Software Techniques – Quo vadis?

Stefan Resmerita, WS2023

- Cost intensive maintenance of software, which is 20 - 30 years old

- Engineering approaches are established at least in sub - domains such as safety critical systems

- The simple, mechanical view is hardly scalable

- Biological systems model → Internet growth by a factor of 100 million

- Development process:
  - Analisys → Design → Implementation

UNIVERSITÄT SALZBURG

Basics of Object-Oriented Analysis and Design

OOAD with UML

UNIFIED
MODELING
LANGUAGE ™

UNIVERSITÄT
SALZBURG

Tools for OO Analysis and Design

OO expectations

- Improved modularity
- Improved reusability
  - Potential for reusable software architectures (= generic complex components) has not been fully investigated so far
- Support for OO modeling is important

What can be expected from OOAD Tools (I)

Great designs come from great designers, not from great tools.

Tools help bad designers create ghastly designs much more quickly.

Grady Booch (1994)

What can be expected from OOAD Tools (II)

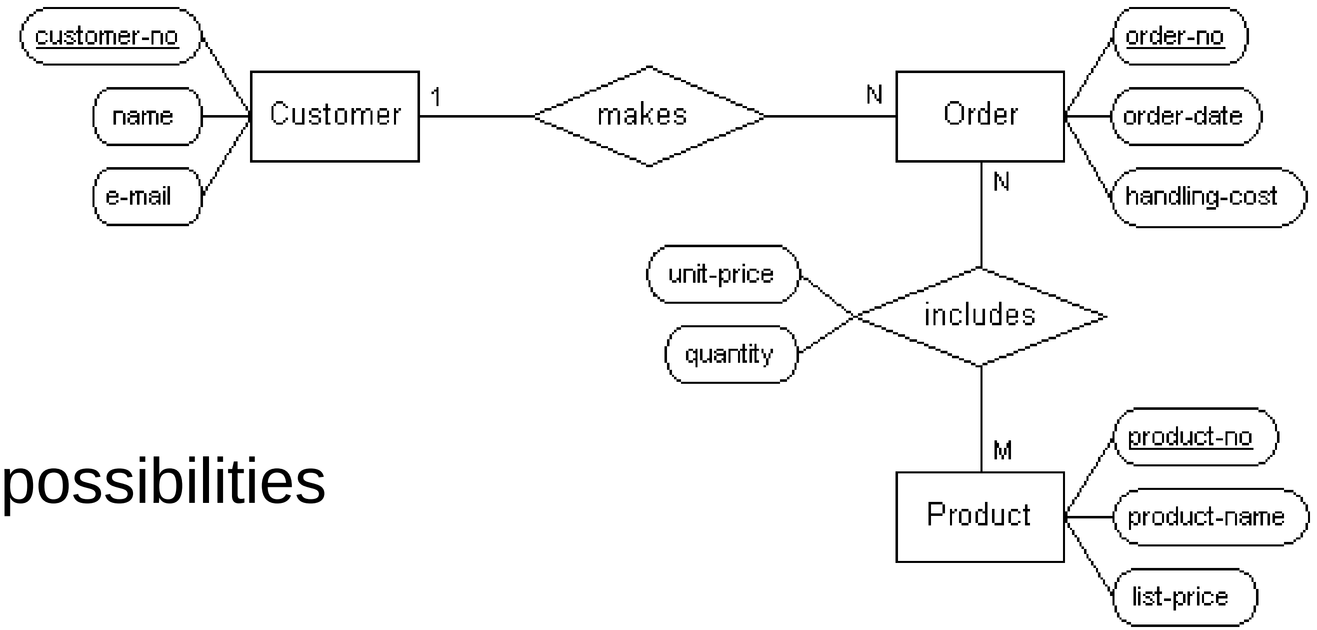
- OOAD tools can :
  - Provide and edit diagrams based on various OO notations
  - Check consistency and constraints
    - Does an object have the called method?
    - Are the invariants (e.g. single instance, etc.) satisfied?
    - ...
  - Completness evaluation
    - Are all the Methods/Classes used?
    - ...

Conventional (SA/SD) versus OO tools (I)

The main differences regard two aspects:

- (1) Software Architecture
  - Conventional tools are based on a separation between data and functions
  - OO tools are based on the grouping of data and functions into meaningful „closed“ objects

Conventional (SA/SD) versus OO tools (II)



- (2) Semantic possibilities

  Relationships in the conventional ER
  - One-to-one (1:1) – has_a, is_a
  - One-to-many (1:m) – owns, contains, is_contained_in
  - Many-to-many (m:n) – consists_of

UNIVERSITÄT SALZBURG

Conventional (SA/SD) versus OO tools (III)

OO modeling needed more comprehensive means of expression

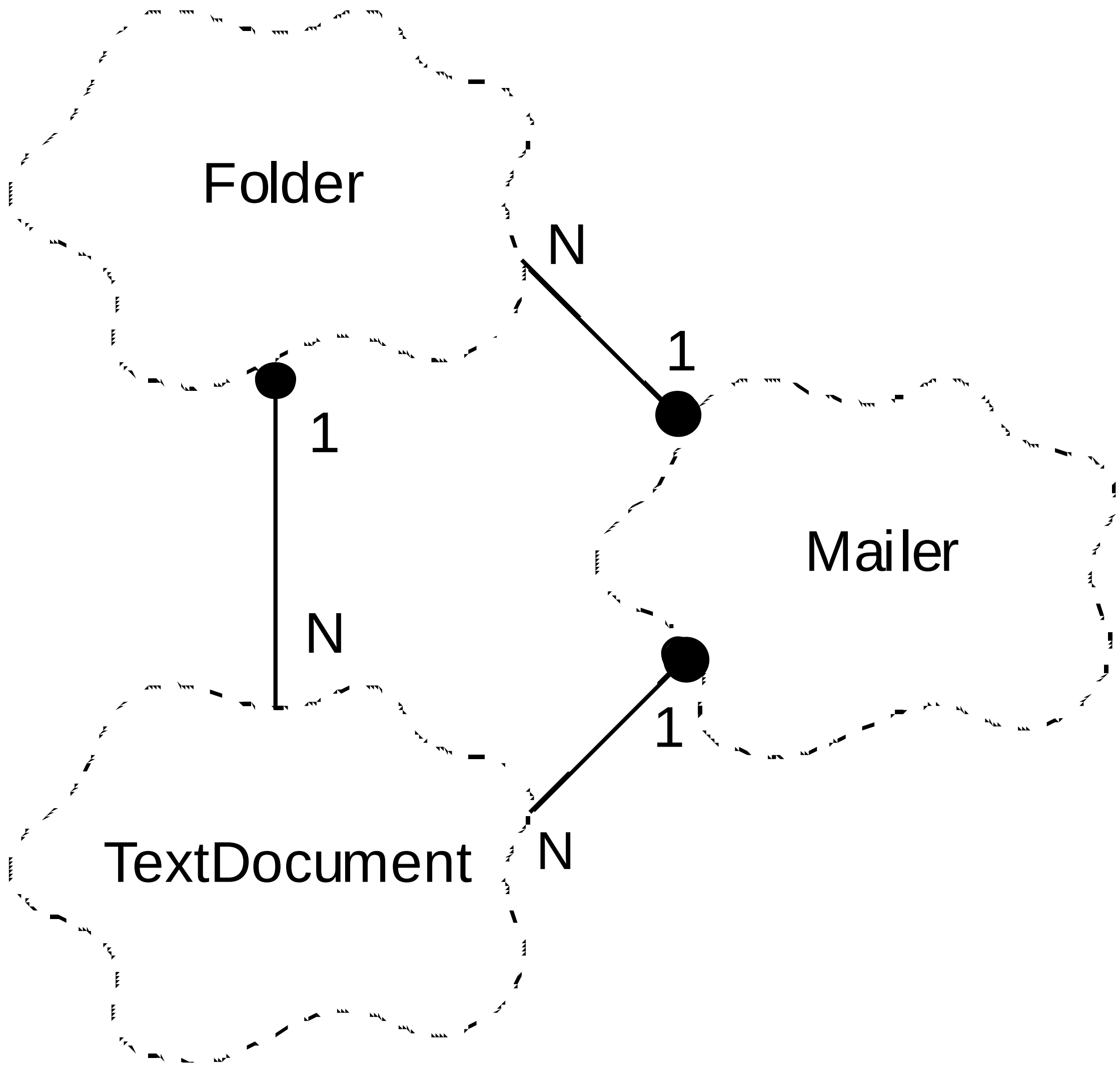
- Class/Object relations and dependencies
  - Inheritance
  - Association
  - Has_a (by value, by reference)
  - Uses_a (by value, by reference)
- Class attributes
  - Is_abstract, is_metaclass
  - Is_parameterized
- Access rights

