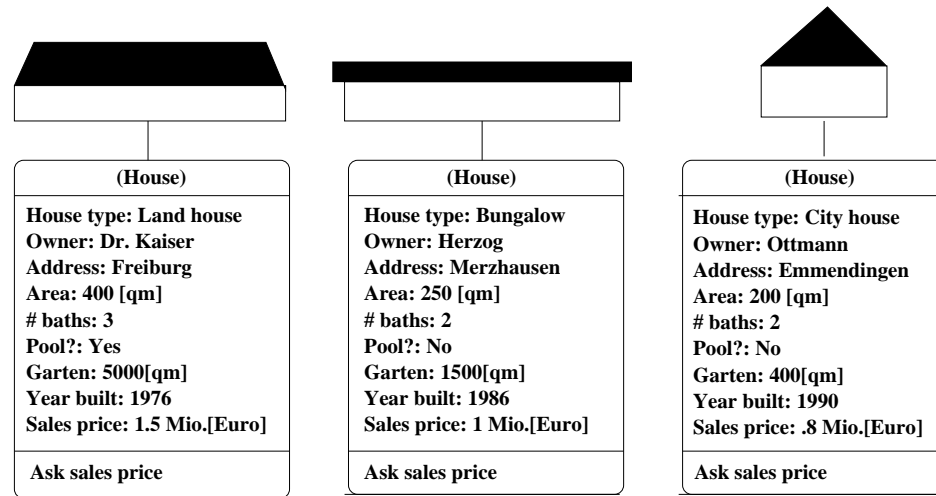
- Procedure M calls either M_1 once or M_2 multiple times
 - M passes B to M_1 and receives A back and M receives C from M_2
- Example: architecture of the module **order book**.



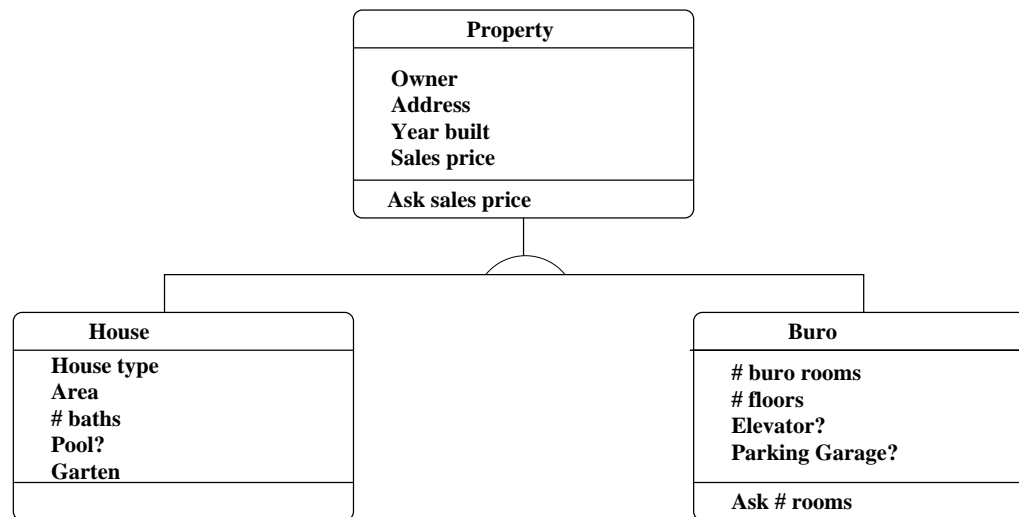
- Can be supported by CASE-tools, e.g., for the automatic generation of classes or (module) signatures.

Class models (High-Level)

- **Emphasis:** objects and relationships **Goal:** clarity and reusability
- **Objects** are grouped into **classes** depending on their **attributes** and **methods**



- **Class hierarchy** formalizes **inheritance**, expresses shared attributes/methods.



OO-Modeling

- Many proposed methodologies: OMT, OOA/OOD, UML, ...

- **Example: Object Modeling Technique (OMT)** includes:

Object model: class hierarchies and associations between classes (like E/R)

Functional model: describes information flow between objects (like data-flow)

Dynamic Model: event oriented, e.g. (extended) automata (specify control)

- Advantages
 - Supports modular development of reusable systems
 - Often corresponds to real (e.g., graphic) objects
- Disadvantage: semantics not so clear

History of modeling languages

- Different branches: formal versus semi-formal

Formal: research since the 1960s. VDM, Z, algebraic specification, process algebras, . . .

Semi-formal: E/R-diagrams, data-flow diagrams, . . . from the 1970s

- Short history of OO modeling
 - Proliferation of methods from 1970–1990 (OOA/OOD, OMT, OOSE, ...)
 - **Method war** between the different languages and methods
 - **Campaign** of the “3 Amigos” (now all at Rational Software Corporation)
 - Grady Booch’s OOD
 - James Rumbaugh’s OMT
 - Ivar Jacobson’s OOSE (based on “Use Cases”) and Objectory
 - Standardization in the **Unified Modeling Language**, 1995–???

UML overview

What? 9 languages for modeling different views of systems

- **Use Case models** describe the users' view of the system
- **Static models** describe parts of the system and their relationships
- **Dynamic models** describe the (temporal) behavior of the system

Why?

- **De facto** standard!
- Tool support (e.g., from Rational)
- More-or-less intuitive

Why not?

- Semantics? What do the models **mean**?
- Support for analysis is (currently) weak
- Complicated and cryptic bits resulting from committee driven development

UML: Overview (cont.)

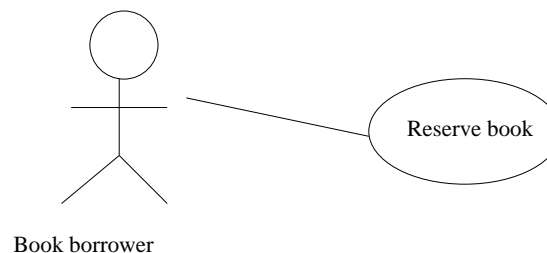
Major Area	View	Diagrams	Main Concepts
structural	static view	class diagram	class, association, generalization, dependency, realization, interface
	use case view	use case diagram	use case, actor, association, extend, include, use case generalization
	implementation view	component diagram	component, interface, dependency, realization
	deployment view	deployment diagram	node, component, dependency, location
dynamic	state machine view	statechart diagram	state, event, transition, action
	activity view	activity diagram	state, activity, completion transition, fork, join
	interaction view	sequence diagram	interaction, object, message, activation
		collaboration diagram	collaboration, interaction, collaboration role, message
model management	model management	class diagram	package, subsystem, model
extensibility	all	all	constraint, stereotype, tagged values

UML: Overview (cont.)

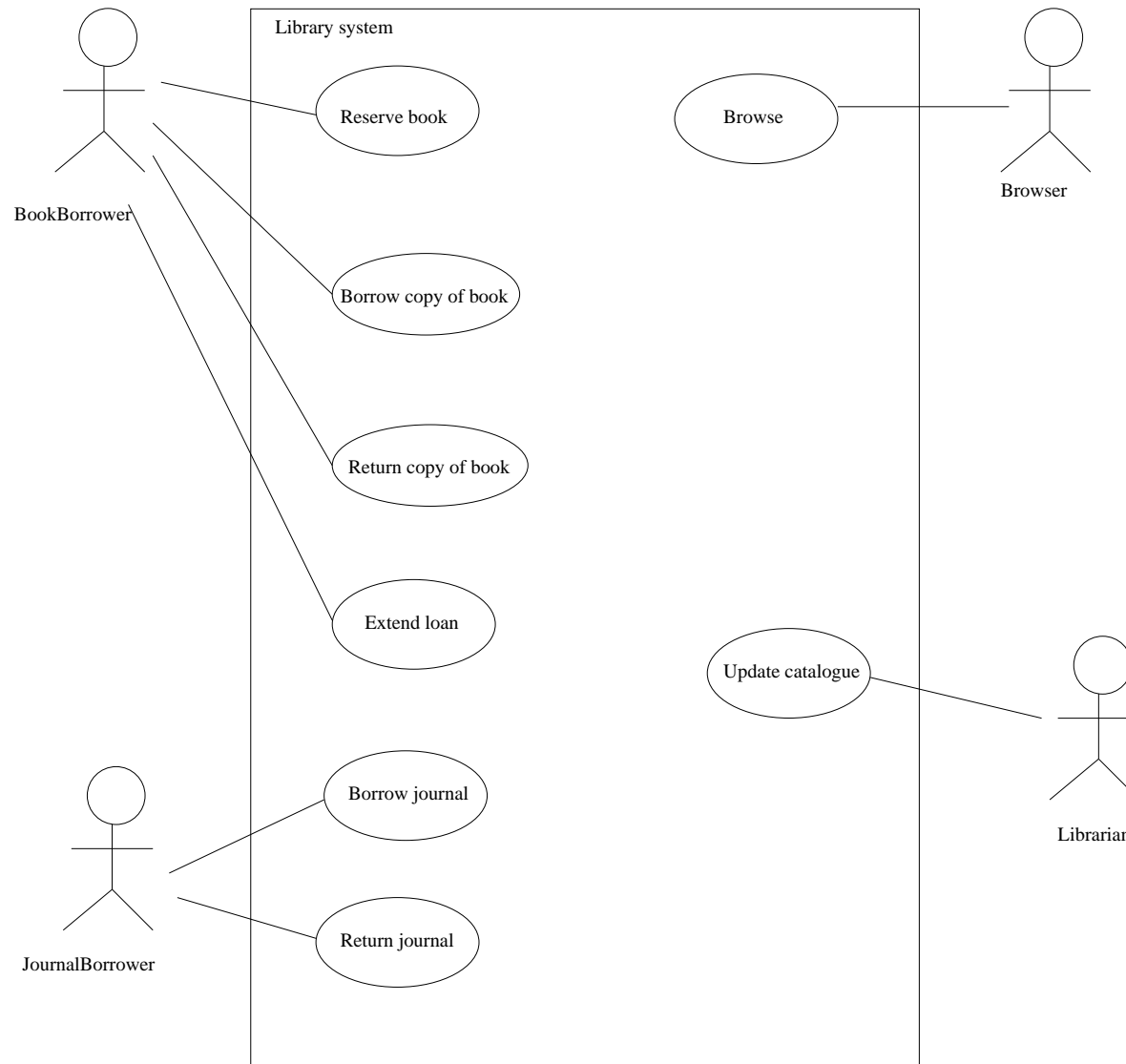
- UML is a mixed blessing
 - Building models is a fine thing!
 - UML is but one of many possibilities
 - More appropriate for requirements analysis than design
- We will consider only parts of UML
 - In particular: Use Cases, Class Models, Statecharts
 - Full details \Rightarrow class on OO-development
- Let's start with Use Cases!

Use Cases

- Use Cases are used for high-level requirements analysis
 - Focuses on **requirements** of **particular (classes of) users**
 - A user is anything outside of the system (e.g., human or another system)
- Developed from Jacobson (ca. 1990), based on idea of **scenarios**
- **Incredibly simple!** Diagrams represent possible interactions between:
 - Actors:** prototypical users in certain roles
 - Use Cases:** their tasks
- **Example:** a library system Use Case



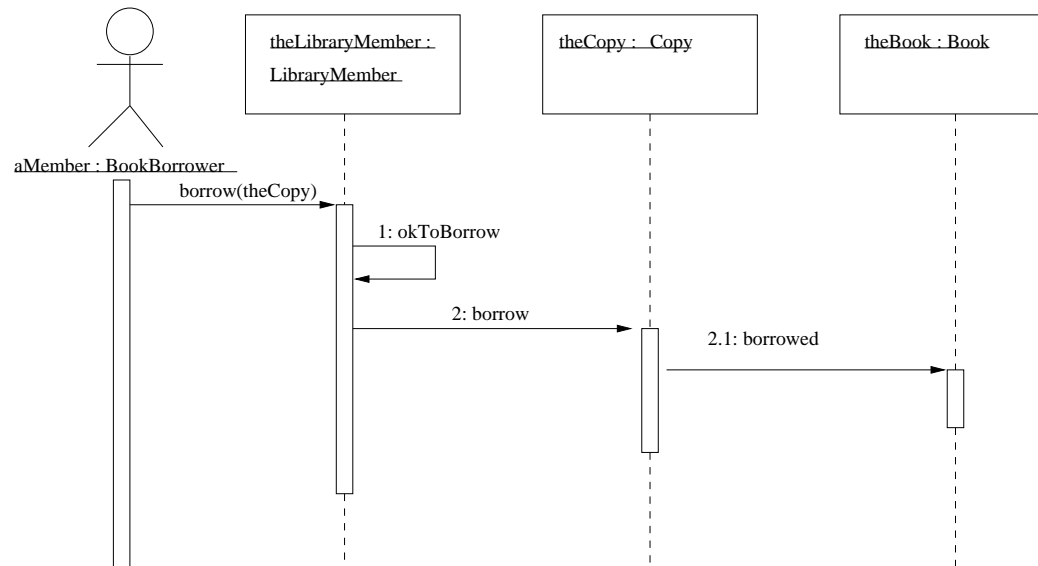
Use Case Diagrams



Collection of Use Cases (+ scenarios) describes the system functionality

Use Cases (cont.)

- Use Cases are extended with text or other models
 - Extensions describe steps and results



Sequence diagram describes a possible execution in a loan scenario

- How does one identify Use Cases?

Identify services that the system provide that yield results of value.
(Use Case shouldn't be too trivial.)

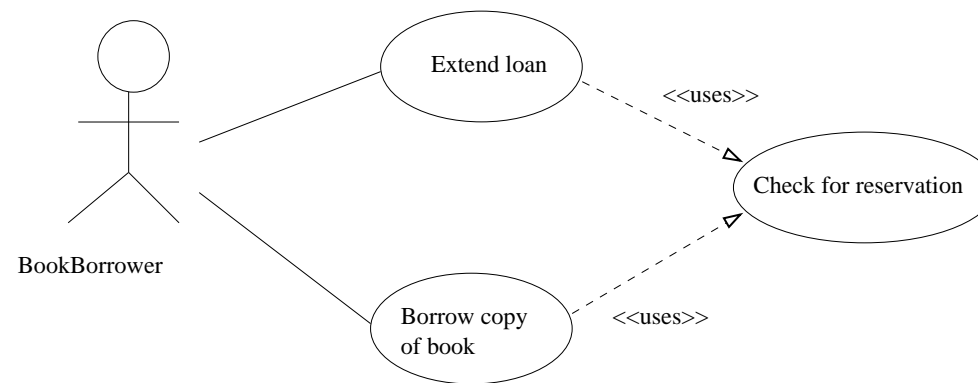
Use Cases: possibilities for further structuring

- Development **in the large** requires structuring models!
- Use Cases offer several possibilities.
 - Extensions provide a preview of the UML-game!
- Tradeoff: More complicated modeling languages are less intuitive!

Moreover, changes/extensions tend to vary over time
 \rightsquigarrow the UML Standardization problem

Reuse via <<uses>>

- Common behavior can be factorized
- **Example:** Both “Extend loan” and “Borrow copy of book” must check if the book is reserved



- Describes the “source” Use Case used by the “target” Use Case
- Syntax matters!
 - directed arrow indicates dependency
 - *stereotype* <<uses>> classifies the kind of dependency
- Advantage: smaller models and discovery of possible reuse.

<<uses>> — Example

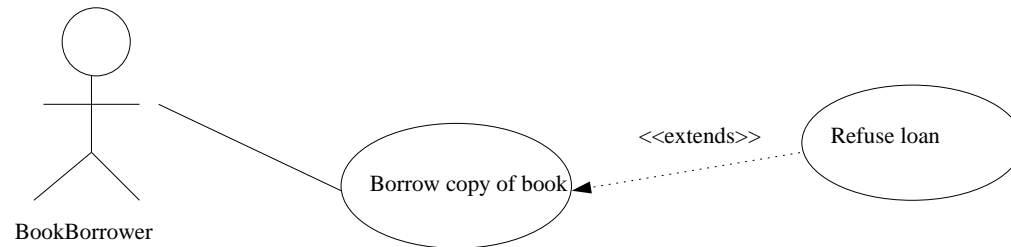
Borrow copy of book: A BookBorrower presents a book. The system checks that the potential borrower is a member of the library, and that s/he does not already have the maximum number of books on loan. The maximum is 6 unless the member is a staff member, in which case it is 12. If both checks succeed, the system checks if the book is reserved (use case Check for reservation), in which case the system refuses to lend the book. Otherwise it records that this library member has a copy of the book on loan and prompts for the book to be stamped with the return date.

Extend loan: A BookBorrower asks (in person or by telephone) to extend the loan of a book. The system checks whether there is a reservation of the book (use case Check for reservation. If so ...

Check for reservation: Given a copy of the book, the system searches the list of outstanding reservations for reservations on this book. If it finds any, it compares the number of them, n , with the number of copies m , known to be on the reserved bookshelf. If $n > m$ then the system returns that this copy is reserved, otherwise that it is not.

Exceptional behavior: <<extend>>

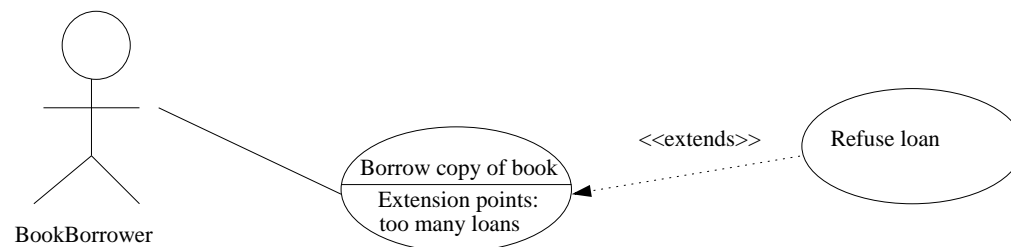
- Can be used to factor different behavior into a single scenario



- Syntax: dashed arrow now from exception to main case!
- Scenario description explains:

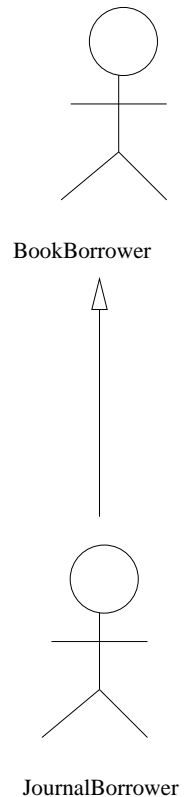
Condition: when the exception holds

Extension Point: when the condition should be checked (optional)

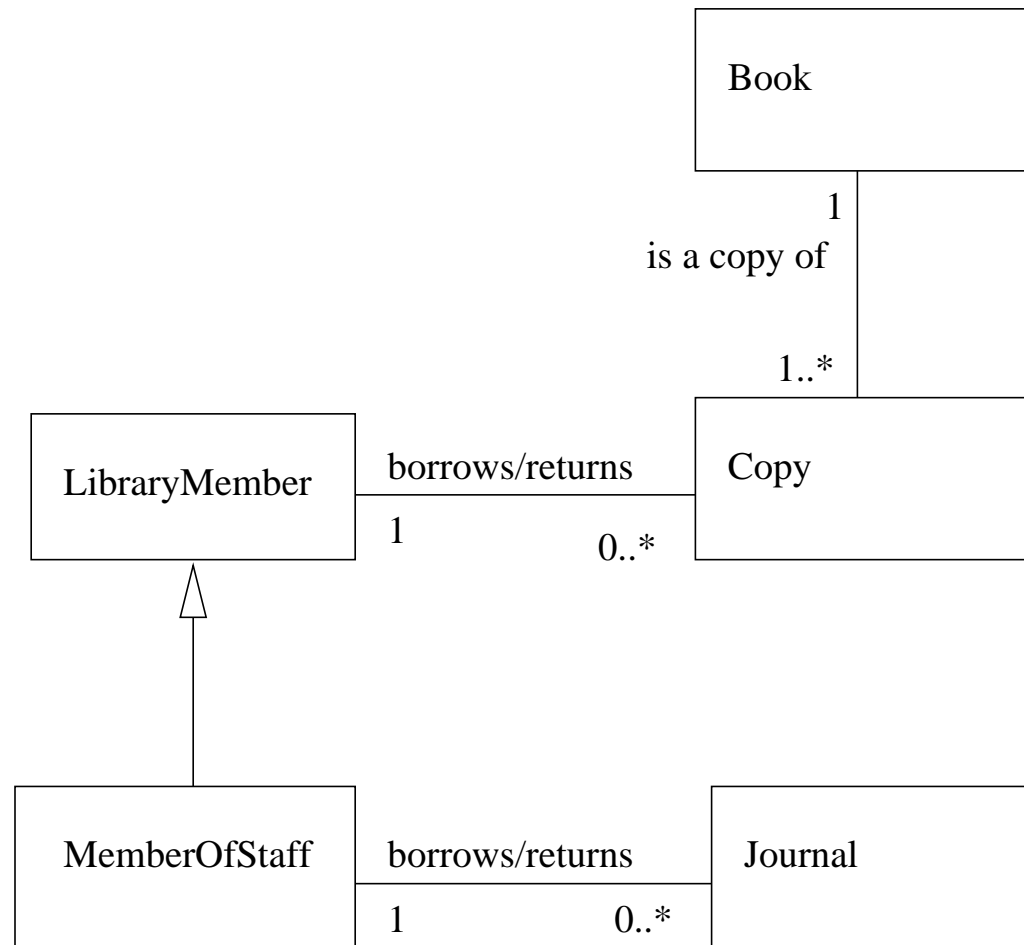


Generalization

- Generalization relates two actors or two Use Cases to each other
- **Example:** each JournalBorrower is a BookBorrower.
- Semantics: Every Bookborrower-Use Case holds for each JournalBorrower.
- Specifies additional functionality. E.g., reservation generalizes telephone-reservation



We will see more of this notation later



Conclusion

- Use Cases models useful for initial requirements analysis

Question: What isn't covered?

- Idea is simple, but effective in stimulating analysis and provides simple structuring mechanisms
- Further extensions are possible:
Different icons for actors, interfaces, processes, ...
- Extensions show the spirit of UML, as well as advantages and disadvantages

Modeling Classes

David Basin

Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Spring 2002

Overview

- We shall continue our study of semi-formal specification languages
- Today: UML class-diagrams
 - Definition, motivation, application area
 - Variants and extensions
- **Goal:** gain insight into a popular, semiformal modeling language and its application to modeling static, structural aspects of systems

Basic definitions and concepts

- In object oriented modeling, **objects** are the main unit of abstraction
 - Used both for modeling requirements, design, and implementation
- In an object oriented model:
 - Objects carry out **activities**
 - Interface to objects is **event oriented**
 - **Example:** A robot has sensors, actuators, control units, etc.
- Comparison: functional decomposition
 - Decomposition of problem into functions (rather than activities)
 - Interface is data-oriented (input/output of functions)
 - **Example:** A compiler has a parser, code generator, . . .

Definitions and concepts (cont.)

- One **interacts** with objects. An object has:
 - State:** Encapsulated data. Consists of **attributes** or **instance variables**. Part of the state can be **mutable**.
 - Behavior:** An object reacts to **messages** by changing its state and generating further messages.
 - Identity:** An object is more than a set of values and methods. It has an existence and a name.
- **Def:** An **interface** defines which messages an object can receive.
 - Describes behavior without describing implementation or state
 - Often one differentiates between **public interfaces**, which all objects can use, and **private interfaces**, that (only) the object itself can use.
- An example: a watch interface (+ denotes public)
 - + reportTime() : Time
 - + resetTimeTo(newTime:Time)

Definitions and concepts (cont.)

- A **class** describes objects with similar structure and behavior

```
class Point {  
    int x = 0, y = 0;  
    void move(int dx, int dy)  x += dx; y += dy;    }
```

- A class has a fixed interface and defines attributes and methods
- Advantages of classes

Conceptual: Many objects share similarities. E.g., the 10,000+ bank customers

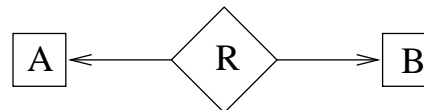
Implementation: Only one implementation

Further advantages: (to be presented later)

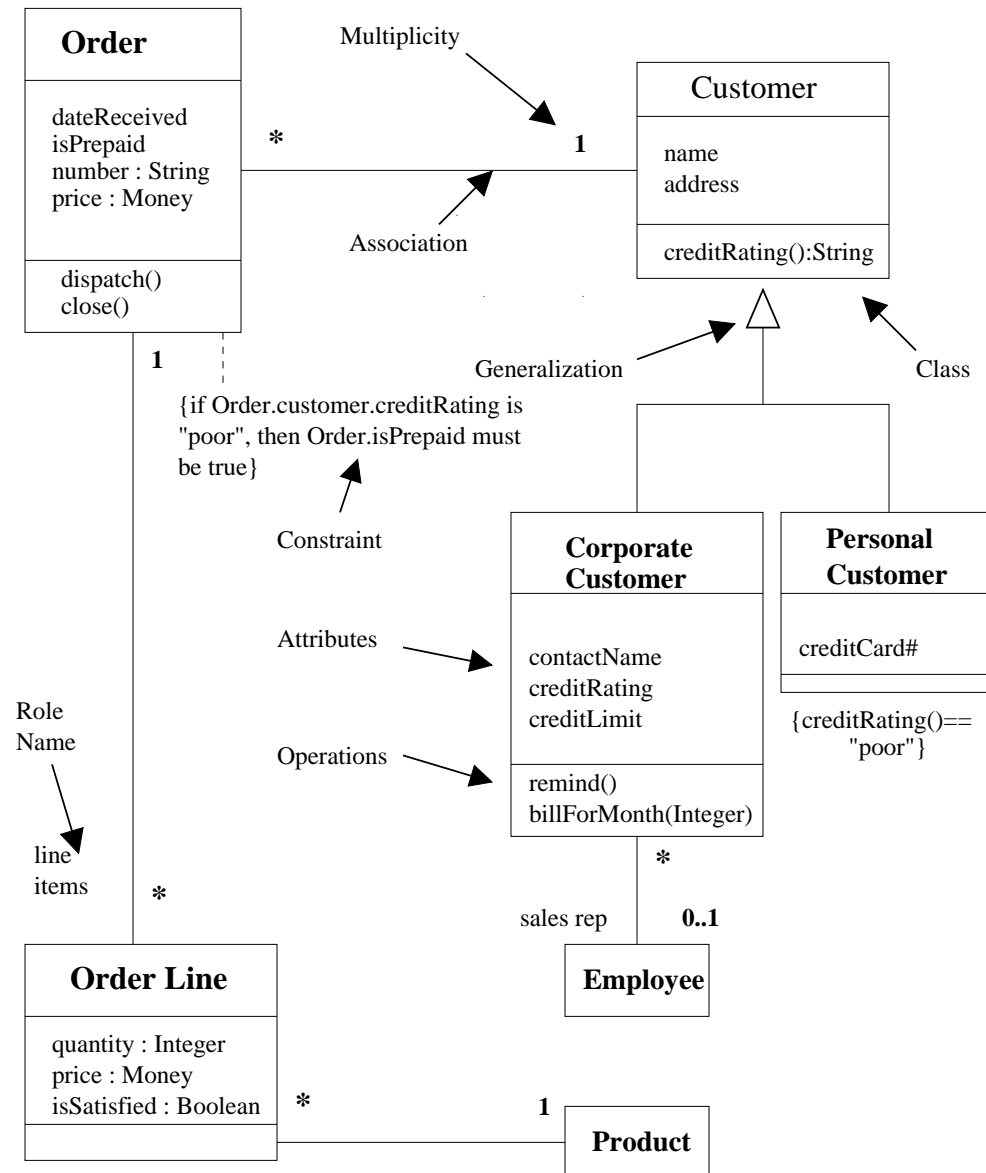
- **Inheritance** or **overriding** of methods
- **Dynamic binding**, where method implementations determined at run-time

Today's topic: Class diagrams

- Language to model the **static view** of a system
- A **class diagram** describes the **kind of objects** in a system and their different **static relationships**.
- Kinds of relationships include
 - Associations:** e.g., can a customer rent videos?
 - Subtypes:** e.g. is a nurse a person?
- Central model of object oriented analysis with the largest applicability
- Generalizes other kinds of static modeling languages, e.g. E/R diagrams



First an example



Reasons for using class diagrams

Conceptual: represents concepts in application domain.

Independent of an implementation.

Specification: Specifies interfaces and gives hints to the semantics.

Implementation: Describes what must be implemented.

- UML does not distinguish between these uses! Class diagrams may be employed for different purposes.
- Example: in requirements modeling one omits many details. Decisions like where state and behavior are localized, how one navigates between objects, etc. are given later in development models.

What are classes?

A class describes a **set of objects** with **equivalent roles** in a system. Examples:

Tangible, real-world things: Airplanes, computers, beer kegs, . . .

Roles: Library member, student, teacher, . . .

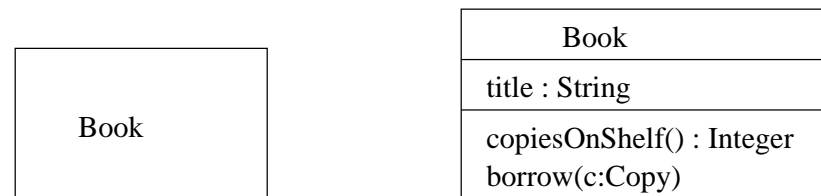
Business things: Orders, accounts, . . .

Application things: Power-on buttons, . . .

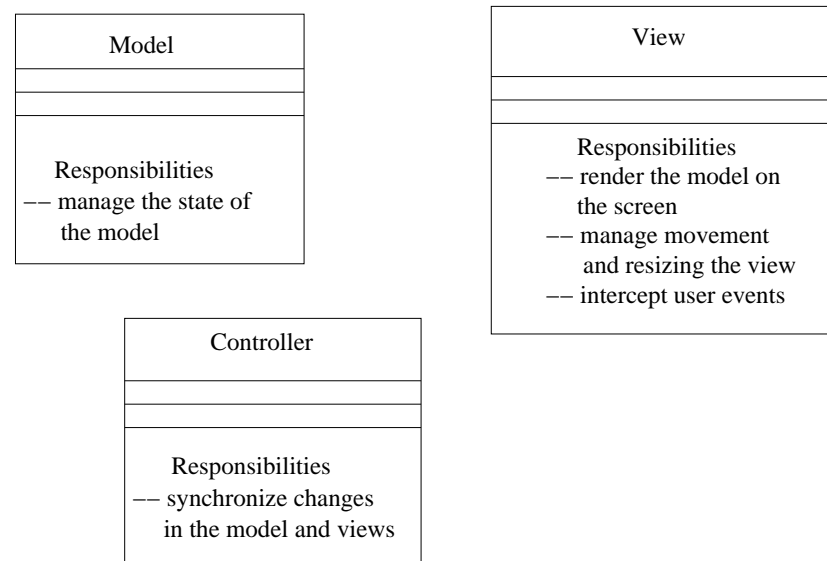
Data structures: Lists, hash-tables, . . .

Visual representation of classes

- A class is represented as a square, with optional attributes and operations



- **Attributes** define the state (data values) of the object
- **Operations** (or **methods**) define how objects effect each other
- One can also specify **responsibilities**



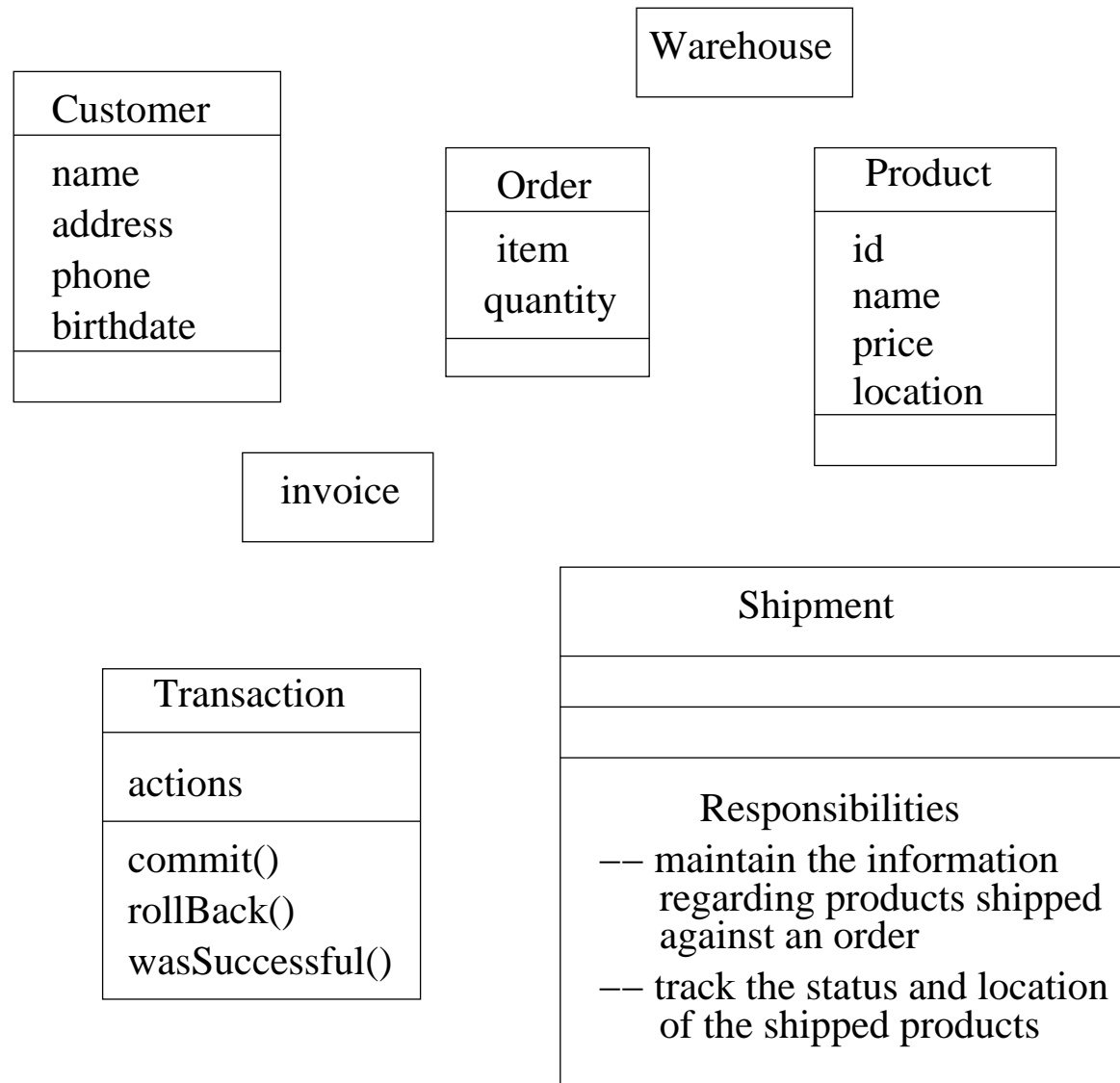
Identifying classes

- What constitutes a good class model?
 - The objects should satisfy the desired requirements
 - Classes should represent significant classes of objects in the domain, in order to improve maintainability and reusability
- How does one create a good class model? One possibility is:
Divine inspiration: When the result is good, everyone is happy!
- But classes have data and responsibilities. Other possibilities are:
Data driven design: Identify all system data and divide it into classes.
Afterwards consider operations.
Responsibility driven design: Start with the operations or even the responsibilities (as in the last example).

Procedure I: classes as nouns

- Two step procedure for identifying classes:
 1. **Identify** possible classes. These are the nouns and noun-phrases used in the requirements analysis. Use the singular.
 2. **Consolidate** the results. Delete those whose name is
 - **redundant** (one of many equivalent names)
 - **unclear** (alternative: further clarify)
 - an **event** or an **operation** (without state, behavior, and identity)
 - a simple **attribute**
 - **outside of the system scope**, e.g. a library system probably doesn't require a library class. Don't forget, the objects **are** the system!
- Experience, imagination, and patience are helpful

Brainstorming — example



Procedure II: CRC Cards

- Alternative approach suggested by Beck & Cunningham 1989 (pre-UML)
- For each class one notes on a card:
 - Class:** Name
 - Responsibilities:** of objects of the class
 - Collaborators:** helpers that aid in fulfilling the responsibilities
- If there are too many responsibilities or collaborators, create new classes!
- Distribute the card to the developing team and choose a Use Case scenario
 - Play the scenario through, switching, as necessary, between the collaborators
 - In doing so, discover missing responsibilities or collaborators
- Afterwards, add attributes and methods

Brainstorming with CRC-Karten

LIBRARYMEMBER	
Responsibilities	Collaborators
Maintain data about copies currently borrowed Meet requests to borrow and return copies	COPY

COPY	
Responsibilities	Collaborators
Maintain data about a particular book copy Inform corresponding BOOK when borrowed and returned	BOOK

BOOK	
Responsibilities	Collaborators
Maintain data about one book Know whether there are borrowable copies	

Relations

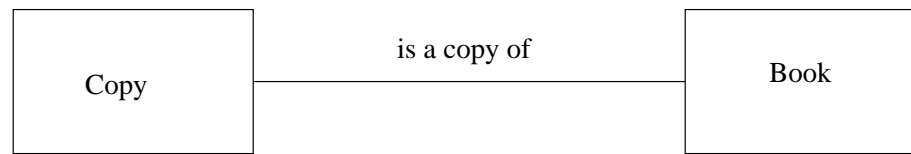
- Objects of a class collaborate with objects of other classes. **Example:** a house
Classes: walls, doors, windows, lights, ...
Structural relationships: Walls adjoin other walls. Doors are built into walls, etc.
Other relationships: Different kinds of windows. E.g., some can not be opened, others having single glass, others double, etc.
- UML supports modeling different relationships between objects or classes

<i>Relationship</i>	<i>Function</i>	<i>Notation</i>
Association	Describes connection between instances of classes	————
Generalization	A relationship between a more general description and a more specific variety	————>
Dependency	A relationship between two model elements	- - - - >
Realization	Relationship between a specification and its implementation	- - - - >
Usage	Situation where one element requires another for proper functioning	- - - - >

Associations describe relationships between objects in a class. The others describe relationships between classes (or interfaces), instead of their instances.

Associations

- Relates objects and other instances of a system
- Semantics: a relation like E/R models

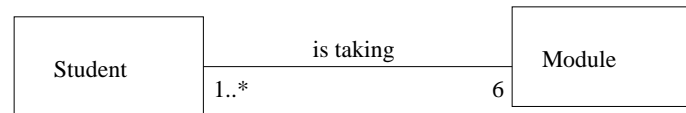


- Examples:
 - Fred **borrows** copy 17 of Book *XYZ*
 - An object of class *A* **sends a message** to an object of class *B*
 - An object of class *A* **creates** an object of class *B*

Associations can be annotated

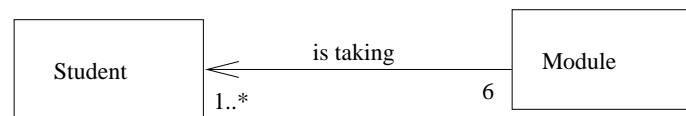
Name: e.g. *is a copy of*

Multiplicity: a number n , or a range $n..m$, or an arbitrary number $*$



- Each student takes 6 courses and each course has at least one student.
- **Semantics:** constrains the relation

Navigation: a line can be directed with an arrow

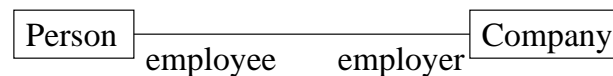


- Module objects can send students messages, but not vice versa.
- **Semantics:** relation can be queried in only one direction

One can omit details in the early modeling phases. The result is then ambiguous.

Annotation (cont.)

Role: describes the roles played by objects in the association



- Makes relationships easier to read and understand
- No semantic consequences

Directed names: describes the direction that a name should be read

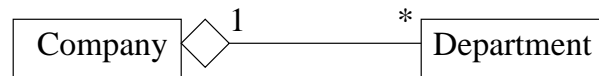


- Independent of navigation (direction)
- Also no semantic consequences

Associations: aggregation and composition

Notational support for certain kinds of frequently occurring associations

Aggregation: whole-parts relation



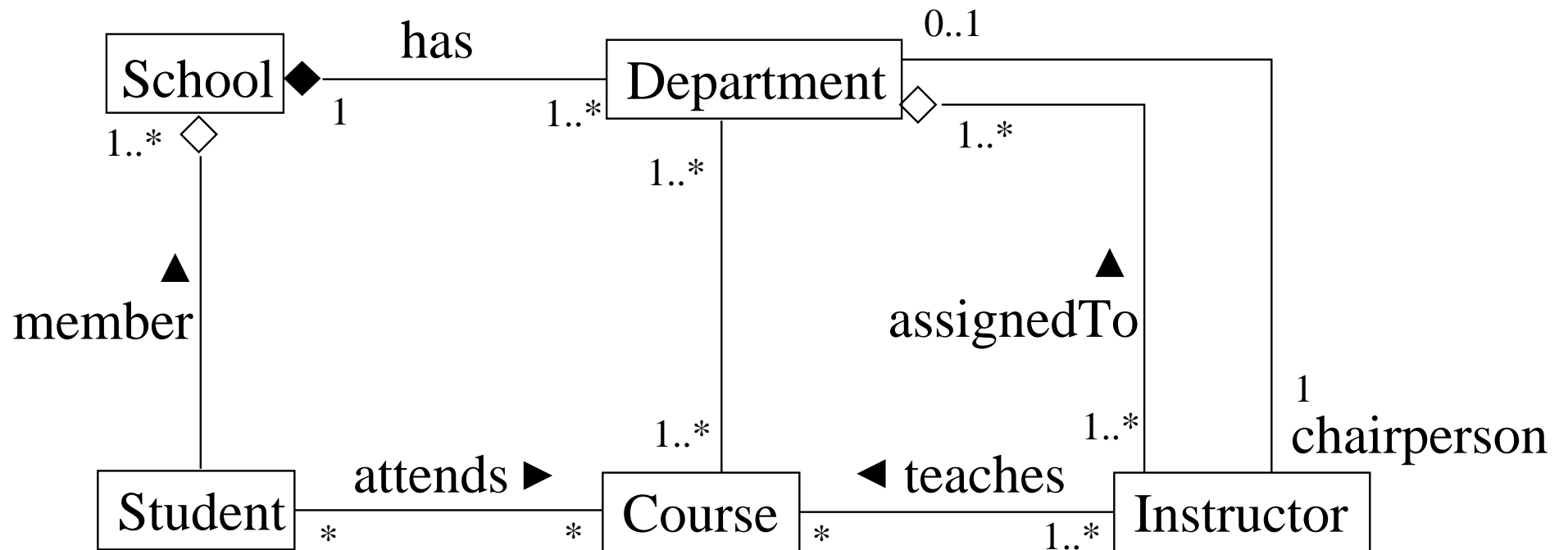
- Diamond marks the “whole”
- Typically no name. Implicitly “is a part of”

Composition: aggregation where the “part” has no independent existence



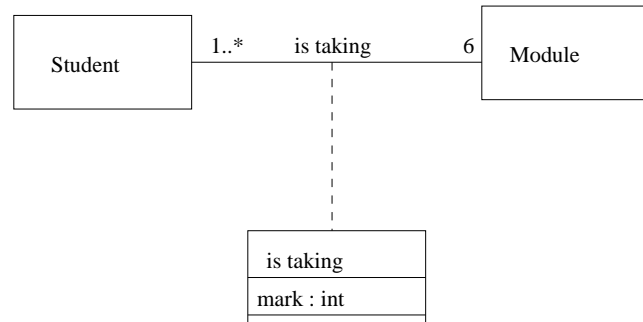
When the “whole” is deleted, so are the parts

An example

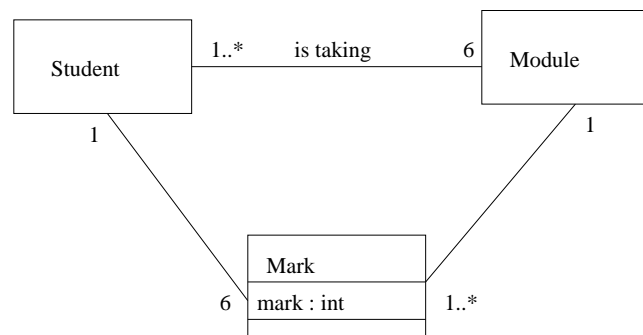


Association classes

- An association can itself have properties
- An **association class** models things with both association and class properties



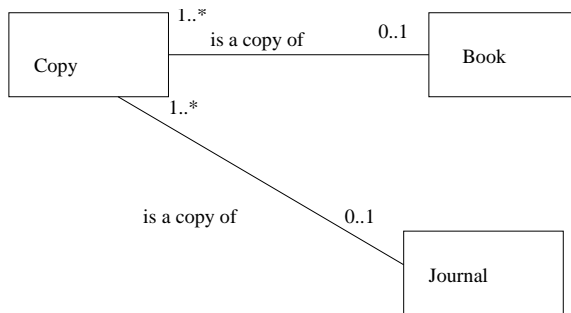
- An association class can be replaced with multiple associations



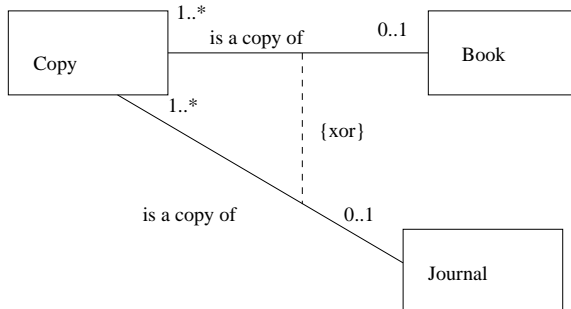
Question: What is the semantics of these diagrams? Are the two equivalent?

Object constraints

- Associations correspond to relations, i.e., sets of tuples.



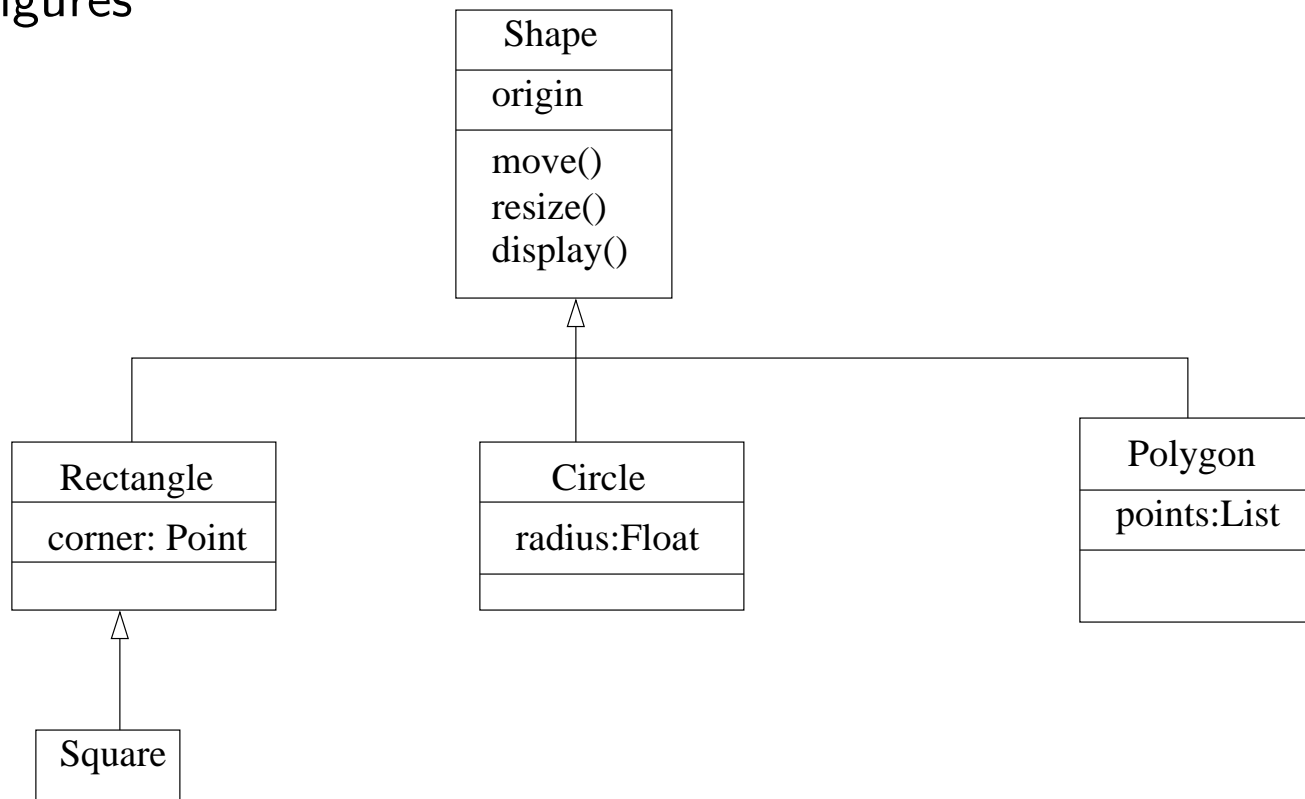
- One can introduce constraints to tighten the semantics (i.e., remove tuples)



- UML offers an extension, **OCL**, where constraints are given by logical formula
We will later consider an alternative, the formal language **Z**

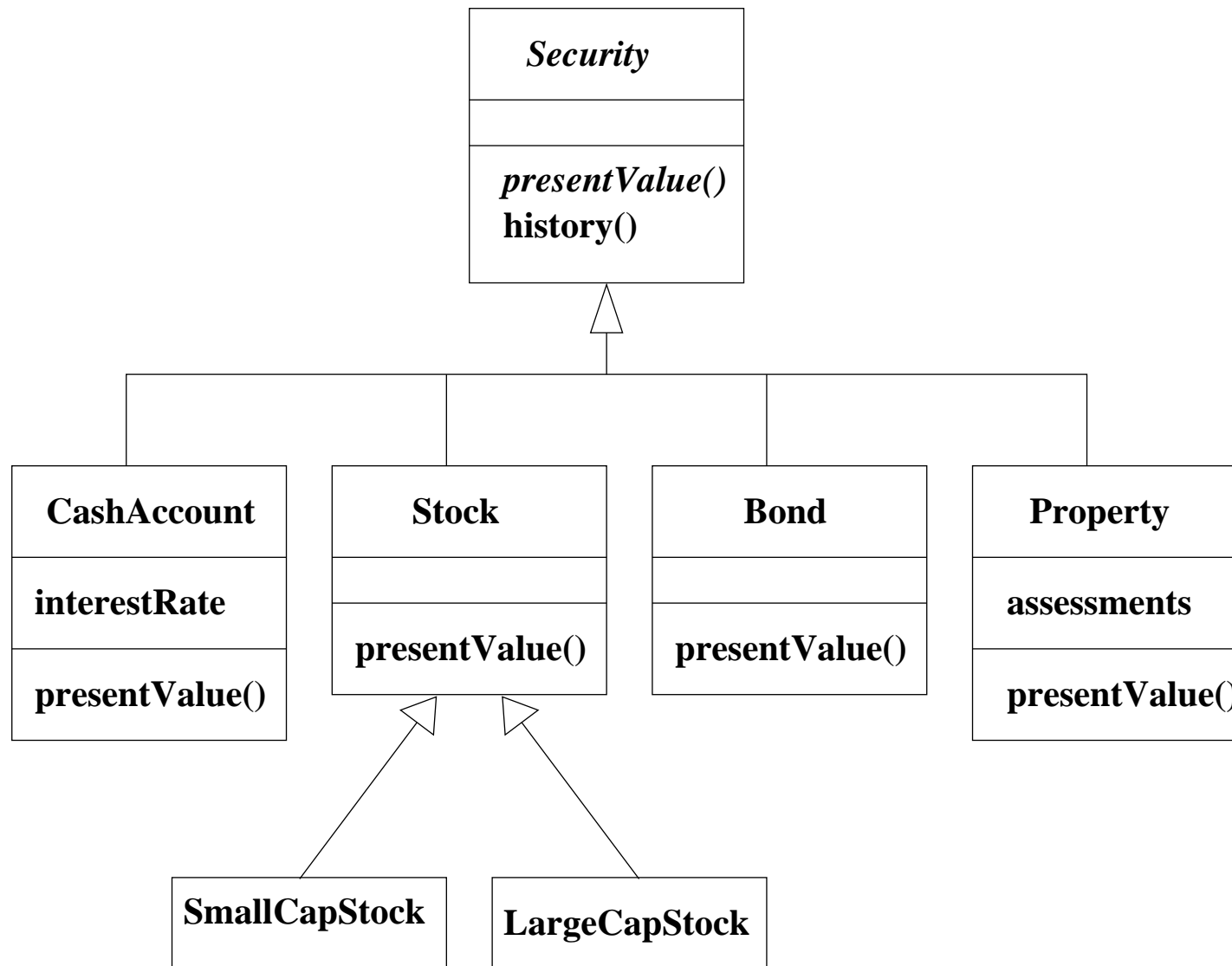
Generalization

- Relation between a general thing (**superclass**) and a specific thing (**subclass**). Sometimes called **is-a**.
- Example: figures



- Semantically: an object of a subclass can be substituted for a superclass object

Generalization — abstract classes



Conclusion

- Class diagrams support the modeling static, structural relationships
- Central concept of OO-modeling and design!
- UML offers even more (parameterization, visibility, qualifiers, etc.)
 \implies topics for a specialized course
- Case-studies in more detail later

Dynamic Modeling

David Basin

Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Spring 2002

Overview

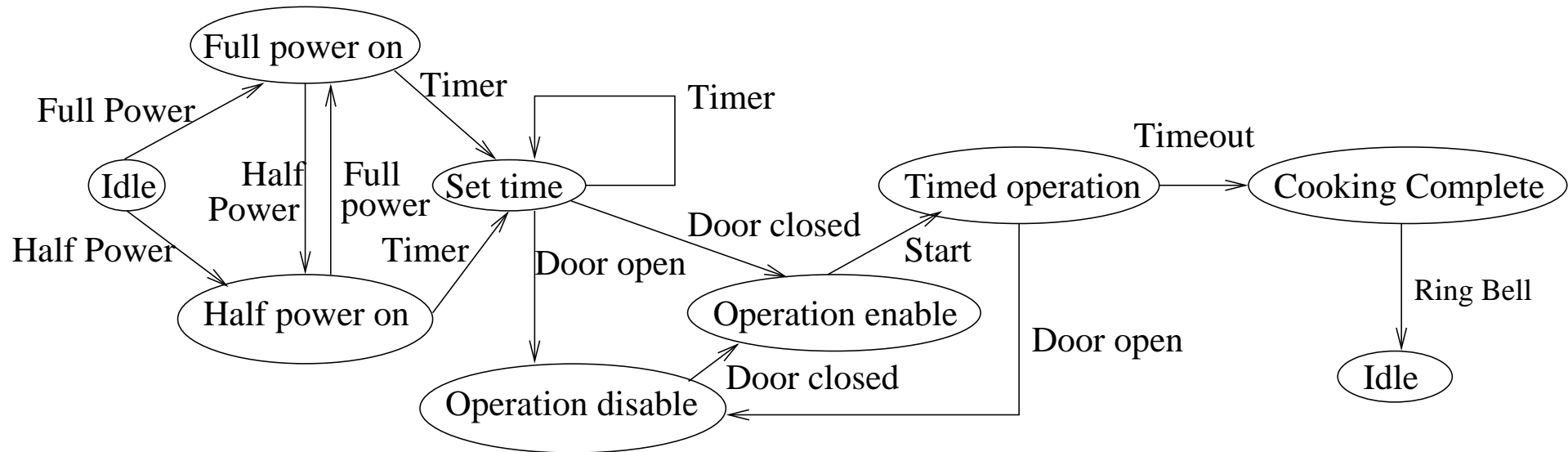
- Dynamic modeling: **within** and **between** **objects**
- Introduction to different kinds of models
 - Statecharts:** state oriented; both within and between
 - Activity diagrams:** state oriented; emphasizes coordination between objects
 - Sequence diagrams:** Information flow (messages) between objects
- Applications

Dynamic modeling — state oriented

- Model dynamic aspects of systems: **control** and **synchronization**
 - What are the **states** of the system?
 - Which **events** does the system react to?
 - Which **transitions** are possible?
 - When are **activities** (functions) started and stopped?
- **Example:** When the left mouse button is pressed, an option menu appears
- Such models correspond to **transition systems** $\langle \Sigma, Q, \delta \rangle$
Also called **state machines** or (an extended variant of) **automata**

State machine modeling

- Machines specify system states and control flow



- State \rightsquigarrow system function transition \rightsquigarrow event

State machine modeling (cont.)

