Software Engineering

Lesson 03
UML / Structural Diagrams
v2.4

Uwe Gühl

Fall 2007/ 2008

Contents

- Structure Diagrams
  - Context – UML 2 Diagram types
  - Overview
  - Diagrams in detail
- Introduction to Object Orientation
- Object
- Class
  - Diagrams and Java-Code
- Same values and identity
- Inheritance
- Polymorphism
- Associations between Classes

Structure Diagrams Context

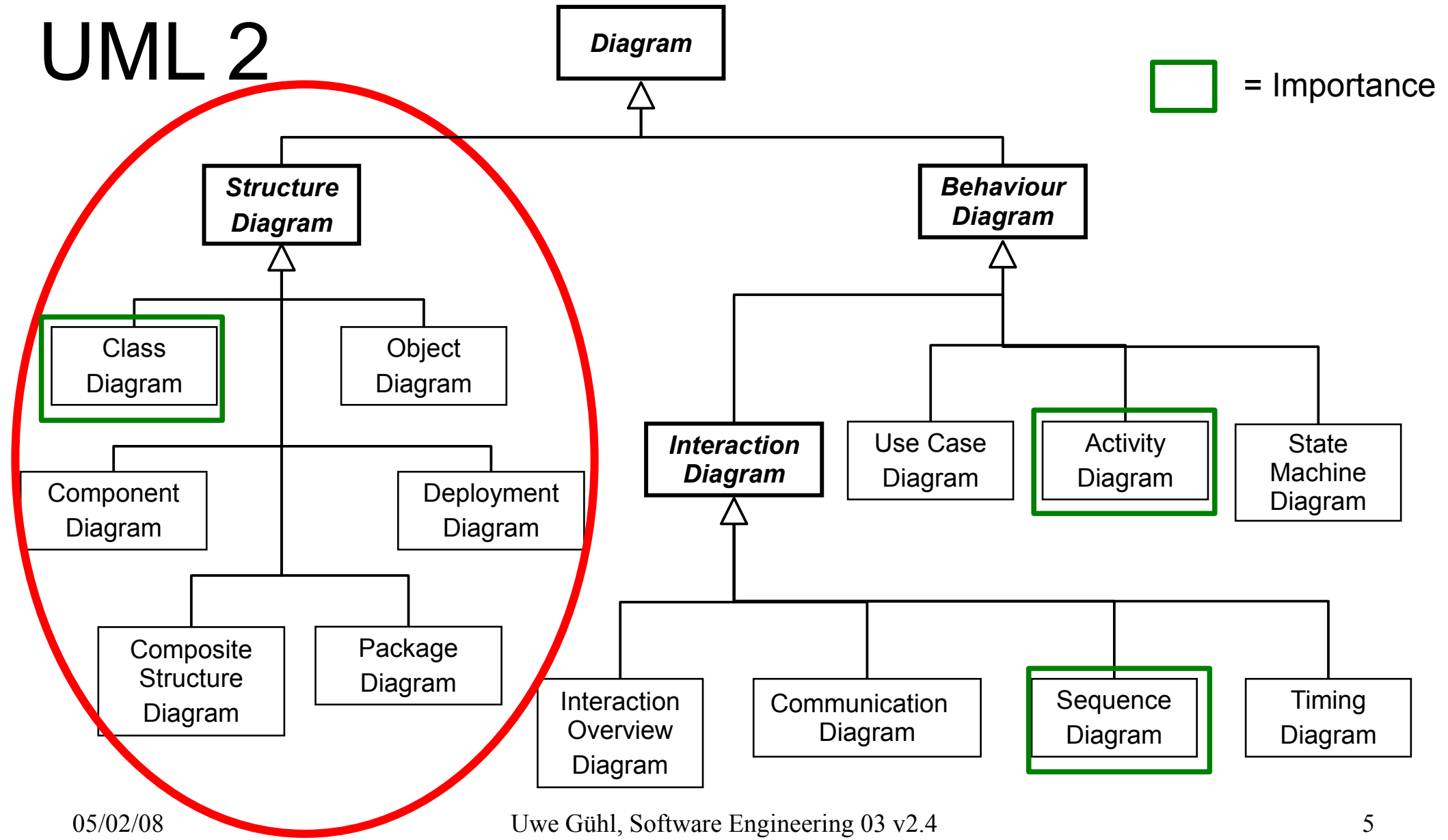
- A big milestone in the development of the Unified Modeling Language (UML) was the version 2.0 established in 2005
- Most UML tools now support most of UML 2.0
- The current UML version available is 2.1.1 (November 2007)

http://www.uml.org/#UML2.0

Structure Diagrams Context

- UML Structure and Behaviour Diagrams
  - Structure Diagrams show basic information of a class, or a complete organization of an architecture or whole system. They support a structural view focussing on the static structure of the system using objects, attributes, operations, and relationships.
  - Behaviour Diagrams describe the dynamic behaviour view focussing on object activities and interactions as well as internal changes of states. Use Case Diagrams show functional requirements from the user's point of view.

Structure Diagrams Overview



Structure Diagrams Overview

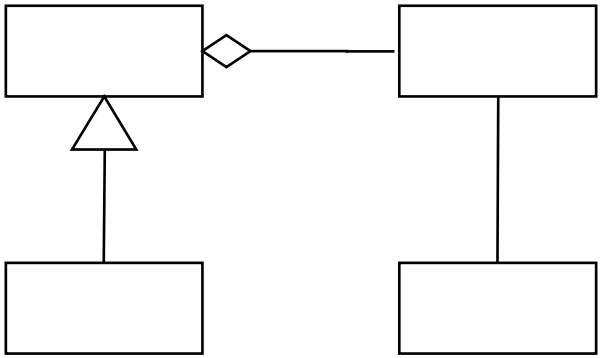
- ***Structure Diagrams***
  - **Class Diagrams** are the central diagrams in UML, showing classes and the relationships between them
  - **Object Diagrams** show instances of the abstract structures in Class Diagrams
  - **Component Diagrams** demonstrate system interrelations

Structure Diagrams Overview

- ***Structure Diagrams***
  - **Deployment Diagrams visualize** the execution architecture of systems and focus on hardware
  - **Composite Structure Diagrams** show the internal structure of a class or component and indicate interaction points to other parts of the system
  - **Package Diagrams** depict the logical system structure. Classes are bundled to packages to keep the overview

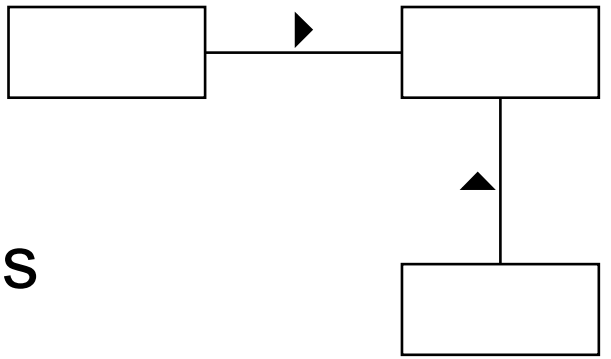
Structure Diagrams
Class Diagram

- Questions to be answered
  - What are the classes of my system?
  - What are their relationships?
- Strengths
  - Describes statical structure
  - Shows structure dependencies and data types
  - Basic for Behaviour Diagrams



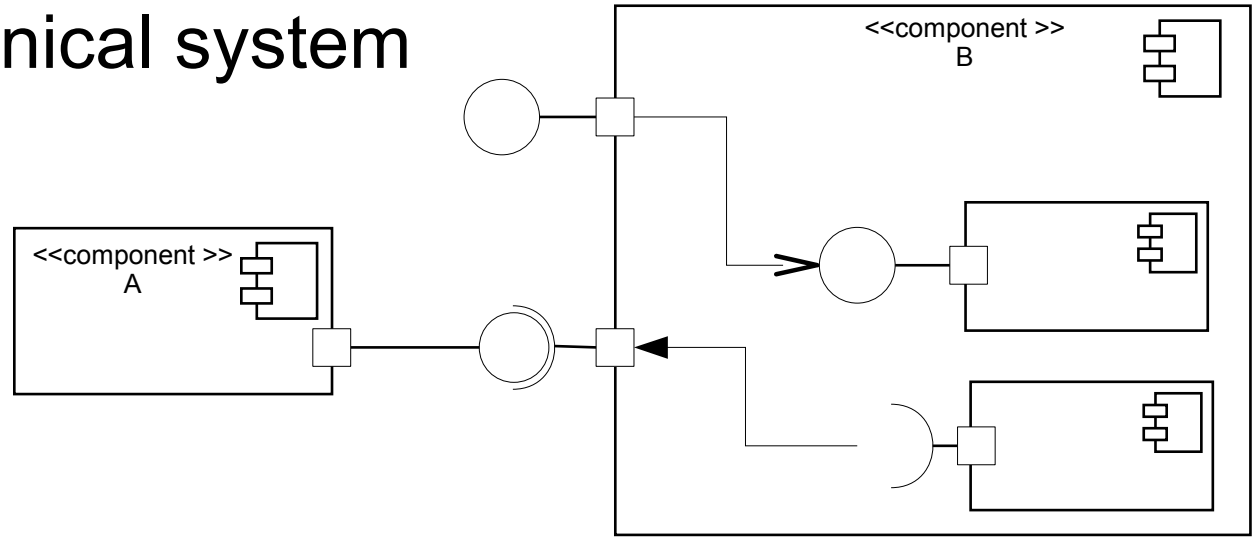
Structure Diagrams
Object Diagram

- Questions to be answered
  - What's the status of my system at a particular time (snapshot of Class Diagram)
- Strengths
  - Shows objects and their attribute values at a given time
  - Details like in the corresponding Class Diagram
  - Very good presentation of relations of quantity