OO Techniques at the beginning of the 90s

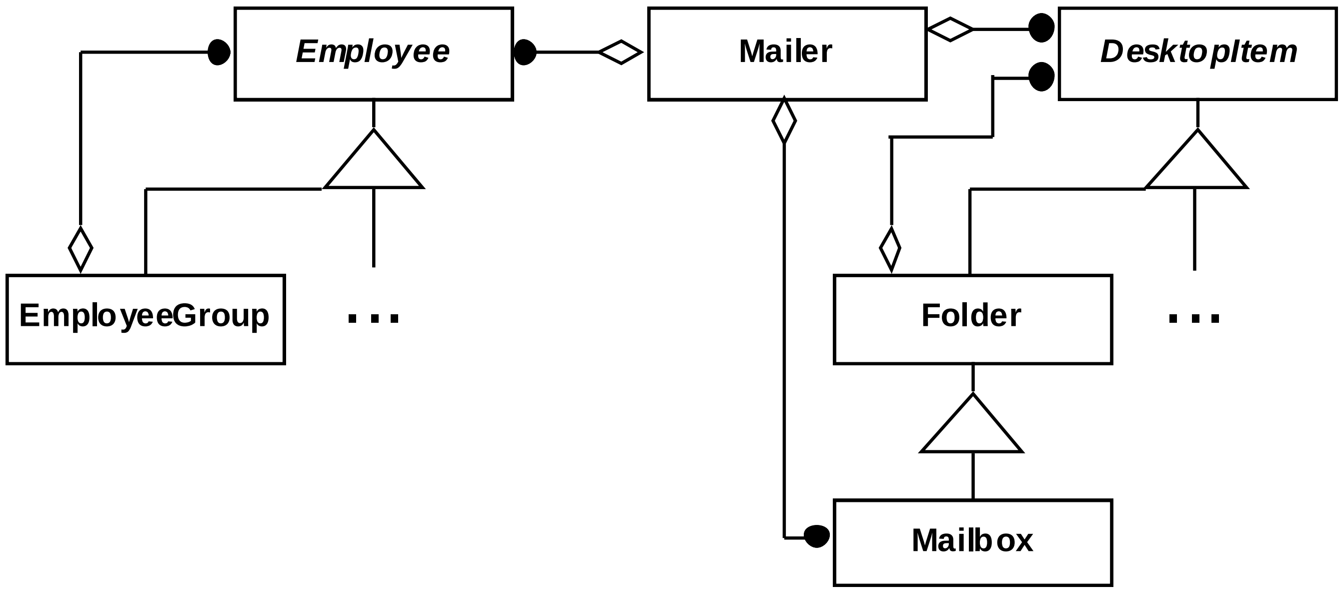
- OOD / Rational Rose
  Grady Booch
- Object Modeling Technique (OMT)
  James Rumbaugh et al.
- OO Software Engineering (OOSE)
  Ivar Jacobson et al.
- OO Analysis (OOA)
  Peter Coad und Ed. Yourdon
- Responsibility-Driven Design (RDD)
  Rebecca Wirfs-Brock et al.
- OO System Analysis (OOSA)
  Sally Shlaer and Steve Mellor
- . . .

UNIVERSITÄT SALZBURG

Example for Booch notation



Example of OMT notation



Common features of OOAD methods (I)

- They aim to represent the physical world without artificial transformations as a software system
  - Application of the same concepts in all phases of software development
  - The border between Analysis and Design became more blurred

- Moreover, very vague usage guidelines were indicated

Common features of OOAD methods (II)

- OOAD methods permit modeling of the following aspects of a system:
  - Static aspects
    - The Class/Object model stands in the foreground
    - Higher abstraction levels are represented by Subsystems
  - Dynamic aspects
    - Interaction diagram
    - State diagram
    - Use case diagram

Differences between OOAD methods

- The differences between the methods used to lie mostly in the notation
- The notations were to a large extent language independent

  => Standardization need was obvious

All of the OO methodologies have much in common and should be contrasted more with non-OO methodologies than with each other.

James Rumbaugh (1991)

UML influences

- The Unified Modeling Language contains various aspects and notations from different methods
  - Booch
    - Harel (State Charts)
  - Rumbaugh (Notation)
  - Jacobson (Use Cases)
  - Wirfs-Brock (Responsibilities)
  - Shlaer-Mellor (Object Life Cycles)
  - Meyer (Pre- und Post-Conditions)

The UML standard

- The first draft (version 0.8) was published in 1995
- Various adjustments and the inclusion of Ivar Jacobson led to version 0.9 in 1996
- Version 1.0 (an then 1.1) was submitted to the Object Management Group (OMG) in 1997 as basis for standardisation
- Version 1.3 came out in 1999
- Version 1.4.2 became an international standard in 2005
- Current OMG standard: version 2.5.1 (December 2017)

The Unified Modeling Language (I)

What is UML?

- Language
  - Communication
  - Exchange of ideas
- Graphical modeling language
  - Drawings, words and rules for representing aspects of software systems

The Unified Modeling Language (II)

What is UML not?

- No method
  - Specifies how models are made but not which and when
  - This is a task of the software development process

Method = Process + Modeling Language

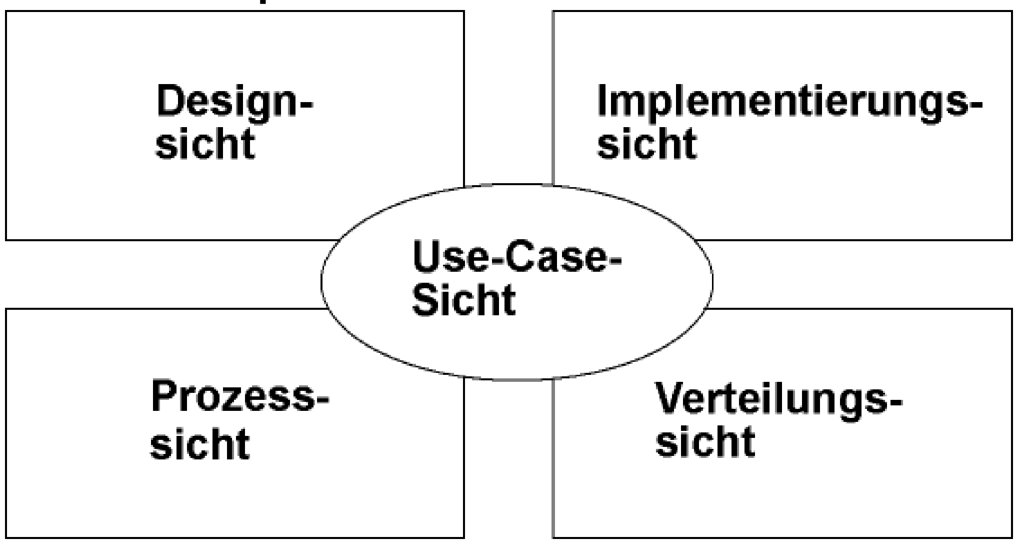
The Unified Modeling Language (III)

Why is UML needed?

- Model as specification
- Model as visualization
- Model for analysis
- Model for design
  - Forward and reverse engineering
- System documentation

The Unified Modeling Language (IV)

- Models
  - Projections of systems on certain aspects
  - Used for understanding
  - Used for specification

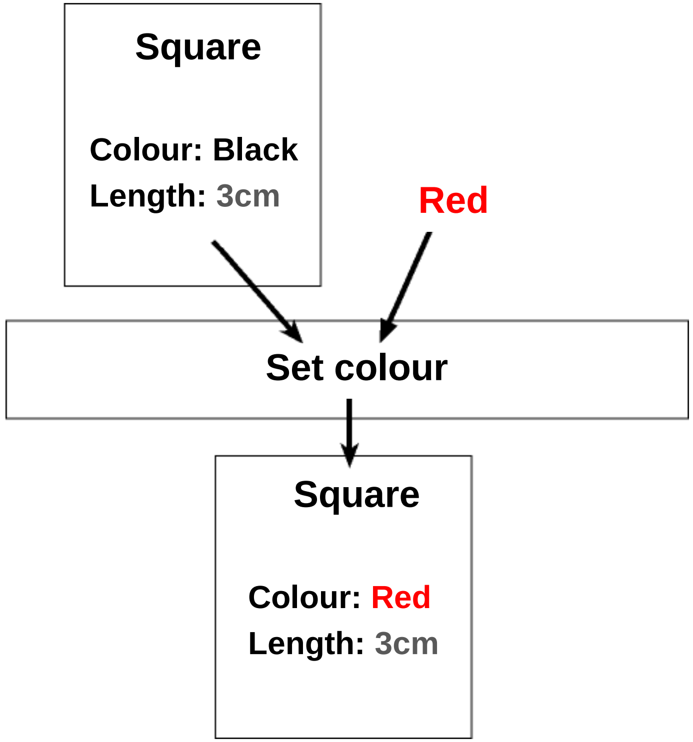


OO concepts
UML representation

- Objects, Classes, Messages/Methods
- Inheritance, Polymorphism, Dynamic Binding
- Abstract Classes, Abstract Coupling

