Konzepte objektorientierter Programmierung

Prof. Dr. Peter Müller

Software Component Technology

Wintersemester 03/04

ETH

Eidgenössische Technische Hochschule Zürich

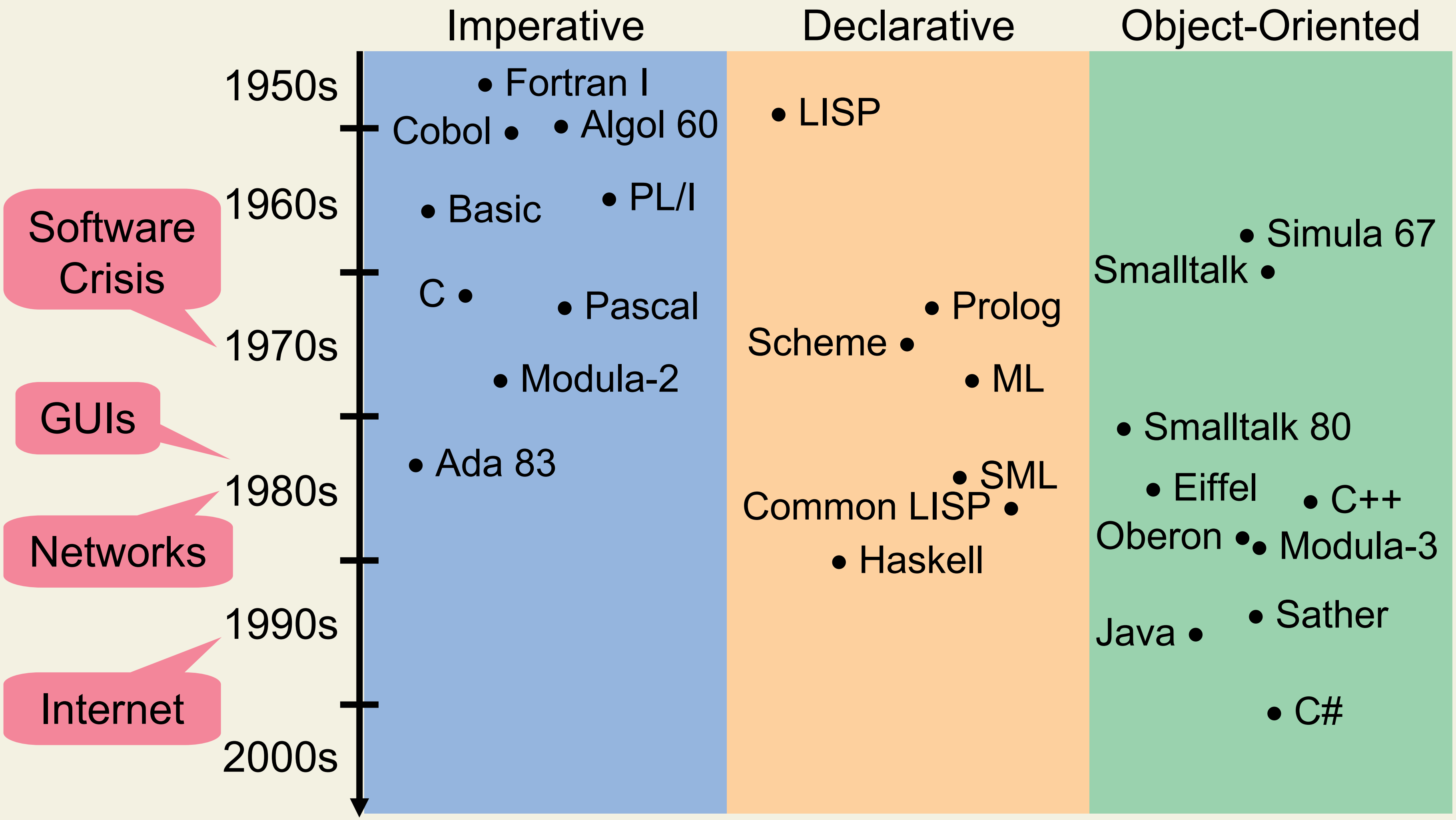
Swiss Federal Institute of Technology Zurich

Programming Paradigms

- Definition of *Paradigm*:
  A theoretical framework of a scientific discipline consisting of concepts, methods, techniques, theories, and standards.

