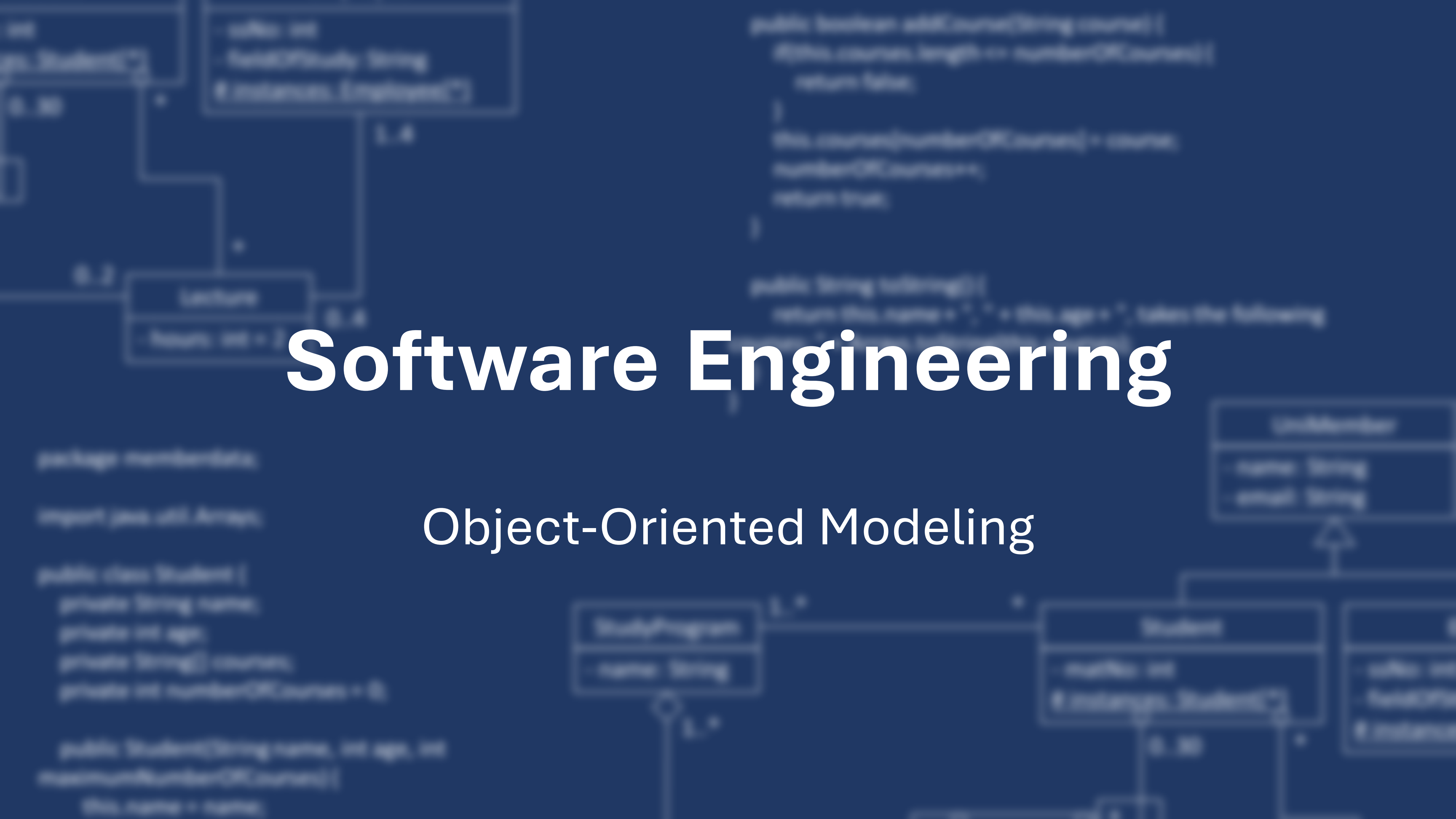


Software Engineering

Object-Oriented Modeling

Structure of the OOSE Lectures

Revisit and deepen basics of programming.

Revisit and deepen basics of object-oriented programming.

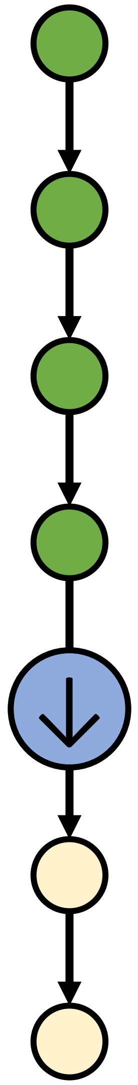
Cover advanced object-oriented principles.