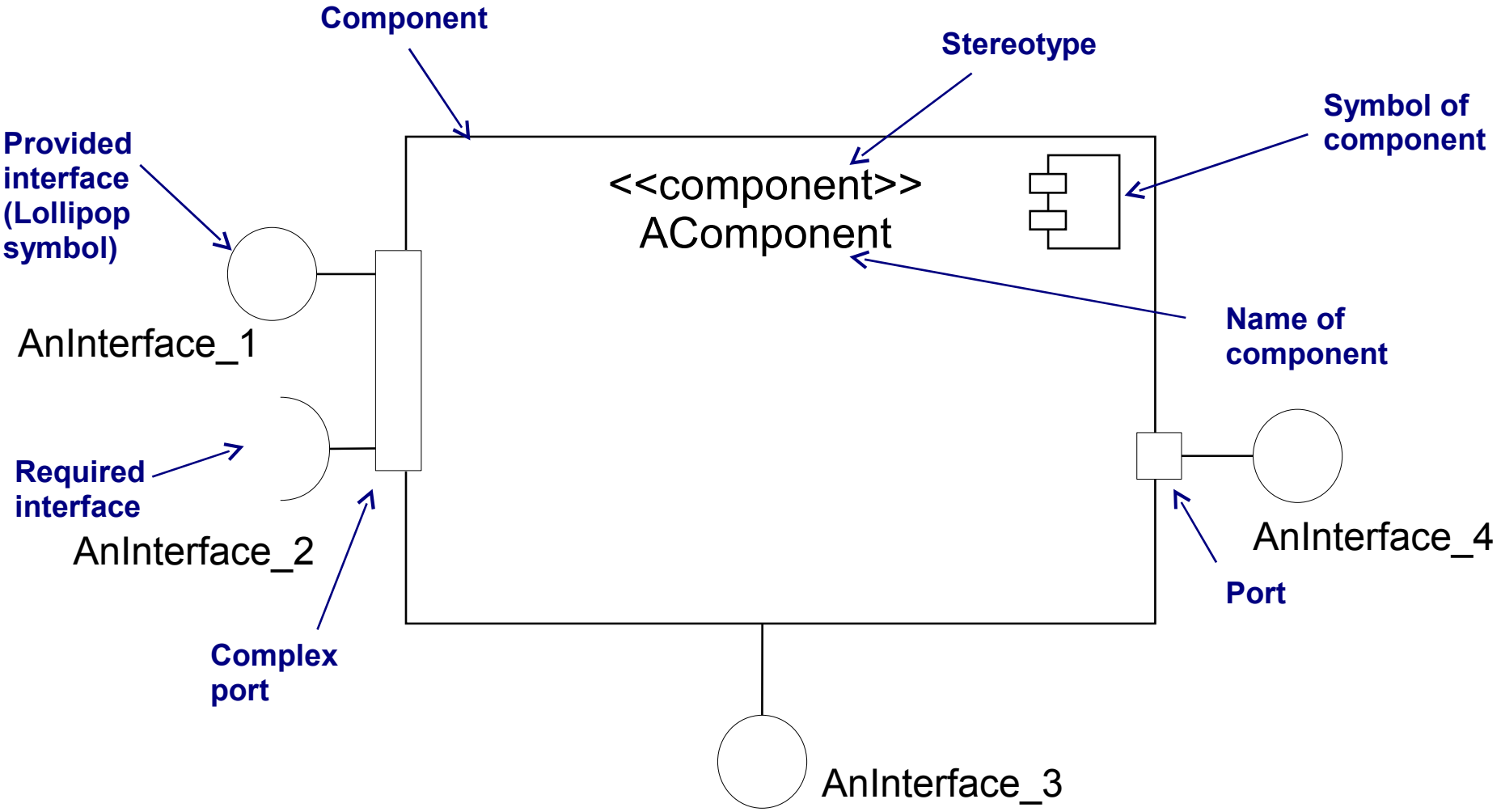
Structure Diagrams
Component Diagram

- Questions to be answered
  - How could classes could be bundled to components?
  - What are the relationships of the components?
- Strengths
  - Organizes technical system components
  - Enables modelling of interfaces



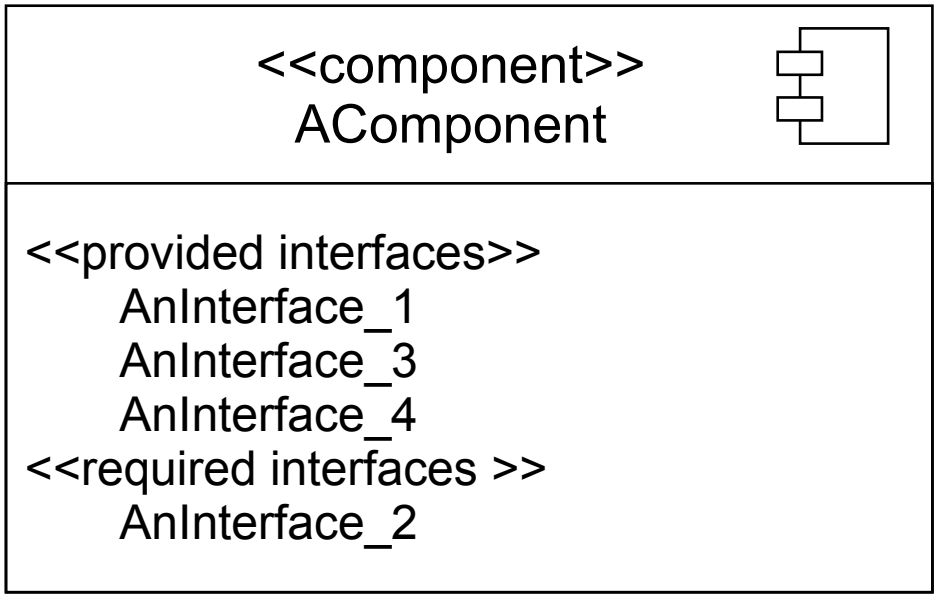
Structure Diagrams
Component Diagram

- Black Box representation



Structure Diagrams
Component Diagram

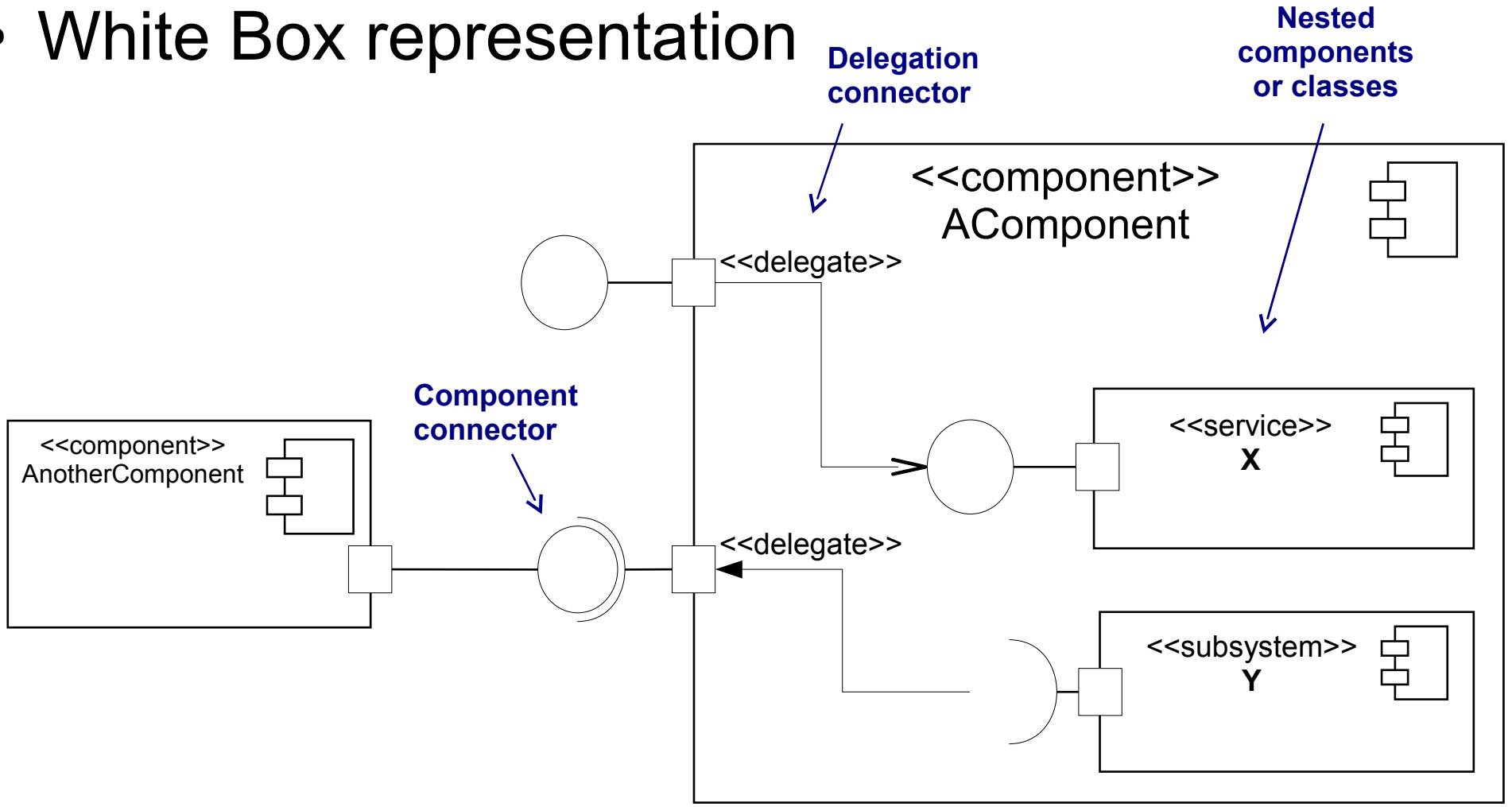
- Black Box representation - alternative



Stereotypes for components could be for example
<<specification>>
<<implement>>
<<entity>>
<<service>>
<<subsystem>>.

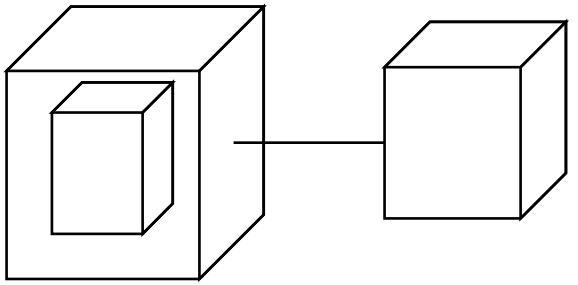
Structure Diagrams
Component Diagram

- White Box representation



Structure Diagrams
Deployment Diagram

- Questions to be answered
  - How does the environment (hardware, server, data bases, …) of the system look like?
  - How is the deployment of the components at runtime?
- Strengths
  - Shows runtime environment of the system – mainly hardware
  - Enables presentation of software server
  - High abstraction level, few notation elements

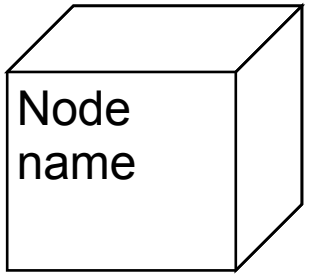


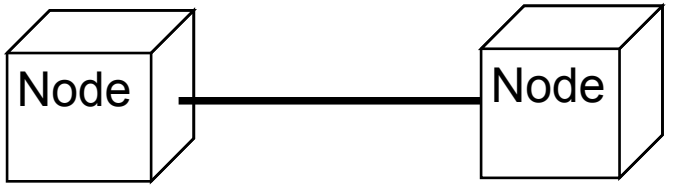
Structure Diagrams
Deployment Diagram

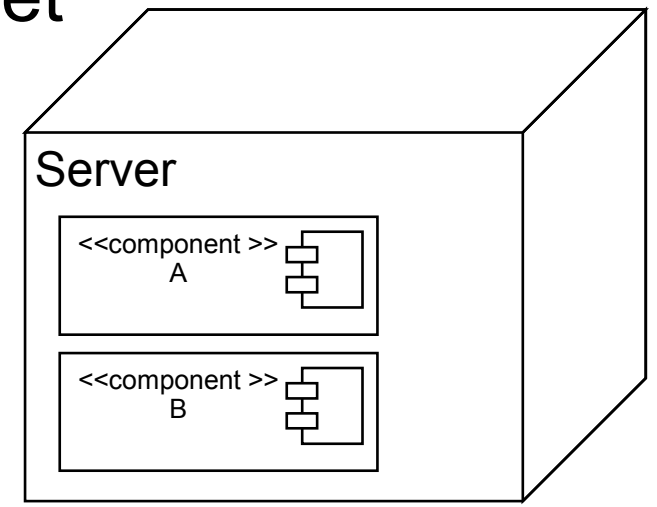
- Deployment diagrams
  - depict the physical resources in a system including
    - nodes
    - components
    - connections
  - show
    - hardware of a system
    - software that is installed on that hardware
    - middleware used to connect the machines

Structure Diagrams Deployment Diagram

- Notation of elements
  - Node
    A physical resource that executes code components.
  - Association
    refers to a physical connection between nodes, e. g. Ethernet
  - Components and Nodes
    Components inside a node deploys them