- Imperative / procedural
  – we ask “how”
- Declarative (functional, logic)
  – we ask “what”
- Object-oriented
  – we ask “who”

History of Programming Languages



Agenda for Today

1. Introduction

1.1 Requirements

1.2 Concepts

1.3 Course Outline

Objectives

- Motivation for object-oriented programming
- Distinction between core concepts, language concepts, and language constructs

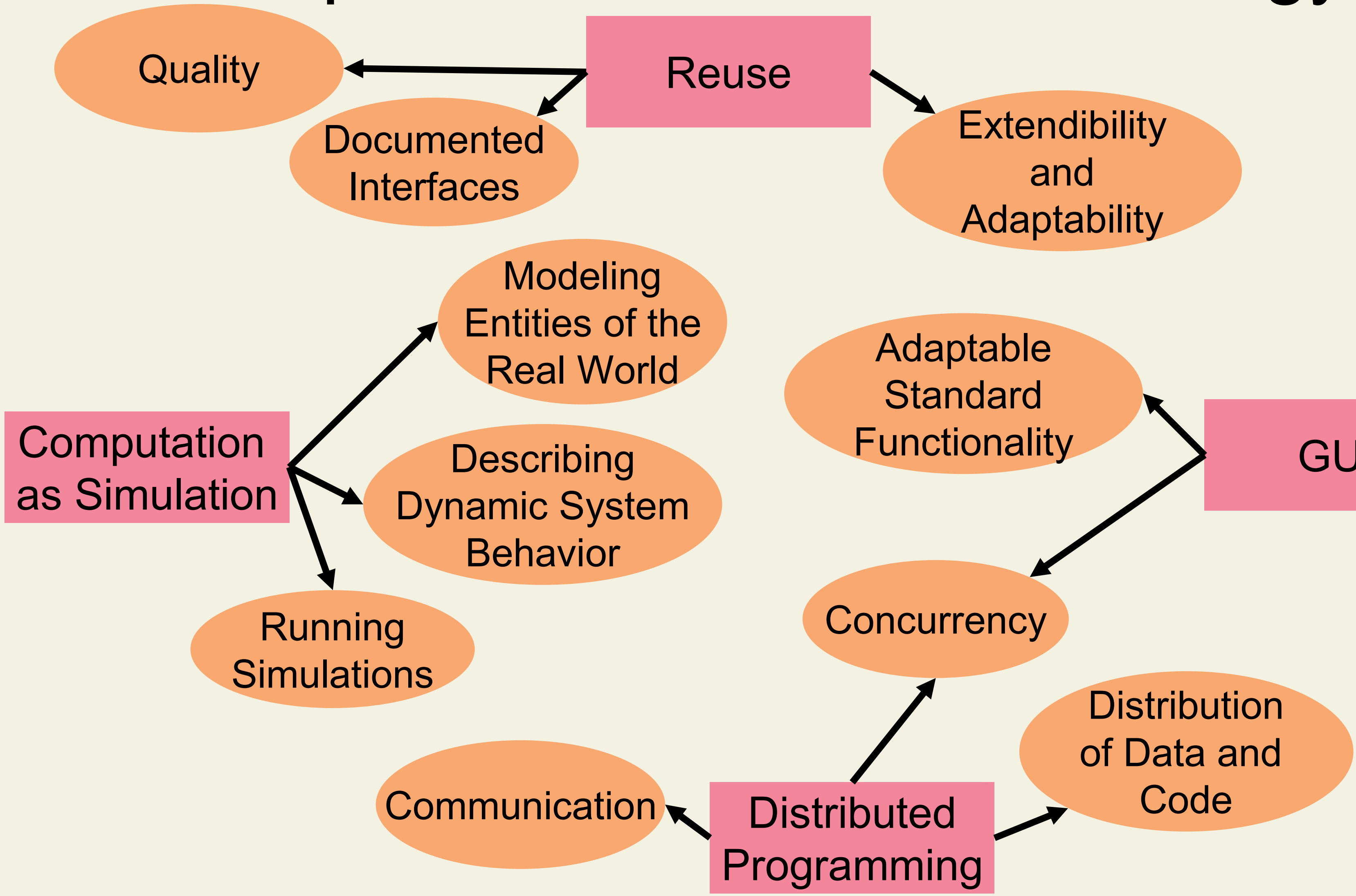
1. Introduction

1.1 Requirements

1.2 Concepts

1.3 Course Outline

New Requirements in SW-Technology



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Example: Reusing Imperative Programs

- Scenario: University Administration System
  - Models students and professors
  - Stores one record for each student and each professor in a repository
  - Procedure printAll prints all records in the repository

An Implementation in C

```
typedef struct {
  char *name;
  char *room;
  char *institute;
} Professor;

typedef struct {
  char *name;
  int   reg_num;
} Student;
```

```
void printStudent( Student *s )
  { … }

void printProf( Professor *p )
  { … }
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

An Implementation in C (cont'd)

```
typedef struct {
  enum { STU,PROF } kind;
  union  {
    Student *s;
    Professor *p;
  } u;
} Person;

typedef Person **List;
```

```
void printAll( List l ) {
  int i;
  for ( i=0; l[ i ] != NULL; i++ )
    switch ( l[ i ] -> kind ) {
    case STU:
      printStudent( l[ i ] -> u.s );
      break;
    case PROF:
      printProf( l[ i ] -> u.p );
      break;
    }
}
```

Extending and Adapting the Program

- Scenario: University Administration System
  - Models students and professors
  - Stores one record for each student and each professor in a repository
  - Procedure printAll prints all records in the repository
- Extension: Add assistants to system
  - Add record and print function for assistants
  - Reuse old code for repository and printing

Step 1: Add Record and Print Function

