Software Engineering
Introduction

Prof. Dr. Peter Müller

Chair of Programming Methodology

Spring Semester 2010

1. Introduction

1.1 Course Outline

1.2 Motivation

1.3 Software Qualities

1.4 Software Engineering Principles

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Approach

- **This course will *entirely* focus on *object-oriented* software engineering**

- **You will have to carry out a *project* from the problem statement to deployment**

- **Exercise sessions will be used for**
  - Student presentations
  - Discussions
  - Introductions to software engineering tools

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

After this Course, you should

- Be able to produce **high-quality software**
- Be able to deal with **complexity** and **change**

- Have the **technical knowledge** (main emphasis)
- Have an overview of the **managerial knowledge**
- Have an overview of relevant **tools**

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Course Outline (tentative)

1. Introduction
2. Requirements Elicitation
3. Analysis
4. System Design
5. Detailed Design
6. Implementation
7. Testing and Quality Assurance
8. Process Models
9. Project Management

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Guest Lecture

- # Harald Gall, Universität Zürich:
  Software Evolution: Analysis and Visualization
  - April 26, 2010

Grading

- **Projects**
  - Ungraded, but **must be completed successfully**
  - To pass, you must submit **all four deliverables in time and in sufficient quality**

- **Exam**
  - Written exam in the **exam session**
  - Knowledge from project will be **essential**!

- **Grade is determined entirely by the exam**

Course Infrastructure

- Web page:
  www.pm.inf.ethz.ch/education/courses/kse

- Slides will be available on the web page two days before the lecture (Thursday)

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Literature

- No single book covers course content

- Good general books on Software Engineering
  - Bernd Bruegge, Allen H. Dutoit: Object-Oriented Software Engineering. Prentice Hall, 2004.
  - Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli: Fundamentals of Software Engineering. Prentice Hall, 2002.

- **See course web page for a comprehensive list**

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

The Projects

- **Software Engineering** projects
  - Not just programming projects

- Topic: An in-flight entertainment system
  - Focus on **development process rather than result**

- Projects can be done in **teams of three or four**

- Details will be explained in first exercise session

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Background Knowledge

- **The lecture focuses on concepts**

- **For the projects, you will also need knowledge about the technology**
  - UML
  - Java
  - Various tools

- **We expect you to acquire this knowledge!**

Exercise Sessions

- ## Thursday, 8:00-10:00
  - Valentin Wüstholz (IFW B42)
- ## Friday, 13:00-15:00
  - Hermann Lehner (IFW C42)
- ## Monday, 15:00-17:00
  - Joseph Ruskiewicz (IFW A34)

- ## **Exercises start on Thursday this week**

ETH
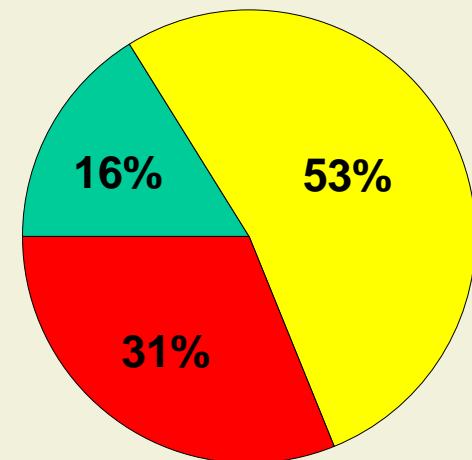Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

1. Introduction

1.1 Course Outline
1.2 Motivation
1.3 Software Qualities
1.4 Software Engineering Principles

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Software – a Poor Track Record

- Software bugs cost the U.S. economy an estimated $59.5 billion annually, or about 0.6 percent of the gross domestic product

- 84% of all software projects are unsuccessful
  - Late, over budget, less features than specified, cancelled

- The average unsuccessful project
  - 222% longer than planned
  - 189% over budget
  - 61% of originally specified features

16%   53%

31%

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Quality of Today's Software …