Structure Diagrams
Deployment Diagram

- Notation of elements

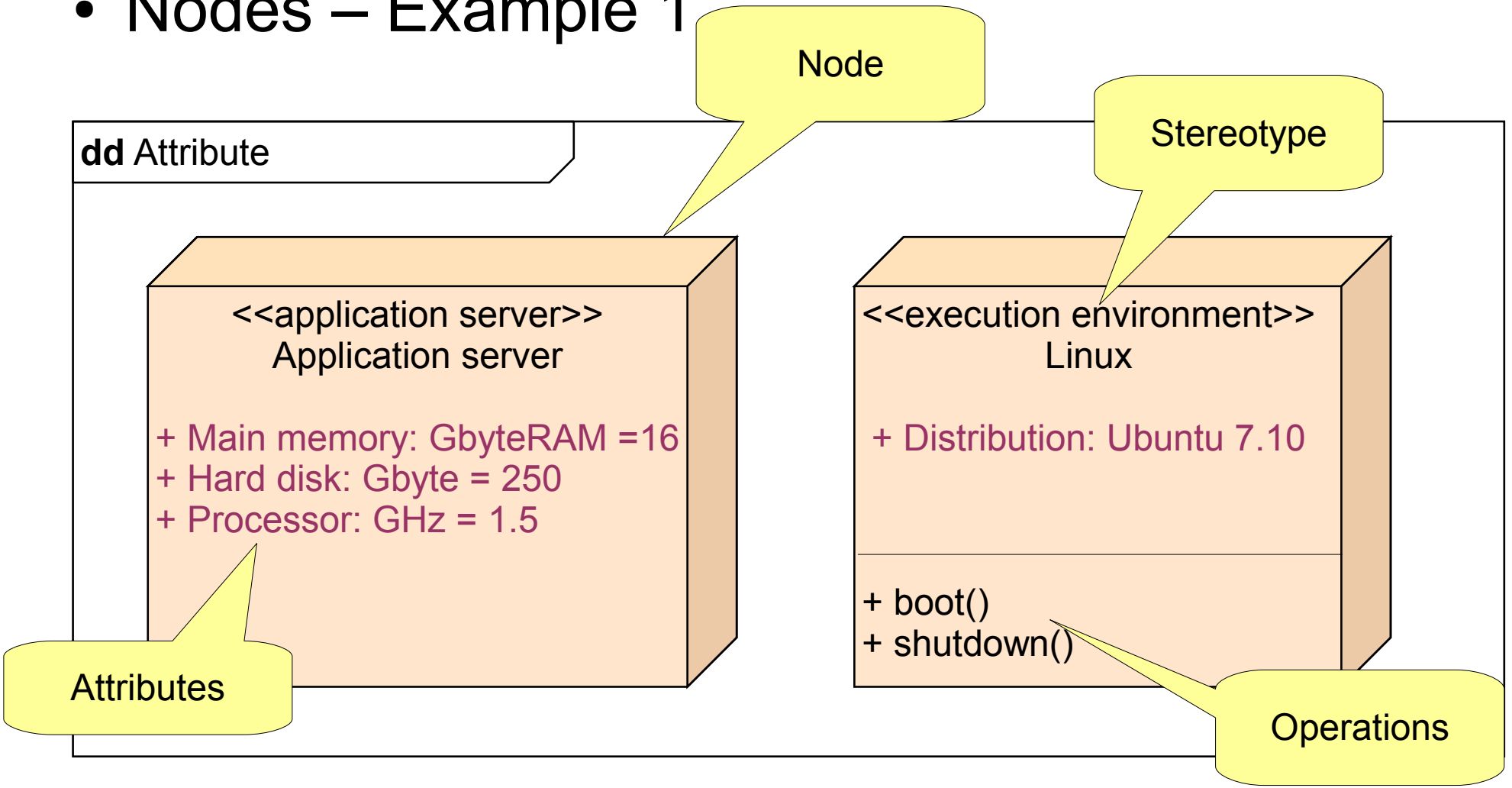
<<device>>	General node type
<<execution environment>>	... for components like J2EE server or operating system
<<application server>>	Specifies an application server contenting in general an <<execution environment>>
<<client workstation>>	Takes services of an <<application server>>
<<mobile device>>	Mobile phone, notebook
<<embedded device>>	Integrated systems

Structure Diagrams
Deployment Diagram

- Nodes offer a possibility to represent a system resource:
  - Device: Hardware unities at runtime (e. g. computer, hard disk)
  - Execution Environment:
    - Run time environment
      (e. g. operating system, EJB-Container,…)
    - Could offer explicit interfaces as well – like EJB Server

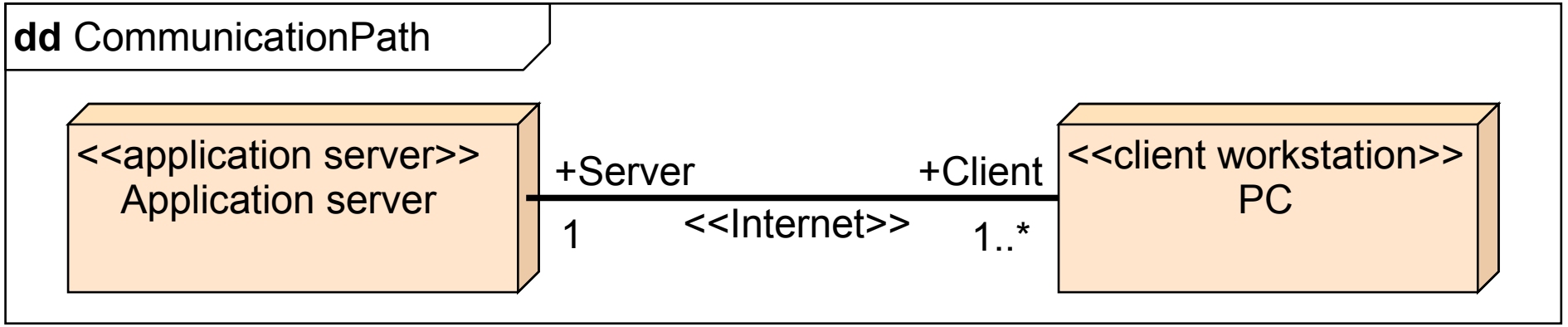
Structure Diagrams
Deployment Diagram

- Nodes – Example 1



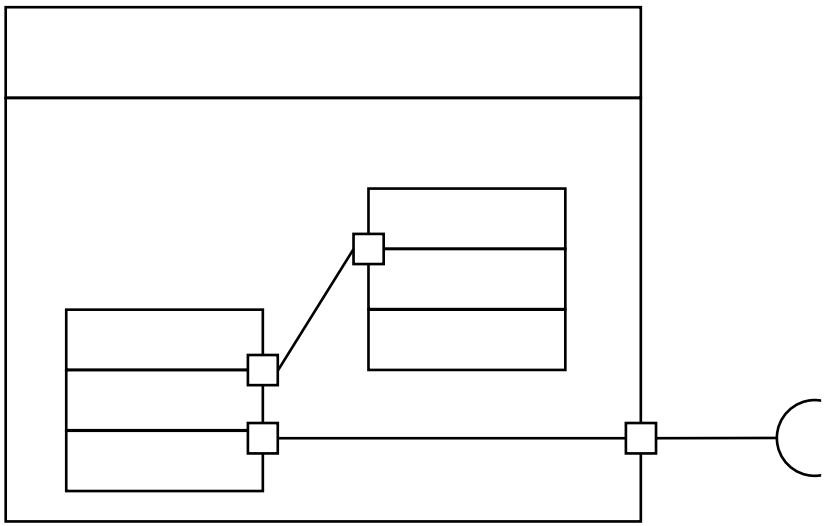
Structure Diagrams
Deployment Diagram

- Nodes – Example 2
  - Association between two nodes to exchange signals / messages
  - including multiplicity
  - Stereotypes (*UML does not define Standard stereotypes*)



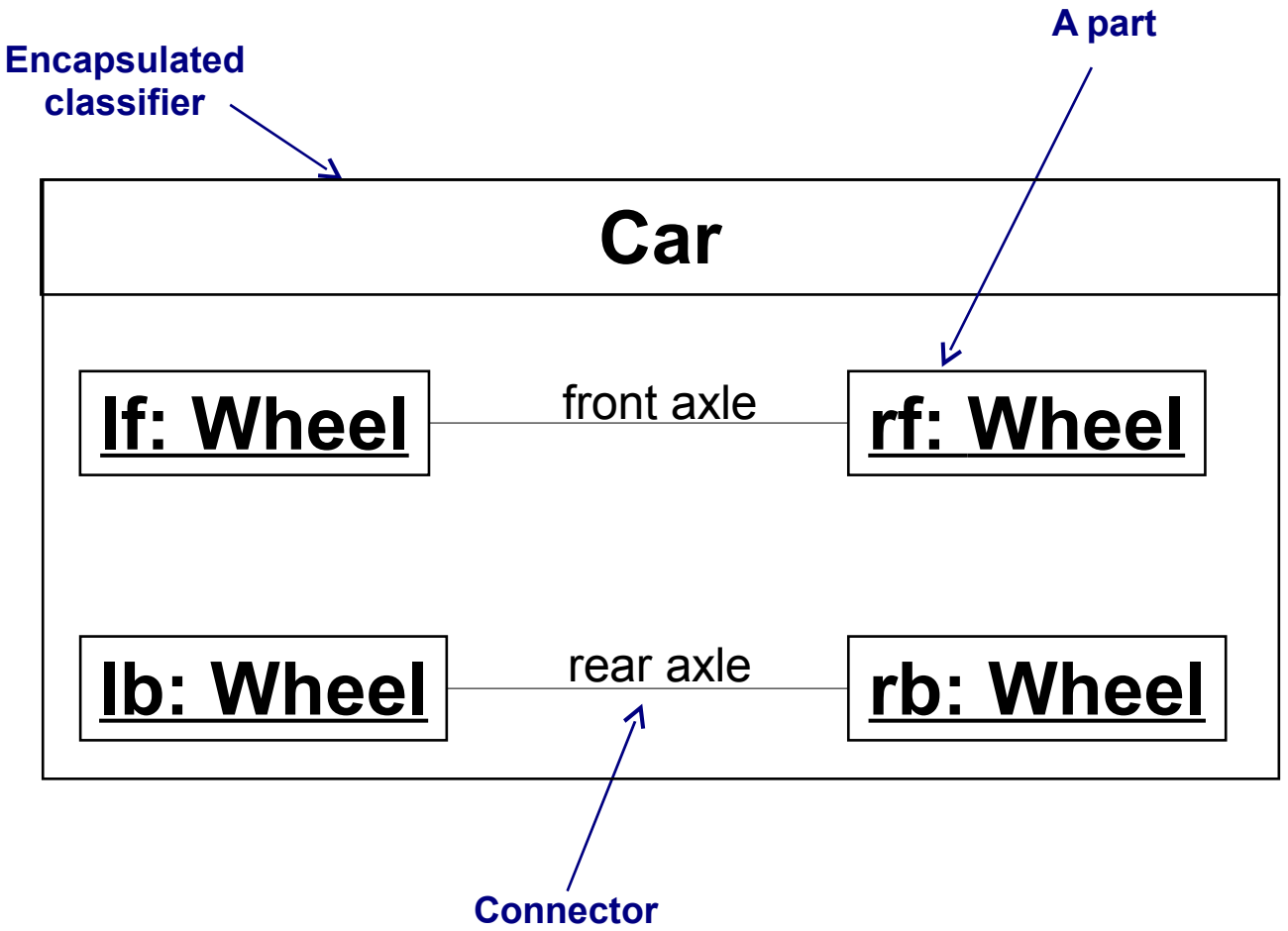
Structure Diagrams
Composite Structure Diagram

- Questions to be answered
  - How does the inner part of a class, component, or subsystem look like?
- Strengths
  - Good for top down modelling
  - Medium detail level
  - Modelling of part relationships with ports