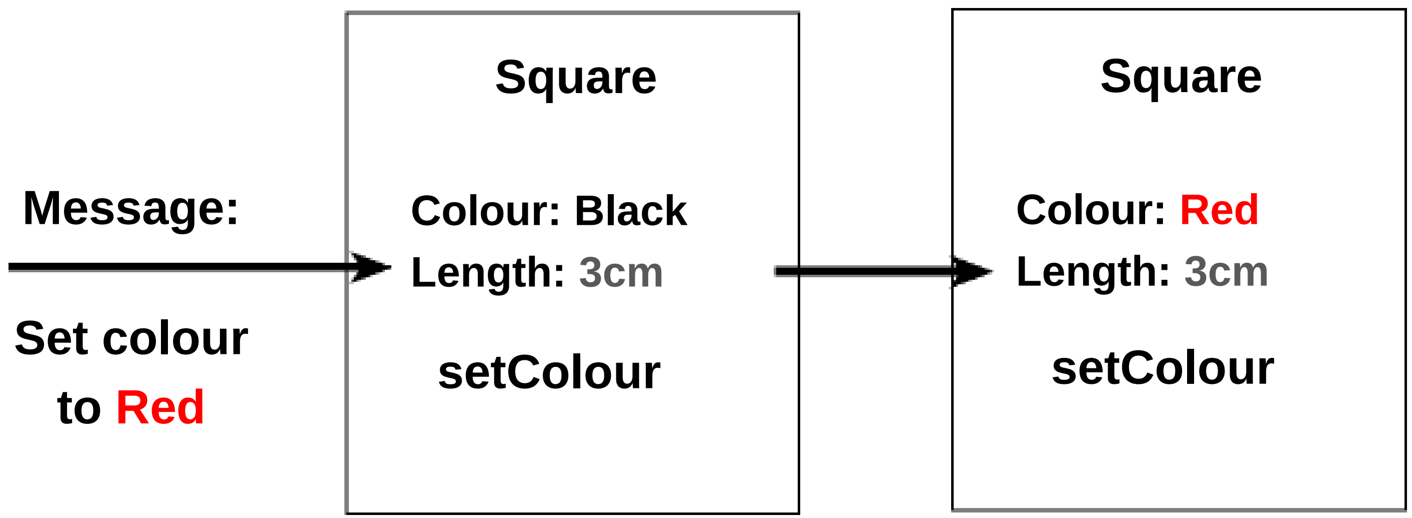
OO versus Procedural (I)

- Procedural: Separation between data and procedures



OO versus Procedural (II)

- Object-oriented: Data and procedures form a logical unit → an Object



UNIVERSITÄT SALZBURG

Objects(I)

An object is a representation of

- A physical entity
  - E.g. Person, Car, etc.
- A logical entity, e.g.:
  - Chemical process,
  - Mathematical formula,
  - etc.

Objects (II)

The main characteristics of an object are:

Its identity

Its state

Its behavior

Objects (III)

- State

  The state of an object consists of its static attributes and their dynamic values

  - Values can be primitive: int, double, boolean

  - Values can be references to other objects, or they can be other objects

Objects(IV)

- Example
  - Drinks machine

    Pay

    1) Ready
    2) Busy
    3) Ready

    Remove drink

  - Attributes – values
    - Paid: boolean
    - Cans: number of cans

Objects(V)

- The behavior of an object is specified by its **methods** (=operations)

- In principle, methods are conceptually equivalent to procedures/functions:

Methods = Name + Parameters + Return values

Objects(VI)

- Example
  - Rectangle
    - Name of the operation: `setColor`
    - Parameter: name of the color (e.g. `Red`)
    - Return values: none
- Calling an operation of an object is reffered to as sending a message to the object

Objects(VII)

- Identity

  The identity of an object is the characteristic that differentiates the object from all the other objects

- Two objects can be different even if their attributes, values and methods coincide

Object – Orientation

- Classification
  - Object grouping
- Polymorphism
  - Static and dynamic types
  - Dynamic binding
- Inheritance
  - Type hierarchy

Classification

- Class

  A class represents a set of objects that have the same structure and the same behavior

  A class is a template from which objects can be instantiated

Classification Example

- Class Person
  - Attributes:
    - Name: String
    - Age: int
  - Operations:
    - eat, sleep, ...

- Object of type Person: Oliver
  - Attributes:
    - Name: Oliver
    - Age: 24

Class as a template/type (I)

- Comparison with C

```
struct{
        int day, month, year;
   } date;
   date d1, d2;
```

⇒ All are accessible

⇒ There is no method

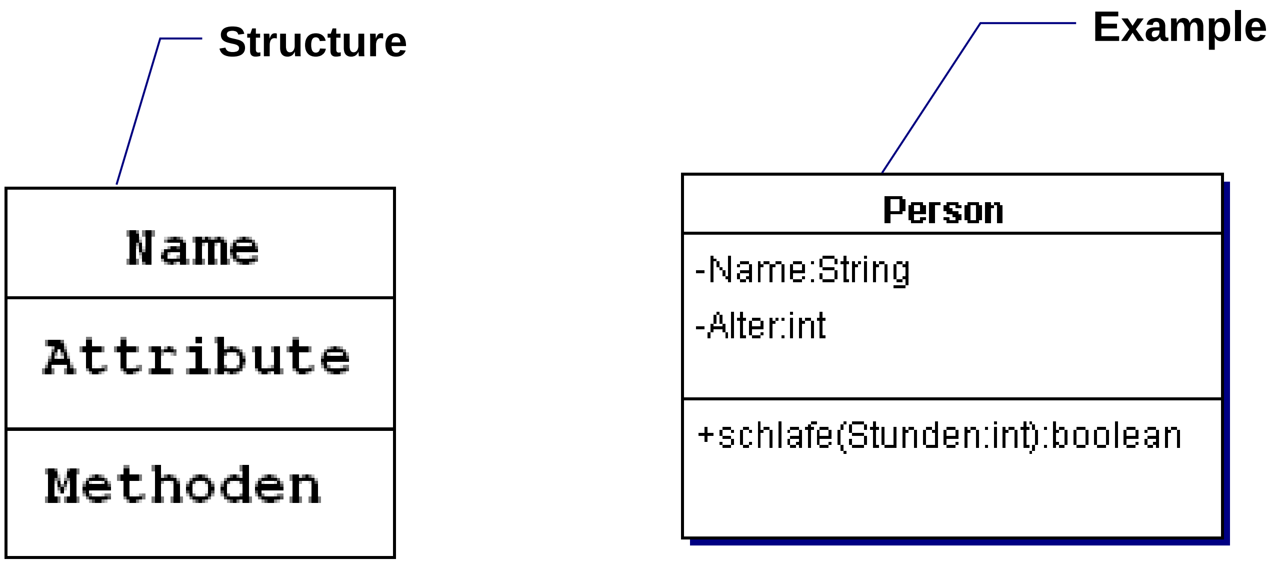
Class as a template/type (II)

A class indicates which type an object has, i.e., which messages it understands and which attributes it has.

- A class consists of
  - A unique name
  - Attributes and their types
  - Methods/Operations

Classes in UML (I)

- UML notation for a class:



UNIVERSITÄT SALZBURG

Classes in UML (II)

Notation for attributes:

A	only the attribute‘s name
: C	only the attribute‘s class
A : C	attribute‘s name and class
A : C = D	attribute‘s default value

`timeWhenStarted`	→ A
`: Date`	→ : C
`timeWhenStarted : Date`	→ A : C
`timeWhenStarted : Date = 1.1.1999`	→ A : C = D
`timeWhenStarted = 1.1.1999`	→ A = D

Classes in UML (III)

Notation for Methods/Operations:

`m()`	only the method name
`m(arguments):R`	method name, arguments
	type of returning parameter

Example:

`printInvoice()`	→ m()
`printInvoice(itemNo: int):bool`	→ m(arguments): R

Classes in UML (IV)

- Adornments (decorations) : additional graphical elements (represented by triangles in the Booch method)

- Methods and attributes have attached graphic symbols to express access rights: **`public, private, protected`**

  Example:

  ```
  +sleep(Hours:int)
  ```

- Standalone adornment: Note

Classes in Java

```
public class Person{

  String name;
  int age;
  ...

  public int getAge(){
    return age;
  }
  public void setAge(int theAge){
    age = theAge;
  }
}
```

Class name

Attributes

Operations

Using classes in Java

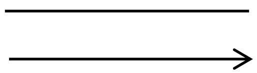
- Classes are used in Java to specify the type of variables and to instantiate objects
- Keyword: new
- Example:

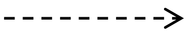
```
Person manager = new Person("Martin");
```

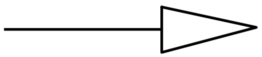
Declaration of variable „manager"

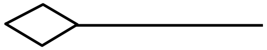
Instantiation of an object of class Person with name Martin

Class relationships (I)


Association


Dependence

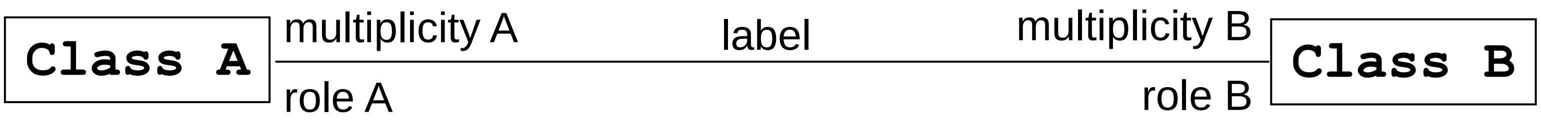

Inheritance


Aggregation (has-a)