… Has Major Impact on Users

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
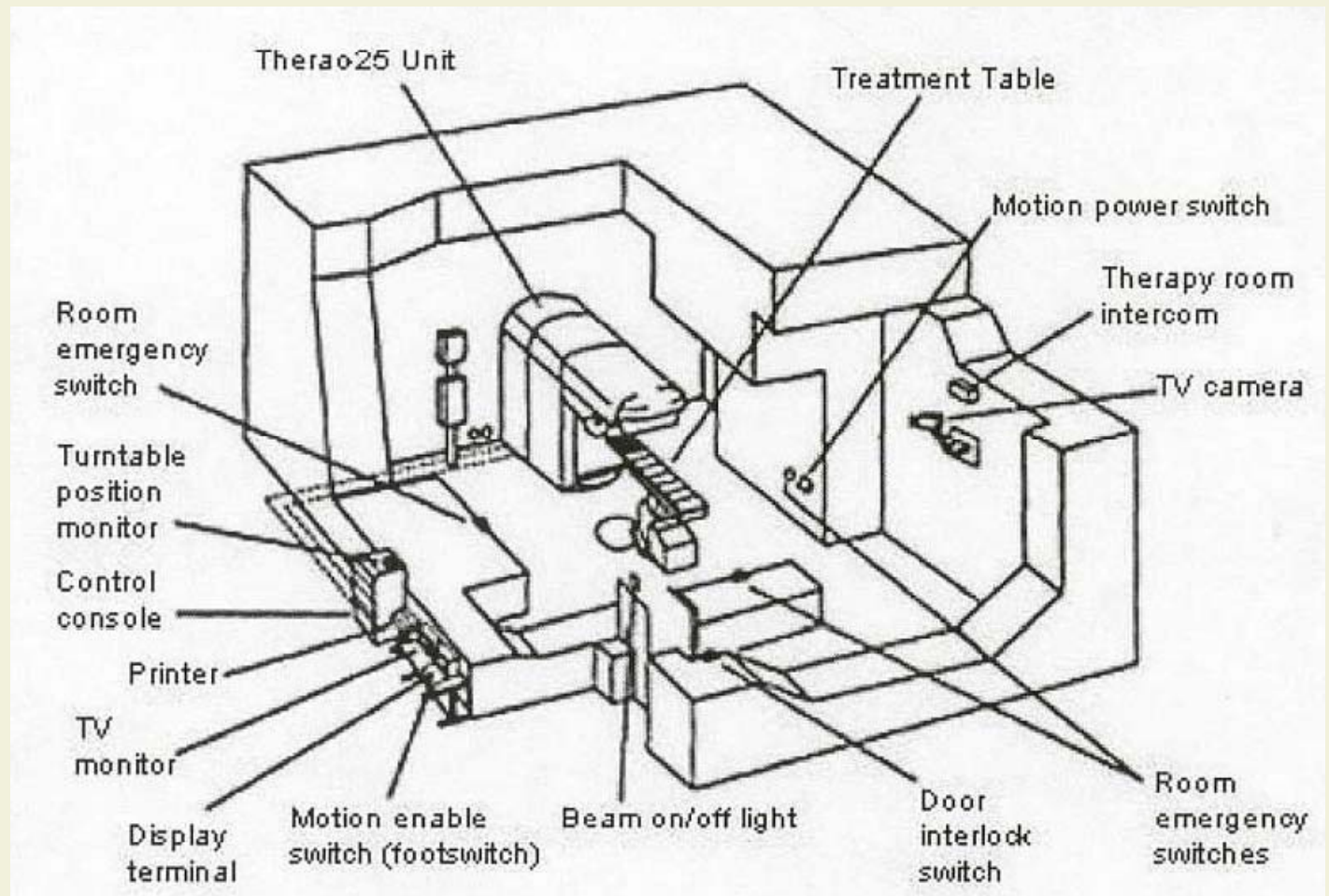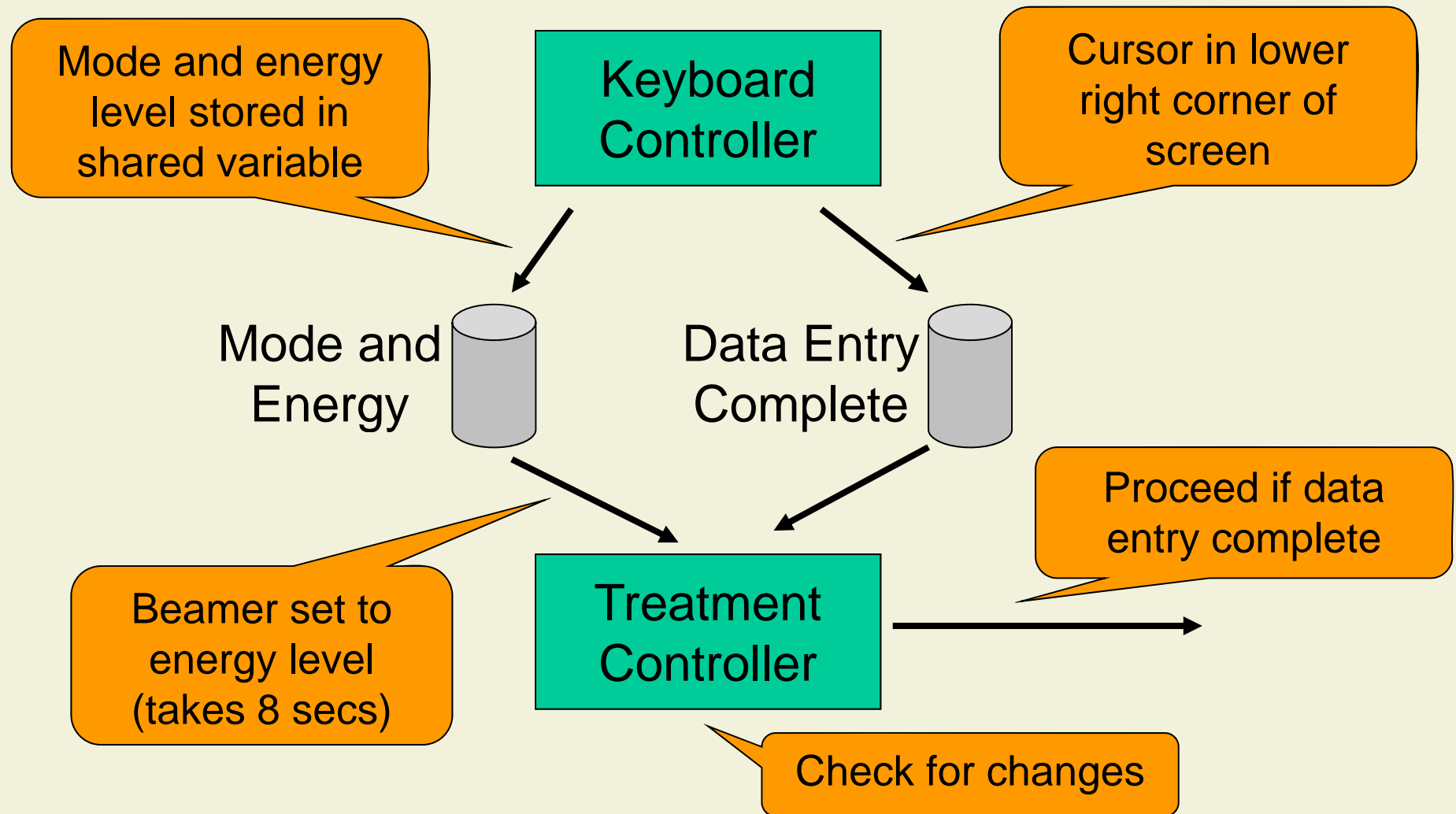
The Therac-25 Accident