```
typedef struct {
  char *name;
  char *room;
  char *institute;
} Professor;

typedef struct {
  char *name;
  int   reg_num;
} Student;

typedef struct {
  char *name;
  char  PhD_student;  /* ‘y‘, ‘n‘ */
} Assistant;
```

```
void printStudent( Student *s )
  { … }

void printProf( Professor *p )
  { … }

void printAssi( Assistant *a )
  { … }
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Step 2: Reuse Code for Repository

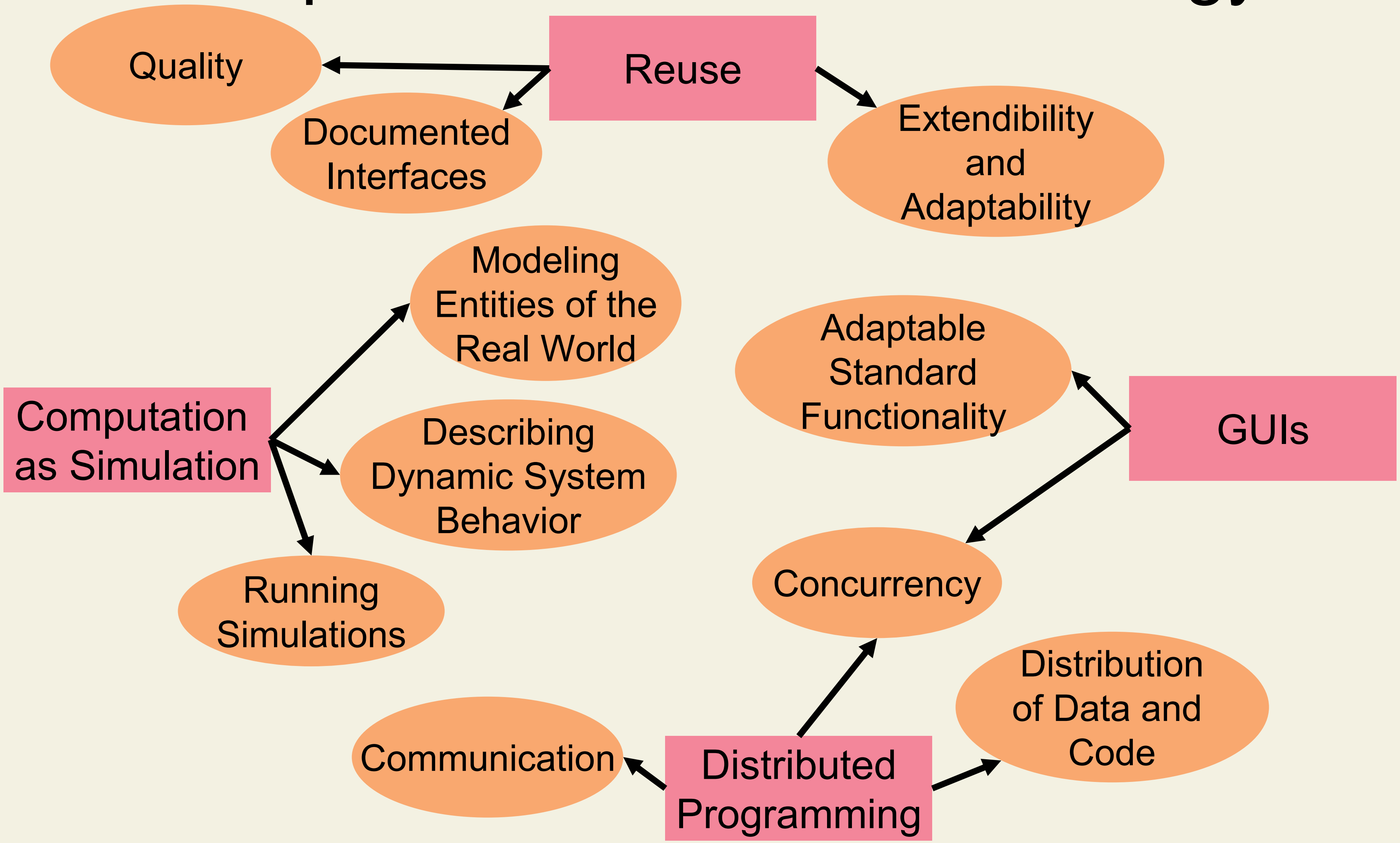
```
typedef struct {
  enum { STU,PROF,ASSI } kind;
  union  {
    Student *s;
    Professor *p;
    Assistant *a;
  } u;
} Person;

typedef Person **List;
```

```
void printAll( List l ) {
  int i;
  for ( i=0; l[ i ] != NULL; i++ )
    switch ( l[ i ] -> kind ) {
    case STU:
      printStudent( l[ i ] -> u.s );
      break;
    case PROF:
      printProf( l[ i ] -> u.p );
      break;
    case ASSI:
      printAssi( l[ i ] -> u.a );
      break;
    }
}
```