An association can be refined by other relations

Often one models first only the fact that two classes are related and refines later this general notation element

Class relationships (II)



- Each association can be named with a text label (like in the ER-model)
- Role names can be specified at association ends
- Multiplicity can be marked at association ends
- A class can have an association with itself, expressing a relationship between objects of the same class

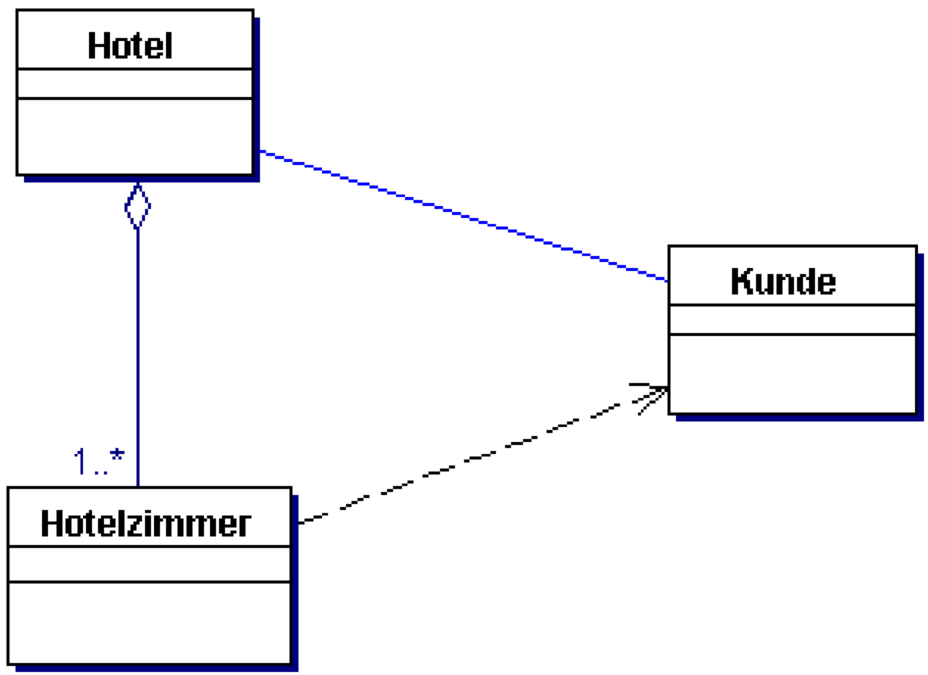
Class relationships (III)

Multiplicity specification:

1	exactly one
*	any (0 or more)
0..*	any (0 or more)
1..*	1 or more
0..1	0 or 1
2..5	range of values
1..5, 9	range of values or nine

Class relationships (IV)

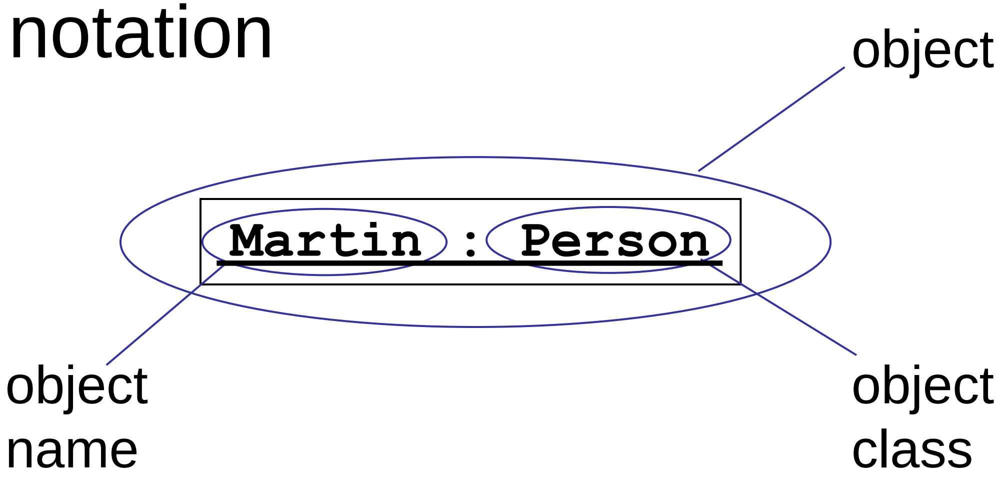
Example:



UNIVERSITÄT SALZBURG

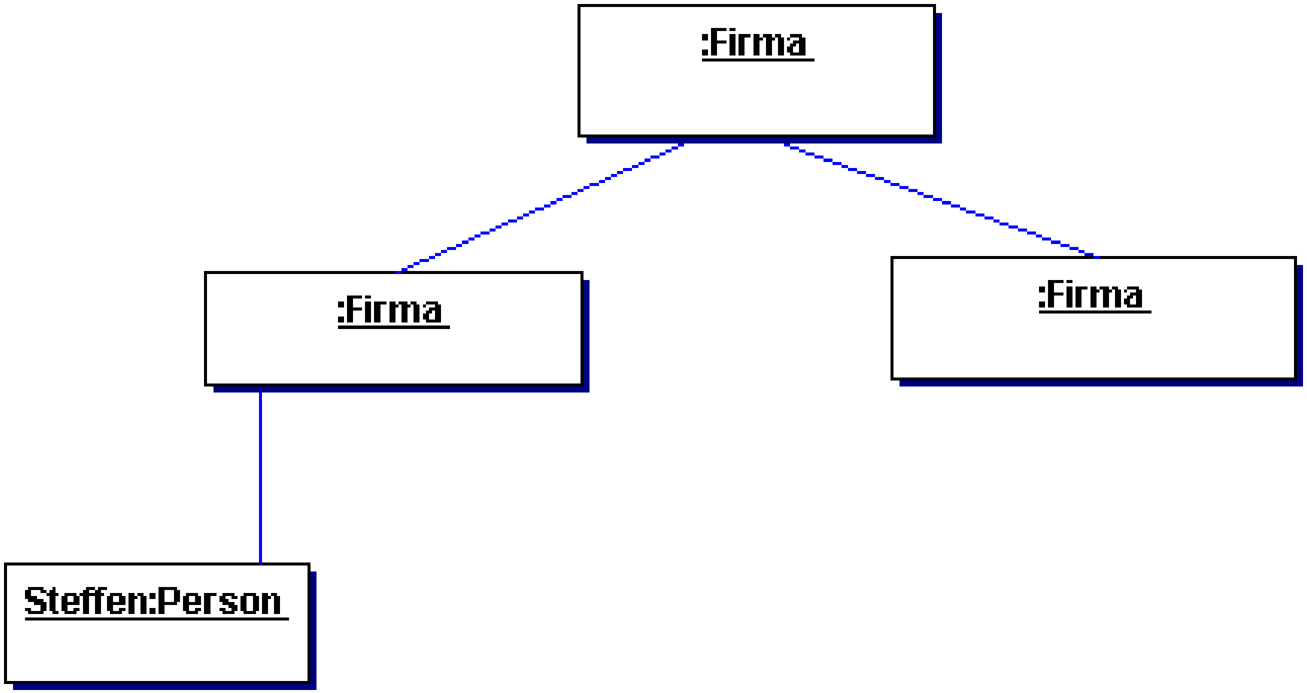
Objects in UML

- Object notation



An object diagram provides a run time snapshot of the system, representing objects and the connections between them

Object diagram



Inheritance

Polymorphism

Dynamic Binding

Inheritance (I)

- A class defines the type of an object

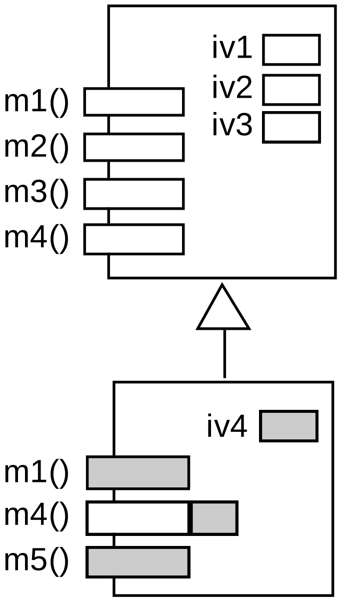
- If one models for example a class **Customer** and a class **CorporateCustomer**, one expects that each object of type **CorporateCustomer** to be also of type **Customer**. The type **CorporateCustomer** is a *subtype* of **Customer**.