- Therac-25 is a medical linear accelerator

- High-
  energy
  X-ray
  and
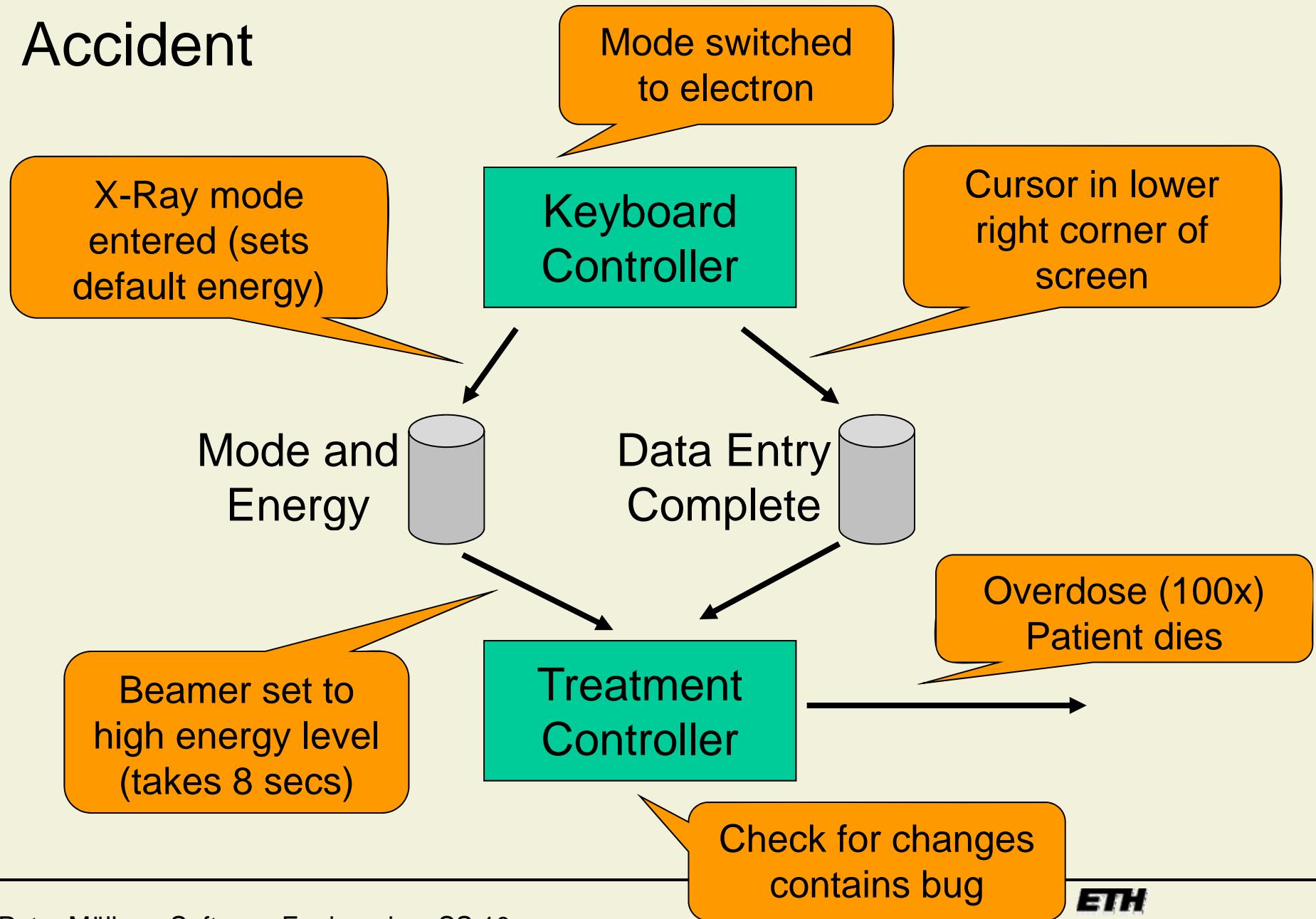 electron
  beams
  destroy
  tumors

Therac-25 System Design

- **Therac-25 is completely computer-controlled**
  - Software written in assembler code
  - Therac-25 has its own real-time operating system
- **Software partly taken from ancestor machines**
  - Software functionality limited
  - Hardware safety features and interlocks
- Hazard analysis
  - Extensive testing on hardware simulator
  - Program software does not degrade due to wear, fatigue, or reproduction process
  - Computer errors are caused by hardware or by alpha particles

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Therac-25 Software Design

Mode and energy level stored in shared variable

Keyboard Controller

Cursor in lower right corner of screen

Mode and Energy

Data Entry Complete

Proceed if data entry complete

Beamer set to energy level (takes 8 secs)

Treatment Controller

Check for changes

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Accident

Mode switched
to electron

X-Ray mode
entered (sets
default energy)

Keyboard
Controller

Cursor in lower
right corner of
screen

Mode and
Energy

Data Entry
Complete

Overdose (100x)
Patient dies

Beamer set to
high energy level
(takes 8 secs)

Treatment
Controller

Check for changes
contains bug

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Analysis of the Therac-25 Accident

- **Changed requirements** were not considered
  - In Therac-25 software is safety-critical

- **Design** is too **complex**

  - Concurrent system, shared variables (race conditions)

- **Code** is **buggy**

  - Check for changes done at wrong place

- **Testing** was **insufficient**

  - System test only, almost no separate software test

- **Maintenance** was **poor**

  - Correction of bug instead of re-design (root cause)

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Challenge: Complexity

- **Complexity is caused by**
  - Complexity of the problem domain
  - Complexity of the development process
  - Flexibility of software

- **Aspects of complexity**
  - Multi-person construction (team-effort)
  - Multi-version software
  - Often conflicting objectives
  - Development and operation lasts many years

Challenge: Change

- **Change is caused by**
  - Bug fixes
  - Changing requirements
    (adding, enhancing, removing features)
  - Changing environment
  - Changing development team

- **Each implemented change erodes the structure of the system, which makes the next change even more expensive**

Software Engineering: Definition 1

- *A collection of techniques, methodologies, and tools that help with the production of*
  - *a high quality software system*
  - *with a given budget*
  - *before a given deadline*
  - *while change occurs*

  [Brügge]

- Constraints are important

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Software Engineering: Definition 2

- *The application of a systematic, disciplined, and quantifiable approach to the development, operation, and maintenance of Software; that is, the application of engineering to software*

[IEEE, ANSI]

- Software engineering spans whole product lifecycle

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Science vs. Engineering

Engineering

- The **application of science** to the **needs of humanity**

- **Application of knowledge, mathematics, and practical experience** to the design of useful objects or processes

Science

- Knowledge covering **general truths** or the operation of **general laws**

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computer Science vs. Software Engineering

Software Engineering

- The **application of computer science,** mathematics, project management **to build** high quality **software**

Computer Science

- Computability, algorithms and complexity, programming languages, data structures, databases, artificial intelligence, etc.

Related Areas

> Is this requirement addressed on hardware or software level (or both)?

> Will the project be completed in time and budget?