How to model OO systems (UML) and map models to code.

Object-oriented modeling techniques.

Design patterns as means to realize OO concepts (I).

Design patterns as means to realize OO concepts (II).



Last Lecture

How to model (OO) systems?

- Object diagrams for example runtime configurations.
- Class diagrams for the structure of a system.
- Sequence diagrams for the behavior of a system.

How to map models to code?

- Class diagrams → code.
- Sequence diagrams → code.
- Activity diagrams → code.

Aims of this Lecture

Get to know (OO) modeling principles.

About object behavior:

- Liskov
- Design by Contract.
- Behavior Protocols

About dependencies:

- SO(L)ID.
- Cohesion, Coupling, KISS, DRY and others.
- How NOT to model software.

What goes wrong with software?

Adapted from *Bad Signs of Rotting Design*, Robert C. Martin, 1996.

From good design to rotting software

1. You start with a clear picture of what your system should do and how to implement it.
2. Then, requirements start changing.
3. Over time, the software development becomes harder, even easy changes scare you. The software starts rotting.
4. Eventually, the whole system must be re-implemented.

Signs of rotting software

- Rigidity: changes are hard to implement; not changing code becomes the standard.
- Fragility: Small changes already lead to chains of problems.
- Immobility: Modules are so intertwined that reusing anything is impossible.
- Viscosity: It's easier to hack new functionality than to stick to the initial design.

Interfaces

Interfaces tell us how to call an operation.

- They do not provide access control.

- They do not tell us what effects calling this operation will have (design by contract).
- They do not tell us the order in which operations can be called (behavior protocols).
- They do not tell us what the operation needs (required interfaces).

https://www.sueddeutsche.de/digital/forschung-so-hat-es-die-informatik-pionierin-liskov-an-die-spitze-geschafft-1.3168623

Liskov Substitution Principle

We want to answer the following question we raised in the OOP lectures:

What does it mean that a dynamic type is compatible to a static type?

And why?

Static vs. Dynamic Type

Static type: the type used in a declaration. Also called declared type or compile-time type.

Dynamic type: type of the expression at runtime. Also called runtime-time type.

```
Person adrian = new Student("Adrian", "Bajraktari", 26, 2969443);
```

Static type   Dynamic type

- The dynamic type must be compatible to the static type.

Liskov Substitution Principle (in non-mathematical)

Type `B` is subtype of type `A`

$\leftrightarrow$

Instances of `B` can always substitute instances of `A`

$\leftrightarrow$

Instances of `B` need at most and provide at least as much as instances of `A`

In simple: When expecting an instance of `A` but receiving an instance of `B`, we want to be able to treat it the same as an instance of `A`.

Liskov Substitution Principle

When expecting an instance of `A` but receiving an instance of `B`, we want to be able to treat it the same as an instance of `A`.

How can we interact with objects?

- Read or write their instance variables.
- Call their instance methods.

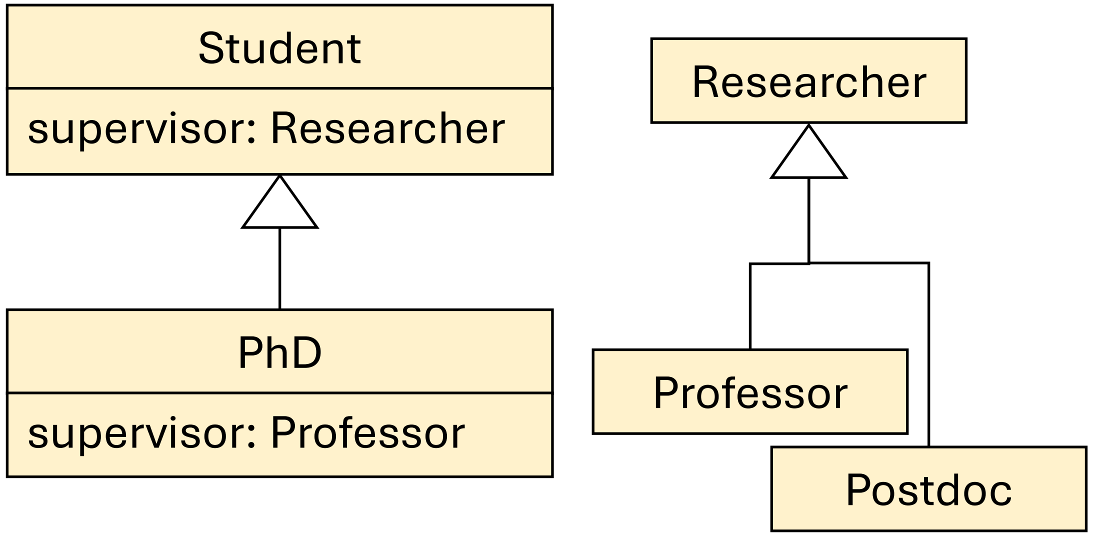
$\}$ Instances of `B` must provide at least all members of `A`.