States and transitions must be explained (semantics):

States:

State	Description
Half Power On	Power is 300 Watt
Full Power On	Power is 600 Watt
Set Time	Timer turned on
⋮	⋮

Transitions/Events:

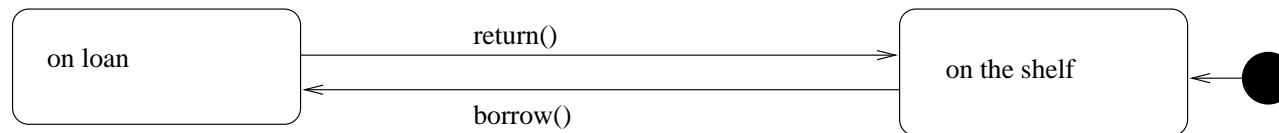
Event	Description
Half Power	'Half Power' button pressed
Full Power	'Full Power' button pressed
Timer	'Timer'-knob turned
⋮	⋮

Statecharts

- Ordinary state machines have several weaknesses:
 - No hierarchy:** flat state space
 - Hierarchy is conceptually useful
 - Supports development using iterated refinement
 - No parallelism:** Machines are combined via product construction
 - Conceptually inadequate
 - Not possible in practice without tool support
- **Example:** Electronic watch
- Statechart extension (Harel 1987) solves this problem and supports models where **time** and **reactivity** can be modeled

A simple example

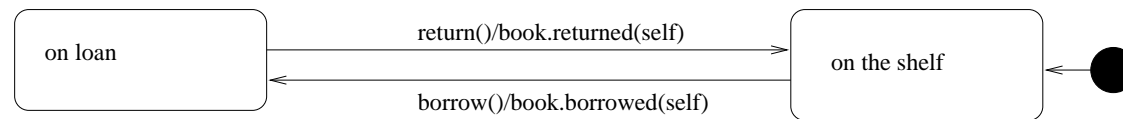
- Statecharts explain **how** and **when** an object reacts
- Example: Borrowing a book \implies statechart for **Copy**



- Syntactic entities: **states**, **transitions**, **events**, **stop symbol** and **start symbol**
- State depends on the attribute variable **on the shelf**
- Interface supplies **borrow** and **return** methods
- Partial specification: model specifies what happens in a particular state. Extension with **error/exception** states is possible
- Note abstraction: the **copy** object can have other attributes (e.g., ISBN number, etc.). These play no role here.

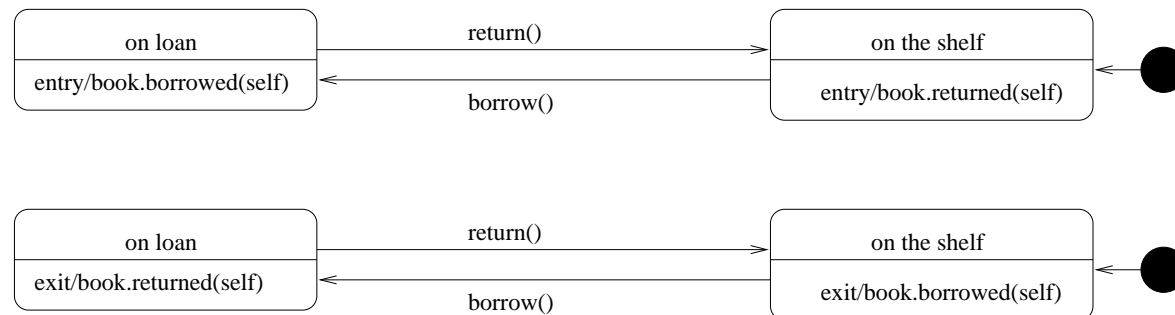
Actions

- Transitions can specify actions



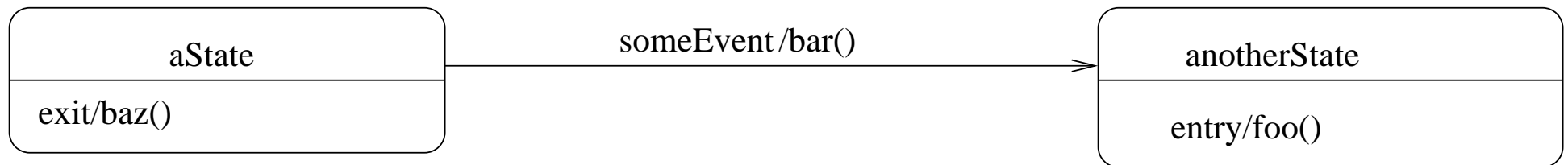
- Syntax: E/A where E is an event and A an action
- An action is what an object does, e.g., send a message
- Here the **Copy** object sends a message to the **Book** object

- States can also specify actions:



- Entry/Exit: action possible (also in combination with transition actions)
- Both examples have the same meaning

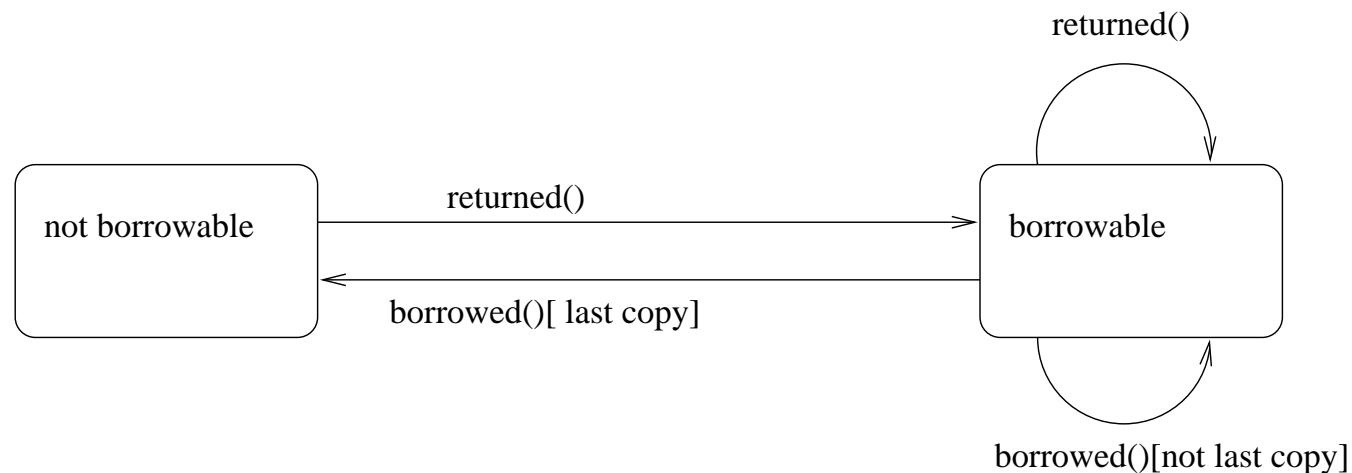
Semantics can be important



Question: In which sequence do the actions occur?

Actions with constraints

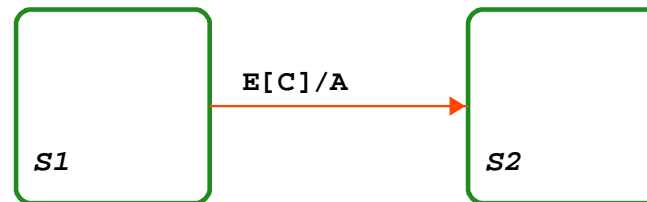
- **Guards** are used to restrict transitions



- A guard is a Boolean expression
- UML is open (= vague) concerning precise syntax
 - Can be (precise, natural) language, OCL, or a programming language

What does a statechart mean?

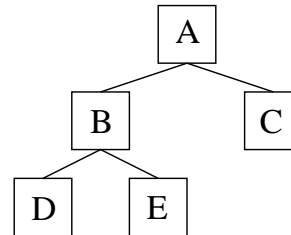
- Semantics as a transition system



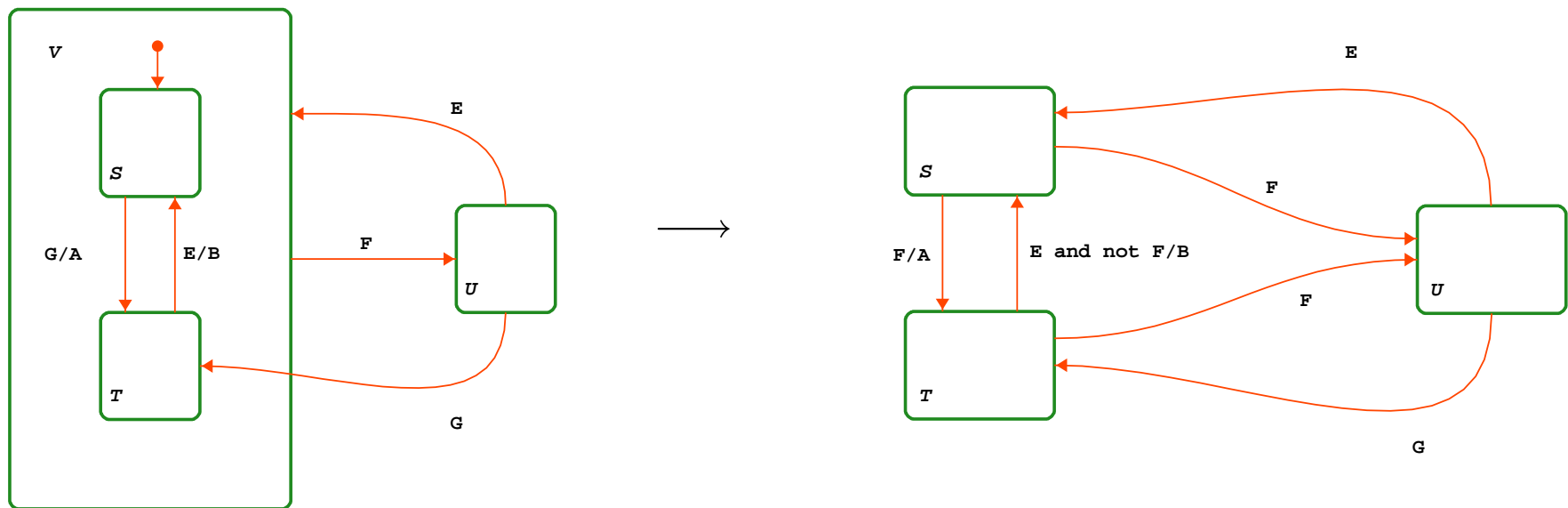
- Transition occurs when
 - System is in $S1$
 - Event E occurred in the last step
 - Condition C holds
- Events immediately follow
- A formal semantics (with time) is actually rather subtle!

Hierarchy

- States are tree structured:

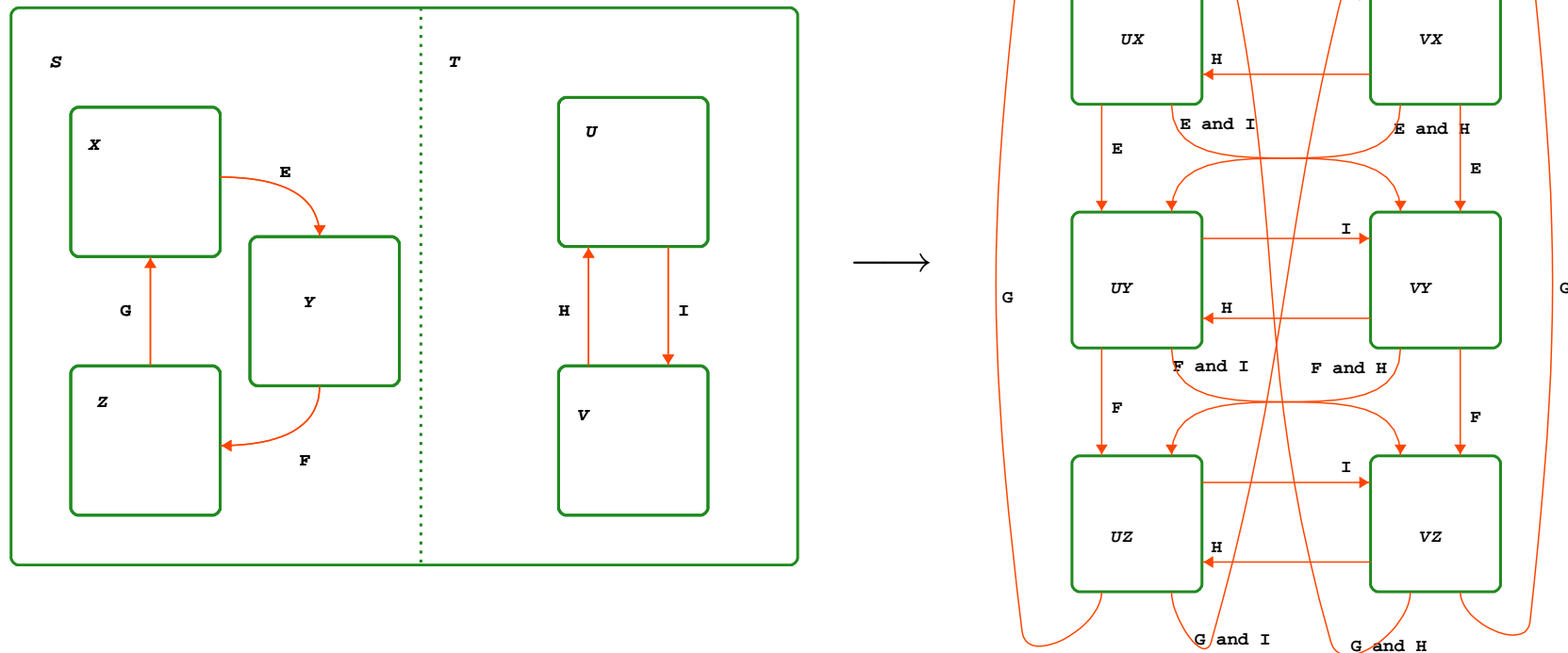


- System is in a **ground state** and in all parent states
- “Higher-level”-transitions have priority over “lower-level”-transitions

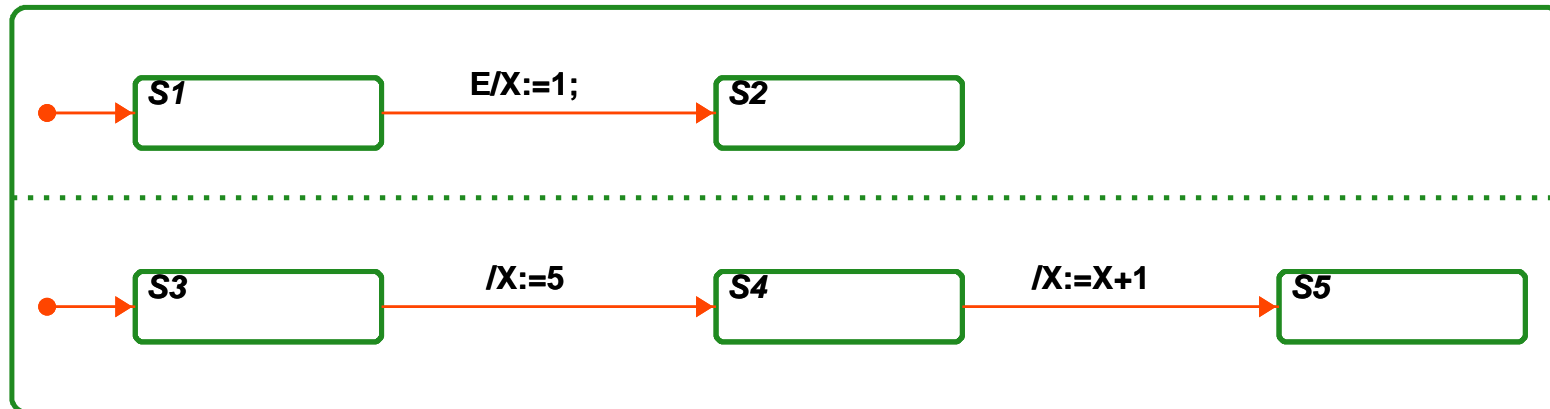
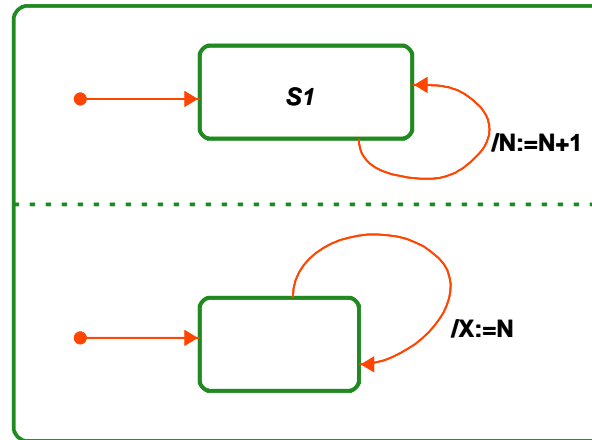


Parallelism

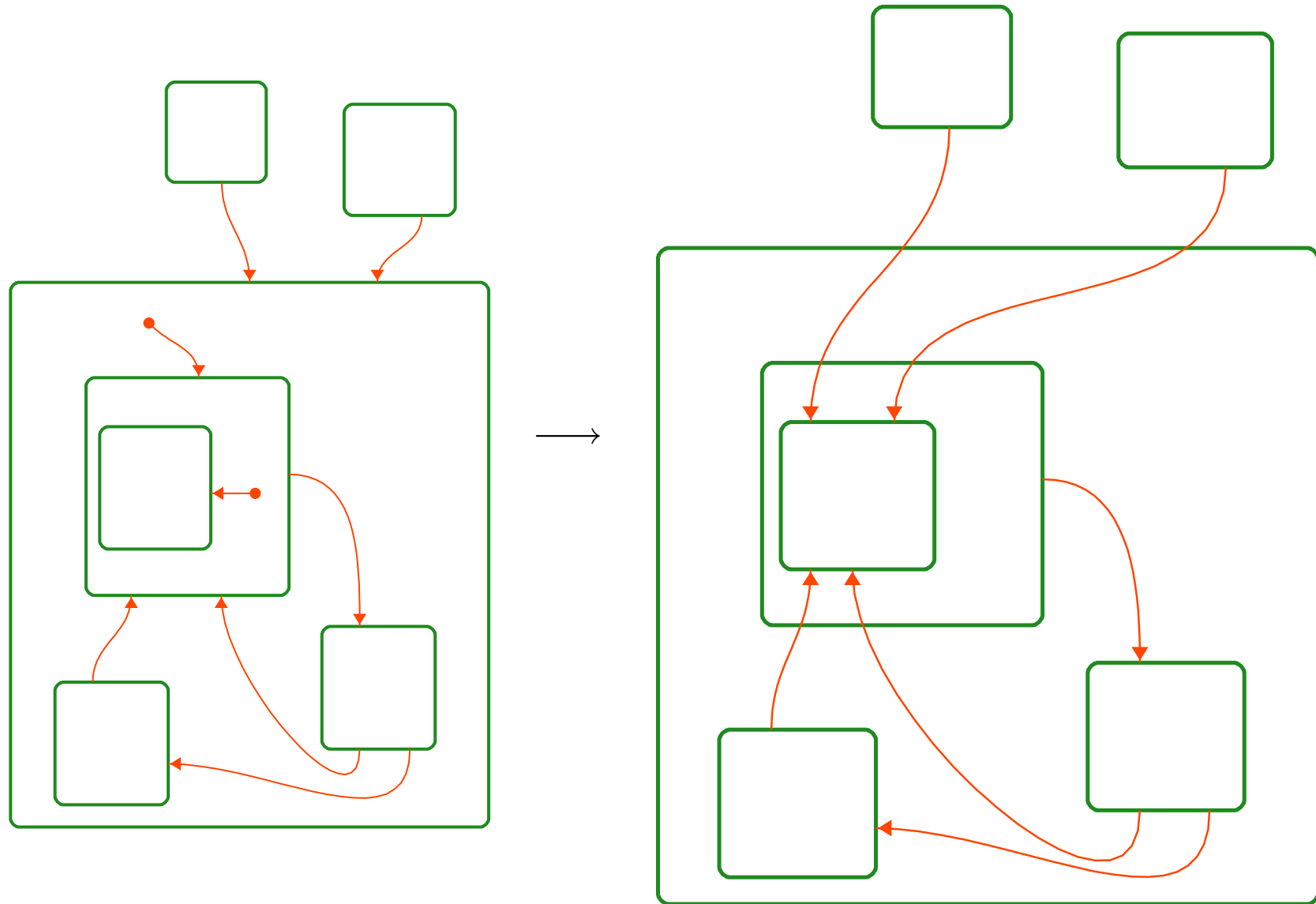
- **OR**-states: System in at most one state of current level
- **AND**-states: Simultaneously in all states of current level \implies **parallelism**



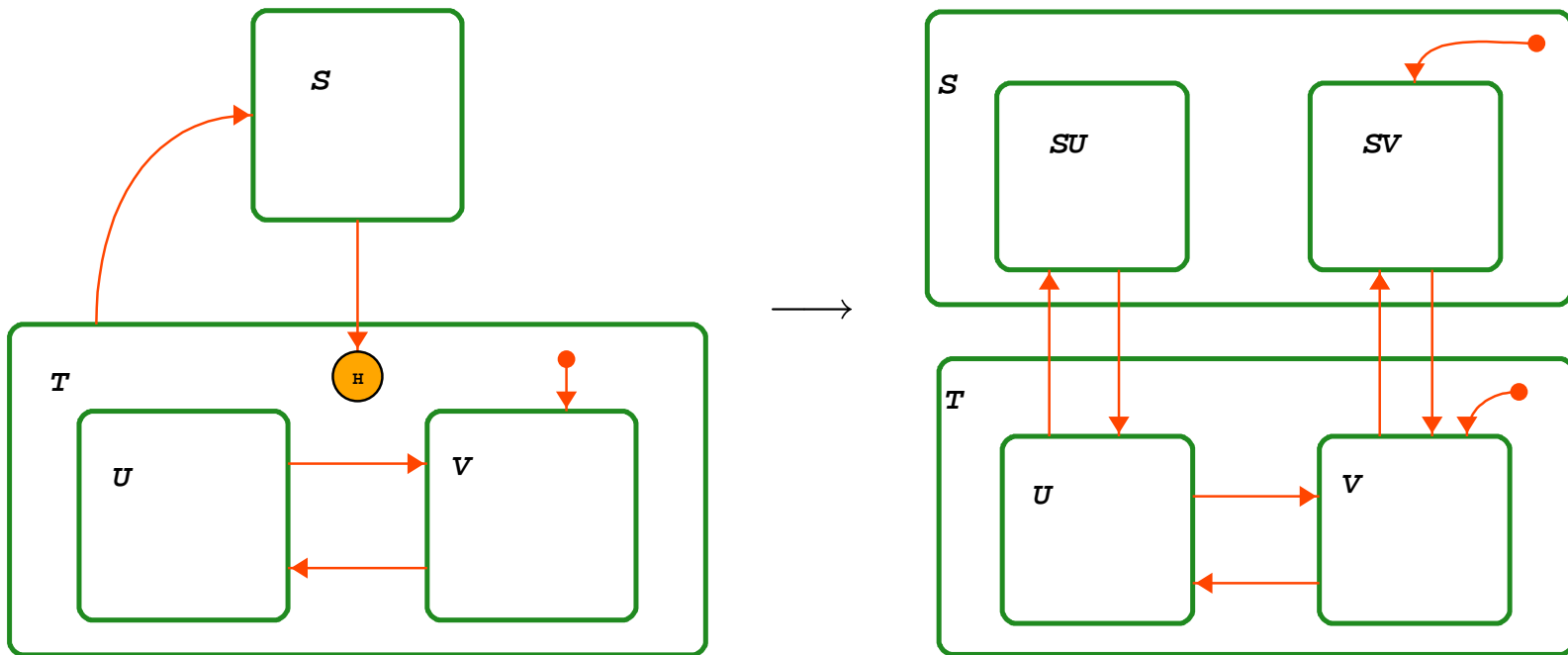
Parallism complicates semantics



Default connector

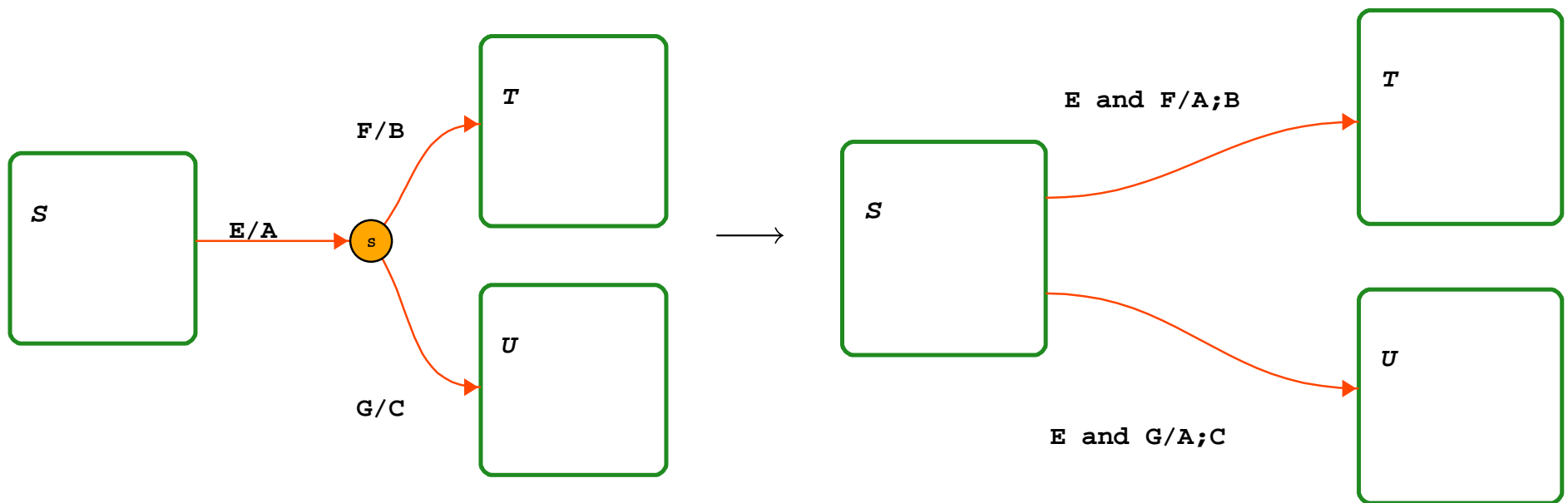


History connector

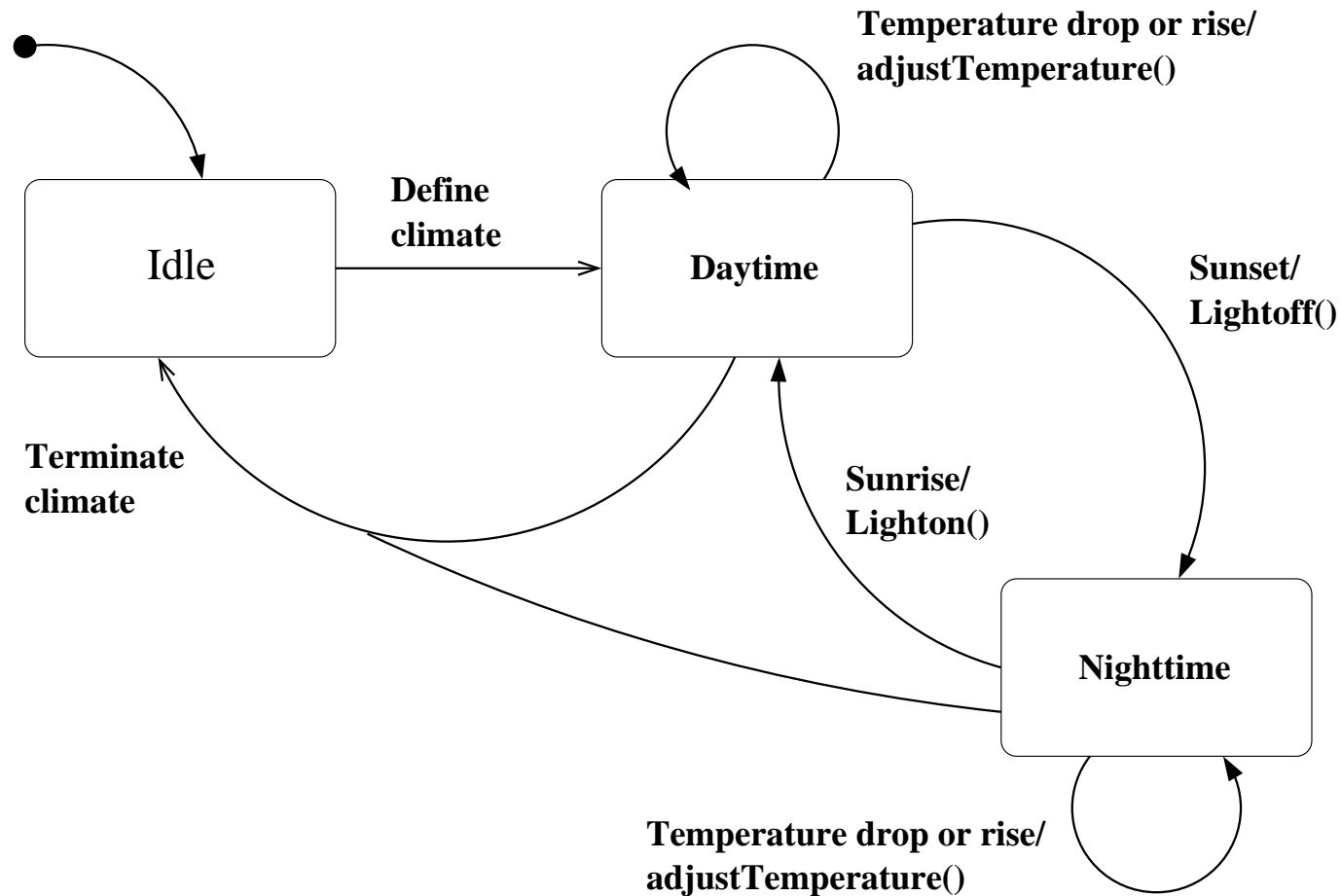


Describes which state was last active

Switch connector

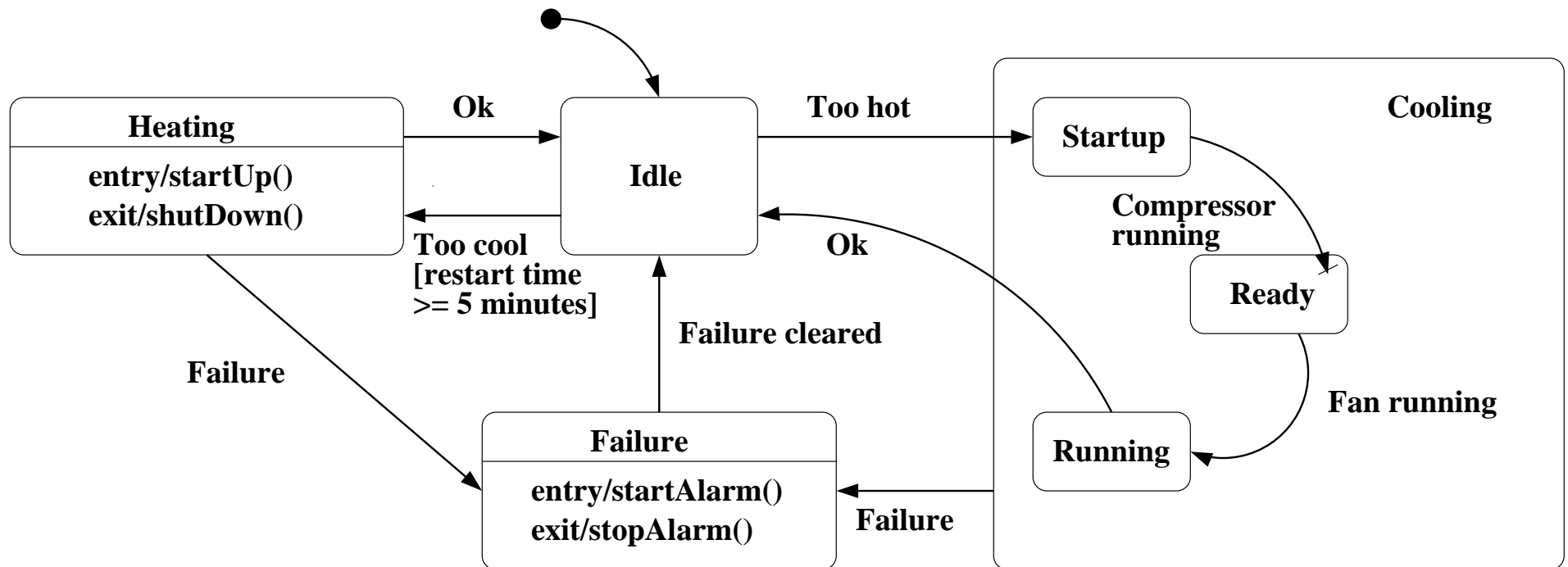


Example #1: simple environment controller



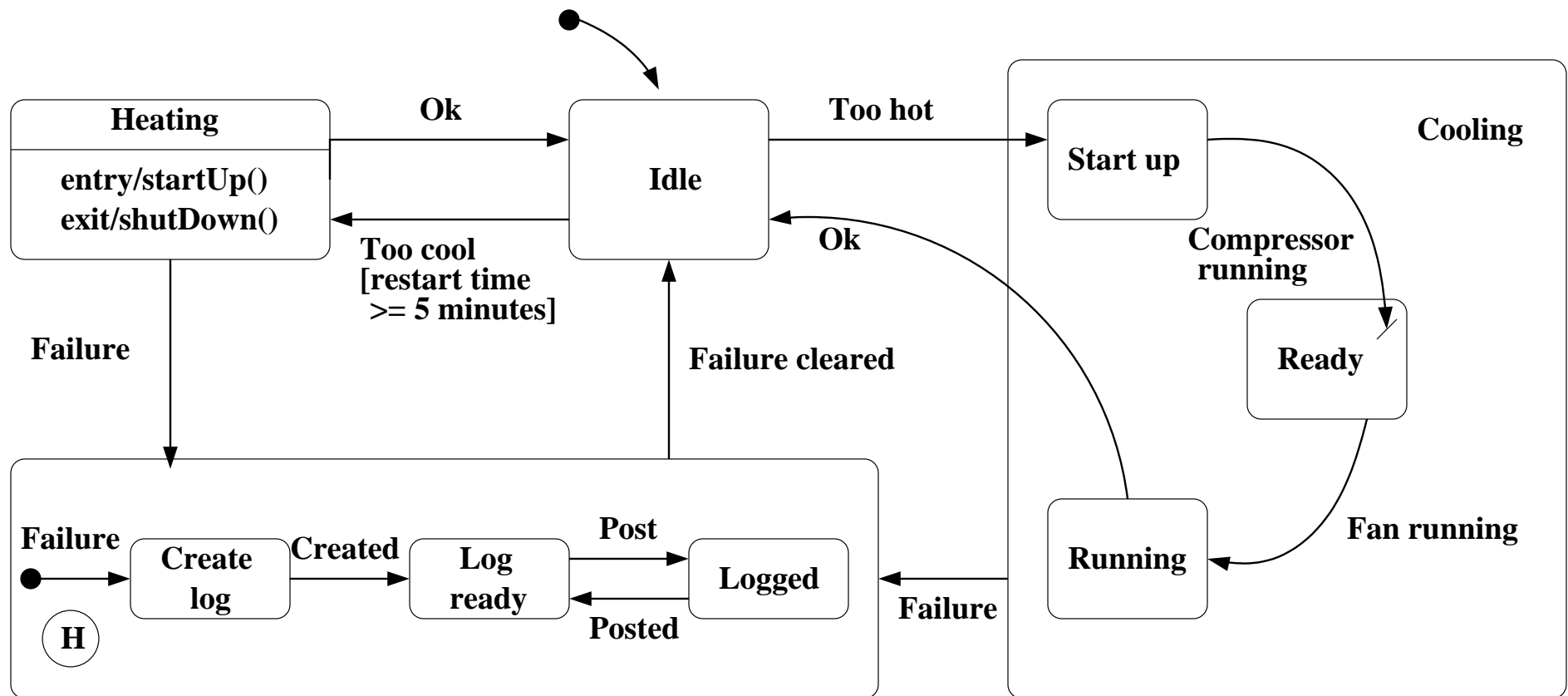
- Controls light and temperature
- Natural language specification of events

Example #2: climate controller



Demonstrates nested states and logical conditions

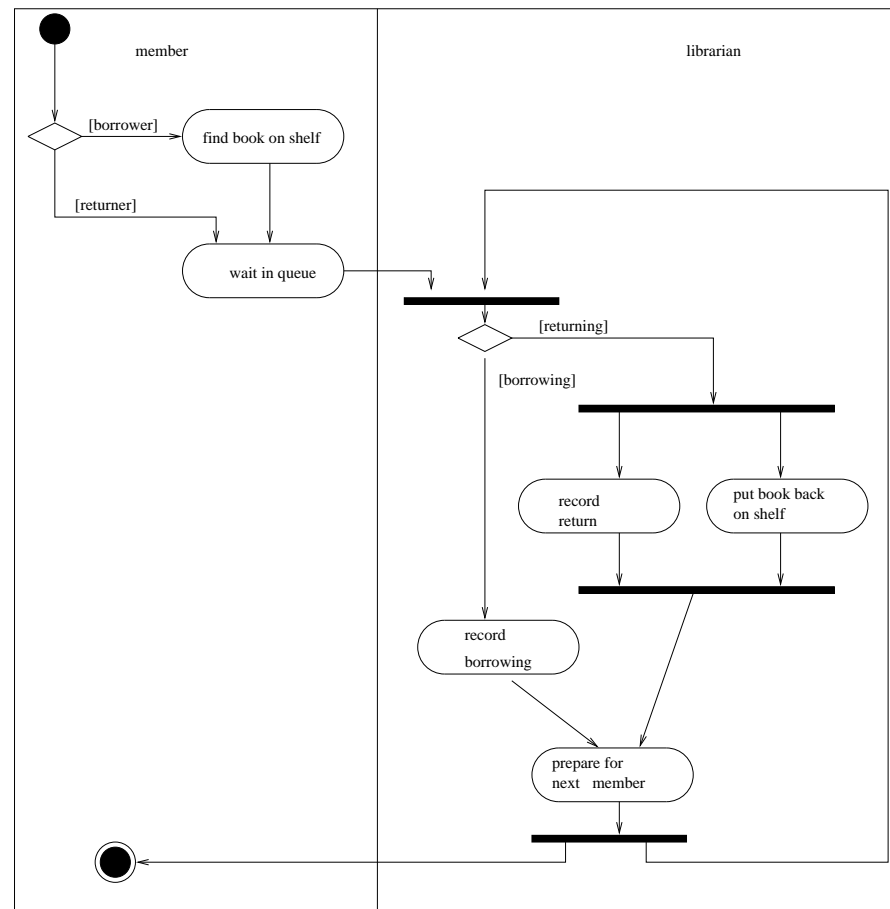
Example #3: climate controller with error handling



- **History connector** used to create log only once
- Semantics subtle: interaction between **Failure cleared** und **Post**

Activity Diagrams

- A (simple) alternative to statecharts
 - Emphasizes synchronization **within** and **between** objects
- An example: work in a library



Syntax/Semantics

- Syntax:

Activity: a kind of state, left when the activity is finished

Transition: normally not labeled, since the event is the end of an activity

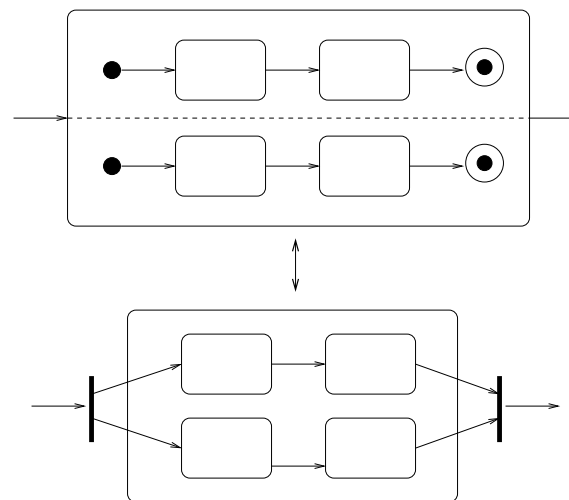
Synchronization bars: describes synchronization points

Decision diamonds: shows decisions, alternative to guards

Start und end markers: like in statecharts

Swimlanes: shows which object carries out which activity

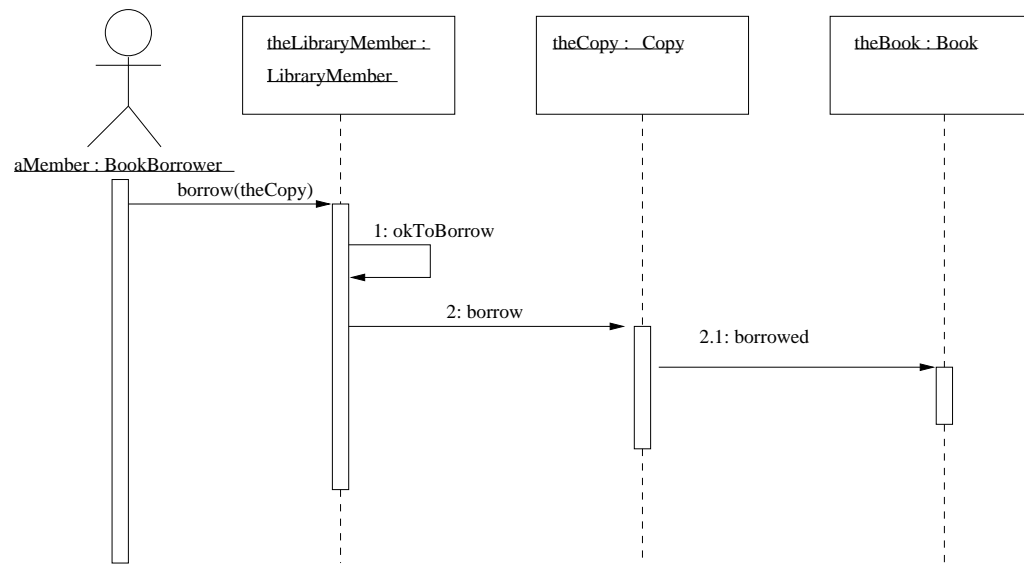
- Semantics: More-or-less intuitive. Can be translated into statecharts



Sequence diagrams

- UML offers two (slightly different) kinds of diagrams to model interaction
 - Sequence diagrams and interaction diagrams
 - We only consider the first here
- Shows how objects interact in execution scenarios by exchanging messages
 - Works well with CRC-card modeling: shows how responsibilities fulfilled through collaboration
 - Describes also temporal aspects of object behavior and (optionally) object life-time in a scenario

Example: book loan scenario



- Syntax

Name of objects (instead of classes!)

Lifeline of the object with activation rectangle

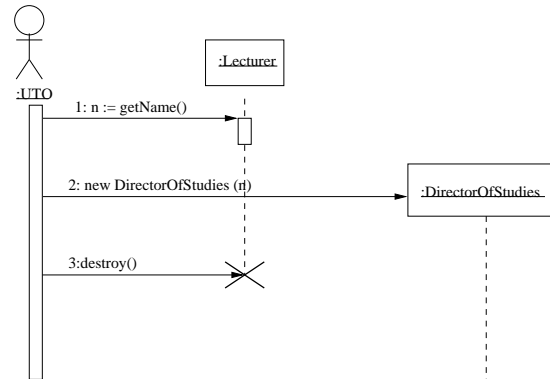
Arrows that indicate message passing; further annotation possible

- Restrictions

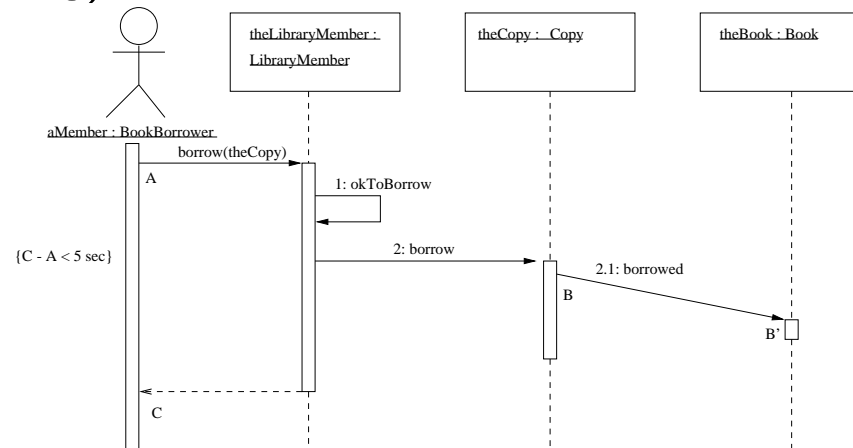
- Branching points possible, but difficult; one scenario at a time is standard
- No support for hierarchy
- No **liveness**; one only specifies what **can** happen, not what **must** happen

Sequence diagrams — extensions

- Creation and deletion of objects



- Constraints (here timing)



- Other extensions supported like annotation for multi-threaded applications (e.g., thread numbering, special arrow for asynchronous message passing, etc.)

Conclusion

- Systems have both static and dynamic aspects
- Dynamics can be viewed/modeled in different ways E.g., focusing on state state versus communication
- Diagrams can be combined to give “fuller” system model
⇒ we will see an example shortly!

Case Study in Semi-formal Modeling

David Basin

Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Spring 2002

Overview

- We will study two examples of using UML diagrams to model systems
 - Class administration program
 - A simulation program
- Shows integration of diagrams in the modeling process
- **Disclaimer:** 30 minute case studies are (over)simplifications.

Example 1: Class Administration Program

- **Project:** System to administer the 4th year of a computer science class.
- The **Status Quo** is as follows:
 - In summer, a committee decides which course modules are offered in winter semester (CS4).
 - The dean makes the teaching assignments.
 - Each teacher creates a course description and sends it to the CS4-coordinator.
 - Someone in the “Undergraduate Teaching Office” (UTO) produces a version of the course handbook. The CS4-coordinator produces a HTML version with `latex2html`.
 - The CS3-coordinator sends a student list to the CS4-coordinator and UTO.
 - Each student has a supervisor that functions as “Director of Studies” (DoS). This assignment is made in the first year.
 - Students register for modules, the UTO checks validity, and the DoS is consulted in borderline cases.
 - The UTO produces a lecture-participation list for each lecturer.
 - Changes, administration of grades, ...

Problem 1 (cont.)

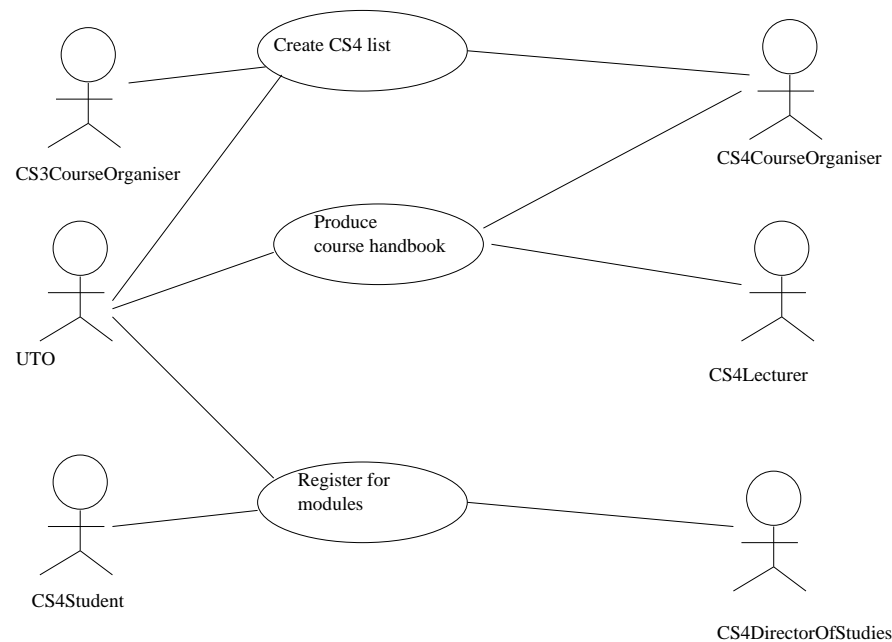
These requirements are imprecise. Further analysis is needed to understand them.

- Are students only CS-students?
- What must additionally be updated? (Web pages, ...)
- What is a course handbook? How many of them are there?
- What constitutes a valid module combination? In some Universities, one must take advanced “honors courses”, etc.

Problem 1 (cont.)

- Desired: an improvement of the Status Quo!
 - Less work for everyone, including the coordinator.
 - On-line registration for teaching modules.
 - On-line information.
 - Support or automation of the production of teaching material (course handbooks) and lists.
- Let's consider part of the requirements:
(We leave queries out — standard support from database system)
- Use Cases are
 - Production of course handbooks
 - Administration from CS4-lists
 - Registration for modules

A Possible Use Case Model



Produce Course Handbook: This use case takes place after the syllabus committee has determined the set of available modules and the department head has allocated duties to lecturers.

The CS4 course organizer updates the core (module-independent) sections of each course handbook by getting the current text from the system, modifying it, and returning the modified version to the system.

The lecturer of each module, similarly, updates the module descriptions by getting the text from the system, updating it, and returning it.

The updates can happen in any order. When the updates are complete, the system sends a complete text of the handbook by email to the UTO, who prints it and updates the web pages.

Brainstorming with CRC Cards

- **Example:** CRC cards for “Produce Course Handbook”

Class name: HonoursCourse	
Responsibilities	Collaborators
Keep collection of modules Generate course handbook text	Module

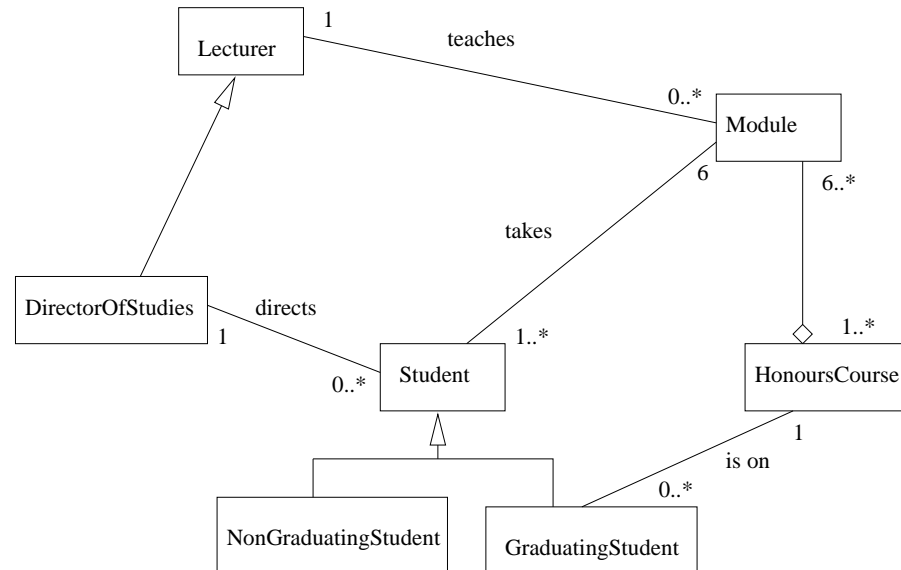
Class name: DirectorOfStudies	
Responsibilities	Collaborators
Provide human DoSs' interface to the system	

Class name: Module	
Responsibilities	Collaborators
Keep description of course Keep Lecturer of course	

- Basis for class diagram and scenario simulation.

Class Diagram

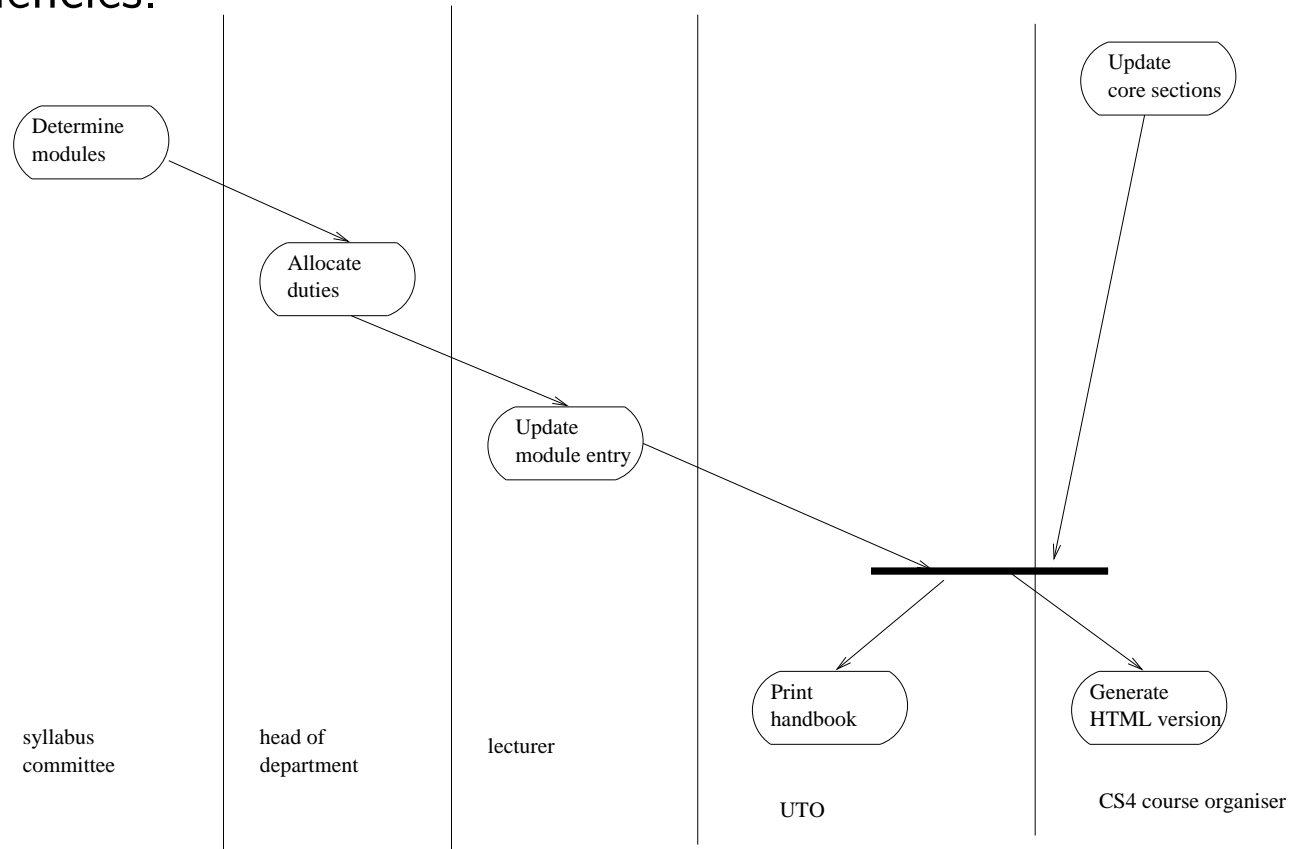
- First version here. Should be refined later.



- **Questions:** Which of the following statements are consistent with this model?
 - Not all lecturers teach some CS4-Module.
 - Not every DoS is a CS4-lecturer.
 - All students must take an honors course.

Dynamic Modeling

- State-charts: No classes with interesting state transitions.
Data modelling plays the largest role in design. Behavior is relatively simple.
- Activity diagrams can be used to model **Work Flow**, e.g., task synchronization and dependencies.



Example 2: A Simulations Program

- **Project:** Develop an **integrated modeling environment** to support discrete event simulation.