- **Systems Engineering**
  - Complex systems with software and hardware
  - Interdisciplinary
  - Example: Therac-25

- **Project Management**
  - Organizes and leads the project work to meet project requirements
  - Concerned with time, budget, procurement, communication, etc.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

1. Introduction

1.1 Course Outline

1.2 Motivation

1.3 Software Qualities

1.4 Software Engineering Principles

Representative Software Qualities

Correctness

Maintainability

Performance

Verifiability

Robustness

Understandability

Scalability

Reusability

Reliability

Evolvability

Usability

Portability

Security

Repairability

Interoperability

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Correctness

- Correct software meets its **functional requirements** specification

- Correctness is a **mathematical property**

- Can be enhanced by

  - Appropriate tools (e.g., high-level languages)
  - Standard algorithms and libraries
  - An established development process

- Example: security control system of the "Meteor" line of the Paris metro is proven to be correct

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Robustness

- Robust software **behaves "reasonably"**, even in circumstances **not covered by the specification**

- Can be enhanced by
  - Assertions (Design by Contract)
  - Software monitoring
  - Defensive programming

- Example: database system performs a controlled shutdown when hardware error occurs
  - No data is corrupted
  - Behavior is logged for later analysis or retry

Security

- Secure software is **protected against unauthorized access** to or **modification** of information

  - Confidentiality, integrity, availability

- Can be enhanced by

  - Cryptography

  - Proven protocols

- Example: internet banking uses cryptography to protect transmitted data from leaking and manipulation

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Reliability

- **Reliable software has a high probability to operate as expected over a specified interval**

- **Reliability is a statistical property**

- **Can be enhanced by**
  - Fault avoidance (e.g., careful design)
  - Fault tolerance (e.g., redundancy)
  - Fault detection (e.g., testing)

- **Example: telephone system establishes a connection > 99.9% of the time**

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Performance

- **High-performance software is fast and consumes a small amount of memory**
  - Response time
  - Throughput
  - Memory usage

- Can be enhanced by
  - Considering performance when designing the software architecture
  - Code optimization (performance tuning)

- Example: a stock trading system handles up to 100'000 orders per hour

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Scalability

- Scalable software shows **increased performance** under an increased load **when resources** (typically hardware) **are added**

- Can be enhanced by
  - De-centralized architectures
  - Low complexity of algorithms

- Examples
  - Peer-to-peer file exchange systems scale easily to millions of users
  - A routing protocol is scalable if the size of the routing table grows as O(log N), where N is the number of nodes

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

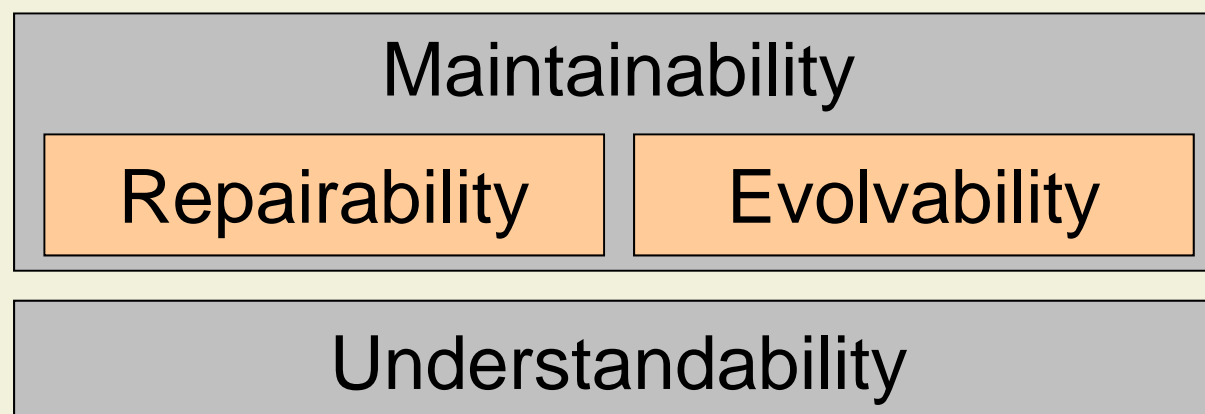# Usability (User Friendliness)

- **Usable software is found easy to use by humans**
  - Subjective (e.g., experts and novices have difference requirements)

- **Can be enhanced by**
  - Offering different user interfaces
  - Adaptable user interfaces (maybe even automatically)
  - New forms of human computer interaction (e.g., speech)

- **Example: order system offers a GUI for occasional users and a command line interface for experts**

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Interoperability

- **Interoperable software can coexist and cooperate with other systems**

- Can be enhanced by
  - Well-documented interfaces (e.g., file formats, protocols)
  - Standard interface formats (e.g., XML)

- Examples
  - A word processor can incorporate a spreadsheet table or graph
  - By using a web service, any application can query Google

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Maintainability

- **Maintainable software enables or simplifies modification after initial development**
  - **Corrective** maintenance (bug fixing)
  - **Adaptive** maintenance (adaptation to changed environment, e.g., new version of operating system)
  - **Perfective** maintenance (improvement, e.g., new functions)

Maintainability	
Repairability	Evolvability

Understandability

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zürich

Maintainability (cont'd)

- **Can be enhanced by**
  - Modular design, narrow interfaces
  - Good documentation
  - Extensive test suite (preferably automated)

- **Example: Deutsche Bank order system was developed in 1960s and has been maintained since then**
  - New hardware, operating system, database system
  - New functionality (internet banking, intraday trading)

Verifiability

- **Properties** of verifiable software **can be verified** easily
    - Testing
    - Formal verification

- Can be enhanced by
    - Software monitors (e.g., to measure performance)
    - Modular design

- Example: Assertions (contracts) enable runtime assertion checking to find bugs

Reusability

- **Reusable software can be reused, adapted, and composed to develop new products**
  - Different levels of granularity from methods to applications

- **Can be enhanced by**
  - Modular design, narrow interfaces, parameterization
  - Good documentation
  - Object technology (inheritance, overriding)

- **Example: class libraries of OO-languages such as C#, Eiffel, Java, etc.**

Portability

- **Portable software can run in different environments** (e.g., hardware, operating system)

- Can be enhanced by
  - Isolation of dependencies on environment
  - Layered architectures
  - Virtual machines

- Example: Java applications can run in any environment that provides a virtual machine ("write once, run anywhere")