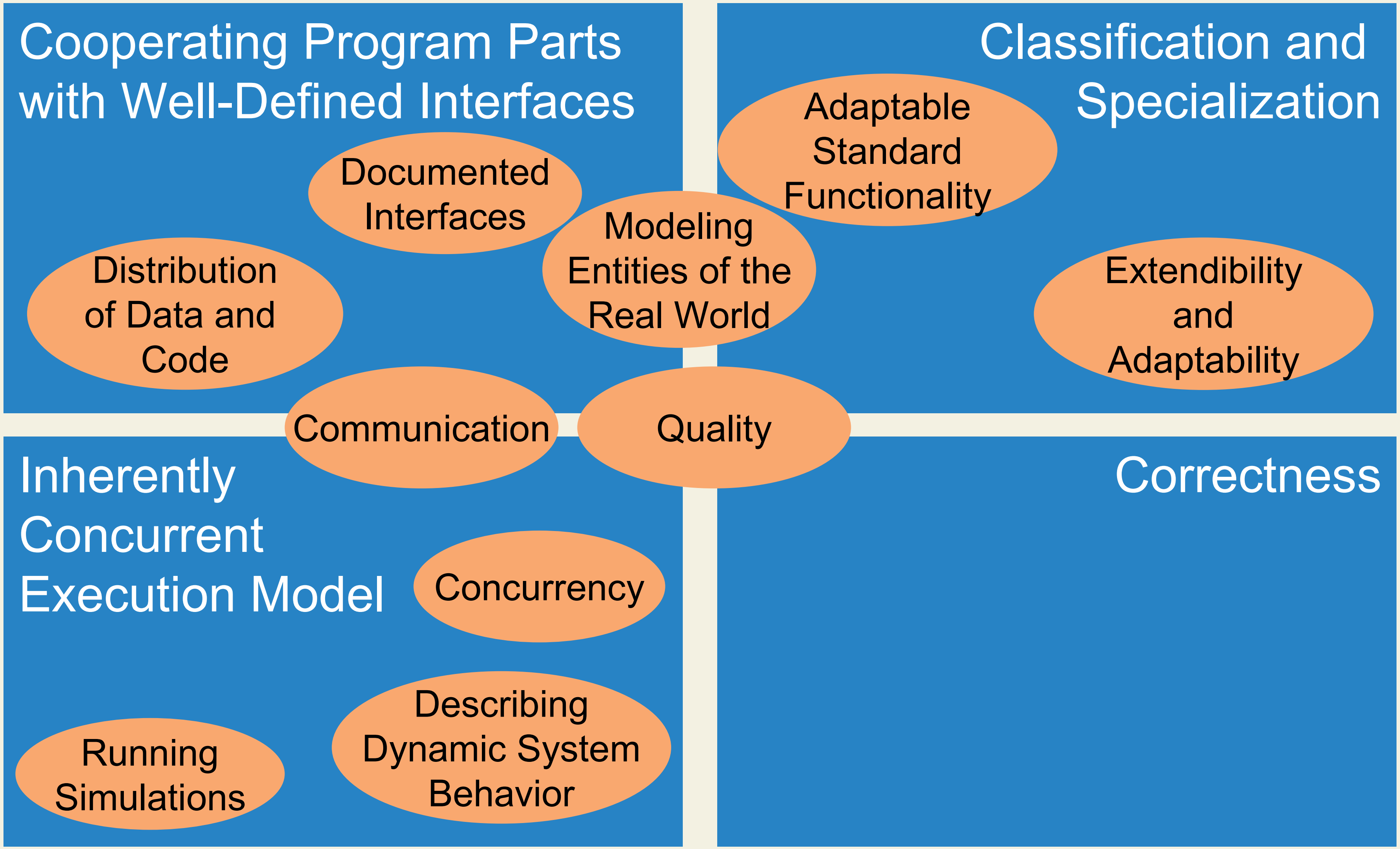
Reuse in Imperative Languages

- No explicit language support for extension and adaptation
- Adaptation usually requires modification of reused code
- Copy-and-paste reuse
  - Code duplication
  - Difficult to maintain
  - Error-prone

New Requirements in SW-Technology



Core Requirements



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

From Requirements to Concepts

What are the concepts of a programming paradigm

- That allow one to express **concurrency** naturally?
- That structure programs into **cooperating program parts with well-defined interfaces**?
- That are able to express **classification and specialization** of program parts without modifying reused code?
- That facilitate the development of **correct programs**?

