• A **data structure** is a special type to administer and interlink data, in order to be able to access and manipulate this data properly. Data structures are always interconnected with operations.

• The **definition** of data structures is done by a **concrete specification** of the data management and of the required operations. This concrete specification determines the general behavior of the operations and prescinds it from the concrete implementation of the data structure [Wik104].

• Data structures permit to model and implement the relations between data in a way understandable by humans

• Data structures are of fundamental relevance for the practical usage of programming languages by humans. [Webe98].
An example is the simplified representation of a product structure in PDM-system. In this example, a product is made of several parts, that are described by the attributes geometry, weight, costs and version number.
Example of a data structure (2)

L1 WINDOWFAY_NEW AS
  L2 AE_ACTUATORUNIT
    L3 AE_BOTTOM_PART
    L3 AE_MECHANIC_A
    L3 AE_ACTUATOR
    L3 AE_BOX_P
  L2 PE_ROTOR_ASSEMB
    L3 PE_ROTOR_BOT
• In several domains, a solution must be found to a technical problem
• For creating this solution, an amount of **activities** is required, meaning that a process is running
• To run this process, a **processor** (human, automat, machine, computer) is required.
• A **connection** must exist to this processor. **Communication devices** (speech, button press, move lever, machine language) are used for this purpose.
• A problem is solved, if the solution path is given in a format understandable by the processor.
Definition 1.9: An algorithm is a **precise** (meaning written in a determined language), **finite** description of a general procedure with the usage of executable **elementary** steps.

Characteristic properties of algorithms

- precise description → unambiguous communication
- elementary step → sequential parallel execution
- Described with a finite number of steps → Algorithm is finite
- The result is always unambiguous → Algorithm is determinate
**Views on algorithms**

**Functional view:** what: the problem to be solved by the algorithm

**Operational view:** how: the modality, how the task is solved

- Elementary step
- Description of the processing
- Data and parameters, that are processed by the algorithm

For an algorithmic solvable problem (functional view) there exist different solution paths (operational view)

**Therefore:** differentiate the specification of a task from its algorithmic solution
Elements that describe an algorithm

Elements:

• Use a (primitive) function, or

• **Sequence:** a consecution of steps

Usage:

1. **Loop:** repeated execution of a function or a sequence

2. **Condition:** conditionated execution of a function function, sequence or loop
Computer Science for Engineers: Topics

Lecture Contents

1. Introduction

2. Basics
   - 2.1. Information representation and processing
     - 2.1.1. Alphabet - Data - Signals - Information
     - 2.1.2. Coding - number systems
     - 2.1.3. Boolean algebra and propositional logic
     - 2.1.4. Computer architecture
     - 2.1.5. Introduction to data structures
     - 2.1.6. Introduction to algorithms
   - 2.2. Programming paradigms
     - 2.2.1. Term programming
     - 2.2.2. Procedural programming
     - 2.2.3. Object-oriented programming

3. Object Orientation
What is programming?

- **Programming** is an activity in which a specific problem is tried to be solved through **systematic utilization of a given programming language**
- A program is an **algorithm verbalized into a programming language**
- 2 types of **programming languages**:
  - **Machine language**, esp. assembly language: optimized towards a specific processor
  - **Higher level programming languages**: allow a problem-oriented formulation of a designed algorithm
- Programmers use **higher level programming languages** when developing applications. In the programs (interpreted or compiled), the **source code is automatically translated from computer instruction to machine instructions**, so that ultimately the source code can be worked upon by the **hardware** (the processor)
Basics about programming

• Using a certain computer serves predominately to complete a **purpose** (i.e. construction, navigation of airplanes, text processing, money transfers, or simply for computer games)

• One of the deciding prerequisites of use is the computers’ **error-free performance** and it’s **intuitive** operation mode

  - The computer must be designed in such manner that the user doesn’t have to program the computer. The user only must use the computer and not program it

  - The programmer writes the order of instructions with the help of high level programming languages and software development tools. These instructions build the application program

  - The system programmer sees the computer from a different point of view. He prepares the software tools, compiler, interpreter, and help menus that serve the programmer

  - The computer hardware changes the order of machine commands into micro-order for the control of single hardware components
Different computer perspectives

### Different computer perspectives

**Perspective**
- User
- Programmer
- System programmer
- Hardware engineer

**Program**
- User program
- Compiler, editors, and software tools
- Machine commands
- Micro program
- Hardware

**Example**

```
if Liststart ≠ nil
  then begin (the list is not empty)
    indicator := Liststart;
    while indicator^.next ≠ nil
      . . .

CAx
CATIA
AutoCAD
ProEngineer

PLM
SmarTeam
SAP
TeamCenter

loop:
  CMP W I0, ! R0
  JEQ list_ende
  MOVE W ! R0, R0
```

---

Prof. Dr. Dr.-Ing. Jivka Ovtcharova – CSE-Lecture – Ch. 3 - WS 08/09 - Slide 12
Historical development of important programming languages

1954
1958
1959
1964
1965
1967
1968
1969
1970
1971
1972
1983
1985
1986
1987
1988
1994
2002

FORTRAN
ALGOL
BASIC
LOGO
SMALLTALK
SIMULA
BCPL
C
PASCAL
C++
Java
C#

Fara04

2. Basics
2.1 Programming paradigms / 2.2.1. Term programming
**Programming Paradigms**

- **Paradigma** = pattern of thought, superior principle
- **Programming paradigm** = basic appendages, how you can manage a computer through programming

<table>
<thead>
<tr>
<th>Paradigma</th>
<th>In the foreground are...</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>imperative</td>
<td>Variables, that correspond to one or more storage cells of a computer, and instructions as abstractions of the processor functions</td>
<td>Assembly languages, FORTRAN, ALGOL</td>
</tr>
<tr>
<td>procedural</td>
<td>Procedures (under programs), to which arguments (parameter) are assigned. The procedures calculate the worth according to an algorithm and return this value</td>
<td>Pascal, C</td>
</tr>
<tr>
<td>functional</td>
<td>(mathematical) functions, that assign a value to a vector of parameters (also functions)</td>
<td>LISP, LOGO</td>
</tr>
<tr>
<td>Logic based</td>
<td>Logical expressions, that in general contain free variables, and that will be verified through an appropriate binding of these variables</td>
<td>PROLOG</td>
</tr>
<tr>
<td>Object oriented</td>
<td>autonome, interaged objects, that communicate through messages and are grouped with other similar objects in classes</td>
<td>Smalltalk, C++, Java, C#</td>
</tr>
</tbody>
</table>
Definition of procedural programming

Definition 2.0: A procedure is a sequence of instructions, that represent an autonomous part of the program, that can be called from another autonomous part of the program.

- A procedure has an interface, that specifies its name, its allowable input objects (also called arguments) and its result (the resulting data after running the procedure).
- The procedural programing paradigm extends the imperative programming by abstraction mechanisms for the creation of procedures (subprograms), as well as means of expression for the iteration and selection.
Procedural Programming: Code example

```pascal
program P(input, output);
var v : integer;
procedure increment(var a : integer);
begin
  a := a + 1;
end;
procedure decrement(var a : integer);
begin
  a := a - 1;
end;
begin
  v := 0;
  increment(v);
  writeln('Increment result: ');v;
  writeln(v);
  decrement(v);
  writeln('Decrement result: ');v;
  writeln(v);
end.
```

In the program example, a global variable v of type „Integer“ and 2 global procedures (subprograms, subroutines) are declared.

The 2 procedures in this example perform a well-known algebraic function: incrementation, resp. decrementation of a number. Both procedures receive a parameter „a“ of type integer. This parameter is incremented, resp. decremented with 1.

In the main program (between the keywords “begin” and “end.”):
1. the variable “v” is initialized with the value 0
2. the procedure increment (v) is called with „v“ as parameter. v is incremented and the result is stored in v. After calling increment(v) we have v=1
3. The write and writeln prints the result on the screen
4. The procedure decrement is called analogously the procedure increment
5. The write and writeln prints the result on the screen
6. end. – the program ends
• Widely spread for **system and application programming**
  - **UNIX** and **WINDOWS** operating systems are mostly written in C

*What is an operating system?* **It is a software that facilitates the functioning of a computer. It administrates the main memory, the input-output devices and manages the execution of programs.**
• **Widely distributed** by system and user programming
  - The programming language C is an essential part of the UNIX operating system: the C-compiler is installed together with the operating system. Many programs in the UNIX world are delivered as source code and must be locally compiled

**What is a compiler?** A compiler *(also called translator)* is a computer program, that transforms the source code of a program, written in a specific programming language, into a semantically equivalent program, written in the target language. Usually, the source code that was written by a programmer in an established programming language is translated into assembler code, byte code or code written in machine language. This translation of the source program *(the source code)* into the target program is also called **compilation**.
Advantages of Procedural Programming (3)

- **Transparency** of the language
- Support of the *structured programming* through determined selection of program and control mechanisms
- Possibility to exactly describe the data that is given through a multitude of data types
- Simple design of the language, as well as efficient implementation
Disadvantages of Procedural Programming

- Changes in the data structures prove to be very elaborate and error-prone
- High costs for the maintenance and service of large complex programs
- Careless supply of procedures or false interpretation can lead to the system run time errors
- Searching for errors is difficult
- Limited reusability
- Hard to read due to chained complex procedures
Object Oriented Languages – Story of development (1)

- First object oriented language in year 1967:
  - SIMULA-67: first object oriented language (OO)
  - Application area: simulation
  - No practical meaning
- 70’s:
  - SMALLTALK: till the mid 80’s the only OO language with practical application

It is no coincidence that the first OO programming language comes from the domain of modeling and simulation. The OO paradigm is very well suited for modeling real, physical states.
Since 1988 fast dispersal of OO programming languages

- Eiffel: a completely OO language
- Different OO extensions:
  - C -> C++
  - Pascal -> Object Pascal (Delphi)

Present state:
- Due to its compatibility to C, C++ prevailed on a large base
- Smalltalk is increasingly loosing its importance
- Java is established as a language for the development of web applications, but is not restricted only to this domain
The object-oriented paradigm is based on the concept that the world is built of a heap of objects that communicate with each other by messages.

An object is described by data, that characterises its attributes, and by methods, that determine the access on data.

The methods spectrum of an object determines how the objects reacts on messages it receives. Access on object data is possible through the object methods.
Advantages of object-oriented programming languages

- Reusability of existing program parts and programs by inheritance and class libraries.

- Avoidance of duplicates in code by reusability.

- Code maintenance is done by adjusting only small parts of the program.

- Abstract classes are the basis for consistent interfaces for other classes.

- Programming errors can be avoided due to the high, problem-oriented abstraction level.
Disadvantages of Object Oriented Programming Languages

- **Long orientation time** in object oriented thinking
- Knowledge about algorithm formulation is not enough.
- The **inner structure of classes and libraries must also be known** in order to be able to extend the programs in a correct way.
Recommended literature (1)


Recommended literature (2)


[Stud05] Rudi Studer: „Grundlagen der Informatik 1, SS 2005“, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Uni-Karlsruhe, 2005


Computer Science for Engineers

Lecture 3

Object Orientation (part 1)

Prof. Dr. Dr.-Ing. Jivka Ovtcharova
Dipl. Wi.-Ing. Dan Gutu
14th of November 2008
Outline

Lecture Content

1. Preface
2. Basics
3. Object orientation

3.1. Introduction and basic concepts
3.1.1. Introduction
3.1.2. Objects and classes
3.1.3. Attributes of object orientation

3.2. Object-oriented modelling with UML
3.2.1. Intro
3.2.2. Case Application models
3.2.3. Static models
3.2.4. Dynamic models
3.2.5. summary

3.3. Methods of the OO Analysis and Design
Definition 2.1: Object orientation (OO) serves as the description of problem classes, where the single components are modeled through objects. The description includes not only the characteristics (attribute), but also the corresponding operations (methods).

- Object orientation is not only used in programming, but also in:
  - Modeling and simulation of complex technical systems
  - Analysis and Software-Design
  - Data base technologies
  - Design of user interfaces
Goal of Object Orientation

- „Natural“ Modeling and Design
  - Abstraction concepts of OO are based on reality (through classes and objects)
  - Metaphors for relationships and behavior: association, inheritance, message exchange, polymorphism
  - In contrast to procedural programming, OO methods are integrated in the different phases of the development process: analysis, design, and realization

- Re-use of the source code
- Re-use of OO designs: design pattern
• Mapping of parts of the real world into a program:
  - The real problem is illustrated through a program
  - The illustrated elements of the program are related to each other and can interact with each other
• An **object** is a theoretic or real entity of the environment or in the software. Objects are characterized by [Schn98]:
  - *Identity*: specification, resp. instance of a class
  - *Data aspect*: Attributes, resp. properties
  - *Activity aspect*: behavior, given by the methods of the object

• A class is a **entity**, by which the **properties** and the **behavior** of the objects is described. Classes can be in a **class hierarchy**. [Schn98]

• The **attribute** is a property of an entity provided with a name. Usually, its meaning is given by its name [Schn98].
• A **method** is a program fragment, where a received message is processed and the inner state of the addressee is modified, accordingly. Methods can be compared with the procedures of the procedural programming languages and are assigned to an object. [Schn98]

• A **message** is an entity, from which all **interactions between objects** are derived. It is composed of the **specification of the addressee** (an object), a **method** that can be processed by the addressee, as well as the required **parameters** [Schn98].

• **Inheritance**: **propagation of properties** (attributes, methods) to another class or to another object [Schn98].

• **A class library** is a collection of reusable software components that use the constructs of the object oriented programming paradigm. [Schn98]
Objects

3.1.2 Objects and classes

- An **object is an existing „thing“** from the problem area of the software (the user world)

- An object has a **defined behavior**
  - Behavior is comprised from a number of exactly defined operations to fulfill a task
  - One such operation is performed as soon as a message is received

- An object has an **inner condition**
  - The state of an object is private, that means, it is known only inside the object
  - The consequence of the operation of an object (by receiving a message) is dependent on the current state of the object

- An object has an **explicit identity**
  - The identity of an object is independent of its other characteristics
  - Multiple different objects with identical behavior and identical inner conditions can exist together in the same system
State – behavior – identity: example

Source: [Booc91]
Example: Object „Upper arm of a robot“ (1)

- Labeling of the object »Upper Arm of a Robot« by three attribute values (character)
- Manipulation of the angle using four methods (operations)

**Attributes:**
- Current angle
- Maximal angle
- Minimal angle

**Operations:**
- Adjust grand position, parameter: angle
- Report current position, return value: angle
- Adjust new position, parameter: angle
- Initialize min and max angle, parameter: min and max angle
Example: Object „Upper arm of a robot“(2)

- Behavior of the object „Upper Arm of a Robot“:
  - Reaction to messages such as ‘report current position’, ‘initialize min angle’

- The condition of the upper arm is given through its characteristics (current, max, and minimum angle)

- The object „Upper Arm of a Robot“ exists independently from its characteristics
A class is the building plan of an object

- Classes correspond to data types
- In the class, the properties of the objects are defined, as well as their behavior. The object is a concrete specification of the class
  - Class: „the car has a color“, Object: “the color of the car is silver“
- There are several objects of a class, each object can be identified unambiguously.
• A **class** is a **description of one or more similar objects**. „Similar“ means that a class describes only one object of a certain type
• A class describes the **construction**, the **processing possibilities** and the possible **behavior** of objects of this class
• A class definition is comprised of:
  - Definition of the attributes of the class (local criteria)
  - Definition of the relationship to other classes
  - Definition of the operations that are possible for the objects of the class or on the class itself

Source: [Schn88]
Classes have the following relationships to one another:

- **Association**: The objects in one class are associated with objects of another class. Often also the other way around - that means, that associations are frequently bi-directional
  - A *car* has an *engine*, an *engine* is part of the *car*

- **Usage**: The objects of a class use attributes or operations of another class to provide for their own attributes and operations
  - A *human being* is driving the *car*

- **Inheritance**: A class is a sub class of another class. It «inherits» all attributes and operations of this class, that means, all objects of the class possess them without having to be defined in the local class
  - A *car* inherits all the properties of the class *vehicle*.
• Attributes can be objects of other classes.
• In a class, this circumstance is described by a relation to another class
• An object is at this point a reference to another object
According to UML notation the class is represented in a rectangle and is comprised of:

- **Class name**: Naming of a class from the language area of application field. As the rule, a main word or an adjective (e.g., "vehicle" or "civil vehicle")

- **Attribute**: A data value (e.g., position, size, color,...). The objects of a class have the data value of this class

- **Operation**: A function (e.g., delay, delete, change size,..). Application to objects of a class
• An **object attribute of a class** is the description of a **data element** that is available in each object of the class. The name of the object attribute is given in the **description of the class**.

• An object carries an individual and unchangeable attribute value for each object attribute.

**Notation:**

```
Class
Attribute_1
...
Attribute_n
```

```
Object : Class

. . . ...
```

**Example:**

```
Team meeting

<table>
<thead>
<tr>
<th>title: String</th>
</tr>
</thead>
<tbody>
<tr>
<td>beginning: Date</td>
</tr>
<tr>
<td>duration: int</td>
</tr>
</tbody>
</table>
```
Class `Team_meeting` with attributes:
- title: String
- beginning: Date
- duration: int

`ar12: team_meeting`
- title = "12. Department discussion"
- beginning = 5-10-98 10:00
- duration = 120 min

`ar13: team_meeting`
- title = "13. Department discussion"
- beginning = 5-17-98 10:00
- duration = 120 min

`ar14: team_meeting`
- title = "exhibition preparation"
- beginning = 5-12-98 9:00
- duration = 180 min

Object diagram with examples for object attributes:
- `O` is an instance of `C`. 

3.1. Introduction and Basic Concepts
3.1.2 Objects and classes

Object attribute

Object
The classes define the static structure of the system. The dynamic structure (the structure during runtime of the program) reflects the static structure. The relations between classes mould the possible relations between their objects. An object can have a relation to another object only if the classes of these two object have defined these relations.
Outline

Lecture Content

1. Preface
2. Basics
3. Object orientation

3.1. Introduction and basic concepts
   3.1.1. Introduction
   3.1.2. Objects and classes
   3.1.3. Attributes of object orientation

3.2. Object-oriented modelling with UML
   3.2.1. Intro
   3.2.2. Case Application models
   3.2.3. Static models
   3.2.4. Dynamic models
   3.2.5. summary

3.3. Methods of the OO Analysis and Design
Properties of object orientation (1)

- **Enclosure:**
  
  Data enclosure allows the *separation of the internal implementation of objects* to external access. Access is granted only via an predefined interface for making it independent of implementation details.

- **Instantiation:**
  
  Instantiation means the *creation of an object of a specified class*, that represents its (abstract) description. This instance of the class is a concrete occurrence of that class.

- **Inheritance:**
  
  Classes can be derived from other classes. Each class inherits the attributes and methods from the upper class.

- **Polymorphism:**
  
  The concept of polymorphism means, that properties or methods of a class can be referenced by objects, *without knowing the concrete specification of the class inside those respective objects.*
Properties of object orientation (2)

- Enclosure
- Instantiation of classes
- Inheritance
- Polymorphism

This object oriented model of a spring-mass-system will explain the features of object orientation.

Instead of ideal (undamped) springs, the spring-mass-system can have a damped spring.
Properties of object orientation

- **Enclosure:**
  
  Data enclosure allows the *separation of the internal implementation of objects* to external access. Access is granted only via a predefined interface for making it independent of implementation details.

- **Instantiation:**
  
  Instantiation means the *creation of an object of a specified class*, that represents its (abstract) description. This instance of the class is a concrete occurrence of that class.

- **Inheritance:**
  
  Classes can be derived from other classes. Each class inherits the attributes and methods from the upper class.

- **Polymorphism:**
  
  The concept of polymorphism means, that *properties or methods of a class* can be referenced by objects, *without knowing the concrete specification of the class inside those respective objects.*
In object oriented programming language, a class is defined through the **summary of a heap of data and methods** that operate on it.

The data is represented through a **set of variables**, that have been newly created for each instantiated class (noted as *attribute*, member variables, instance variables or *instance criteria*).

The operations (methods) are available in executable programming code **only once**. However, they can operate at each call on the data of a certain object.

---

A **message** is an entity, on which all interactions between objects are based. A message is build of the indication of the respective **object** and an **operation**, that can be performed by that object, as well as the required **data**. [Schn98]
• The **attributes** represent the **state of an object**. They can **differ in each instance of a class** and **change during their lifetime**

• Objects can hide their attributes, making invisible to the outside world these attributes, as well as their values

• Objects encapsulate their data and allow data access only via predefined interfaces

• The **methods** represent the **behavior of the object**

• The **behavior of an object** of a class is determined by the **definition of its methods in the class** and is dependent on the corresponding program code and the current state of the object

• This **summarization of methods and variables in a class is known as enclosure**
- **The object „spring“ encapsulates its attributes:** spring flange, spring constant, spring length and spring equation. *The object contains and administrates independently the knowledge on its physical behavior.*

- The modeling- and simulation (M&S) system has no direct access to the attributes of the object (example: the spring equation)

- The system can determine the physical behavior of the object only by sending it an message (example: “compute equation”)

- The message is send by calling the respective method, example `Spring.ComputeEquation()`.

```plaintext
model Spring
    Flange a "left flange";
    Flange b "right flange";
    parameter Real c "spring constant";
    parameter Length l "spring length";

equation
    b.f - a.f = c * ( (b.s - a.s) - l );
end Spring;
```
• The **package-concept:**
  - All classes in Java are organized in packages. Packages can be **nested up to any depth** (ex. `java.awt.image`)
  - The package-structure is shown in the filesystem through the **folder structure** (ex. `/java/awt/image`)
  - A package has his own **namespace** and the classes in a package can be accessible either only **locally** (from the package – **private** class) or from **outside** the package (**public** class)
  - Classes from another package must be **imported explicitly** (ex., `import java.awt.Button`). All **classes** from the package `java.lang` are imported by default.
Properties of object orientation

• **Enclosure:**
  
  Data enclosure allows the separation of the internal implementation of objects to external access. Access is granted only via a predefined interface for making it independent of implementation details.

• **Instantiation:**
  
  Instantiation means the creation of an object of a specified class, that represents its (abstract) description. This instance of the class is a concrete occurrence of that class.

• **Inheritance:**
  
  Classes can be derived from other classes. Each class inherits the attributes and method from the upper class.

• **Polymorphism:**
  
  The concept of polymorphism means that properties or methods of a class can be referenced by objects, without knowing the concrete specification of the class inside those respective objects.
3.1.3 Attributes of object orientation

- Creation of an object (an instance) of a class
- This instance is a concrete specimen of that class
• The model of the spring-mass-system is created by instantiating (creating objects) of the given classes Flange, Spring and Mass,
• These objects are interconnected to a complete system
• Each object can be initialized with the corresponding parameters, ex. the spring with its constant \( c=k \)

```plaintext
model SpringMassSystem
    parameter Mass m;
    parameter Real k;
    Flange F;
    Spring S (c = k);
    SlidingMass M (mass = m);
equation
    connect (F.a, S.a);
    connect (S.b, M.a);
end SpringMassSystem;
```
Properties of object orientation

• **Enclosure:**

  Data enclosure allows the separation of the internal implementation of objects to external access. Access is granted only via a predefined interface for making it independent of implementation details.

• **Instantiation:**

  Instantiation means the creation of an object of a specified class, that represents its (abstract) description. This instance of the class is a concrete occurrence of that class.

• **Inheritance:**

  Classes can be derived from other classes. Each class inherits the attributes and methods from the upper class.

• **Polymorphism:**

  The concept of polymorphism means, that properties or methods of a class can be referenced by objects, without knowing the concrete specification of the class inside those respective objects.
• Through **inheritance**, a class **will not be completely newly defined**, but will be **derived from other classes**. In this case, the class **inherits all characteristics of this class and adds other characteristics if desired**

• Inheritances **can have multiple levels** - that means, a derived class can be the basis class for other classes. In this way, multi level hierarchies are created. The hierarchies represent the taxonomy and the structure of the modeled problem area.
• **Problem:** in the M&S system we do not want the restriction of objects from masses (ideal springs). Our model shall be extendable by *new elements*, like damped springs or damper. We want to *reuse as much as possible of the existing code* in order to spare time and costs and reduce the probability of errors.

• **Solution:** *through inheritance, we isolate the attributes and the behavior of that is common to several objects into an upper class.* Common features of springs and dumpers are:

1. Both have 2 connecting points (flanges).
2. Both are elastic elements: their length is modified by applying a force.
   - Delimitation: there are also other elements with 2 connecting points, that are non-elastic, like the beam. Therefore, common feature 1 and common feature 2 should be placed on different hierarchical levels of the inheritance tree.
partial model TwoFlange
   Flange a; Flange b;
end TwoFlange;

partial model Compliant extends TwoFlange;
   Force f; Distance s;
equation
   s = b.s - a.s; 0 = a.f + b.f; f = b.f;
end Compliant;

model Spring extends Compliant;
   parameter Real c; parameter Length l;
equation
   f = c * (s - l);
end Spring;
Inheritance (4)

- **TwoFlange** is the class that contains 2 flanges \textit{a} and \textit{b}. The class Flange can be considered as given.

- The class **Compliant** inherits from the class TwoFlange. This means, that a compliant part (Compliant) is an element with 2 connecting points (TwoFlange).

- The class **Compliant** defines the behavior of a compliant part by an equation, that defines the dependence between the force \( f \) and the distance \( s \) between the two flanges \textit{a} and \textit{b}.

- The class **Spring** inherits from the class Compliant. This means, that a Spring is a Compliant (compliant part). In addition, Spring defines the parameter specific to a spring: spring constant \( c \) and spring length \( l \). The equation describes the behavior of the spring.

```plaintext
partial model TwoFlange
    Flange a; Flange b;
end TwoFlange;

partial model Compliant extends TwoFlange;
    Force f; Distance s;
    equation
        s = b.s - a.s; 0 = a.f + b.f; f = b.f;
end Compliant;

model Spring extends Compliant;
    parameter Real c; parameter Length l;
    equation
        f = c * (s - l);
end Spring;
```
Inheritance relation

3.1.3 Attributes of object orientation

Source: [Booc91]
Properties of object orientation

- **Enclosure:**
  Data enclosure allows the separation of the internal implementation of objects to external access. Access is granted only via a predefined interface for making it independent of implementation details.

- **Instantiation:**
  Instantiation means the creation of an object of a specified class, that represents its (abstract) description. This instance of the class is a concrete occurrence of that class.

- **Inheritance:**
  Classes can be derived from other classes. Each class inherits the attributes and methods from the upper class.

- **Polymorphism:**
  The concept of polymorphism means, that properties or methods of a class can be referenced by objects, without knowing the concrete specification of the class inside those respective objects.
Polymorphism (1)

- **Polymorphism** directly translated means „many sidedness“ and designates first the **ability of variables of type object of a class to gather together objects of different classes.** This doesn’t happen uncontrolled, but rather is limited to object variables of type X or all objects in the class X or a further derived class.

- There are many forms of **polymorphism**:
  - Generic polymorphism
  - Parametric polymorphism
  - Inheritance polymorphism
Polymorphism (2)

- As for this lecture, we will limit ourselves to **inheritance polymorphism**
- A message can initiate different behavior (different operations), depending on which subclass of the upper class the object belongs to.
Polymorphism (3)

• The upper class (vehicle) inherits its properties and methods:
  - The subclasses (bus, ship, plane) have contain all the method start(), but they
    interpret it in different ways
• One demands from a vehicle to start, but each concrete vehicle has a
different way of doing it.

Class: Vehicle
Method: start()
• **Problem:** in the M&S system we can model and simulate only an ideal spring, because we only have defined the class `Spring`. For precise studies, we need a model that allows us to represent without an extra effort the behavior of different types of springs, both ideal and real (damped) springs

• **Solution:** the inheritance hierarchy provides the possibility to consider *common attributes and behavior* of compliant elements. Both the damped spring and the ideal spring are compliant part, such that a class `DampedSpring` can be derived from the class `Compliant`. In the spring-mass-model, a `Compliant`-object is used instead of a `Spring`-object. This object of an upper class can contain both `Spring` and `DampedSpring` objects, that have different behavior according to polymorphism
• Additionally to the class `Spring`, that describes the behavior of an ideal spring, we define the class `DampedSpring`.

• `DampedSpring` is derived from `Compliant`, too. This means, that a damped spring can receive and interpret the message that can be processed by an compliant element, for example „Compute length modification“

```model DampedString extends Compliant;
    parameter Real c "spring constant";
    parameter Real d "damping";
    parameter Length l "length of unstretched spring";
    Velocity v "relative velocity between flanges b and a";

equation
    v = der(s);
    f = c * (s - l) + d * v;
end DampedString;
```
Polymorphism (6)

An ideal or damped spring can be introduced to the system as a compliant part object. The M&S system decides during runtime, if the method for the ideal spring or for the damped spring should be called.
Beispiel: Industrie-Roboter (DLR, Dynasim, KUKA)

1000 nichttriviale algebraische Gleichungen, 80 Zustände.
mit “Mixed-Mode Integration”:
schneller als Echtzeit auf 650 Mhz PC.
3.1.3 Attributes of object orientation

Beispiel: Hardware-in-the-Loop Simulation von Automatikgetrieben
(verschiedene KFZ Hersteller)

Steuergert (Hardware)

gewunschter Druck

Steering element

Fahrer + Motor + Wandler +
1D Fahrzeugdynamik

Courtesy ZF Friedrichshafen

Information Management in Engineering

Prof. Dr. Dr.-Ing. Jivka Ovtcharova – CSE-Lecture – Ch. 3 - WS 08/09 - Slide 73
Application example (3): large, detailed, vehicle model

3D-Mechanik
60 joints, 70 bodies

- 25000 non-trivial algebraic equations
- 320 states

Motor (Combustion)

Drive chain (automatic drive)

Hydraulik
Literature (1)


[HiKa99] M. Hitz, G. Kappel: „UML@Work“, dpunkt.verlag, 1999


What is UML?

- UML stands for "Unified Modeling Language".
- The standardization of UML occurs through the OMG (Object Management Group) [www.omg.org](http://www.omg.org).
- UML integrates almost all techniques up till now with:
  - Data modeling (Entity Relationship Diagrams)
  - Descriptions of workflows and object behavior
  - Structure implementation
- The visualization with UML supports the entire development process.
- UML is not a method, but rather a notation (description technique).
- UML is independent from the developing environment (languages, operating system, etc.)

---

**3.2. object oriented modelling with UML**

**3.2.1. Introduction**
What is UML not?

- **Not a programming language** (advantageous for the modeling of complex circumstances)
- **Not redundant free**; often multiple representation or style possibilities: sequence and collaboration diagrams,
- **Not a process description, no procedure model** (even when it appears so)
- **No automated tools** or tool descriptions
- UML diagrams are **limited** in what they can express. Not everything can be represented with UML
• UML shows the problem from **multiple perspectives**. Repetition of each aspect in the form of a definite diagram.

(example: the design of a house, which is comprised of a floor plan, outline, lateral view, etc.)

- Each of these diagrams has a **special application area** for which it was purposely designed.

• The diagrams contain **different graphical elements**

- The meaning of the element (semantic) is exactly determined and in all diagram forms the same.

• The UML can be very complex due to the many diagrams, but strongly represents the problem to be solved.
Diagram types in UML

- **Use-Case-Model**
  - Abstract description of the collaboration of actors with a system. Also sometimes called a black-box system.

- **Static Model**
  - Class and instance diagrams: the most important diagram type of UML.

- **Dynamic Model**
  - Activity diagram: shows the progression of activities
  - Interaction diagrams: graphical representation of a scenario (with emphasis on a timely progression)
  - Condition diagram: shows the progression of conditions
### Diagram Types and Application Areas

<table>
<thead>
<tr>
<th>Diagram Type</th>
<th>Diagrams</th>
<th>Phase</th>
<th>Application Area</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use cases</strong></td>
<td><strong>Use case diagrams</strong></td>
<td>• Requirements</td>
<td>Business processes, general applications</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Definition</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Creation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Delivery</td>
<td></td>
</tr>
<tr>
<td><strong>Static Models</strong></td>
<td><strong>Class diagrams</strong></td>
<td>• Definition</td>
<td>Everywhere, the class diagram is the most important diagram in UML</td>
</tr>
<tr>
<td></td>
<td><strong>Instance diagrams</strong></td>
<td>• Creation</td>
<td></td>
</tr>
<tr>
<td><strong>Dynamic Models</strong></td>
<td><strong>Interaction diagrams</strong></td>
<td>• Requirements</td>
<td>Shows the information flow and the time flow of the cooperation of the objects</td>
</tr>
<tr>
<td></td>
<td><strong>Collaboration diagrams</strong></td>
<td>• Definition</td>
<td>Timeflow structure with few messages</td>
</tr>
<tr>
<td></td>
<td><strong>Sequence diagrams</strong></td>
<td>• Creation</td>
<td>Timeflow structure with few classes</td>
</tr>
<tr>
<td></td>
<td><strong>State diagrams</strong></td>
<td>• Requirements</td>
<td>Representation of the dynamic behavior</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Definition</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Creation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Delivery</td>
<td></td>
</tr>
</tbody>
</table>
Outline

Lecture Content

1. Preface
2. Basics
3. Object orientation

3.1. Introduction and basic concepts
   3.1.1. Introduction
   3.1.2. Objects and classes
   3.1.3. Attributes of object orientation

3.2. Object-oriented modelling with UML
   3.2.1. Intro
   3.2.2. Use Case models
   3.2.3. Static models
   3.2.4. Dynamic models
   3.2.5. Summary

3.3. Methods of the OO Analysis and Design
### Diagram types and application areas

<table>
<thead>
<tr>
<th>Diagram type</th>
<th>Diagram</th>
<th>Phase</th>
<th>Application area</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use cases</strong></td>
<td><strong>Use case diagrams</strong></td>
<td></td>
<td>Business processes, general applications</td>
</tr>
<tr>
<td><strong>Static Models</strong></td>
<td><strong>Class diagrams</strong>&lt;br&gt;<strong>Instance diagrams</strong></td>
<td></td>
<td>Everywhere, the class diagram is the most important diagram in UML</td>
</tr>
<tr>
<td><strong>Dynamic Models</strong></td>
<td><strong>Interaction diagrams</strong>&lt;br&gt;<strong>Collaboration diagrams</strong>&lt;br&gt;<strong>Sequence-diagrams</strong></td>
<td></td>
<td>Shows the information flow and the time flow of the cooperation of the objects&lt;br&gt;Timeflow structure with few messages&lt;br&gt;Timeflow structure with few classes</td>
</tr>
<tr>
<td></td>
<td><strong>State diagrams</strong></td>
<td></td>
<td>Representation of the dynamic behavior</td>
</tr>
</tbody>
</table>
Use Case Model (UCM)

- Use case models describe on an abstract level the collaboration of actors (users) with a system (computer).
- Use-case models define:
  - **Actors**: what is outside the system
  - **Use case**: what the system should achieve
- Tasks of the use case model are:
  - Limiting the system (a simple rectangle symbolizes the system borders)
  - Describing the functionality (from the user’s point of view)

The use case model can be understood to be the contract between customer and developer.
Elements of the UCM

- **Actor:**
  - Are all that **must exchange information with the system**
  - Doesn’t have to be a person
  - Can be another system that communicates with the compiled system

- **Use Case:**
  - Can contain more than 1 user scenario
  - Describes how a **business transaction** should run through the interaction of an actor with the system and
  - Which steps the user must take in order to manage a specific operation.
Relationships between use cases and actors

• An **inheritance** hierarchy between actors is possible: when different actors can play similar roles, that means that they inherited these roles from the abstract actors.

• The following relationship exists between the use cases:
  - The **“extended”** relationship shows that a **use case extends another use case**
  - The user relationship lets us see that inside a use case, another use case appears - but which also could have appeared alone.
Example of a Use Case Diagram

3.2. object oriented modelling with UML

3.2.2. Use Case Model

System

Read design drawing

Modify design drawing

Print design drawing

actors

Employee

Constructor

Work scheduler

Use cases

System borders


[Otte05] Prof. Dr.-Ing. Martin Otter: Vorlesung „Objektorientierte Modellierung mechatronischer Systeme“, DLR, 2005
Web:http://www.eat.ei.tum.de/lehre/otter-vorlesungen.html