Inheritance (II)

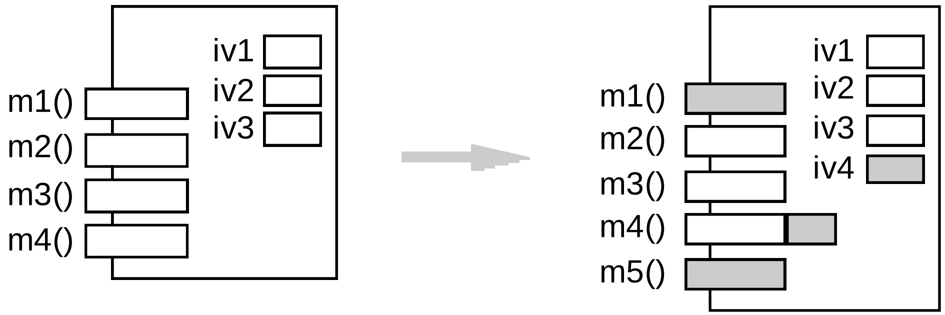
- A superclass generalizes a subclass
- A subclass specializes a superclass
- A subclass **inherits** methods and attributes of its superclass

Inheritance(III)

- A subclass has the following possibilities to specialize its behavior:
  - Defining new operations and attributes
  - Modifying existing operations (overriding methods of the superclass)

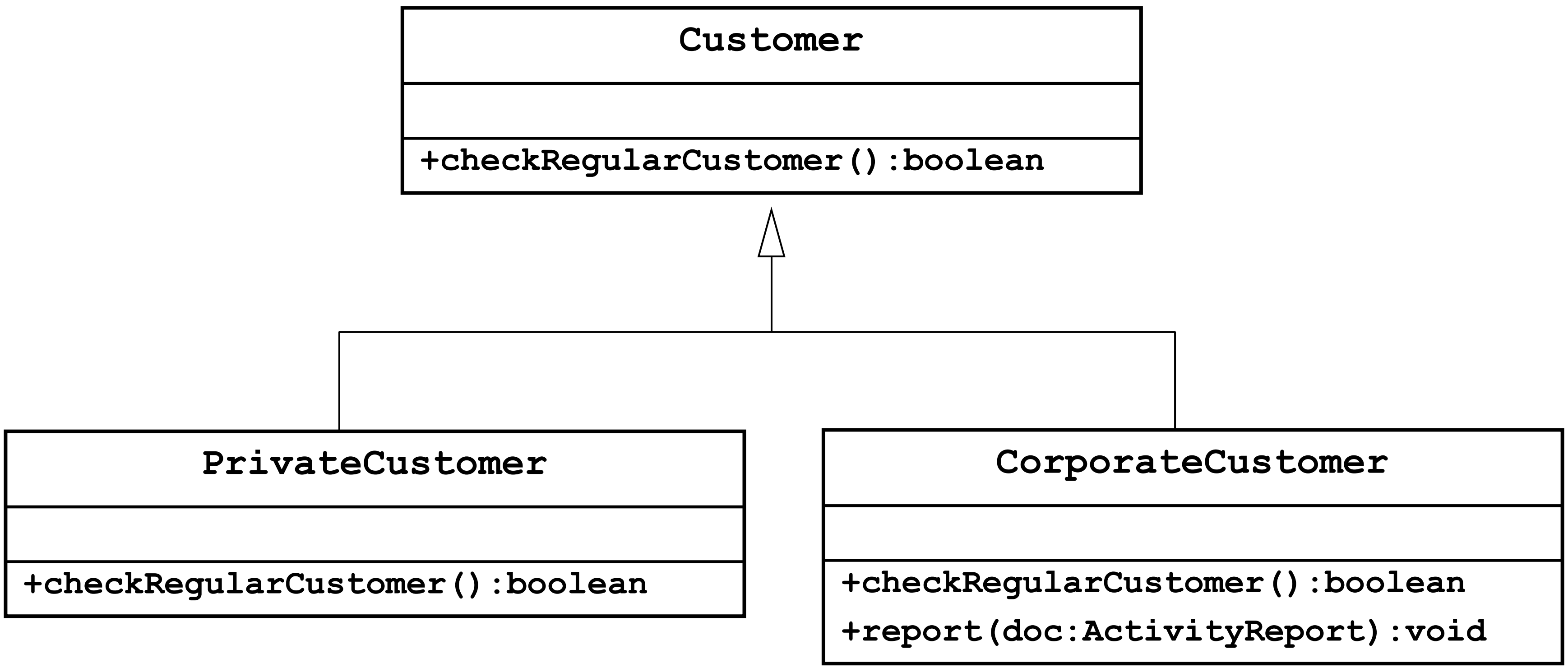


Flatten view:



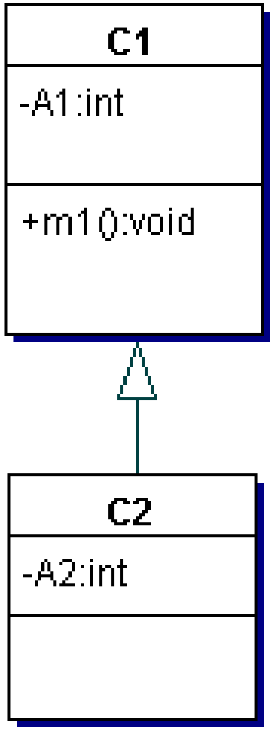
Inheritance (IV)

- UML Notation



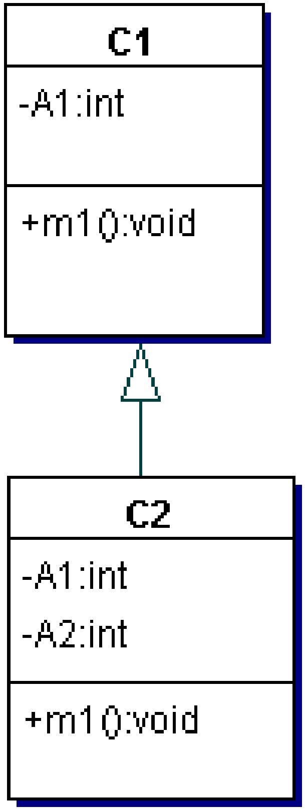
Inheritance (V)

„delta“ view



Flat view

(not in standard UML!)



Inheritance and access rights

- Private members of a superclass are **not accessible** in subclasses
- Protected members of a superclass are accessible **only** in subclasses
- Public members are accessible **everywhere**
- Access rights can be specified globally for a superclass (C++):

  class R : *private A*{ /* ... */ };

  class S : *protected A*{ /* ... */ };

  class T : *public A*{ /* ... */ };

Inheritance in Java

- Java supports single inheritance, where each class has at most one superclass

- The keyword is **`extends`**

Example:

```
public class CorporateCustomer extends Customer{
    ...
}
```

Inheritance in C++

```
class Base {
  protected: int i;
};

class Derived_1 : private Base {
  int f(Base* b) { return b->i; }
  int g(Derived_1* d) { return d->i; }
};

class Derived_2 : public Base {
  int f(Base* b) { return b->i; }
  int g(Derived_1* d) { return d->i; }
  int f1() { return i; }
};
```

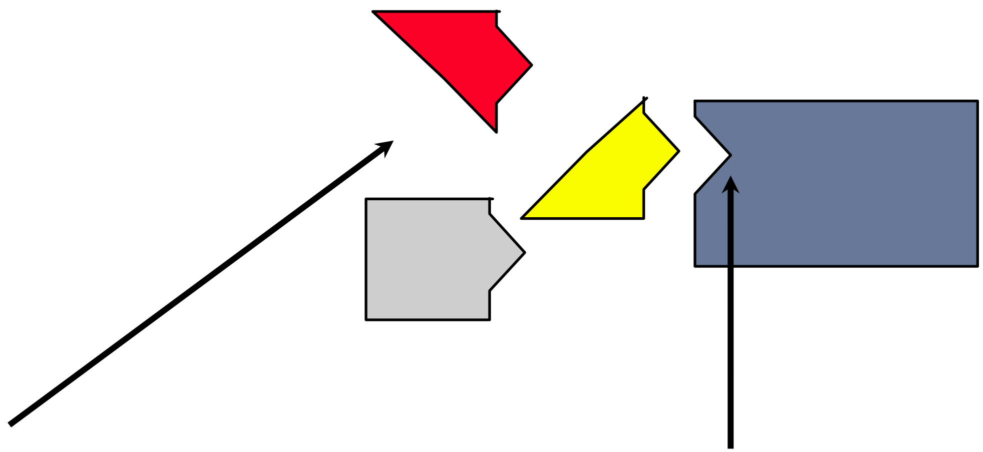
Inheritance

Polymorphism

Dynamic Binding

Polymorphism (I)

