

3.3 Class Diagrams in Design

Subject/Topic/Focus:

- Introduction to Class Diagrams in Design

Summary:

- Modeling Views
- Compartments, Properties, Multiplicity, Attributes, Operations
- Abstract Classes, Interfaces
- Associations
- Templates

Literature:

- [Fowler99]
- [Booch98]

Three Views on OO-Modeling

Conceptual view **(early analysis)**

- classes represent the application concepts
- language and system independent
- few classes, few diagrams

Design view **(late analysis, early design)**

- classes represent SW-interfaces
- describes the solution of difficult problems of the implementation (risk minimization)
- structures the system in layers, subsystems and packages

Implementation view **(design and programming)**

- classes represent the code in a programming language
- is directly mapped on the implementation
- example: mapping of associations

Views: Hints

- Views are not a part of UML but of the *Unified Process*.
- Views are important for the modeling and model revision.
- Each diagram must be assigned uniquely to exactly one view.
- Reading the diagrams, the view of the drawing must be known.
- The implementation view is often used, but the other ones are at least as important!



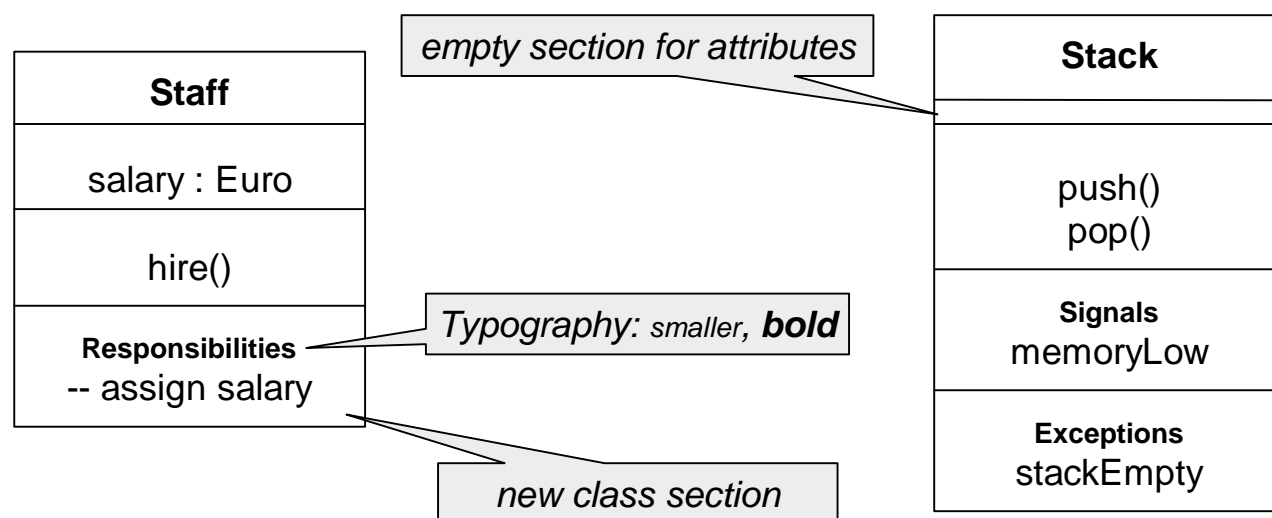
Design: Activities and Concepts

Activities and Goals	Relevant Concepts
1) In the class design refine conceptual classes and identify new classes. <ul style="list-style-type: none"> • define contracts 	<input type="checkbox"/> attributes <input type="checkbox"/> operations <input type="checkbox"/> multiplicity
2) Design relations between classes <ul style="list-style-type: none"> <input type="checkbox"/> Design of interfaces <ul style="list-style-type: none"> • minimize dependencies, „lean“ interfaces • reuse code 	<input type="checkbox"/> abstract classes <input type="checkbox"/> dependencies <input type="checkbox"/> realization <input type="checkbox"/> parameterization, instantiation
<input type="checkbox"/> Refine the conceptual semantic relations and identify new relations.	<input type="checkbox"/> associations <input type="checkbox"/> aggregation <input type="checkbox"/> generalization

Compartments

A **compartment** is a section in the graphical notation of classes, in which parts of the class belonging together are noted. Standard sections for classes are the ones for attributes and operations. Beside these further class parts can be defined, e.g., for signals, responsibilities or exceptions.