Let's investigate both cases in detail.

Instance Variables

Why are subtypes not allowed to "override" instance variables?

Let's examine the following example:



```
Student adrian = new Student(...);

adrian.supervisor = andreas; //Professor ✓
adrian.supervisor = mersedeh; //Postdoc  ✓
```

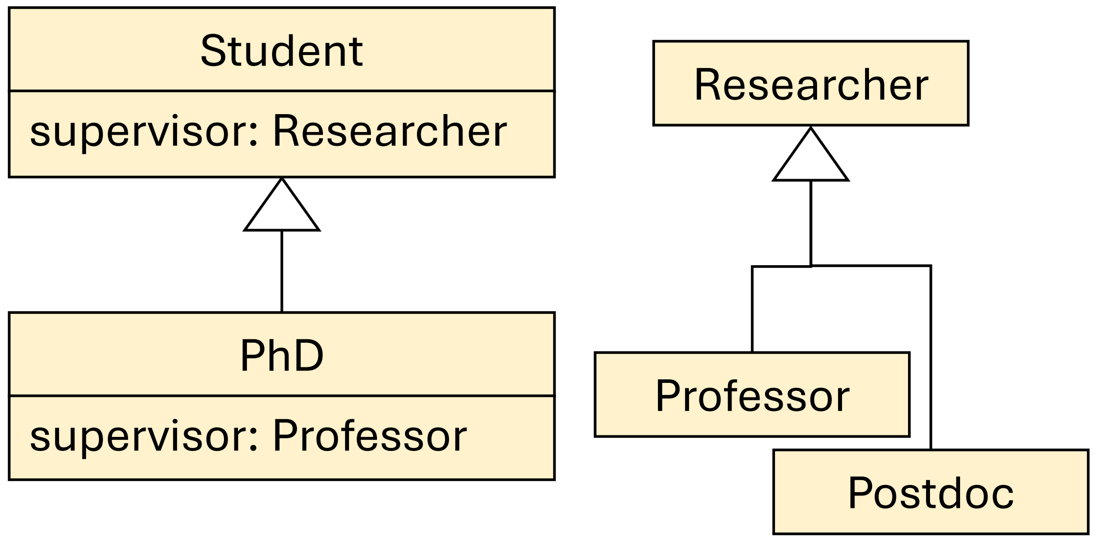
```
Student adrian = new PhD(...);

adrian.supervisor = andreas; //Professor ✓
adrian.supervisor = mersedeh; //Postdoc  ✗
```

So, when we allow overriding of instance variables, we encounter a problem when setting them to a subtype of the original's type.

Instance Variables

Why are subtypes not allowed to "override" instance variables?

Let's examine the following example:



```
Student adrian = new Student(...);
adrian.supervisor = mersedeh; //Postdoc

Researcher r = adrian.supervisor;
```
✓