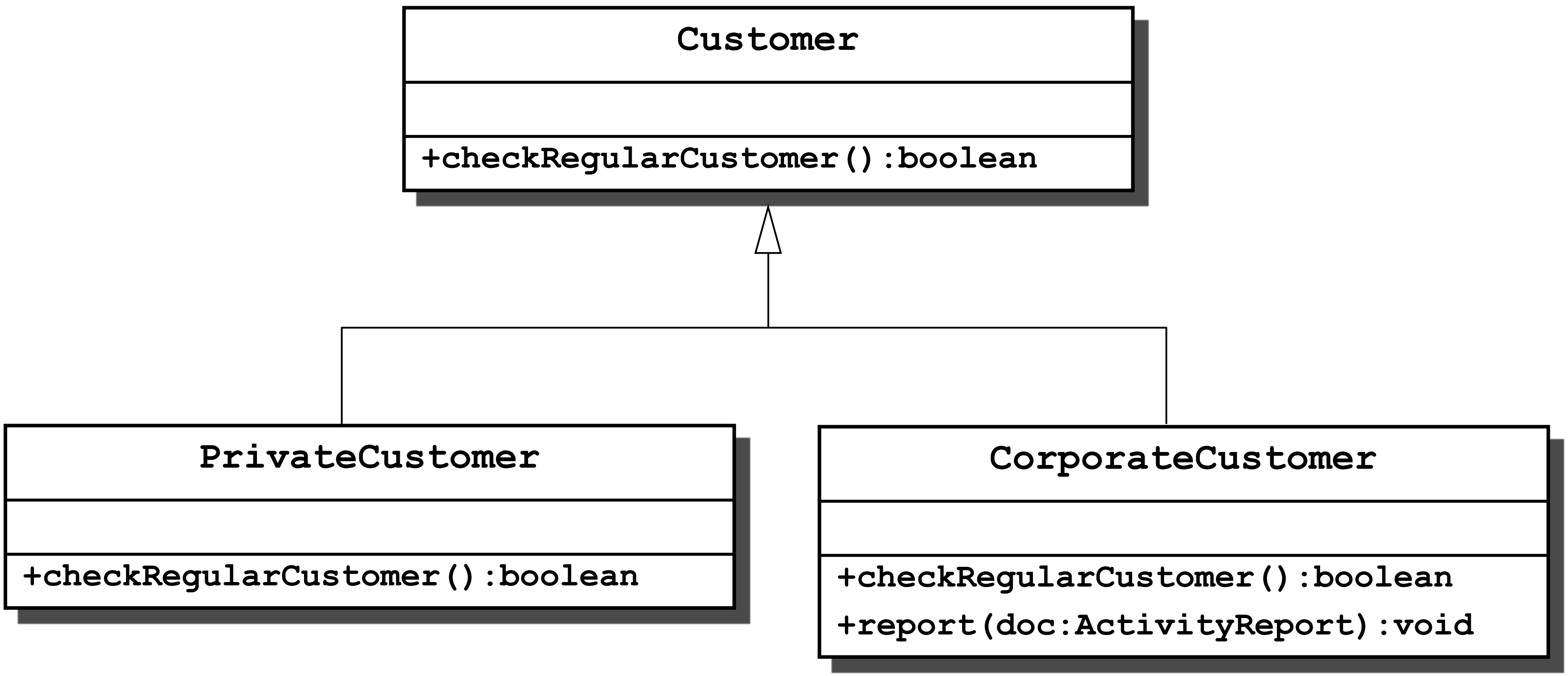
- An object type can be poly (=multiple) morph (=form). This can be depicted in the same way as plug-compatibility:



Objects compatible with the plug

„Plug“-Standard

Inheritance example revisited

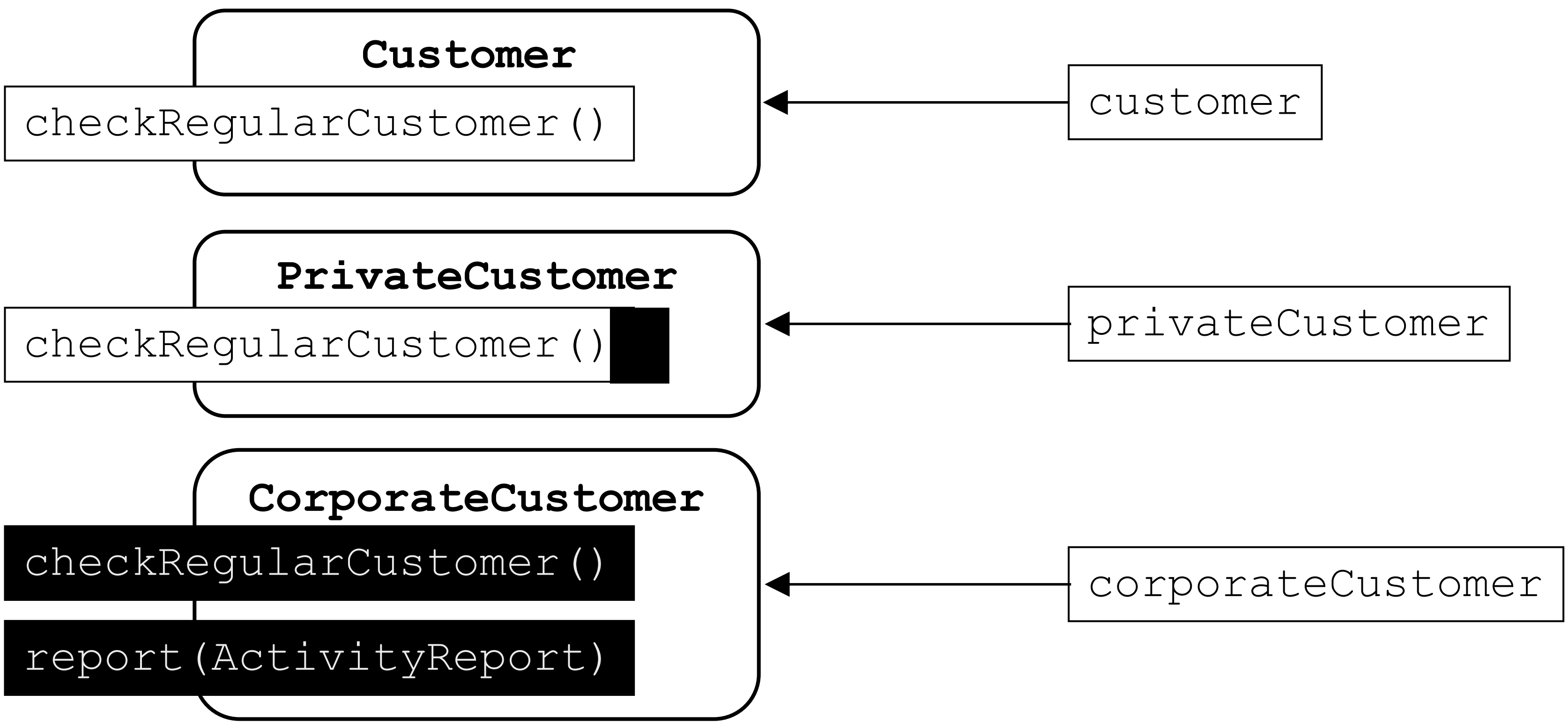


Polymorphism (II)

- Objects of type **CorporateCustomer** (subclass) keep at least the same contract as objects of type **Customer** (superclass).
- Therefore it is meaningful to consider that an object of class A_i, which is a subclass of class A, **is not only of type** A_i but also of the types given by all A_i's superclasses (starting with A).
- **An object has not only one type. It has multiple types**, and the number of types is given by the position of the class from which the object is generated in the class hierarchy.

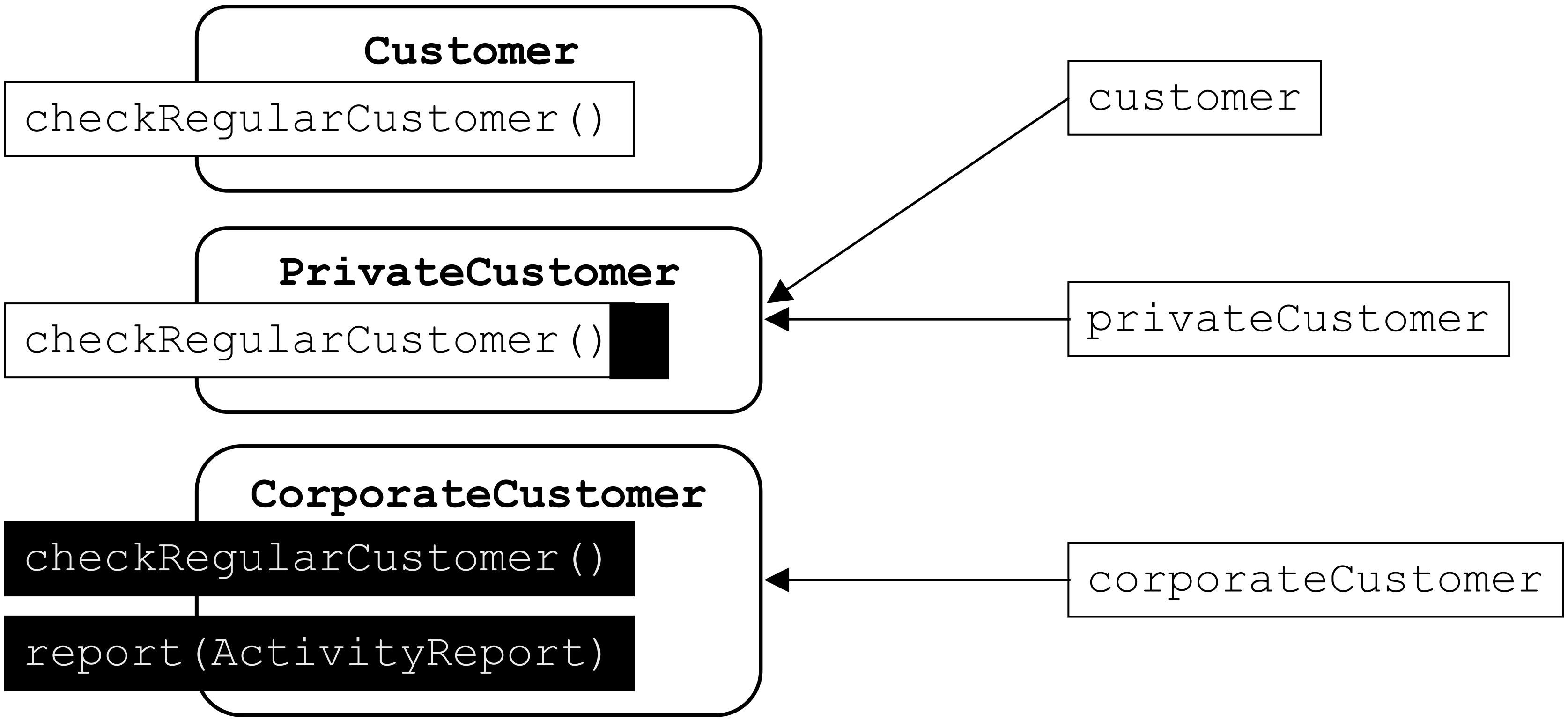
Polymorphism – Example (I)

```
Customer customer = new Customer();
PrivateCustomer privateCustomer = new PrivateCustomer();
CorporateCustomer corporateCustomer= new CorporateCustomer();
```