Notation:



Class Diagrams in Design

3.3.5

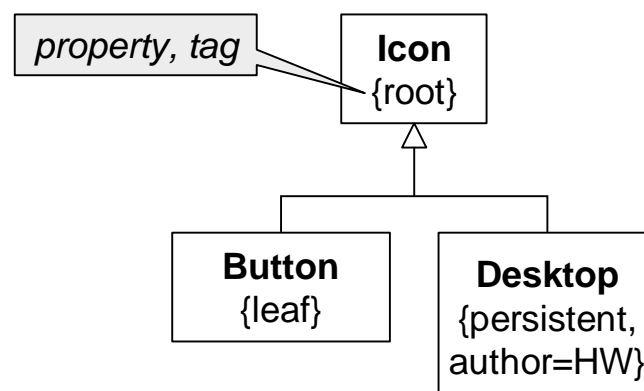
Classes: Property, Tag

A **property / tag** specifies an (implementation defined) property of a class or an operation.

Examples:

- ❑ *root*: starting point of a class tree resp. a class hierarchy.
- ❑ *leaf*: class from which no other class is allowed to inherit.
- ❑ *persistent*: instances of this class are durable, i.e., live longer than the application.

Notation:



Class Diagrams in Design

3.3.6

Attributes (1)

- An **attribute** is a property of an object or class.
- An attribute has no independent existence and can not be part of a relation independent from its object or class. So an attribute is not an object.
- In an object-oriented programming language an attribute of an object is represented by a data field and an attribute of a class by a class attribute (*static* in Java).
- The set of the data fields available for an object and their signatures are defined by the class of the object. The class also defines the set of class attributes and their signatures.
- Attributes used for the identification of objects are called **keys**.
- The value of the attribute (the attribute instance) is an individual property of an object (instance). The values of all attributes of an object together define its state completely. So an object aggregates the values of its attributes. (Always it owns beyond its state an state-independent object identity and a reference to its class.)
- The name of the attribute is a value selector using the dot notation (o.name, o.age).

Attributes (2)

Characteristics:

- Changeability:** The value can be changeable, addOnly (for sets) or frozen.
- Initial value:** Value or expression evaluating to an initial value.
- Multiplicity:** scalar value, optional, value set
- Name:** name of the attribute
- Scope:** instance attribute or class attribute
- Type:** type of the attribute: name of an interface or class
- Visibility:** of the attribute outside of its name space (the name space is the class)

Attributes: Notation

Syntax:

```
[“«stereotype»”] [visibility] name [“multiplicity”] [“:” type] [“=” initvalue] [“{” “properties” “}”]
```

Visibility:

- ☐ + (*public*) Everyone accessing the class may access the attribute.
- ☐ # (*protected*) Only (transitive) subclasses of the class may access the attribute.
- ☐ - (*private*) Nobody outside the class may access the attribute.

Multiplicity: like for associations, e.g., [3], [2..*]

Properties:

- ☐ Syntax: [“{” tag “=” value “}”]
- ☐ Examples: {changeable,persistent}, {author=Holm Wegner, request=/LE-37/}

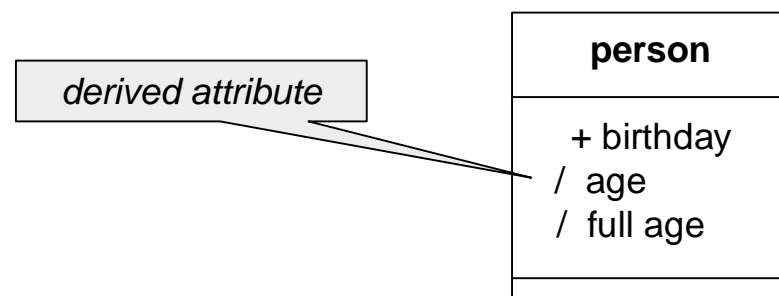
Scope: underlined = class attribute, otherwise instance attribute.

Derived Attributes

A **derived attribute** is an attribute whose value can be derived from an expression on other attributes.