Support the **development** of simulation models (e.g., a specialized CASE-tool) and their **simulation**.
- Based on a **process oriented view** of simulation.
- Simulation should both report on events (system traces) and provide statistical summary information.

A Detailed Description (potential classes in red)

System users are developers who build **simulation models** and check that they run without error and experimenters who run them and collection **statics** about what happens.

Entities modeled are **active entities** and **passive entities**. Active entities represent things that carry out activities, like workers in a factory. When a real-life worker does something that affects other real-life things, the active entity modeling that worker causes a **simulated event** which affects those entities that model those things.

Passive entities are **resources**, **semaphores**, and **buffers**. Although inactive, they affect the behavior of active entities, e.g., a process with insufficient resources blocks. Passive entities also report information on their state over time.

The behavior of an active entity is determined by a sequence of events to be simulated. At any point in the simulation, an active entity is in one of three states.

1. Active, where it responds to an event. Only one active entity can be in this state at a time and its **event time** defines the current **simulated time**.
2. Blocked, waiting for a request to be satisfied by a passive entity.
3. Waiting for simulated time to reach this object's next event time.

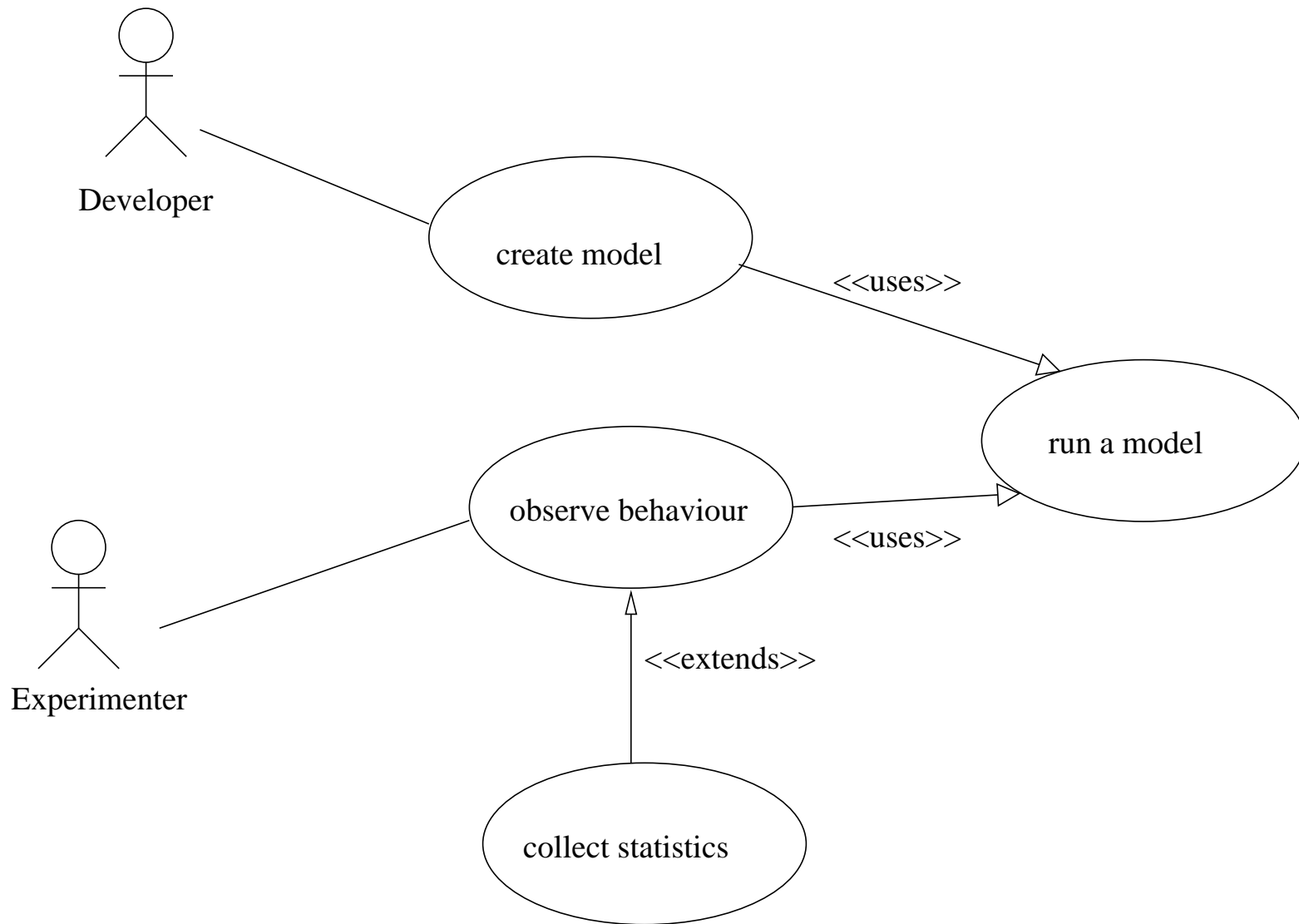
Description (cont.)

Simulated events arise as messages either from a **scheduler** or from a passive entity whose state has changed. Simulated events should send messages to a trace file so that an experimenter can follow the internal behavior of the model.

Statistics are collected by updating information about passive entities and other values for which information is needed. Examples of values being monitored and their derived statics are **counts** of how many times something occurs and the **average value over time** of something like a queue's length.

The conditions under which a model executes are varied to observe how the system would respond. The values varied are read from an **external data set**, which is set up before the model is run.

Use Case Model



Use Cases (cont.)

create model: **Developer** creates an initial model by interacting with editing functions of the tool. He then creates a dataset and checks the model by following the behavior defined in the **run a model** use case. If there are errors, he modifies the model and runs it again.

observe behavior: **Experimenter** selects a model, created earlier (see **create model**). He selects or creates a dataset which determines conditions such as resources and durations of variable delays. He then follows the behavior in the **run a model** use case and, when the model has run, reads the trace generated.

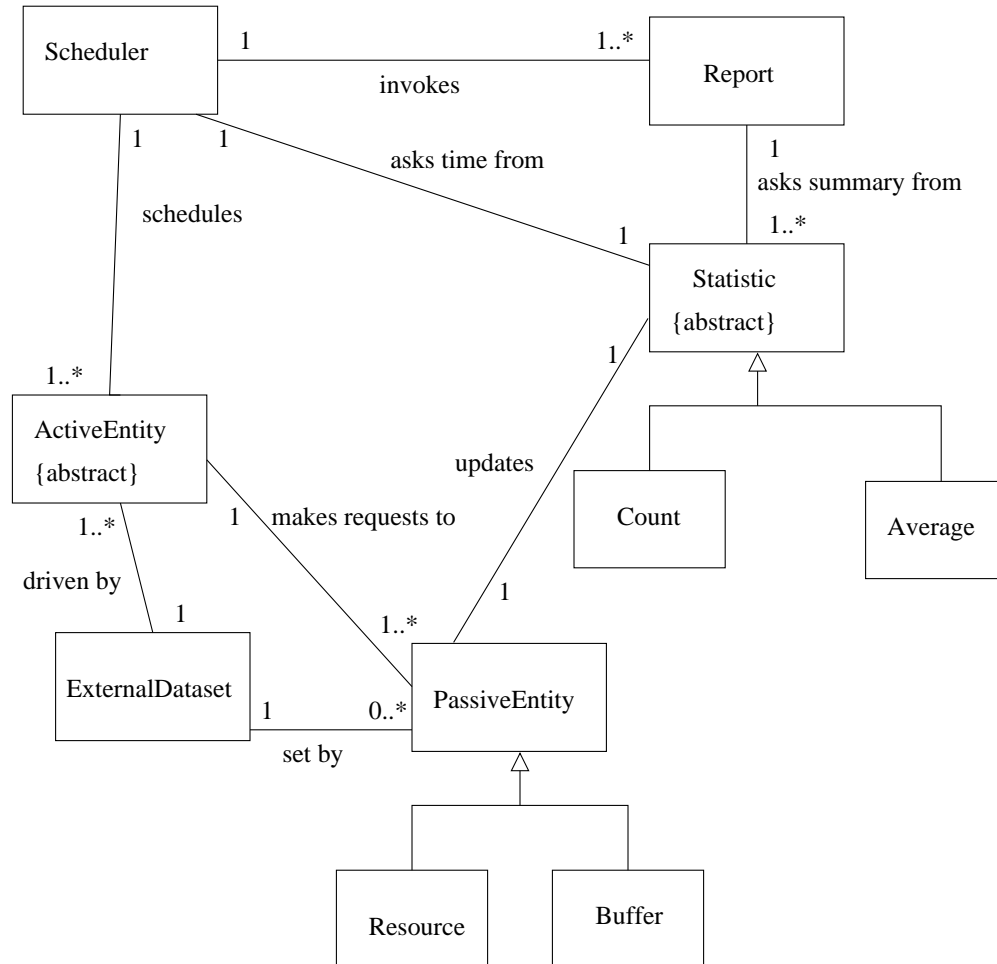
collect statistics: Before using the **run a model** use case, the **Experimenter** sets a flag indicating that statistics should be collected and reported by writing them to a file at the end of the run.

run a model: This use case assumes a model/dataset has been selected. The actor starts the model, which consists of instances of objects used to model and observe the system. Each active entity sets itself to an initial simulated event time and to an initial next simulated event, which it reads from the dataset. Each passive entity reads its initial settings from the same dataset.

The entities act out behavior specified by the description of each active entity's life cycle, under the control of the scheduler, which ensures that events occur in the order of their simulated time. Passive entities ensure that constraints are respected. ...

Class Diagram

Version I (rough):



N.B.: **ActiveEntity** is abstract since the developer provide a concrete realization (through subclassing). **Statistic** also abstract, although two instances are provided.

Class Diagram (cont.)

- Version I can be refined to come closer to an implementation.

We specify associations, attributes, methods. E.g.,

- Scheduler invokes Report

Direction: One way, from **Scheduler** to **Report**.

Meaning: The **Scheduler** has the possibility of receiving statistics from **Report**-Objects, at the end of a simulation.

Implementation: The Collection Attribute **reports** in the Scheduler, and the operation **report()** in Report

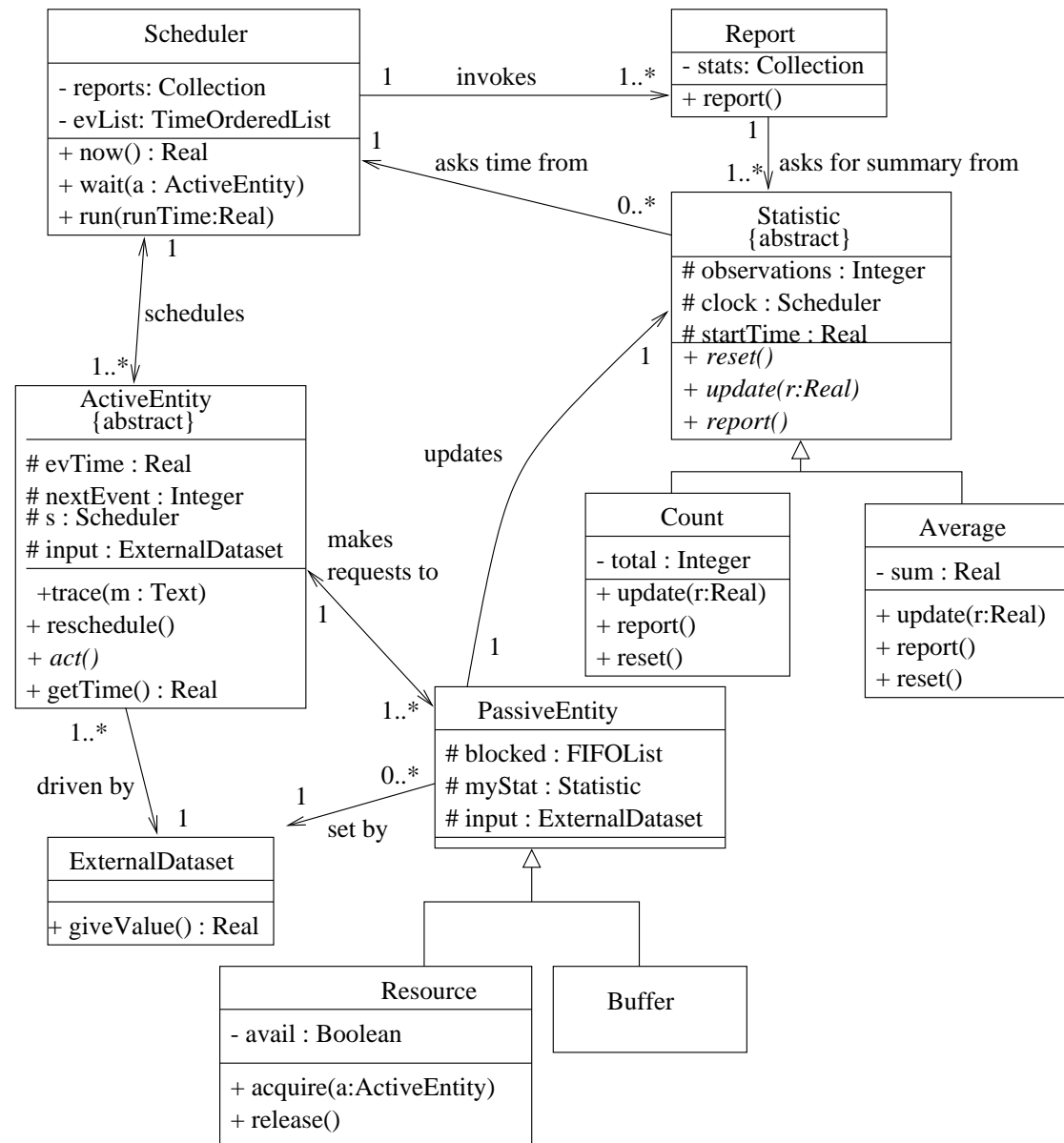
- Scheduler schedules ActiveEntity

Direction: Two way: each **Scheduler** must be able to activate and **ActiveEntity**, and each **ActiveEntity** requires the **Scheduler**, in order to be entered in the event list.

Meaning: The **Scheduler** controls the sequence in which **ActiveEntities** are executed, guaranteeing that execution occurs in the correct order.

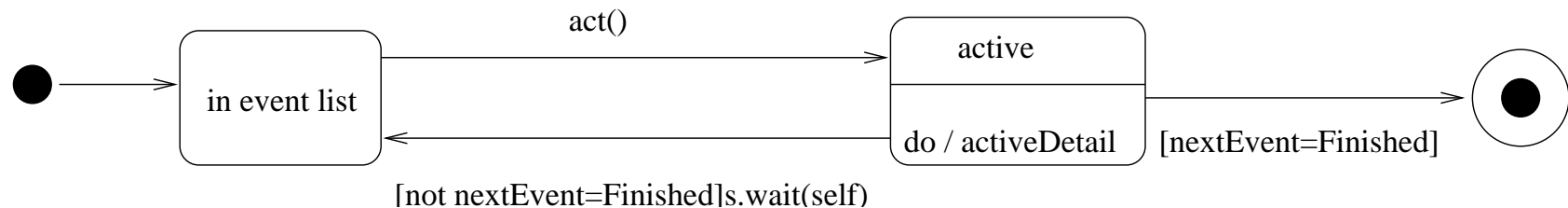
Implementation: ...

Refined Class Diagram



One class in detail — **ActiveEntity**

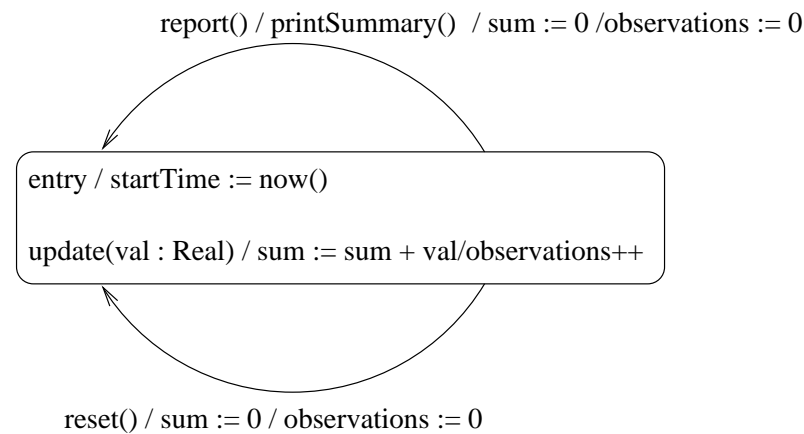
- Abstract Class: No operation for **act()**. Can only be used after generalization.
- Attributes are **protected**. Can be used in subclasses.
- State/behavior:



- One begins in the state **in event list**, i.e., one waits in **evList** of the **Schedulers**. A transition occurs when the object receives an **act()**-message,
- Upon leaving the state **active**, the object either terminates or sends itself to the **Scheduler** in the form of a **wait**-message.
- **ActiveEntity** has a nested state machine named **activeDetail**, which specifies specialized details from **active**. In each specialization of **ActiveEntity**, the **Developer** must later define **activeDetail**.

Another Class — Average

- Specializes the abstract class `Statistic`
- 3 Methods. Meaning can be fixed with a state-chart.



Summary

- Models yield different views of a system at different levels of abstraction.
- Models compliment textual documentation.
- Useful for planning, development, etc. ...
- Iterating brain-storming/diagrams/discussions is helpful in building models and improving general problem understanding.
- Building models is a skill that must be acquired through practice!

We have not shown any false starts here.

Formal Modeling with Z: An Introduction

Luca Viganò

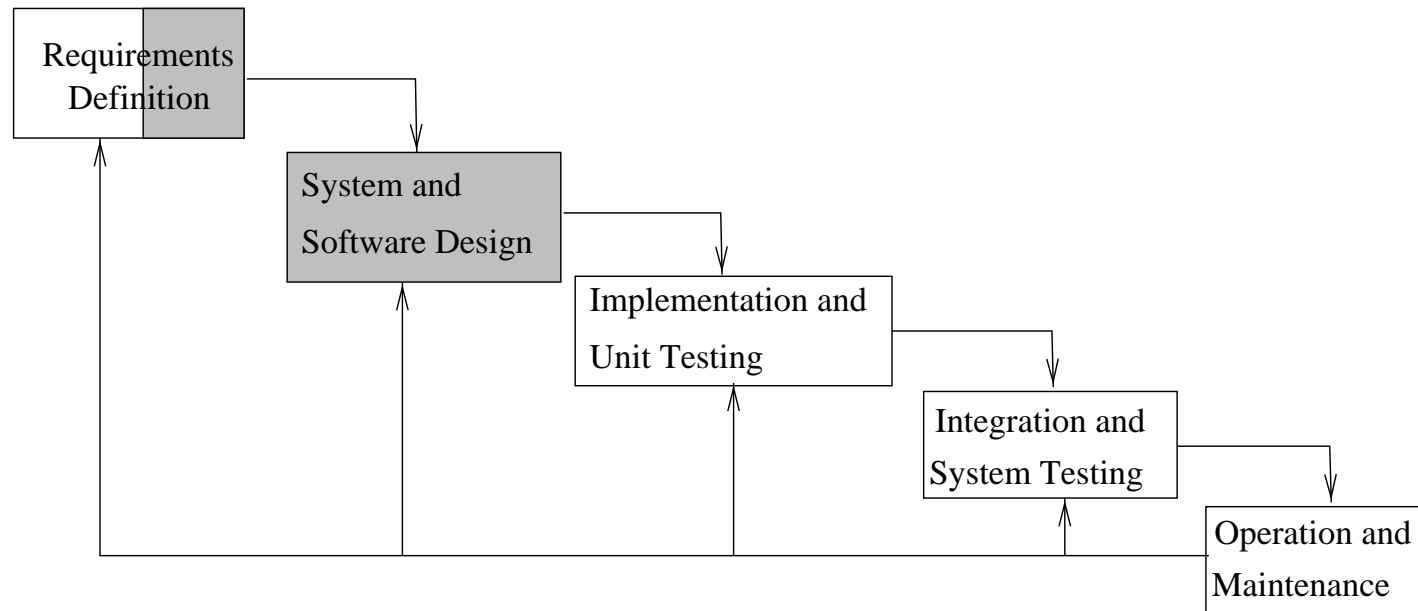
Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Spring 2002

Overview

- Today:
 - Modeling with formal languages: why and how?
 - A standard introductory example.
 - The Z language — first definitions.
- Next classes: Z in detail, the mathematical toolkit, and applications.

Modeling



- **Goal:** specify the requirements as **far** as possible, but **abstract** as possible.
- **Definition:** A **model** is a construction or mathematical object that describes a system or its properties.

Which Modeling Language?

- There are hundreds! Differences include:

System view: static, dynamic, functional, object-oriented,...

Degree of abstraction: e.g. requirements versus system architecture.

Formality: informal, semi-formal, formal.

Religion: OO-school (OOA/OOD, OMT, Fusion, UML), algebraic specification, Oxford Z/CSP-Sect, Church of HOL,...

- Examples:

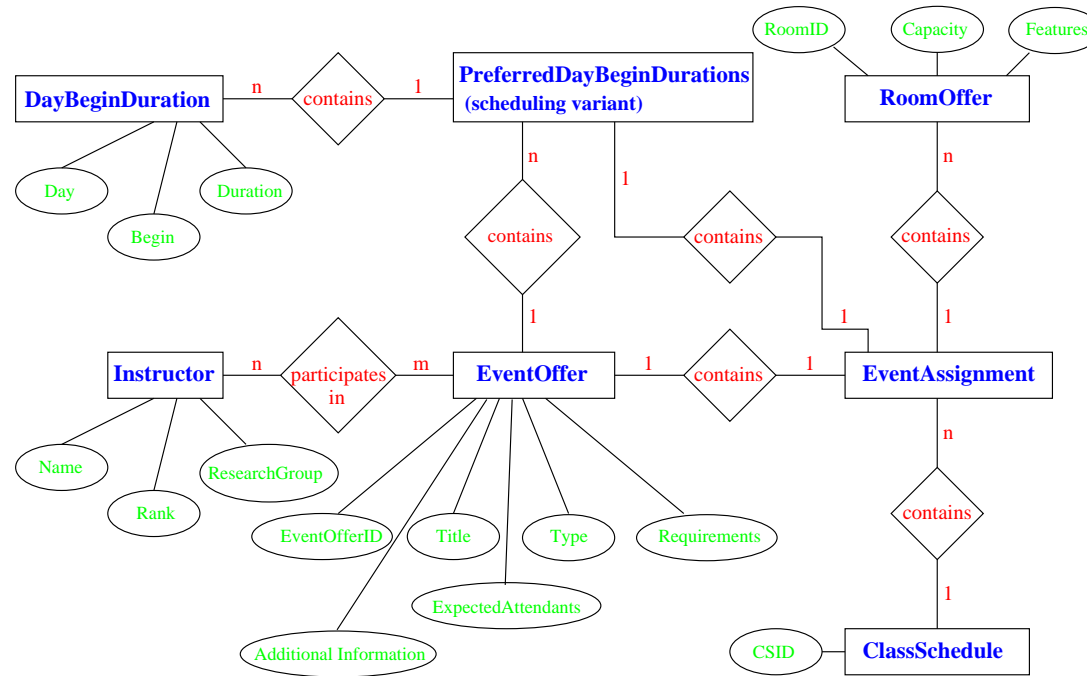
Function trees, data-flow diagrams, E/R diagrams, syntax diagrams, data dictionaries, pseudo-code, rules, decision tables, (variants of) automata, Petri-nets, class diagrams, CRC-cards, message sequence charts,...

- Why are UML or other semi-formal languages not enough?

Disadvantages of Semi-Formal Languages

- Modeling is detailed and intuitive (and "simple", i.e. also for managers and laymen).
- Semantics of models/diagrams is often imprecise.
- Often only syntax.

Example: Problems with E/R-Diagrams



- Are the relations "directed"?
- Several properties cannot be specified graphically (e.g. constraints).
- etc.

We will employ Z to formalize semi-formal diagrams and models.

Formal Languages

- A **language** is **formal** if its **syntax** and **semantics** are defined formally (mathematically).
- Formal languages allow for the design, the development, the verification, the testing and the maintenance of a system:
 - remove ambiguities and introduce precision,
 - structure information at an appropriate abstraction level,
 - support the verification of design properties,
 - are supported by tools and systems.
- Using mathematics (and formal methods) may appear to be expensive, but in the long run it pays off (and how!).

Z (“zed”)

- Is a very expressive formal language.
- Based on **first-order logic with equality** (PL1=) and **typed set-theory**.
- Has a **mathematical toolkit**: a library of mathematical definitions and abstract data-types (sets, lists, bags, ...).
- Supports the **structured** modeling of a system, both **static** and **dynamic**:
 - modeling/specification of data of the system,
 - functional description of the system (state transitions).
- Is supported by several tools and systems.

Z and other Formal Languages/Methods

- A number of successfully employed **Formal Methods** are based on PL1= with type-theory, e.g.
 - VDM (“Vienna Development Method”, 80’s),
 - B (applied extensively in France).
- Other **formal languages**:
 - Equational logic or Horn logic (in algebraic specifications),
 - PL1=,
 - Higher-order logic (HOL).
- Z:
 - Applied successfully since 1989 (Oxford University Computing Laboratory), e.g. British government requires Z-specifications for security-critical systems.
 - Is (will soon be) an ISO standard.

An Example (1)

A mathematical model that describes the intended behavior of a system is a **formal specification**.

Q: Why do we need such a specification if we can simply write a program?
Why not directly implement the program?

A: A program can be quite cryptic, and therefore we need a specification that describes formally the intended behavior at the appropriate abstraction level.

An Example (2)

Example: What does the following simple SML-program do?

```
fun int_root a =  
  (* integer square root *)  
    let val i    = ref(0);  
        val k    = ref(1);  
        val sum  = ref(1);  
    in while (!sum <= a) do  
        (k    := !k+2;  
         sum  := !sum + !k;  
         i    := !i+1);  
    !i  
  end;
```

An Example (3)

- The program is efficient, short and well-structured.
- The program name and the comment suggest that `int_root` simply computes the "integer square root" of the input, but is it really the case?
- Moreover: What happens in special input cases, e.g. when the input is 0 or -3?
- Such questions can be answered by code-review (or reverse-engineering), but this requires time and can be problematic for longer programs.
- The key is **abstraction**: understanding the code must be separated from understanding its "function".

For example, consider a VCR whose only documentation is the blue-print of its electronic.

A Example (4)

- Solution: we can specify the program in Z.

Formalize **what** the system must do without specifying/prescribing **how**.

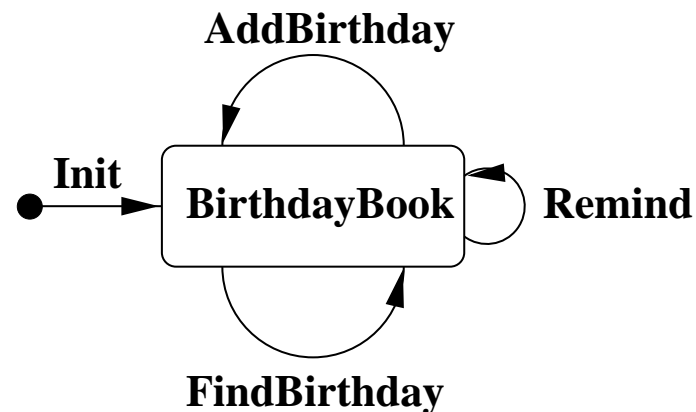
- We specify `int_root` in Z by means of a so-called **axiomatic definition** (or **axiomatic description**):

$int_root : \mathbb{Z} \rightarrow \mathbb{N}$	declaration
$\forall a : \mathbb{N} \bullet \text{let } y = int_root(a) \bullet$	predicate
$y * y \leq a < (y + 1) * (y + 1)$...
$\forall a : \mathbb{N} \setminus \{0\} \bullet int_root(-a) = 0$	predicate

More about Z \rightsquigarrow implementation in a few weeks.

An Introductory Example: The Birthdaybook

- The Birthdaybook is a small database containing peoples' names and their birthdays.
- A simple event-model:



- A **structured Z-specification in 3 steps**:
 1. Define the (auxiliary) **functions** and **types** of the system.
 2. Define the **state-space** of the system.
 3. Define the **operations** of the system (based on the relations of the state-space).

The Birthdaybook: Z-Specification (1)

Step 1. Define the (auxiliary) **functions** and **types** of the system:

- **Basic types**

$[NAME, DATE]$

The precise form of names and dates is not important (e.g. strings, 06/03, 03/06, 6.3, 06.03, March 6, 6.Mar, or ...).

The Birthdaybook: Z-Specification (2)

Step 2. Define the **state-space** of the system using a Z-**schema**:

<i>Birthdaybook</i>	Name of schema
$known : \mathbb{P} NAME$	declaration of typed variables
$birthday : NAME \leftrightarrow DATE$	(represent observations of the state)
$known = \text{dom } birthday$	relationships between values of vars
	(are true in all states of the system and are maintained by every operation on it)

Notation and remarks:

- *known* is the **set** (symbol \mathbb{P}) of names with stored birthdays,
- *birthday* is a **partial function** (symbol \leftrightarrow), which maps some names to the corresponding birthdays,
- The relation between **known** and **birthday** is the *invariant* of the system:
the set *known* corresponds to the domain (dom) of the function *birthday*.

The Birthdaybook: Z-Specification (3)

- Example of a possible state of the system:

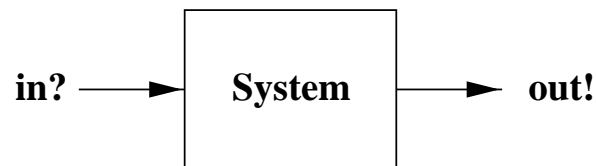
$$\begin{aligned} \textit{known} &= \{\textit{Susy}, \textit{Mike}, \textit{John}\} \\ \textit{birthday} &= \{\textit{John} \mapsto \textit{25.Mar}, \\ &\quad \textit{Susy} \mapsto \textit{20.Dec}, \\ &\quad \textit{Mike} \mapsto \textit{20.Dec}\} \end{aligned}$$

- Invariant $\textit{known} = \text{dom } \textit{birthday}$ is satisfied:
 - $\textit{birthday}$ stores a date for exactly the three names in \textit{known} .
- N.B.:
 - no limit on stored birthdays,
 - no particular (prescribed) order of the entries,
 - each person has only one birthday ($\textit{birthday}$ is a function),
 - two persons can have the same birthday.

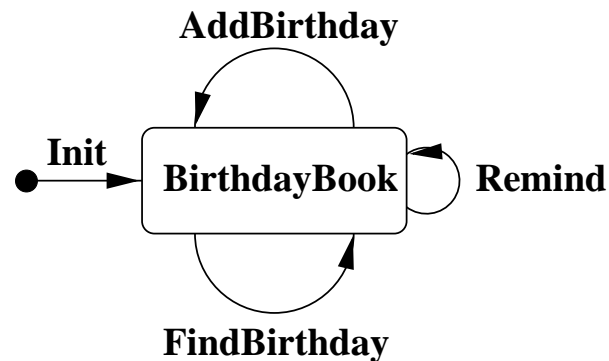
The Birthdaybook: Z-Specification (4)

Step 3. Define the **operations** of the system (based on the relations of the state-space).

- Some operations **modify** the state and some leave it **unchanged**.
- Some operations have input and/or output:



- Examples of operations: AddBirthday, FindBirthday, Remind (and Init).



The Birthdaybook: Z-Specification (5)

Add the birthday of a person, who is not yet known to the system:

$AddBirthday$ $\Delta BirthdayBook$ $name? : NAME$ $date? : DATE$ <hr/> $name? \notin known$ $birthday' =$ $birthday \cup \{name? \mapsto date?\}$	Name of operation (schema) structured import (symbol Δ) input of operation (symbol $?$) input of operation (symbol $?$) <hr/> precondition for success of operation extend the birthday function (if precondition is satisfied)
--	---

- This schema **modifies** the state:
 - it describes the state **before** (variables without $'$),
 - and that **after** the operation (variables with $'$).
- Note that we do not specify what happens when the precondition is not satisfied.
- It is possible to extend (refine) the specification so that an error message is generated.

The Birthdaybook: Z-Specification (6)

- We expect that *AddBirthday* extends the set of known names with the new name:

$$known' = known \cup \{name?\}$$

- We can use the specification of *AddBirthday* to **prove** this, by exploiting the invariant of the state before and after the operation:

$known'$	$=$	$\text{dom } birthday'$	[invariant after]
	$=$	$\text{dom}(birthday \cup \{name? \mapsto date?\})$	[spec of <i>Addbirthday</i>]
	$=$	$(\text{dom } birthday) \cup (\text{dom}\{name? \mapsto date?\})$	[fact about dom]
	$=$	$(\text{dom } birthday) \cup \{name?\}$	[fact about dom]
	$=$	$known \cup \{name?\}$	[invariant before]

- Proving such properties ensures that the specification is correct:

We can analyze the behavior of the system without having to implement it!

The Birthdaybook: Z-Specification (7)

Find the birthday of a person known to the system:

FindBirthday $\Xi \text{BirthdayBook}$ $name? : NAME$ $date! : DATE$	Name of operation (schema) structured import (symbol Ξ) input of operation (symbol $?$) output of operation (symbol $!$)
$name? \in known$ $date! = birthday(name?)$	precondition for success of operation output of operation (if successful)

This schema leaves the state **unchanged** and is equivalent to:

FindBirthday $\Delta \text{BirthdayBook}$ $name? : NAME$ $date! : DATE$
$known' = known$ $birthday' = birthday$ $name? \in known$ $date! = birthday(name?)$

The Birthdaybook: Z-Specification (8)

- Find out who has his birthday at some particular date:

$\begin{array}{l} \textit{Remind} \\ \hline \exists \textit{BirthdayBook} \\ \textit{today?} : \textit{DATE} \\ \textit{cards!} : \mathbb{P} \textit{NAME} \\ \hline \textit{cards!} = \{ n \in \textit{known} \mid \textit{birthday}(n) = \textit{today?} \} \end{array}$
--

cards! is a set of names, to whom "birthday-cards" should be sent.

- Initial state of the system:

$\begin{array}{l} \textit{InitBirthdayBook} \\ \hline \textit{BirthdayBook} \\ \hline \textit{known} = \emptyset \end{array}$

known = \emptyset implies *birthday* is also empty.

The Birthdaybook: Z-Specification (9)

- What does the Z-specification tell us about the implementation?
- It describes what the system does without specifying/prescribing how.
- For example, the Z-specification identifies **legal** and **illegal** data and operations. Illegal operations are for instance:
 - simultaneous addition of the birthdays of two persons,
 - addition of the birthday of a person who is already known to the system ($name? \in known$).An operation *ChangeBirthday* is not specified and could be added, or only realized in the implementation.
- More in the next classes.

Summary

- Z is an expressive language (PL1= and typed set-theory).
- Z supports structured, static and dynamic, modeling.
- More about Z:

<http://archive.comlab.ox.ac.uk/z.html>

- Tools and systems: see the course webpage.
 - ZETA (an open environment, including a type-checker; emacs Zeta-Mode)
[/usr/local/zeta](#)
 - HOL-Z tool (an embedding of Z in the theorem prover Isabelle).
 - Object-Z (an object-oriented extension of Z).
 - Books about Z and LaTeX style-file.

Formal Modeling with Z: Part II

Luca Viganò

Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Spring 2002

What Z is...

- Z is based on:
 - first-order logic with equality (PL1=)
 - typed set-theory
- Z supports the structured modeling of a system at an appropriate abstraction level, both static and dynamic, i.e.
 - formal modeling/specification of the data of a system,
 - functional modeling of the system (state transitions).
- Z allows one also to model and verify the refinement of a design:
 - Model a system by means of simple mathematical data-types.
 - Refinement of the model, based on the design-decisions, and that is closer to an implementation.
 - Refinement process can be iterated up to executable code.

...and what Z is not

- Z **not** intended for the modeling of:
 - non-functional properties such as usability, efficiency, reliability, etc.
 - temporal or concurrent behavior.
- Other methods, which can be used in combination with Z, are better suited for such modeling.

Overview

- The Z Language and The Mathematical Toolkit.
 - A *semantic library* is defined on top of the “basis language”.
- Next classes: Applications.

Main Components: Expressions and Predicates

- A small kernel, which we extend to the **Z mathematical toolkit**.
- **Expressions** E and **predicates** P :

$E ::= c_x \mid v_x$	[constants, variables]
$\mid E E$	[application]
$\mid (E, \dots, E)$	[cartesian product]
$\mid \{v_1 : E; \dots; v_n : E \mid P \bullet E\}$	[comprehension]
$\mid \langle \! \langle \text{tag}_1 \rightsquigarrow E, \dots, \text{tag}_n \rightsquigarrow E \rangle \! \rangle$	[tagged records (bindings)]
$\mid E.\text{tag}$	[element selection in record]

$P ::= \text{true} \mid \text{false} \mid c_x(E, \dots, E)$	
$\mid \neg P \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \mid P \Leftrightarrow P$	
$\mid \forall x : S \bullet P \mid \exists x : S \bullet P$	[\forall declaration • predicate]

Typed Set-Theory (1)

A brief overview of **typed set-theory**
(what is needed for the specification, refinement, and verification in Z).

- A **set** is a **(well-defined) collection of objects**.
- Small sets can be defined **by extension**:

$$Oceans == \{atlantic, arctic, indian, pacific\}$$

$$SEinstructors == \{Basin, Viganò', Wolff\}$$

- **Notation:** n is a (new) name for the expression e

$$n == e$$

Typed Set-Theory (2)

- **Notation** and some **axioms**

- Sets S and T are **equal** iff they have the same elements:

$$S = T \quad \text{iff} \quad (\forall x : S \bullet x \in T) \text{ and } (\forall x : T \bullet x \in S)$$

provided that x does not occur free in S or in T .

- Expression e belongs to a set defined by extension iff it is equal to one of the elements of the set:

$$e = u_1 \vee \dots \vee e = u_n \quad \text{iff} \quad e \in \{u_1, \dots, u_n\}$$

\Rightarrow Order and occurrence (frequency) of the elements is not important.

Set-Comprehension (Typed Set-Theory, 3)

- **Comprehension**: given a non-empty set S , it is possible to define a new set by considering only the elements of S that satisfy some property p :

$$\{x : S \mid p\} \quad \text{e.g.: } \{i : \mathbb{Z} \mid i \geq 0\} \text{ or } \{x : SEinstructors \mid x \in profs\}$$

- In general:

$$\{x : S; y : T \mid p \bullet e\} \quad \{\text{declaration} \mid \text{predicate} \bullet \text{expression}\}$$

- The value of $\{x : S; y : T \mid p \bullet e\}$ is the set of values of the expression e when the variables in the declaration take all the values that satisfy the predicate p . (Also: $\{\text{source} \mid \text{filter} \bullet \text{pattern}\}$.)
- E.g.: $\{x : Person; y : Class \mid (x \in profs) \wedge (teaches(x, y)) \bullet phone(x)\}$
- p and e are both optional:

$$\{x : S \mid p\} = \{x : S \mid p \bullet x\} \text{ and } \{x : S \bullet e\} = \{x : S \mid true \bullet e\}$$

Set-Comprehension (Typed Set-Theory, 3.1)

$\{\text{source} \mid \text{filter} \bullet \text{pattern}\}$

- A detailed example: “evaluation” of $\{i : \mathbb{N} \mid (i > 4) \bullet (2 * i) + 1\}$

Set-Comprehension (Typed Set-Theory, 3.1)

$\{\text{source} \mid \text{filter} \bullet \text{pattern}\}$

- A detailed example: “evaluation” of $\{i : \mathbb{N} \mid (i > 4) \bullet (2 * i) + 1\}$

Source (i.e. declaration) is the set of natural numbers:

$$\{i : \mathbb{N}\} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, \dots\}$$

Set-Comprehension (Typed Set-Theory, 3.1)

$$\{\text{source} \mid \text{filter} \bullet \text{pattern}\}$$

- A detailed example: “evaluation” of $\{i : \mathbb{N} \mid (i > 4) \bullet (2 * i) + 1\}$

Source (i.e. declaration) is the set of natural numbers:

$$\{i : \mathbb{N}\} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, \dots\}$$

We add the filter (i.e. predicate), i.e. only elements bigger than 4:

$$\{i : \mathbb{N} \mid (i > 4)\} = \{5, 6, 7, 8, \dots\}$$

Set-Comprehension (Typed Set-Theory, 3.1)

$$\{\text{source} \mid \text{filter} \bullet \text{pattern}\}$$

- A detailed example: “evaluation” of $\{i : \mathbb{N} \mid (i > 4) \bullet (2 * i) + 1\}$

Source (i.e. declaration) is the set of natural numbers:

$$\{i : \mathbb{N}\} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, \dots\}$$

We add the filter (i.e. predicate), i.e. only elements bigger than 4:

$$\{i : \mathbb{N} \mid (i > 4)\} = \{5, 6, 7, 8, \dots\}$$

Then we add the pattern (i.e. expression), and obtain the transformed elements:

$$\{i : \mathbb{N} \mid (i > 4) \bullet (2 * i) + 1\} = \{11, 13, 15, 17, \dots\}$$

Set-Comprehension (Typed Set-Theory, 3.1)

$$\{\text{source} \mid \text{filter} \bullet \text{pattern}\}$$

- A detailed example: “evaluation” of $\{i : \mathbb{N} \mid (i > 4) \bullet (2 * i) + 1\}$

Source (i.e. declaration) is the set of natural numbers:

$$\{i : \mathbb{N}\} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, \dots\}$$

We add the filter (i.e. predicate), i.e. only elements bigger than 4:

$$\{i : \mathbb{N} \mid (i > 4)\} = \{5, 6, 7, 8, \dots\}$$

Then we add the pattern (i.e. expression), and obtain the transformed elements:

$$\{i : \mathbb{N} \mid (i > 4) \bullet (2 * i) + 1\} = \{11, 13, 15, 17, \dots\}$$

- As a comparison: $\{i : \mathbb{N} \bullet (2 * i) + 1\} = \{1, 3, 5, 7, \dots\}$.
- Exercise: “evaluation” of $\{i : \mathbb{N}; j : \mathbb{N} \mid (j = 2 * i) \bullet i + j\}$

Power-sets and Cartesian Products (Typed Set-Theory, 4)

- The **power-set** $\mathbb{P} S$ is the set of all sub-sets of S , e.g.:

$$\mathbb{P}\{x, y\} = \{\emptyset, \{x\}, \{y\}, \{x, y\}\}$$

Axiom: $T \subseteq S$ iff $T \in \mathbb{P} S$

- The **cartesian product** $S \times T$ consists of all pairs (x, y) , where $x \in S$ and $y \in T$.

$$S \times T = \{x : S; y : T \mid true\}$$

Axioms:

- $(x_1, \dots, x_n) \in (S_1 \times \dots \times S_n)$ iff $(x_1 \in S_1) \wedge \dots \wedge (x_n \in S_n)$
- $t = (x_1, \dots, x_n)$ iff $(t.1 = x_1) \wedge \dots \wedge (t.n = x_n)$ (projection)

Union, Intersection, Difference (Typed Set-Theory, 5)

Axioms for **union**, **intersection**, **difference**:

- $x \in (S \cup T)$ iff $(x \in S) \vee (x \in T)$
- $x \in (S \cap T)$ iff $(x \in S) \wedge (x \in T)$
- $x \in (S \setminus T)$ iff $(x \in S) \wedge (x \notin T)$
- etc.

Types (Typed Set-Theory, 6)

- Each value x in a specification is assigned (has) exactly one **type**:
the biggest set (in the specification) for which $x \in S$.
- \mathbb{Z} has only one pre-defined type: **the integers** \mathbb{Z} .
- All other types are built from \mathbb{Z} , **basis-types** and **free types**.

\Rightarrow The **type expressions** are:

$\tau_B ::= \mathbb{Z}$	[the integers]
B	[Basis types and free types]
$\mathbb{P} \tau_B$	[Power-set types]
$\tau_B \times \dots \times \tau_B$	[Product types]
$[tag_1 \rightsquigarrow \tau_B, \dots, tag_n \rightsquigarrow \tau_B]$	[Tagged record types]

Types (Typed Set-Theory, 7)

- **Basis types**: internal structure is not further specified, e.g.

$[NAME]$

- **Free types**: an enumeration of constants, e.g.

$COLORS ::= red \mid orange \mid yellow \mid green \mid blue \mid indigo \mid violet$

(Similar to programming languages.)

Types (Typed Set-Theory, 8)

Let T and U be types.

- **Power-set types:** $\mathbb{P} T$ is the type of all subsets of T .
 - $\mathbb{P} T$ is the type of the set whose elements have type T .
 - Example: $\{-1, 0, 1, 2\}$ has type $\mathbb{P} \mathbb{Z}$.
- **Product types:** $T \times U$ is the type of all (ordered) pairs of elements of T and elements of U .
 - Examples:
 - * by definition: $_ + _$ has type $(\mathbb{Z} \times \mathbb{Z}) \rightarrow \mathbb{Z}$, and $(3, 4)$ has type $\mathbb{Z} \times \mathbb{Z}$.
 - * $COLORCODE == COLORS \times \mathbb{Z}$ has type $\mathbb{P}(COLORS \times \mathbb{Z})$.
 - Alternative notation for $T \times U$: $T \leftrightarrow U$ or $T \rightarrow U$.
- **Tagged record types:** later (defined as **schema types**).

Types (Typed Set-Theory, 9)

- Each value x in a specification is assigned (has) exactly one type.
 - The assignment is not explicit (syntactically impossible, as it is not possible to declare that a constant has a particular type)
 - but it is implicit in a declaration (by “set-membership”):
$$x : S \text{ with } S \text{ a set of type } \mathbb{P} \tau_B.$$
- Important:
 - Each type is a set, but not all sets are types!
 - A type is a “maximal” set.
 - Example:
 - * The natural numbers are not a type, but they are a set.
(Formal definition later.)
 - * A natural number (e.g. 1 or 3) has type \mathbb{Z} .

Types (Typed Set-Theory, 10)

- Each value x in a specification is assigned (has) exactly one type.
 - ⇒ **type-checking algorithms** can detect inconsistencies in the use of variable names and expressions (in a Z-document).
 - These algorithms are implemented in tools and systems that support Z (e.g. ZETA).
 - They increase the trust in a formal specification.
 - They cannot, however, verify the interpretation of names and inferences!
 - Example:
 $\{2, 4, \text{red}, \text{yellow}, 6\}$ [TYPE ERROR! Elements have different type]

Sets are typed, i.e. all elements must have the same type.

Types (Typed Set-Theory, 11)

- **Judgments** and **inference rules** allow one to identify well-typed/-formed expressions and predicates.

For example: to rule out $3 < \emptyset$ and $5 \wedge \mathbb{Z}$.

- Let Σ_B and Γ_B be a constants-environment and a variables-environment ($c :: \tau$ and $v :: \tau$, with τ in τ_B).
- **Def.:** A **judgment \vdash_E for expressions** is a tuple $(\Sigma_B, \Gamma_B, e, \tau)$, where $e \in E$ and $\tau \in \tau_B$. We then write:

$$\Sigma_B, \Gamma_B \vdash_E e :: \tau$$

“Expression e has type τ in the context of the environments Σ_B and Γ_B ”.

- **Def.:** A **judgment \vdash_P for predicates** is a tuple (Σ_B, Γ_B, p) , where $p \in P$. We then write:

$$\Sigma_B, \Gamma_B \vdash_P p$$

Types (Typed Set-Theory, 12)

- Some inference rules for expressions:

$$\frac{}{\{c :: \tau\} \cup \Sigma_B, \Gamma_B \vdash_E c :: \tau} \text{Const} \quad \frac{\Sigma_B, \Gamma_B \vdash_E e_1 :: \tau_1 \quad \dots \quad \Sigma_B, \Gamma_B \vdash_E e_n :: \tau_n}{\Sigma_B, \Gamma_B \vdash_E (e_1, \dots, e_n) :: \tau_1 \times \dots \times \tau_n} \text{Prod}$$

$$\frac{}{\Sigma_B, \{v :: \tau\} \cup \Gamma_B \vdash_E v :: \tau} \text{Var} \quad \frac{\Sigma_B, \Gamma_B \vdash_E e :: \tau_1 \rightarrow \tau_2 \quad \Sigma_B, \Gamma_B \vdash_E e' :: \tau_1}{\Sigma_B, \Gamma_B \vdash_E e e' :: \tau_2} \text{Appl}$$

- Some inference rules for well-formed predicates:

$$\frac{}{\Sigma_B, \Gamma_B \vdash_P \text{true}} \text{True} \quad \frac{\Sigma_B, \Gamma_B \vdash_P p}{\Sigma_B, \Gamma_B \vdash_P \neg p} \text{Not} \quad \frac{\Sigma_B, \Gamma_B \vdash_P p \quad \Sigma_B, \Gamma_B \vdash_P p'}{\Sigma_B, \Gamma_B \vdash_P p \Rightarrow p'} \text{Implies}$$

$$\frac{\Sigma_B, \Gamma_B \vdash_E e_1 :: \tau_1 \quad \dots \quad \Sigma_B, \Gamma_B \vdash_E e_n :: \tau_n \quad \Sigma_B, \Gamma_B \vdash_E p :: \mathbb{P}(\tau_1 \times \dots \times \tau_n)}{\Sigma_B, \Gamma_B \vdash_P p(e_1, \dots, e_n)} \text{PredAppl}$$

Types (Typed Set-Theory, 13)

- We can now show $_ + _ (3, 4) :: \mathbb{Z}$, where Σ_0 contains all arithmetic constant declarations (including $_ + _ :: (\mathbb{Z} \times \mathbb{Z}) \rightarrow \mathbb{Z}$, $3 :: \mathbb{Z}$ and $4 :: \mathbb{Z}$):

$$\frac{\frac{\frac{}{\Sigma_0, \Gamma \vdash_E _ + _ :: (\mathbb{Z} \times \mathbb{Z}) \rightarrow \mathbb{Z}} \text{Const} \quad \frac{\frac{\frac{}{\Sigma_0, \Gamma \vdash_E 3 :: \mathbb{Z}} \text{Const} \quad \frac{\frac{}{\Sigma_0, \Gamma \vdash_E 4 :: \mathbb{Z}} \text{Const}}{\Sigma_0, \Gamma \vdash_E (3, 4) :: \mathbb{Z} \times \mathbb{Z}} \text{Prod}}{\Sigma_0, \Gamma \vdash_E (3, 4) :: \mathbb{Z} \times \mathbb{Z}} \text{Appl}}{\Sigma_0, \Gamma \vdash_E _ + _ (3, 4) :: \mathbb{Z}} \text{Appl}$$

- Try to prove $3 < \emptyset$ or $5 \wedge \mathbb{Z}$!

Types (Typed Set-Theory, 14)

- Our use of types restricts the way we define and use sets.
 - E.g.: $x \in S$ is valid only when the type of S is the power-set of type of x .
- These restrictions are however welcome, as they rule out the paradoxes of untyped set-theory!

E.g.: the set of sets that are not element of themselves (cf. Russell's barber paradox) is not typable!

Definitions of Objects in Z (1)

- **Declarations:**
 - **Basis types.**
 - **Free types.**
 - **New variables:**

$$x : S$$

where S is either \mathbb{Z} or an already defined set (but not necessarily a type).

- **Examples:**

$i : \mathbb{Z}$	$[i \text{ is an integer}]$
$n : \mathbb{N}$	$[n \text{ is a natural number } (n \text{ has type } \mathbb{Z})]$

Definitions of Objects in Z (2)

- **Abbreviations:**

$symbol == term$ e.g.: $SEinstructors == \{Basin, Viganò, Wolff\}$

- $symbol$ is a new name for an already defined $term$.
- Each abbreviation can be eliminated from a specification.

- **Generic abbreviation:** introduce parameterized symbols

$symbol\ parameters == term$

Example:

$$\mathbb{P}_1 T == \{a : \mathbb{P} T \mid a \neq \emptyset\} \quad \Rightarrow \quad \mathbb{P}_1\{0, 1\} = \{\{0\}, \{1\}, \{0, 1\}\}$$

We can also define generic infix-symbols, e.g. $S\ rel\ T == \mathbb{P}(S \times T)$

Definitions of Objects in Z (3)

- **Axiomatic definitions**: introduce constraints on the definiendum.

$$\frac{x : S}{p}$$

$$\frac{\text{Declaration}}{\text{Predicate (axiom for object } x : S)}$$

- Corresponds to $(x : S) \wedge p$ (important for formal reasoning).
- Beware that in this way one can introduce inconsistencies in the specification.
- Example: formal definition of the **natural numbers**:

$$\frac{N : \mathbb{P}\mathbb{Z}}{\forall z : \mathbb{Z} \bullet (z \in N) \Leftrightarrow (z \geq 0)}$$

Definitions of Objects in Z (4)

- Special case: when p is *true* then

$$| x : S \quad (\text{can also be inconsistent, when } S = \emptyset)$$

- Generic axiomatic definitions

$$\frac{\frac{[X]}{x : X}}{p}$$

The set X is a formal parameter of the definition.

- There can be several parameters, declarations and predicates:

$$\frac{\begin{array}{l} \text{Declarations} \\ \hline \text{Predicate} \\ \dots \\ \text{Predicate} \end{array}}{d1, d2, d3 : \mathbb{Z} \quad \begin{array}{l} d1 + d2 = 7 \\ d1 < d2 \\ d1 + d3 = 0 \end{array}}$$

Corresponds to

$$(d1 : \mathbb{Z}) \wedge (d2 : \mathbb{Z}) \wedge (d3 : \mathbb{Z}) \wedge (d1 + d2 = 7) \wedge (d1 < d2) \wedge (d1 + d3 = 0)$$

Sets and predicates

- All objects in Z are sets
 \Rightarrow **predicates** are defined in terms of the sets of objects that satisfy them.

- Examples:

Definition of predicate “is a crowd” over sets of persons:

$$\frac{crowds : \mathbb{P}(\mathbb{P} Person)}{crowds = \{s : \mathbb{P} Person \mid \#s > 2\}} \quad (\# \text{ is size or cardinality of set } s)$$

so that

$$\begin{array}{ll} \{Jack, Janet, Chrissy\} \in crowds & \text{is true} \\ \{Adam, Eve\} \in crowds & \text{is false} \end{array}$$

Z and Booleans (1)

- Several formal (and programming) languages have a **Boolean data type**, with **true** and **false**.
- Z does not! (As it is not needed.)
- Example: Microwave oven (see semi-formal dynamic modeling)
 - In Z-look-alike language:

$$\frac{power, door : BOOLEAN \quad [This \text{ is not } Z! \text{ No built-in Boolean type}]}{power \Rightarrow door}$$

- Problem: when *door* is *true*, is it open or closed?
- In Z: use descriptive binary enumerations:

$$\begin{aligned} POWER &::= off \mid on \\ DOOR &::= closed \mid open \end{aligned}$$

so that we can then write: $(power = on) \Rightarrow (door = closed)$

Z and Booleans (2)

- Another example: relation *odd*
 - In Z-like language: *odd* as Boolean function

| $odd : \mathbb{Z} \rightarrow BOOLEAN$ [This is not Z! No built-in Boolean type]

and then $odd(3) = true$, $odd(4) = false$, etc.

- In Z: relations are sets (with \in and \notin)

| $odd : \mathbb{P}\mathbb{Z}$ and then $3 \in odd$, $4 \notin odd$, etc.

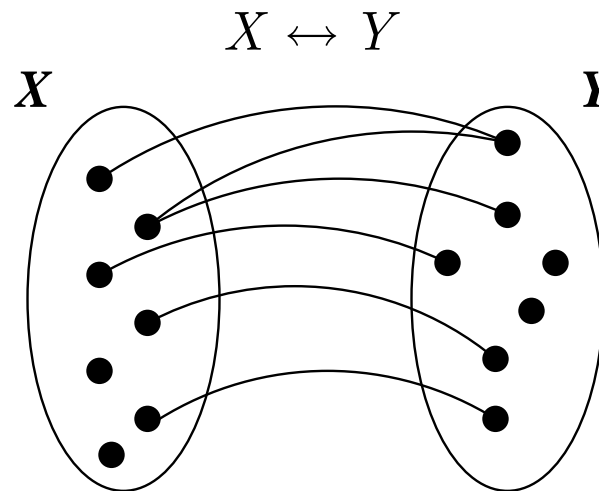
Or as prefix: | $odd_ : \mathbb{P}\mathbb{Z}$ and then $odd(3)$, $\neg odd(4)$, etc.