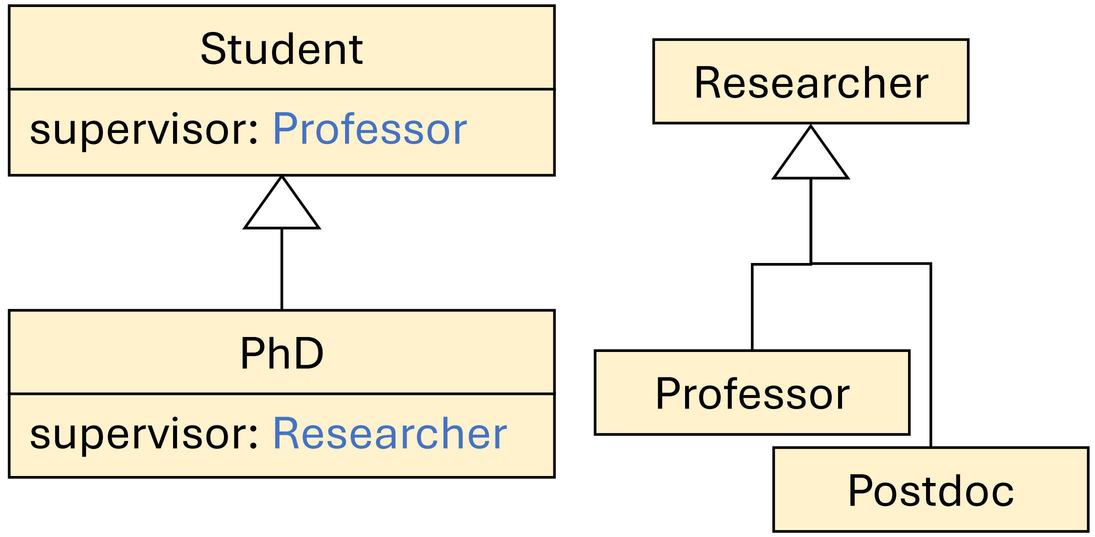
```
Student adrian = new PhD(...);
adrian.supervisor = andreas; //Professor

Researcher r = adrian.supervisor;
```
✓

For reading instance variables, however, we have no problem.

This is why in some languages, overriding of immutable instance variables is allowed.

Instance Variables

Let's turn things around:



```
Student adrian = new Student(...);
adrian.supervisor = andreas; //Professor   ✓
Professor p = adrian.supervisor;           ✓
```

```
Student adrian = new PhD(...);
((PhD)Adrian).setSuper(mersedeh);          ✓
//Postdoc

Professor p = adrian.supervisor;           ✗
//Postdoc
```

Allowing more general types in overriding instance variables gets rid of the writing problem by restricting the values to the more specific type, but now introduces uncertainty when reading the values.

Intermediate Observations

Neither subtypes nor supertypes work well for overriding variables. Thus, Java and many other OO languages do not allow this.

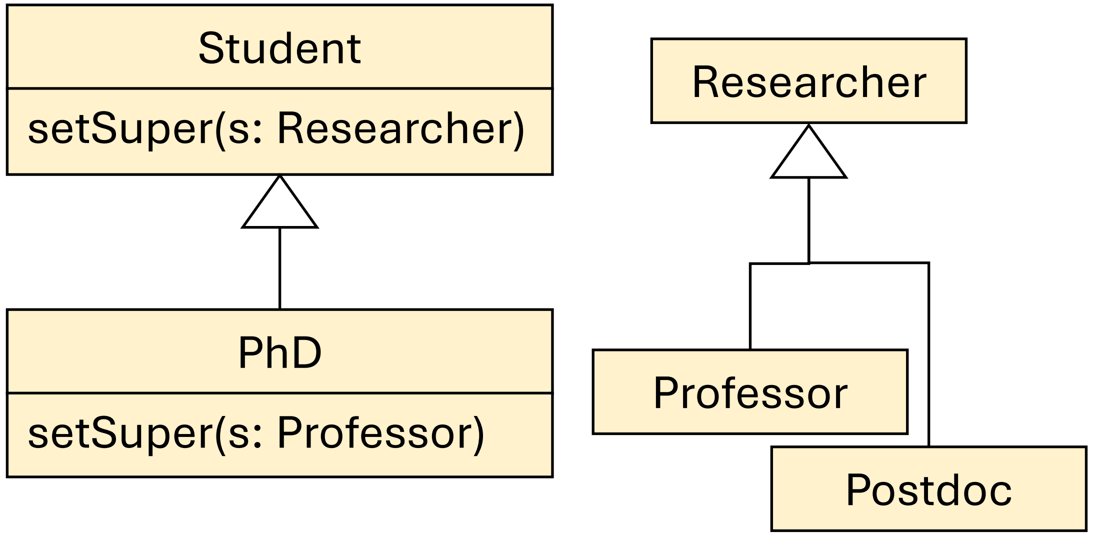
Instead, they are hidden (see OOP lectures).

Now what about methods?

For this, we investigate parameter types and return types.