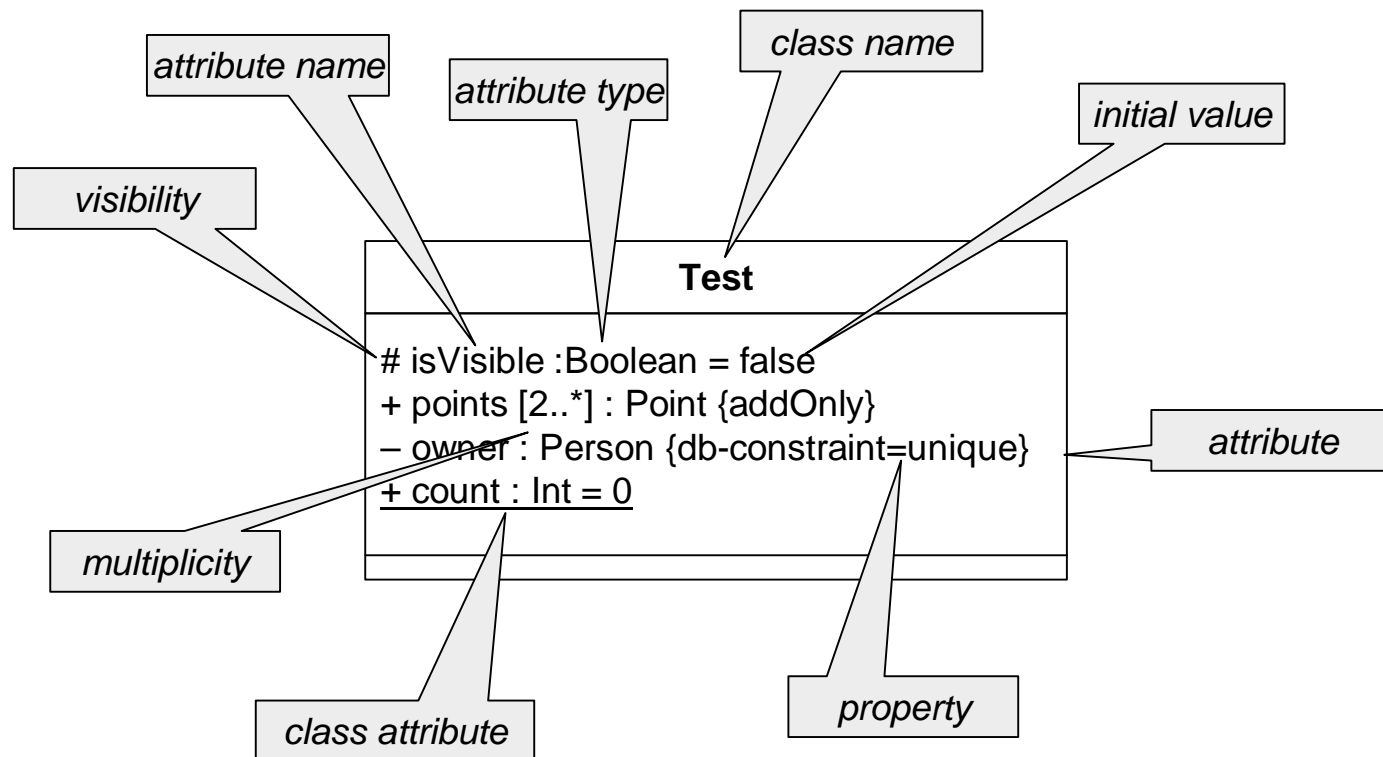
Example: Age computed from date of birth

Notation:



Note: Mostly, derived attributes are implemented by methods!

Attributes: Example



Class Attributes

A **class attribute** (static attribute, static member) is an attribute assigned to a class and not to a single object (as instance of the class).

Syntax: [Java]

```
static data_type attribute_name;
static data_type attribute_name = expression;
```

Dynamic semantics: [Java]

An unqualified access is possible inside the class declaration (or in its subclasses). Outside the class the attribute name must be qualified with the class name (`class_name.attribute_name`). Class attributes are initialized at link time.

When accessing class attributes a **static binding** and no dynamic binding takes place.

Class Attributes: Example

Java Example:

A counter counting the number of the instances generated .

```
class Counted {
    static int count = 0;
    Counted(...) {
        count = count + 1;
    }
    ...
}
```

Operations

UML distinguishes:

- Operations:** Signatures for the behavior of a class.
- Methods:** Implementation of an operation.

Characteristics:

- Visibility:** like attributes
- Polymorphism:** If the operation can be redefined in subclasses, it is polymorph.
- Abstract:** An abstract operation has no implementation in this class, i.e., no method.
- Parameter:** input and output parameters
- Return value**
- Multi-process behavior:** Are the methods allowed to be called by several processes in parallel, are they synchronized or must they synchronize themselves?

Operations: Overriding

Overriding (Redefinition)

A subclass defines a method with the same name as an inherited method. Thereby the inherited method is hidden at dynamic binding.

Signature in Java: name of the method and number and types of the formal parameters.

A method overrides the methods in every super class and super interface having the same signature. An overwritten method can be accessed using the keyword **super**.

Restriction/request: overriding and overwritten methods are not allowed to have different return types, conflicts in the throws-clauses, or conflicting visibilities.

```
class Point {
    void display(int x, int y){ ... }
    double distance(Point p) { ...}
}
```

```
class Point2 extends Point{
    void display(int x, int y) { ... } /* overriding */
    int display(int x, int y) { ... } /* won't work */
}
```

Operations: Overloading

A name is called **overloaded** if in the visibility range more than one declaration for the name is visible.