Relations (1)

- **Binary relation**: a set of ordered pairs (a subset of a Cartesian product)

$$X \leftrightarrow Y \equiv \mathbb{P}(X \times Y) \quad (X \text{ and } Y \text{ sets})$$

Graphical representation
by means of **Venn-diagrams**:



Relations (2)

Example: University telephone list (database)

$$telephones : Person \leftrightarrow Phone$$

$$\Rightarrow telephones \subseteq Person \times Phone$$

E.g.:

$$(Vigano', 8243) \in telephones$$

Equivalent **maplet** notation:

$$Vigano' \mapsto 8243 \in telephones$$

A person can have more than one telephone number:

$$Basin \mapsto 8240 \in telephones \quad \text{and} \quad Basin \mapsto 8241 \in telephones$$

Or two people can share a telephone number:

$$Wolff \mapsto 8247 \in telephones \quad \text{and} \quad Brucker \mapsto 8247 \in telephones$$

Relations (3)

- The full state of the system (corresponds to a table):

$telephones : Person \leftrightarrow Phone$

$telephones =$

$\{ Basin \mapsto 8240,$
 $Basin \mapsto 8241,$
 $Vigano' \mapsto 8243,$
 $Wolff \mapsto 8247,$
 $Brucker \mapsto 8247,$
 $Ayari \mapsto 8244,$
 $\dots \}$

Person	Phone
Basin	8240
Basin	8241
Vigano'	8243
Wolff	8247
Brucker	8247
Ayari	8244
...	...

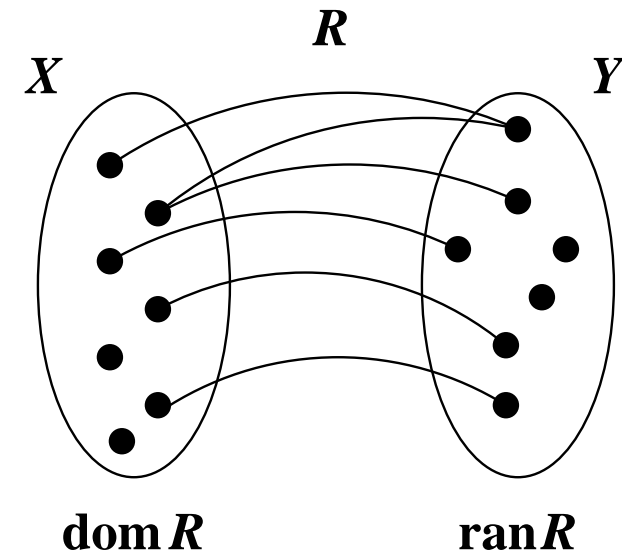
Relations (4)

Important **notation** and **properties**:

$first(Vigano', 8243) = Vigano'$ and $second(Vigano', 8243) = 8243$

Domain and **range** of a relation:

$[X, Y]$
$dom : (X \leftrightarrow Y) \rightarrow \mathbb{P} X$
$ran : (X \leftrightarrow Y) \rightarrow \mathbb{P} Y$
$\forall R : (X \leftrightarrow Y) \bullet$
$dom R = \{x : X \mid \exists y : Y \bullet (x, y) \in R\} \wedge$
$ran R = \{y : Y \mid \exists x : X \bullet (x, y) \in R\}$



Equivalent notation:

- $dom R = \{x : X; y : Y \mid x \mapsto y \in R \bullet x\}$
- $ran R = \{x : X; y : Y \mid x \mapsto y \in R \bullet y\}$

e.g.: $x \in dom\ telephones \quad \text{iff} \quad \exists y : Phone \bullet x \mapsto y \in telephones.$

Operations on relations: Queries (Relations, 5)

- There are several rules and axioms about dom and ran.
- **Domain restriction** $_ \triangleleft _$ and **anti-restriction** $_ \triangleleft _$

$$\begin{array}{l}
 [X, Y] \\
 \hline
 _ \triangleleft _, _ \triangleleft _ : (\mathbb{P} X) \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y) \\
 \hline
 \forall A : \mathbb{P} X; R : (X \leftrightarrow Y) \bullet \\
 \quad A \triangleleft R = \{x : X; y : Y \mid (x \mapsto y \in R) \wedge (x \in A) \bullet x \mapsto y\} \wedge \\
 \quad A \triangleleft R = \{x : X; y : Y \mid (x \mapsto y \in R) \wedge (x \notin A) \bullet x \mapsto y\}
 \end{array}$$

- **range restriction** $_ \triangleright _$ and **anti-restriction** $_ \triangleright _$

$$\begin{array}{l}
 [X, Y] \\
 \hline
 _ \triangleright _, _ \triangleright _ : (X \leftrightarrow Y) \times (\mathbb{P} Y) \rightarrow (X \leftrightarrow Y) \\
 \hline
 \forall R : (X \leftrightarrow Y); A : \mathbb{P} Y \bullet \\
 \quad R \triangleright A = \{x : X; y : Y \mid (x \mapsto y \in R) \wedge (y \in A) \bullet x \mapsto y\} \wedge \\
 \quad R \triangleright A = \{x : X; y : Y \mid (x \mapsto y \in R) \wedge (y \notin A) \bullet x \mapsto y\}
 \end{array}$$

Operations on relations: Queries (Relations, 5.1)

- Examples:

- $\{Basin, Wolff\} \triangleleft telephones = \{Basin \mapsto 8240, Basin \mapsto 8241, Wolff \mapsto 8247\}$
- $\{Basin, Wolff, Viganò'\} \triangleleft telephones \triangleright \{8240, 8247\} = \{Basin \mapsto 8241, Viganò' \mapsto 8243\}$

- N.B.: $A_1 \triangleleft R \triangleright A_2 = (A_1 \triangleleft R) \triangleright A_2 = A_1 \triangleleft (R \triangleright A_2)$

Operations on relations: Overriding (Relations, 6)

- **Overriding** \oplus models database updates:

$$\begin{array}{l}
 \hline [X, Y] \hline \\
 \hline - \oplus - : (X \leftrightarrow Y) \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y) \\
 \hline \\
 \hline \forall R, S : (X \leftrightarrow Y) \bullet \\
 \quad R \oplus S = ((\text{dom } S) \triangleleft R) \cup S \\
 \hline
 \end{array}$$

Example:

$$\begin{array}{ll}
 \text{telephones} = & \text{telephones} \oplus \{Brucker \mapsto 8248, Ayari \mapsto 8249\} = \\
 \{Basin \mapsto 8240, & \{Basin \mapsto 8240, \\
 \dots, & \dots, \\
 Brucker \mapsto 8247, & \Rightarrow Brucker \mapsto 8248, \\
 Ayari \mapsto 8244, & Ayari \mapsto 8249, \\
 \dots\} & \dots\}
 \end{array}$$

Operations on relations: Images (Relations, 7)

- Relational image of A under R :

$$\begin{array}{l}
 [X, Y] \\
 \hline
 -(_) : (X \leftrightarrow Y) \times (\mathbb{P} X) \rightarrow (\mathbb{P} Y) \\
 \hline
 \forall R : (X \leftrightarrow Y); A : \mathbb{P} X \bullet \\
 \quad R(_)A = \{y : Y \mid \exists x : A \bullet x \mapsto y \in R\}
 \end{array}$$

$$\Rightarrow R(_)A = \text{ran}(A \triangleleft R)$$

Example:

$$\begin{aligned}
 \text{telephones}(\{Basin, Wolff\}) &= \text{ran}(\{Basin, Wolff\} \triangleleft \text{telephones}) \\
 &= \{8240, 8241, 8247\}
 \end{aligned}$$

Operations on relations: Inverse (Relations, 8)

- **Relational inverse:**

- Relations are “directed”, i.e. $R : (X \leftrightarrow Y)$ relates an element of X with one of Y .
- $_{\sim}$ inverts source and target of a relation.

$\frac{[X, Y]}{_{\sim} : (X \leftrightarrow Y) \rightarrow (Y \leftrightarrow X)}$ <hr/> $\forall R : (X \leftrightarrow Y) \bullet$ $R^{\sim} = \{x : X; y : Y \mid (x \mapsto y \in R) \bullet y \mapsto x\}$

- Example:

$$\{Wolff \mapsto 8247, Viganò' \mapsto 8243\}^{\sim} = \{8247 \mapsto Wolff, 8243 \mapsto Viganò'\}$$

Operations on relations: Composition (Relations, 9)

- Composition of relations:

$$\begin{array}{c}
 [X, Y, Z] \text{-----} \\
 \hline
 - \circ - : (X \leftrightarrow Y) \times (Y \leftrightarrow Z) \rightarrow (X \leftrightarrow Z) \\
 \hline
 \forall R : (X \leftrightarrow Y); S : (Y \leftrightarrow Z) \bullet \\
 R \circ S = \{x : X, z : Z \mid (\exists y : Y \bullet (x \mapsto y \in R) \wedge (y \mapsto z \in S)) \bullet x \mapsto z\}
 \end{array}$$

Example:

telephones : *Person* \leftrightarrow *Phone* and *departments* : *Phone* \leftrightarrow *Department*
 $\Rightarrow \{ \text{Vigano}' \mapsto \text{software} - \text{engineering} \} \in \text{telephones} \circ \text{departments}$

Also: backwards directed composition:

$$\begin{array}{c}
 [X, Y, Z] \text{-----} \\
 \hline
 - \circ - : (Y \leftrightarrow Z) \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Z) \\
 \hline
 \forall R : (X \leftrightarrow Y); S : (Y \leftrightarrow Z) \bullet \\
 S \circ R = R \circ S
 \end{array}$$

Operations on relations: Iteration and Closures (Rel., 10)

- Iterated composition:

$$\begin{array}{l}
 [X] \\
 \hline
 \text{iter} : \mathbb{Z} \rightarrow (X \leftrightarrow X) \rightarrow (X \leftrightarrow X) \\
 \hline
 \forall R : (X \leftrightarrow X) \bullet \\
 \quad \text{iter } 0 \ R = \text{id } X \wedge \\
 \quad (\forall i : \mathbb{Z} \bullet \text{iter}(i+1) \ R = R \circ (\text{iter } i \ R)) \wedge \\
 \quad (\forall i : \mathbb{Z} \bullet \text{iter}(-i) \ R = R \circ (\text{iter } i \ (R^\sim)))
 \end{array}$$

- Closures: transitive closure $_^+$ and reflexive-transitive closure $_^*$

$$\begin{array}{l}
 [X] \\
 \hline
 _^+, _^* : (X \leftrightarrow X) \rightarrow (X \leftrightarrow X) \\
 \hline
 \forall R : (X \leftrightarrow X) \bullet \\
 \quad R^+ = \bigcap \{S : (X \leftrightarrow X) \mid (R \subseteq S) \wedge (S \circ S \subseteq S)\} \wedge \\
 \quad R^* = \bigcap \{S : (X \leftrightarrow X) \mid (\text{id } X \subseteq S) \wedge (R \subseteq S) \wedge (S \circ S \subseteq S)\}
 \end{array}$$

Latex and E-mail Formats

name	type	latex	e-mail	math
integers	$\mathbb{P}\mathbb{Z}$	<code>\num</code>	<code>%Z</code>	\mathbb{Z}
naturals	$\mathbb{P}\mathbb{Z}$	<code>\nat</code>	<code>%N</code>	\mathbb{N}
upto	$\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{P}\mathbb{Z}$	<code>..</code>	<code>..</code>	\dots
numbers	\mathbb{Z}	<code>-2,0,5,...</code>	<code>-2,0,5,...</code>	$-2, 0, 5, \dots$
less	$\mathbb{Z} \leftrightarrow \mathbb{Z}$	<code><</code>	<code><</code>	$_ < _$
equality	$\alpha \leftrightarrow \alpha$	<code>=</code>	<code>=</code>	$=$
emptiness	$\mathbb{P}\alpha$	<code>\{\}</code>	<code>\{ }</code>	$\{ \}$
membership	$\alpha \leftrightarrow \mathbb{P}\alpha$	<code>\in</code>	<code>%e</code>	$_ \in _$
membership	$\alpha \leftrightarrow \mathbb{P}\alpha$	<code>:</code>	<code>:</code>	$_ : _$
subset	$\mathbb{P}\alpha \leftrightarrow \mathbb{P}\alpha$	<code>\subseteq</code>	<code>%c_</code>	$_ \subseteq _$
union	$\mathbb{P}\alpha \times \mathbb{P}\alpha \rightarrow \mathbb{P}\alpha$	<code>\union</code>	<code>%u</code>	$_ \cup _$

name	type	latex	e-mail	math
set of subsets	$\mathbb{P} \alpha \rightarrow \mathbb{P} \mathbb{P} \alpha$	<code>\power</code>	<code>%P</code>	\mathbb{P}
cardinality	$\mathbb{P} \alpha \rightarrow \mathbb{Z}$	<code>\#</code>	<code>#</code>	$\#$
cartesian product	$\mathbb{P} \alpha \times \mathbb{P} \beta \rightarrow \mathbb{P}(\alpha \times \beta)$	<code>\cross</code>	<code>%x</code>	$_ \times _$
union	$\mathbb{P} \alpha \times \mathbb{P} \alpha \rightarrow \mathbb{P} \alpha$	<code>\union</code>	<code>%u</code>	$_ \cup _$
relation	$\mathbb{P} \alpha \times \mathbb{P} \beta \rightarrow \mathbb{P} \mathbb{P}(\alpha \times \beta)$	<code>\rel</code>	<code><-></code>	$_ \leftrightarrow _$
domain	$\mathbb{P} \mathbb{P}(\alpha \times \beta) \rightarrow \mathbb{P} \alpha$	<code>\dom</code>	<code>\dom</code>	dom
domain restriction	$\mathbb{P} \alpha \times (\alpha \leftrightarrow \beta) \rightarrow (\alpha \leftrightarrow \beta)$	<code>\dres</code>	<code>< </code>	$_ \triangleleft _$
override	$(\alpha \leftrightarrow \beta) \times (\alpha \leftrightarrow \beta) \rightarrow (\alpha \leftrightarrow \beta)$	<code>\oplus</code>	<code>(+)</code>	$_ \oplus _$

Formal Modeling with Z: Part III

Luca Viganò

Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Spring 2002

Overview

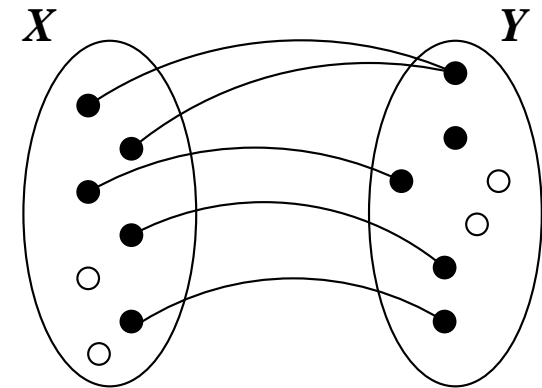
- We continue with: The Z Language and The Mathematical Toolkit.
 - A *semantic library* is defined on top of the “basis language”.
- Next classes: Applications.

Functions (1)

Functions are particular relations, where each element of a set is related to at most one element of another set.

The set $X \rightarrowtail Y$ of the **partial functions** is:

$$X \rightarrowtail Y == \{f : X \leftrightarrow Y \mid \forall x : X; y_1, y_2 : Y \bullet (x \mapsto y_1 \in f) \wedge (x \mapsto y_2 \in f) \Rightarrow y_1 = y_2\}$$



Example (from *Birthdaybook*): $birthday : NAME \rightarrowtail DATE$

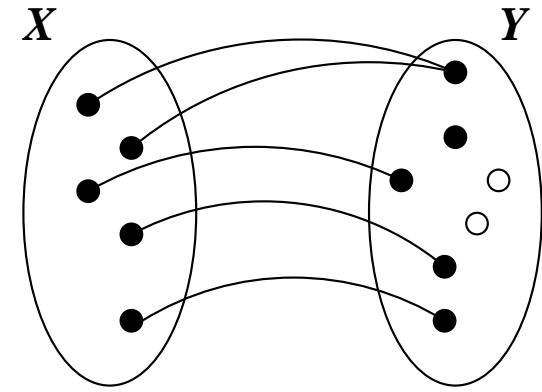
but $telephones : Person \leftrightarrow Phone$ is not a function!

Functions (2)

A **total function** from X to Y is a partial function from X to Y , which maps each element of X to exactly one element of Y .

The set $X \rightarrow Y$ of all such functions is:

$$X \rightarrow Y == \{f : X \rightarrowtail Y \mid \text{dom } f = X\}$$



Examples:

- Total function *double*

$$\frac{\text{double} : \mathbb{N} \leftrightarrow \mathbb{N}}{\forall m, n : \mathbb{N} \bullet (m \mapsto n \in \text{double}) \Leftrightarrow (m + m = n)}$$

- Functions on relations: in the definitions of dom , \triangleleft , \oplus , etc.
e.g.: $-\oplus- : (X \leftrightarrow Y) \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y)$

Examples of function application (Functions, 3)

- Evaluation of $double(double(2))$.

$$\frac{double : \mathbb{N} \leftrightarrow \mathbb{N}}{\forall m, n : \mathbb{N} \bullet (m \mapsto n \in double) \Leftrightarrow (m + m = n)}$$

corresponds (informally, that is with ‘...’)

$$double = \{0 \mapsto 0, 1 \mapsto 2, 2 \mapsto 4, \dots\}$$

so that $double(double(2)) = double(4) = 8$

- $(- + -) = \{\dots, (1, 1) \mapsto 2, (1, 2) \mapsto 3, \dots, (3, 4) \mapsto 7, \dots\}$, so that

$$(- + -)(3, 4) = 7$$

Rules for function application (Functions, 4)

$$\frac{\exists_1 p : f \bullet p.1 = a \quad (a \mapsto b) \in f}{b = f(a)} \text{ app-intro (if } b \text{ does not occur free in } a)$$

$$\frac{\exists_1 p : f \bullet p.1 = a \quad b = f(a)}{(a \mapsto b) \in f} \text{ app-elim (if } b \text{ does not occur free in } a)$$

Lambda notation (Functions, 5)

We can express

$$f = \{x : X \mid p \bullet x \mapsto e\}$$

also by means of Lambda notation:

$$f = \lambda x : X \mid p \bullet e$$

λ Declaration | Constraint • Expression

Example:

$$\left| \begin{array}{l} \text{double} : \mathbb{N} \leftrightarrow \mathbb{N} \\ \hline \text{double} = \lambda m : \mathbb{N} \bullet m + m \end{array} \right|$$

$$\left| \begin{array}{l} \text{double} : \mathbb{N} \leftrightarrow \mathbb{N} \\ \hline \forall m, n : \mathbb{N} \bullet \\ (m \mapsto n \in \text{double}) \Leftrightarrow (m + m = n) \end{array} \right|$$

Properties of functions (Functions, 6)

- Set $X \rightharpoonup Y$ of **partial injections**:

$$X \rightharpoonup Y == \{f : X \rightarrowtail Y \mid f^\sim \in Y \rightarrowtail X\}$$

- Set $X \rightarrowtail Y$ of **total injections**:

$$X \rightarrowtail Y == \{f : X \rightarrow Y \mid f^\sim \in Y \rightarrowtail X\}$$

- Set $X \twoheadrightarrowtail Y$ of **partial surjections**:

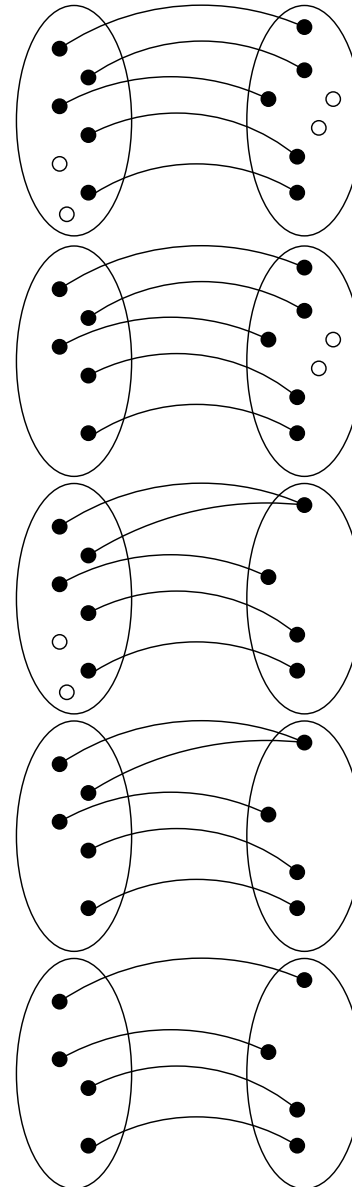
$$X \twoheadrightarrowtail Y == \{f : X \rightarrowtail Y \mid \text{ran } f = Y\}$$

- Set $X \twoheadrightarrow Y$ of **total surjections**:

$$X \twoheadrightarrow Y == \{f : X \rightarrow Y \mid \text{ran } f = Y\}$$

- Set $X \rightarrowtailtail Y$ of **total bijections**:

$$X \rightarrowtailtail Y == (X \rightarrowtail Y) \cap (X \twoheadrightarrow Y)$$



Finite sets and finite functions (Functions, 7)

A **finite set** F is a set whose elements are enumerable up to a natural number n .

- That is, there is a total bijection with domain $1, 2, \dots, n$ and range F .
- Example: $n = 3$ and $F = \{a, b, c\} \implies \{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}$.
- **Number range** operator ('upto'):

$$\frac{}{\vdash \dots \vdash : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{P} \mathbb{N}}$$

$$\vdash \forall m, n : \mathbb{N} \bullet m .. n = \{i : \mathbb{N} \mid m \leq i \leq n\}$$

\implies **Set of all finite subsets:**

$$\mathbb{F} X == \{ s : \mathbb{P} X \mid \exists n : \mathbb{N} \bullet \exists f : (1 .. n) \twoheadrightarrow s \bullet true \}$$

Finite sets and finite functions (Functions, 8)

When the set X is finite then

- $\mathbb{F} X = \mathbb{P} X$
- $\#$ is the size (cardinality) of X

$[X]$	
$\# : \mathbb{F} X \rightarrow \mathbb{N}$	
$\forall s : \mathbb{F} X; n : \mathbb{N} \bullet$	
$n = \#s \Leftrightarrow \exists f : (1 .. n) \twoheadrightarrow s \bullet true$	

Finite function: domain is a finite set.

- The sets of finite functions (or injections) from A to B are:

$$A \twoheadrightarrow B == \{f : A \rightarrow B \mid \text{dom } f \in \mathbb{F} A\} \quad \text{or} \quad A \twoheadrightarrow B == (A \twoheadrightarrow B) \cap (A \twoheadrightarrow B)$$

Sequences (1)

- **Sequences** are ordered collections (unlike sets).
- Example: the months form a sequence.

The name declaration

$$MONTHS ::= jan \mid feb \mid mar \mid apr \mid may \mid jun \mid jul \mid aug \mid sep \mid oct \mid nov \mid dec$$

does not however imply an ordering \implies seq-operator

$$\frac{year : \text{seq } MONTHS}{year = \langle jan, feb, mar, apr, jun, jul, aug, sep, oct, nov, dec \rangle}$$

- Example:
 - $\langle \langle feb, mar \rangle, \langle \rangle, \langle apr \rangle \rangle \in \text{seq}(\text{seq } MONTHS)$ [empty sequence $\langle \rangle$]
 - $\langle 77, 5, 6, 18, 43, 61 \rangle \in \text{seq } \mathbb{N}$
 - $\langle \{1, 2, 83\}, \emptyset \rangle \in \text{seq}(\mathbb{P} \mathbb{N})$

Sequences (2)

- The set of **finite sequences** of elements of the set X is

$$\text{seq } X == \{f : \mathbb{N} \multimap X \mid \exists n : \mathbb{N} \bullet \text{dom } f = 1 \dots n\}$$

so that $\langle \text{apr}, \text{jan}, \text{dec}, \text{sep} \rangle$ is an alternative notation for the set

$$\{1 \mapsto \text{apr}, 2 \mapsto \text{jan}, 3 \mapsto \text{dec}, 4 \mapsto \text{sep}\}$$

- N.B.: here we only considered **finite** sequences, but one can extend \mathbb{Z} with a theory of **infinite** sequences.

Operations on Sequences (3)

- Sequences are functions \implies we can apply them on numbers, e.g.

$$\sigma = \langle apr, jan, dec, sep \rangle \implies \sigma(3) = dec$$

- Functions are sets \implies we can apply set-operators, e.g.

$$\# \langle apr, jan, dec, sep \rangle = 4$$

- Concatenation** (sequence constructor):

$[X]$ $- \frown - : (\text{seq } X) \times (\text{seq } X) \rightarrow (\text{seq } X)$ <hr style="width: 30%; margin-left: 0;"/> $\forall \sigma, \tau : \text{seq } X \bullet$ $\sigma \frown \tau = \sigma \cup \{n : \text{dom } \tau \bullet (n + \#\sigma) \mapsto \tau(n)\}$

Operations on Sequences (4)

- Sequence destructors: **head**, **tail**

$[X]$
$head : seq\ X \rightarrow X$ $tail : seq\ X \rightarrow seq\ X$
$head = \forall \sigma : seq\ X \mid \sigma \neq \langle \rangle \bullet$ $\quad head\ \sigma = \sigma(1)$ $\forall \sigma : seq\ X \mid \sigma \neq \langle \rangle \bullet$ $\quad \#tail\ \sigma = \#\sigma - 1$ $\quad \forall i : 1 .. \#\sigma - 1 \bullet (tail\ \sigma)(i) = \sigma(i + 1)$

so that when $\sigma \neq \langle \rangle$

$$\langle head\ \sigma \rangle \frown \langle tail\ \sigma \rangle = \sigma$$

Operations on Sequences (5)

- **Filtering:** $\sigma \upharpoonright F$ is the sequence σ ‘filtered’ by the set $F \subseteq X$ (order of elements is respected).
- **Extraction:** $E \upharpoonright \sigma$ is the sequence of elements of σ , which occur in a position in σ , whose index occurs in the set $E \subseteq \mathbb{N}_1$.
- Examples:
 - $\langle a, b, c, d, e, d, c, b, a \rangle \upharpoonright \{a, d\} = \langle a, d, d, a \rangle$
 - $\langle \rangle \upharpoonright F = \langle \rangle$
 - $\sigma \upharpoonright \emptyset = \langle \rangle$
 - $\{0, 1, 3, 5\} \upharpoonright \langle apr, jan, dec, sep \rangle = \langle apr, dec \rangle$
 - $E \upharpoonright \langle \rangle = \langle \rangle$
- Formal definition (and further operators): see literature.

Further Sequences (6)

- The set of the **non-empty finite sequences** of elements of the set X is

$$\text{seq}_1 X == \{f : \text{seq } X \mid f \neq \langle \rangle\}$$

- The set of **injective finite sequences** of elements of the set X is

$$\text{iseq } X == \text{seq } X \cap (\mathbb{N} \multimap X)$$

An injective sequence contains no repetitions or duplicates, e.g.

- $\langle jan, feb, mar \rangle$ is injective,
- $\langle jan, feb, jan \rangle$ not.

Further Operations on Sequences (7)

Further operations on sequences can be found in the literature, including:

- Sequence constructor: **distributed concatenation**
- Sequence destructors: **front** and **last**

$$\begin{array}{l}
 [X] \\
 \hline
 \text{front} : (\text{seq}_1 X) \rightarrow (\text{seq } X) \\
 \text{last} : (\text{seq}_1 X) \rightarrow X \\
 \hline
 \text{front} = \lambda \sigma : \text{seq}_1 X \bullet (1 \dots \# \sigma - 1) \triangleleft \sigma \\
 \text{last} = \lambda \sigma : \text{seq}_1 \bullet \sigma(\# \sigma)
 \end{array}$$

Head and tail can alternatively be defined with λ and seq_1 .

Further Operations on Sequences (8)

Further operations on sequences can be found in the literature, including:

- **Reverse:**

$$\begin{array}{l} [X] \\ \hline rev_ : (\text{seq } X) \rightarrow (\text{seq } X) \\ \hline \forall \sigma : \text{seq } X \bullet \\ \quad rev \sigma = \lambda n : \text{dom } \sigma \bullet \sigma(\# \sigma - n + 1) \end{array}$$

Multisets

- **Multisets** (or **bags**) are sets in which elements may occur more than once (the order is not important, as for sets).
- The set of **multisets** of elements of the set X is

$$\text{bag } X == X \rightarrow \mathbb{N} \setminus \{0\}$$

- Finite multisets are “parenthesized” with ‘ \llbracket ’ and ‘ \rrbracket ’, and the element-symbol is Ξ , so that
 - $\llbracket 3, 5, 3, 1, 9 \rrbracket = \llbracket 1, 3, 3, 5, 9 \rrbracket \neq \llbracket 1, 3, 5, 9 \rrbracket$,
 - $3 \Xi \llbracket 3, 5, 3, 1, 9 \rrbracket$.
- See literature for further operators, such as union, intersection, difference, and ‘occurrence-count’.

Schemata (1)

- Z consists of two sub-languages:
 - **mathematical language**, which allows us to model design aspects, objects and their relations.
 - **schema language**, which allows us **structure**, **compose** and **split** modelings (data, functions and predicates).
- Z-schemata
 - can be used as declarations, as types, and as predicates,
 - can model the state space (states and state transitions) of a system,
 - can be used to verify formal modelings.

Syntax of Schemata (Schemata, 2)

- Vertical syntax:

name
declaration of typed variables (represent observations of the state)
relationships between values of vars (invariants of the system)

<i>SchemaOne</i>
$a : \mathbb{Z}$ $c : \mathbb{P} \mathbb{Z}$
$a \in c$ $c \neq \emptyset$

- Horizontal syntax:

name $\hat{=}$ [declarations | predicate]

- Name and constraints are optional, e.g.

$a : \mathbb{Z}$ $c : \mathbb{P} \mathbb{Z}$

corresponds to

$a : \mathbb{Z}$ $c : \mathbb{P} \mathbb{Z}$
<i>true</i>

Schemata as Types and Declarations (Schemata, 3)

- To introduce **tagged record-types**.

$\begin{array}{l} \text{SchemaTwo} \\ a : \mathbb{Z} \\ c : \mathbb{P}\mathbb{Z} \end{array}$

corresponds to a **composite data-type** with two different components: an integer a and a set of integers c .

- We can represent it as a **binding**, e.g.

$\langle a \rightsquigarrow 2, c \rightsquigarrow \{1, 2, 3\} \rangle$ $[a \text{ is bound to } 2, c \text{ to } \{1, 2, 3\}]$

- Components are “saved” **by name** (in Cartesian products: **by position**), and are selected by the operator ‘ $_{-}._{-}$ ’.

When s is an object of schema-type *SchemaOne* is, then

- $s.a$ is its integer component,
- $s.c$ is its set component.

Schemata as Types and Declarations (Schemata, 4)

- Bindings express the **semantics** of schemata:

A	
$c_0 : S_0$	
\dots	
$c_n : S_n$	
P	

is equivalent to

$$A \hat{=} \{c_0 : S_0; \dots; c_n : S_n \mid P \bullet \langle c_0 \rightsquigarrow c_0, \dots, c_n \rightsquigarrow c_n \rangle\}$$

- This is a **characteristic binding**: each component is bound to a value with the same name.

Schemata as Types and Declarations (Schemata, 5)

Example:

<i>SchemaThree</i>	
$a : \mathbb{Z}$	
$c : \mathbb{P}\mathbb{Z}$	
$a \in c$	
$c \neq \emptyset$	
$c \subseteq \{0, 1\}$	

describes the set of bindings

$$\text{SchemaThree} = \{ \langle a \rightsquigarrow 0, c \rightsquigarrow \{0\} \rangle, \langle a \rightsquigarrow 0, c \rightsquigarrow \{0, 1\} \rangle, \\ \langle a \rightsquigarrow 1, c \rightsquigarrow \{1\} \rangle, \langle a \rightsquigarrow 1, c \rightsquigarrow \{0, 1\} \rangle \}$$

that is, the set

$$\{ a : \mathbb{Z}; c : \mathbb{P}\mathbb{Z} \mid (a \in c) \wedge (c \neq \emptyset) \wedge (c \subseteq \{0, 1\}) \bullet \langle a \rightsquigarrow a, c \rightsquigarrow c \rangle \}$$

Schemata as Types and Declarations (Schemata, 6)

- N.B.: in a characteristic binding $\langle a \rightsquigarrow a \rangle$, the component (i.e. the tag) a on the left of \rightsquigarrow is bound to the value of the variable a on the right of \rightsquigarrow !
- When S is the name of the schema, we can write θS to denote the characteristic binding of the components of S , e.g.

$$\theta SchemaThree = \langle a \rightsquigarrow a, c \rightsquigarrow c \rangle$$

so that the set of bindings of *SchemaThree* in the previous slide are equal to

$$\{SchemaThree \bullet \theta SchemaThree\}$$

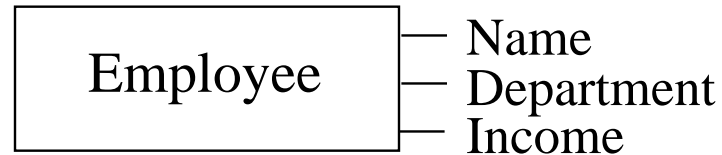
- When a schema name is used, where one would expect a set or a type, then it represents the corresponding set of bindings. For each schema S , the declaration $s : S$ is an abbreviation of $a : \{S \bullet \theta S\}$. (The variable s is declared to be a binding of appropriate type that meets the constraint part of schema S .)
- Details: see J. Woodcock and J. Davies, *Using Z*, pages 154–158.

Data Modeling with Z and E/R-diagrams (1)

- E/R-diagrams can be translated into Z specifications.
- In particular: Z allows one to formalize the constraints on data, which are implicit in E/R-diagrams.
- Translation takes place in 3 steps:
 1. The basis entities.
 2. The aggregate entities.
 3. The relations.

Basis entities (Z and E/R-diagrams, 2)

The basis entity



corresponds to the Z-schema (with built-in constraints):

$[NAME]$

$DEPARTMENT ::= A \mid B \mid C$

Employee

Name : *NAME*

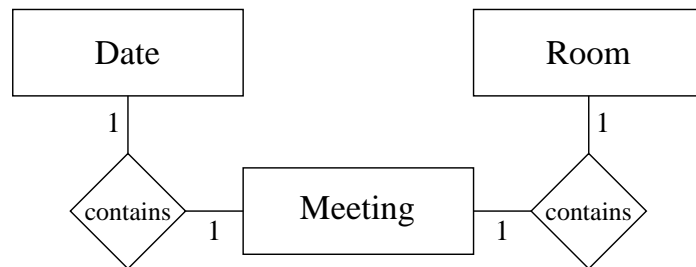
Dept : *DEPARTMENT*

Income : 1000 .. 5000

if *Dept* = *A* **then** $(1000 \leq Income) \wedge (Income \leq 3000)$
 else if *Dept* = *B* **then** $(3000 \leq Income) \wedge (Income \leq 5000)$
 else $(4000 \leq Income) \wedge (Income \leq 6000)$

Aggregate entities (Z and E/R-diagrams, 3)

The aggregate entity



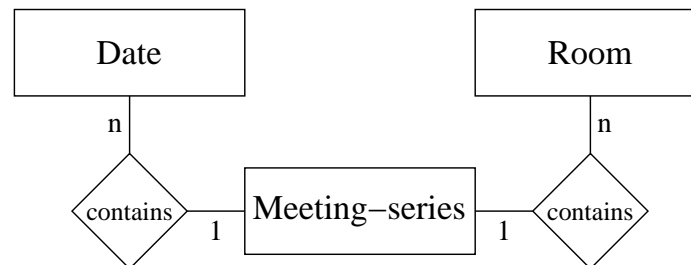
corresponds to

$$Meeting == Date \times Room$$

or

$$Meeting == Room \times Date$$

and



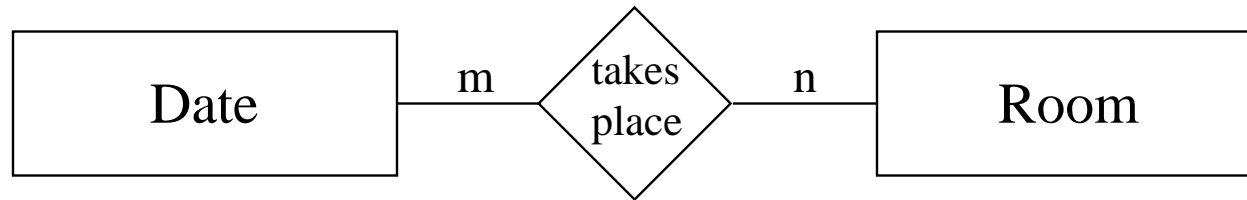
corresponds to

$$Meeting - Series == (\text{seq } Date) \times (\text{seq } Room)$$

or

$$Meeting - Series == (\mathbb{P} Room) \times (\mathbb{P} Date)$$

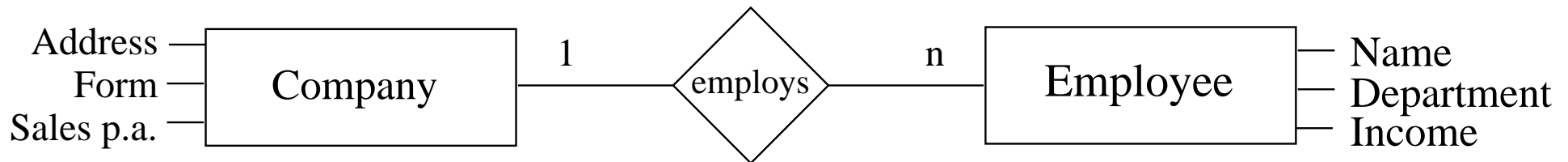
Relations (Z and E/R-diagrams, 3)



corresponds to the relation

$$takes - place == Date \leftrightarrow Room$$

and

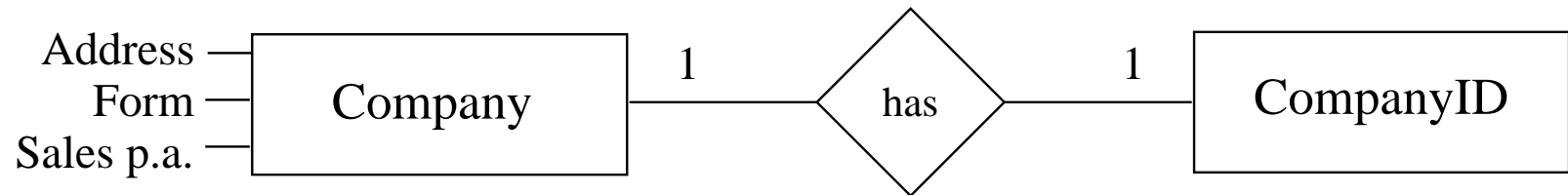


corresponds to the function

$$employs == Employee \rightarrow Company$$

Basis Entities (Z and E/R-diagrams, 5)

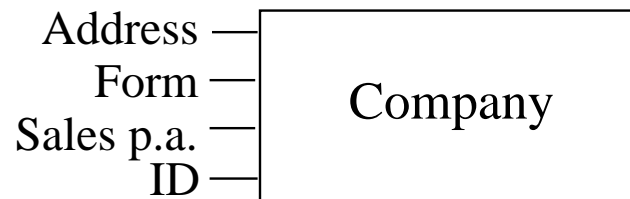
N.B.: When unique identifiers (IDs) are present, then it makes sense to carry out an expansion:



so that there exists a partial injection

$$has == CompanyId \rightsquigarrow Company$$

The above is roughly equivalent to



Summary

- What we have seen:
 - The Z Language and The Mathematical Toolkit.
 - The mathematical language and the schema language.
- What we will see:
 - More about schemata and the schema calculus.
 - Applications.

Latex and E-mail Formats

name	latex	e-mail	math
partial function	<code>\pfun</code>	<code>- -></code>	$_{-} \dashrightarrow _{-}$
total function	<code>\fun</code>	<code>--></code>	$_{-} \rightarrow _{-}$
finite partial function	<code>\ffun</code>	<code>- -></code>	$_{-} \multimap _{-}$
injection	<code>\inj</code>	<code>>--></code>	$_{-} \hookrightarrow _{-}$
finite injection	<code>\finj</code>	<code>>- -></code>	$_{-} \multimap _{-}$
partial injection	<code>\pinj</code>	<code>>- -></code>	$_{-} \hookrightarrow _{-}$
surjection	<code>\surj</code>	<code>-->></code>	$_{-} \twoheadrightarrow _{-}$
partial surjection	<code>\psurj</code>	<code>- ->></code>	$_{-} \twoheadrightarrow _{-}$
bijection	<code>\bij</code>	<code>>-->></code>	$_{-} \xrightarrow{\sim} _{-}$

Latex and E-mail Formats

name	latex	e-mail	math
seq-operator	<code>\seq</code>	<code>\seq</code>	<code>seq</code>
iseq-operator	<code>\iseq</code>	<code>\iseq</code>	<code>iseq</code>
seq brackets	<code>\langle</code> <code>\rangle</code>	<code>%< %></code>	<code><-></code>
nil	<code>\nil</code>	<code>\nil</code>	<code><></code>
concatenation	<code>\cat</code>	<code>^</code>	<code>- ^ -</code>
head	<code>head</code>	<code>head</code>	<i>head</i>
tail	<code>tail</code>	<code>tail</code>	<i>tail</i>
filter	<code>\filter</code>	<code> \</code>	<code>- -</code>
extract	<code>\extract</code>	<code>/ </code>	<code>- -</code>

Formal Modeling with Z: Part IV

Luca Viganò

Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Spring 2002

Schemata

- Z consists of two sub-languages:
 - **mathematical language**, which allows us to model design aspects, objects and their relations.
 - **schema language**, which allows us **structure**, **compose** and **split** modelings (data, functions and predicates).
- Z-schemata
 - can be used as declarations, as types, and as predicates,
 - can model the state space (states and state transitions) of a system,
 - can be used to verify formal modelings.

Overview

- More about schemata and the schema calculus.
 - Schemata as declarations, types and predicates.
 - Operations on schemata (conjunction, inclusion, decoration, etc.).
 - Modeling of states and state transitions.
- Application.
 - *BirthdayBook*, 8-Queens and radiation therapy system.

Syntax of Schemata

Vertical syntax:

name
declaration of typed variables (represent observations of the state)
relationships between values of vars (invariants of the system)

<i>SchemaOne</i>
$a : \mathbb{Z}$ $c : \mathbb{P} \mathbb{Z}$
$a \in c$ $c \neq \emptyset$

Horizontal syntax:

$\text{name} \hat{=} [\text{declarations} \mid \text{predicate}]$

Name and Constraints are optional, e.g.

$a : \mathbb{Z}$ $c : \mathbb{P} \mathbb{Z}$

corresponds to

$a : \mathbb{Z}$ $c : \mathbb{P} \mathbb{Z}$
<i>true</i>

Schemata as Types and Declarations

- To introduce **tagged record types**.

$\begin{array}{l} \text{SchemaTwo} \\ a : \mathbb{Z} \\ c : \mathbb{P}\mathbb{Z} \end{array}$

corresponds to a **composite datatype** with two distinct components: an integer a and a set of integers c .

- We can represent it as a **binding**, e.g.

$\langle a \rightsquigarrow 2, c \rightsquigarrow \{1, 2, 3\} \rangle$ $[a \text{ is bound to } 2 \text{ } c \text{ to } \{1, 2, 3\}]$

- Components are stored **by name** (in Cartesian products: **by position**), and are selected by the operator ' $_{-}._{-}$ '.

When s is an object of schema type *SchemaTwo*, then

- $s.a$ is its integer component,
- $s.c$ is its set component.

Schemata as Types and Declarations (2)

- Bindings express the **semantics** of schemata:

A	
$c_0 : S_0$	
\dots	
$c_n : S_n$	
P	

is equivalent to

$$A \hat{=} \{c_0 : S_0; \dots; c_n : S_n \mid P \bullet \langle c_0 \rightsquigarrow c_0, \dots, c_n \rightsquigarrow c_n \rangle\}$$

- This is a **characteristic binding**: each component is bound to a value with the same name.

Schemata as Types and Declarations (3)

Example:

SchemaThree

$a : \mathbb{Z}$

$c : \mathbb{P}\mathbb{Z}$

$a \in c$

$c \neq \emptyset$

$c \subseteq \{0, 1\}$

describes the set of bindings

$$\begin{aligned} \text{SchemaThree} = \{ & \langle a \rightsquigarrow 0, c \rightsquigarrow \{0\} \rangle, \langle a \rightsquigarrow 0, c \rightsquigarrow \{0, 1\} \rangle, \\ & \langle a \rightsquigarrow 1, c \rightsquigarrow \{1\} \rangle, \langle a \rightsquigarrow 1, c \rightsquigarrow \{0, 1\} \rangle \} \end{aligned}$$

that is, the set

$$\{ a : \mathbb{Z}; c : \mathbb{P}\mathbb{Z} \mid (a \in c) \wedge (c \neq \emptyset) \wedge (c \subseteq \{0, 1\}) \bullet \langle a \rightsquigarrow a, c \rightsquigarrow c \rangle \}$$

Equivalence of Schemata

- Two schemata are **equivalent**, when
 - they introduce the same variables, and
 - set on them the same constraints.
- N.B.: constraints can be implicit in the declarations, e.g.

<i>Birthdaybook</i>	
$known : \mathbb{P} NAME; birthday : NAME \leftrightarrow DATE$	
$known = \text{dom } birthday \wedge birthday : NAME \rightarrow DATE$	

is equivalent to (note also: ‘;’ and ‘ \wedge ’ replaced by separation)

<i>Birthdaybook</i>	
$known : \mathbb{P} NAME$	
$birthday : NAME \rightarrow DATE$	
$known = \text{dom } birthday$	

Equivalence of Schemata (2)

Beware:

$$\frac{\dots}{\begin{array}{l} a \Rightarrow b \\ c \vee d \end{array}}$$

is equivalent to

$$\frac{\dots}{(a \Rightarrow b) \wedge (c \vee d)}$$

but

$$\frac{\dots}{\begin{array}{l} \exists y : T \bullet \\ \quad x < y \vee \\ \quad y < x \end{array}}$$

is equivalent to

$$\frac{\dots}{\exists y : T \bullet (x < y) \vee (y < x)}$$

Schemata as Declarations (1)

A schema can be used as a **declaration**.

- For example, in a comprehension or after a quantifier.
- Effect: introduction of the variables of the declaration part, under the constraints of the predicate part.
- Example:

<i>SchemaOne</i>	_____
$a : \mathbb{Z}$	
$c : \mathbb{P}\mathbb{Z}$	

$a \in c$	
$c \neq \emptyset$	

$$\exists \text{SchemaOne} \bullet a = 0 \quad \Leftrightarrow \quad \exists a : \mathbb{Z}, c : \mathbb{P}\mathbb{Z} \mid (a \in c) \wedge (c \neq \emptyset) \bullet a = 0$$

Schemata as Declarations (2)

Another example:

$MONTHS ::= jan \mid feb \mid mar \mid apr \mid may \mid jun \mid jul \mid aug \mid sep \mid oct \mid nov \mid dec$

Date

$month : MONTHS$

$day : 1 \dots 31$

$month \in \{apr, jun, sep, nov\} \Rightarrow day \leq 30$

$month = feb \Rightarrow day \leq 29$

Then, for quantifier Q and predicate p :

$$Q \text{ Date} \bullet p \quad \Leftrightarrow \quad Q \text{ month} : MONTHS, \text{ day} : 1 \dots 31 \mid \\ (month \in \{apr, jun, sep, nov\} \Rightarrow day \leq 30) \wedge \\ (month = feb \Rightarrow day \leq 29) \bullet p$$

so that

$\exists \text{ Date} \bullet (month = feb) \wedge (day = 29)$ is true

$\forall \text{ Date} \bullet day \leq 30$ is false, as there is $\langle month \rightsquigarrow mar, day \rightsquigarrow 31 \rangle$.

Schemata as Predicates (1)

- A schema can be used as a **predicate**, provided that each component has already been declared as a variable of the correct type.
 - Effect: introduction of a constraint equivalent to the predicate information.
 - Declaration part is deleted; only the constraints remain.
 - Example:

$$\begin{array}{c}
 \text{SchemaOne} \\
 \hline
 a : \mathbb{Z} \\
 c : \mathbb{P}\mathbb{Z} \\
 \hline
 a \in c \wedge c \neq \emptyset
 \end{array}
 \quad \text{and} \quad
 \begin{array}{c}
 \text{SchemaThree} \\
 \hline
 a : \mathbb{Z} \\
 c : \mathbb{P}\mathbb{Z} \\
 \hline
 a \in c \wedge c \neq \emptyset \\
 c \subseteq \{0, 1\}
 \end{array}$$

$$\forall a : \mathbb{Z}; c : \mathbb{P}\mathbb{Z} \mid \text{SchemaOne} \bullet \text{SchemaThree}$$

$$\Leftrightarrow$$

$$\forall a : \mathbb{Z}; c : \mathbb{P}\mathbb{Z} \mid (a \in c) \wedge (c \neq \emptyset) \bullet (a \in c) \wedge (c \neq \emptyset) \wedge (c \subseteq \{0, 1\})$$

That is, all $a : \mathbb{Z}$ and $c : \mathbb{P}\mathbb{Z}$ that satisfy *SchemaOne* must also satisfy *SchemaThree*.

Schemata as Predicates (2)

- The declaration part of a schema can contain constraints, so that

$$\begin{array}{c}
 \text{SchemaFour} \\
 \hline
 a : \mathbb{N} \\
 c : \mathbb{P}\mathbb{N} \\
 \hline
 a \in c \wedge c \neq \emptyset
 \end{array}
 \quad \text{is not equivalent to} \quad
 \begin{array}{c}
 \text{SchemaOne} \\
 \hline
 a : \mathbb{Z} \\
 c : \mathbb{P}\mathbb{Z} \\
 \hline
 a \in c \wedge c \neq \emptyset
 \end{array}$$

- To avoid confusion: **normalization**

Reduces the declaration part to a **canonical form**, e.g.

$$\begin{array}{c}
 \text{SchemaFourNormalized} \\
 \hline
 a : \mathbb{Z} \\
 c : \mathbb{P}\mathbb{Z} \\
 \hline
 a \in \mathbb{N} \\
 c \in \mathbb{P}\mathbb{N} \\
 a \in c \wedge c \neq \emptyset
 \end{array}$$

\Rightarrow

now we see that
SchemaFour contains
 more information
 than *SchemaOne*
 ($a \geq 0$ and $b \in c \Rightarrow b \geq 0$)

Schemata Operators and Calculus

- (Logical) Operations on schemata:
 - Renaming, inclusion, decoration, conjunction and disjunction
 - and other ones (normalization, negation, quantification, hiding and piping).
- ⇒ we can (sequentially) structure and compose specifications.

Renaming of Schemata

- **Renaming** of the components of a schema:

$Schema[new/old]$

\implies introduction of new variables under the same structure of declarations and constraint (systematic substitution), e.g.

$$\begin{array}{c}
 \boxed{\begin{array}{l}
 \text{SchemaOne} \text{ ———} \\
 a : \mathbb{Z} \\
 c : \mathbb{P}\mathbb{Z} \\
 \hline
 a \in c \wedge c \neq \emptyset
 \end{array}}
 \implies
 \begin{array}{c}
 \text{SchemaOne}[q/a, s/c] \\
 \text{is equivalent to}
 \end{array}
 \boxed{\begin{array}{l}
 q : \mathbb{Z} \\
 s : \mathbb{P}\mathbb{Z} \\
 \hline
 q \in s \wedge s \neq \emptyset
 \end{array}}
 \end{array}$$

- N.B.: renaming yields a new schema type.
 - $SchemaOne$: bindings of a and c to values in \mathbb{Z} and $\mathbb{P}\mathbb{Z}$.
 - $SchemaOne[q/a, s/c]$: bindings of q and s to values in \mathbb{Z} and $\mathbb{P}\mathbb{Z}$.

Generic Schemata

- It is possible to rename the components, but not to change their type.
- To employ the same structure for different types: **generic schema** with formal parameters.
- Example

$$\frac{\begin{array}{l} \text{SchemaFive}[X] \text{ —} \\ a : X \\ c : \mathbb{P} X \end{array}}{a \in c \wedge c \neq \emptyset}$$

is equivalent to

$$\frac{\begin{array}{l} \text{SchemaOne} \text{ —} \\ a : \mathbb{Z} \\ c : \mathbb{P} \mathbb{Z} \end{array}}{a \in c \wedge c \neq \emptyset}$$

when X is \mathbb{Z} , and to

$$\frac{\begin{array}{l} \text{SchemaFour} \text{ —} \\ a : \mathbb{N} \\ c : \mathbb{P} \mathbb{N} \end{array}}{a \in c \wedge c \neq \emptyset}$$

when X is \mathbb{N}

A Specification of Sorting

Sort a sequence of objects in non-decreasing order.

$$\begin{array}{c}
 \hline
 [X] \text{-----} \\
 \hline
 \text{nondecreasing} : (X \leftrightarrow X) \rightarrow \mathbb{P}(\text{seq } X) \\
 \hline
 \forall R : X \leftrightarrow X; \sigma : \text{seq } X \bullet \\
 \sigma \in \text{nondecreasing}(R) \Leftrightarrow \\
 (\forall i, j : \text{dom } \sigma \mid i < j \bullet (\sigma(i), \sigma(j)) \in R) \\
 \hline
 \end{array}$$

$$\begin{array}{c}
 \text{Sort}[X] \text{-----} \\
 \hline
 \text{in?}, \text{out!} : \text{seq } X \\
 \text{rel?} : X \leftrightarrow X \\
 \hline
 \text{rel?} \in \text{totord } X \\
 \text{out!} \in \text{nondecreasing}[X](\text{rel?}) \\
 \text{items}(\text{out!}) = \text{items}(\text{in?}) \\
 \hline
 \end{array}$$

- $\text{nondecreasing}(R)$ is a set of sequences in non-decreasing order iff $R : X \leftrightarrow X$ is a **total** order, that is, a reflexive, antisymmetric and transitive order on all elements of X (e.g. \leq for the integers).
- $\text{totord } X$ is the set of all total order for the set X .
- items yields the multiset of the elements of a sequence, e.g.

$$\text{items} \langle a, b, a, a \rangle = \llbracket a, b, a, a \rrbracket = \{a \mapsto 3, b \mapsto 1\}$$

Conjunction of Schemata

Conjunction: allows for the independent specification of a system, which can then be combined (**separation of concerns**).