ETH
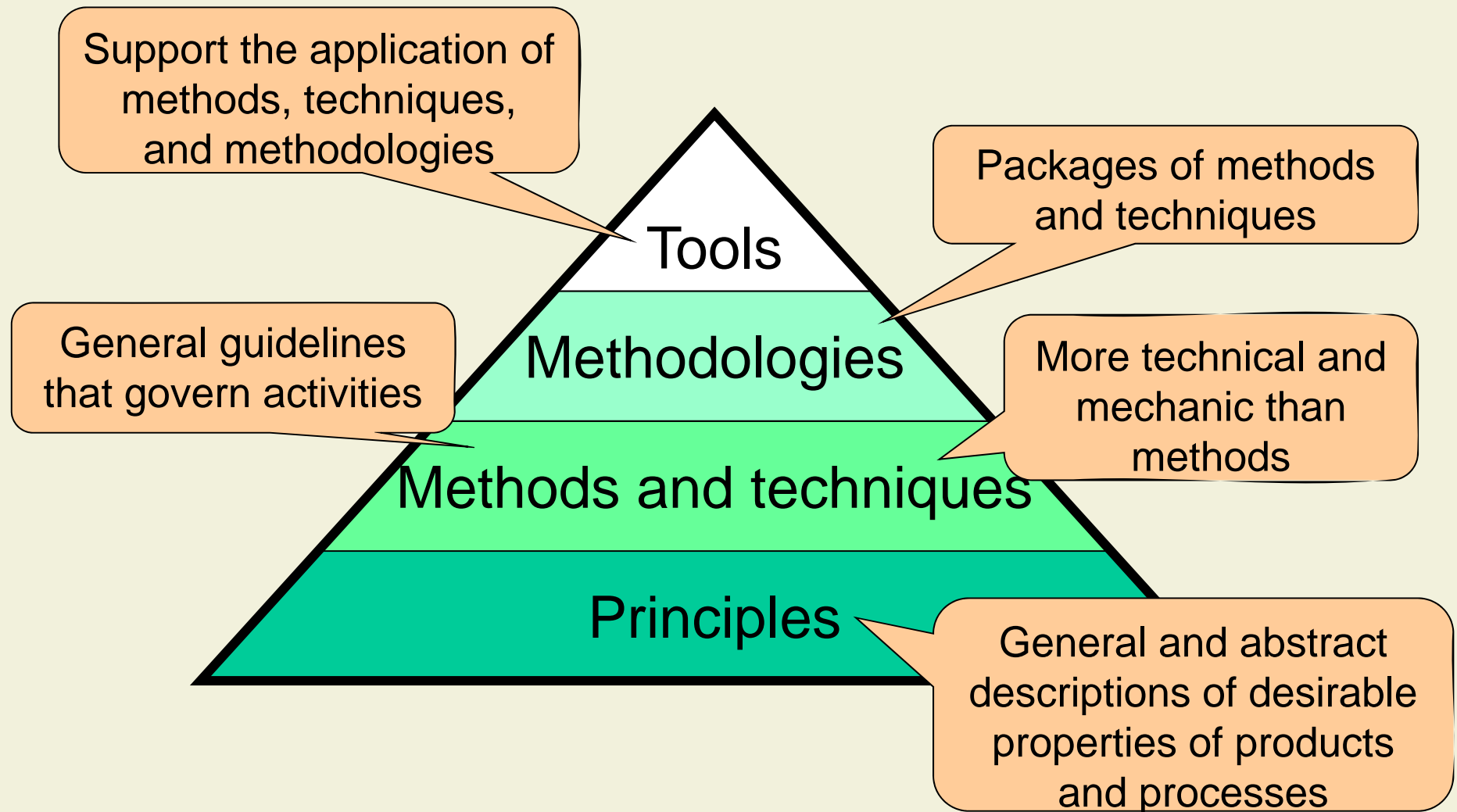Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

1. Introduction

1.1 Course Outline

1.2 Motivation

1.3 Software Qualities

1.4 Software Engineering Principles

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

The Role of Principles

Support the application of methods, techniques, and methodologies

Packages of methods and techniques

General guidelines that govern activities

More technical and mechanic than methods

Tools

Methodologies

Methods and techniques

Principles

General and abstract descriptions of desirable properties of products and processes
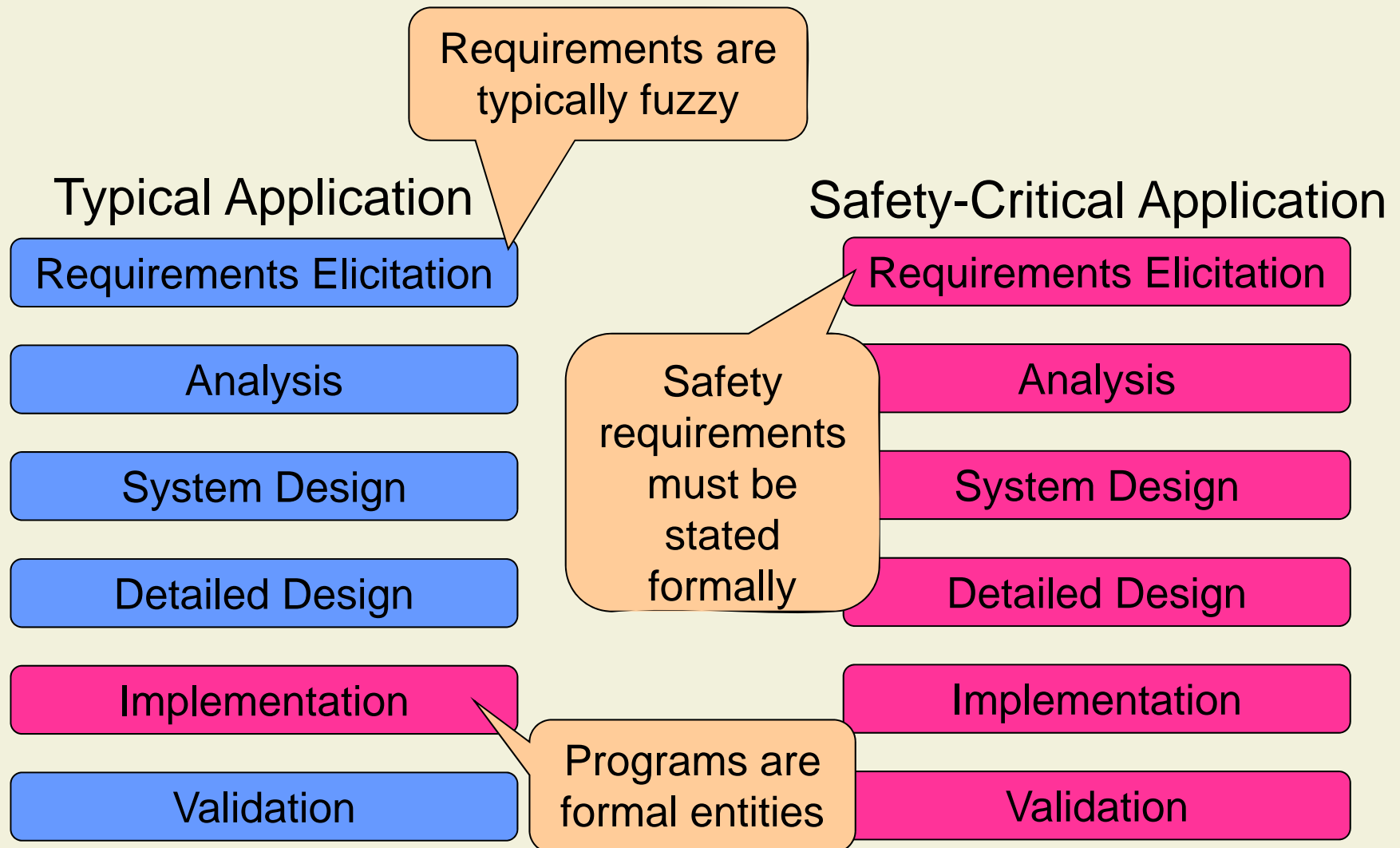
Important Software Engineering Principles