The overloading of a name with more than one meaning requires a dynamic binding of the name to the belonging concept.

Overloading signifies the fact that identifiers can be overloaded. Java allows the overloading of method names inside of a class. The overloading is widely used especially for constructors.

Overloaded method names require a name resolution which already takes place in Java at translation time using the method signature (argument types).

```
class Point {
    int x, y;
    Point (int x, int y) { this.x = x; this.y = y; }
    Point (float x, float y) { this.x = (int)x; this.y = (int)y;}
}
```

```
Point a = new Point(3, 4); Point b = new Point(3.0, 4.0);
```

Operations: Notation

Syntax:

```
[["«stereotype»"] [visibility] name ["("parameter-list)"] [":"return-type] [{"properties"}]]
```

Parameter:

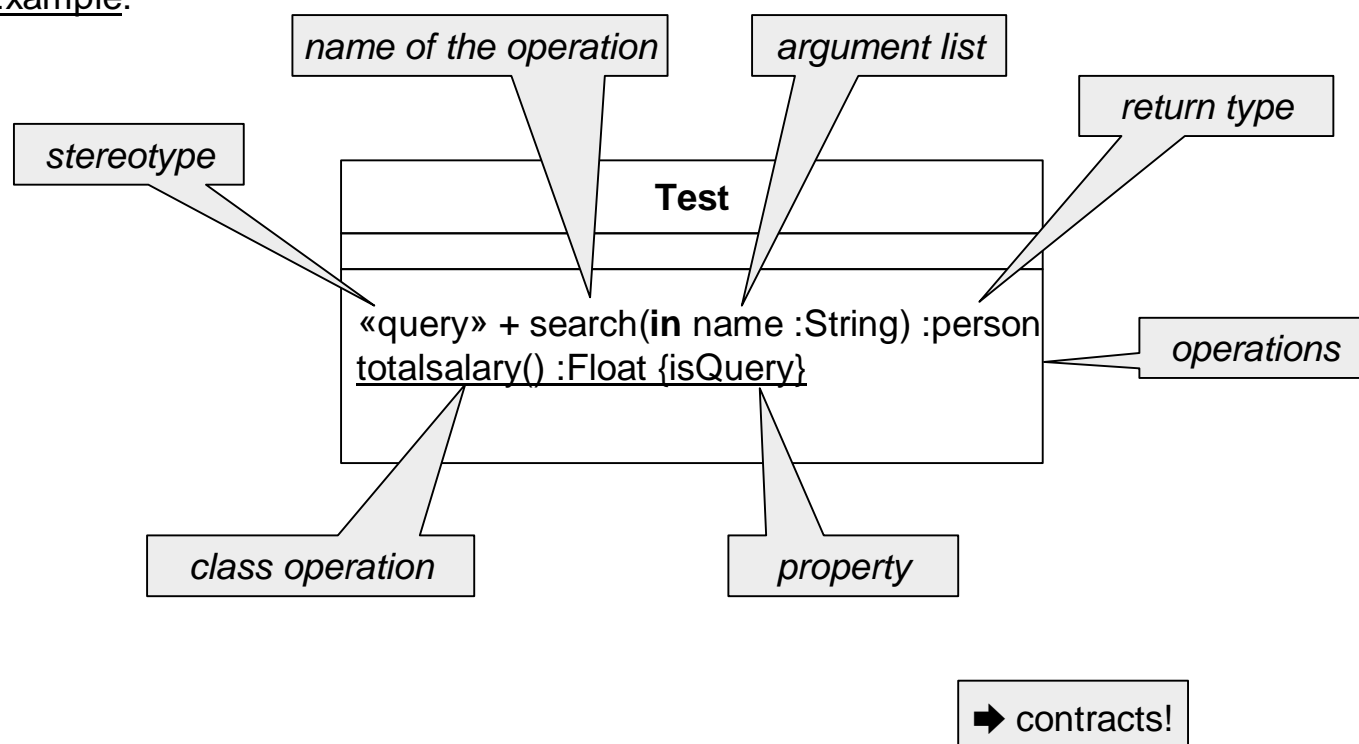
- ❑ Syntax: [direction] name ":" type ["=" default-value]
- ❑ direction: **in**, **out**, **inout**

predefined properties:

- ❑ *leaf*: operation cannot be overwritten in subclasses
- ❑ *isQuery*: no side effects
- ❑ *sequential*: only one caller (process) allowed at a time (caller is responsible!)
- ❑ *guarded*: operation synchronizes several callers autonomously
- ❑ *concurrent*: reentrant, several callers can use the operation in parallel

Operations

Example:



Operations: Class Methods

A **class method** (UML: class operation, Java: static function) is a method attached to a class and not to a single object (as an instance of the class).