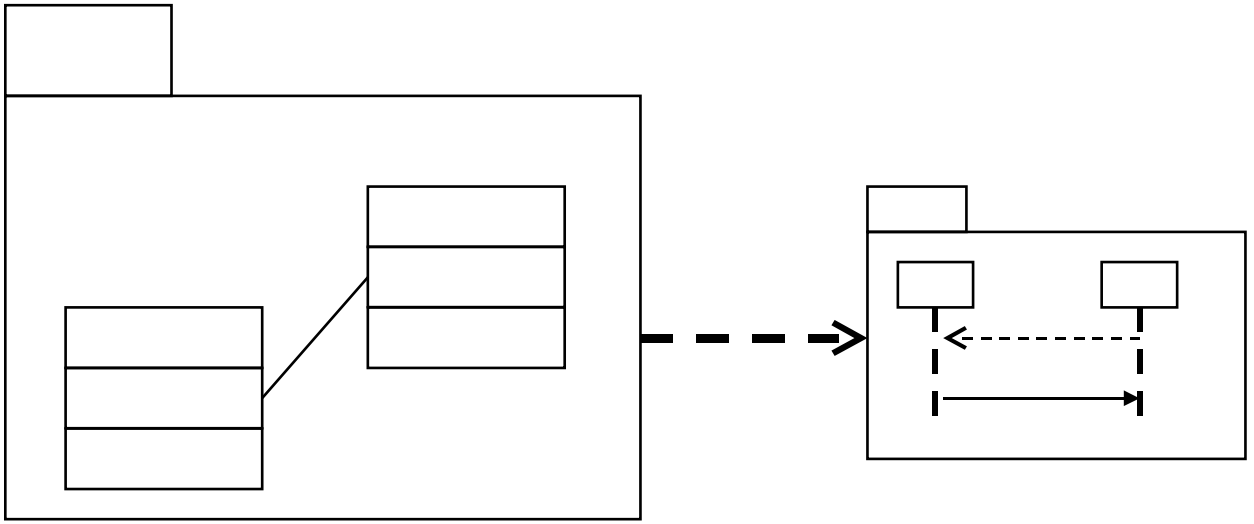
Structure Diagrams
Composite Structure Diagram

- Example

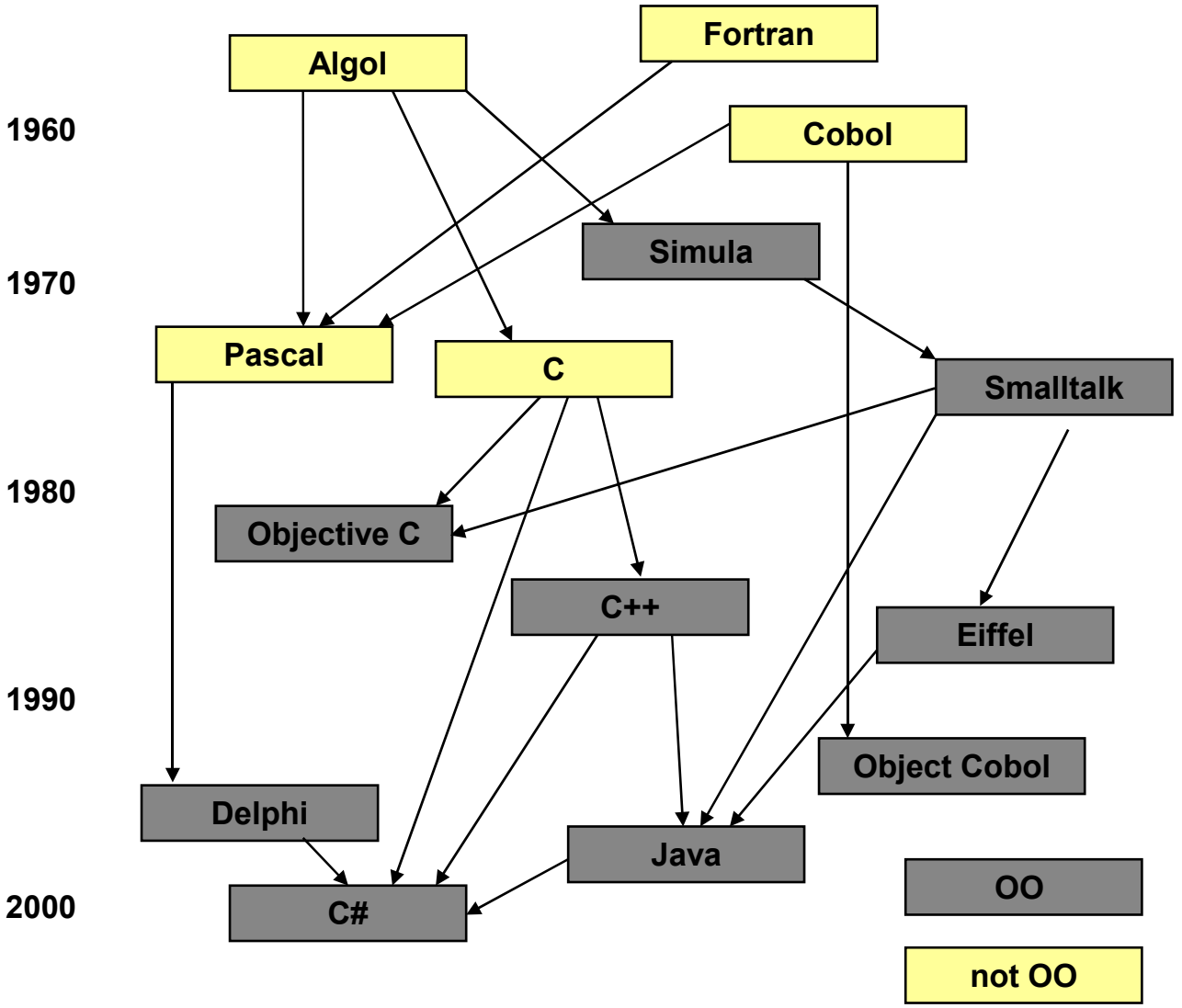


Structure Diagrams
Package Diagram

- Questions to be answered
  - How should I partition my system for clarity?
- Strengths
  - shows how a system is split up into logical groupings by showing the dependencies among them
  - logical hierarchical decomposition of a system.



Introduction to Object Orientation Programming Languages



Introduction to Object Orientation Terms

Object

Class

Instance

abstract Class

instance variable

Subclasses

Inheritance

Superclass

Class variable

multiple inheritance

Method

Message

Overloading

Introduction to Object Orientation Comparison

Pragmatically

Object

+

Class

+

Inheritance

Abstract

Encapsulation

+

Classification

+

Polymorphism

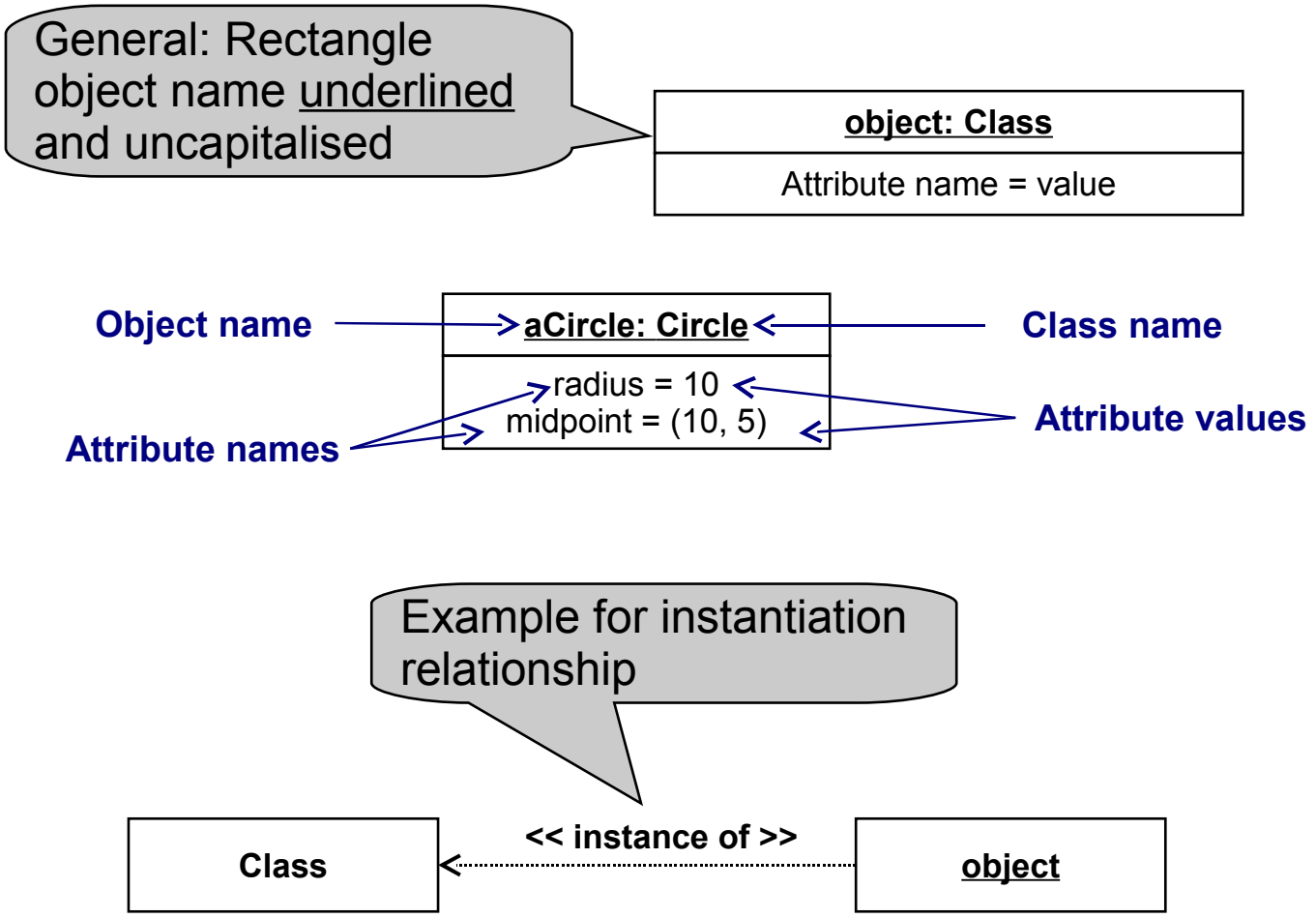
Object

- An object is a capsule for state and behaviour
- Attributes (variables) save the status (the data) of an object
- Operations (Methods, or Services) define the behaviour of the object
- Operations change a status of an object
- An object is a concrete exemplar, an instance, of a class and has an own identity, independent from the values of its attributes

Object

- The idea of encapsulation: Access to the object to get information about status is only possible with defined operations
- Default / Confirmation / Invariance:
  Attribute, that is always assured
  Example: Radius of a circle-object