1. Introduction

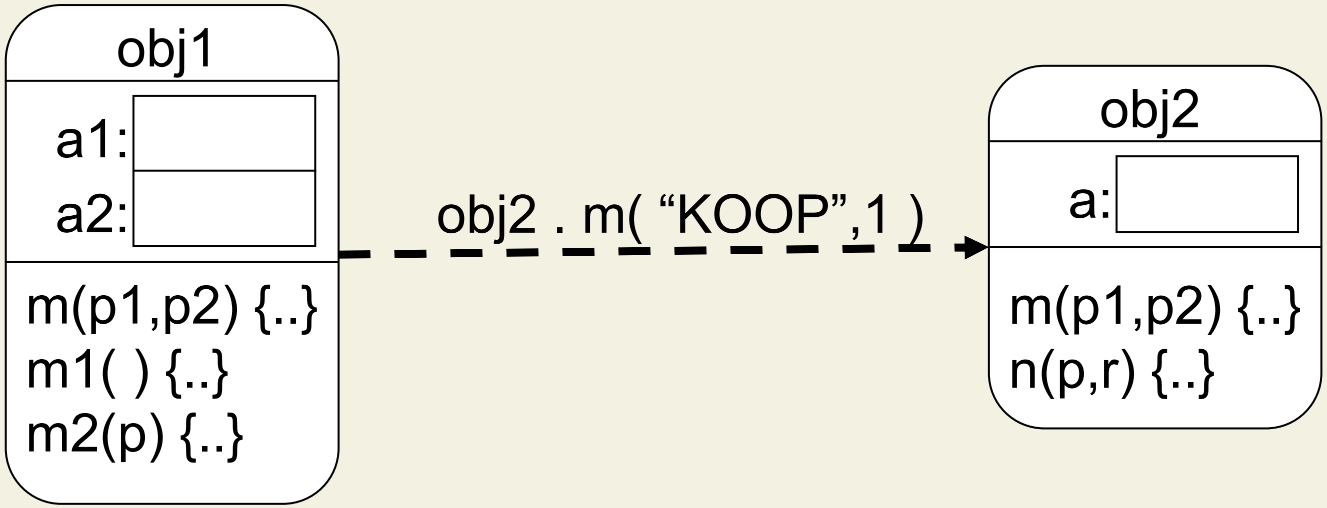
1.1 Requirements

1.2 Concepts

1.3 Course Outline

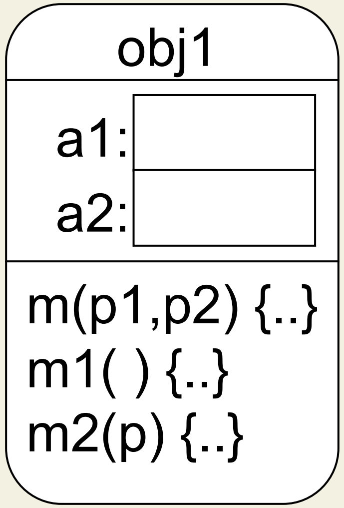
The Object Model

- A software system is a set of cooperating objects
- Objects have state and processing ability
- Objects exchange messages



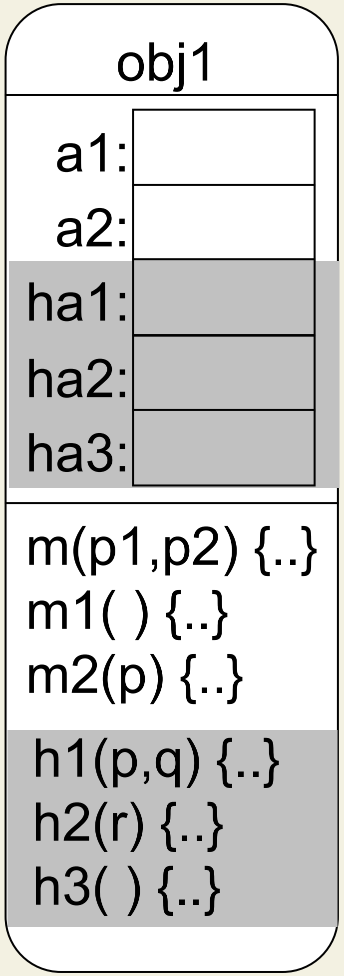
Interfaces and Encapsulation

- Objects have well-defined interfaces
  - Publicly accessible attributes
  - Publicly accessible methods
- Implementation is hidden behind interface
  - Encapsulation
  - Information hiding
- Interfaces are the basis for describing behavior