It is called by sending a message to the class. **No dynamic binding** takes place.

Syntax (Java): The syntax of a class method is like the syntax of a method. It is marked with the modifier **static**.

Static semantics: The static semantics are similar to the static semantics of a method with the difference that in the body of a class method the pseudo variable **this** and non-static data fields or methods are not available.

Class methods may use other class methods, class attributes or objects delivered as parameters only.

Note: A constructor is a class method with a special notation (in Java).

Example: Class Methods

In an object-oriented programming language class methods and class attributes can be used to program in procedural style.

But this is a bad programming style!

```
class Point {
  int x, y;
  ...
  public static boolean smaller( Point p, Point q ) {
    return p.x < q.x && p.y < q.y;
  }
}
```

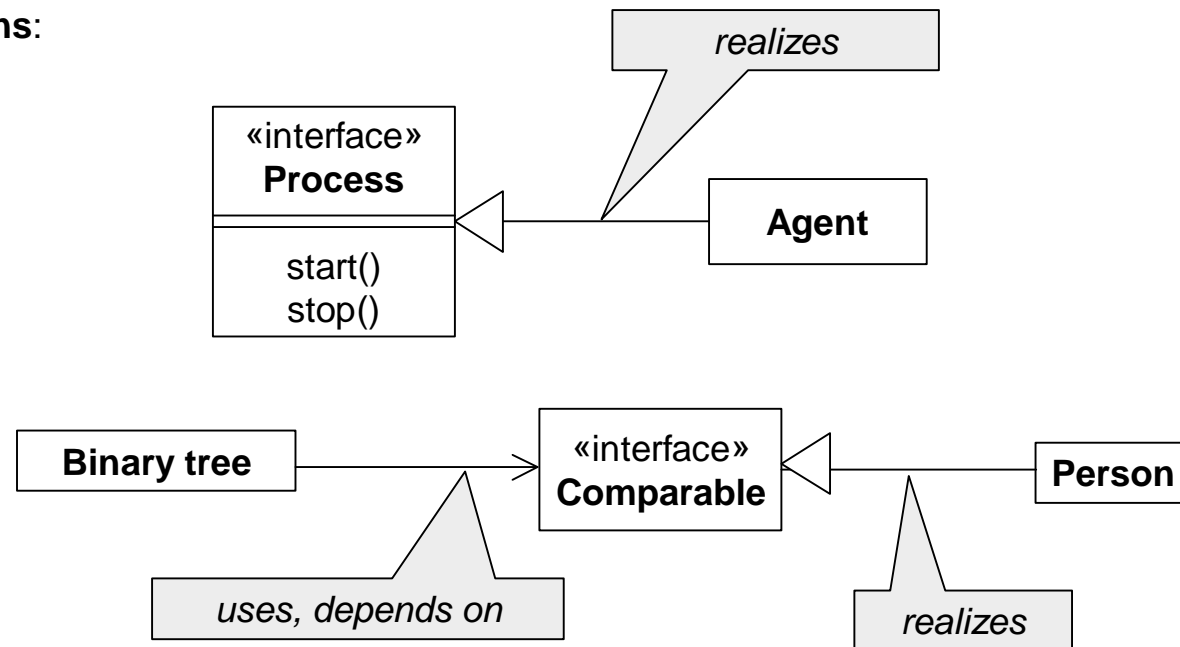
better:

```
class Point {
  int x, y;
  ...
  public boolean smaller( Point q ) {
    return x < q.x && y < q.y;
  }
}
```

Interfaces: Realization and Usage

A class **realizes** (implements) an interface used by other classes.

Notations:



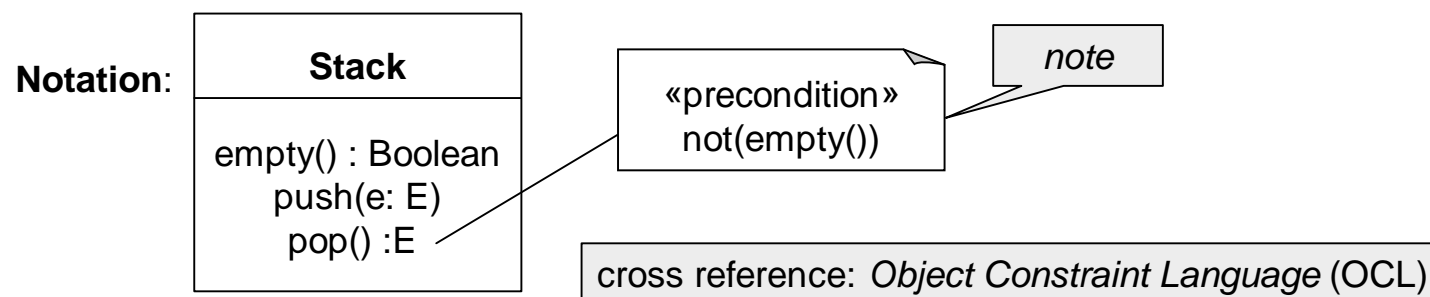
Class Diagrams in Design

3.3.21

Notes

A **note** is an annotation, e.g., of a class, relation or operation. Notes can also be categorized by stereotypes, e.g.,

- ❑ «*requirement*»
- ❑ «*responsibility*» : contracts
- ❑ «*semantics*» : description as structured text
- ❑ method body
- ❑ «*invariant*» : class invariants
- ❑ «*precondition*», «*postcondition*» of operations



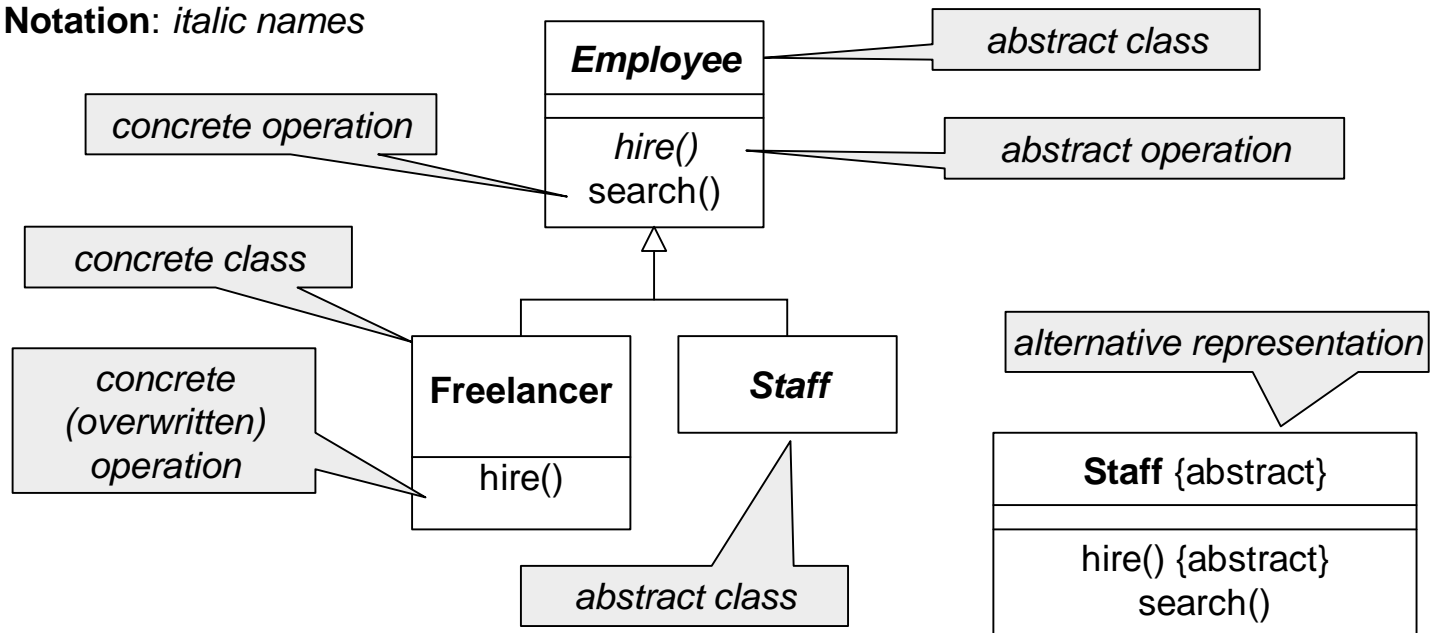
Class Diagrams in Design

3.3.22

Abstract Classes

An **abstract class** is a class which cannot have any instance, e.g., because the implementation of an operation is missing.

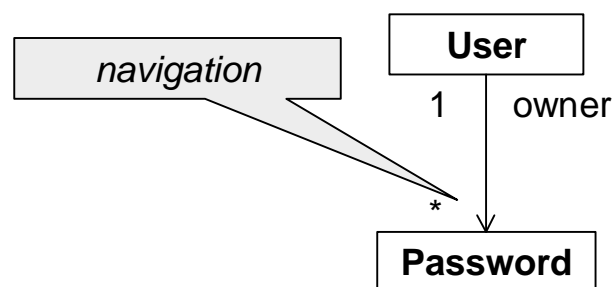
Notation: *italic names*



Associations: Navigability

An association, which can only be followed in one direction, is **navigable**.