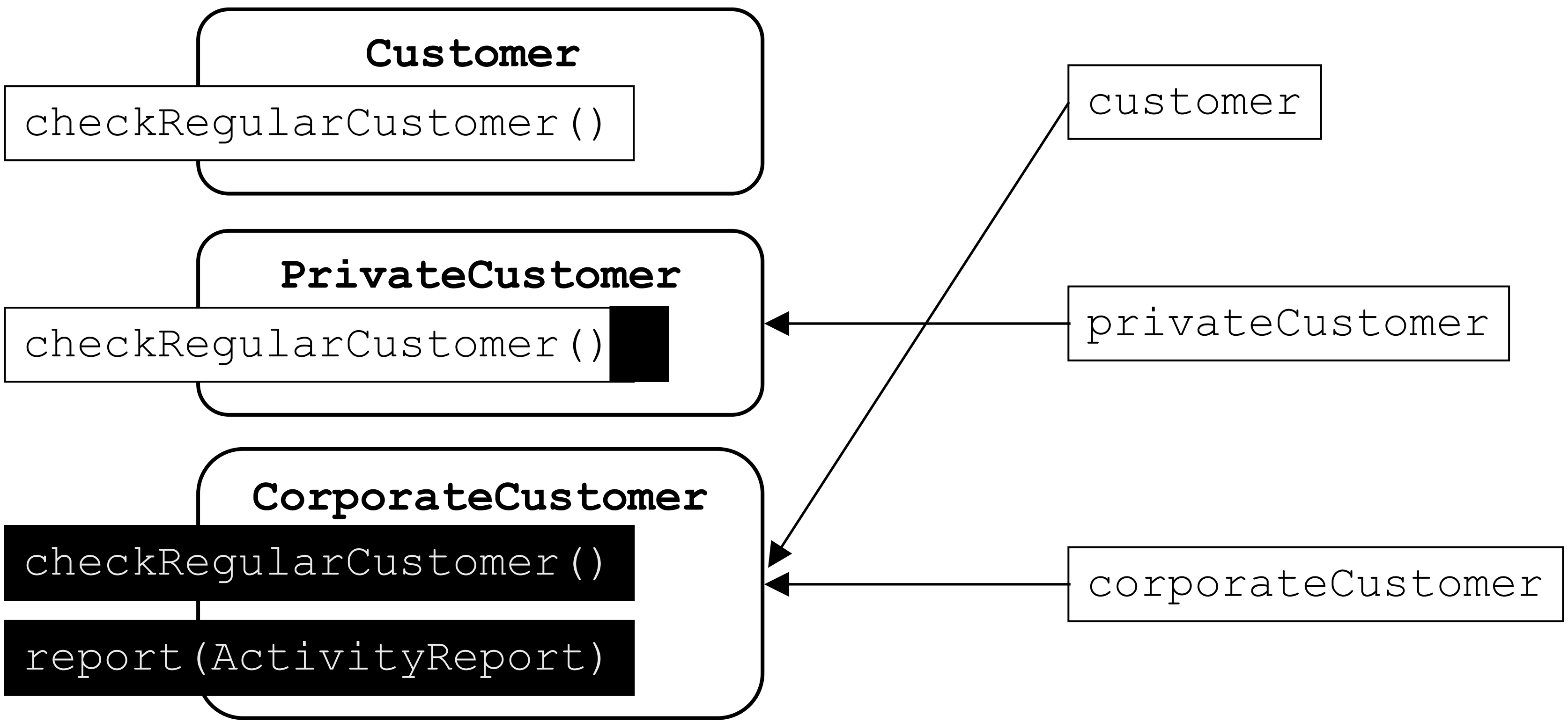
Polymorphism – Example (II)

customer = privateCustomer;   // OK



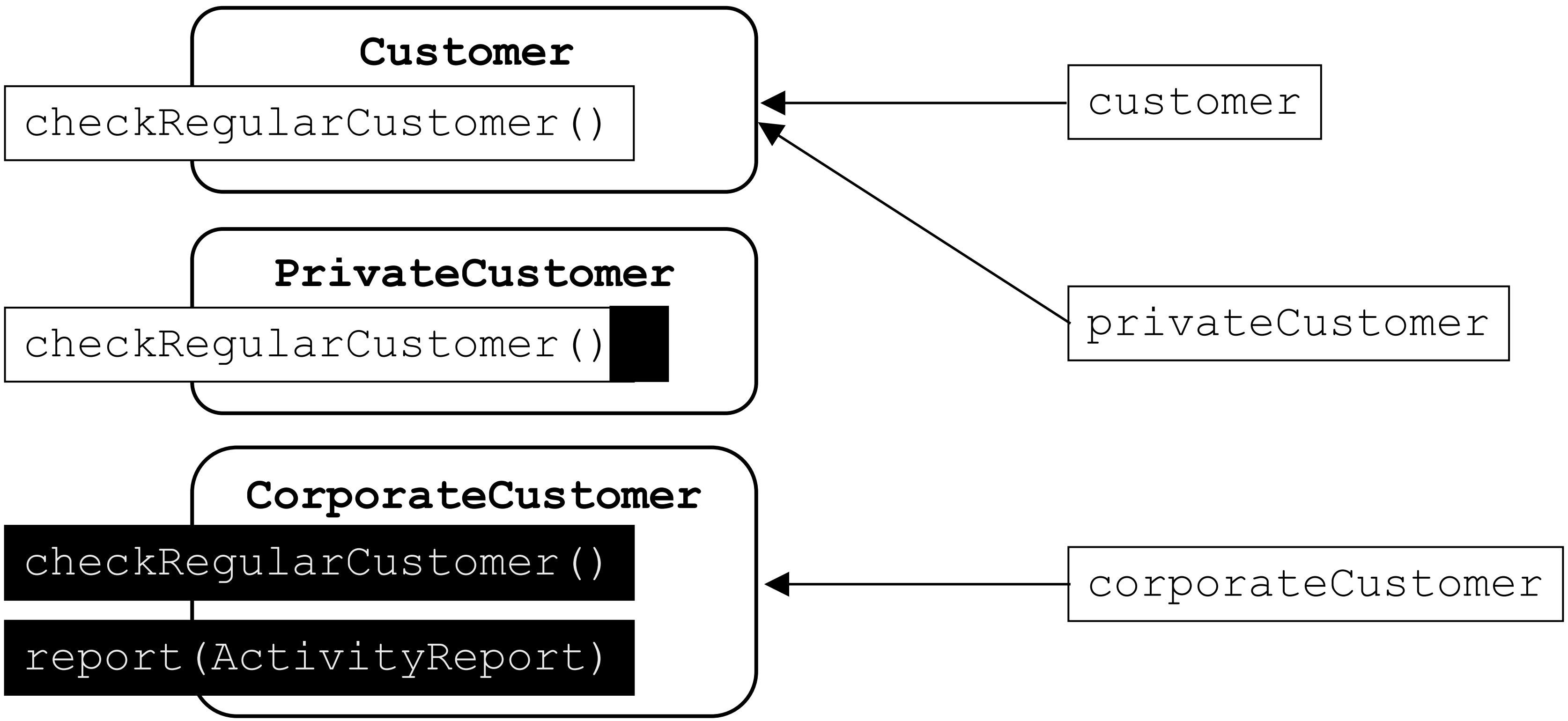
Polymorphism – Example (III)