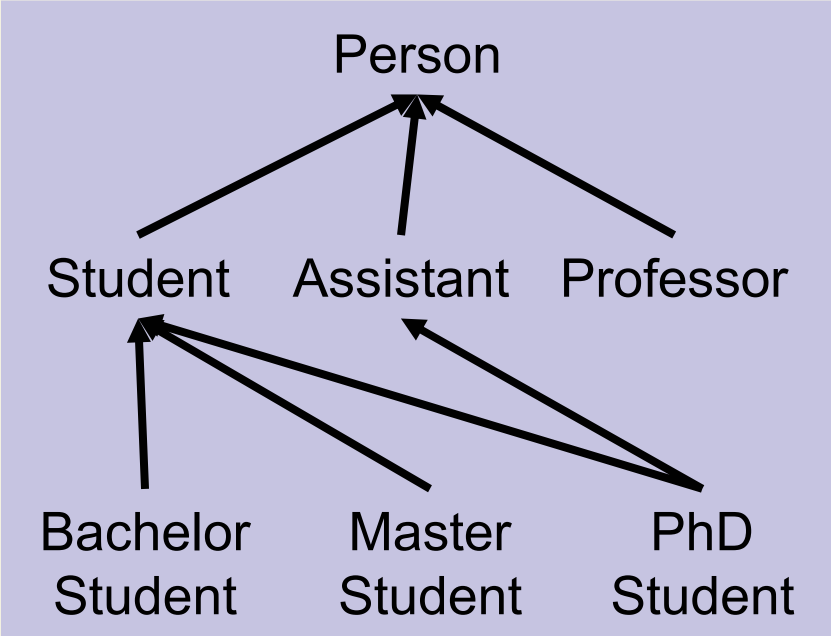
Interfaces and Encapsulation

- Objects have well-defined interfaces
  - Publicly accessible attributes
  - Publicly accessible methods
- Implementation is hidden behind interface
  - Encapsulation
  - Information hiding
- Interfaces are the basis for describing behavior



Classification and Polymorphism

- Classification: Hierarchical structuring of objects
- Objects belong to different classes simultaneously
- Substitution principle: Subtype objects can be used wherever supertype objects are expected



Meeting the Requirements

Cooperating Program Parts with Well-Defined Interfaces

→ Objects (data + code)
→ Interfaces
→ Encapsulation

Classification and Specialization

→ Classification, subtyping
→ Polymorphism
→ Substitution principle

Inherently Concurrent Execution Model

→ Active objects
→ Message passing

Correctness

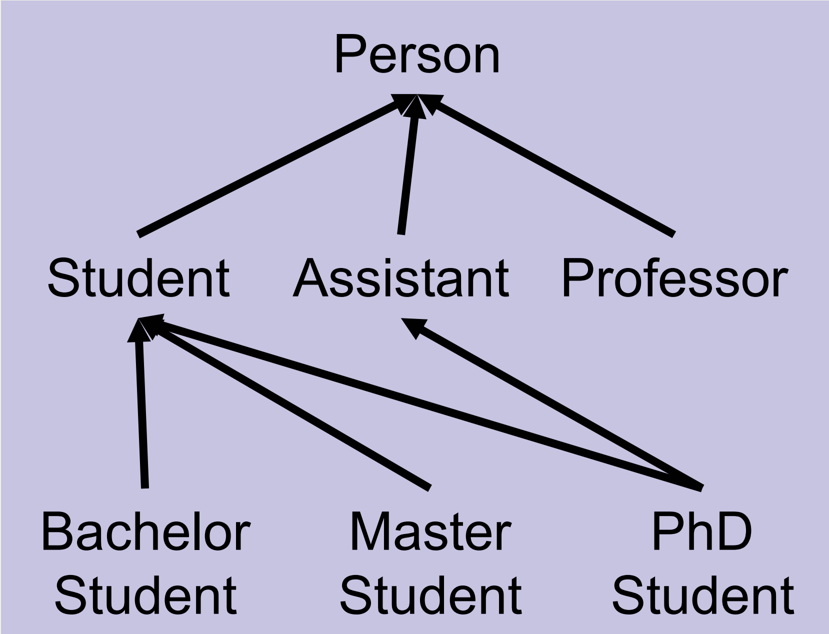
→ Interfaces
→ Encapsulation
→ Simple, powerful concepts

Core Concepts: Summary

- Core concepts of the OO-paradigm
  - Object model
  - Interfaces and encapsulation
  - Classification and polymorphism
- Core concepts are **abstract concepts** to meet the new requirements
- To apply the core concepts we need ways to **express them in programs**
- **Language concepts** enable and facilitate the application of the core concepts

Example: Dynamic Method Binding

- Classification and polymorphism
  - Algorithms that work with supertype objects can be used with subtype objects
  - Subclass objects are specialized
- Dynamic binding: Method implementation is selected at runtime, depending on the type of the receiver object




```
void printAll( Person[ ] l ) {
  for (int i=0; l[ i ] != null; i++)
    l[ i ] . print( );
}
```

OO-Concepts and Imperative Languages

- What we have seen so far
  - New concepts are needed to meet **new requirements**
  - **Core concepts** serve this purpose
  - **Language concepts** are needed to express core concepts in programs
- Open questions
  - Why do we need **OO-programming languages**?
  - Can’t we use the language concepts as **guidelines** when writing imperative programs?
- Let’s do an experiment …
  - Writing object-oriented programs in C

Types and Objects

- Declare types

```
typedef char*               String;
typedef struct sPerson Person;
```

- Declare records with
  - Attributes
  - Methods (function pointers)

```
struct sPerson {
  String name;
  void   ( *print )( Person* );
  String ( *lastName )( Person* );
};
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Methods and Constructors

- Define methods