Notation:



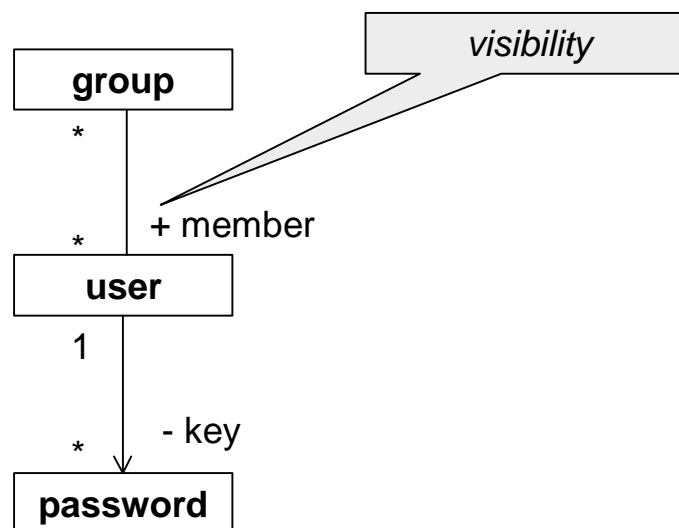
passwords don't "know" which user did choose them

Note: Navigability is orthogonal to multiplicity!

Associations: Visibility

Associations can be protected from accesses like attributes (*private, protected, public*).

Notation:

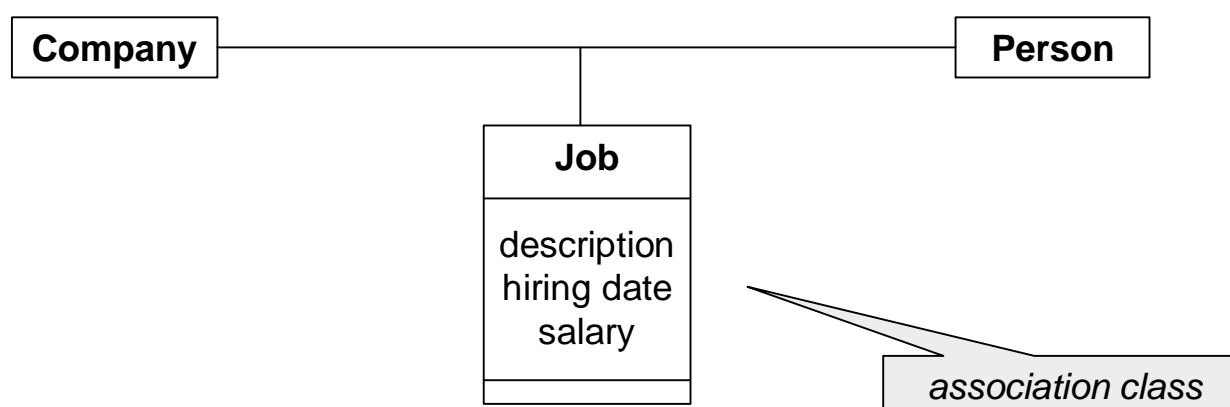


The key of a user is private, so the group cannot access it.

Association Classes

An **association class** aggregates attributes and operations describing an association in more detail.

Notation:

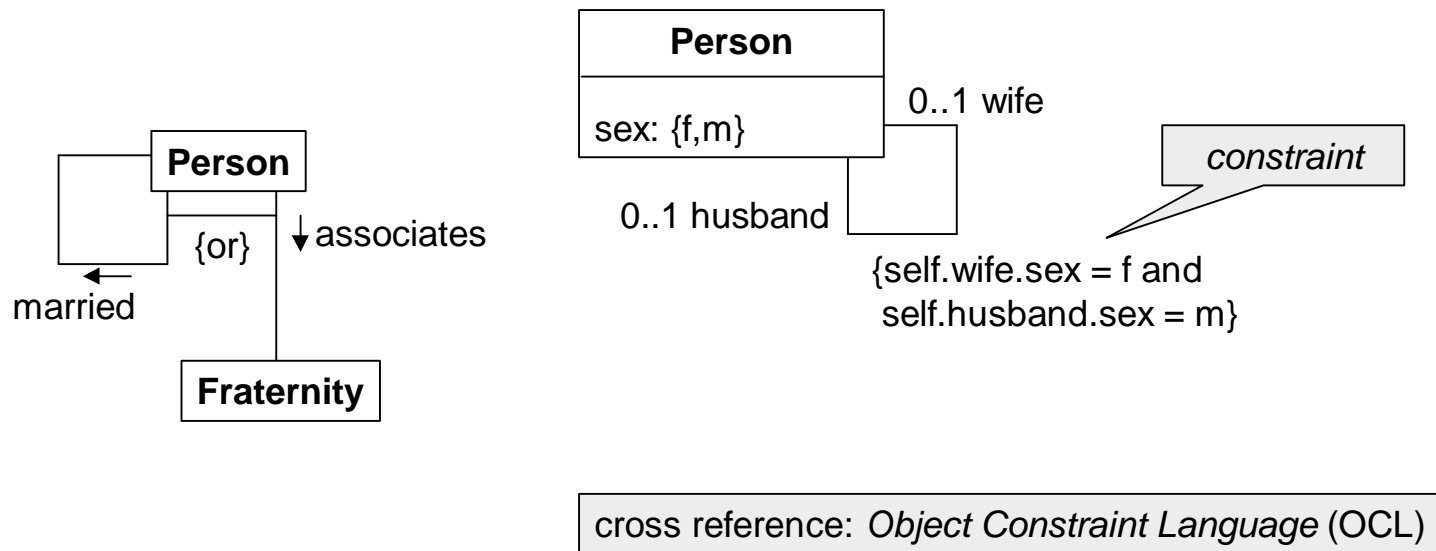


Note: „Reification“: The conversion of relations, processes, and properties into things, i.e., objects and classes.

Associations: Constraints

A **constraint** is a restriction on the combinations of attribute value or relations of a class.

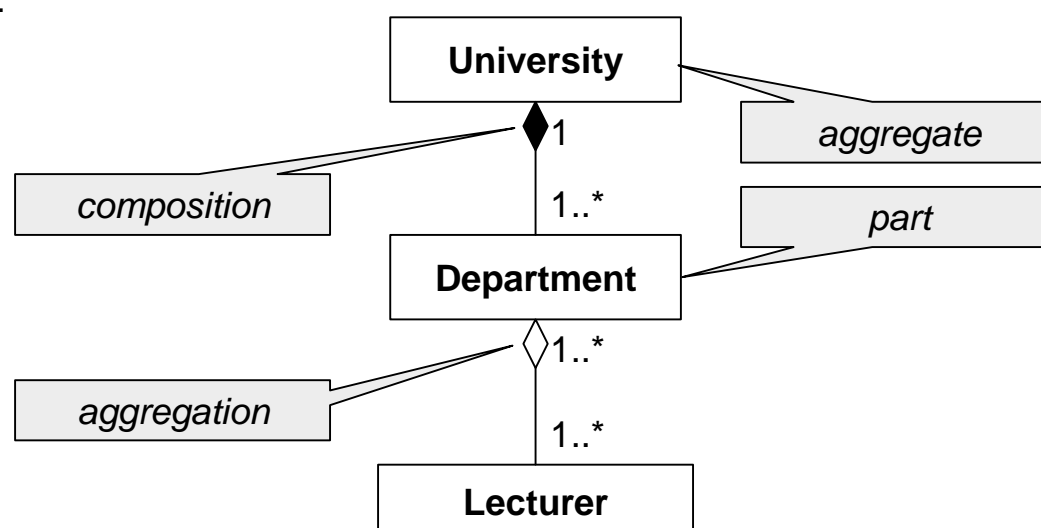
Notation:



Aggregation: Composition

Composition is a special form of aggregation. The composition (the whole) is composed of parts. Each part can be component of exactly one composition only. The parts „die“ with the aggregate, so the aggregate is responsible for their lifetime.

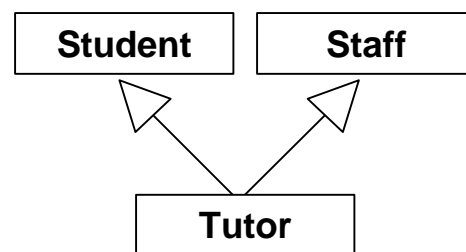
Notation:



Multiple Inheritance

Multiple inheritance allows a class to be subclass of multiple classes (**multiple generalization**).

Notation:



Note: Naming clashes.

Generalization: Use of Stereotypes

Examples of stereotypes which can be used for classification:

- «*implementation*» : The implementation is inherited, but not the interfaces, i.e., attributes and operations may have a different visibility.
- «*complete*» : All subclasses are contained in the model, i.e., no more classes may be derived.
- «*incomplete*» : Other subclasses can be inserted.
- «*disjoint*» : Each object of the super class may only belong to one subclass.
- «*overlapping*» : An object may belong to several subclasses.