- Rigor and formality

- Separation of concerns

- Modularity

- Abstraction

- Anticipation of change

- Generality

- Incrementality

Rigor and Formality

- **Rigor means strict precision**
  - Various degrees of rigor can be achieved
  - Example: mathematical proofs

- **Formality is the highest degree of rigor**
  - Development process driven and evaluated by mathematical laws
  - Examples: refinement
  - Formality enables tool support

- Degree of rigor **depends on application**

ETH
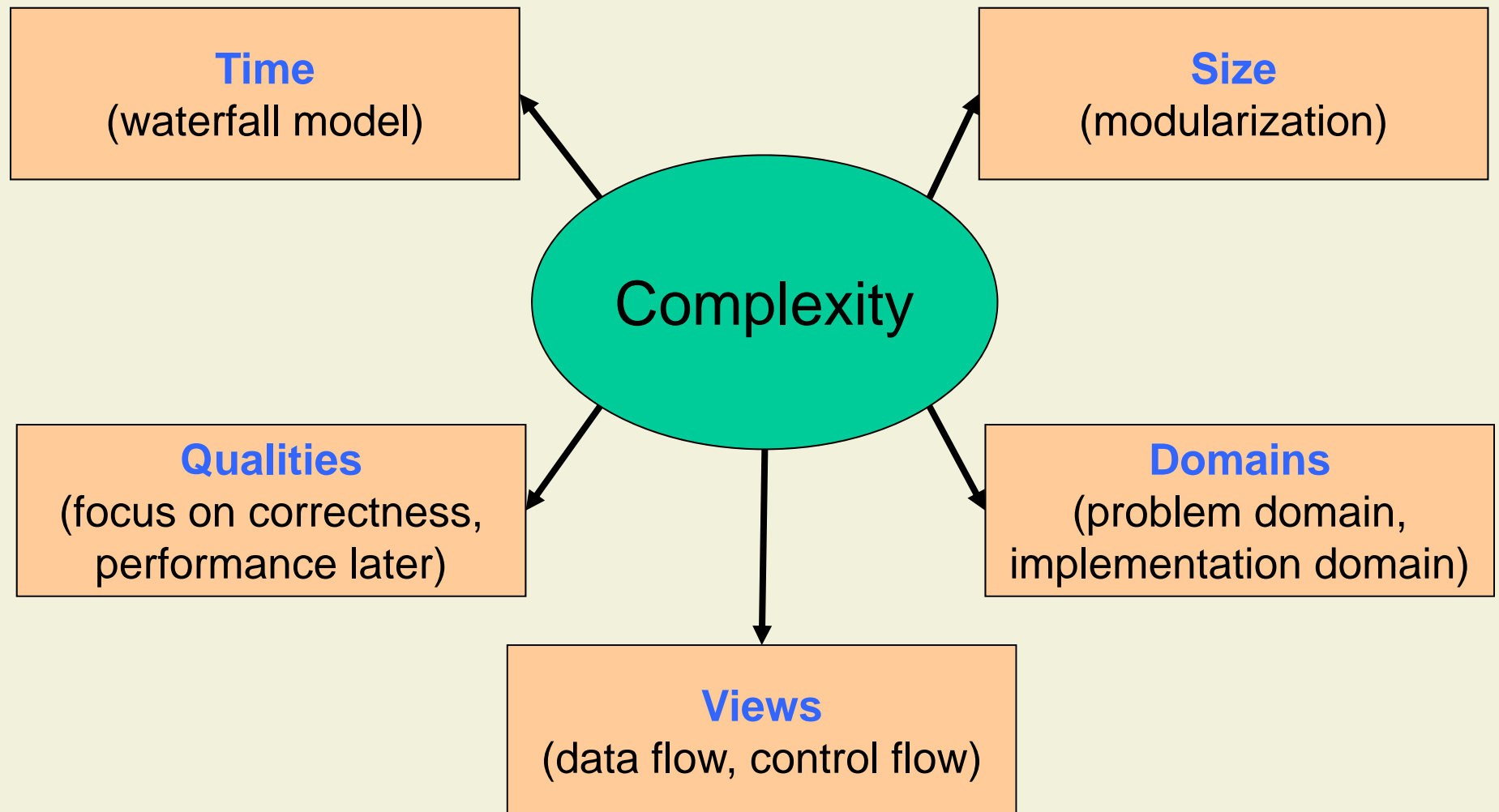Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Rigor and Formality: Examples

Requirements are typically fuzzy

Typical Application

Safety-Critical Application

Requirements Elicitation

Requirements Elicitation

Analysis

Safety requirements must be stated formally

Analysis

System Design

System Design

Detailed Design

Detailed Design

Implementation

Implementation

Programs are formal entities

Validation

Validation

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Rigor and Formality: Compiler Case Study

- **Compilers are critical products**

  - Errors are multiplied on a mass scale

- **Very high degree of formalization**

  - Syntax: regular expressions, grammars, BNF

  - Semantic analysis: attribute grammars

- **Formalization enables tool support**

  - Scanner generators (lex)

  - Parser generators (yacc)

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Separation of Concerns

- Deal with **different aspects** of a problem **separately**

  - **Reduce complexity**

  - Functionality, reliability, performance, environment, etc.