```
void printPerson( Person *this ) {
  printf("Name: %s\n", this->name);
}

String LN_Person( Person *this )
  { … }
```

- Define constructors

```
Person *PersonC( String n ) {
  Person *this = (Person *)
              malloc( sizeof( Person ) );
  this- > name       = n;
  this -> print      = printPerson;
  this -> lastName   = LN_Person;
  return this;
}
```

Using the "Object"

- Declaration

```
struct sPerson {
  String name;
  void    ( *print )( Person* );
  String  ( *lastName )( Person* );
};
```

- Use constructors, attributes, and methods

```
Person *p;
p = PersonC( "Werner Dietl" );
p->name = p->lastName( p );
p->print( p );
```

Inheritance and Specialization

- Copy code
- Adapt function signatures

```
typedef struct sStudent Student;
struct sStudent {
  String name;
  void   ( *print )( Student* );
  String ( *lastName )( Student* );
  int reg_num;
};
```

- Define specialized methods

```
void printStudent( Student *this ) {
  printf(“Name: %s\n“, this->name);
  printf(“No: %d\n“, this->reg_num);
}
```

Inheritance and Specialization (cont'd)

- Reuse LN_Person for Student
- View Student as Person (cast)

```
Student *StudentC( String n, int r ) {
  Student *this = (Student *)
            malloc( sizeof( Student ) );

  this -> name        = n;
  this -> print       = printStudent;

  this -> lastName    =
    (String (*)(Student*)) LN_Person;
  this -> reg_num     = r;

  return this;
}
```

Subclassing and Dynamic Binding

- Student has all attributes and methods of Person
- Casts are necessary

```
Student *s;
Person *p;
s = StudentC( “Werner Dietl“ );
p = (Person *) s;
p -> name = p -> lastName( p );
p -> print( p );
```

- Array l can contain Person and Student objects
- Methods are selected dynamically

```
void printAll( Person **l ) {
  int i;
  for ( i=0; l[ i ] != NULL; i++ )
    l[ i ] -> print( l[ i ] );
}
```

Discussion of the C Solution: Pros

- We can express **objects**, **attributes**, **methods**, **constructors**, and **dynamic method binding**
- By imitating OO-programming, the union in Person and the switch statement in printAll became dispensable
- The behavior of reused code (Person, printAll) can be **adapted** (to introduce Student) **without changing the implementation**

Discussion of the C Solution: Cons

- Inheritance has to be replaced by **code duplication**
- Subclassing can be simulated, but it requires
  - Casts, which is **not type safe**
  - **Same memory layout** of super and subclasses (same attributes and function pointers in same order), which is **extremely error-prone**
- Appropriate language support is needed to apply object-oriented concepts

A Java Solution

```
class Person {
  String    name;
  void      print( ) {
    System.out.println("Name: " +
     name);
  }
  String    lastName( ) { … }
  Person( String n )    { name = n; }
}
```

```
void printAll( Person[ ] l ) {
  for (int i=0; l[ i ] != null; i++)
    l[ i ].print( );
}
```

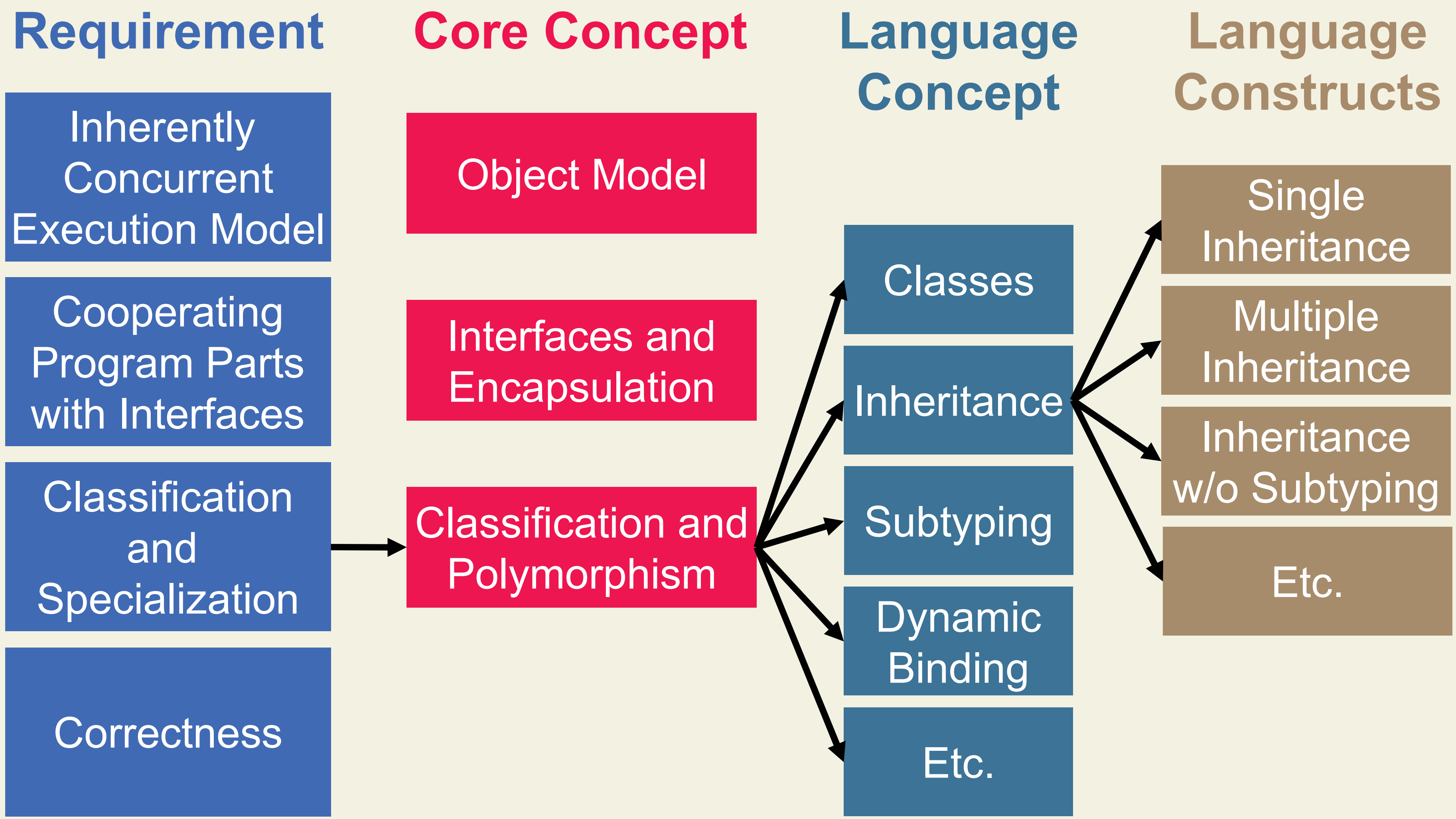
```
class Student extends Person {
  int      reg_num;
  void     print( ) {
    super.print();
    System.out.println("No: " +
     reg_num);
  }
  Student( String n, int i ) {
    super( n );
    reg_num = i;
  }
}
```

Discussion of the Java Solution

- The Java solution uses
  - **Inheritance** to avoid code duplication
  - **Subtyping** to express classification
  - **Overriding** to specialize methods
  - **Dynamic binding** to adapt reused algorithms
- Java supports the OO-language concepts
- The Java solution
  - Is simpler and smaller
  - Easier to maintain (no duplicate code)
  - Type safe

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Concepts: Summary



1. Introduction

1.1 Requirements

1.2 Concepts

1.3 Course Outline

After this Course, you should be able

- To understand the core concepts
- To compare OO-languages

- To understand the language concepts
- To learn new languages faster
- To apply language concepts correctly

- To write better object-oriented programs

Approach

- We discuss the
  - **Concepts** of
    as opposed to languages, implementations, etc.
  - **Object-Oriented**
    as opposed to imperative, declarative
  - **Programming**
    as opposed to analysis, design, etc.

- We try to be language-independent
  - However, Java is used for examples and exercises
- We look at code and analyze programs

Course Outline (tentative)

2. Core and Basic Language Concepts
3. Reusable Components
4. Frameworks
5. Information Hiding and Encapsulation
6. Object Structures and Aliasing
7. Static Safety and Extended Typing
8. Concurrency and Distribution
9. Remote Method Invocation
10. Dynamic Class Loading and Mobile Code
11. Contracts
12. Testing and Extended Static Checking
13. Formal Verification
14. Outlook

Classification and Polymorphism

Interfaces and Encapsulation

Object Model

Correctness

Using this Course

- Master Program:
  This course is a **Fokusfach** for the Major in Software Engineering
- Diploma Program:
  This course is a **Specialized Course** (Vertiefungsfach)

Related Courses

- Informatik IV (SS)
- Advanced Topics in Object-Oriented Development (Meyer, SS)
- Trusted Components (Meyer, WS)
- Semantik von Programmiersprachen (Müller, SS)
- Verteilte Systeme (Mattern, Alonso, Wattenhofer, WS)
- C++ Templates and Generic Programming (Zueff, WS)
- Software Design (Gruntz, SS)

Related Seminars

- Specification and Verification of Object-Oriented Software (Biere, Müller, WS)
- Referenzen und Aliasing in objektorientierten Programmen (Müller, SS)
- FATS Formal Approaches to Software (Biere, Meyer, Müller, Stärk)

Literature

- Poetzsch-Heffter, Arnd: Konzepte objektorientierter Programmierung. Springer-Verlag, 2000
- Budd, Timothy: An Introduction to Object-Oriented Programming. Addison-Wesley, 1991
- Meyer, Bertrand: Object-Oriented Software-Construction (2nd edition). Prentice Hall, 1997

- Horstmann, Cay S. and Cornell, Gary: Core Java, Band 1 – Grundlagen. Markt+Technik, 2003

- See course web page for a comprehensive list