Object Design in UML

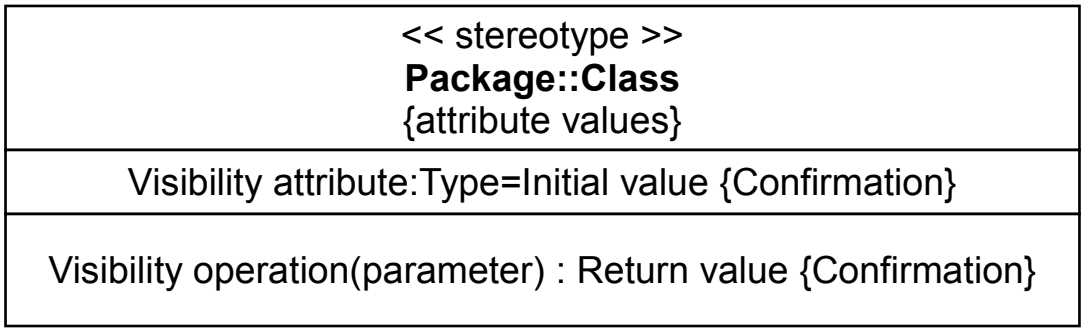


Class

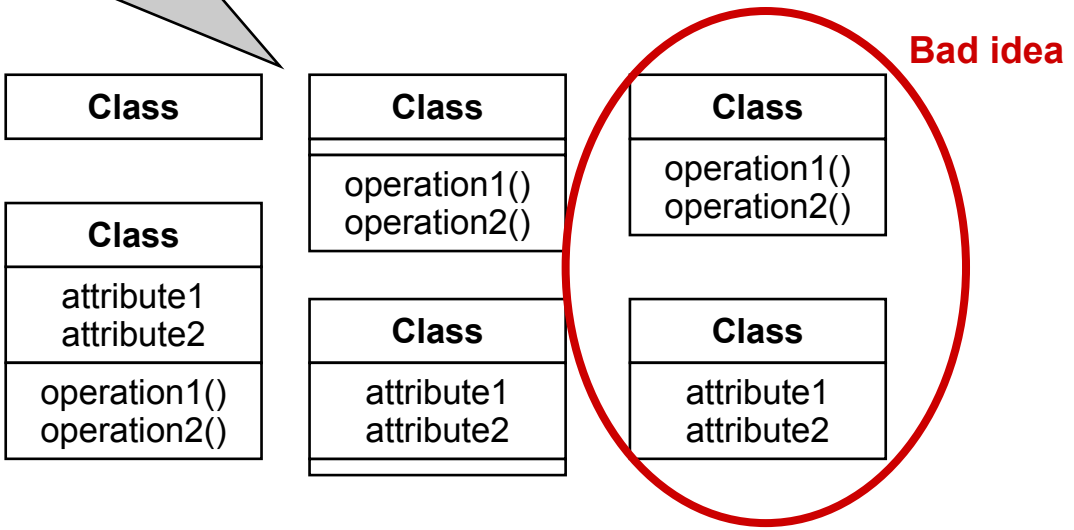
- Definition of attributes and operations for a set of objects – that means classes are templates for concrete objects
- All objects of a class follow this definition, they differ only in the concrete values of the attributes
- Synonym for Class: Type

Class Design in UML

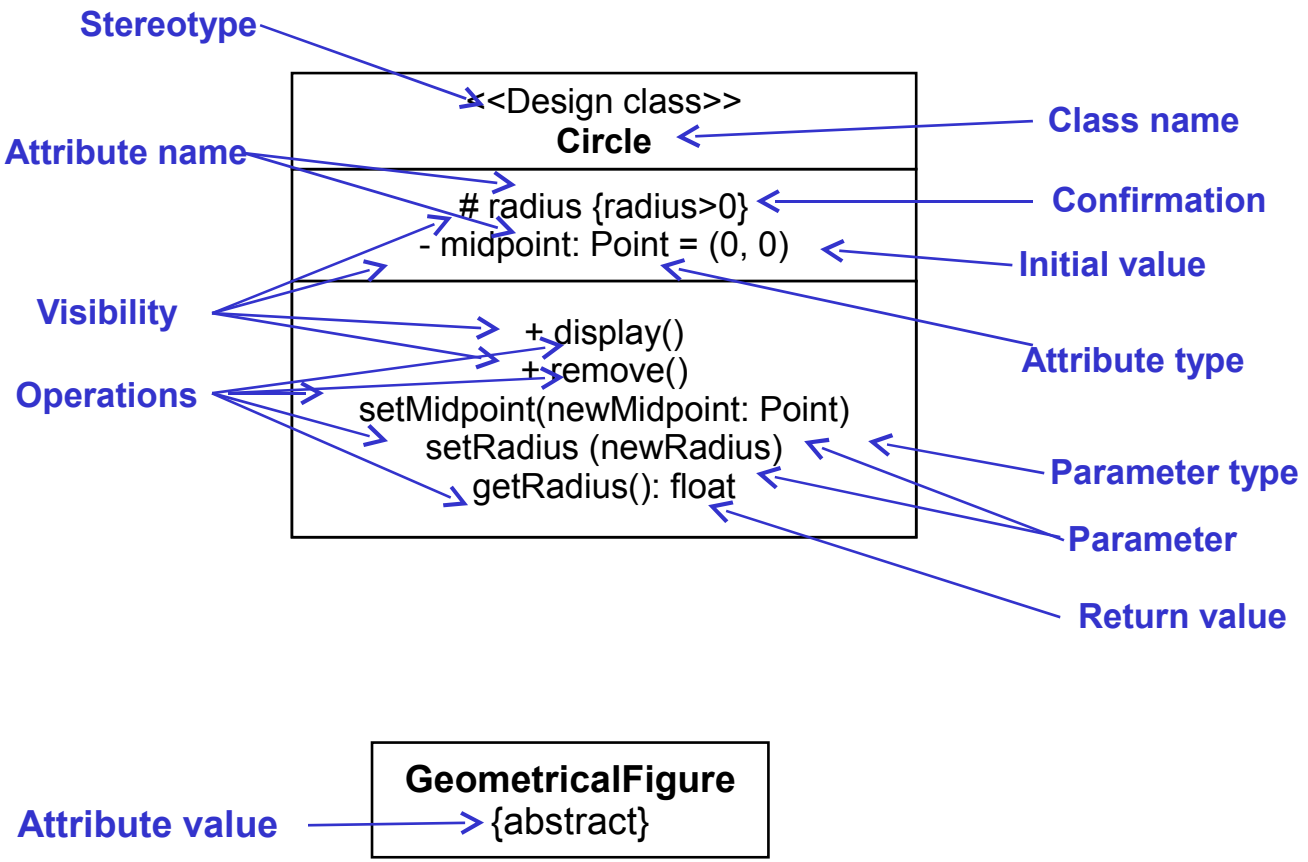
Design as rectangle, name **bold and capitalized.** Below if required attributes and operations, separated by a horizontal line



Alternative shortened descriptions

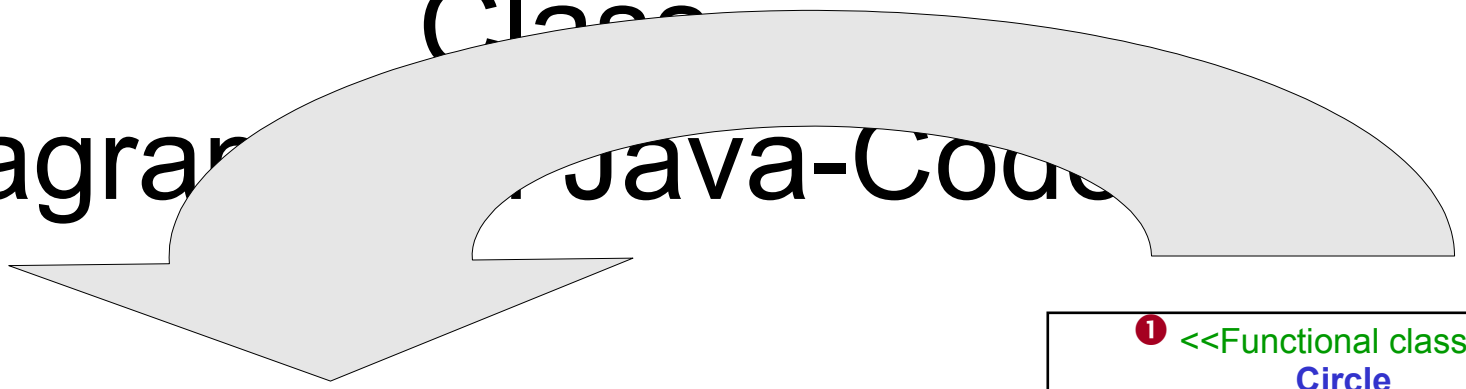


Class Design in UML



Examples for Stereotypes: <<Interface>>, <<Presentation>>, <<Enumeration>>
Examples for attribute values: {readOnly}, {obsolete}, {persistent}, {Version=1.5}, {Author=Hans Meier}

Class Diagram ⇄ Java-Code



❶ <<Functional class>>
Circle

❸

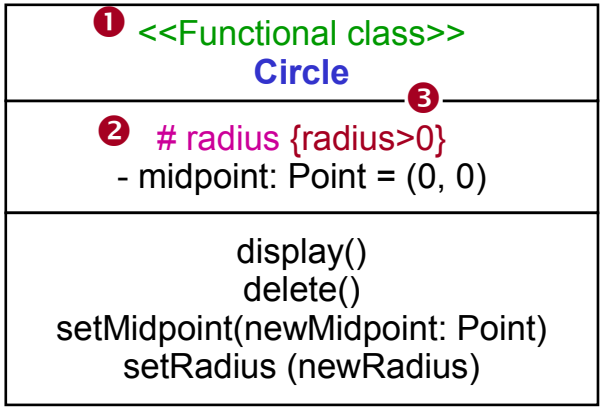
❷ # radius {radius>0}
- midpoint: Point = (0, 0)

display()
delete()
setMidpoint(newMidpoint: Point)
setRadius (newRadius)

```
❶
/* Functional class */
❷public class Circle {
  protected float radius;
  private Point midpoint = new Point(0,0);

  void display(){/* ToDo */};
  void delete(){/* ToDo */};

  void setMidpoint(Point newMidpoint) {
    midpoint = newMidpoint;
  };

  public void setRadius(float newRadius) {
    if (radius <= 0.0)
❸     System.err.println(
      "Fault: Radius must be greater than 0: "
       + newRadius);
    else
       radius = newRadius;
  }
}
```

Class Diagram and Java-Code (2)

1. A UML Diagram may content information, which is not necessary for coding (e. g. Stereotype) – could be explained with comments



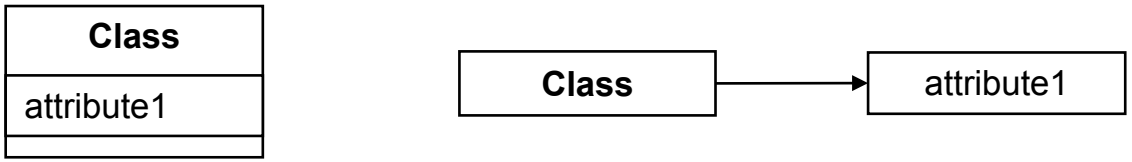
2. Default values in a UML Diagram must possibly be implemented explicitly
3. A valid UML Diagram has not to have all information for a complete program (e. g. visibility, types, program code)

Class Diagram and Java-Code (3)

- Because the UML is (more or less) independent from programming languages, some modelling opportunities exist, which are only interesting for specific languages, e. g.
  - Meta classes for Smalltalk
  - Composition for C++

Class Attributes

- Possible version



- Simplest case
  - Presentation of attribute only as name
- General syntax

```
[visibility] [/] name
[: type] [multiplicity] [=default] [{property-string}]
```

Class Attributes

```
[visibility][/] name
[: type] [multiplicity] [=default] [{property-string}]
```

- The visibility for attributes and operations could be

`+ : public`
Unlimited access

`# : protected`
Attribute is accessible in the defining class and in all subclasses of the defining class

`- : private`
Only the defining class may access

`~ : package`
All classes in the same package may access

Class Attributes

```
[visibility][/] name
[: type] [multiplicity] [=default] [{property-string}]
```

- The slash `[/]` shows if the attribute is derived and could be determined from other elements. For example, length of a line could be a derived attribute constructed from the line's two endpoints.

Class Attributes

```
[visibility][/] name
[: type] [multiplicity] [=default] [{property-string}]
```

- Example:
  - Calculation of the age from date of birth and current system time.
  - Java code (No instance variable of `age`!)