- Many aspects are **related** and **interdependent**

  - Separate unrelated concerns

  - Consider only the relevant details of a related concern

- **Tradeoff**

  - Risk to miss global optimizations

  - Chance to make optimized decisions in the face of complexity is very limited

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Ways to Achieve Separation of Concerns

Time
(waterfall model)

Size
(modularization)

Complexity

Qualities
(focus on correctness,
performance later)

Domains
(problem domain,
implementation domain)

Views
(data flow, control flow)

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Separation of Concerns: Compiler Case Study

- **Correctness** is primary concern

- Other concerns
  - **Efficiency** of compiler and of generated code
  - **User friendliness** (helpful warnings, etc.)

- Example for interdependencies:
  runtime diagnostics vs. efficient code
  - Example: runtime assertion checking
  - Diagnostics simplify testing, but create overhead
  - Typical solution: option to disable checks

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
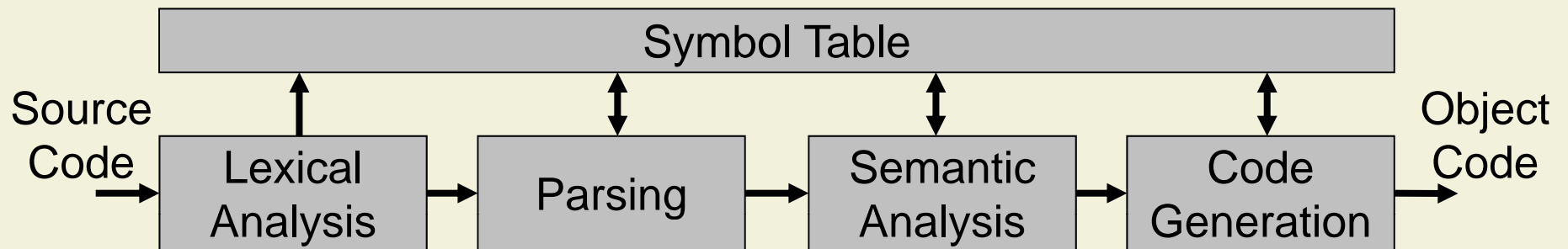
Modularity

- Divide system into modules to **reduce complexity**

- **Decompose** a complex system into simpler pieces
- **Compose** a complex system from existing modules
- **Understand** the system in terms of its pieces
- **Modify** a system by modifying only a small number of its pieces

- See Software Architecture, Lecture 2

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cohesion and Coupling

- **Cohesion** measures **interdependence** of the elements of **one module**

- **Coupling** measures **interdependence between** different **module**

- Goal: **high cohesion** and **low coupling**

Modularity: Compiler Case Study

Symbol Table			

Source
Code → **Lexical Analysis** → **Parsing** → **Semantic Analysis** → **Code Generation** → Object Code

- Compilers are modularized into **phases**
- Each phase has precisely defined input and output
  - High cohesion: common functionality in each phase
  - Low coupling: pipe-and-filter architecture, symbol table

Abstraction

- **Identify** the **important aspects** and **ignore** the **details**

- Abstraction in software engineering
  - **Models** of the real world (omit irrelevant details)
  - **Subtyping** and inheritance (factor out commonalities)
  - Interfaces and **information hiding** (hide implementation details)
  - Parameterization (templates)
  - Structured programming (loops, methods)
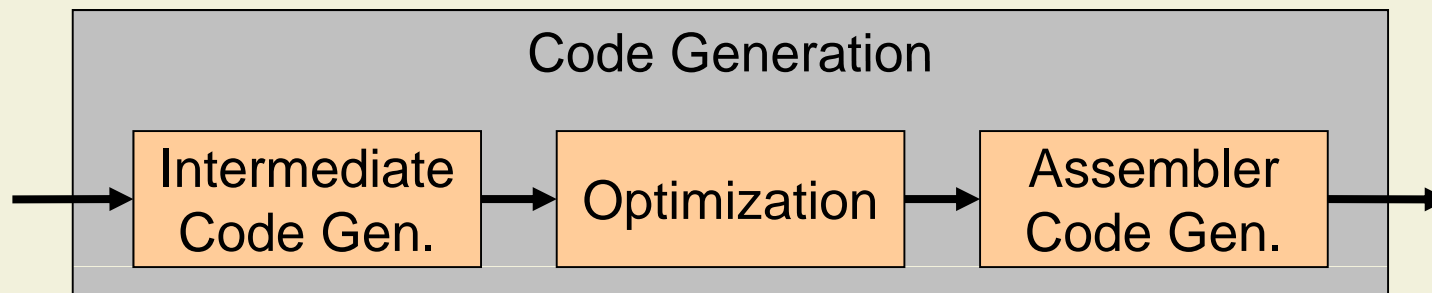 - Layered systems (hide deeper layers in the stack)

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstraction: Compiler Case Study

- ## Abstract syntax

  - Abstract while loop syntax: while(*BoolExpr Stmt*)
  - Concrete Pascal syntax: **WHILE** *BoolExpr* **DO** *Stmt* **;**
  - Concrete Java syntax: **while (** *BoolExpr* **)** *Stmt*

- ## Abstract machines

  - Generate intermediate code for abstract machine
  - Simplifies code generation for different hardware

Code Generation

| Intermediate Code Gen. | → | Optimization | → | Assembler Code Gen. |

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Anticipation of Change

- **Prepare** software **for changes**
  - **Modularization**: single out elements that are likely to change in the future
  - **Abstraction**: narrow interfaces reduce effects of a change