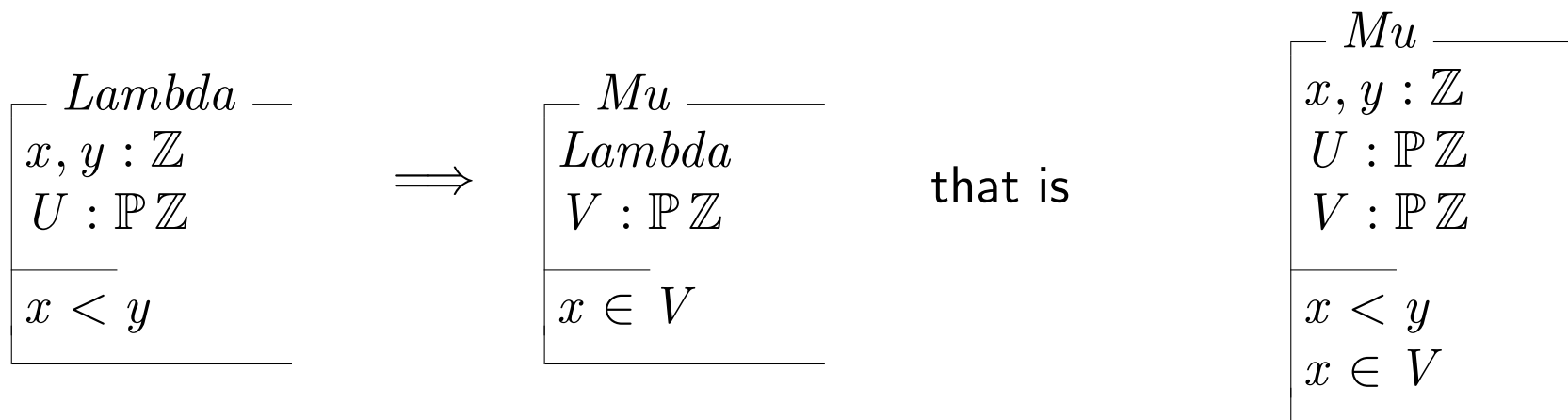
$$\begin{array}{|l} \hline S \\ \hline a : A \\ b : B \\ \hline P \\ \hline \end{array} \quad \text{and} \quad \begin{array}{|l} \hline T \\ \hline b : B \\ c : C \\ \hline Q \\ \hline \end{array}$$

$$\Longrightarrow S \wedge T \hat{=} \begin{array}{|l} \hline a : A \\ b : B \\ c : C \\ \hline P \wedge Q \\ \hline \end{array}$$

N.B.: when the same variable is declared in S and T , then it must be the case that their types coincide (otherwise $S \wedge T$ is undefined).

Inclusion of Schemata

- **Inclusion** (**import**, **inheritance**): has the same effect as conjunction, but it suggest a hierarchical structure.
- Example:



- The initial state of a system is also an inclusion:



Decoration of Schemata (1)

A system-state:

<i>State</i>
$a : A$ $b : B$
P

Each object of a schema-type *State* represents a valid state: a binding of a and b under the predicate P (state-invariant).

An **operation** on the state is described by two copies of *State*, one before and one after the operation (with $'$).

<i>State'</i>		<i>Operation</i>		<i>Operation</i>
$a' : A$ $b' : B$	so that	<i>State</i>		$a : A; b : B$
	we can	<i>State'</i>	that is	$a' : A; b' : B$
$P[a'/a, b'/b]$	write

Decoration of Schemata (2)

An operation that **modifies** the state (structured inclusion Δ):

$\begin{array}{l} \text{AddBirthday} \\ \hline \Delta \text{BirthdayBook} \\ \\ \text{name?} : \text{NAME} \\ \text{date?} : \text{DATE} \\ \\ \text{name?} \notin \text{known} \\ \text{birthday}' = \\ \quad \text{birthday} \cup \{\text{name?} \mapsto \text{date?}\} \end{array}$
--

i.e.

$\begin{array}{l} \text{AddBirthday} \\ \hline \text{known}, \text{known}' : \mathbb{P} \text{NAME} \\ \text{birthday}, \text{birthday}' : \text{NAME} \leftrightarrow \text{DATE} \\ \text{name?} : \text{NAME} \\ \text{date?} : \text{DATE} \\ \\ \text{name?} \notin \text{known} \\ \text{birthday}' = \\ \quad \text{birthday} \cup \{\text{name?} \mapsto \text{date?}\} \\ \text{known} = \text{dom } \text{birthday} \\ \text{known}' = \text{dom } \text{birthday}' \end{array}$
--

In general: ΔState means '*State* and *State*'

$\begin{array}{l} \Delta \text{State} \\ \hline \text{State} \\ \text{State}' \\ \\ \dots \end{array}$	models the state before (variables without ') and that after (variables with ')
--	--

Decoration von Schemata (3)

An operation that **does not modify** the state (structured inclusion Ξ):

$$\frac{\text{FindBirthday} \quad \Xi \text{BirthdayBook} \quad \begin{array}{l} \text{name?} : \text{NAME} \\ \text{date!} : \text{DATE} \end{array}}{\begin{array}{l} \text{name?} \in \text{known} \\ \text{date!} = \text{birthday}(\text{name?}) \end{array}}$$

i.e.

$$\frac{\text{FindBirthday} \quad \begin{array}{l} \text{known}, \text{known}' : \mathbb{P} \text{NAME} \\ \text{birthday}, \text{birthday}' : \text{NAME} \leftrightarrow \text{DATE} \\ \text{name?} : \text{NAME} \\ \text{date!} : \text{DATE} \end{array}}{\begin{array}{l} \text{name?} \in \text{known} \\ \text{date!} = \text{birthday}(\text{name?}) \\ \text{known}' = \text{known} \\ \text{birthday}' = \text{birthday} \\ \text{known} = \text{dom } \text{birthday} \\ \text{known}' = \text{dom } \text{birthday}' \end{array}}$$

In general: ΞState means ' State and State' ', where $\text{variable}' = \text{variable}'$

$$\frac{\Xi \text{State} \quad \Delta \text{State}}{\theta \text{State}' = \theta \text{State}}$$

where θState models the characteristic binding of the components of State

Disjunction of Schemata

Disjunction: models alternatives in the behavior of a system.

$$\begin{array}{|l} \hline S \\ \hline a : A \\ b : B \\ \hline P \\ \hline \end{array} \quad \text{and} \quad \begin{array}{|l} \hline T \\ \hline b : B \\ c : C \\ \hline Q \\ \hline \end{array} \quad \Longrightarrow \quad S \vee T \hat{=} \begin{array}{|l} \hline a : A \\ b : B \\ c : C \\ \hline P \vee Q \\ \hline \end{array}$$

N.B.: when the same variable is declared in S and T , then it must be the case that their types coincide (otherwise $S \vee T$ is undefined).

The Complete *BirthdayBook* (1)

- We can now refine the specification of *BirthdayBook*.
- We have not yet formalized what happens when the input of the operation is incorrect:
 - Does the operation ignore the input?
 - Does the *BirthdayBook*-system crash?
- Example:

$\begin{array}{l} \text{AddBirthday} \\ \Delta \text{BirthdayBook} \\ \text{name?} : \text{NAME} \\ \text{date?} : \text{DATE} \end{array}$ <hr style="border: 0; border-top: 1px solid black; margin: 5px 0;"/> $\begin{array}{l} \text{name?} \notin \text{known} \\ \text{birthday}' = \\ \quad \text{birthday} \cup \{\text{name?} \mapsto \text{date?}\} \end{array}$
--

What happens when $\text{name?} \in \text{known}$???

The Complete *BirthdayBook* (2)

- Extend the specification with schemata that model the errors and the reaction to them.
- We extend each system-operation with an additional output *result!*

$$REPORT ::= ok \mid already_known \mid not_known$$

- and add a general *Success*-schema

<i>Success</i>	_____
<i>result! : REPORT</i>	
<i>result! = ok</i>	

The Complete *BirthdayBook* (3)

Refinement of the addition of a new birthday:

- Input correct: execute *AddBirthday* and return *ok*, i.e.

$$AddBirthday \wedge Success$$

- Input incorrect: add a schema that signals the error

$\neg AlreadyKnown$	_____
$\exists BirthdayBook$ $name? : NAME$ $result! : REPORT$	
$name? \in known$ $result! = already_known$	

Robust *AddBirthday*:

$$RAddBirthday \hat{=} (AddBirthday \wedge Success) \vee AlreadyKnown$$

The Complete *BirthdayBook* (4)

Refinement of the search:

FindBirthday
$\exists \text{BirthdayBook}$ $\text{name?} : \text{NAME}$ $\text{date!} : \text{DATE}$
$\text{name?} \in \text{known}$ $\text{date!} = \text{birthday}(\text{name?})$

NotKnown
$\exists \text{BirthdayBook}$ $\text{name?} : \text{NAME}$ $\text{result!} : \text{REPORT}$
$\text{name?} \notin \text{known}$ $\text{result!} = \text{not_known}$

$$R\text{FindBirthday} \hat{=} (\text{FindBirthday} \wedge \text{Success}) \vee \text{NotKnown}$$

Refinement of *Remind*:

$$R\text{Remind} \hat{=} \text{Remind} \wedge \text{Success}$$

The Complete *BirthdayBook* (5)

We could have written only one schema:

$ \begin{array}{l} \text{--- } RAddBirthday \text{ ---} \\ \Delta BirthdayBook \\ name? : NAME \\ date? : DATE \\ result! : REPORT \end{array} $
$ \begin{array}{l} ((name? \notin known) \wedge (birthday' = birthday \cup \{name? \mapsto date?\}) \wedge (result! = ok)) \\ ((name? \in known) \wedge (birthday' = birthday) \wedge (result! = already_known)) \end{array} $

We must then write explicitly that for incorrect input the state is not modified.

Moreover: the **modularity is lost!**

No separation between normal operations (with general *Success*-schema) and error-handling.

In the literature: more complex forms of modularization of Z-specifications.

Negation of Schemata

Negation of a normalized schema:

$$\boxed{\begin{array}{c} S \\ a : A \\ b : B \\ \hline P \end{array}} \quad \begin{array}{c} \implies \\ \neg S \hat{=} \end{array} \quad \boxed{\begin{array}{c} a : A \\ b : B \\ \hline \neg P \end{array}}$$

A non-normalized schema must be first (implicitly) normalized, e.g.

$$\boxed{\begin{array}{c} SchemaFour \\ a : \mathbb{N} \\ c : \mathbb{P}\mathbb{N} \\ \hline a \in c \wedge c \neq \emptyset \end{array}} \quad \begin{array}{c} \implies \\ \neg SchemaFour \hat{=} \end{array} \quad \boxed{\begin{array}{c} a : \mathbb{Z} \\ c : \mathbb{P}\mathbb{Z} \\ \hline \neg(a \in \mathbb{N} \wedge c \in \mathbb{P}\mathbb{N} \wedge a \in c \wedge c \neq \emptyset) \end{array}}$$

that is, $\neg SchemaFour$ is the negation of *SchemaFourNormalized*.

Quantification and Hiding

We can **quantify** over components of a schema, e.g.

$$\frac{S}{\begin{array}{l} a : A \\ b : B \end{array}} \quad \frac{}{P}$$

$$\forall b : B \bullet S \hat{=} \frac{a : A}{\forall b : B \bullet P} \quad \text{and} \quad \exists a : A \bullet S \hat{=} \frac{b : B}{\exists a : A \bullet P}$$

In general: *quantifier declaration* \bullet *Schema*.

Existential quantification of schemata is also called **hiding**: the quantified components are not visible anymore, but the predicate tells us that they still exist.

$$S \setminus (a) \hat{=} \frac{b : B}{\exists a : A \bullet P}$$

Composition of Schemata

- **Composition** $OpOne \circ OpTwo$ models the sequential execution of two operations. (See also **piping** of schemata in the literature.)
- $OpOne$ yields $State''$, the state immediately before the execution of $OpTwo$.

$$OpOne \circ OpTwo = \exists State'' \bullet (\exists State' \bullet [OpOne; State'' \mid \theta State' = \theta State'']) \wedge (\exists State \bullet [OpTwo; State'' \mid \theta State = \theta State''])$$

Example:

$\frac{OpOne \quad a, a' : A \quad b, b' : B}{P}$	and	$\frac{OpTwo \quad a, a' : A \quad b, b' : B}{Q}$
---	-----	---

$$\implies OpOne \circ OpTwo = (OpOne[a''/a', b''/b'] \wedge OpTwo[a''/a, b''/b]) \setminus (a'', b'')$$

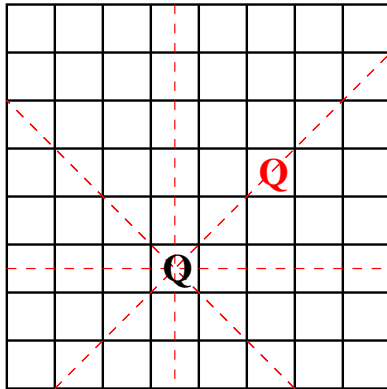
i.e.

$\frac{OpOne \quad a, a' : A \quad b, b' : B}{\exists a'', b'' \bullet P[a''/a', b''/b'] \wedge Q[a''/a, b''/b]}$

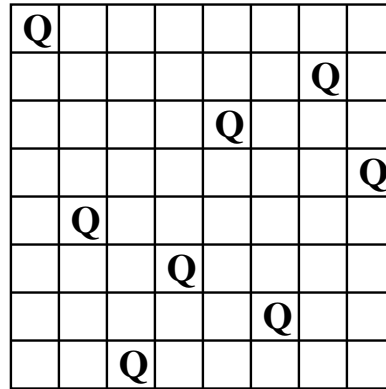
An Example Application: 8-Queens (1)

Problem: position 8 queens on a chess-board so that no queen menaces another (same row, column oder diagonal).

A menace



A solution



Basis-definitions:

$SIZE == 8$

$ROWS == 1..SIZE$

$COLS == 1..SIZE$

$POS == ROWS \times COLS$

The position of the queens

$queens == \mathbb{P} POS$

An Example Application: 8-Queens (2)

- First try: queens cannot be in the same row or column:

$$\begin{array}{|l}
 \text{queens} : \mathbb{P} POS \\
 \hline
 \# \text{queens} = SIZE \\
 \forall q_1, q_2 : \text{queens} \bullet \\
 \quad q_1 \neq q_2 \\
 \quad \Rightarrow \text{first}(q_1) \neq \text{first}(q_2) \wedge \text{second}(q_1) \neq \text{second}(q_2)
 \end{array}$$

That is, a queen q in a row x has an ‘individual’ column.

- This is a bijection:

$$\text{queens} : ROWS \rightarrowtail COLS$$

Each row is ‘assigned’ to one column (and vice versa).

An Example Application: 8-Queens (3)

Diagonals:

- Each square (Pos) lies on 2 diagonals.
- Each diagonal is described by a value on the Y-axis.
- $col = slope \times row + intercept$
- Up-Diag.: $slope = 1$ and $intercept = up$
- Down-Diag.: $slope = -1$ and $intercept = down$

$$\implies up = col - row \text{ and } down = col + row$$

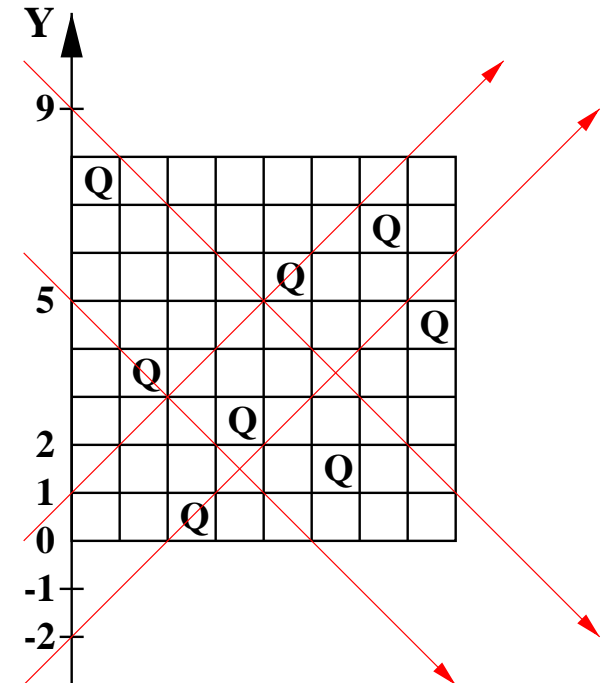
Also:

$$DIAG == (1 - SIZE)..(2 * SIZE)$$

$$up, down : POS \rightarrow DIAG$$

$$\forall x : ROWS, y : COLS \bullet up(x, y) = y - x$$

$$\forall x : ROWS, y : COLS \bullet down(x, y) = x + y$$



An Example Application: 8-Queens (4)

- *up* and *down* are not injective: many squares in a diagonal are assigned the same Y-value.
- But: domain restriction \implies injective

Queens

$$queens : ROWS \twoheadrightarrow COLS$$

$$queens \triangleleft up \in POS \twoheadrightarrow DIAG$$
$$queens \triangleleft down \in POS \twoheadrightarrow DIAG$$

An Example Application: 8-Queens (5)

Summarizing:

$$SIZE == 8$$

$$ROWS == 1..SIZE$$

$$COLS == 1..SIZE$$

$$POS == ROWS \times COLS$$

$$DIAG == (1 - SIZE)..(2 * SIZE)$$

$$up, down : POS \rightarrow DIAG$$

$$\forall x : ROWS, y : COLS \bullet up(x, y) = y - x$$

$$\forall x : ROWS, y : COLS \bullet down(x, y) = x + y$$

Queens

$$queens : ROWS \multimap COLS$$

$$queens \triangleleft up \in POS \multimap DIAG$$

$$queens \triangleleft down \in POS \multimap DIAG$$

The set of the bindings *Queens* contains **all** solutions: model of all facts required for the implementation.

No prescription of a particular implementation strategy or programming language.

An Example Application: Radiation Therapy System (1)

Computer-graphics and algorithmic geometry: points, segments, contours and polygons.

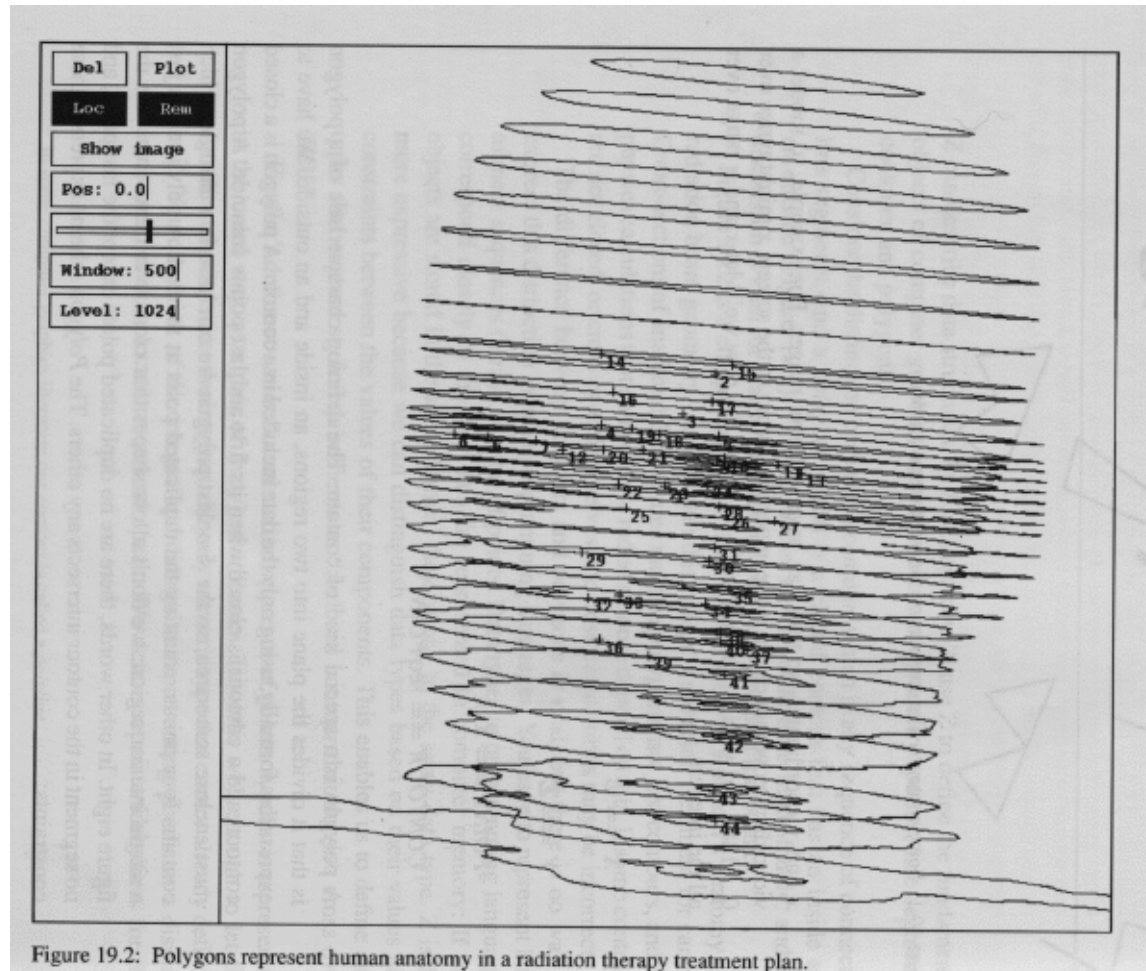


Figure 19.2: Polygons represent human anatomy in a radiation therapy treatment plan.

An Example Application: Radiation Therapy System (2)

Coordinates:

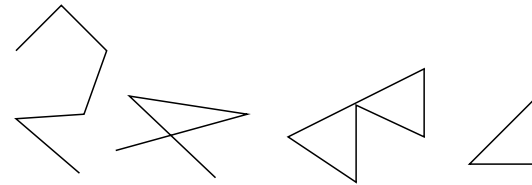
$$X == \mathbb{Z}$$

$$Y == \mathbb{Z}$$

$$POINT == X \times Y$$

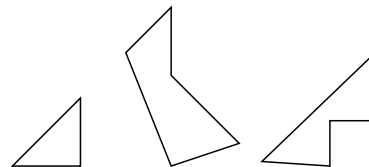
Contour (**polyline**): a sequence of connected points

$$CONTOUR == \text{seq } POINT$$



Calculation of the radiation dosis: the anatomic areas must be particular contours, namely polygons!

Polygon: a closed contour with internal and external zones.



An Example Application: Radiation Therapy System (3)

<i>Polygon</i>	<i>Polygon</i>
$c : \text{CONTOUR}$	A polygon is a contour c
$\#c \geq 4$	c has at least 4 points
$\text{head } c = \text{last } c$	First point is duplicated at end
$\text{front } c \in \text{iseq } \text{POINT}$	No duplicated point except last (iseq)
$\forall s_1, s_2 : \text{segments } c \mid s_1 \neq s_2 \bullet$ $\neg(s_1 \text{ intersects } s_2)$	No segment intersects any others (underline : binary relation as infix)

A **segment** is a pair of points, and *segments* c computes all sequential point-pairs in c , i.e.

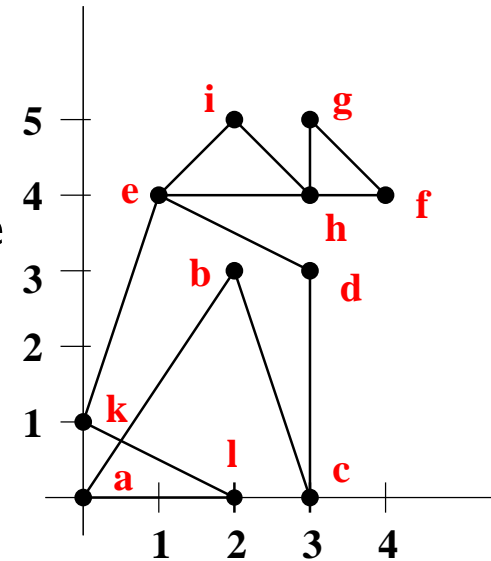
$$\text{segments} == \lambda c : \text{CONTOUR} \bullet \{x, y : \text{POINT} \mid \langle x, y \rangle \in c\}$$

so that

$$\text{segments } \langle x, y, z, x \rangle = \{(x, y), (y, z), (z, x)\}$$

An Example Application: Radiation Therapy System (4)

- Contour $\langle a, b, c, d, e, f, g, i, e, k, l, a \rangle$:
 - Closed, as a is duplicated.
 - No polygon, as e is duplicated and some segments **intersect**:
 - * Segments (a, b) and (k, l) : *cross*,
 - * Segments (e, f) and (g, h) : *touch*.
- Definition of *intersects*:



$$SEGMENT = POINT \times POINT$$

$$intersects, crosses, touches : SEGMENT \leftrightarrow SEGMENT$$

$$\forall s_1, s_2 : SEGMENTS \bullet$$

$$s_1 \text{ intersects } s_2 \Leftrightarrow$$

$$(s_1 \text{ crosses } s_2 \wedge s_2 \text{ crosses } s_1) \vee s_1 \text{ touches } s_2 \vee s_2 \text{ touches } s_1$$

An Example Application: Radiation Therapy System (5)

- Definition of *crosses* and *touches*:
 - Intersection point must not necessarily be an integer.
 - Algorithmic geometry: we don't need to compute the intersection point, in order to say if the segments intersect.
 - We compute the (double) area of a triangle from the coordinates of its angle-points:

$$x == \text{first}[X, Y]; \quad y == \text{second}[X, Y]$$

$$\text{area2} == \lambda a, b, c : \text{POINT} \bullet$$

$$(x\ a * y\ b - y\ a * x\ b) + (y\ a * x\ c - x\ a * y\ c) + (x\ b * y\ c - x\ c * y\ b)$$

- Then: $(c, d) \text{ crosses } (a, b)$ iff $\text{area2}(a, b, c)$ and $\text{area2}(a, b, d)$ have opposite sign, i.e.

$$\text{area2}(a, b, c) * \text{area2}(a, b, d) < 0$$

An Example Application: Radiation Therapy System (6)

$X == \mathbb{Z}; \quad Y == \mathbb{Z}$

$POINT == X \times Y$

$CONTOUR == \text{seq } POINT$

$SEGMENT == POINT \times POINT$

$x == \text{first}[X, Y]; \quad y == \text{second}[X, Y]$

$\text{area2} == \lambda a, b, c : POINT \bullet$

$(x a * y b - y a * x b) + (y a * x c - x a * y c) + (x b * y c - x c * y b)$

$\text{segments} == (\lambda c : CONTOUR \bullet \{x, y : POINT \mid \langle x, y \rangle \in c\})$

$(\text{between_}) == \{i, j, k : \mathbb{Z} \mid i < j < k \vee i > j > k\}$

$(\text{collinear_}) == \{a, b, c : POINT \mid \text{area2}(a, b, c) = 0\}$

An Example Application: Radiation Therapy System (7)

$on : POINT \leftrightarrow SEGMENT$

$touches, crosses, intersects : SEGMENT \leftrightarrow SEGMENT$

$\forall s_1, s_2 : SEGMENT; \quad a, b, c, d : POINT \mid s_1 = (a, b) \wedge s_2 = (c, d) \bullet$
 $(a \text{ on } s_2 \Leftrightarrow collinear(c, a, d) \wedge$
 $((x\ c \neq x\ d \wedge between(x\ c, x\ a, x\ d)) \vee$
 $(y\ c \neq y\ d \wedge between(y\ c, y\ a, y\ d)))) \wedge$
 $(s_1 \text{ touches } s_2 \Leftrightarrow a \text{ on } s_2 \vee b \text{ on } s_2) \wedge$
 $(s_2 \text{ crosses } s_1 \Leftrightarrow area2(a, b, c) * area2(a, b, d) < 0) \wedge$
 $(s_1 \text{ intersects } s_2 \Leftrightarrow$
 $(s_1 \text{ crosses } s_2 \wedge s_2 \text{ crosses } s_1) \vee s_1 \text{ touches } s_2 \vee s_2 \text{ touches } s_1)$

Polygon

$c : CONTOUR$

$\#c \geq 4$

$head\ c = last\ c$

$front\ x \in iseq\ POINT$

$\forall s_1, s_2 : segments\ c \mid s_1 \neq s_2 \bullet \neg(s_1 \text{ intersects } s_2)$

Summary

- What have we seen:
 - The Z Language and The Mathematical Toolkit.
 - The mathematical language, the schema language and the schema calculus.
 - Some applications.
- What will we see: a number of things, including
 - ‘from Z -specifications to programs’.

Language Support For Development In The Large

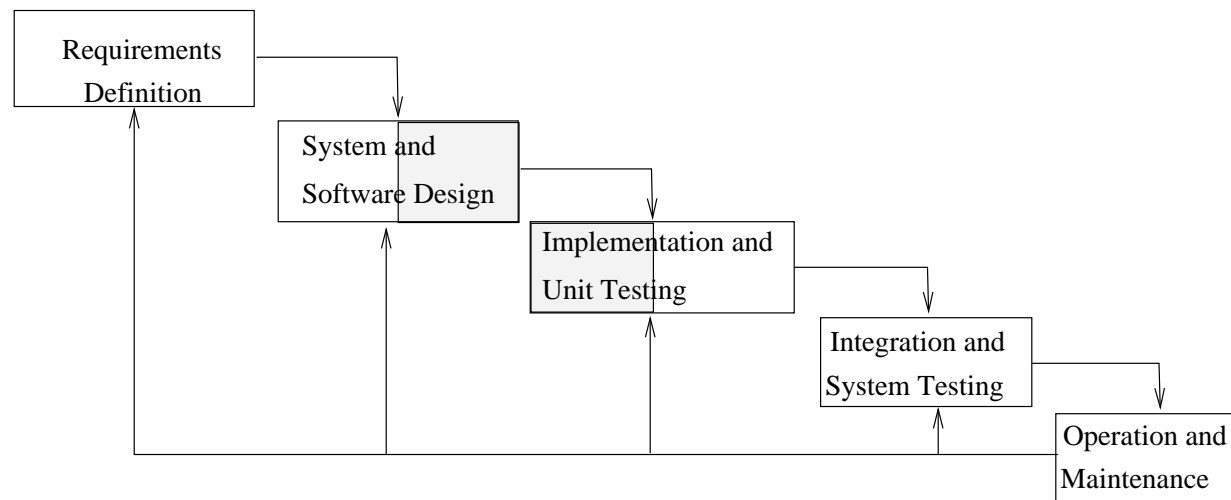
David Basin

Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Spring 2002

Overview: coming weeks

- Concepts for structuring large systems and supporting reusability
- Important in the **modeling** and **implementation** phases



But what does this have to do with software engineering?

- Although high-level design is independent of language/technology ...
- Low-level design and implementation are language/technology dependent!

In the (advanced) development phases one must utilize language and platform-specific features.

- Concretely, we examine both the functional and object oriented paradigms ... and their realization in Haskell, SML, and Java.
- Later even higher-level structuring concepts like middleware.

Software engineering and programming languages are close cousins!

Structuring: historical context

- Programs manipulate, uncontrolled, the data or even the code of other programs. (1950)

This serious violation of the principle of modular construction will result in the immediate revocation of your degree!

- Communication over global variables (C, FORTRAN). (1960)

This violation of encapsulation (information hiding) is only sensible in a few exceptional circumstances.

- Exchange of control information, for example through global flags. (1960)

Also dangerous.

- Data exchange over procedure/method parameters. (1960-1970)

Standard practice.

- Explicitly defined (e.g., Export/Import) interfaces. (1980)

Standard practice. Important for preventing the uncontrolled use of procedures and methods!

Part I: Structuring in Haskell

- Haskell is a strongly typed, higher-order, functional programming language. Supports advanced concepts but easily learned (e.g., in Informatik I).
- Keywords
 - Polymorphic:** Reusability on different kinds of data
 - Higher-order:** Functions take/yield functions
 - Classes:** Overloading of operators and hierarchical structuring
 - Modules:** Control name-space. Used for abstract data types.

Reusability through polymorphism

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (h:t) = (f h):(map f t)
```

```
Main> map (\x -> (x,x)) [1,2,3]
[(1,1),(2,2),(3,3)]
```

```
Main> map (\x -> (x,x)) ["double", "me"]
[("double","double"),("me","me")]
```

```
Main> map (+1) [1,2,3]
[2,3,4]
```

```
Main> map (+1) ["double", "me"]
ERROR: [Char] is not an instance of class "Num"
```

N.B.: parameterized polymorphism

Same implementation for all type instances (kinds of data).

Polymorphism + higher-order functions

```
isort :: [Int] -> [Int]
isort []      = []
isort (x:xs) = ins x (isort xs)
  where ins x [] = [x]
        ins x (y:ys) = if x < y then x:y:ys else y:ins x ys
```

```
Main> isort [3,1,2,1,5]
[1,1,2,3,5]
```

```
Main> isort [3.0, 1.0, 5.2, 3.5]
ERROR: ...
```

Question: How can we sort arbitrary lists?

Answer #1: type classes

- Generalize the type to `isort :: Ord a => [a] -> [a]`
- Recall: `Ord` is a class of types with the ordering

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y      = not (x==y)
```

```
class (Eq a) => Ord a where
  compare          :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min         :: a -> a -> a
```

- Lists of type `[t]` can now be sorted, when $t \in \text{Ord}$

```
Main> isort [1,2,3]
[1,2,3]
Main> isort ["another", "sorting", "example"]
["another","example","sorting"]
```

- Ordering `<` is type indexed.

Answer #2: polymorphism + higher-order functions

```
isort :: (a -> a -> Bool) -> [a] -> [a]
isort rel [] = []
isort rel (x:xs) = ins x (isort rel xs)
  where ins x [] = [x]
        ins x (y:ys) = if x 'rel' y then x:y:ys else y:ins x ys
```

```
Main> isort (<) [3, 1, 5, 3]
[1,3,3,5]
```

```
Main> isort \(a,b) (c,d) -> if a + b < c + d then True else False)
      [(4,2),(1,1), (5,3), (2,2), (1,2)]
[(1,1),(1,2),(2,2),(4,2),(5,3)]
```

```
Main> isort (<) ["another","sorting","example"]
["another","example","sorting"]
```

Example (cont.)

```
lexExtend :: (a -> a -> Bool) -> [a] -> [a] -> Bool
lexExtend rel [] _ = True
lexExtend rel (x:_) [] = False
lexExtend rel (x:xs) (y:ys) =
    if      x 'rel' y then True
    else if y 'rel' x then False
    else    lexExtend rel xs ys
```

```
strOrd :: [Char] -> [Char] -> Bool
strOrd x y = lexExtend (<) (map ord x) (map ord y)
```

```
Main> isort strOrd ["here", "is", "an", "interesting", "example", "to", "sort"]
["an","example","here","interesting","is","sort","to"]
```

Polymorphism + higher-order (cont.)

- How can one sort lists in the reverse order?

```
swapOrd :: (a -> b -> c) -> b -> a -> c
swapOrd rel x y = rel y x
```

```
Main> isort (swapOrd strOrd) ["here", "is", "an", "example", "to", "sort"]
["to","sort","is","here","example","an"]
```

- How can one sort strings backwards?

```
rev :: [a] -> [a]
rev [] = []
rev (x:xs) = rev(xs) ++ [x]
```

```
revOrd :: [Char] -> [Char] -> Bool
revOrd x y = lexExtend (<) (listify x) (listify y)
  where listify s = rev (map ord s)
```

```
Main> isort revOrd ["here", "is", "an", "interesting", "example", "to", "sort"]
["example","here","interesting","an","to","is","sort"]
```

Reusability

- Higher-order functions offer high reusability
- Strong generalization of Unix 'pipe' Idea

```
cat /usr/dict/words | rev | sort | rev | grep "^a" | pr -t -3 | more
```

a	atrophic	anastomotic
amoeba	anamorphic	asymptotic
armada	automorphic	aseptic
addenda	anorthic	apocalyptic
agenda	acyclic	aspartic
anaconda	angelic	antagonistic
althea	alcoholic	anachronistic
azalea	apostolic	autistic
area	acrylic	atavistic
alfalfa	aerodynamic	agnostic
alga	academic	acoustic
aloha	algorithmic	attic
alpha	astronomic	aeronautic
...

Find rhymes (for amature poet): e.g., "'apostolic'" and "'alcoholic'"

Structuring using modules

- Functions abstract computation steps
Analogous to procedures in imperative programming languages.
- **Modules** are the next abstraction level.
 - Modules define an **interface**.
 - In Haskell, modules define collections of values, data types, type definitions, classes, etc. in an environment.
- Haskell modules are very simple
 - Basically control/organize **name spaces**.
 - Can be used for **abstract data types**.

An example

- Default: all data types, functions, ... are exported

```
module Tree where
```

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

```
fringe :: Tree a -> [a]
```

```
fringe (Leaf x) = [x]
```

```
fringe (Node l r) = fringe l ++ fringe r
```

```
...
```

- Module can be imported into other modules

```
module foo where
```

```
import Tree
```

```
... fringe ( ... Leaf ... ) ...
```

All names/bindings are imported

Module — name space control

- One can control imported/exported names.
 - Data types, constructors, and n functions are exported

```
module Tree ( Tree(Leaf,Node), f1, ..., fn )
```

- or without constructors

```
module Tree(Tree, f1, ..., fn)
```

In 2nd case we have an error in:

```
module Goo where
```

```
import Tree  
g = Leaf
```

- We can similarly restrict the name space when importing modules.

ADTS — an application of modules

- A **data type** is one (or more) set(s) of data and related functions. A **signature** describes the types of the functions.

```
data Tree a      -- just the type name
leaf             :: a -> Tree a
node h          :: Tree a -> Tree a -> Tree a
cell            :: Tree a -> a
left,right       :: Tree a -> Tree a
isLeaf          :: Tree a -> Bool
```

- An **abstract data type** is a data type, whose implementation is **encapsulated**. One has access to data **only** through functions declared in the signature.
- Modules are a possible way of implementing ADTs.

Example

- Tree ADT:

```
module TreeADT (Tree, leaf, node, cell, left, right, isLeaf) where

data Tree a = Leaf a | Node (Tree a) (Tree a)
leaf = Leaf
node = Node
cell (Leaf a) = a
left (Node l r) = l
right (Node l r) = r
isLeaf (Leaf _) = True
isLeaf _ = False
```

- Leaf and Node are not exported

⇒ One cannot write a function using pattern matching to decompose trees.

- But you can analyze and decompose data in a representation independent way. And also build data (how?).
- Encapsulation means that changes are always local.

Part II: Structuring in SML

- SML is a strongly typed, higher-order functional programming language.
 - Developed at AT&T/Bell Labs for programming **in the large**.
 - Main difference to Haskell: **eager**, **reference types**, and a very **advanced module system**.
 - Used to solve many real world problems!
- Modules in SML
 - Support encapsulation
 - Separate interface specification (**signature**) from implementation (**structure** or **functor**)
 - Parameterized (polymorphic types)
 - **Functors** support the composition of data types
 - Separate compilation possible

Structures: composite environments

- 2 examples for Booleans

```
structure A1 =  
struct  
  type Bool = bool  
  val True = true  
  val False = false  
  fun Not x = not x  
  fun And x y = x andalso y  
end;
```

```
structure A2 =  
struct  
  type Bool = int  
  val True = 1  
  val False = 0  
  fun And (x:int) y = x * y  
end;
```

Similar to **struct** in C, but one can also specify types and functions.

- Example application

```
- A1.True;  
val it = true : bool  
  
- A2.And (A2.False A2.True);  
val it = 0 : int
```

Signature: specification of interfaces

- Signature for BOOL

```
- signature BOOL =  
  sig  
    type Bool  
    val True : Bool  
    val False : Bool  
    val Not : Bool -> Bool  
    val And : Bool -> Bool -> Bool  
  end;
```

- “Signature Matching” is analogous to type checking.
- The required functions (types, ...) must be defined in the structure in a type-correct way.

```
- structure S1 : BOOL = A1;  
structure S1 : BOOL
```

- Otherwise we have an error

```
- structure S2 : BOOL = A2;  
stdIn:40.1-40.25 Error: unmatched value specification: Not
```

Example: queues

- The signature

```
signature QUEUE =  
sig  
  type 'a queue  
  exception E  
  val empty: 'a queue  
  val enq: 'a queue * 'a -> 'a queue  
  val null : 'a queue -> bool  
  val hd : 'a queue -> 'a  
  val deq : 'a queue -> 'a queue  
end;
```

- N.B. Signature uses parametric polymorphism

Also note exception for error handling (like in Java)

- **Questions:**

- What is the difference between a signature for queues and stacks?
- What is a simple implementation? Efficiency?

A simple implementation

```
structure queue1:QUEUE =  
struct  
  type 'a queue = 'a list  
  exception E  
  
  val empty = []  
  fun enq (q,x) = q@[x];  
  
  fun null (x::q) = false  
    | null _ = true  
  
  fun hd (x::q) = x  
    | hd [] = raise E  
  
  fun deq(x::q) = q  
    | deq [] = raise E  
end;  
  
- queue1.deq (queue1.enq(queue1.enq (queue1.empty, 3),7));  
val it = [7] : int queue1.queue  
  
- stack.deq (stack.enq(stack.enq (stack.empty, 3),7));  
val it = [3] : int stack.queue
```


Example: efficient queues

- Pairs of lists

$$([x_1, x_2, \dots, x_m], [y_1, y_2, \dots, y_n])$$

implement queues

$$x_1, x_2, \dots, x_m, y_n, \dots, y_2, y_1$$

- Supports $O(1)$ enqueue $enq(q, y)$

$$([x_1, x_2, \dots, x_m], [y, y_1, y_2, \dots, y_n])$$

- and $O(1)$ dequeue $deq(q)$

$$([x_2, \dots, x_m], [y_1, y_2, \dots, y_n])$$

- Only trick: When first list is empty, the second is reversed and replaces the first

$$([], [y_1, y_2, \dots, y_n]) \hookrightarrow ([y_n, \dots, y_2, y_1], [])$$

- How efficient is such an implementation?

Efficient queues

```

structure queue2:QUEUE =
  struct
    datatype 'a queue = Queue of ('a list * 'a list);
    exception E;

    val empty = Queue([], []);

    (*Normalized queue, if nonempty, has nonempty heads list*)
    fun norm (Queue([], tails)) = Queue(rev tails, [])
      | norm q = q;

    (*norm has an effect if input queue is empty*)
    fun enq(Queue(heads, tails), x) = norm(Queue(heads, x::tails))
    fun null(Queue([], [])) = true | null _ = false;
    fun hd(Queue(x::_, _)) = x | hd(Queue([], _)) = raise E;

    (*normalize in case heads become empty*)
    fun deq(Queue(x::heads, tails)) = norm(Queue(heads, tails))
      | deq(Queue([], _)) = raise E;
  end;

- queue2.deq (queue2.enq(queue2.enq (queue2.empty, 3), 7));
val it = Queue ([7], []) : int queue2.queue

- queue2.hd (queue2.deq (queue2.enq(queue2.enq (queue2.empty, 3), 7)));
val it = 7 : int

```

Signature + Structure \neq Abstraction

- An **abstract object** is only known through its exported (interface) functions. Realization and internal form are not externally decernable.
- Consider queue1:
 - `queue1.enq ([], 3);`
`val it = [3] : int queue1.queue`
 - `queue1.enq ([], 3) = [3];`
`val it = true : bool`
 - `queue1.deq ["hi", "there", "people"];`
`val it = ["there","people"] : string queue1.queue`
- Data type, but not an abstract data type!
 - Well-defined interface and separate realization ...
 - But no abstraction

'Opaque' signature constraints

- Queue1 again (without constraints)

```
structure queue1 =  
struct  
  type 'a queue = 'a list  
  exception E  
  
  val empty = []  
  fun enq (q,x) = q@[x];  
  fun null (x::q) = false | null _ = true  
  fun hd (x::q) = x | hd [] = raise E  
  fun deq(x::q) = q | deq [] = raise E  
end;
```

- 'Transparent' Constraints

```
structure q : QUEUE = queue1;
```

- 'Opaque' Constraints

```
structure qa :> QUEUE = queue1;
```

Transparent versus opaque constraints

- Transparent

```
- q.enq(q.enq(q.empty,2),1);
val it = [2,1] : int queue.queue
```

```
- q.deq [1,2];
val it = [2] : int queue.queue
```

Well-defined interface, but no encapsulation

- Opaque

```
qa.enq(qa.enq(qa.empty,2),1);
val it = - : int qa.queue
```

```
- qa.hd it;
val it = 2 : int
```

```
- qa.deq [1,2];
stdIn:320.1-320.13 Error: operator and operand don't agree [tycon mismatch]
  operator domain: 'Z qa.queue
  operand:         int list
  in expression:   qa.deq (1 :: 2 :: nil)
```

Generic, abstract data type + encapsulation (information hiding)

Parameterization

- Problem: How does one write new functions that manipulate queues?
- Example: Queue to/from Lists

```
fun foldl f e [] = e
  | foldl f e (h::t) = foldl f (f(h,e)) t
  (* f(xn, ..., f(x_1, e) ...) *)
```

```
fun fromlist l = foldl (fn (x,q) => Q.enq(q,x)) Q.empty l
```

```
fun tolist q = if Q.null q then []
  else Q.hd q :: tolist (Q.deq q)
```

Functions only definable when Q is already bound, i.e., for a given structure.

- Direct abstraction over structures is not possible.

```
fun fromlist l Q = foldl (fn (x,q) => Q.enq(q,x)) Q.empty l
```

```
std_in:0.0-0.0 Error: unbound structure: Q in path Q.empty
```

Structure parameterization (cont.)

Functors: Structures that are parameterized over other structures.

```
- functor TestQueue (Q:QUEUE) =
  struct
    fun foldl f e [] = e                (* f(xn, ..., f(x_1, e) ...) *)
      | foldl f e (h::t) = foldl f (f(h,e)) t

    fun fromlist l = foldl (fn (x,q) => Q.enq(q,x)) Q.empty l

    fun tolist q = if Q.null q then []
                  else Q.hd q :: tolist (Q.deq q) end;

- structure T2 = TestQueue (qa);
structure T2 :
  sig
    val foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
    val fromlist : 'a list -> 'a Q.queue
    val tolist : 'a Q.queue -> 'a list
  end

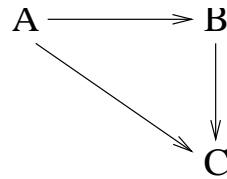
- qa.hd (T2.fromlist [1,2,3,4,5]);
val it = 1 : int

- T2.tolist (T2.fromlist [1,2,3,4,5]);
val it = [1,2,3,4,5] : int list
```

Modules — the idea

- **Module**: a structure or a parameterized structure.
- Signature: fixes the external interface.
- Analogy to functions/types
 - Just as functions can be parameterized by other functions, structures can be parameterized by other structures.
 - Type checking tests composability of functions/data. Signature checking insures that the proper interface is provided.
- Idea at the module level is considerably more powerful!
- We will consider two examples: one artificial, one realistic.

Example #1: A System of 3 Modules



- Dependency: B requires A and C requires A and B

- Simplest structuring:

```
structure A:SIGA = struct ... A-stuff... end;  
structure B:SIGB = struct ... A.goo ... end;  
structure C:SIGC = struct ... A.goo ... B.foo ... end;
```

- Although signatures provide a well-defined interface,
 - dependencies are hidden, and
 - the implementation is fixed prematurely.

Better solution based on functors

```
functor mkA():SIGA = struct ... A-stuff... end;  
  
functor mkB(A:SIGA):SIGB = struct ... A.goo ... end;  
  
functor mkC(structure A:SIGA structure B:SIGB):SIGC =  
    struct ... A.goo ... B.foo ... end;
```

N.B.:

- 0ary functors show independence.
- n -ary ($n > 1$) functors require the keyword 'structure' before each parameter

Example — concretized

```
signature SIGA =  
sig  
  type t  
  val mk: int -> t  
  val p:t*t -> t  
end;
```

```
signature SIGB =  
sig  
  type b  
  type t  
  
  val b0:b  
  val f: b -> t  
end;
```

```
signature SIGC =  
sig  
  type t  
  val test:t  
end;
```

Example (cont.)

```

functor mkA(): SIGA =
struct
  type t = string
  fun mk(i:int) = str(chr((i + ord #"a") mod 128))
  fun p(n,m) = n ^ m                                end;

functor mkB(A:SIGA): SIGB =
struct
  type b = string
  type t = A.t
  val b0 = "abc"
  fun f(s) = A.mk(size s)                            end;

functor mkC(
  structure A:SIGA
  structure B:SIGB):SIGC =
struct
  type t = A.t
  val test = A.p(A.mk 1, A.p(A.mk 4, B.f(B.b0))) end;

```

SML answers:

```

stdIn:161.17-161.49 Error: operator and operand don't agree [tycon mismatch]
  operator domain: t * t      operand:          t * ?.t
  in expression:   A.p (A.mk 4,B.f B.b0)

```

Why?

Example (with sharing constraints)

```
functor mkC(  
  structure A:SIGA  
  structure B:SIGB  
  sharing type A.t = B.t):SIGC =  
struct  
  type t = A.t  
  val test = A.p(A.mk 1, A.p(A.mk 4, B.f(B.b0)))  
end;
```

```
structure A = mkA();  
structure B = mkB(A);  
structure C = mkC(structure A = A  
                  structure B = B);
```

example:

```
- A.mk 1;  
val it = "b" : t  
  
- C.test;  
val it = "bed" : C.t
```

Example #2: generic arithmetic

- Arithmetic with zero/sum/prod

```
signature ZSP =  
sig  
  type t  
  val zero : t  
  val sum   : t * t -> t  
  val prod  : t * t -> t  
end;
```

- Typical structure

```
structure IntZSP:ZSP =  
struct  
  type t = int;  
  val zero = 0;  
  fun sum   (x,y) = x+y: t;  
  fun prod  (x,y) = x*y: t;  
end;
```

Matrices: a completely different structure

- $n \times m$ Matrix

$$\begin{pmatrix} n_{1,1} & n_{1,2} & \dots & n_{1,m} \\ n_{2,1} & n_{2,2} & \dots & n_{2,m} \\ \vdots & \vdots & & \vdots \\ n_{n,1} & n_{n,2} & \dots & n_{n,m} \end{pmatrix}$$

is represented as a list of lists

$$[[n_{1,1}, n_{1,2}, \dots, n_{1,m}], [n_{2,1}, n_{2,2}, \dots, n_{2,m}], \dots [n_{n,1}, n_{n,2}, \dots, n_{n,m}]]$$

- How do we build matrices with a ZSP interface?

- Problem 1: matrices over what kind of elements?

Solution: parameterize matrices over ZSP elements

- Problem 2: representation of 0?

Solution: instead of an $m \times n$ matrix zero-matrix, we define a single 0 element, where $0 + A = A + 0 = A$ and $0 \times A = A \times 0 = 0$.

Matrices as functors

```
functor MatrixZSP (Z: ZSP) : ZSP =
struct
  type t = Z.t list list;

  fun pairmap f ([],[]) = []
    | pairmap f ((h::t),(h'::t')) = (f (h,h')) :: (pairmap f (t,t'));

  val zero = [];

  fun sum (rowsA,[]) = rowsA
    | sum ([],rowsB) = rowsB
    | sum (rowsA,rowsB) = pairmap (pairmap Z.sum) (rowsA,rowsB);

  fun dotprod pairs = foldl Z.sum Z.zero (pairmap Z.prod pairs);

  fun transp ([]::_ ) = []
    | transp rows = map hd rows :: transp (map tl rows);

  fun prod (rowsA,[]) = []
    | prod (rowsA,rowsB) =
      let val colsB = transp rowsB
      in map (fn row => map (fn col => dotprod(row,col)) colsB) rowsA
      end;
end;
```


Matrices as functors (cont.)

```
- structure IntMatrix = MatrixZSP (IntZSP);  
  
- IntMatrix.sum ([[1,2],[3,4]], [[5,6],[7,8]]);  
val it = [[6,8],[10,12]] : IntMatrix.t  
  
- IntMatrix.sum ([[1,2],[3,4]], IntMatrix.zero);  
val it = [[1,2],[3,4]] : IntMatrix.t  
  
- IntMatrix.prod ([[1,2],[3,4]], [[0,1],[1,0]]);  
val it = [[2,1],[4,3]] : IntMatrix.t
```

Graph application (reusability)

- Some graph algorithms operate over binary matrices

Adjacency matrix (i, j) means there is an edge from node i to node j .

- We interpret zero as false, sum as disjunction and product as conjunction.

$$0 + x = x + 0 = x \quad x \times 0 = 0 \times x = 0$$

```
structure BoolZSP:ZSP =
struct
  type t = bool;
  val zero = false;
  fun sum    (x,y) = x orelse y;
  fun prod   (x,y) = x andalso y;          end;

- BoolZSP.sum(BoolZSP.zero,true);
val it = true : BoolZSP.t

- structure BoolMatrix = MatrixZSP (BoolZSP);

- BoolMatrix.sum(BoolMatrix.zero, [[true,false],[false,true]]);
val it = [[true,false],[false,true]] : BoolMatrix.t
```

Graph connectivity

- $G \times G$ iff there is a path from i to j of length 2.
- There is a path from i to j iff $(i, j) = \text{true}$ in $G + G \times G + G \times G \times G + \dots$

```
- fun closure t f un = (* compute t un f(t) un f(f(t)) un ... *)
  let fun closure_with ft res =
    let val nft = f ft
        val nres = un (res, nft) in
      if res = nres then res else closure_with nft nres      end
  in closure_with t t end;
```

```
val closure = fn : ''a -> (''a -> ''a) -> (''a * ''a -> ''a) -> ''a
```

```
fun graph_closure g = closure g (fn x => BoolMatrix.prod(x, g)) BoolMatrix.sum;
```

```
- graph_closure [[false,true,false,false],
                  [false,false,true,true],
                  [false,true,false,false],
                  [false,false,false,false]];

val it =
  [[false,true,true,true],
   [false,true,true,true],
   [false,true,true,true],
   [false,false,false,false]] : BoolMatrix.t
```

Graph example: conclusion

- Same functor works for, e.g., real and complex numbers
- Functor application can be iterated

```
- structure IntMatMat = MatrixZSP (MatrixZSP (IntZSP));  
structure IntMatMat : ZSP  
  
- IntMatMat.sum ([ [[1]], [[2]] ], [ [[5]], [[6]]  ]);  
val it = [[6]], [[8]] : IntMatMat.t
```

- Functors support a signature-correct composition of structures
⇒ Functors enable the correct composition of systems from subsystems
- **Question:** What does **correct** mean? How powerful is this notion?

Part III: Structuring in OO-languages

- Object orientation is both a **world view** as well as a particular **language feature**, that supports construction in the large, in particular encapsulation and reusability.
- World view: the world consists of agents that offer services.
Example: If you want to send flowers to your mother, go to a florist and order them. How the florist carries this out is his business.
- Technically, OO-languages offer different structuring mechanisms.
 - Classes, interfaces, inheritance, polymorphism, ...
 - Support both classic ADTs and OO-development

Let's start with an example

- Queues as a Java Class

```
import java.util.LinkedList; // Implements List interface (extends collection)

class Queue {
    private LinkedList list;
    public Queue()           { list = new LinkedList(); }
    public void enq(Object x) { list.addLast(x); }
    public Object deq()       { return list.removeFirst(); }
    public Object hd()        { return list.getFirst(); }
    public boolean isEmpty()  { return list.size() == 0; }
}
```

- Encapsulation and name-space control through:

import: specifies the class (**LinkedList**) in a package (**java.util**)

public/protected/private: specifies further control

⇒ information hiding for queues.

Classes versus modules

- Classes and modules both support
 - Definition of interfaces
 - Encapsulation
- Non-abstract classes also specify the implementation.
 - Alternative: abstract methods or interfaces
- SML-approach separates signature and implementation
 - Supports creation of multiple instances
- Differences in further structuring capabilities
 - Inheritance versus parameterization through functors.

Interfaces

- Interfaces defines objects can interact with each other
 - When one takes **responsibilities** seriously (informal/Z/...), an interface fixes both syntactic and semantic properties.
 - I.e., an interface defines a **contract**!
- Advantage: decouples client and provider.
 - Any implementation that satisfies the contract is considered correct.
 - Unfortunately Java/C++/... can only statically check quite restricted aspects, e.g., that a class which implements an interface provides all required methods.

Example: Lists (from java.util)

```
public interface Collection {
int size();           // the number of elements in this collection
boolean isEmpty();    // true if collection contains no elements
boolean contains(Object o); // true if collection contains Object o
Iterator iterator();  // returns an iterator over elements in collection
boolean add(Object o); // Adds an element. True if collection changed
boolean remove(Object o); ... }

public interface List extends Collection {
int size();
...
Object get(int index);           // Get element at index in a list
Object set(int index, Object element); // Set element at index in a list
... }
```

Semantic aspects are stated informally as comments.

Example: Interface **Collection** is identical to **Set**, but the contract is different.

The Set interface places additional stipulations, beyond those inherited from the Collection interface, on the contracts of all constructors and on the contracts of the add, equals and hashCode method. ... The additional stipulation on constructors is, not surprisingly, that all constructors must create a set that contains no duplicate elements.

Interfaces, types, and polymorphism

- In the OO-world, a **type** corresponds to a set of objects that implement a particular interface.
 - The types names all operators and input/output parameters
 - This constitutes a **very** weak semantic contract!

- **Polymorphism:** An object of a subtype can always be used in place of an object of a supertype.

Example: A **Linkedlist** can always be used where a **List** is expected.

- **Question:** What is the difference to **parametric polymorphism** in Haskell and SML?

Example: queues (again)

- Multiple classes can implement the same interface.

```
public interface Queue {...}
```

```
class CircularArrayQueue implements Queue  
{ CircularArrayQueue(int capacity) {...}  
  ... }
```

```
class LinkedListQueue implements Queue  
{ LinkedListQueue() {...}  
  ... }
```

- Interface can be used as a type.
 - When objects are always typed by the interface,

```
Queue expressLane = new CircularArrayQueue(1000);  
expressLane.add(new Customer('Harry'));
```

- then later changes are trivial (just change the constructor!)

```
Queue expressLane = new LinkedListQueue();  
expressLane.add(new Customer('Harry'));
```

Inheritance

- Two kinds:
 - Inheritance of implementation:** The declaration of **Subclasses** enables the reuse of implementation.
 - Inheritance of interfaces:** Inherit interfaces and/or contracts.
- Java supports simple inheritance of implementations and multiple inheritance of interfaces.

Example: Inheritance of implementation

```
public abstract class Shape {
    public double area() { return 0.0; }
    public double volume() { return 0.0; }
    public abstract String getName(); }

public class Point extends Shape {
    protected int x, y;

    public Point(int a, int b) { x = a; y = b}
    ...
    public String getName() { return "Point";} }

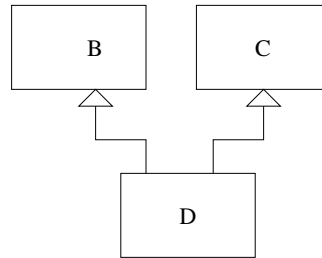
public class Circle extends Point {
    protected double radius;

    public Circle(double r, int a, int b) {
        super(a,b);  radius = (r >= 0 ? r : 0);
    }

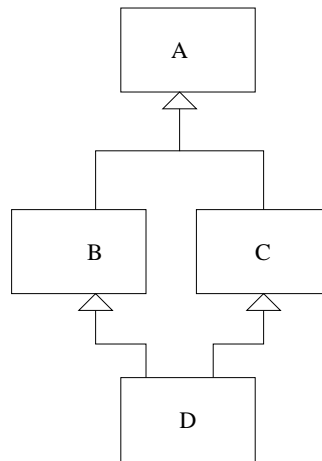
    public double area() { return Math.PI * radius * radius;}
    ...
    public String getName() { return "Circle";} }
```

Multiple inheritance of implementations?

- Simple case (e.g., everything disjoint) is unproblematic



- Non-trivial in general



- What happens in “method overriding” in *B* or *C*?
- Normally *B* and *C* have their own state. What happens in *D*?

Multiple inheritance (cont.)

- Some OO-languages allow multiple inheritance

Complicated precedence rules are used to eliminate ambiguity

- Java forbids multiple inheritance and supports, as an alternative, multiple inheritance of **interfaces**.

Requires that overlapping fields (variables, methods, ...) have identical types.

- Still not completely free of problems

```
interface Fish { int getNumberOfScales(); }  
  
interface Piano { int getNumberOfScales(); }  
  
class Tuna implements Fish, Piano {  
    // You can tune a piano, but can you tuna fish?  
    int getNumberOfScales() { return 91; }  
}
```

Example: sorting

- Without interfaces:

```
abstract class Sortable
{ public abstract int compareTo(Sortable b); }

class ArrayAlg
{ public static void isort(Sortable[] a) { ... } }
```

- Suppose we want to sort employees based on their salary:

```
class Employee extends Sortable
{ ...
  public int compareTo(Sortable b) // compares (this) employee with employee b
  { Employee eb = (Employee) b;
    if (salary < eb.salary) return -1;
    if (salary > eb.salary) return 1;
    return 0; } ...
}
```

- Works, but only sometimes!

Example (cont.)

- Suppose we want to sort **Tiles** (rectangles plus “z-order”, i.e., depth in display)

```
class Tile extends Rectangle
{   private int z;
    public Tile(int x, int y, int w, int h, int zz)    { super(x,y,w,h);    z = zz;
}
```

- **Problem:** Cannot also be a subclass from Sortable.
- **Solution:** Inheritance of implementation and interface

```
public interface Comparable { public int compareTo(Object b); }

class Tile extends Rectangle implements Comparable
{   private int z;
    public Tile(int x, int y, int w, int h, int zz)    { super(x,y,w,h);    z = zz;

    public int compareTo(Object b) { Tile tb = (Tile)b; return z - tb.z; }
    ...
}
```

Inheritance: the best thing since sliced bread?

Advantages:

- Interface as contract
- Encapsulation
- Reusable/modifiable code
- Support for “frameworks” and other OO development principles.

Disadvantages:

- Explosion of classes
- Slower execution (price of “Late Binding”)
- High program complexity (model can be useful here!)

Conclusions

- Complexity of development/implementation \implies structuring mechanisms.
- At the language level:
 - Conceptual:** Subsystem, object, ADTs, interfaces, ...
 - Language support:** Encapsulation, name-spaces, modules, functors, interfaces, inheritance, polymorphism, ...
- Structuring important in bridging design and implementation.
 - Support allows (direct) realization of high-level concepts.
 - Some CASE-tools partially automate mappings, e.g., from class model to class hierarchy or from state chart to a state machine.
 - Important and nontrivial research topic (also here)!

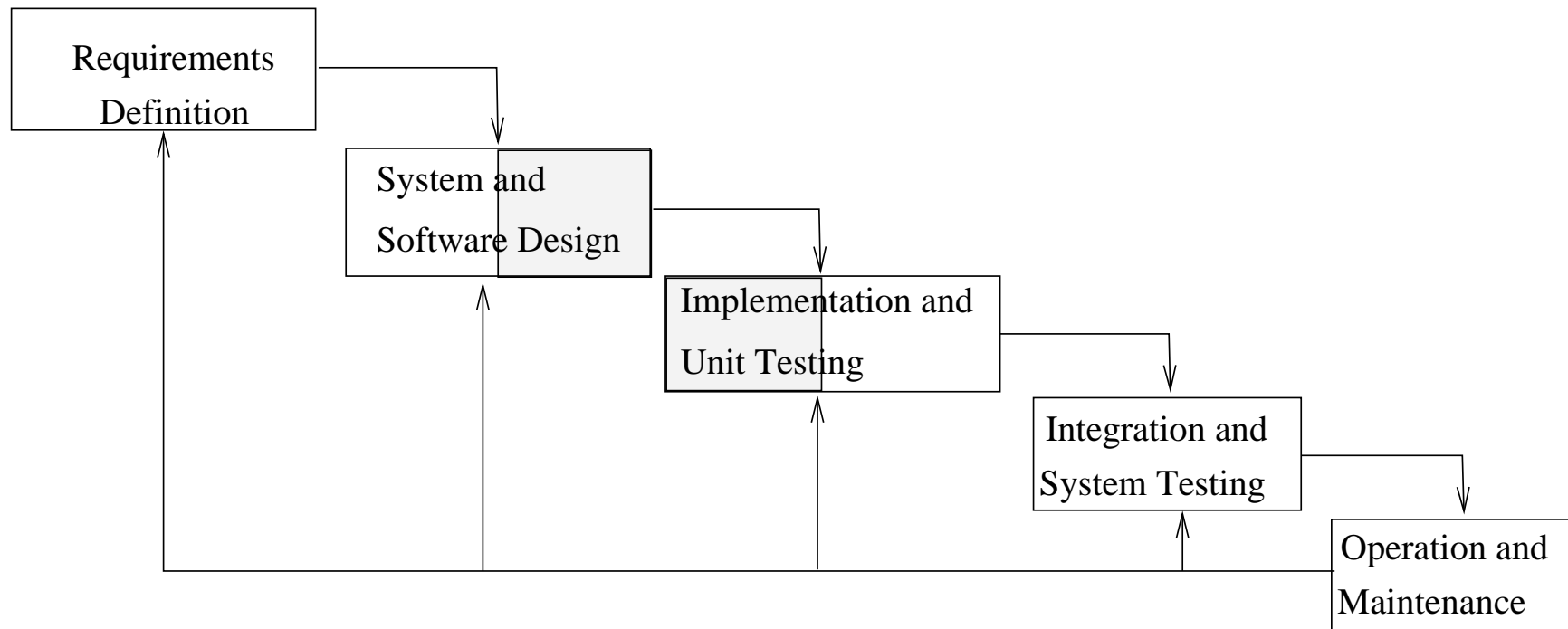
From Models to Code

David Basin

Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Spring 2002

Where are we?



Question: How do we make the transition from models to code?

General Idea

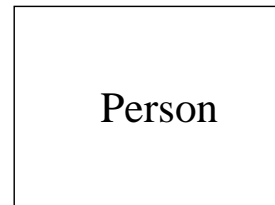
- Models are refined in steps, i.e., by incrementally adding information about static and dynamic properties
- Also possible are **refactorization** steps, whereby responsibilities are redistributed, e.g., by introducing **design patterns** or using standard components.
- At some point one must come to code! (base case)
- We will consider this process starting from both UML and Z models
 - Some aspects are relatively simple, e.g., class diagrams \implies class hierarchy.
 - Others are difficult. E.g., synthesis from sequence diagrams. **Why?**
- Process can be partially supported by CASE tools
 - Typically (only) code generation from class diagrams
 - “Round-Trip Engineering” sometimes also supported
 - Many open (research) problems

Code from class diagrams

- Class diagrams are well suited for code generation
 - Static constraints have a relative simple semantics
 - Interpretation is often unambiguous e.g., A is a subclass of B
 - Diagram components reflect aspects of OO-languages
- The situation is less straightforward for functional and dynamic aspects
 - Specifications are not constructive
 - Functionality is often underdefined (e.g., message sequences)
 - A (uniquely determined) function can be implemented by many algorithms
- Let's begin with class diagrams!

Classes \Rightarrow Classes

- Simplest case:



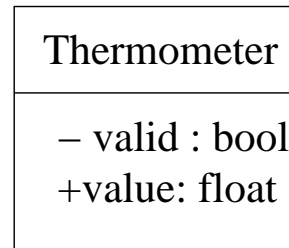
- Corresponding class declaration and method stub.

```
public class Person { public Person() {} }
```

- Explanation
 - The system has one class “Person”
 - It has one constructor (Is this assumption sensible?)
 - Nothing specified/generated beyond this
- Abstract class names (*italicized*) are implemented as abstract classes

Classes with attributes

- Attributes can be annotated:



- Annotation translated to public (+) or private (–) classifier
- Alternative: encapsulate state and generate get/set methods

```
public class Thermometer {  
    private boolean valid = false;  
    private float value = 0.0;  
  
    public Thermometer() {};  
    public Thermometer(float value){ this.value = value; }  
    public float getValue() { return value; }  
    public float setValue(float value) { this.value = value; }  
}
```

- Use of get/set methods is a design decision (not specified by UML)

Classes with operations

- Operations can be translated to method stubs.

Thermometer
– valid : bool + value : float
resetTo(val:float):bool

```
public class Thermometer
{
    private boolean valid;
    private float value;
    public boolean resetTo(float val) { }
}
```

- Or to an entire method (if code is also given):

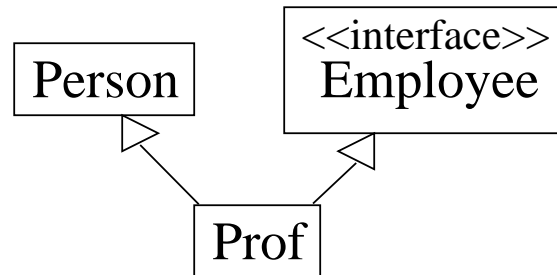
Thermometer
– valid : bool + value : float
resetTo(val:float):bool

```
if((val > 0) && val < 100))
{
    value = val;
    return true;
}
else return false;
```

```
public boolean resetTo(float val) {
    if((val > 0) && val < 100) { value = val; return true;}
    else return false }
}
```

Generalization

- Example with class/interface generalization



- Corresponds to

```
public interface Employee { ... }
public class Person {...}
```

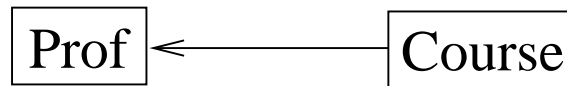
```
public class Prof extends Person implements Employee
{
    public Prof() { super(); } // calls Person
}
```

- Code generation with multiple inheritance possible for other languages.

Question: Can/should models be independent of the implementation language?

Associations

- Simple, directed associations



Course objects may send messages to professor objects

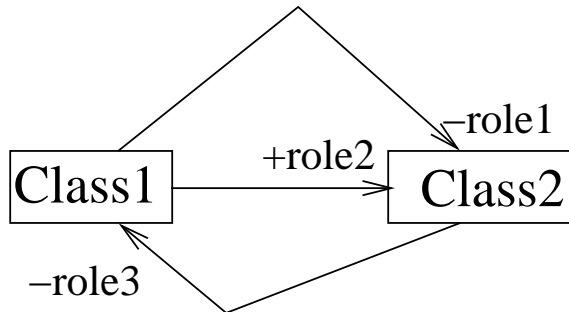
- Possible implementation: instance variable in class course + accessor functions.

```
public class Course {
    private Prof prof;

    public Prof getProf() { return prof; }           // Optional ‘‘structuring’’
    public void setProf(Prof prof) { this.prof = prof; }
}
```

Associations with direction and roles

- An instance variable is created for each role

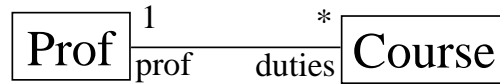


- Translation:

```
public class Class1 {  
    private Class2 role1;  
    public Class2 role2;  
}
```

```
public class Class2 {  
    private Class1 role3;  
}
```

Associations (cont.)



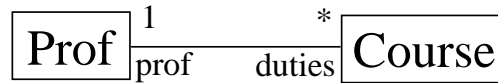
- Simplest translation: “Stubs mit Arrays”

```
public class Prof {
    public Course[] duties;
}
```

```
public class Course {
    public Prof prof;
}
```

- Alternative: implement associations using a relational database.
 - See database class for translating UML to SQL (and JDBC)
 - Some CASE tools support this through so-called “Technology Mappings”. One can then map constraints into SQL.

Associations (cont.)



- Specialized translations also possible
(here Prof manages a set of duties and each duty tracks an associated Prof.)

```

<<Prof.java>>
private OrderedSet duties;

public boolean hasInDuties(Course elem)
{ return this.duties.get(elem) != null; }

public Enumeration dutiesElements()
{ return duties.elements(); }

public void addToDuties(Course elem)
{ if(!this.hasInDuties(elem))
  { this.duties.add(elem); }
}

public void removeFromDuties(Course elem)
{ if(this.hasInDuties(elem))
  { this.duties.remove(elem);
    elem.SetProf(null); }
}

```

```

<<Course.java>>
private Prof prof;

public Prof getProf() { return prof; }

public void setProf(Prof prof)
{ if(this.prof != prof) { // new partner
  if(this.prof != null) { // inform old
    Prof oldProf = this.prof;
    this.prof = null;
    oldProf.removeFromDuties(this);
  }
  this.prof = prof;
  if(prof != null) { // inform new
    prof.addToDuties(this);
  }
}
}

```

Code generation from Z

- **Synthesis from Specifications** is a deep, specialized topic
 - The problem is difficult: data refinement, algorithmic development, . . .
 - The problem is undecidable

$$\frac{\text{Halt} \quad \begin{array}{l} f : \mathbb{N} \rightarrow \mathbb{N} \\ x : \mathbb{N} \end{array}}{f(x) = 1 \leftrightarrow M_x(x) \downarrow}$$

where M_x encodes the x th Turing Machine.

- Here we give only a taste (details in advanced courses)

Refinement

- **Refinement:** the transformation of an abstract specification into a concrete one
- Simple example:
 - Abstract:** $x' > x$
 - Concrete:** $x' = x + 1$
 - Implementation:** $x' = x + 1$ (syntax depending on programming language)
- Refinement leads to **stronger** models/formalisms
 - Semantically: more deterministic (fewer behaviors)
 - Logically: concrete specification implies abstract specification
- Correctness of refinement

$$x' = x + 1 \Rightarrow x' > x$$

A refinement example

- Refining sets: $[X]$

$Abstract$ $s : \mathbb{P} X$

- with abstract storage operation

$AStore$ $\Delta Abstract$ $x? : X$
$s' = s \cup \{x?\}$

Question: How do we implement this “system” in a PL with arrays and lists?

Answer: Via data-refinement, whereby we “implement” sets by sequences.
Sequences can later be implemented in the target language.

Refinement (cont.)

- Concrete state with sequences

Concrete $ss : \text{seq } X$

- Store element at end

$CStore$ $\Delta \text{Concrete}$ $x? : X$
--

$ss' = ss \frown \langle x? \rangle$

Question: In what sense does a sequence implement a set?

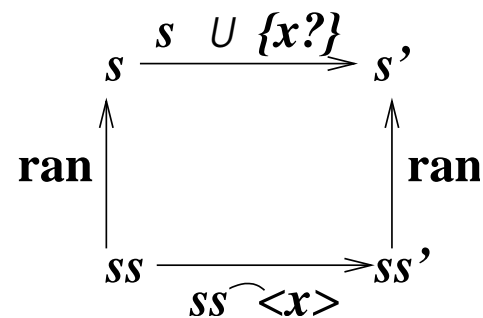
$$\{1, 2, 3\} \stackrel{?}{=} \langle 2, 3, 1, 3 \rangle$$

Is this refinement correct?

Refinement — Correctness

- Correctness (relative to an invariant):

Prove that the **concrete implementation is stronger than the abstract one**.



- Returning to our example:

invariant: $s = \text{ran } ss$ (i.e., also $s' = \text{ran } ss'$)

abstract operation: $s' = s \cup \{x?\}$

concrete operation: $ss' = ss \cap \langle x? \rangle$

- Refinement is correct when:

$$s = \text{ran } ss \wedge s' = \text{ran } ss' \wedge ss' = ss \cap \langle x? \rangle \Rightarrow s' = s \cup \{x?\}$$

Proof of correctness

- Antecedent

$$s = \text{ran } ss \wedge s' = \text{ran } ss' \wedge ss' = ss \cap \langle x? \rangle$$

- Consequence

$$s' = s \cup \{x?\}$$

- Equivalence proof (based on antecedent and 2 lemmas)

$$\begin{aligned} s' = s \cup \{x?\} &\Leftrightarrow \text{ran } ss' = s \cup \{x?\} \\ &\Leftrightarrow \text{ran}(ss \cap \langle x? \rangle) = s \cup \{x?\} \\ &\Leftrightarrow \text{ran } ss \cup \text{ran } \langle x? \rangle = s \cup \{x?\} \\ &\Leftrightarrow \text{ran } ss \cup \{x?\} = s \cup \{x?\} \\ &\Leftrightarrow s \cup \{x?\} = s \cup \{x?\} \\ &\Leftrightarrow \text{true} \end{aligned}$$

From Z to Code

- There are “pragmatic methods” to generate programs from Z specifications

We consider several here. Methods also depend on the programming language employed (Haskell versus C/C++ versus Java ...).

- Idea:

Z types	\rightsquigarrow	data type definitions
Z axiomatic definitions	\rightsquigarrow	constants (or infrequently changing data)
schemata	\rightsquigarrow	mutable data

Data types

- Free types correspond enumeration types in programming languages

$$FAULT ::= overload \mid line_voltage \mid overtemp \mid ground_short$$

S	
...	
$faults : \mathbb{P} FAULTS$	
...	

- In Haskell: `data Fault = Overload | Line_voltage | Overtemp | Ground_short`
- The schema S defines to a tuple $(\dots, faults, \dots)$, where $faults$ is realized as a list of type `[Fault]`
- In C:

```
typedef enum { OVERLOAD = 0, LINE_VOLTAGE, OVERTEMP, GROUND_SHORT} fault;

#define N_FAULTS GROUND_SHORT + 1
int faults[N_FAULTS];
```

Relations

$$\frac{\text{phone} : \text{Name} \leftrightarrow \mathbb{Z}}{\text{dom phone} = \text{subscribers}}$$

- Encoding (in Haskell):

```
type Name = String
type Phone = (Name,Int)
```

```
phones :: [Phone]
phones = [("Luca",2038243),("Stefan",2038244),("Abdel",2038244)]
```

- Using a relational database is another possibility.
- Functions (over finite domains) also implementable as array or list of pairs.
Example: declaration $u : \mathbb{N} \rightarrow \mathbb{Z}$ can be realized in C as:

```
int u[MAXVAL];
```

Other possibilities: hash-tables, vectors (Java), or a procedure.

Schemata

- Schemata specify mutable data

Correspond to (composite) variable declarations, or records, or structures (in C), or classes (in C++/Java), etc.

- Example: $S \hat{=} [x, y : \mathbb{Z}]$ can be translated as:

- `int x,y;`
- or `typedef struct { int x,y; } S;`
- or `public class S { int x, y; }`

- Instance variables $sa, sb, sc : S$ interpreted (in C) as $S \ sa, sb, sc;$

In Java, one produces 3 objects of class S .

Schemata (cont.)

- A larger example

PS $contactor : SWITCH$ $preset, setpoint, output : SIGNAL$ $faults : \mathbb{P} FAULT$ <hr/> $contactor = open \Rightarrow setpoint = 0$ $contactor = open \Rightarrow output \leq \epsilon$
--

- C implementation:

```
typedef int signal
typedef enum { OPEN, CLOSED } switch;

typedef struct power_supply
{
    switch contactor;
    signal preset, setpoint, output;
    int faults[N_FAULTS]
} PS;
```

Invariants correspond to **properties** of code, but **not code** itself!

From Constraints to Code

- Constraints can be refined to code.
- **Syntax:** $A \sqsubseteq C$ for C Concrete refines A Abstract.
- **Semantics:** Every refinement law can be cast as $P \wedge Q \sqsubseteq S$ Should hold when Hoare triple $\{P\} S \{Q\}$ is provable.

- **Example:**

$$x' = x \wedge y' = x \sqsubseteq t = x; x = y; y = t$$

Refinement law asserts provability of the Hoare triple

$$\{true\} t = x; x = y; y = t; \{x' = y \wedge y' = x\}$$

- Topic **refinement calculi** treated in advanced courses.
 - There are a variety of formal approaches, some with computer support.
 - We present here just examples of possible refinement steps.

Expressions, Sets, and Functions

- One can implement expressions (fairly) directly in languages like C, Java, ...

$$+, *, \textit{div}, \textit{mod} \rightsquigarrow +, *, /, \%$$

$$=, \neq \rightsquigarrow ==, !=$$

$$\wedge, \vee \rightsquigarrow \&\&, ||$$

$$\textit{true}, \textit{false} \rightsquigarrow 1, 0$$

E.g., $x \bmod 2 \neq 0 \sqsubseteq x \% 2 != 0$

- Set membership: $x \in s \sqsubseteq \textit{Search for } x \textit{ in data structure } s$

If implementation is fixed, e.g., Boolean Arrays, then test is trivial: $x \in s \sqsubseteq s[x]$

- Function application: refinement depends on data structures.

Table: $u(x) \sqsubseteq \textit{Find item in } u \textit{ with key } x$

Function: $f(x) \sqsubseteq f(x)$

Assignment

- No change \Rightarrow no code

$$x' = x \sqsubseteq (\text{empty statement})$$

- Single change \Rightarrow simple assignment

$$x' = e \wedge y' = y \wedge z' = z \sqsubseteq x = e$$

- Multiple changes can~~not~~ necessarily be refinement to multiple assignments

$$x' = y \wedge y' = x \not\sqsubseteq x = y; y = x$$

- General solution requires data-flow analysis/auxiliary variables.

$$x' = e_1(x, y) \wedge y' = e_2(x, y) \sqsubseteq t = x; x = e_1(x, y); y = e_2(t, y)$$

- Direct assignment impossible when the target is a data structure.

$$S' = S \cup \{x\} \sqsubseteq \text{Put } x \text{ in data structure } S$$

Example: for boolean arrays: $S' = S \cup \{x\} \subseteq s[x] = \text{TRUE}$

Logical Connectives

- Conjunction: can sometimes be refined to a conditional statement

$$p \wedge s \sqsubseteq \text{if}(p) \ s$$

Example:

$$\begin{aligned} x = e_1 \wedge x' = e_2 &\sqsubseteq \text{if}(x = e_1) \ x' = e_2 \\ &\sqsubseteq \text{if}(x == e_1) \ x = e_2 \end{aligned}$$

- Disjunction: (where p and q describe different conditions)

$$(p \wedge s) \vee (q \wedge t) \sqsubseteq \text{if}(p) \ s; \text{ else if}(q) \ t$$

Special case:

$$(p \wedge s) \vee (\neg p \wedge t) \sqsubseteq \text{if}(p) \ s; \text{ else } t$$

- N.B.: These rules are not complete. E.g.,

$$d \neq 0 \wedge n = q' * d + r' \wedge r' < d \sqsubseteq ???$$

Quantifiers

- Quantifiers (over finite domains) can be replaced by loops

$$\forall x : S \bullet p(x) \sqsubseteq b = 1; \text{ “for } (x \in S)\text{” if } (!p(x)) \text{ } b = 0;$$

Here “*for* ($x \in S$)” is informal. Can be concretized, depending on particular data structures. E.g., S implemented with Boolean Arrays:

$$\forall x : S \bullet p(x) \sqsubseteq b = 1; \text{ for } (i = 0; i < n; i++) \text{ if } (!p(s[i])) \text{ } b = 0;$$

- Question:** How can $\exists x : S \bullet p(x)$ be refined?

Structuring

- Z has a flat, global state:

$$\begin{aligned} S &= \{x, y : \mathbb{Z}\} \\ Op &= \{\Delta S \mid x' = x + y\} \end{aligned}$$

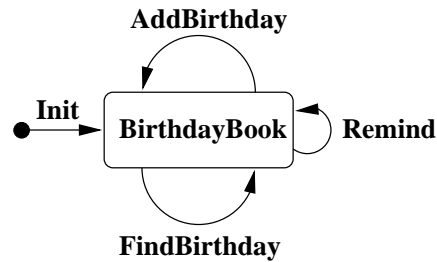
- Direct implementation (e.g., in C) with global variables.

```
int x, y;      /* alternatively packed in a structure */  
  
void op(void) {x = x + y;}
```

- In Java: state is localized in objects:

```
public class S {  
    private int x, y;  
  
    public void op() { x = x + y;}  
}
```


Z Specification \Rightarrow a Class



- Entire specification corresponds to a class

<div style="border-bottom: 1px solid black; margin-bottom: 10px;"> <i>Birthdaybook</i> </div> <div> $known : \mathbb{P} NAME$ $birthday : NAME \leftrightarrow DATE$ </div> <hr style="border: 0.5px solid black; margin: 5px 0;"/> <div> $known = \text{dom } birthday$ </div>

```

class BirthdayBook {
    private HashSet known;
    private Hashtable birthday;

    public BirthdayBook() {
        known = new HashSet();
        birthday = new Hashtable();
    } ...
}
  
```

Birthday Book (conts.)

- Each Δ -Schema defines a method

$\begin{array}{l} \text{AddBirthday} \\ \hline \Delta \text{BirthdayBook} \\ \text{name?} : \text{NAME} \\ \text{date?} : \text{DATE} \\ \hline \text{name?} \notin \text{known} \\ \text{birthday}' = \text{birthday} \cup \{\text{name?} \mapsto \text{date?}\} \end{array}$
--

- Refined as:

$$\begin{array}{l} \text{name?} \notin \text{known} \wedge \text{birthday}' = \text{birthday} \cup \{\text{name?} \mapsto \text{date?}\} \\ \sqsubseteq \text{if}(\text{name?} \notin \text{known}) \text{ birthday}' = \text{birthday} \cup \{\text{name?} \mapsto \text{date?}\} \\ \sqsubseteq \text{if}(!\text{known.contains}(\text{name})) \text{ birthday.put}(\text{name}, \text{date}); \end{array}$$

- The result (+ Constraint)

```
public void AddBirthday(String name, String date) {
    if(!known.contains(name)) {
        birthday.put(name, date);
        known.add(name);
    }
}
```

Birthday Book (Cont.)

- The same holds for Ξ -Schemata.

FindBirthday

Ξ *BirthdayBook*

name? : *NAME*

date! : *DATE*

name? \in *known*

date! = *birthday*(*name?*)

```
public String FindBirthday(String name) {
    if(known.contains(name)) {
        String date = (String)birthday.get(name);
        return date;
    }
    else return "NO-ENTRY"; }
```

Alternative: throw an exception. (Dis)Advantages?

- Control is not fixed.

An event model can specify how “the machine” reacts to events

Conclusion

- Program synthesis is a creative procedure
 - CASE tools can help with the skeleton
 - In general one cannot (completely) replace creativity
- Prerequisite for an (automatic or manual) refinement is a formal specification language
- Birthday book example illustrates the way from Z to code
 - Z specification corresponds to a machine
 - Control is defined elsewhere (e.g., state chart)
- There are OO-variants of Z (Object-Z, OOZE, Z++) that better support OO-features and even control

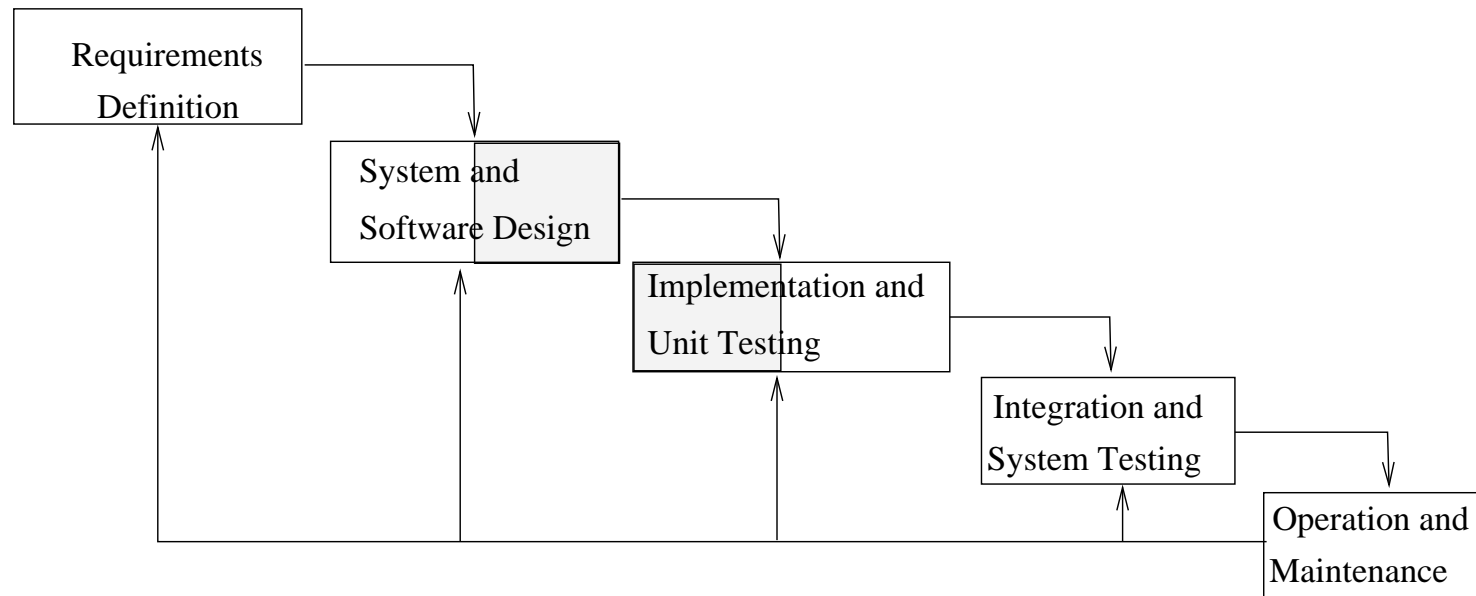
Models to Code (Patterns)

David Basin

Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Spring 2002

Where are we?



- Recall: how can models be used?

Conceptual: To formalize domain concepts. No direct relationship to an implementation.

Specification: To specify interfaces and (in part) semantics

Implementation: Basis for implementing object oriented design

- A conceptual or coarse-grain model is not necessarily adequate as a fine-grained model or for implementation

Problems/Solutions

- Problems
 - High-level models omit implementation details
 - Design models may be too problem specific
- Solutions
 - Refine model with additional details
 - “Refactorize” model to realize a system with comparable behavior, however with improved (structural, efficiency, ...) properties
 - * Decomposition geared towards reuse of existing components
 - * Application of patterns/frameworks/... to increase reusability
- Let's examine one such possibility: design patterns

Design patterns — the idea

- **Goal:** describe a general solution for a class of development problems
- **Description:** (Alexander, on architecture, 1977):

Each pattern describes a **problem** which occurs over and over again in our environment, and then describes the core of the **solution** to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.
- Concept popularized by the **Gang of Four**: Gamma, Helm, Johnson, Vlissides
 - They developed a catalog of design patterns
 - Encompassed “expert” design principles for improving software quality.
- **Aim:** Simpler development, increased reusability, documentation, ...
- General classification for patterns also given

Name, applicability domain, structure (UML-like models), examples, ...

An example: observer pattern (in gang-of-4 format)

Name: Observer

Type: Object behavioral

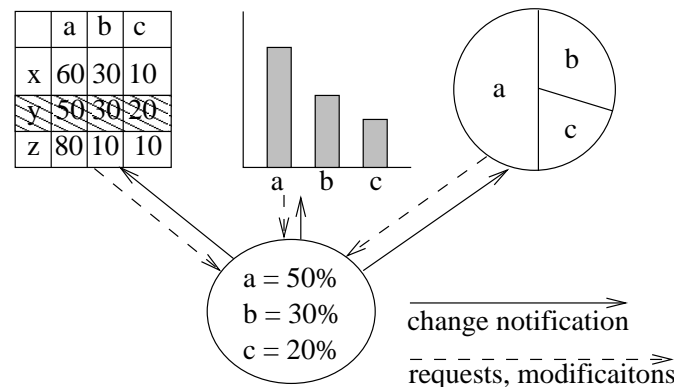
Intent: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Also Known As: Dependents, Publish-Subscribe

Observer pattern (cont.)

Motivation: A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

For example, many GUIs separate presentation from the underlying application data allowing classes defining application data and presentation to be independently reused.



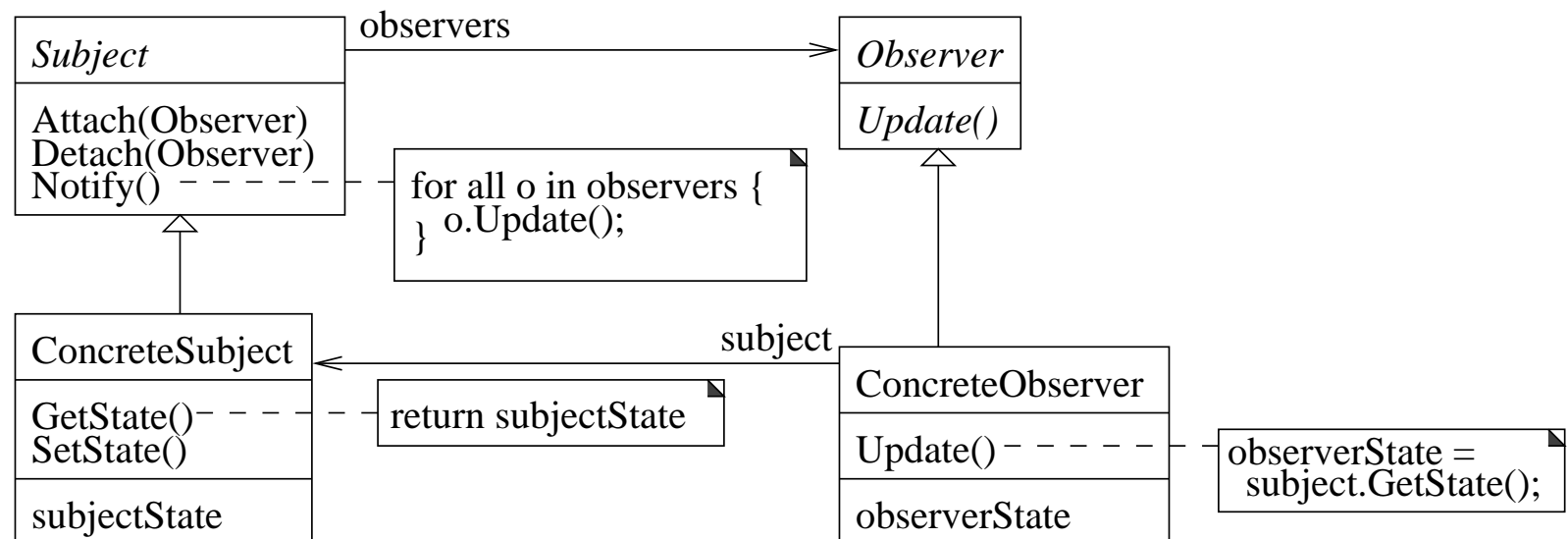
The observer pattern describes how to establish these relationships. The key objects in this pattern are **subject** and **observer**. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state.

Observer Pattern (cont.)

Applicability: Use the observe pattern in any of the following situations.

- When an abstraction has two aspects, one dependent on the other. Encapsulating these in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many others must be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Structure:



Observer Pattern (cont.)

Participants:

Subject: Knows its observers and provides an interface for attaching and detaching Observer objects.

Observer: Defines an updating interface for objects that should be notified of changes in a subject.

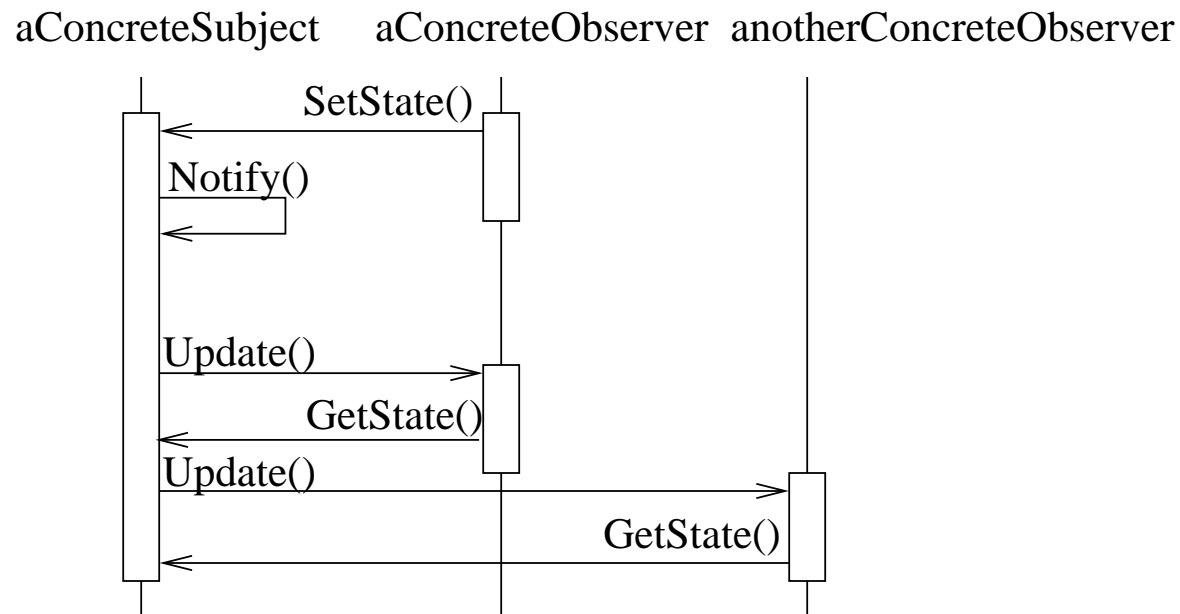
ConcreteSubject: Stores state of interest to ConcreteObserver objects and sends a notification to its observers when its state changes.

ConcreteObserver: Maintains a reference to a ConcreteSubject object, stores state that should stay consistent with the subjects, and implements the Observer updating interface to keep its state consistent.

Observer Pattern (cont.)

Collaborations:

- ConcreteSubject notifies its observers whenever a change occurs.
- After being informed of a change, a ConcreteObserver may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.



Observer Pattern (cont.)

Consequences:

The Observer pattern lets you vary subjects and observers independently. Further benefits and liabilities include:

1. *Abstract coupling between Subject and Observer.* All a subject knows is that it has a list of (Observe class conform) observers. ... They can belong to different layers of abstraction in a system.
2. *Support for broadcast communication.* Unlike an ordinary request, the notification that a subject sends needn't specify its receiver. The notification is broadcast automatically to all objects that subscribed to it. The subject doesn't care how many interested objects exist; its only responsibility is to notify its observers. It's up to the observer to handle or ignore a notification.
3. *Unexpected updates.* Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject. A seemingly innocuous operation may cause a cascade of updates. This problem is aggravated by the fact that the simple update protocol provides no details on *what* changed in the subject. Without additional protocol to help observers discover what changed, they may be forced to work hard to deduce the changes.

Observer Pattern (cont.)

Implementation:

- 1) *Mapping subjects to their observers.* The simplest way for a subject to keep track of observers is to store explicit references to them. Such storage may be too expensive when there are many subjects and few observers. One solution is to trade space for time by using an associative look-up (e.g., a hash table) to maintain the subject-to-observe mapping. ...
- 3) *Who triggers the update?* The subject and its observers rely on the notification mechanism to stay consistent. But who actually calls Notify to trigger the update? Here are two options:
 - (a) Have state-setting operations on Subject call Notify after they change the subject's state. This is easy to implement but has the disadvantage that consecutive operations will cause consecutive updates, which may be inefficient.
 - (b) Make clients responsible for calling Notify at the right time. ...
- 6) *Avoiding observer-specific update protocols: the push and pull models.* Implementations of the Observer pattern often have the subject broadcast additional information about the change (as an argument to Update). At one extreme, which we call the **push model**, the subject sends observers information about the change, whether they want it or not. At the other extreme is the **pull model**; the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter.

Observer pattern — in Java

- The Observe Pattern is implemented in Java using *Observer* Interface.

```
public void update(Observable o, Object arg)
```

This method is called whenever the observed object is changed. An application calls an Observable object's notifyObservers method to have all the object's observers notified of the change. Parameters:

o - the observable object.

arg - an argument passed to the notifyObservers method.

- **Observable** Class defined as follows: (“data” = “subject”)

```
public class Observable extends Object
```

This class represents an observable object, or "data" in the model-view paradigm. It can be subclassed to represent an object that the application wants to have observed. An observable object can have one or more observers. ...

Methods:

Observable(): Construct an Observable with zero Observers

void addObserver(Observer o): Adds an observer to the set of observers for this object ...

Design patterns: conclusion

- Design patterns offer a technique for structuring fine development
Other techniques for coarse-grained structuring, e.g., architectures.
- Use is wide spread, e.g., see Java libraries!
- Good example of the use of UML diagrams to present conceptual ideas
- Patterns are integrated in some CASE-TOOLS for documentation and code-integration

Graphical User Interfaces- Designs, Models, Implementations

Burkhart Wolff

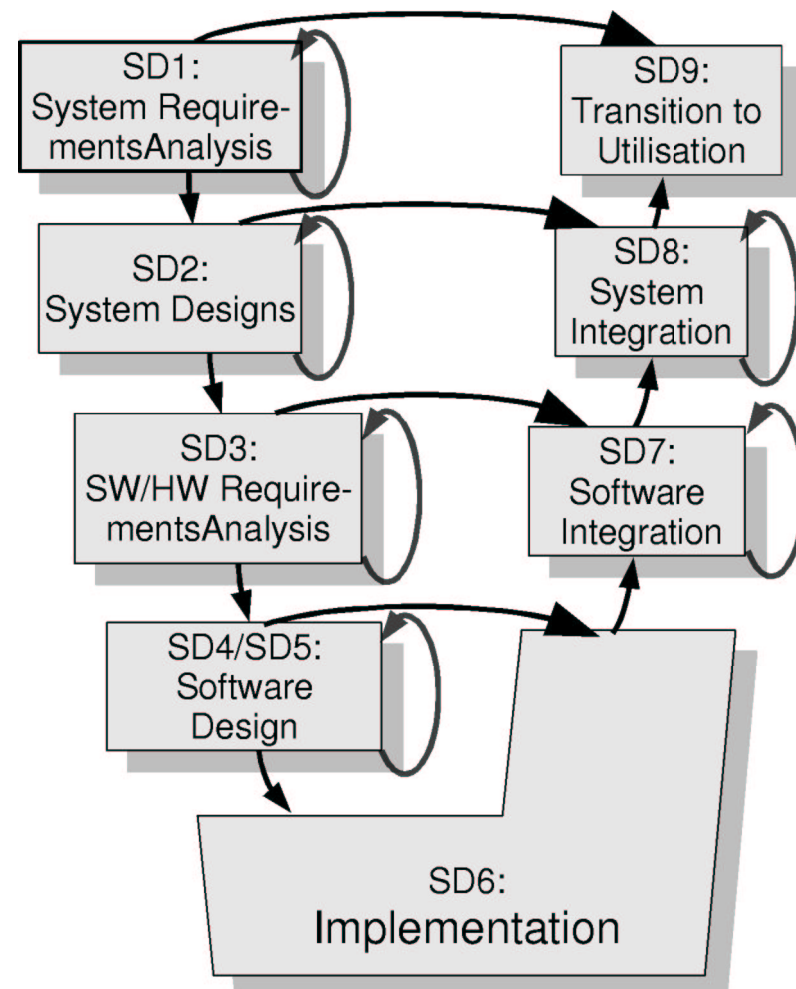
AG Softwaretechnik

Overview

- **Designing** the User Interfaces (UI): goals, principles, techniques
- **Modeling** and **Validating** *graphical* UI's
- Concrete Technologies:
 - Implementation in **Java/Swing**
 - Implementation in **SmlTk**

Overview

- Where are we in the development process?
- ... there are bits all over ...



Designing the User Interface: goals, principles, techniques

- General Goals
 - intuitive and easy user interaction
 - adapting to human abilities, backgrounds, motivations . . .
 - adapting to human perceptual, cognitive and motor abilities.
 - . .

⇒ key-issue for "information society" !!!

- VRS
- speech recognition and generation
- GUI's

⇒ research field: HCI

Designing the GUI: goals, principles, techniques

- Eight Design-Goals (cf. [Shneiderman 98])
 - reduce short-term memory load
 - visual representation of objects
 - adaptivity for novice and expert users
 - consistency
 - conformance to user expectance
 - "informative" feedback, simple error handling
 - easy reversal of actions
 - design dialogs to yield closure

Designing the GUI: goals, principles, techniques

- Designing a GUI:
Art or Software Engineering?
- **artistic GUI** solutions exist:
 - specialized for games or highly graphical application programs (Kai's Power Tools on Mac)
 - Web-Design Solutions
- mostly: GUI's are just **engineering tasks** . . .
 - controlled by GUI-style-guides (Mac, Motif, Windows ...)
 - reaction to "conformance to user expectance"
 - reaction to technology (GUI-component libraries)

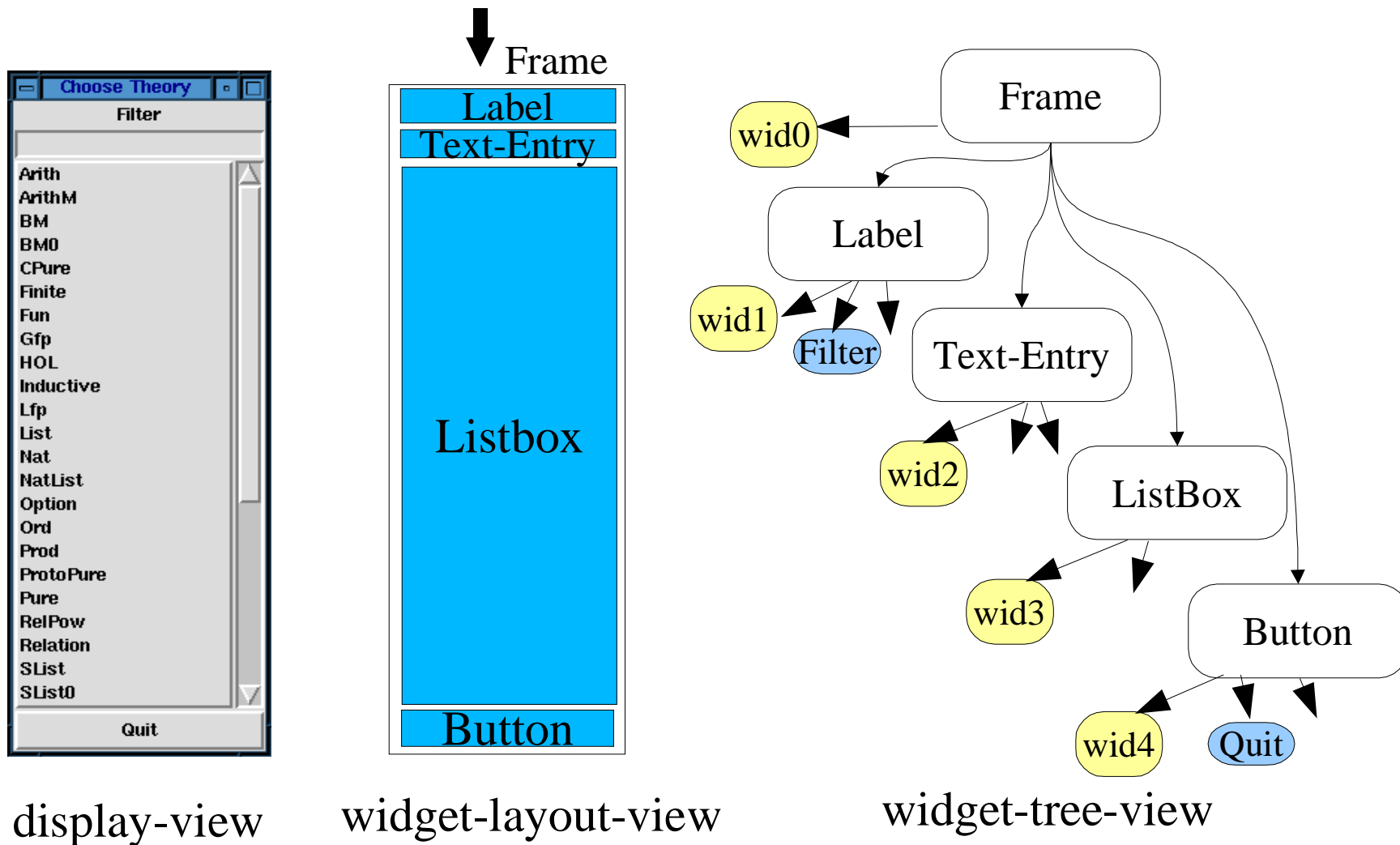
Designing the GUI: goals, principles, techniques

- Graphical Display with Colors
- Special hardware for (re)-drawing and overlay blocks
- Mostly used:

the *WIMP*-Style Design
[Dix et. al 98]

- windows, icons, menus, pointers or
widgets, icons, mice and pull-down menus
- core: a data structure widget (Tk) or
graphical component (awt/Swing),
organized in a tree-like structure.

Designing the GUI: goals, principles, **techniques**



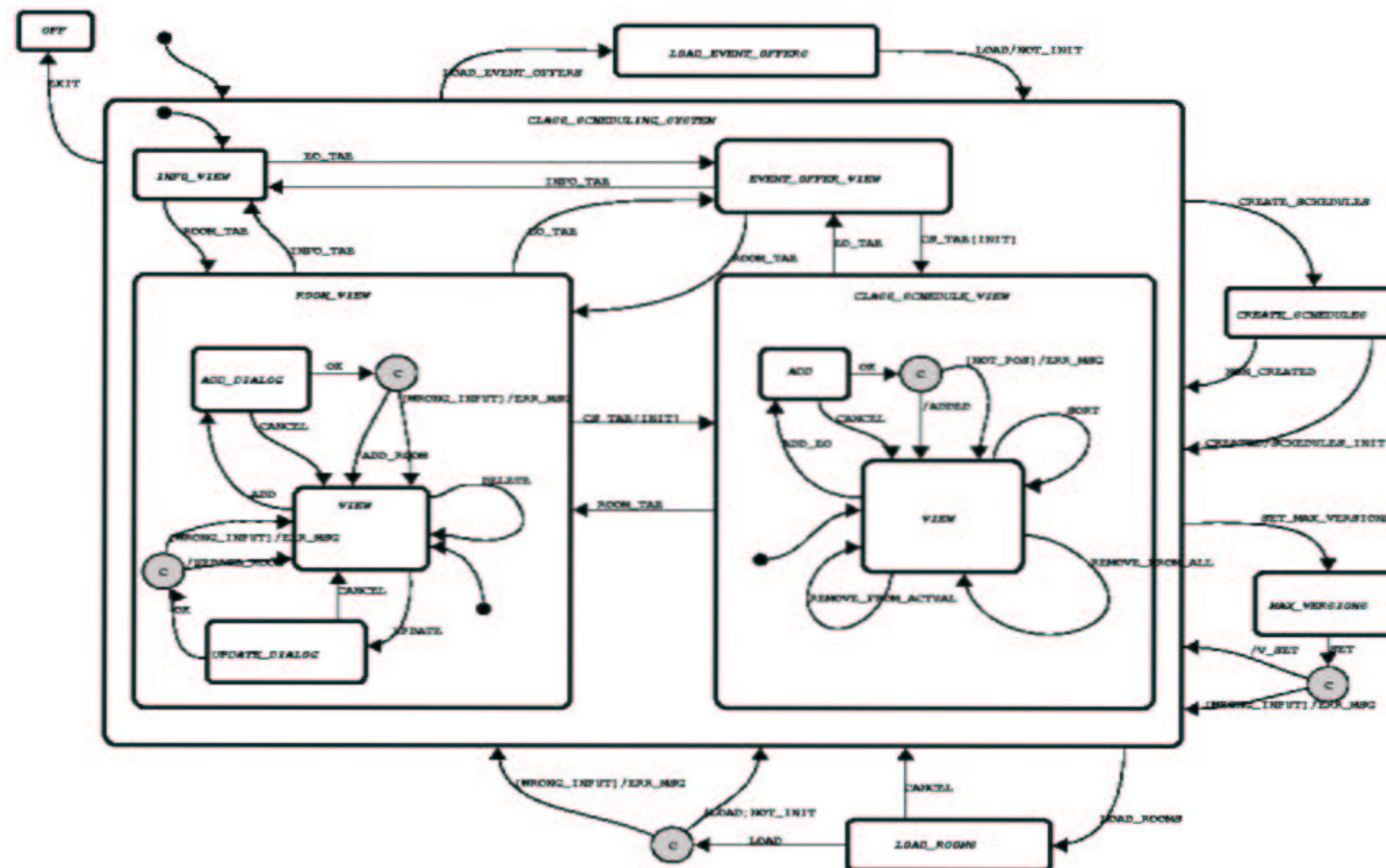
Designing the GUI:

goals, principles, **techniques**

- GUI's are reactive systems:
binding *Events* to *Actions*
- ⇒ GUI's are inherently event/action-systems
- ⇒ ... all problems of concurrent programming
 - interleavings,
 - locking of resources,
 - synchronization,
 - call-backs, notifiers, etc. . .
- ⇒ ... GUI-development often underestimated

Modeling and Validating GUI's

- The obvious choice:
Event-Modeling Formalisms such as **StateCharts**



Modeling and Validating GUI's

- System integration level:
 - system: ergonomic metrics and criteria (GOMS-model)
- Software integration level
 - by hand
 - by replay
 - by generated **replay-scripts** (Test-Case-Generation)
 - by **verification**

Validation

- by generated replay-scripts

SUN-TEST †, . . .

- + Test-Case-Generation
Path-Generation from StateChart,
on *dummy* data-model
- + Test-Case-Generation based on Data-Model
for System-Integration Test

Validation: by Verification

- Formalizing the GUI-Event-Model
 - in temporal logic
 - in process algebra formalisms such as CSP (concurrent sequential processes, Hoare/Roscoe)
- Formalizing design goals
- Proving *Refinement* of GUI-Event-Model

Validation by Verification - an Example in CSP

Basissets for Events:

WIDS == {wid1,wid2,wid3,wid4}
EVENTS == {return, button-1,button-2,button-3,
 button-1-R,button-2-R,button-3-R}
ACTIONS == {read_text, read_pos,
 set_cursor, unset_cursor,
 close_window,open_window}

Application as interleaving of window
threads with a clipboard thread

$$APPL = win_1[|cbe|] \dots [|cbe|] win_n [|cbe|] clipboard$$

Validation by Verification - an Example in CSP

- The behaviour of a window as a *process*:

```

win1 = letrec csr t = winwid2?return → winwid2!read_text
      → rec?x → redisplaywid3!x → csr x
    □ winwid3?button-1 → winwid3!read_pos
      → rec?p → winwid3!read_text(p)
      → rec?x → clibboard!x → set_cursor -> csr t
    □ win_?button_1-R → unset_cursor → csr t
    □ winwid3?button-1 → close_window → Skip
in redisplaywid3!"" → csr ""

```


Validation by Verification - an Example in CSP

Potential for analysis

Checking usability properties such as:

GUI deadlockfree

any win_i will be left after max. 3 quit-button events

similar: undo possible for all win_i

Analysis possibly automatic with Tool *FDR* !

Concrete Technologies:

- Implementation in Java-Swing
- Implementation in SmlTk

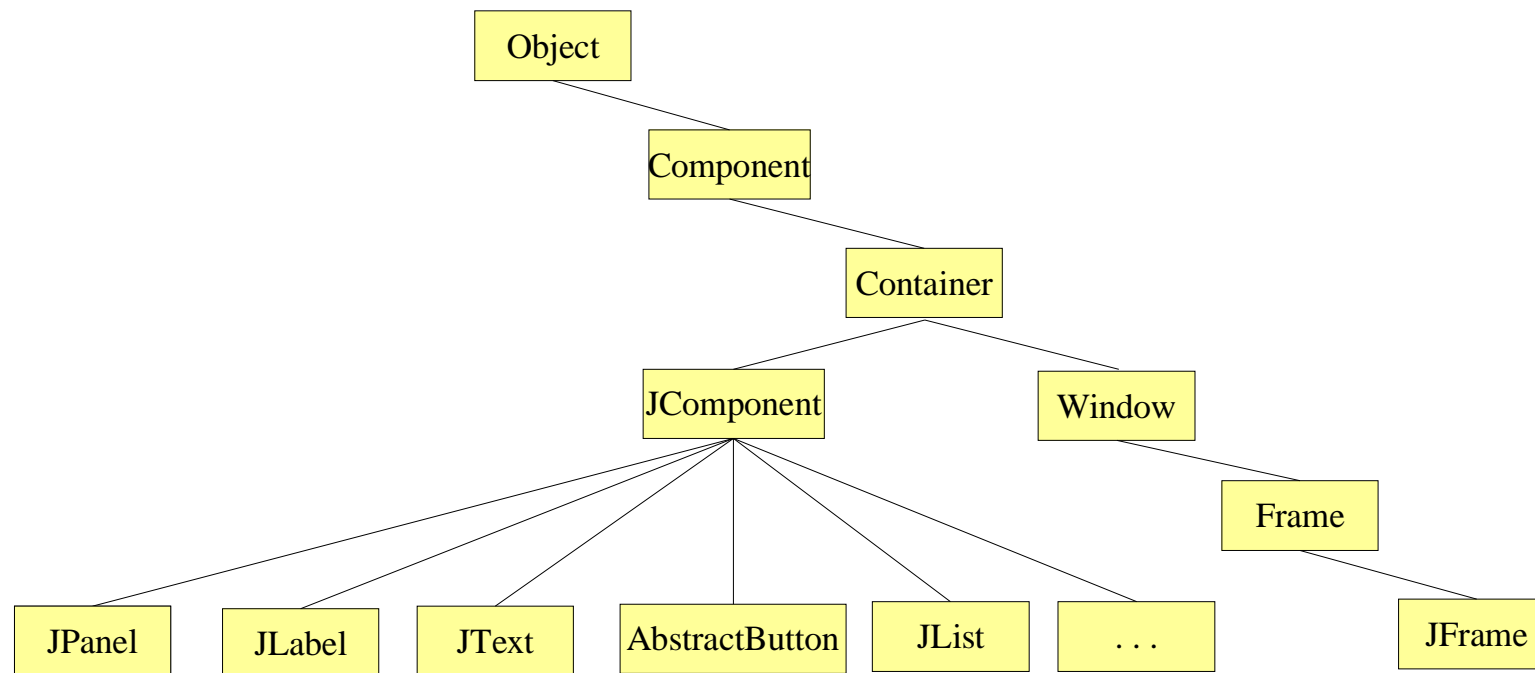
Concrete Technologies: Java/Swing

- Popular, powerful Technology
(originated by Netscape (since 1996) & SUN)
 - object-oriented library design
 - potential for concurrent operations
via Java-threads
 - support for customization
(look-and-feels such as "Metal", "Motif", "Windows")
 - High-level Components such as TreeLists, Tables, etc.
 - Comes in two flavors:
 - Java/AWT: peer-approach (problems with portability)
 - Java/Swing: painting-approach (problems with speed)

⇒ Swing recommended; shares Event-Model with AWT.

Concrete Technologies: Java/Swing

Window Components organized in Class-Hierarchy:



Component: repaint, setLayout, add, isVisible, setVisible, isShowing, getLocation, setLocation, getSize,...

Window: toFront, toBack

Frame: setResizable, setTitle, setIconImage

Concrete Technologies: Java/Swing

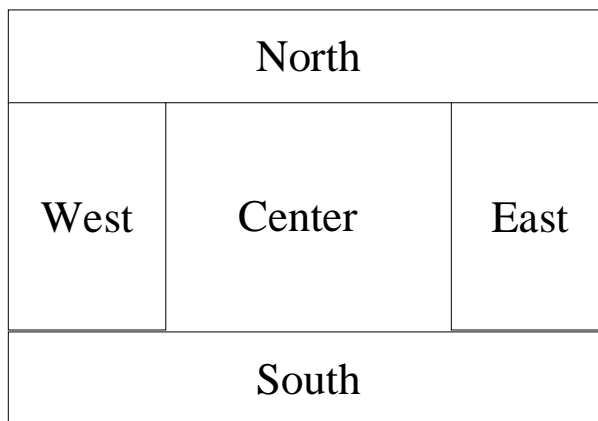
- Methods organized in **Model-View-Controller** (MVC)
Design Patterns
 - content (gui-state; eg. state of text-fields, ...)
 - visual appearance ("configurations", color, size, ...)
 - behaviour (reaction to events)
- Remarks:
 - it is not that **original** (Tk has it too)
 - not much more than a **stylistic guideline**
 - sometimes difficult to separate view and control ([HC 99]379)
 - even in simple cases a "grief for the programmer"
([HC 99]380)

Concrete Technologies: Java/Swing

Layout Management

controlled by a default Layout Manager (replacable)

border Layout:



```
class MyPanel extends JPanel
{
    setLayout(new BorderLayout());
    . . .
    add(yellowButton, "South");
}
```

Concrete Technologies: Java/Swing

Event-Model (from AWT) : Observer-Pattern.

Listeners \Rightarrow Event-Consumers (Container)

Events \Rightarrow Listeners (occurring in Containers)

low-level events

KeyEvent, MouseEvent, TextEvent, WindowEvent, ...

semantic events

ActionEvent, WindowEvent,

Concrete Technologies: Java/Swing

Listeners : Characterized by Java *Interfaces*

ActionListener

WindowListener

```
public void windowClosing(WindowEvent e)
public void windowClosed(WindowEvent e)
public void windowIconified(WindowEvent e)
public void windowOpened(WindowEvent e)
public void windowDeiconified(WindowEvent e)
public void windowActivated(WindowEvent e)
public void windowDeactivated(WindowEvent e)
```

KeyListener, MouseListener, . . .

For convenience: `KeyAdapter`, `WindowAdapter`, ... with defaults

Concrete Technologies: Java/Swing

- Preemptive Multitasking: Java Threads

- create kill
- group threads
- control priorities
- locking objects or methods:
built-in in Java

```
public synchronized void put(int value) {  
    // CubbyHole locked by the Producer  
    ..  
    // CubbyHole unlocked by the Producer  
}
```

- more and more locked Components in Swing \Rightarrow
problems with efficiency . . .

Concrete Technologies: A Java/Swing Example

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class SimpleExampleFrame extends JFrame
    implements KeyListener
{
    public SimpleExampleFrame ()
    {
        setTitle("Please enter name:");
        setSize(220,120);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing
                (WindowEvent e)
            {
                System.exit(0);
            }
        } );

        Container contentPane = getContentPane();

        JPanel panel = new JPanel();
        JLabel label = new JLabel("name:");
        panel.add(label, "West");

        entry = new JTextField(12);
        entry.addActionListener(this);

        panel.add(entry, "East");

        contentPane.add(panel,"North");

        JButton quitButton = new JButton("Quit");
        quitButton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent evt)
            {
                System.exit(0);
            }
        });

        JPanel panel2 = new JPanel();
        panel2.add(quitButton);
        contentPane.add(panel2,"South");
    }
}

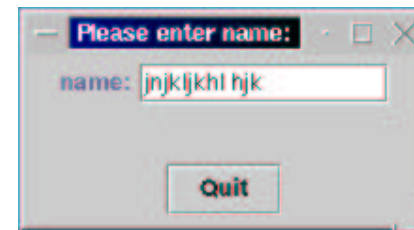
```

Concrete Technologies: A Java/Swing Example

```
public void actionPerformed(ActionEvent evt)
{
    Object source = evt.getSource();
    if (source == entry)
    {
        String h = entry.getText();
        super.setTitle(h);
    }
    else {}
}

private JTextField entry;
}

public class SimpleExample {
    public static void main(String[] args)
    {
        JFrame f = new SimpleExampleFrame();
        f.show();
    }
}
```

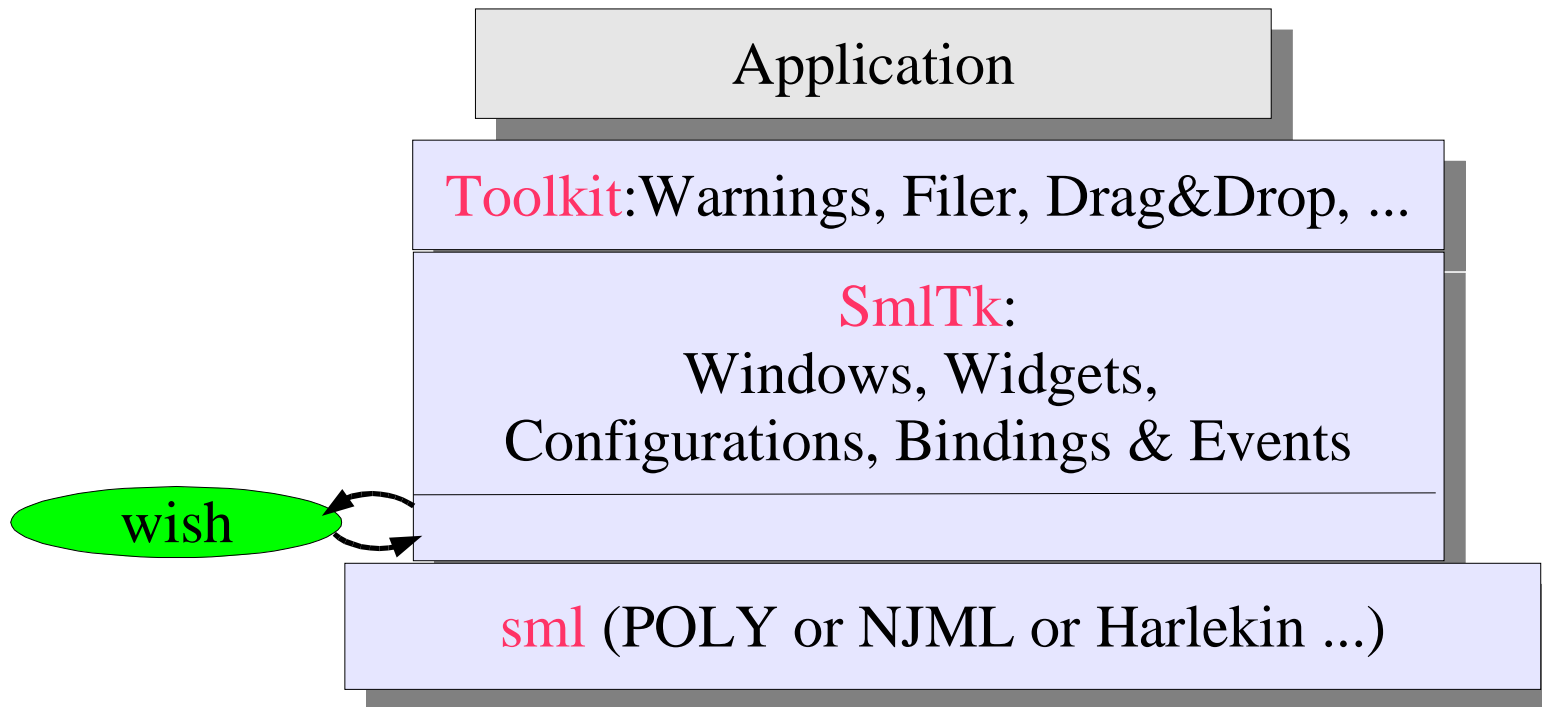


Concrete Technologies: Java/Swing

- Beyond Swing:
 - Advanced Toolkits
 - Accessibility API
 - 2D - API
 - Drag-and-Drop API
- Java Foundation Classes Library (JFC)

Concrete Technologies: SmlTk

Overall Architecture



Concrete Technologies: SmlTk

- Topmost Type: Windows(->Java: JFrame)

```
type Window = WinId * string * Widget list * (unit-> unit)
```

- Widgets (Graphical Objects)(->Java:JComponent)

```
datatype Widget =  
  Frame    of WidId * Widget list * Pack list * Configure list * Binding list  
| Label    of WidId * Pack list * Configure list * Binding list  
| Button   of WidId * Pack list * Configure list * Binding list  
| Entry    of WidId * Pack list * Configure list * Binding list  
| Listbox  of WidId * ScrollType * Pack list * Configure list * Binding list  
| Menubutton of WidId * bool * MItem list * Pack list * Configure list * Binding list  
| Entry     of WidId * Pack list * Configure list * Binding list  
| TextWid  of WidId * ScrollType * AnnoText * Pack list * Configure list * Binding list  
| Canvas   of WidId * ScrollType * CItem list * Pack list * Configure list * Binding list  
| . . .
```

Concrete Technologies: SmlTk

- Configurations (Java: -> "View")

```
datatype Configure =
  Width      of int      (* -width      <val> *)
| Height     of int      (* -height     <val> *)
| Borderwidth of int      (* -borderwidth <val> *)
| Relief     of RelKind   (* -relief     <val> *)
| Foreground of Color     (* -foreground <val> *)
| Background of Color     (* -background <val> *)
| Text       of string    (* -label      <val> *)
| Font       of Font      (* -font       <val> *)
| Variable   of string    (* -variable   <val> *)
| Value      of string    (* -value      <val> *)
| Icon       of IconKind  (* -bitmap or -image ... *)
| Cursor     of CursorKind (* -cursor     <val> *)
| Command    of SimpleAction (* -command    <val> *)
| Anchor     of AnchorKind (* -anchor     <val> *)
| FillColor  of Color     (* -fill       <val> *)
| Outline    of Color     (* -outline    <val> *)
| OutlineWidth of int      (* -width      <val> *)
| . . .
```

- Bindings (Java: -> Listeners; "Control")

```
datatype Binding = BindAct of Event * Action
```

Concrete Technologies: SmlTk

- Events (Java: -> low level events)

```
datatype Event =
    KeyPress    of string
  | KeyRelease of string
  | ButtonPress  of int Option.option
  | ButtonRelease of int Option.option
  | Enter  | Leave  | Motion
  | UserEv of string
  | Shift of Event  | Ctrl of Event  | Lock of Event  | Any of Event
  | Double of Event | Triple of Event
  | ModButton of int* Event
  | Alt of Event  | Meta of Event
  | Mod3 of Event  | Mod4 of Event  | Mod5 of Event
```

- Packing (Java: -> Layout Management)

```
datatype Edge      = Top | Bottom | Left | Right
datatype Style     = X  | Y  | Both
datatype Pack      = Expand of bool | Fill of Style | PadX of int
                   | PadY of int  | Side of Edge
```


Concrete Technologies: SmlTk

The running example in SmlTk:

```
structure SmallExample : sig main : unit -> unit end =  
struct  
  val mainID = mainWindowId()  
  val entID = newWidgetId()  
  val label = Label(newWidgetId(), [Side Left], [Text "name:"], [])  
  val input = let fun endInput _ = changeTitle mainID (readTextAll entID)  
                in Entry(entID, [], [Width 20], [Bind("", endInput)]) end  
  val quit = let fun stop _ = closeWindow mainID  
                in Button(newWidgetId(), [Side Bottom],  
                          [Text "Quit", Command stop], []) end  
  val topblock = Frame(newWidgetId(), [label, input], [Side Top], [], [])  
  val enterwin = (mainID, "Please enter name", [topblock, quit], fn _ => ())  
  
  fun main ()= startTcl [enterwin];  
end;
```



⇒ Quite compact !!!

Conclusion

- Design, Analysis of GUI's can be amazingly complex!
- Open problems in Analysis and Validation (while basics are clear and simple)
- Java/Swing:
 - Data-Model for Widgets neatly organized via inheritance
 - MVC, Observer Pattern (in events) somewhat awkward
 - threads problematic in current implementation
- Functional Programming and GUI's:
 - **no problem** (even in pure languages like Haskell/Tk)
 - ... while libraries are clearly more limited as Java/Swing

Bibliography

- [Dix&al 89] A. Dix, J. Finlay, G. Abowd, R. Beale: Human Computer Interaction (2.ed.). Prentice Hall, 1998.
- [Shn 98] B. Shneiderman: Designing the User-Interface - Strategies for Effective Human-Computer Interaction (3.ed). Addison Wesley, 1998.
- [HC 99] C.S. Horstmann, G. Cornell: Core Java. Volume I-II. SUN MICROSYSTEM PRESS, Prentice Hall. 1999.
- [Swing] http://www.amazon.com/exec/obidos/ISBN%3D0201433214/javasoftsunmicroA/002-8104625_2161664
<http://java.sun.com/docs/books/tutorial/uiswing/index.html>

Bibliography

[smltk 01] C. Lüth, B. Wolff:
sml_tk: Functional Programming for Graphical
User Interfaces
http://www.informatik.uni-bremen.de/~cxl/sml_tk

[Haskell/Tk] <http://www.informatik.uni-bremen.de/~ewk/WB.html>

softtech

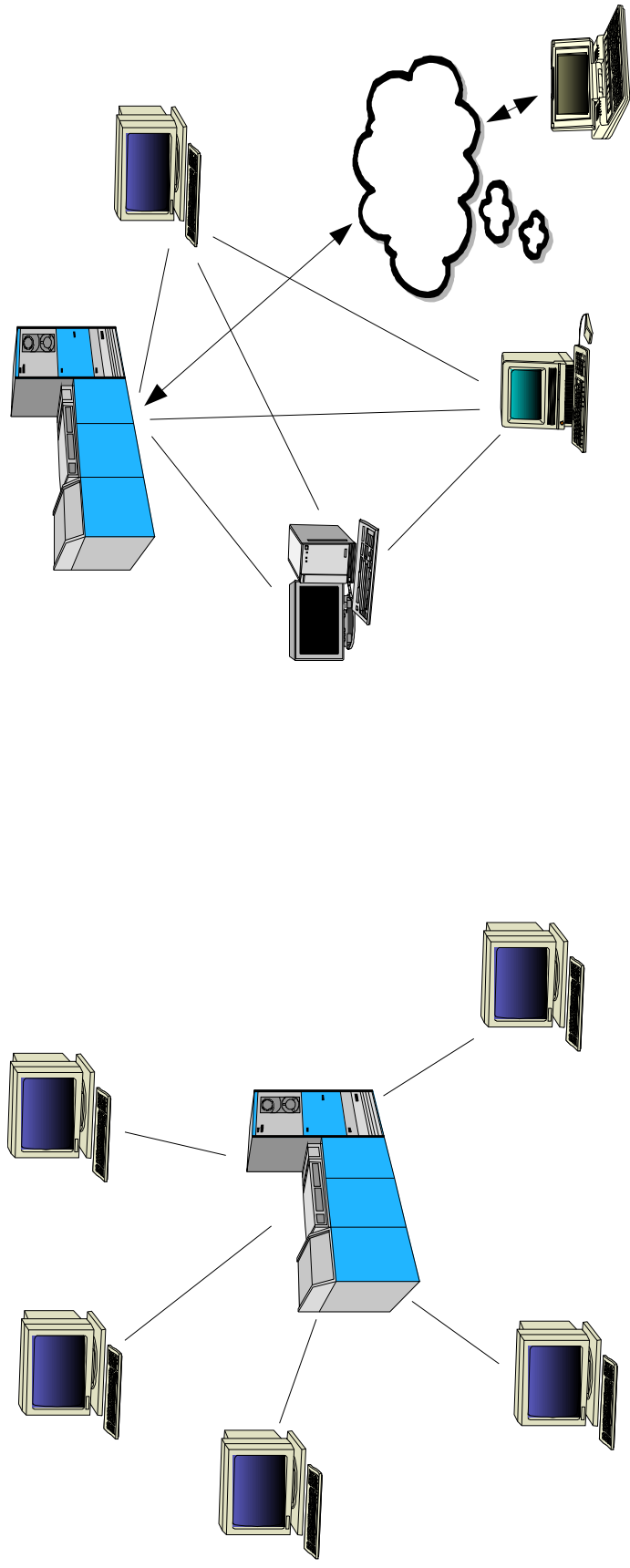


CORBA meets Software Engineering

Frank Rittinger

Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Distributed Systems



Distributed Object Systems

- Objects can be server or/and clients
 - Objects can use services of other objects
 - Objects can be created and deleted dynamically
 - Objects encapsulate data and algorithms
 - Complex communication between objects
- complex programming task

Middleware

- Universal communication platform
- Access and location transparency
- Provides general services
- OO concept used to encapsulate different systems (integration of legacy systems)
- Examples: COM+/.NET, EJB, CORBA

Common Object Request Broker Architecture --- CORBA

- Object Management Group (OMG)
 - Consortium
 - ~800 members
 - Use existing technologies
 - Specifications: UML, XMI, CORBA
- Architectural standard
- Platform and language independent: IDL
- Services and facilities
- Self-describing system
- Dynamic method invocations

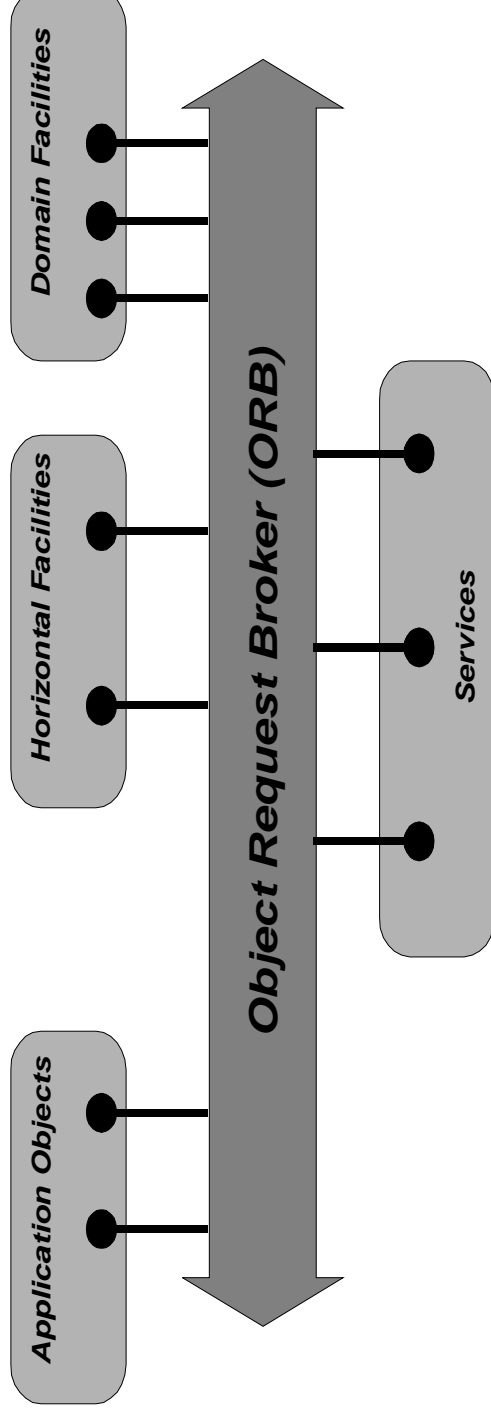
Defining Objects: IDL

- Separation of interface and implementation
- C++/Java syntax
- No control constructs
- Features:
 - Modules, interfaces
 - Multiple inheritance
 - Arrays and sequences
 - Exceptions
 - Basic types, struct, enum, typedef

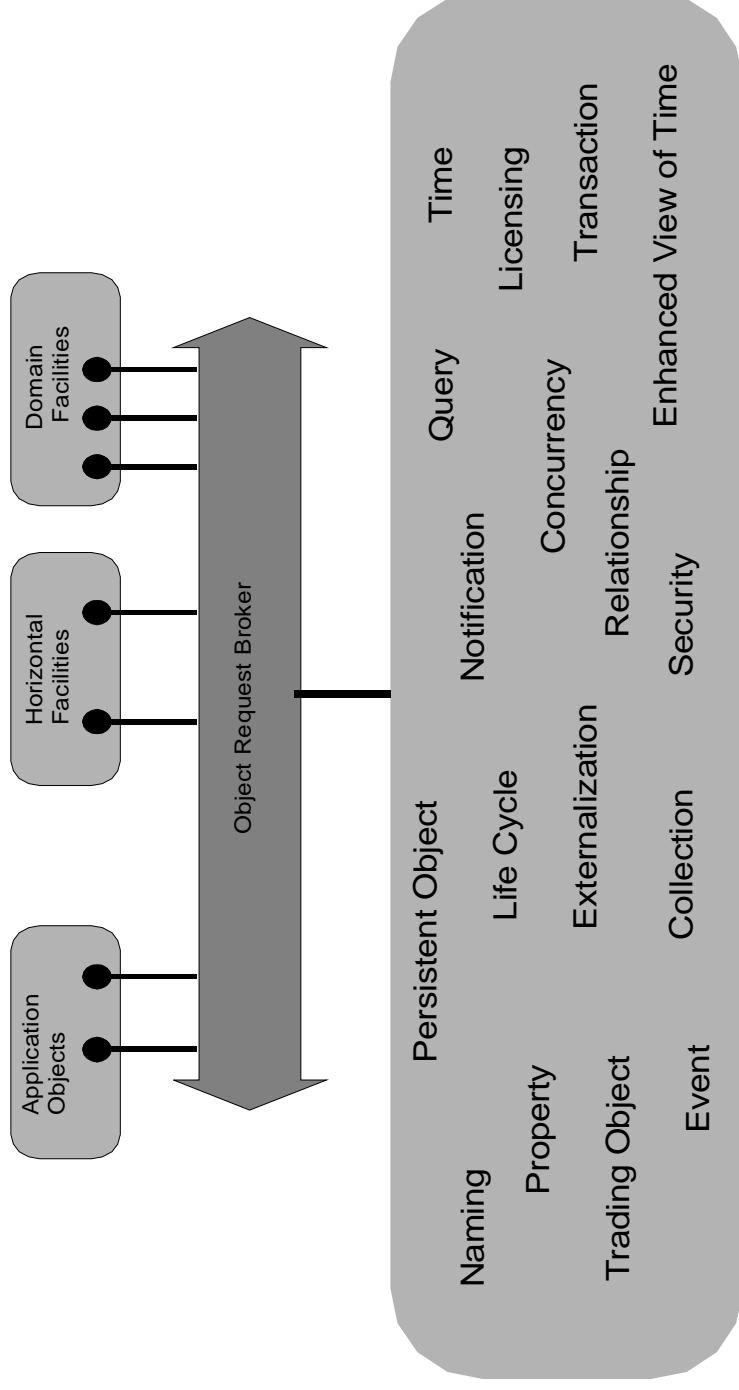
```
/*  
 * Interface for a distributed  
 * Time Service.  
 */  
  
enum Day {Mo, Tu, We, Th, Fr, Sa, Su}  
  
struct Time {  
    unsigned short minute;  
    unsigned short hour;  
    Day day;  
    unsigned short date;  
    unsigned short month;  
    unsigned short year;  
};  
  
interface TimeService {  
    Time getLocalTime();  
};
```

Object Management Architecture

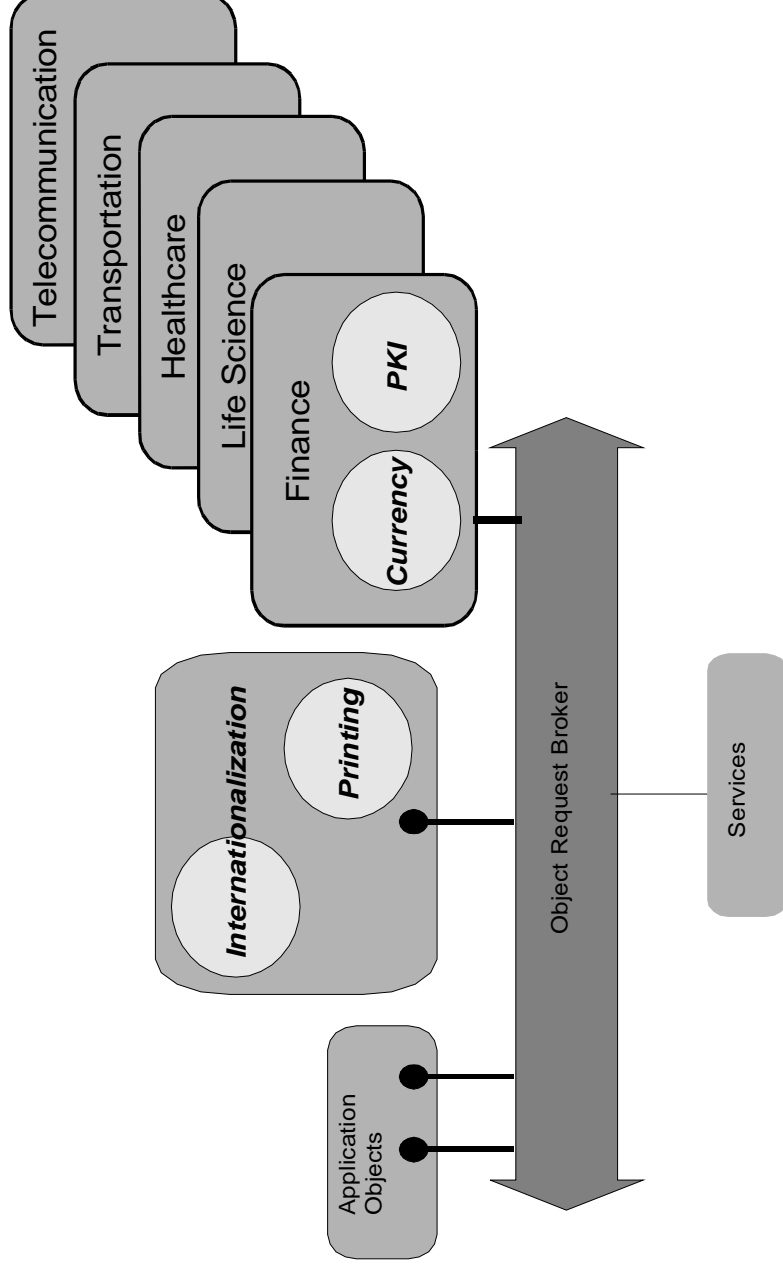
- **Object model** properties of objects
- **Reference model** interactions between objects



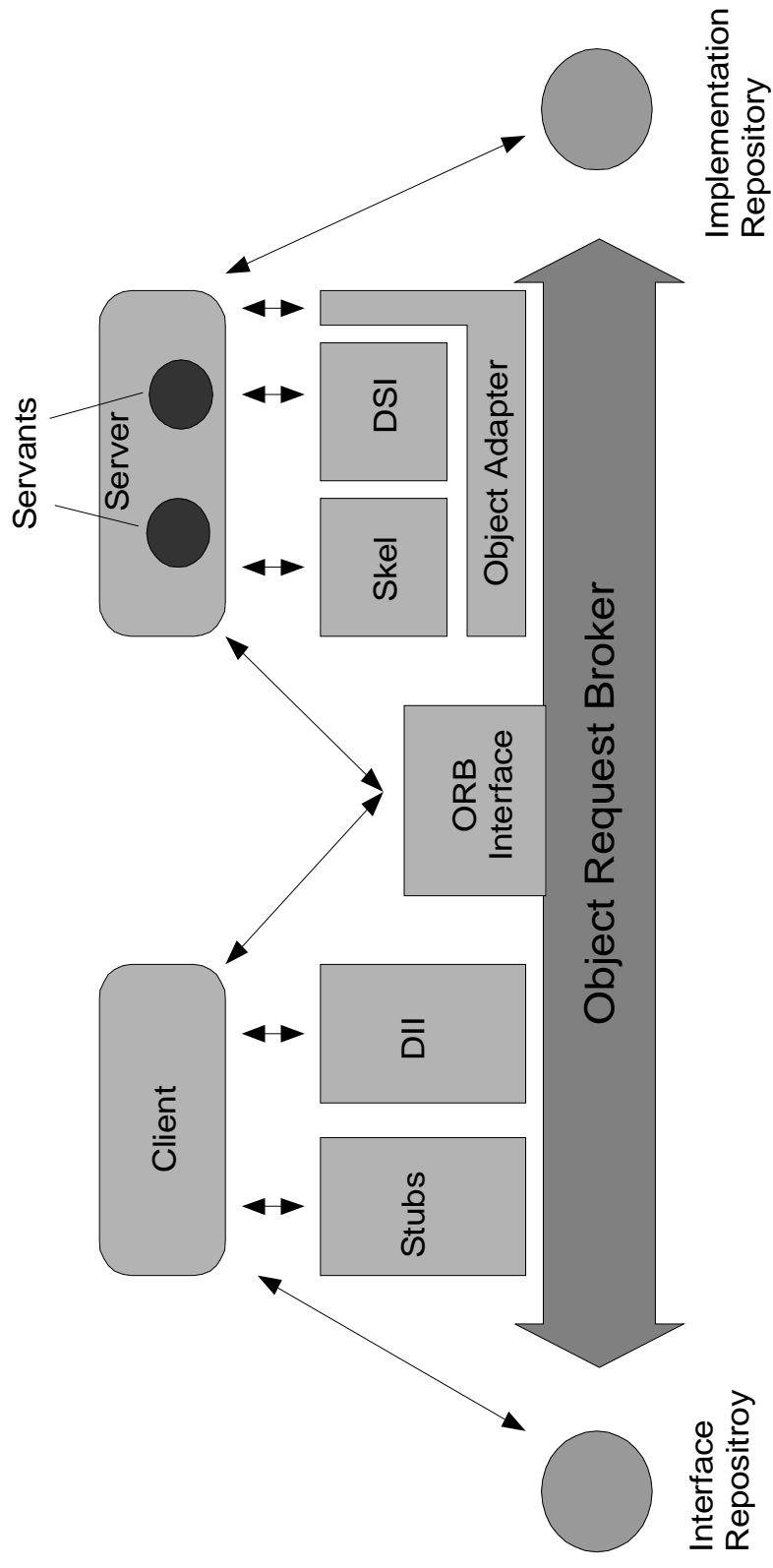
Services



Facilities



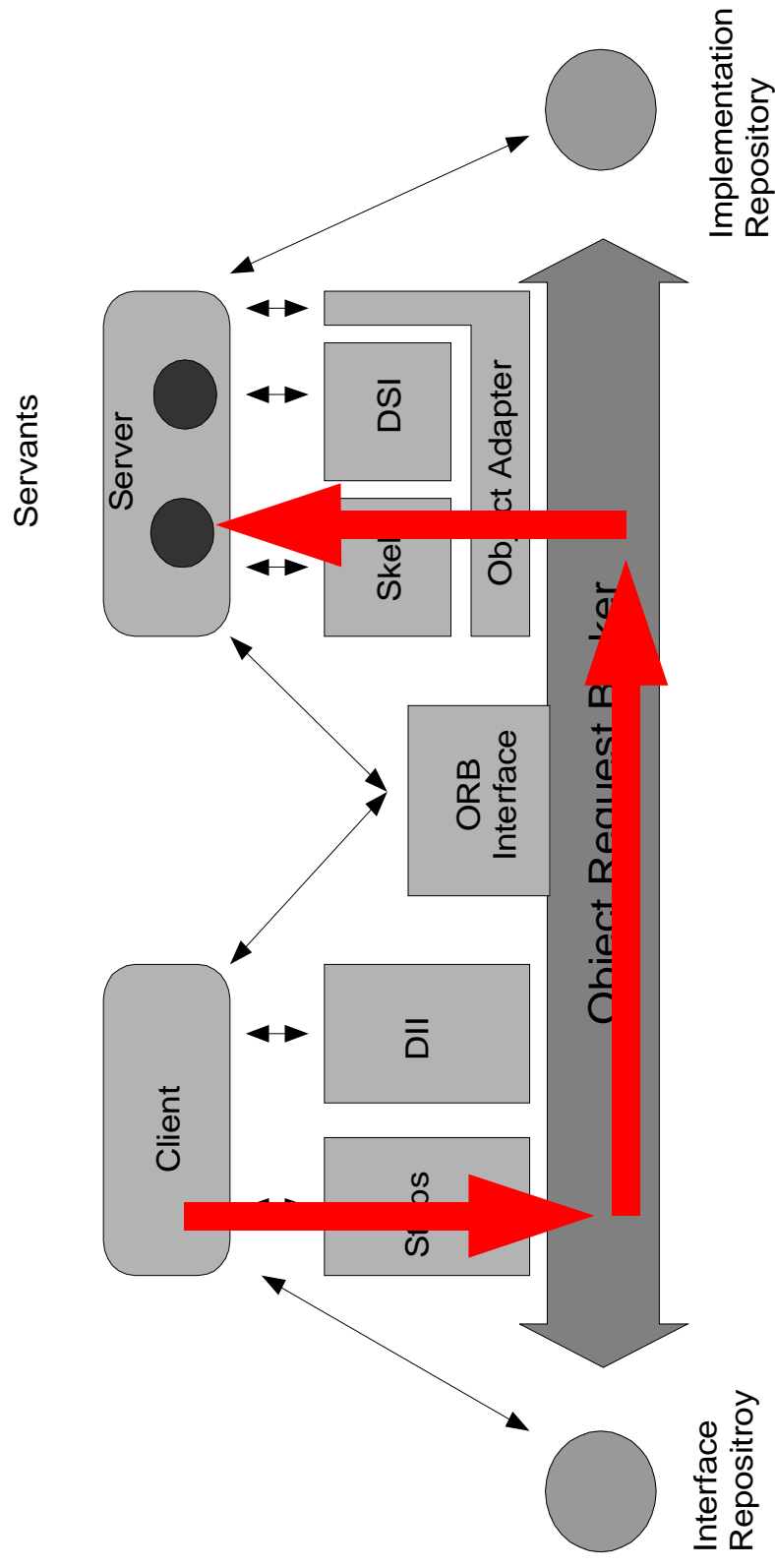
Overview of CORBA



Object Request Broker - ORB

- Object bus which hides details
- Tasks:
 - Registration of object invocations
 - Finding objects
 - Marshalling of parameters
 - Method invocation
 - Marshalling of results
- Implementations: library, process, os

Object Invocation

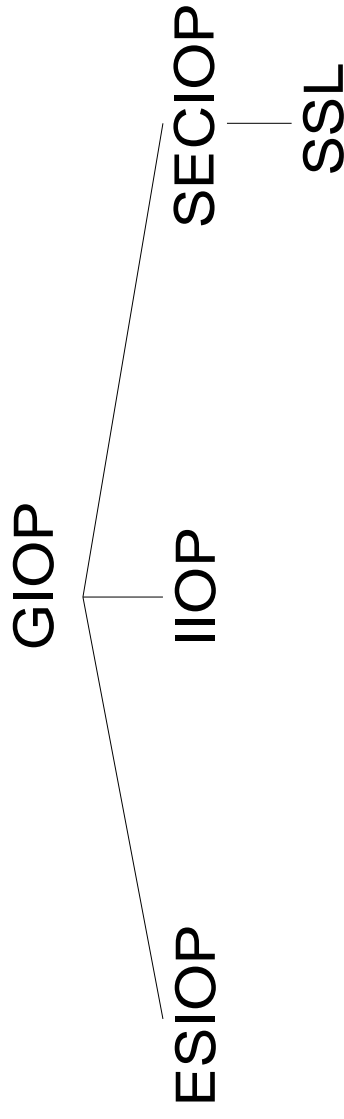


Static vs. Dynamic

- Stubs and Skeletons contain marshaling code
- Static type checking
- All known at compile time
- Simple
- Efficient
- Construction at run time
- Metadata in repository
- Type check only metadata
- Complex
- Flexible

Interoperability

- CORBA 1.0 - no interoperability
- GIOP defined in IDL
- Exchange basic types, references or objects
- IOR = repository ID + endpoint + object key



Software Development in the large

- Abstraction
- Structure
- Interfaces
- Reuse

CORBA provides the mechanisms and technology to define the building blocks and to combine them while ignoring the technicalities.

A Step Further: Components

- Known from all engineering disciplines
- Higher abstraction level than objects
- COTS: commercial of the shelf
- Faster and cheaper development
- CCM: CORBA Component Model
 - Extends IDL
 - Focus on configuration and integration
 - But, not yet done.

Advanced CORBA

- CASE: iO ArcStyler generates CORBA code
- Formal analysis of CORBA:
 - CORBAsec model in Z
 - Model checking of GIOP with SPIN
- GNOME uses CORBA
- UBS: 400 Unix servers and 40000 PCs worldwide
- Thames Water: in the office and in the field
- CORBA on a PDA in a hospital

Further Reading

- OMG: <http://www.omg.org>
- ORBs: http://www.cetuslinks.org/oo_corba.html
- Java programming with CORBA.
Andreas Vogel. CD/1.5/62.
- The CORBA reference guide.
Alan Pope. CD/1.5/49
- Formal specifications of CORBA:
 - Z model of CORBA Security (Frank Rittinger)
 - SPIN analysis of GIOP (Prof. Leue)



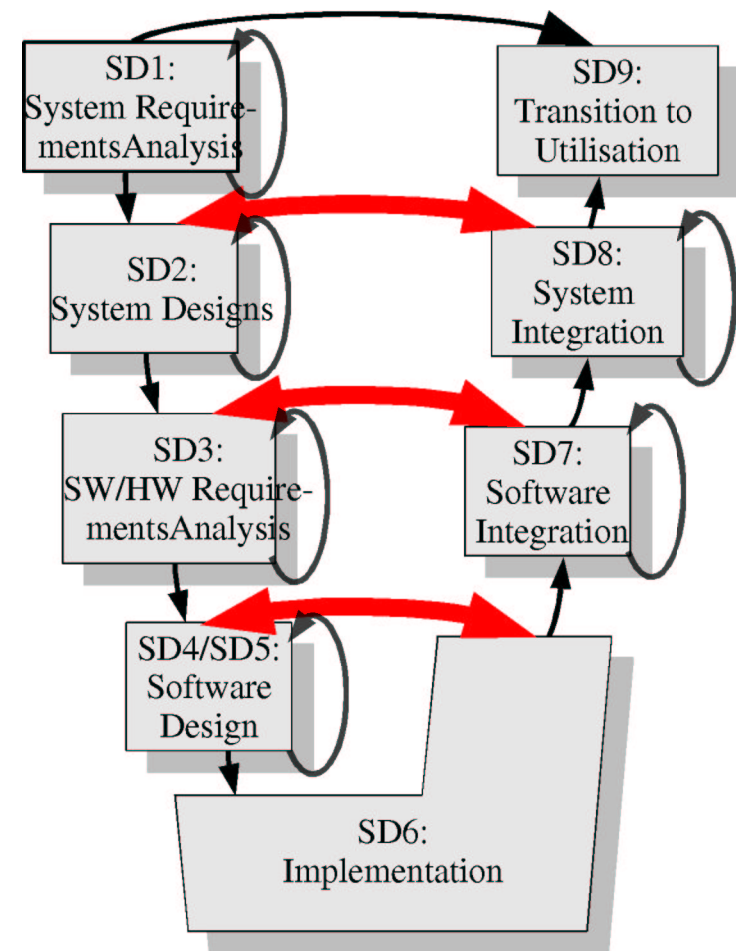
Validation of Software Systems (1)

Burkhart Wolff

AG Softwaretechnik

Overview

- Where are we in the Development Process?
- Validation =
"Are we building the right product?"
"Are we building the product right?"
- Validation of
 - Implementation
 - Software-Integration
 - System-Integration



Overview

- Three Dimensions of the Problem:

- Validation techniques
(verification(1), inspection(2), **testing(3)**, . . .)



- Validation levels
(again: System-Integration(1), Software-Integration(2), **Implementation(3)**)



- Validation targets
(GUI(1), **functional kernel(2)**, component interaction(3),
system interaction with the environment(4))



- **These talks: emphasis on testing functional kernels**
- **Note: not all of these combinations are meaningful!**

T	L	T
3	3	2

revisited

Modeling and Validating GUI's

- System Integration Level:

- techniques:

- inspection,
 - ergonomic metrics (GOMS-model) and *usability criteria*,
 - non-functional requirements check (e.g. timing, stability under stress, . . .)

2	1	1
---	---	---

- Software Integration Level



- by hand
 - by replay
 - by generated **replay-scripts** (Test-Case-Generation)
 - by **verification**

3	2	1
---	---	---

1	3	1
---	---	---


revisited

Validation of GUI's

- Test-Case-Generation
Path-Generation from StateChart,
Test-program based on Dummy-Kernel 
- Test-Case-Generation based on Data-Model
for System-Integration Test 

Validation of GUI's: by Verification

revisited

- Formalizing the GUI-Event-Model 
 - in temporal logic
 - in process algebra formalisms such as CSP (concurrent sequential processes, Hoare/Roscoe)
- Formalizing Design Goals
- Proving *Refinement* of GUI-Event-Model

Verification and Transformation

1	2	2
---	---	---

1	3	2
---	---	---

- Idea: Using Logical Rules to Establish Equivalence Between Formal Specification and Implementation
 - post-verification approach:

given: program, specification
procedure: show (interactively!) equivalence in some logic (Hoare-Logic, Dynamic Logic, PL1, HOL, Z, . . .)

Verification and Transformation

1	2	2
---	---	---

1	3	2
---	---	---

- variant of post-verification:
development by refinement

given: sequence of specifications S_1, \dots, S_n

procedure: show (semi-interactively): S_{i+1} *refines* S_i
for some Refinementrelation *refines*.

generate automatically code from S_n .

examples: Atelier B (Z-logic),
KIV (Dynamic Logic),
TkWinHOL, IsaWin

(Back's Refinement Calculus)

Verification and Transformation

1	2	2
---	---	---

1	3	2
---	---	---

- Evaluation:
 - advantage: Applicable (in principle!) on
 - software-integration,
 - software-architecture,
 - algorithms.
 - Can produce the highest degree of quality
 - disadvantage:
 - (at least intellectually:) *very expensive*
 - automatic support still active research area
 - it is still difficult to integrate full-blown verification SE-process
 - reserved to mission critical or safety critical applications

Verification and Transformation

1	2	2
---	---	---

1	3	2
---	---	---

– transformational approach:

given: specifications S_1

procedure: apply (highly interactively)
transformation rules (like DivideAndConquer)
produce sequence of specifications such

that:

$$S_{i+1} \text{ refines } S_i$$

for some refinement relation *refines*.

generate automatically code from S_n .

examples: TkWinHOL, KIDS, TAS, CIP, . . .

Validation of Code by Inspection



- Idea: Check a program by somebody else . . .
 - programming team makes system release
 - group of reviewers (*peers*) checks several aspects of system by line-by-line code review
 - coordination meetings discussing errors found (no solutions discussed)
 - result: protocol over errors and statistics
- ⇒ system corrected by programming team
- ⇒ precondition: inspections must be planned and peers must be independent

Inspection: A Checklist

DATA REFERENCE

1. Unset variables used?
2. Subscripts within bounds?
3. Noninteger subscripts?
4. Dangling references?
5. Correct attributes when aliasing?
6. Record and structure attributes match?
7. Computing addresses of bit strings?
Passing bit-string arguments?
8. Based storage attributes correct?
9. Structure definitions match across procedures?
10. String limits exceeded?
11. Off-by-one errors in indexing or subscripting operations?

DATA DECLARATION

1. All variables declared?
2. Default attributes understood?
3. Arrays and strings initialized properly?
4. Correct lengths, types, and storage classes assigned?
5. Initialization consistent with storage class?
6. Any variables with similar names?

COMPUTATION

1. Computations on nonarithmetic variables?
2. Mixed-mode computations?
3. Computations on variables of different lengths?
4. Target size less than size of assigned value?
5. Intermediate result overflow or underflow?
6. Division by zero?
7. Base-2 inaccuracies?
8. Variable's value outside of meaningful range?
9. Operator precedence understood?
10. Integer divisions correct?

COMPARISON

1. Comparisons between inconsistent variables?
2. Mixed-mode comparisons?
3. Comparison relationships correct?
4. Boolean expressions correct?
5. Comparison and Boolean expressions mixed?
6. Comparisons of base-2 fractional values?
7. Operator precedence understood?
8. Compiler evaluation of Boolean expressions understood?

Inspection: A Checklist

CONTROL FLOW

1. Multiway branches exceeded?
2. Will each loop terminate?
3. Will program terminate?
4. Any loop bypasses because of entry conditions?
5. Are possible loop fallthroughs correct?
6. Off-by-one iteration errors?
7. DO/END statements match?
8. Any nonexhaustive decisions?

INTERFACES

1. Number of input parameters equal to number of arguments?
2. Parameter and argument attributes match?
3. Parameter and argument units system match?
4. Number of arguments transmitted to called modules equal to number of parameters?
5. Attributes of arguments transmitted to called modules equal to attributes of parameters?
6. Units system of arguments transmitted to called modules equal to units system of parameters?
7. Number, attributes, and order of arguments to built-in functions correct?
8. Any references to parameters not associated with current point of entry?
9. Input-only arguments altered?
10. Global variable definitions consistent across modules?
11. Constants passed as arguments?

INPUT/OUTPUT

1. File attributes correct?
2. OPEN statements correct?
3. Format specification matches I/O statement?
4. Buffer size matches record size?
5. Files opened before use?
6. End-of-file conditions handled?
7. I/O errors handled?
8. Any textual errors in output information?

OTHER CHECKS

1. Any unreferenced variables in cross-reference listing?
2. Attribute list what was expected?
3. Any warning or informational messages?
4. Input checked for validity?
5. Missing function?

Validation of Code by Inspection



- Evaluation:
 - important: psychological factors, experience of peers . . .
 - empirical data:
 - costs: 15% - 20% of development costs
 - 60%-70% of errors can be found (really ???)
 - relative high "return of investment"
 - problems:
 - results depends on subjective measures and personal form,
 - does not scale up to large systems,
 - relatively high costs,
 - inspections are sometimes obstacle for improvements.

Validation of Code by *Automated* Inspection

- Idea:

Using static analysis techniques for finding *dangerous constructs*

- type checking
- checking conformance to documentation and format guidelines
- checking absence of "unusual" data-flows or control-flows
- software-metrics
(e.g. McCabes Cyclometric Numbers, etc.)

Validation of Code by *Automated* Inspection

- Evaluation:
 - advantage:
 - automatic,
 - can be used by programmers
 - disadvantage:
 - does not use semantics of a program
(perfectly typed programs can be completely buggy . . .)
 - software metrics: foundation controversial . . .

Testing



- General Remarks. Testing is:
 - *the* most used quality assurance technique
 - often used in an unsystematic way
 - (potentially) *complementary* to formal specifications
 - if assumptions on the environment of a system have to be made (test as experiment)
 - if specifications model "the wrong thing", testing may tell us ...
 - very expensive
 - 50 % of cost
 - 50 % of development time (Myers, *Art of Software Testing*)

Testing



- General Remarks.




Testing is:

- not really **replacing** verification

Program testing can be used to show the presence of bugs, but never to show their absence (Dijkstra)

- admittedly a way to increase the **trustworthiness**
- based on sometimes not fully understood **heuristics**

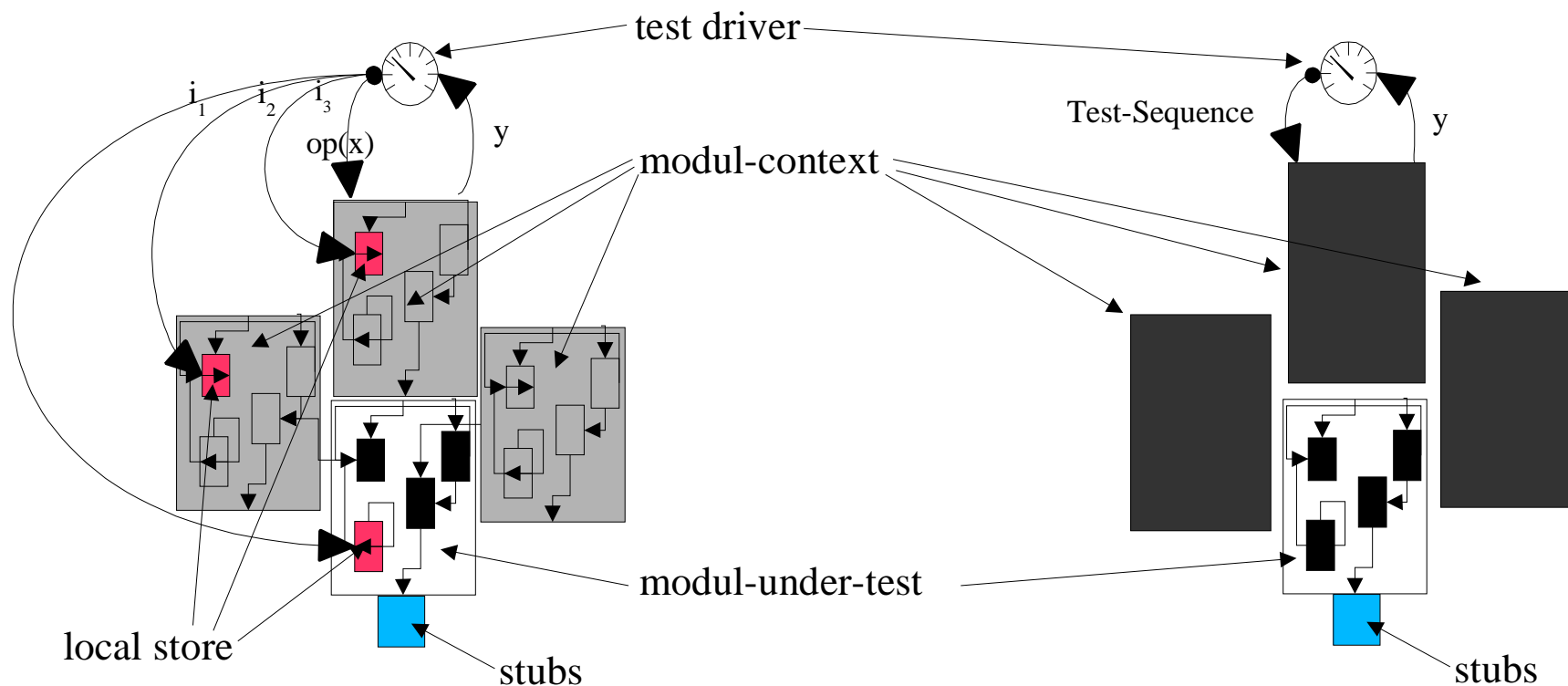
Testing

- Along the process-model-levels, we distinguish:
 - acceptance test
usability, alpha-test, beta-test, cf. slide 5. 
 - integration test
testing along the interfaces of modules
testing the interaction of the components. 
(goal: check if preconditions of subcomponents are respected.
"Normal" system input should not produce precondition violations that result in overall "exceptional" behaviour)
 - unit test
tests of one function or module. 
(goal: test cases for "normal" and "exceptional" beh.)

Integration Test



- Principles: Init-Access-Testing vs. Sequence Testing



Test: $\text{init}(i_1, i_2, i_3); \text{op}(x); \text{check}(\text{get } y)$

Test: $\text{op}_1(x_1); \dots; \text{op}_n(x_n), \text{check}(\text{get } y)$

Integration Test



- Evaluation:
 - the init-acces-approach needs **access to all internal states**.
this is sometimes neither possible (source unknown)
nor desirable (complexity! Against encapsulation of state!).
however, this approach is comparatively **simple**
(provided data for unit tests is available).
 - the sequence-test approach requires no internal knowledge
over modules - but may be more **difficult** to perform . . .
C.f. testing of hardware components . . .
 - both approaches **depend on** test-cases for
exceptional behaviour from **unit tests**.

Unit Test: An Introductory Example.

- The problem:



A program reads 3 integers. Their values represent the length of the 3 edges of a triangle. The program decides if the triangle is

- equilateral ("gleichseitig")
- isosceles ("gleichschenkelig")
- scalene ("ungleichseitig")

Unit Test: Introductory Example

1. Do you have a test case that represents a *valid* scalene triangle? (Note that test cases such as 1,2,3 and 2,5,10 do not warrant a “yes” answer, because there does not exist a triangle having such sides.)
2. Do you have a test case that represents a valid equilateral triangle?
3. Do you have a test case that represents a valid isosceles triangle? (A test case specifying 2,2,4 would not be counted.)
4. Do you have at least three test cases that represent valid isosceles triangles such that you have tried all three permutations of two equal sides (e.g., 3,3,4; 3,4,3; and 4,3,3)?
5. Do you have a test case in which one side has a zero value?
6. Do you have a test case in which one side has a negative value?
7. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is equal to the third? (That is, if the program said that 1,2,3 represents a scalene triangle, it would contain a bug.)
8. Do you have at least three test cases in category 7 such that you have tried all three permutations where the length of one side is equal to the sum of the lengths of the other two sides (e.g., 1,2,3; 1,3,2; and 3,1,2)?
9. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is less than the third (e.g., 1,2,4 or 12,15,30)?
10. Do you have at least three test cases in category 9 such that you have tried all three permutations (e.g., 1,2,4; 1,4,2; and 4,1,2)?
11. Do you have a test case in which all sides are 0 (i.e., 0,0,0)?
12. Do you have at least one test case specifying noninteger values?
13. Do you have at least one test case specifying the wrong number of values (e.g., two, rather than three, integers)?
14. For each test case, did you specify the expected output from the program in addition to the input values?

Unit Test

- Hmm, could this task be accomplished

systematically ?

- Even better, could this task be accomplished

automatically ?

How To Talk about "Testing"



- Approach: State-Of-The-Art **Systems**
- Approach: Testing **Techniques**
- Approach: Test-Case Adequacy **Criteria**

– instead of

"how to get a test set"

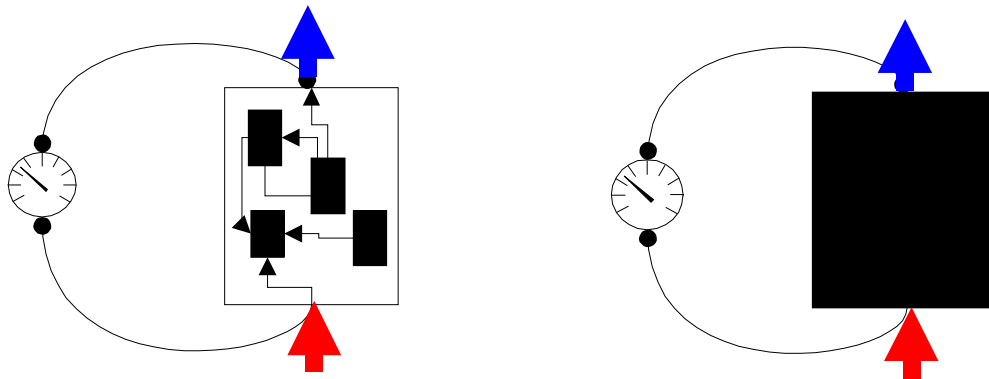
we ask

"how to get a test set right?"

When do we have tested "enough"?

Testing Techniques: Two Classes

White-Box vs. Black-Box Testing Techniques



A "Finer" Classification of Test Adequacy Criteria

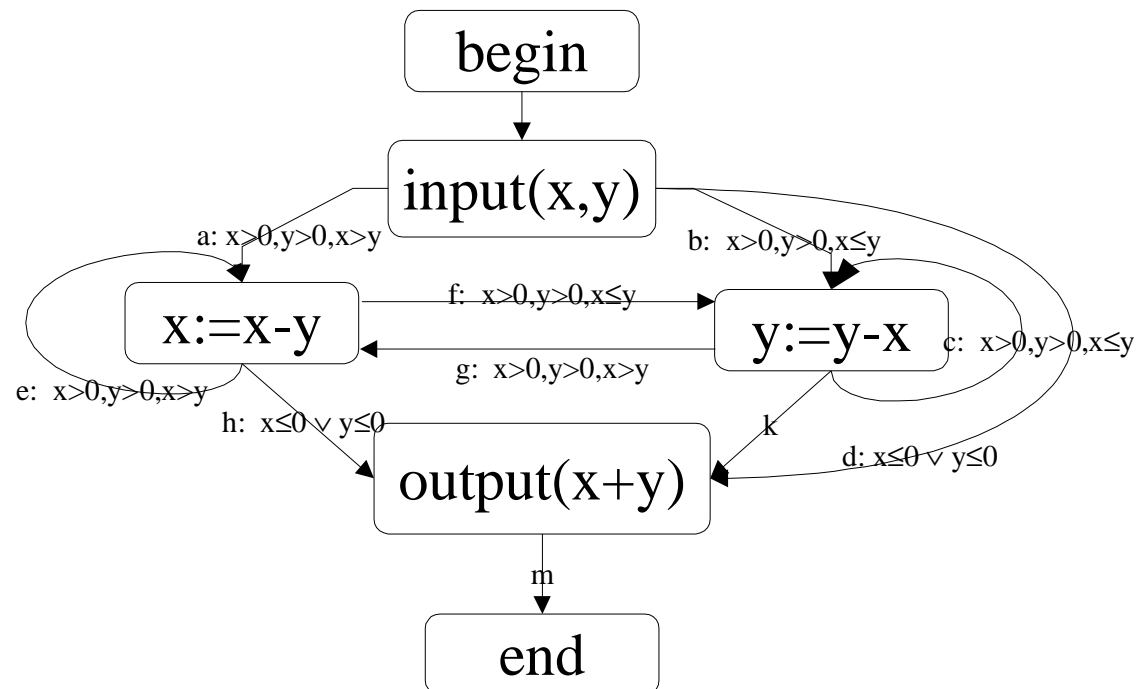
- Following [ZHM 97], we use adequacy criteria. We distinguish criteria for:
 - structural testing
 - "Couverage of a particular *part* of the structure of the program or specification"
 - fault -based testing
 - "focus on detecting *faults* of programs"
 - error -based testing
 - "focus on *errors* produced by programs"

Program-Based Structural Testing (control flow based)

```

begin
  input(x,y);
  while(x>0 and y > 0) do
    if (x>y) then x := x-y
      else y := y-x
    endif
  endwhile;
  output (x+y)
end;

```



Program-Based Structural Testing (control flow based)

Test Cases following Statement Covering Method
(generated "by hand"):

Path	x	y	output
d/k	0	3	3
b/c/k	1	2	1
a/e/h	2	1	1

*How can the generation of x and y be automated?
(output MUST BE CHECKED BY "HIGHER INSIGHT" !!!)*

Program-Based Structural Testing (control flow based)

Idea:

conjoin the path formulas and negated non-path-formulas (see below),
apply substitution according to assignment rule in Hoare-Logic, simplify,
choose ground substitution (if none, path impossible).

Example for path b/c/k:

$$\begin{aligned}
 & x > 0 \wedge y > 0 \wedge x \leq y \\
 & \wedge (x > 0 \wedge y > 0 \wedge x \leq y) [y := y-x] \\
 & \wedge \neg (x > 0 \wedge y > 0 \wedge x > y) [y := y-x][y := y-x] \text{ (* negated g branch *)} \\
 \equiv & \quad x > 0 \wedge y > 0 \wedge x \leq y \\
 & \wedge x > 0 \wedge y > x \wedge 2x \leq y \\
 & \wedge \neg x > 0 \vee \neg y > 2x \vee \neg 3x > y \\
 \equiv & \quad x > 0 \wedge y > 0 \wedge 2x \leq y \wedge y \leq 3x
 \end{aligned}$$

Solution: $x=1, y=2$.

Program-Based Structural Testing (control flow based)

- Criteria
 - statement coverage
 - branch coverage
 - path coverage w.r.t. some set of paths P
 - cyclomatic-number criterion (McCabe)
 - multiple condition coverage
- Evaluation:
 - potentially fully automatic
 - uses well-known techniques from compiler construction
 - foundation semanticless,
sometimes heuristic, sometimes metric-mystic.

Program-Based Structural Testing (data flow based)

- Analogously: Dataflow Graphs (AST)

"based on variables **occurring** within a program, let it be as **definition** or **use**".

- Criteria:

- all definitions
- all uses
- k-tuples, k-dr interaction, context coverage ...
- interprocedural data flow
- dependence coverage ...

Program-Based Structural Testing (data flow based)

- Evaluation

- variable occurrence: access to **truly semantic properties**
 - constant functions
 - data independence
 - check for "hidden states" possible
 - automatic
 - **problems** with aliases and pointers
- ⇒ **no practical relevance**
for programs in the large and in C⁺⁺

Specification-Based "Structural" Testing

- Gaudel, Dick/Faivre, Santen, Stepney ...

as we will see next week ...

- $\text{TestSet} : \mathbb{P}(I \times O) = I \rightarrow \mathbb{P}O = \{(i:I, f:O \rightarrow \text{bool})\}$

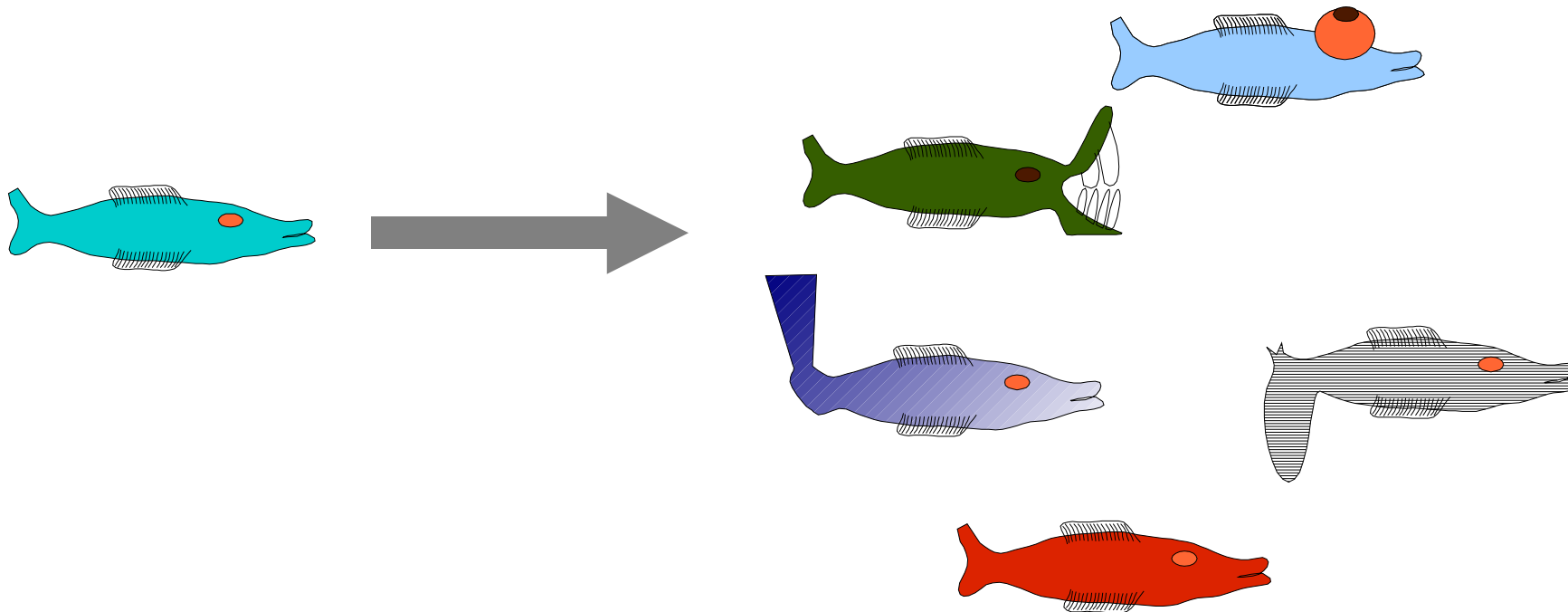
TestSet's equivalent to pairs of
input I and *oracle functions* f

Fault-Based Adequacy Criteria

- Error Seeding
- Mutant Generation

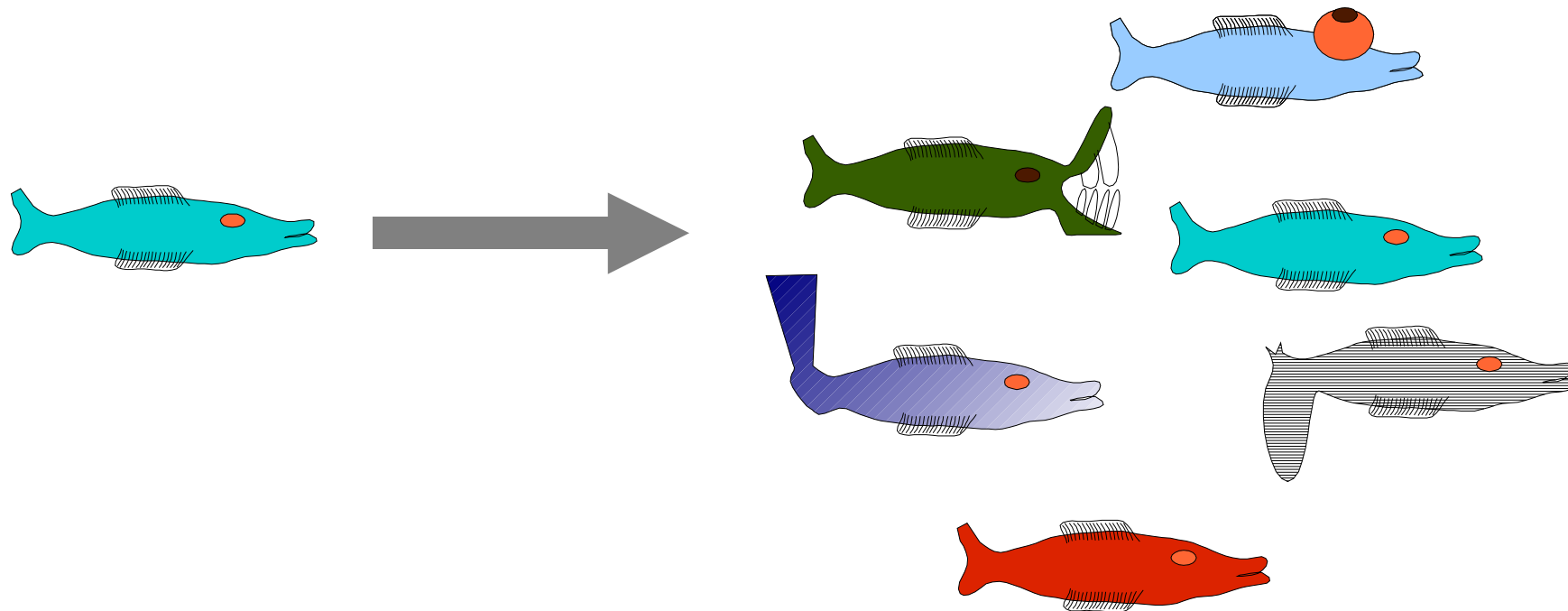
Fault-Based Adequacy Criteria

- Error Seeding
- Mutant Generation



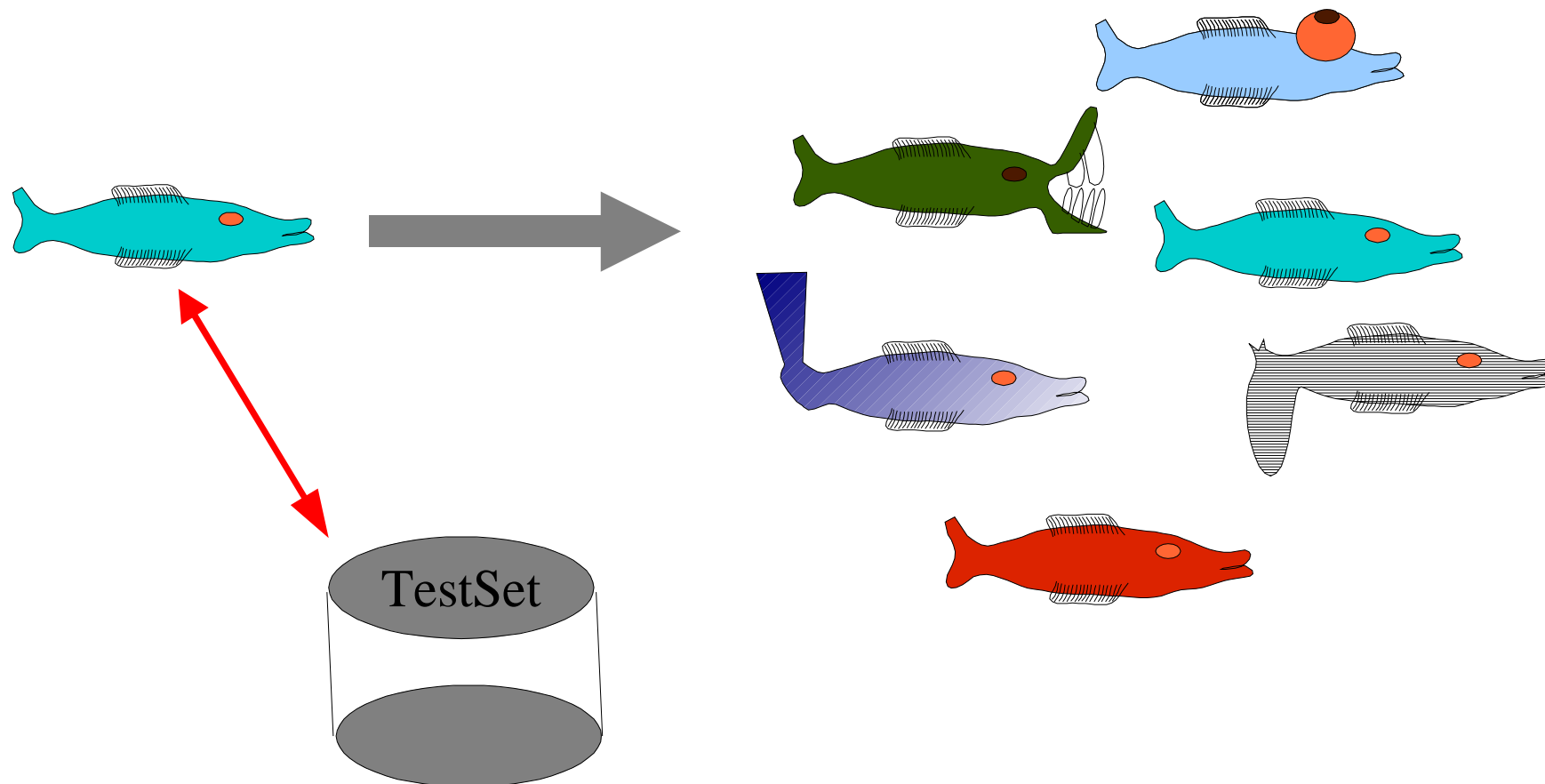
Fault-Based Adequacy Criteria

- Error Seeding
- Mutant Generation



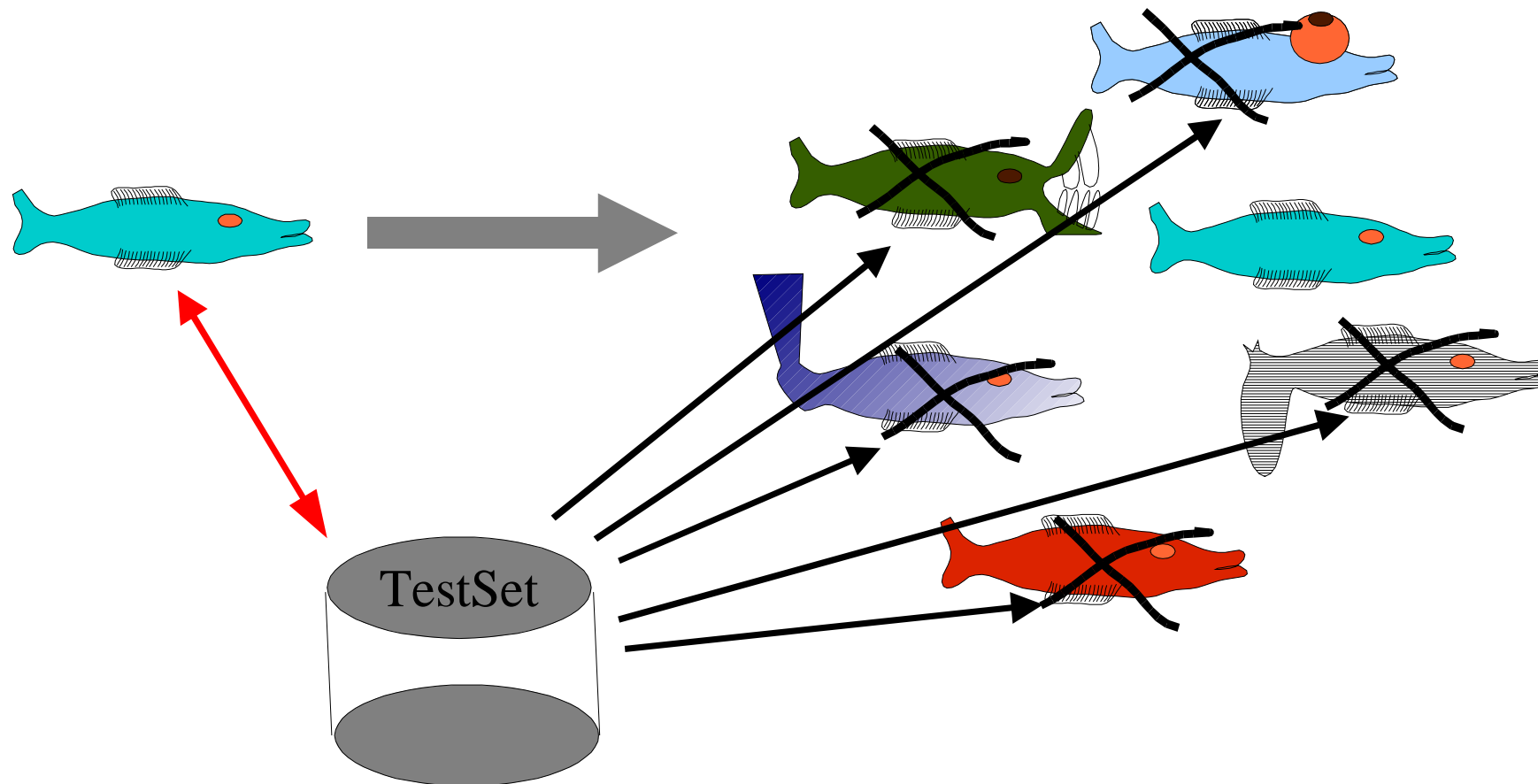
Fault-Based Adequacy Criteria

- Error Seeding
- Mutant Generation



Fault-Based Adequacy Criteria

- Error Seeding
- Mutant Generation



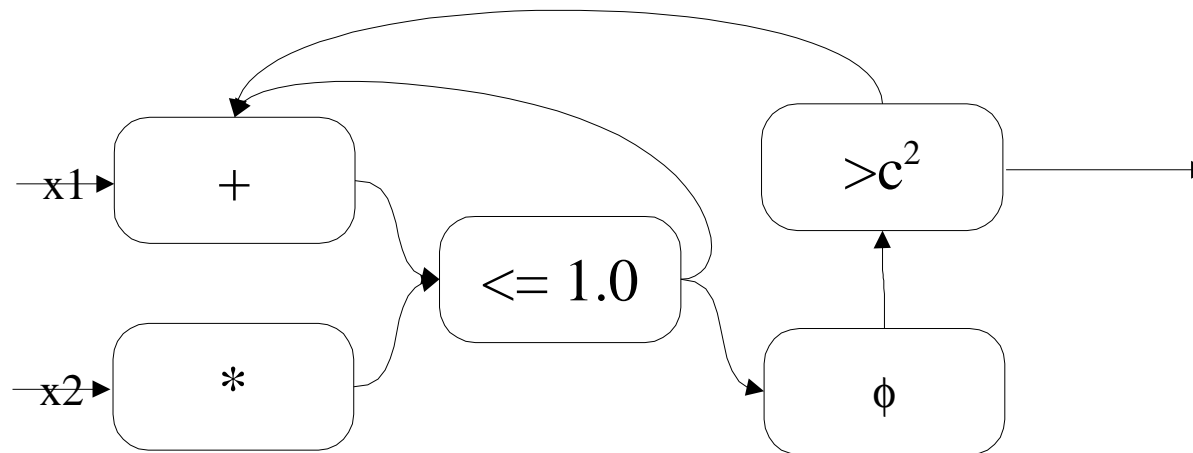
Fault-Based Adequacy Criteria

- Evaluation

- fits well into **conventional software production**: simply controls "quality engineers"
- yahoo! no formal spec necessary !
- good means to **locate errors**
- **looks automatic**
- mutant generation via transformations
- **Problem:** Many equivalences - human costs
- **Problem** for error-seeding:
it is difficult to find a good error-model for software
- **Problem:** large computation resources required
- ⇒ Up to now, works only for **hardware-testing** (Stuck-At-Analysis)

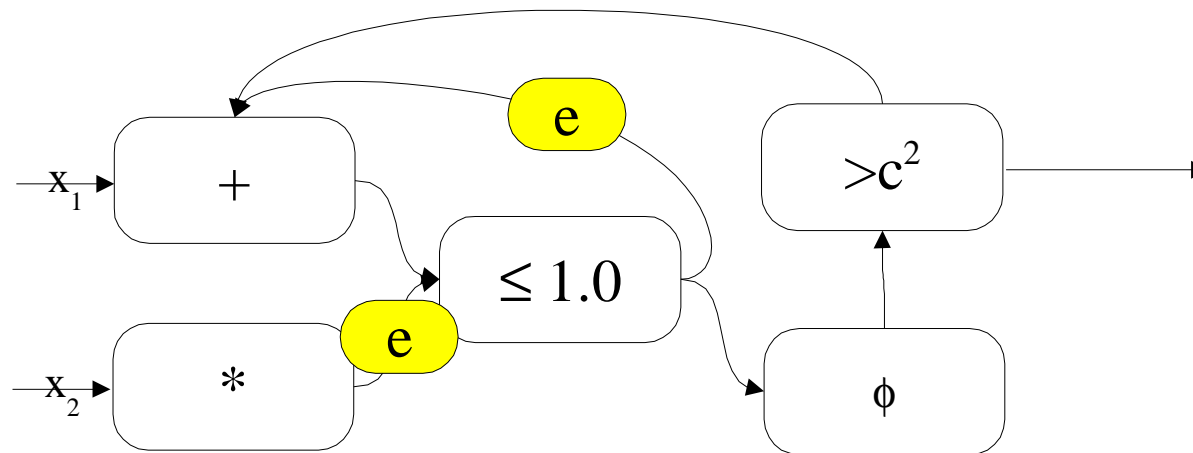
Fault-Based Adequacy Criteria

- Perturbation Tests



Fault-Based Adequacy Criteria

- **perturbations** (*error-functions*)



- for algorithms with fixed number of variables with continuous input domain

Fault-Based Adequacy Criteria

- Adequacy criteria:
 - Adequacy in detecting perturbations
 - Adequacy of detecting predicate perturbation
- Evaluation
 - good, well-understood semantic basis
 - under certain conditions guarantees for error-detection
 - very limited application area

Error-Based Adequacy

- Specification-Based Input Space Partitioning

"basic idea is to **partition** the input/output behaviour space into subdomains ..."

- Gaudel, Dick/Faivre, Santen, Stepney ...

as we will see next week ...

- Program-Based Input Space Partitioning

"input test-cases along **paths in control flow** graphs using symbolic execution"

- Boundary Analysis
- Functional Analysis

Conclusion

- wide spectrum of validation methods
 - testing can be formal, too
 - basics of testing are simple,
but difficult to scale up
 - Evaluation:
 - conventional (semantic free) tests rather dubious
(with the exception of data-flow based testing)
 - specification and program should both be used
 - trend to systematic tests and formal methods
- ⇒ more in the next lecture . . .

Conclusion II

Specification Based Testing

- test-case-generation **is** deduction
- **many problems due to size**
- for good tests:
 - a lot of different techniques need to be combined,
 - a lot of criteria need to be combined
- challenges: size, sequencing,
non-determinism,
time, embedded systems



Validation of Software-Systems (2)

Burkhart Wolff

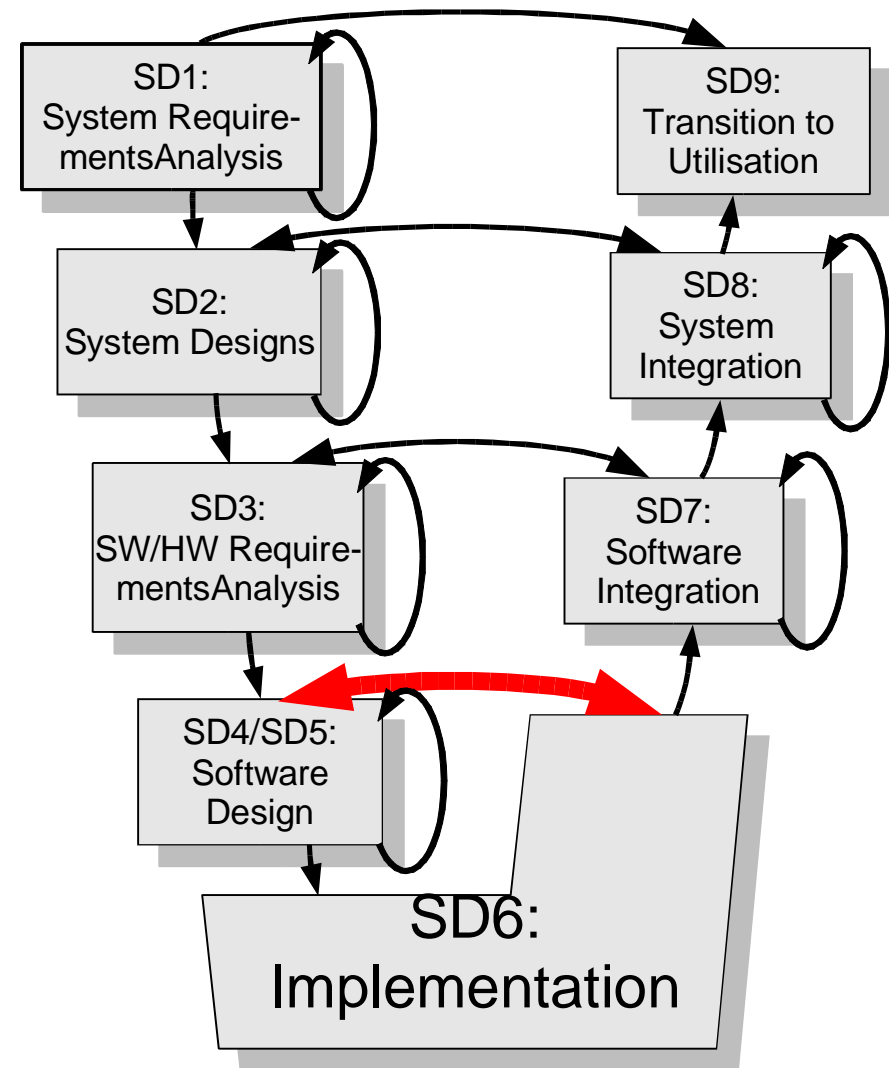
AG Softwaretechnik

Overview

- Specification based Unit Tests:

Where are we in the Development Process?

HERE !!!



Overview

- DNF-Method
- Handling Quantifiers in the "Algebraic Method"
- Test-Sequence Generation (Dick/Faivre)
- Abstraction Techniques
- Conclusion

Unit Test: An Introductory Example.

- The Problem:

A program reads 3 integers. Their values represent the length of the 3 edges of a triangle. The program decides if the triangle is

- equilateral ("gleichseitig")
- isosceles ("gleichschenkelig")
- scalene ("ungleichseitig")

Unit Test: Introductory Example

1. Do you have a test case that represents a *valid* scalene triangle? (Note that test cases such as 1,2,3 and 2,5,10 do not warrant a “yes” answer, because there does not exist a triangle having such sides.)
2. Do you have a test case that represents a valid equilateral triangle?
3. Do you have a test case that represents a valid isosceles triangle? (A test case specifying 2,2,4 would not be counted.)
4. Do you have at least three test cases that represent valid isosceles triangles such that you have tried all three permutations of two equal sides (e.g., 3,3,4; 3,4,3; and 4,3,3)?
5. Do you have a test case in which one side has a zero value?
6. Do you have a test case in which one side has a negative value?
7. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is equal to the third? (That is, if the program said that 1,2,3 represents a scalene triangle, it would contain a bug.)
8. Do you have at least three test cases in category 7 such that you have tried all three permutations where the length of one side is equal to the sum of the lengths of the other two sides (e.g., 1,2,3; 1,3,2; and 3,1,2)?
9. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is less than the third (e.g., 1,2,4 or 12,15,30)?
10. Do you have at least three test cases in category 9 such that you have tried all three permutations (e.g., 1,2,4; 1,4,2; and 4,1,2)?
11. Do you have a test case in which all sides are 0 (i.e., 0,0,0)?
12. Do you have at least one test case specifying noninteger values?
13. Do you have at least one test case specifying the wrong number of values (e.g., two, rather than three, integers)?
14. For each test case, did you specify the expected output from the program in addition to the input values?

Unit Test

- Hmm, could this task be accomplished

systematically ?

- Even better, could this task be accomplished

automatically ?

Approximate the Infinite by the Finite

- What does this mean *Conceptually*?
 - **Uniformity Hypothesis:**
System behaves "uniformly" in some (finite) set of (infinite) input sets (the *partitioning* of input). Then we can test on representatives of input sets.
 - **Regularity Hypothesis:**
Assumes that all input t with a certain complexity $|t|$ less than a bound n are sufficient to establish the correct behaviour of a system.

In event-models, the regularity hypothesis can also be interpreted as follows: After some finite set of finite input-sequences, the system reaches a tested state. The system is assumed to behave like a finite automaton.

Approximate the Infinite by the Finite

- What does this mean *deductionally/computationally*?

- Uniformity Hypothesis:

$$\frac{\bigcup_{i:1..n} S(i) = A \wedge \forall i:1..n \bullet \exists x:S(i) \bullet P(x)}{\forall t:A \bullet P(t)}$$

- Regularity Hypothesis:

$$\frac{\forall t \bullet |t| < n \Rightarrow P(t)}{\forall t \bullet P(t)}$$

Note: this is quite induction-like. . .

Approximate the Infinite by the Finite

- The Uniformity Hypothesis gives rise to the following

IDEA:

Compute the Partitions via **D**isjunctive-**N**ormal-**F**orms!

$$\forall x_1, \dots, x_n \bullet D_1(x_1, \dots, x_n) \vee \dots \vee D_m(x_1, \dots, x_n)$$

and choose for each D_i arguments such that the disjoint becomes valid !

DNF-Method

- Example:

triangle : $\mathbb{P}(\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z})$

$\forall x, y, z : \mathbb{Z} \bullet \text{triangle}(x, y, z) = x > 0 \wedge y > 0 \wedge z > 0 \wedge$
 $x + y > z \wedge y + z > x \wedge x + z > y$

res ::= equilateral | isosceles | scalene | error

program : $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \Downarrow$ res

$\forall x, y, z : \mathbb{Z} \bullet \text{triangle}(x, y, z) \wedge$
 $\text{program}(x, y, z) = \text{if } x=y \text{ then if } y=z \text{ then equilateral}$
 $\qquad \qquad \qquad \text{else isosceles}$
 $\qquad \qquad \qquad \text{else if } y=z \text{ then isosceles}$
 $\qquad \qquad \qquad \text{else if } x=z \text{ then isosceles}$
 $\qquad \qquad \qquad \text{else scalene } \vee$
 $\neg \text{triangle}(x, y, z) \wedge \text{program}(x, y, z) = \text{error}$

DNF-Method

- We have the following rules:
 - specification message:
 $(a = (\text{if } Q \text{ then } x \text{ else } y)) = ((Q \Rightarrow a = x) \wedge (\neg Q \Rightarrow a = y))$
 - logical stuff:
 $(a \Rightarrow b) = (\neg a \vee b), \neg\neg a = a, \dots$
 - distributivity (the core of DNF computation):
 $(P \wedge (Q \vee R)) = (P \wedge Q \vee P \wedge R), \quad ((P \vee Q) \wedge R) = (P \wedge R \vee Q \wedge R)$
 - data type information:
 $\text{equilateral} \neq \text{isosceles}, \text{equilateral} \neq \text{scalene}, \text{equilateral} \neq \text{error}, \dots$
 - conjunction-congruence rule:

$$\frac{P = P' \quad \begin{array}{c} [P'] \\ Q = Q' \end{array}}{(P \wedge Q) = (P' \wedge Q')}$$

DNF-Method

- We apply these rules in 4 term-rewriting processes, each computing a normal form
- **Result** (in Isabelle/HOL(98)):
 - $x = y \wedge y = z \wedge \text{triangle}(z, z, z) \wedge \text{program}(z, z, z) = \text{equilateral} \vee$
 $x = y \wedge y \neq z \wedge \text{triangle}(y, y, z) \wedge \text{program}(y, y, z) = \text{isosceles} \vee$
 $y = z \wedge x \neq z \wedge \text{triangle}(x, z, z) \wedge \text{program}(x, z, z) = \text{isosceles} \vee$
 $x = z \wedge y \neq z \wedge \text{triangle}(z, y, z) \wedge \text{program}(z, y, z) = \text{isosceles} \vee$
 $x \neq y \wedge x \neq z \wedge y \neq z \wedge \text{triangle}(x, y, z) \wedge \text{program}(x, y, z) = \text{scalene} \vee$
 $\neg \text{triangle}(x, y, z) \wedge \text{program}(x, y, z) = \text{error}$
 - Further unfolding of the definition of triangle and DNF-computation yields 6 more cases (for error).

DNF-Method

- For each of the clauses in the DNF we generate a solution and get the test case:

x	y	z	program
2	2	2	equilateral
1	1	2	isoscele
2	1	1	isoscele
1	2	1	isoscele
4	5	3	scalene

Normal Behaviour

x	y	z	program
-2	2	2	error
1	-2	2	error
2	1	-1	error
1	2	4	error
4	1	2	error
1	4	2	error

Exceptional Behaviour

DNF-Method

- Evaluation:
 - Test-Class-Generation automatic (in principle)
for specs with one outermost universal quantification
 - Allows "massage" of specifications
 - Requires theorem proving . . .
 - Produces quite "minimal" Test-Set
 - But: significant blow-ups may occur . . .
- ⇒ Test-Generation theoretically shallow,
but technologically hard problem (BDD's, . . .)

Handling Quantifiers by the "Algebraic Method"

- The DNF-approach requires a special form.
 - What happens with arbitrary specification formulas?
 - What happens with Quantifiers?
 - How to approximate an **infinite** model by a **finite** one?

Handling Quantifiers by the "Algebraic Method"

- What to do with universal quantifiers?

- find finite bound and enumerate conjunctions
- apply uniformity hypothesis
- apply regularity hypothesis

$$\begin{aligned}(\forall x \bullet x = \{a,b,c\} \wedge P(x)) \\ = P(a) \wedge P(b) \wedge P(c)\end{aligned}$$

- What to do with existential quantifiers?

- one point rule
- find finite bound and enumerate disjunctions
- other techniques

$$(\exists x \bullet x = t \wedge P(x)) = P(t)$$

$$\begin{aligned}(\exists x \bullet x = \{a,b,c\} \wedge P(x)) \\ = P(a) \vee P(b) \vee P(c)\end{aligned}$$

(Mona, Constr. Sat., Resolution . . .)

Handling Quantifiers by the "Algebraic Method"

- Example: Specification of *insert*

$$\text{perm} : \text{seq } \mathbb{Z} \leftrightarrow \text{seq } \mathbb{Z}$$

$$\text{asc} : \mathbb{P}(\text{seq } \mathbb{Z})$$

$$\forall x, y: \text{seq } \mathbb{Z} \bullet \text{perm}(x, y) = \exists f : 1..\#x \rightarrow 1..\#y \bullet$$

$$\forall i: 1..\#x \bullet x(i) = y(f(i))$$

$$\forall x: \text{seq } \mathbb{Z} \bullet \text{asc}(x) = \forall i: 1..\#x-1 \bullet x(i) \leq x(i+1)$$

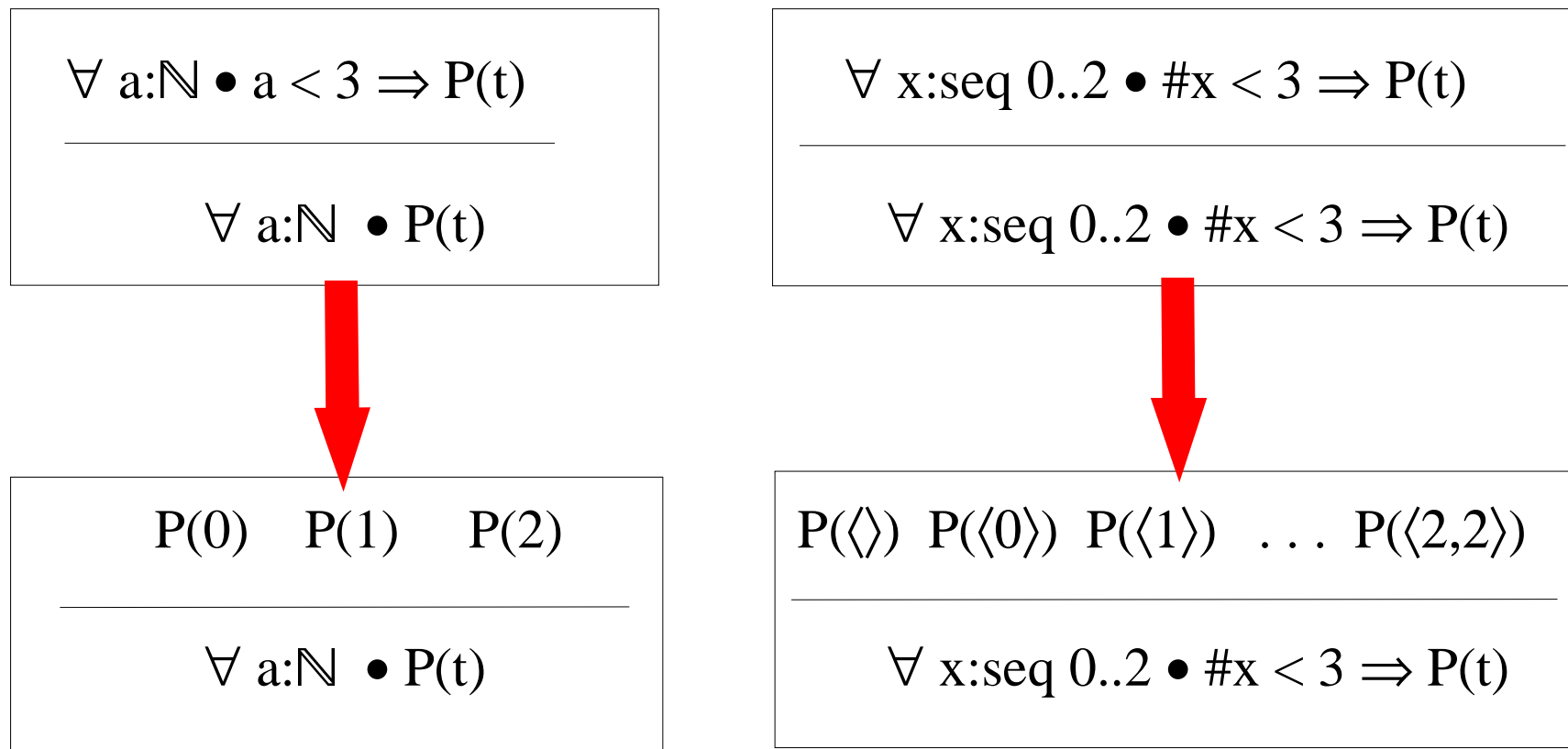
$$\text{insert}: \mathbb{Z} \times \text{seq } \mathbb{Z} \rightarrow \text{seq } \mathbb{Z}$$

$$\forall a: \mathbb{Z}, x: \text{seq } \mathbb{Z} \bullet \text{asc } x \Rightarrow \text{let } y = \text{insert}(a, x)$$

$$\bullet \text{asc}(y) \wedge \text{perm}(\langle a \rangle^\frown x, y)$$

Handling Quantifiers by the "Algebraic Method"

- Idea: We apply Regularity Hypothesis:



$H(0, \langle 2, 2 \rangle)$	$H(1, \langle 2, 2 \rangle)$	$H(2, \langle 2, 2 \rangle)$
$H(0, \langle 2, 1 \rangle)$	$H(1, \langle 2, 1 \rangle)$	$H(2, \langle 2, 1 \rangle)$
$H(0, \langle 2, 0 \rangle)$	$H(1, \langle 2, 0 \rangle)$	$H(2, \langle 2, 0 \rangle)$
$H(0, \langle 1, 2 \rangle)$	$H(1, \langle 1, 2 \rangle)$	$H(2, \langle 1, 2 \rangle)$
$H(0, \langle 1, 1 \rangle)$	$H(1, \langle 1, 1 \rangle)$	$H(2, \langle 1, 1 \rangle)$
$H(0, \langle 1, 0 \rangle)$	$H(1, \langle 1, 0 \rangle)$	$H(2, \langle 1, 0 \rangle)$
$H(0, \langle 1, 2 \rangle)$	$H(1, \langle 1, 2 \rangle)$	$H(2, \langle 1, 2 \rangle)$
$H(0, \langle 1, 1 \rangle)$	$H(1, \langle 1, 1 \rangle)$	$H(2, \langle 1, 1 \rangle)$
$H(0, \langle 2 \rangle) \ H(0, \langle 1, 0 \rangle)$	$H(1, \langle 2 \rangle) \ H(1, \langle 1, 0 \rangle)$	$H(2, \langle 2 \rangle) \ H(2, \langle 1, 0 \rangle)$
$H(0, \langle 1 \rangle) \ H(0, \langle 0, 2 \rangle)$	$H(1, \langle 1 \rangle) \ H(1, \langle 0, 2 \rangle)$	$H(2, \langle 1 \rangle) \ H(2, \langle 0, 2 \rangle)$
$H(0, \langle 0 \rangle) \ H(0, \langle 0, 1 \rangle)$	$H(1, \langle 0 \rangle) \ H(1, \langle 0, 1 \rangle)$	$H(2, \langle 0 \rangle) \ H(2, \langle 0, 1 \rangle)$
$H(0, \langle \rangle) \ H(0, \langle 0, 0 \rangle)$	$H(1, \langle \rangle) \ H(1, \langle 0, 0 \rangle)$	$H(2, \langle \rangle) \ H(2, \langle 0, 0 \rangle)$

$$\forall x: \text{seq } \mathbb{Z} \bullet H(0, x)$$

$$\forall x: \text{seq } \mathbb{Z} \bullet H(1, x)$$

$$\forall x: \text{seq } \mathbb{Z} \bullet H(2, x)$$

$$\forall a: \mathbb{Z}, x: \text{seq } \mathbb{Z} \bullet H(a, x)$$

$$H(a, x) = \text{asc } x \Rightarrow \text{asc}(\text{insert}(a, x)) \wedge \text{perm}(\langle a \rangle^\frown x, \text{insert}(a, x))$$

$$\text{Software Engineering } H'(a, x) = \text{asc}(\text{insert}(a, x)) \wedge \text{perm}(\langle a \rangle^\frown x, \text{insert}(a, x))$$

H'(0,⟨2,2⟩)		H'(1,⟨2,2⟩)		H'(2,⟨2,2⟩)	
H'(0,⟨1,2⟩)		H'(1,⟨1,2⟩)		H'(2,⟨1,2⟩)	
H'(0,⟨1,1⟩)		H'(1,⟨1,1⟩)		H'(2,⟨1,1⟩)	
H'(0,⟨1,2⟩)		H'(1,⟨1,2⟩)		H'(2,⟨1,2⟩)	
H'(0,⟨1,1⟩)		H'(1,⟨1,1⟩)		H'(2,⟨1,1⟩)	
H'(0,⟨2⟩)	H'(0,⟨1,0⟩)	H'(1,⟨2⟩)	H'(1,⟨1,0⟩)	H'(2,⟨2⟩)	H'(2,⟨1,0⟩)
H'(0,⟨1⟩)	H'(0,⟨0,2⟩)	H'(1,⟨1⟩)	H'(1,⟨0,2⟩)	H'(2,⟨1⟩)	H'(2,⟨0,2⟩)
H'(0,⟨0⟩)	H'(0,⟨0,1⟩)	H'(1,⟨0⟩)	H'(1,⟨0,1⟩)	H'(2,⟨0⟩)	H'(2,⟨0,1⟩)
H'(0,⟨⟩)	H'(0,⟨0,0⟩)	H'(1,⟨⟩)	H'(1,⟨0,0⟩)	H'(2,⟨⟩)	H'(2,⟨0,0⟩)
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮		⋮		⋮	
⋮					

$$H(a, x) = \text{asc } x \Rightarrow \text{asc}(\text{insert}(a, x)) \wedge \text{perm}(\langle a \rangle^{\frown} x, \text{insert}(a, x))$$

Algebraic Method

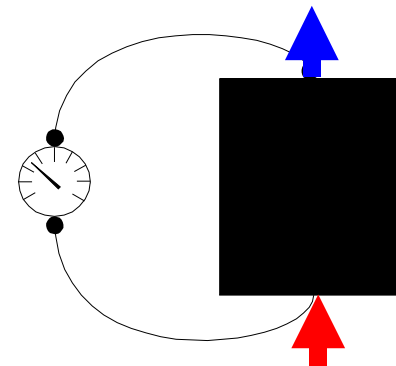
- Observation: The $H'(\dots)$ represent Test-Oracles!
 - evaluate each $insort(x,y)$ with ground x and y by the program under test and replace its value z
 - interpret the

$$H'(x,y,z) = \text{asc}(\text{insort}(x, y)) \wedge \text{perm}(\langle x \rangle^y, z)$$

by symbolic evaluation (such as ZETA)

NOTE: Quantifiers now finite !

- this process represents an oracle:
 - true - succesful
 - false - counterexample found !



Algebraic Method

- Evaluation:
 - The form of the regularity hypothesis shows that *testing* approximates *induction*
 - Heuristic:
 - choice of n
 - choice of the *measure* $|t|$
 - Technique quite powerful in connection with animation and evaluation techniques of (finite) Z specifications . . .

Test-Sequence Generation (Dick/Faivre)

- See Brucker Slides . . .

Conclusion

- (White-Box) Testing can be *very* formal
- Challenging subject for deduction and partial evaluation
- Testing can be an *approximation to verification* in the sense:
 Testing = Verification under
 Testing Hypothesis
- The adequate choice of test-hypothesis remains speculative . . .

Conclusion

- Systematic Test-Case Generation possible:
 - requires specification
 - makes testing hypotheses explicit
 - converges against verification
- Test-Case Generation often computationally hard
- A lot of research needs to be done . . .

Publications

- [Myers 97] The Art of Software Testing, Wiley Interscience, Wiley & Sons, 1997, ISBN 0-471-04328-1
- [Beiz 90] Boris Beizer: Software Testing Techniques Thomson Computer Press, 1990, ISBN 1-850-32880-3
- [BGJ 98] F. Belli, M. Grochtmann, O. Jack: Erprobte Modelle zur Quantifizierung der Software-Zuverlässigkeit. Informatik Spektrum 21:131-140(1998).
- [DB 98] J. Derrick and E. Boiten: Testing Refinements by Refining Tests. In: J.P.Bowen, A.Fett, M.G.Hinchey (Eds): ZUM98: The Z Formal Specification Notation. LNCS 1493, 1998.
- [DF 93] J. Dick, A. Faivre: Automating the Generation and Sequencing from Model-Based Specifications. FME93, pp 268-284, LNCS670.
- [DIN 9001] DIN EN ISO 9001: Normen zum Qualitätsmanagement und zur Qualitätssicherung. Berlin,: Deutsches Institut für Normung e.v. 1994.
- [DLS 78] R.A. DeMillo, W.M. McCracken, R.J. Matin, J.F. Passafiume: Test-case generation from Prolog-based specifications. IEEE Soft. (March), 49-57, 1978.
- [Gau 95] M.-C Gaudel: Testing can be formal, too. TAPSOFT 95, LNCS 915, pp.82 – 96.

Publications

- [Ham 77] R.G.Hamlet: Testing programs with the aid of a compiler. IEEE Trans. Softw. Eng. 3, 4 (July), 279-290,1977.
- [Heiz 88] Bill Heizel: The Complete Guide to Software Testing Wiley QED, Wiley & Sons, 1988, ISBN 0-471-56567-9
- [Hen 88] M. C. Henessee: Algebraic Theory of processes. MIT Press, 1988.
- [HNS 97] S. Helke, T. Neustruppy, T. Santen: Automating Test Case Generation from Z Specifications with Isabelle. Proceedings of ZUM'97. LNCS 1212. 1997.
- [HP 94] H.-M. Hoercher and J. Peleska: The Role of Formal Specifications in Software Test. Tutorial, held at the FME '94.
- [HP 95] H.-M. Hoercher and J. Peleska: Using formal specifications to support software testing. Software Quality Journal 4, 309-327 (1995).
- [Ried 97] E.H. Riedemann: Testmethoden für sequentielle und nebenläufige Software-Systeme.
- [Pel 96] J. Peleska: Test Automation for Safety-Critical Systems: Industrial Applications and Future Developments. In: FME 96, LNCS 1051.

Publications

- [PS 96] J. Peleska and M. Siegel: From Testing Theory to Test Driver Implementation. In M.-C. Gaudel and J. Woodcock (Eds.): FME '96: Industrial Benefit and Advances in Formal Methods. LNCS 1051, Springer-Verlag, Berlin Heidelberg New York (1996) 538-556.
- [Ried 97] E. H. Riedemann: Testmethoden für sequentielle und nebenläufige Software-Systeme. Teubner Stuttgart. 1997.
- [Rosc 98] A.W. Roscoe: The Theory and Practice of Concurrency. Prentice Hall, 1998.
- [Step 95] S. Stepney: Testing as Abstraction. In: J.P. Bowen, M.G. Hinchey(Eds): The Z Formal Specification Notation. LNCS 967, 1995.
- [Wojt 88] H. Wojtkowiak: Test und Testbarkeit digitaler Schaltungen. B. G. Teubner, 1988.
- [Zeil 83] S.J. Zeil: Testing for perturbations of program statements. IEEE Trans. Softw. Eng. SE-9,3,(May), 335-346. 1983.
- [ZHM 97] H. Zhu, P. Hall, J. May: Software Unit Test Coverage and Adequacy, ACM Computing Surveys, Vol. 29, No. 4, Dec. 1997