customer = corporateCustomer;   // OK



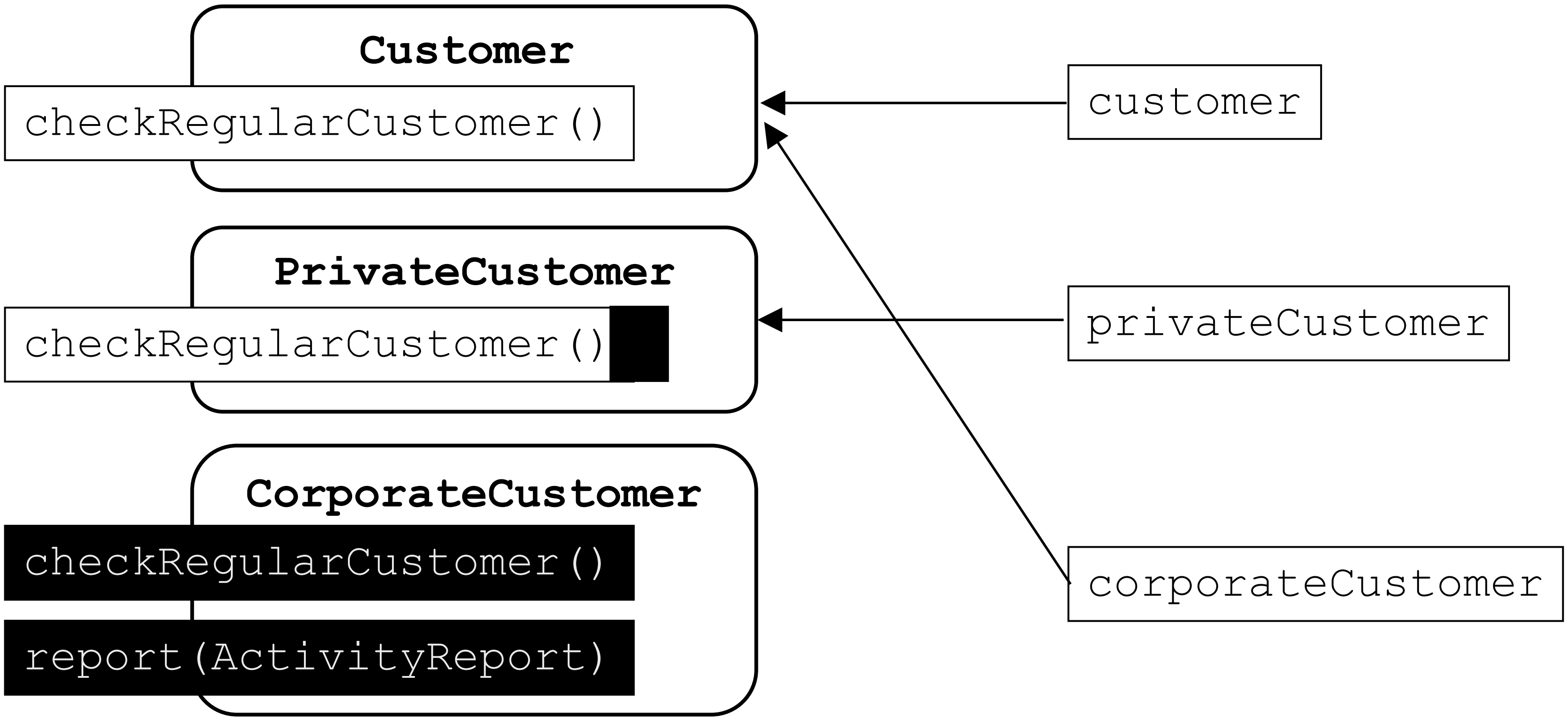
Polymorphism – Example (IV)

privateCustomer = customer;   // wrong



Polymorphism – Example (V)

corporateCustomer = customer;   // wrong



Polymorphism – Example (VI)

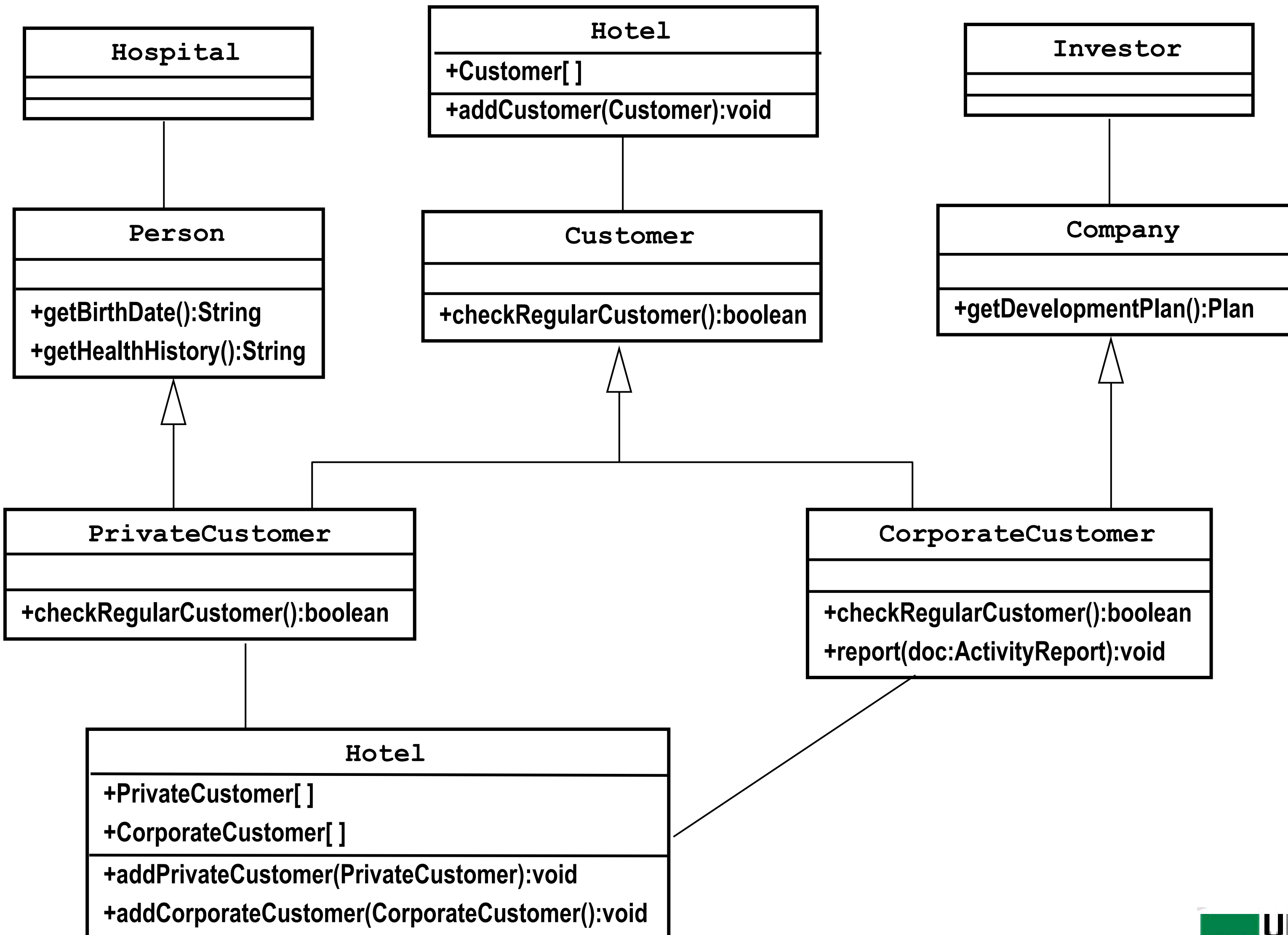
- The reason for failure is that an object which is an instance of class `Customer` does not understand all method calls that an object which is an instance of class `CorporateCustomer` understands.

```
(1) corporateCustomer = customer;
(2) corporateCustomer.report(monthlyReport);
```

(1) Type mismatch: cannot convert from `CorporateCustomer` to `Customer`

(2) The method `report(activityReport)` is undefined for the type `Customer`.

Polymorphism – Example (VII)



Static and dynamic type

- Static type
  - Accurately given by the declaration in the program text
  - Example: **`customer`** is of static type **`Customer`**

- Dynamic type
  - The type of the referenced object at runtime
  - Example: after the assignment **`customer=corporateCustomer`**, the dynamic type of **`customer`** is **`CorporateCustomer`**

- A variable with a static type can have several dynamic types during its lifetime, depending of the width and depth of the class hierarchy

Dynamic binding (I)

Dynamic binding: The compiler **does not specify which method is called at runtime .** The method is determined at runtime based on

- The method name
- The variable‘s dynamic type

```
Customer c;
if (i > 0) then
      c = new CorporateCustomer();
else
      c = new PrivateCustomer();
...
c.checkRegularCustomer();
```

Dynamic binding (II)

When `(i > 0)` is true, the variable `c` references an object generated from the class `CorporateCustomer` (and thus has the dynamic type `CorporateCustomer`). Hence, the call to `checkRegularCustomer()` is linked to the method as implemented in `CorporateCustomer`.

- In Java, all methods are dynamically bound, except for the ones explicitly marked by using the keyword `static`.

- In C++, by contrast, methods must be explicitly marked as dynamically bound by using the keyword `virtual`.