- Risk: developers spend too much time to make software changeable and reusable

- Software product lines
  - Many similar versions of software (e.g., for different hardware)
  - Examples: software for cell phones, sensors

Anticipation of Change: Example

- Fee computation for bank accounts

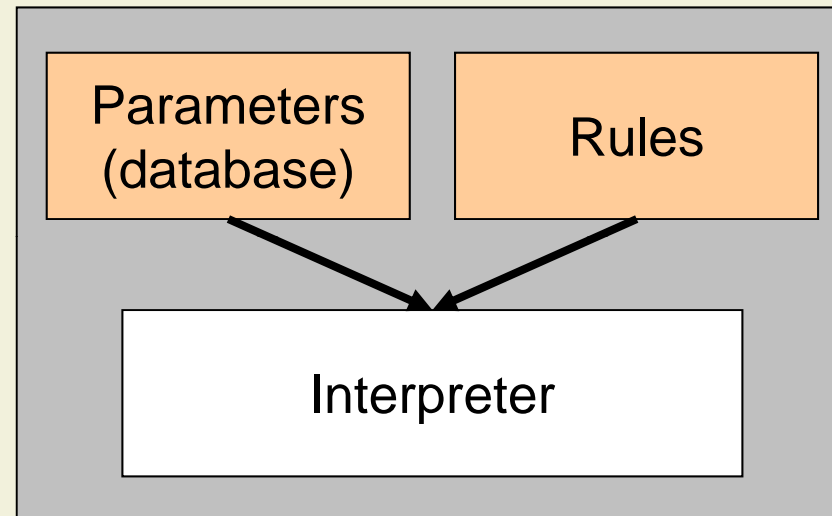- Original design: computation and values hard-coded

```
int computeFee( ) {
  if( balance >= 2000)
    return 0;
  else
    return monthlyFee;
}
```

- Changes (within two years)
  - Different values $\rightarrow$ required program change
  - Different rules $\rightarrow$ required program change
  - Different groups of clients $\rightarrow$ required additional logic

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Anticipation of Change: Example (cont'd)

- **Better design: interpreter**

- **Parameters and rules can be changed by banker**
  - For instance, by editing an Excel file
- **Code remains unchanged (less testing)**

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Anticipation of Change: Compiler Case Study

- ## Typical changes

  - New versions of processors and operating systems

  - New target machines
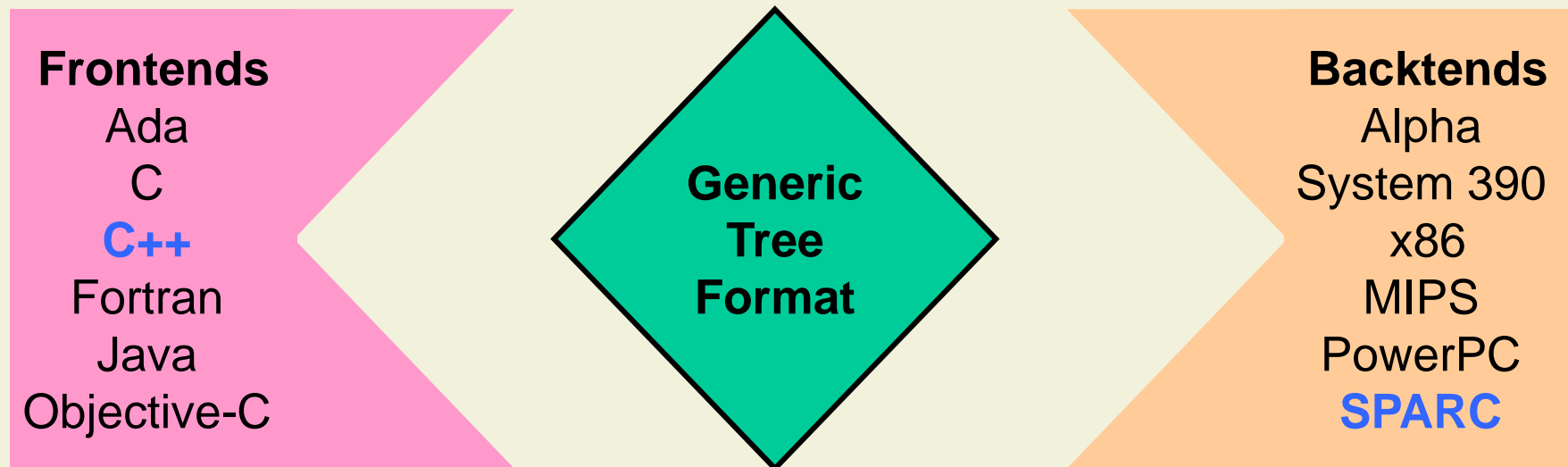
  - Language and library extensions (e.g., standards)

- ## Preparation

  - Use intermediate code

  - Put machine-dependent code (e.g., I/O, threads) into standard library

Generality

- **Attempt to find more general problem behind problem at hand**

  - Apply **standard solutions and tools**

- **A general solution is more likely to be reusable**

  - Examples: spreadsheets, database

- **General solution may be less efficient**

- **Example**

  - Semantic analysis: Is C a subclass of D?

  - Subclass relation is an acyclic graph

  - Use adjacency matrix and compute transitive closure

Generality: Compiler Case Study

- ## The GNU compiler decouples

  - Frontend (scanner, parser, analysis)

  - Backend (code generation, optimization)

- ## Frontends and backends can be combined in various ways

Frontends
Ada
C
C++
Fortran
Java
Objective-C

Generic Tree Format

Backtends
Alpha
System 390
x86
MIPS
PowerPC
SPARC

Incrementality

- **Characterizes a process which proceeds in a stepwise fashion**
  - The desired goal is reached by creating successively closer **approximations** to it

- Examples
  - Incremental software life cycles (e.g., spiral model)
  - Prototypes, early feedback
  - Project management is inherently incremental

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Incrementality: Compiler Case Study

- **Language can be extended incrementally**
  - Java 1.0: core language
  - Java 1.1: inner classes
  - Java 1.2: Swing GUI library
  - Java 1.4: enhanced libraries
  - Java 5: genericity, boxing
  - Java 6: annotations

- **Compiler can be enhanced incrementally**
  - Supported language subset
  - Runtime diagnostics
  - Optimizations

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich