Fortgeschrittene Aspekte objektorientierter Programmierung

Advanced Aspects of Object-oriented Programmierung

Arnd Poetzsch-Heffter

AG Softwaretechnik

TU Kaiserslautern

Sommersemester 2009

0. Preliminaries

Cooperation:

The course material is based on a collaboration with Prof. Peter Müller (ETH Zürich/Microsoft Research). Several of the slides are of his lecture „Konzepte objektorientierter Programmierung.“

Organisational Information:

The course module consists of 2 hours lecture and 1 hour practical work/assignments („Übungen”).

The modul exam is an oral exam about the content of the lecture and the assignments.

There are web pages containing the course material and information about the course:

softech.informatik.uni-kl.de/

under item „Lehre“

Literature:

Will be given along with the course, some literature is already given on the web pages.

Overview and Structure of Course:

1. Introduction
2. Objects, Classes, Inheritance
3. Subtyping and Parametric Types
4. Object Structures, Aliasing and Encapsulation
5. Specification and Checking
6. Concurrency and Distribution
7. Program Frameworks
8. Component Frameworks

1. Introduction

Overview:

- The object-oriented paradigm
- Software engineering and programming challenges
- Programming object systems
- Properties of programs

1.1 The Object-Oriented Paradigm

Explanation: (Paradigm/Paradigma)

A framework consisting of concepts, methods, techniques, theories, and standards.

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

„The basic philosophy underlying object-oriented programming is to make the programs as far as possible reflect that part of the reality they are going to treat.

It is then often easier to understand and to get an overview of what is described in programs. The reason is that human beings from the outset are used to and trained in the perception of what is going on in the real world.

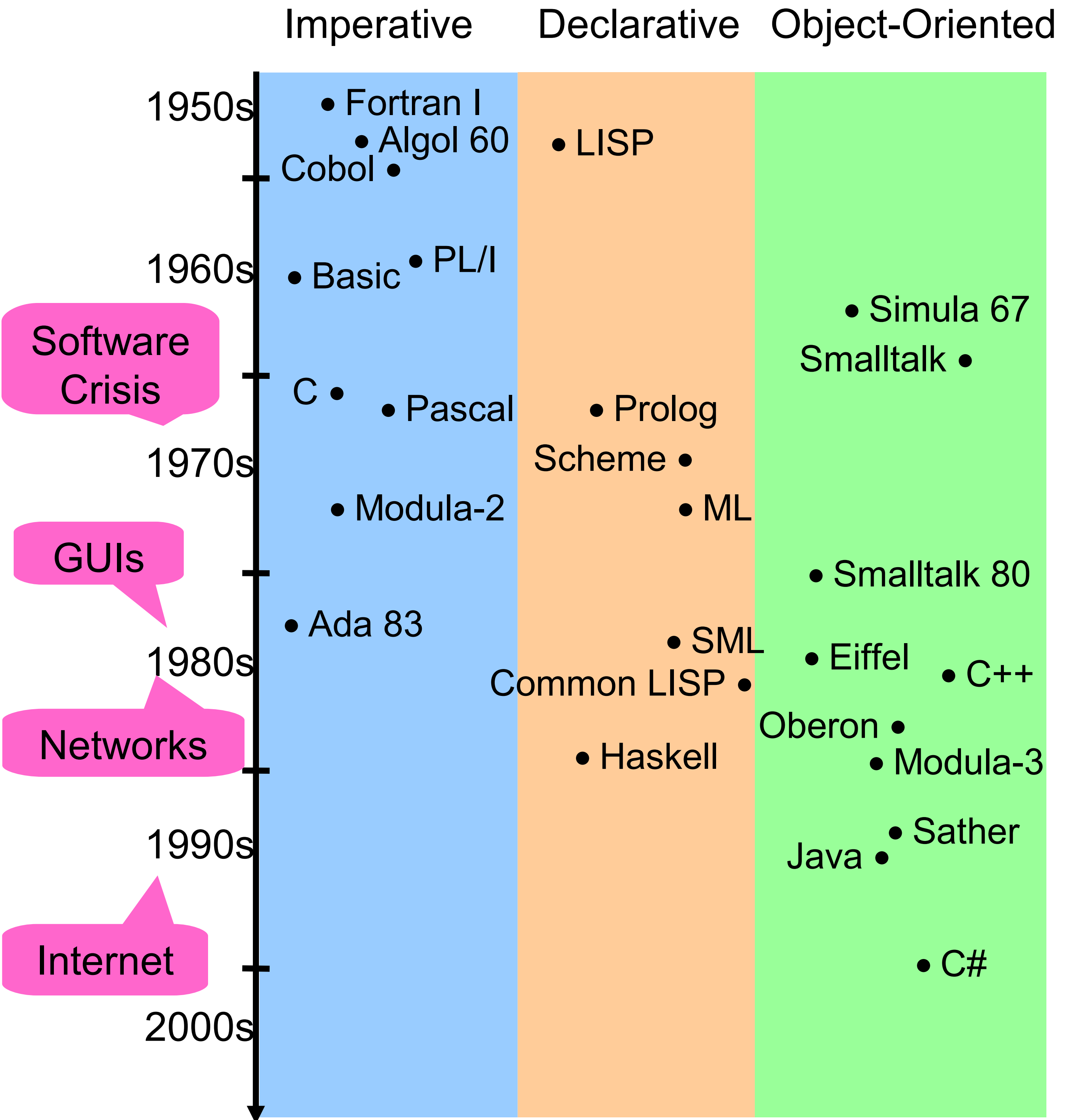
The closer it is possible to use this way of thinking in programming, the easier it is to write and understand programs.“

[from: Object-oriented Programming in the
BETA Programming Language]

„Born in the ice-blue waters of the festooned Norwegian coast; amplified along the much grayer range of the Californian Pacific; viewed by some as a typhoon, by some as a tsunami, and by some as a storm in a teacup – a tidal wave is reaching the shores of the computing world."

[from: Object-oriented Software Construction]

History of programming:



1.2 Software Engineering and Programming Challenges

Objectives of subsection:

- Motivation for object-oriented programming
- System construction vs. language concepts
- Distinction between core concepts, language concepts, and language constructs

Main challenges:

- Reuse:
  - clear, well documented interfaces
  - adaptability, extensibility
  - support for parameterized components

- Human computer interface, GUIs:
  - units with complex dynamic behavior
  - support of frameworks for core functionality
  - make the programs reflect part of the reality

- Distributed computing:
  - concurrency
  - communication
  - distributed state
  - mobile code

Reuse

We consider:

- extensibility & adaptability in a simple example:
  imperative vs. object-oriented
- abstract algorithms

Example: (adaptibility/extensibility)

Scenario: University Administration System

– Models students and professors
– Stores one record for each student and each professor in a repository/array
– Procedure printAll prints all records in the repository

Simple implementation in C:

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

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

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

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

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

typedef Person** PersonArray;

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

Extending and adapting the program:

- Old scenario: as above
- 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 for assistant

```
typedef struct {
  ... /* as above  */
}  Professor;

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

```
typedef struct {
 ... /* as above  */
}  Student;

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

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

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

Step 2: Reuse code for repository

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

typedef Person** PersonArray;

void printAll( PersonArray ar ) {
    int i;
    for ( i=0; ar[ i ] != NULL; i++ ) {
        switch ( ar[ i ] -> kind ) {
        case STU:
            printStud( ar[ i ] -> u.s );   break;
        case PROF:
            printProf( ar[ i ] -> u.p );    break;
        case ASSI:
            printAssi( ar[ 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

Abstract algorithms for reuse:

Example: (abstract algorithm)

Implement a sorting algorithm `sort` for *all* lists with elements of some type T having a procedure

```
boolean compare (T, T);
```

`sort` has the following properties:

- it does not need to know what the T‘s look like (i.e. it abstract from T‘s concrete implementation)
- it can in particular be „re“-used for new datatypes implementing `compare`
- it allows for separate development with clear interfaces

Human computer interfaces, GUIs

Tasks of user interfaces:

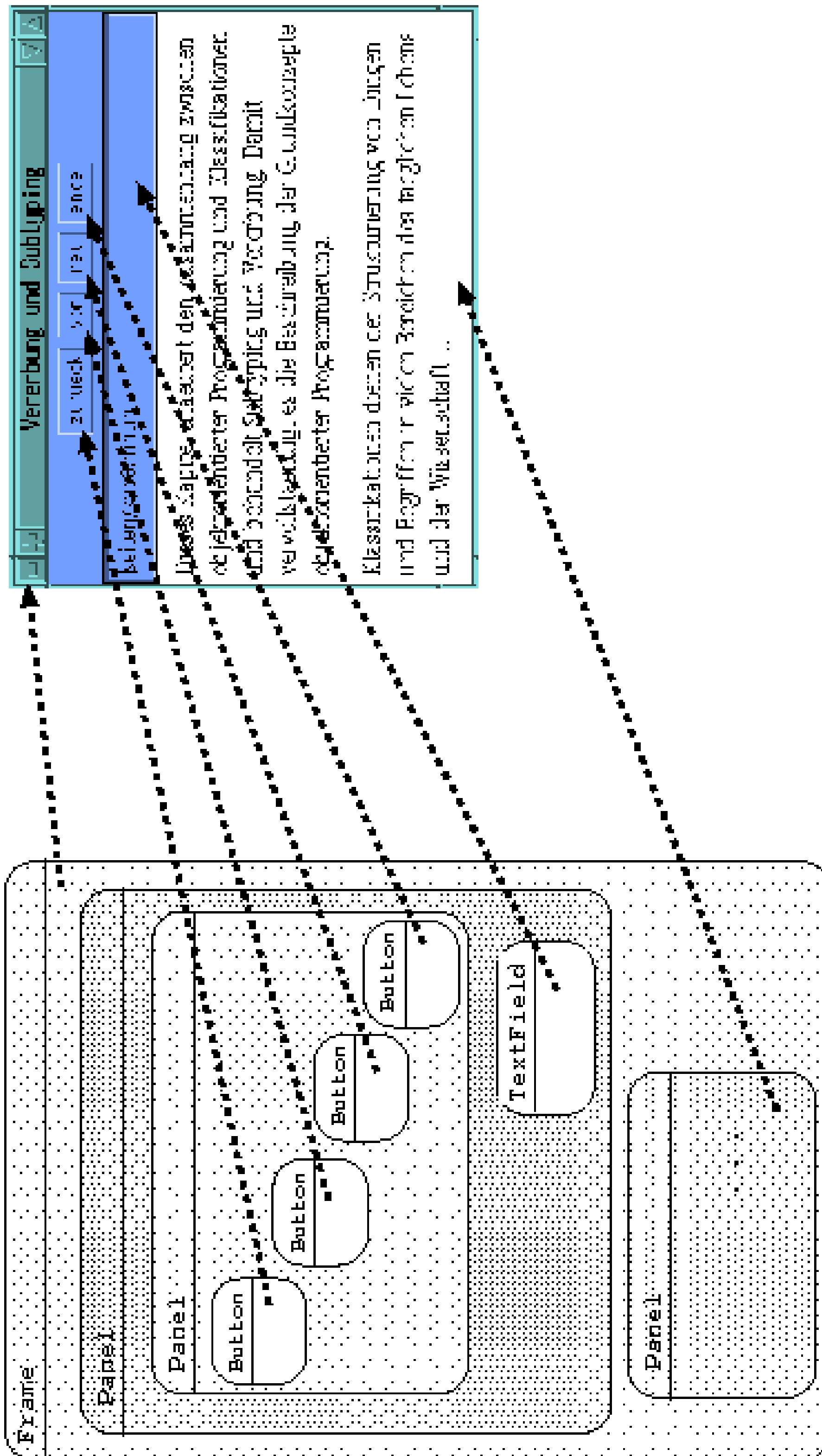
Should support flexible interaction between users and applications:

- controlling the applications usually with concurrent triggers
- input is often complex data like:
  - mouse, joy stick movements
  - speech
  - images
- output is often complex data like:
  - graphics, videos
  - dynamic visualizations

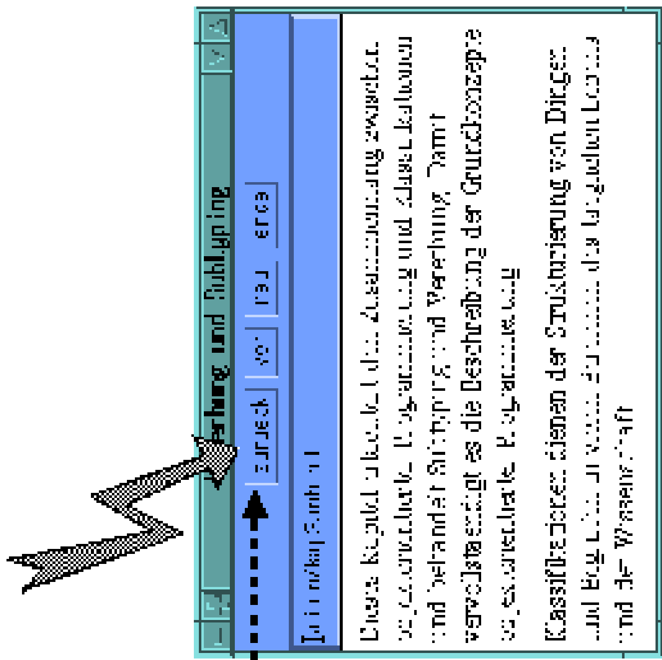
Requirements for software technology:

- powerful and flexible components that
  - provide core functionality
  - cooperate
  - hide complexity
- easy to use, reflect intuition of user

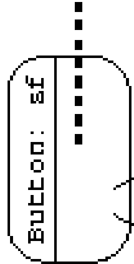
Example: (relation GUI and objects)



1. Benutzer klickt Schaltfläche an



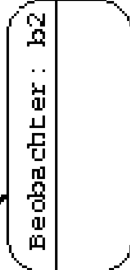
2. actionPerformed-Ereignis tritt an Button-Objekt sf auf



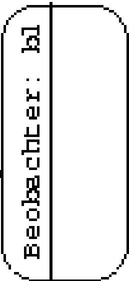
Referenzen auf die bei sf registrierten Beobachter b1,b2

3. sf benachrichtigt die Beobachter b1,b2

actionPerformed



actionPerformed



Distributed computing

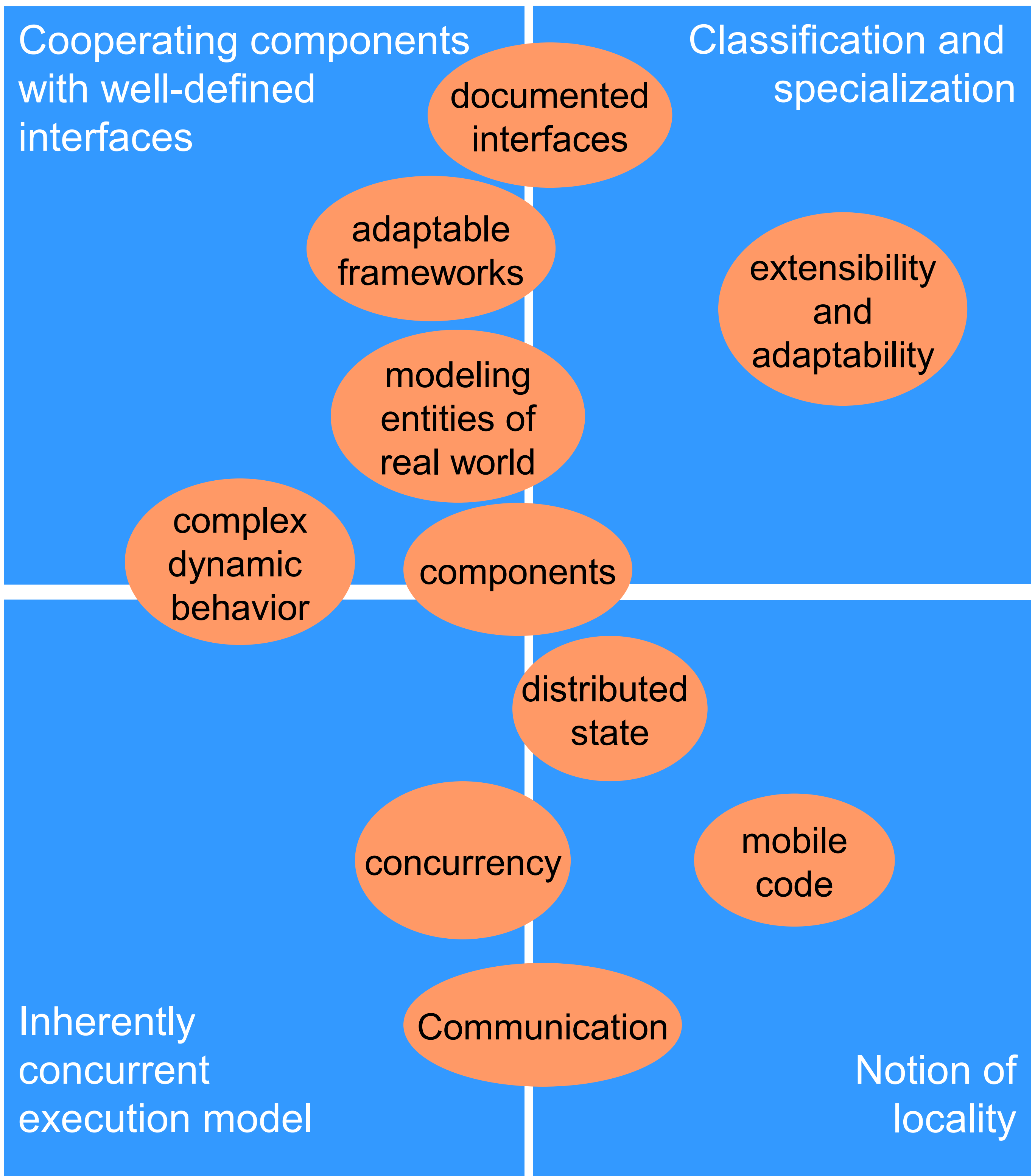
- requires communication mechanism
- means to say who executes the computation and where it should take place (notion of *location*)
- has to support concurrency (different requests from different locations)
- has to work with distributed state
- makes mobile code desirable

Goals:

- Make *local* and *distributed* programs as similar as possible
- Make adaptation from local to distributed programs as simple as possible
- Hide concurrency when reasonable

Required programming concepts

From challenges to required programming concepts:



How can the required concepts be realized?

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 **distributed programs**?

1.3 Programming Object Systems

Explanation: (Object system)

An *object system* is a system that is modelled and described as

- a collection of cooperating objects where
- objects have state and processing ability and
- objects exchange messages.

Collection of objects:

Basic object model:

obj1

a1:
a2:

m(p1,p2) {..}
m1( ) {..}
m2(p) {..}

obj2 .m(“KOOP”,1)

obj2

a:

m(p1,p2) {..}
n(p,r) {..}

Extended object model – Interfaces:

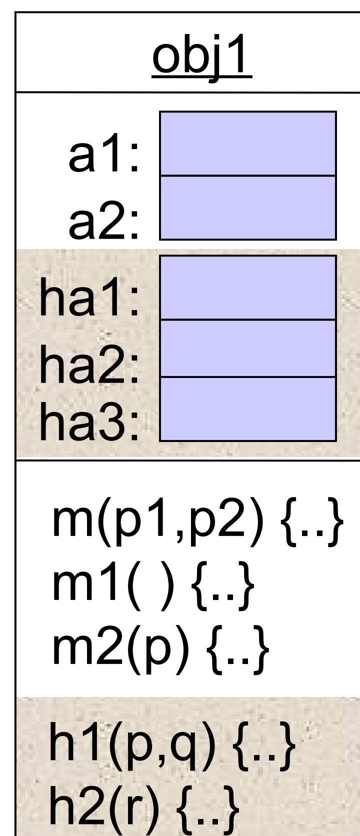
- Objects have well-defined interfaces
  - Publicly accessible attributes
  - Publicly accessible methods
- Interfaces allow to hide implementation details (*information hiding*)
- Interfaces provide a protection boundary (*encapsulation*)
- Interfaces are the basis for abstract description of behavior

Example: (Object interface):

Object obj1 from above may have:

- „private“ attributes

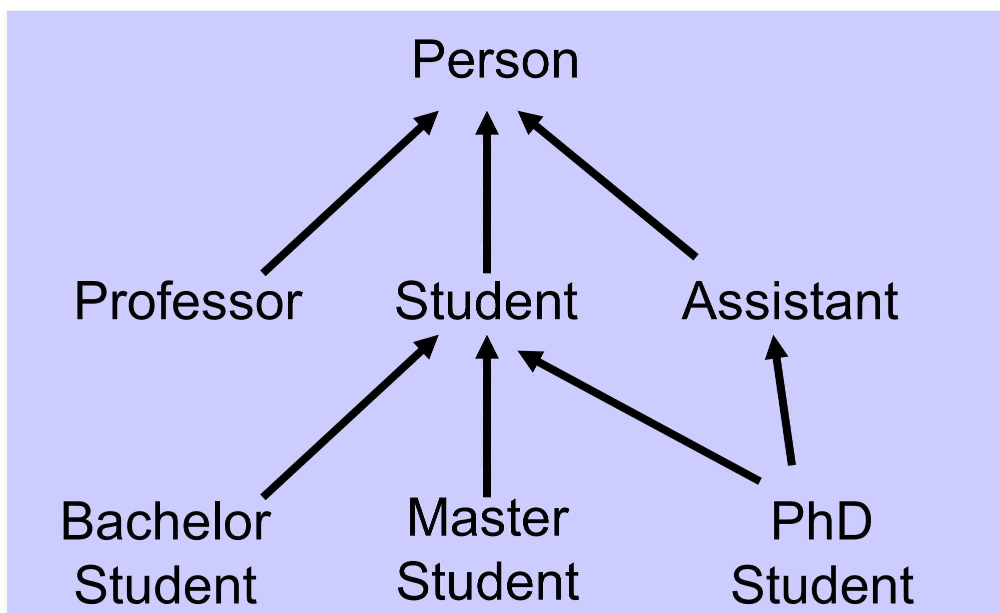
- „private“ methods



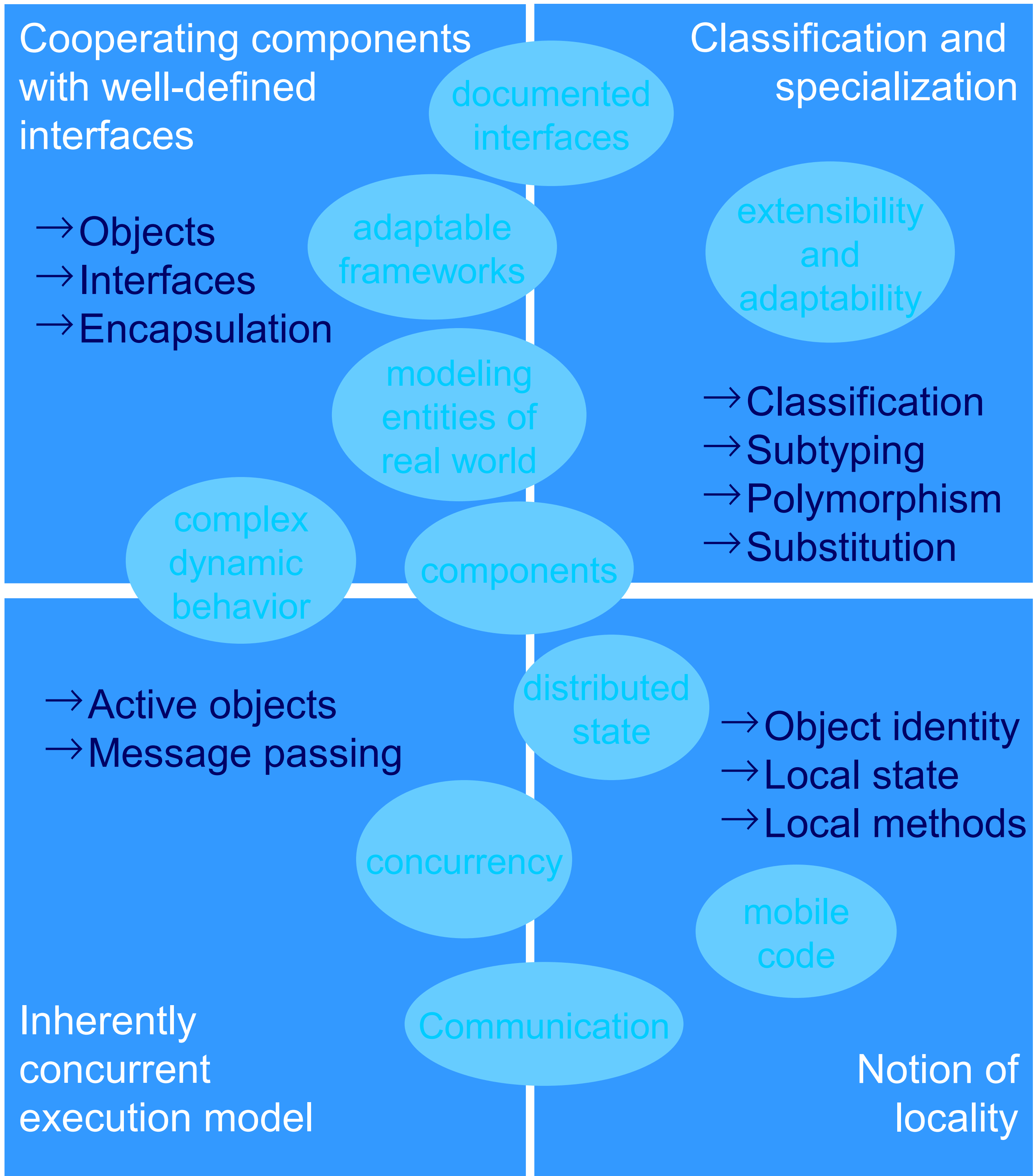
Extended object model – Classification and Polymorphism:

- Objects with same implementation are put into one class / are described by one class
- Objects/classes can be typed according to the interfaces they provide
- Objects can be hierarchically classified according to the types they implement (*subtyping*):
    - objects can belong to several types (*polymorphism*)
    - type hierarchies are extensible
- Substitution principle: Subtype objects can be used wherever supertype objects are expected

Example: (Classification/subtyping):



Relation to required concepts

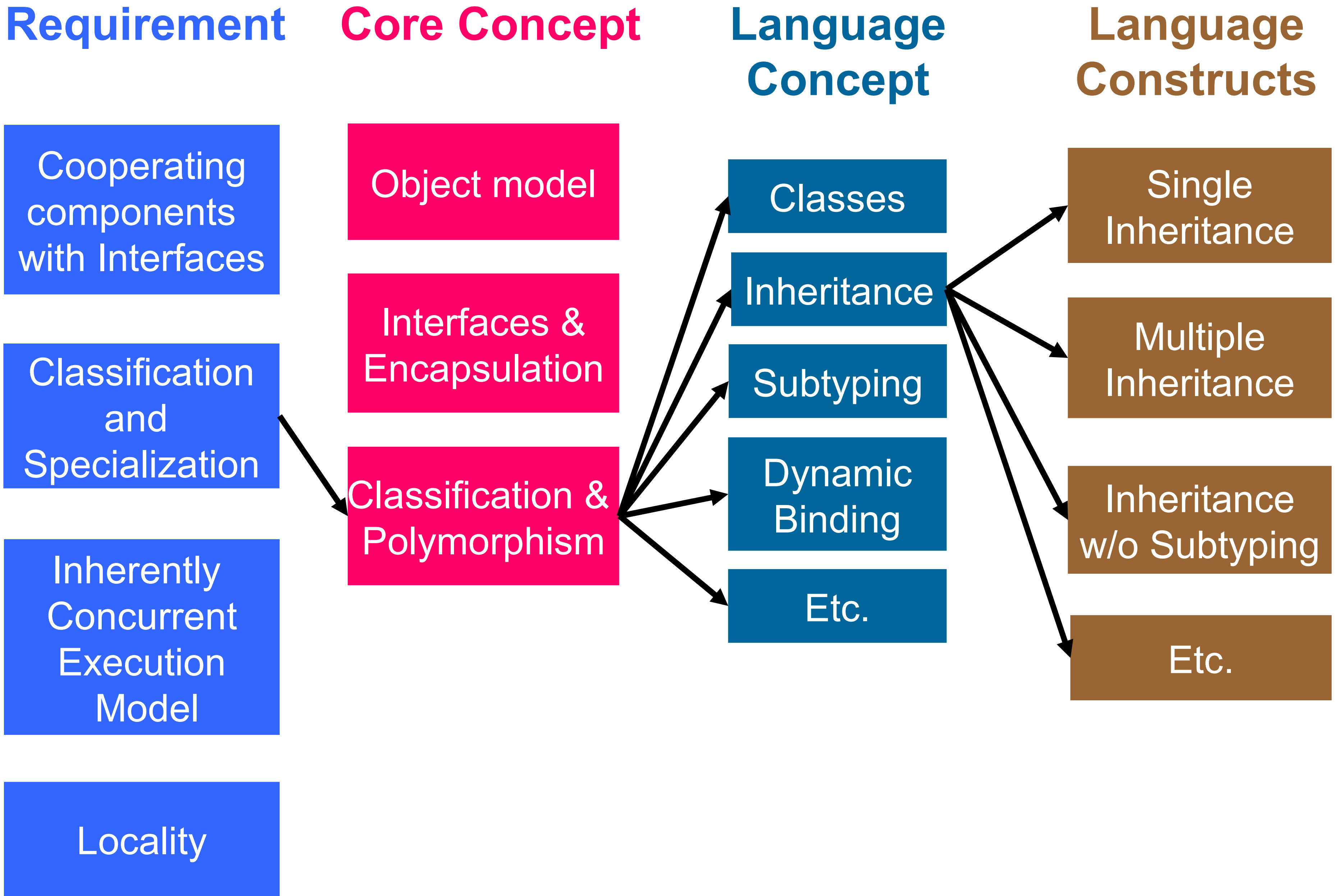


Concepts: Summary

- Core concepts of the OO-paradigm
  - Software based on object model
  - Interfaces and encapsulation
  - Classification and polymorphism
- Core concepts are **abstract concepts**
- To apply the core concepts we need ways to **express them in programs**
- ***Language concepts*** enable and facilitate the application of the core concepts

Remark:

The abstract OO-concepts can as well be applied when using non-OO languages (e.g. imperative languages). However, it is more difficult and less elegant.



Requirement
Core Concept
Language Concept
Language Constructs
Cooperating components with Interfaces
Classification and Specialization
Inherently Concurrent Execution Model
Locality
Object model
Interfaces & Encapsulation
Classification & Polymorphism
Classes
Inheritance
Subtyping
Dynamic Binding
Etc.
Single Inheritance
Multiple Inheritance
Inheritance w/o Subtyping
Etc.

1.4 Properties of Systems & Programs

Explanation: (Specification)

A ***specification*** describes properties

- of a system or program implementing a system
- in a precise way.

Remarks:

- A program satisfies/meets a specification.
- Specifications can be declarative or model-based.
- Programs are developed from specifications.
  Specifications bridge the gap between the intuition about a system and its implementation.
- Specification should be
  - documented
  - checked } verified
  - proved } verified

  We consider program-level specification as an essential part of programming.
- Techniques and tools for handling specifications and programs together are available.

Different kinds of properties:

We distinguish between

- functional properties:
  input-output behavior, modifications, invariants, etc.
- non-functional properties:
  usability, reusability, readability, portability, scalability, efficiency, etc.

We will concentrate on functional properties, in particular:

- Type properties
- Class invariants
- Method specifications
- Interface and encapsulation properties

Goals:

- get a better understanding of programming
- learn specification and checking techniques
- learn new language features and constructs

Examples: (Properties)

1. Type properties:

```
class Entry<ET> {
   ET element;
   Entry<ET> next;
   Entry<ET> previous;

   Entry(ET element, Entry<ET> next,
                Entry<ET> previous) {
     this.element = element;
     this.next = next;
     this.previous = previous;
   }
}

class LinkedList<ET> {
   Entry<ET> header =
       new Entry<ET>(null, null, null);
   int size = 0;

   LinkedList() { ... }
   ET getLast()  { ... }
   ET removeLast() { ... }
   void addLast(ET e) { ... }
   int size() { return size; }
}
```

```
class Test {
 public static void main(String[] args) {

    LinkedList<String> ls =
               new LinkedList<String>();
    ls.addLast("erstes Element");
    ls.addLast("letztes Element");
    ls.getLast().indexOf("Elem");
       // yields 8

    LinkedList<Object> lo =
               new LinkedList<Object>();
    lo.addLast( new Object() );
    lo.addLast( new Object() );
    lo.getLast().indexOf("Elem");
       // program error
       // detected by compiler
}}
```

2. Access properties:

```
class Capsule {
   private Vector v;
   ...
}
```

Only objects of class `Capsule` can access vectors referenced by objects of class `Capsule`. True??

3. Assertions (Zusicherungen):

a. Simple property:

```
...
C cobj = new C(...);
assert cobj.x != null ;
```

b. Loop invariant (Schleifeninvariante):

```
public static int isqrt( int y ){
  int count = 0, sum = 1;
  while (sum <= y) {
    count++;
    sum += 2 * count + 1;
    assert  count*count <= y
         && sum==(count+1)*(count+1);
  }
  return count;
}
```

c. More complex property for an AWT-fragment:

```
...
Container c;
Button b;
...
c.remove(b);
assert !EX Container cex: !EX int i:
    cex.getComponents()[i] == b;
```

Remark:

- Assertions allow to formulate properties about the state of program variables.
- If assertions are specified by Java expressions, these expressions should not have side-effects.
- Asssertions can be used to express preconditions of methods.
- Assertions can be proved or dynamically checked.

4. Method specifications:

```
public class IntMathOps {

  /*@ public normal_behavior
    @  requires   y >= 0
    @  modifiable \nothing
    @  ensures    \result*\result <= y
    @      &&  y < (Math.abs(\result)+1)
    @            *(Math.abs(\result)+1);
    @*/
 public static int isqrt(int y){ ... }
}
```

5. Class invariant:

```
public class List {
  int length;
  ListElems le;
  //@ invariant length == le.leng();
  ...
}
```

Further kinds of functional properties:

- Event specifications:
  - occurrence of exceptions
  - modifications of variables
  - invocation of methods
- Termination
- History constraints:
  - relations between two states
- Temporal properties:
  - Is something true until an event happens?
  - Will something eventually happen?