Person
dateOfBirth: Date / age: int
+ getAge(): int

```
public class Person {
   GregorianCalendar dateOfBirth = new GregorianCalendar();

   int getAge(){
      return   GregorianCalendar.getInstance().get(Calendar.YEAR)
             - dateOfBirth.get(Calendar.YEAR);
   }
}
```

Class Attributes

```
[visibility][/] name
[: type] [multiplicity] [=default] [{property-string}]
```

- [: type] stands for data type of an attribute
- Die UML offers predefined data types:
  - `Integer`
  - `String`
  - `Boolean`
- There are no limits in data types, even complex data types, which are defined by other classes, are allowed

Class Attributes

```
[visibility][/] name
[: type] [multiplicity] [=default] [{property-string}]
```

- [multiplicity] shows the attribute's multiplicity in square brackets.
- The lower value is on the left, the higher on the right side, separated by two dots `[n .. m]`
- Limits are natural numbers including 0, and "*" for not limited
- If the multiplicity is omitted, 1 is the assumed value

Class Attributes

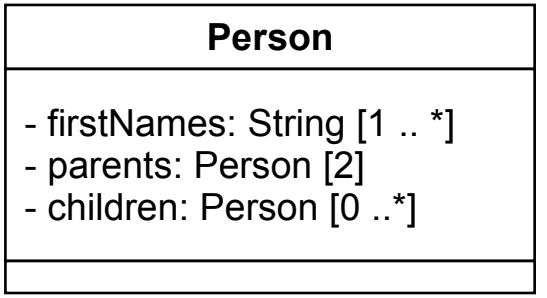
```
[visibility][/] name
[: type] [multiplicity] [=default] [{property-string}]
```

- Often used
  - [0 .. 1]: optional attribute
  - [1 .. 1]: (abbreviated [1]): Attribute may not be empty. If no multiplicity is given, the default [1 .. 1] is valid
  - [0 .. *]: (abbreviated [*]): optional any: any number of instances or empty.
  - [1 .. *]: minimal one instance, no limit
  - [n .. m]: fix: minimal n, maximal m elements. If n = m, then [n] is sufficient.

Class Attributes

```
[visibility][/] name
[: type] [multiplicity] [=default] [{property-string}]
```

- Example
  - The class Person contents attributes for at least one first name, both parents and any children
  - Java-Code:




```
public class Person {
   private String[] firstNames; // minimal one first name!
   private Person[] parents;    // exactly two parents
   private Person[] children;   // any number of children
}
```

Class Attributes

```
[visibility][/] name
[: type] [multiplicity] [=default] [{property-string}]
```

- Default: The default value is set automatically

Class Attributes

```
[visibility][/] name
[: type] [multiplicity] [=default] [{property-string}]
```

- Following property strings can be applied to a given attribute
  - {readOnly}, values may not be changed
  - {union}, e.g. `weekdays : String {union}`
  - {subsets <property-name>}, e.g.
    `workingDays [subsets weekdays]`
    `holidays [subsets weekdays]`

Class Attributes

```
[visibility][/] name
[: type] [multiplicity] [=default] [{property-string}]
```

- {redefines <property-name>}
  The redefines property string is used to redefine an attribute inherited from a superclass to change its name.
- {ordered}, contents of a set ordered without duplicates
- {bag}, no order, duplicities allowed, e.g. `{4,1,4,2}`
- {seq} or {sequence}, contents is ordered and duplicates are allowed
- {composite} attribute is responsible for deleting of contained values

Class Attributes

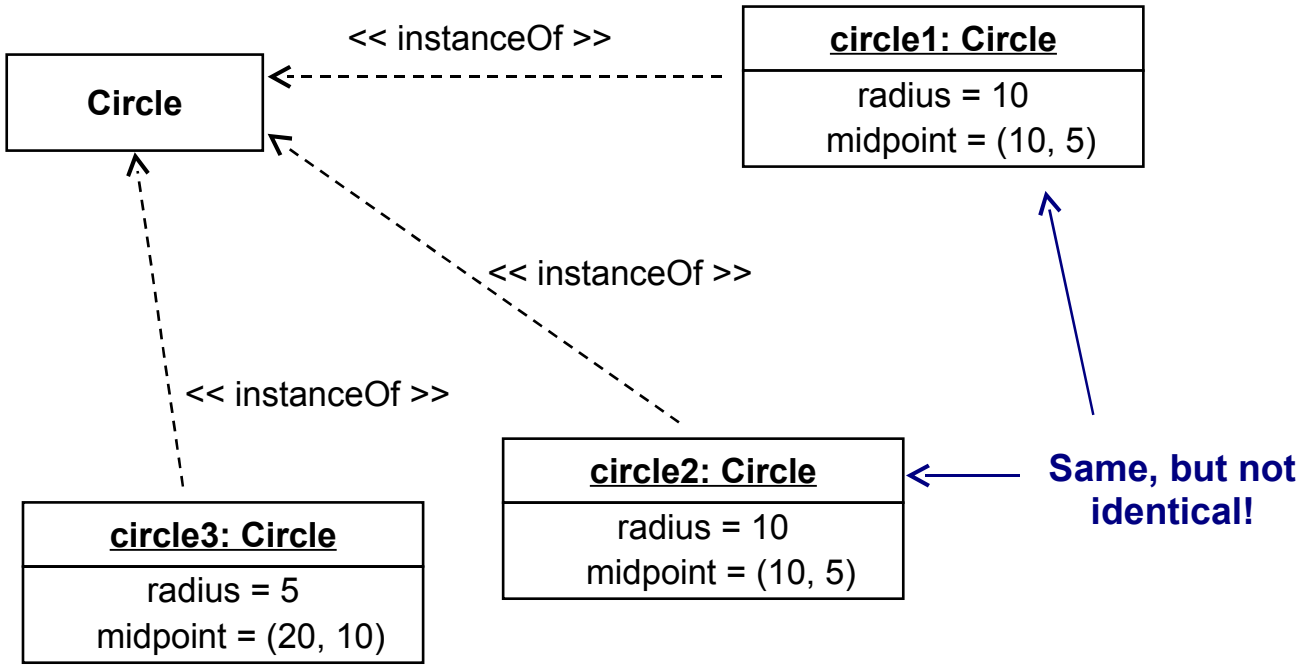
File
- separator: char - name: String
+ getSeparator() : char + setSeparator(char) + getName(): String + setName(String)

- **Static attributes (and methods as well)** get underlined.
- Example:
  - The class file contents a static variable separator, which keeps the path separator of a specific operation system (e. g. ‚/‘ oder ‚\‘)
  - Java code

```
public class File {
  private static char separator;
  private String name;

  public static String getSeparator() {
    return separator;
   }
  public String getName() {
    return separator;
   }
  ...
}
```

Same values and identity

- **Same values:** Same values in attributes
- **Identical:** Same object, same memory address



Same values and identity

- Same Values: Use method "equals(...)"
- Identical: same object, same memory address
  Use Equality operator "=="

```
Circle circle1 = new Circle(10, new Point(10,5));
Circle circle2 = new Circle(10, new Point(10,5));
Circle circle4 = circle1;

//circle1 and circle2 are the same:
circle1.equals(circle2)    // true
circle1 == circle2         // false

//circle4 and circle1 are identical:
circle4.equals(circle1)    // true
circle4 == circle1         // true
```

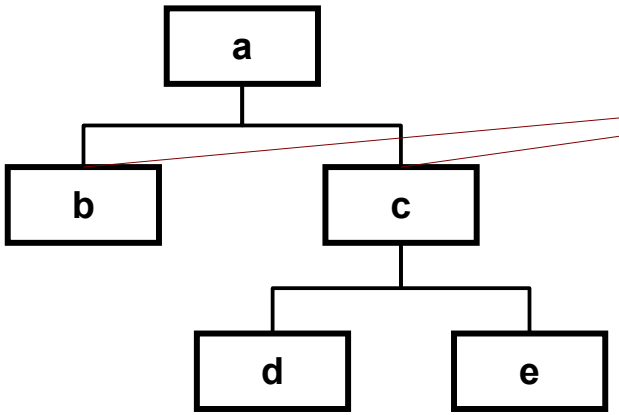
Attention: If you accidentally misuse „same“ instead of „identical“ objects (and vice versa) – it's difficult to detect that kind of failures! (especially collections; ⇨ Deep Copy versus Shallow Copy)!

Same values and identity Excerpt: Deep / Shallow Copy

- A different copy strategy could be a very important difference, especially concerning complex objects and collections
  - Deep Copy
    Copy of an object and of all referenced objects
  - Shallow Copy
    Copy of an object and of all the references.
    Notice: The referenced objects get not copied, only the references, pointing at the original referenced objects!

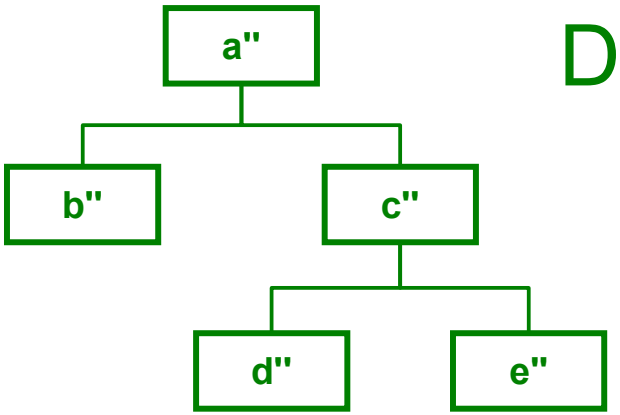
Same values and identity
Excerpt: Deep / Shallow Copy

- Example



Shallow Copy

Attention – Side effect!
Changing a'.b means
you change a.b as well!



Deep Copy

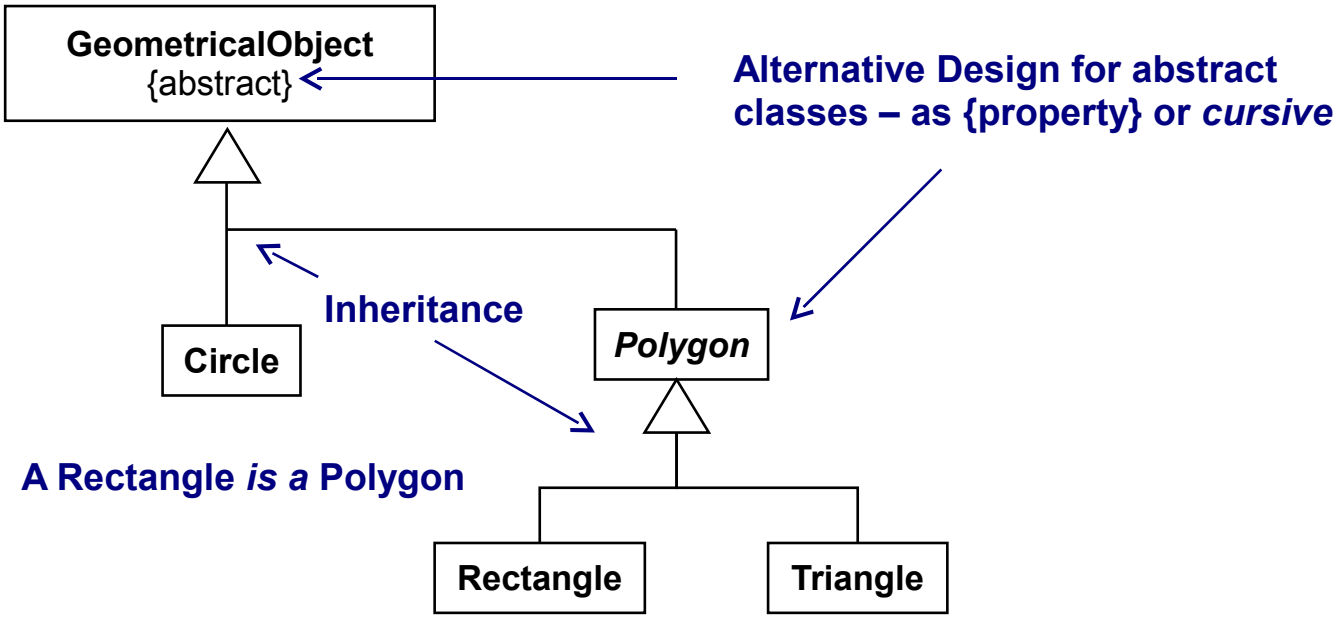
Inheritance

- Concept to define common similarities between classes only once
- Subclasses may
  - Add attributes and operations to the inherited ones from the superclass
  - Overwrite inherited operations
- Differ
  - Simple inheritance – only one superclass (Smalltalk, Java)
  - Multiple inheritance – many superclasses are possible (C++)

Inheritance

- Abstract levels possible
- Classes could be abstract or concrete
- There are no instances from abstract classes
- Abstract class is always superclass
  (makes it sense otherwise ?)
  Idea: Structuring
- Concrete Classes may have subclasses

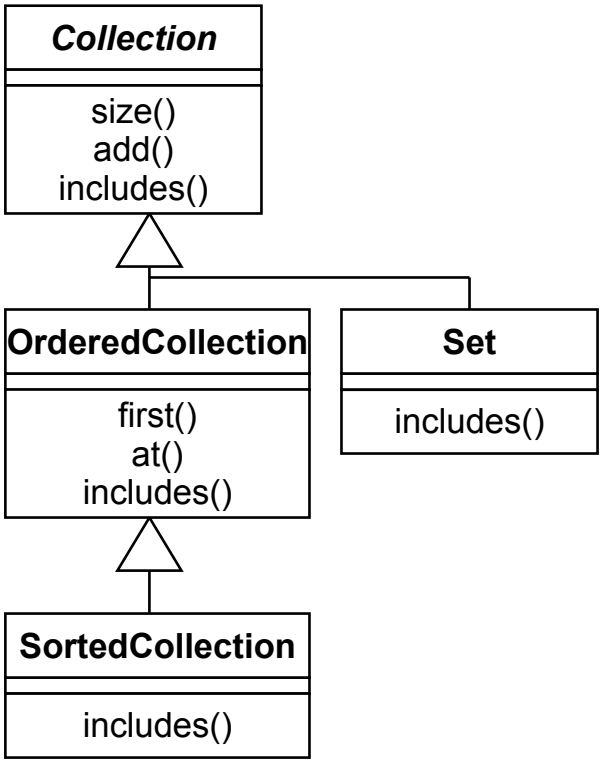
Inheritance



Java-Code:

```
public abstract class GeometricalObject {}
public class Circle extends GeometricalObject{}
public abstract class Polygon extends GeometricalObject {}
public class Rectangle extends Polygon {}
public class Triangle extends Polygon {}
```

Inheritance

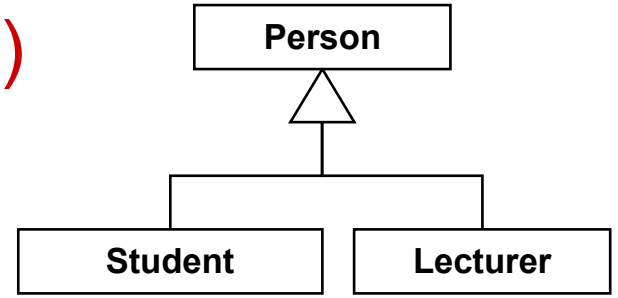


- The abstract Class ***Collection*** offers the common interface for all subclasses.
- The subclasses understand the sum of all operations of all their super classes, completed with own operations
- **SortedCollection** understands
  - size(),
  - add(),
  - includes(),
  - first(), and
  - at().
- The implementation of an operation could be changed independent from the same signature in the sub class, e. g. to optimize the implementation or to consider additional attributes
  Example: includes()

Inheritance
Pros and Cons

- Pro: high locality:
  Similarities of classes are described in only one class
- Pro: easy to extend
  Only the changed behaviour has to be defined in the subclass

- Con: inflexible modelling:
  Changes in future could not be described (➔„Roles“)



- Con: Specialization relationships are not always clear (Example: Rectangle and Square)
- Con: Complexity rises, when more classes are established

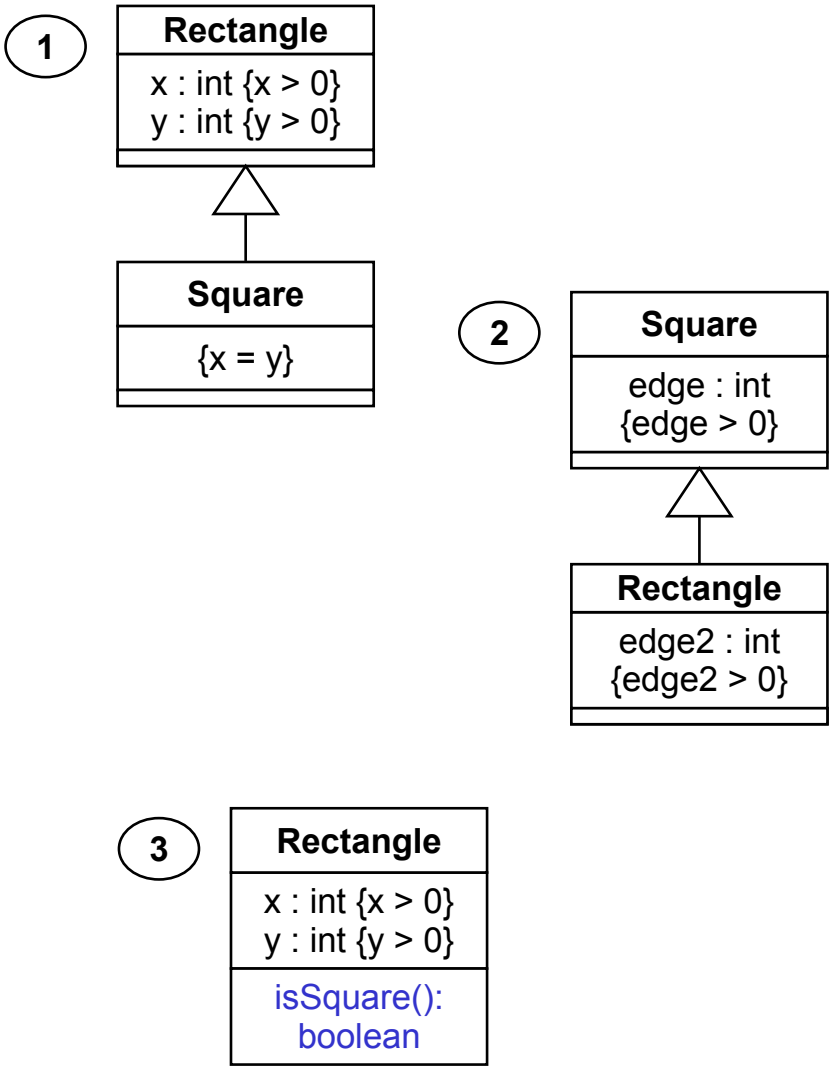
Could be avoided by functionality of an IDE

Inheritance Discussion

(1) Discussion: Wasting disk space in using Squares (useless variable)

(2) Discussion: In static typed programming languages like C++ and Java Square could never be used, if a Rectangle is expected.

(3) Is a class for Square really necessary? How to handle Rectangles having the same length of edges?
➔ Square as property of Rectangles

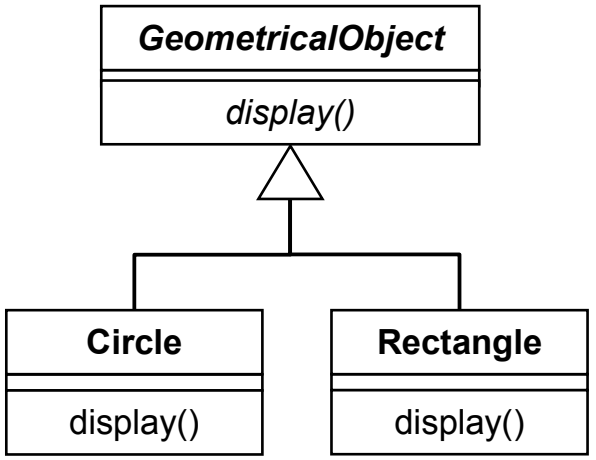


Polymorphy

- Polymorphy means that the same operation has different behaviour in different classes
- Polymorphy is the object oriented alternative to if / else or case statements.

Polymorphy

- Example: Before compiling it is not known, to which object the message display() will be sent. During runtime it will be decided (→ „Late Binding“)



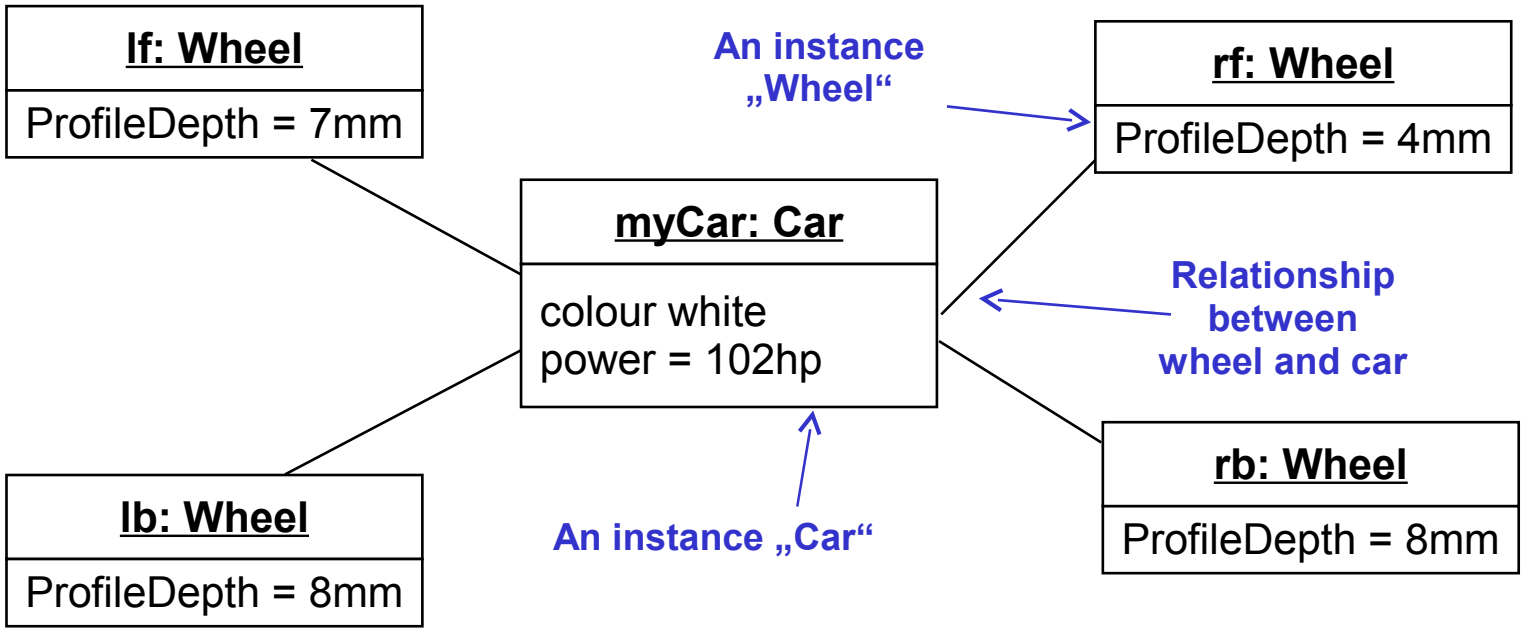

```
LinkedList geomObjects = new LinkedList();
geomObjects.add(new Circle(10, 10, 5);
geomObjekte.add(new Rectangle(0,0,10,20);
Iterator elements = geomObjects.iterator();
while(elements.hasNext())
{
  Object gm = elements.next();
  ((GeometricalObject) gm).display();
}
```

Circle.display()

Rectangle.display()

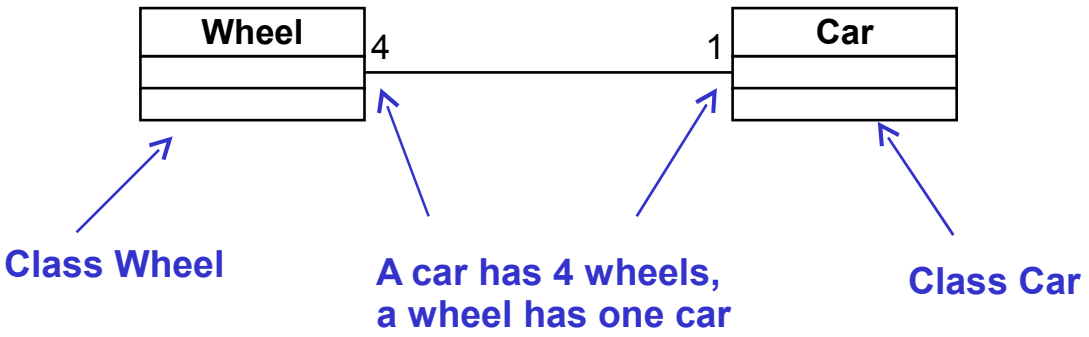
Associations

- Between Objects and Classes could be associations
- Example for object relationship:
  A car has four wheels



Associations between Classes

- Associations could show cardinalities
- Example: A car has four wheels



Associations between Classes

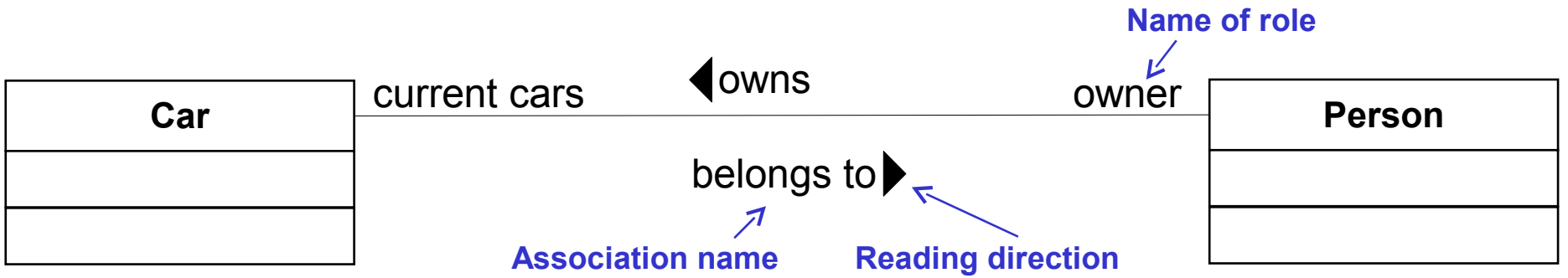
- Classes could have any association with any other class
- Associations could have different cardinalities
- Associations could be directional, showing in which direction to navigate
- A cross indicates, if a navigation should not be possible 
- If there is no cross and no arrow it means that the navigation is not specified (since UML 2.0)

Associations between Classes

- Associations could have role names
  - A role name describes, how a class interprets an associated class
- Associations could be named. The meaning could be clarified with a reading direction visualized with a small filled triangle.
  - Don't mistake the reading direction with the navigation direction in directional relationships

Associations between Classes

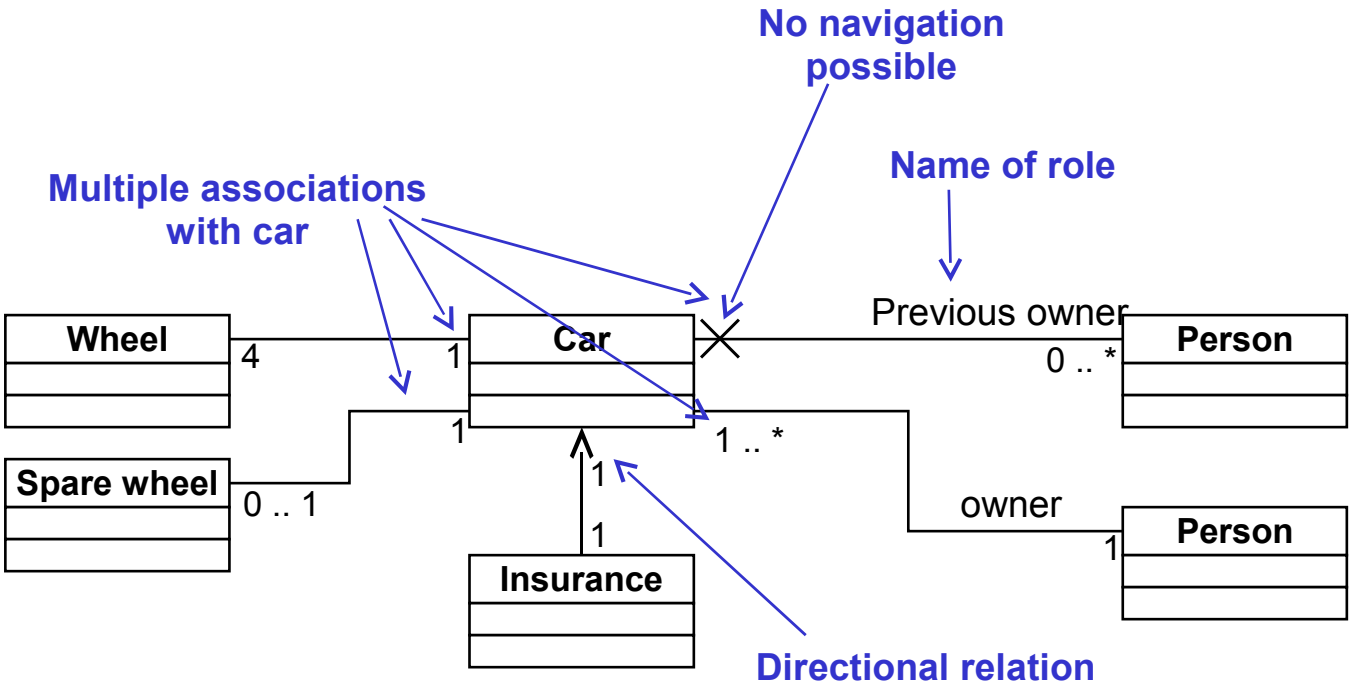
- Example



- A car belongs to a person
- A person owns a car

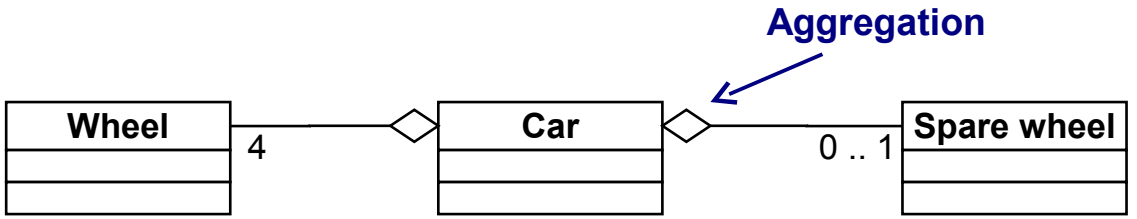
Associations between Classes

- Example



Associations between Classes Aggregation

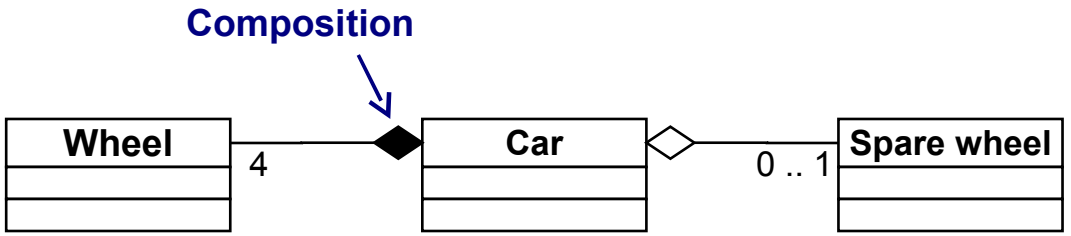
- Aggregation relationship models part- whole-relationship
- Example for aggregation:
  Wheels as parts of a car



A car *has a* wheel
A wheel *is part of* a car

Associations between Classes Composition

- Composition is a strict aggregation: Parts can't exist themselves
- Originally out of C++: The composite is in the physical memory of the other
- Example: A car has four wheels. In this system a wheel could not exist without a car:



- Another example: Account and invoice line item

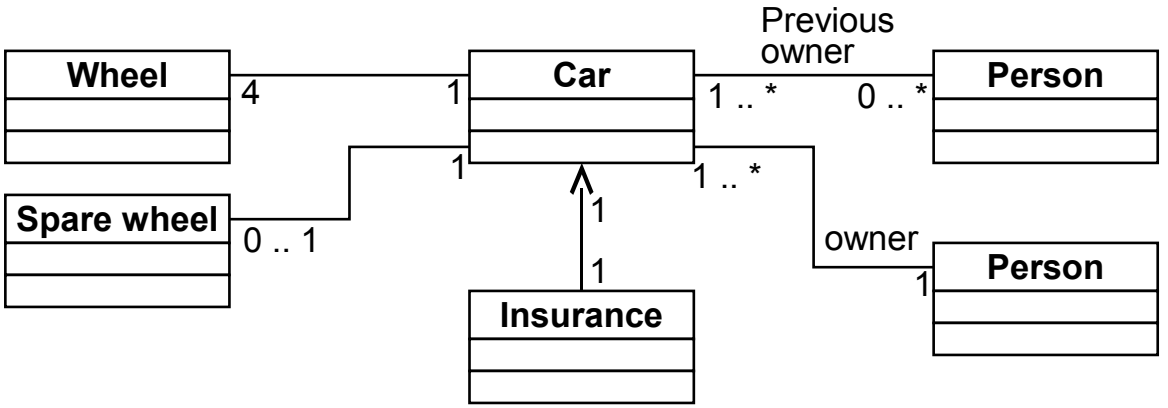
Associations between Classes Coding

- Directional relationship:
  Instance variable in object, where arrow begins.
- ...:1 relationship:
  Instance variable is from type of referring object.
- ...:n relationship:
  Instance variable contents an array or a collection class (e. g. vector)
- 0..n relationship:
  Instance variable may be NULL as well. In a 1..n relationship the variable may not be empty

Associations between Classes Coding

- Aggregation is not a specific element in Java. It should be only considered in programming logic in dependency between objects
- In Java is no difference in implementation between aggregation and composition, different to C++ for instance

Associations between Classes Coding




```
public class Insurance {
   public Car car;
}
public class Person {
  public Car[] currentCars;
  public Car[] formerCars;          /* could be empty */
}
public class Car {
  public Person[] previousOwner;   /* no previous → null */
  public Person owner;
  public SpareWheel spareWheel;    /* max. 1 */
  public Wheel[] wheel;            /* exactly 4 */
}
```

Links

- http://www.agilemodeling.com/essays/umlDiagrams.htm
- http://www.visual-paradigm.com/VPGallery/