Parameter Types

Given the following model:



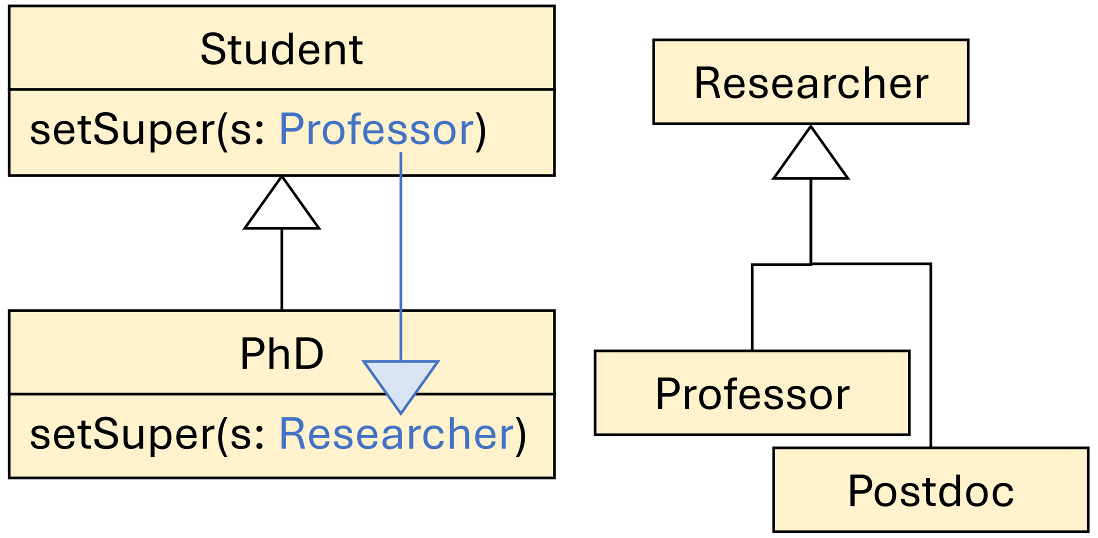
```
Student adrian = new Student(...);

adrian.setSuper(andreas); //Professor   ✓
adrian.setSuper(mersedeh); //Postdoc    ✓
```

```
Student adrian = new PhD(...);

adrian.setSuper(andreas); //Professor   ✓
adrian.setSuper(mersedeh); //Postdoc    ✗
```

We can say that overriding method's parameter types cannot be subtypes of their counterparts in the supertype.

Parameter Types

Given the following model:



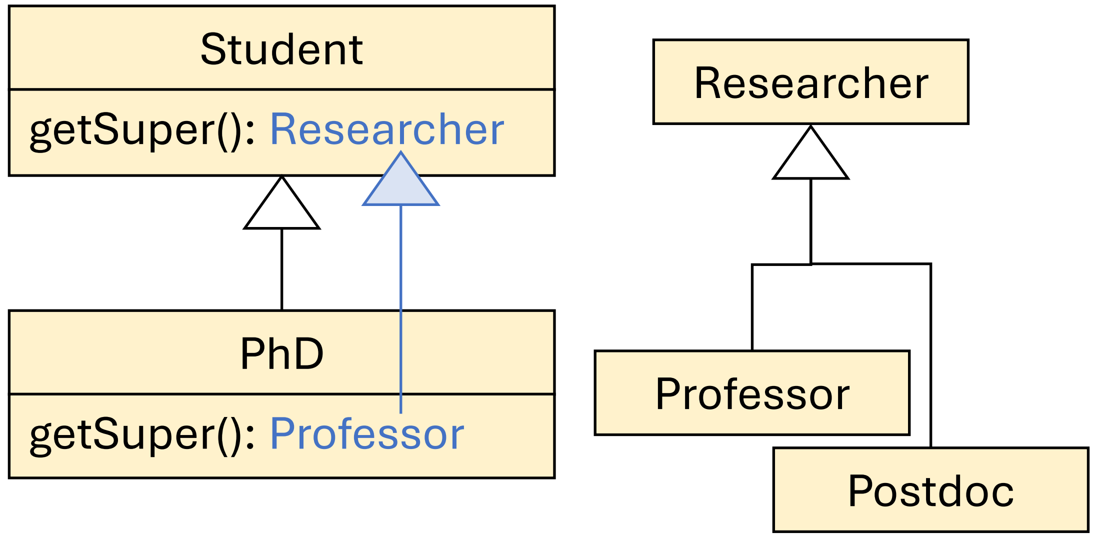
```
Student adrian = new Student(...);

adrian.setSuper(andreas); //Professor
```
✓

```
Student adrian = new PhD(...);

adrian.setSuper(andreas); //Professor
```
✓

For parameter types, we can define them to be more general in the overriding method. Parameter types are contravariant (against the subtyping relation).

Return Types

Given the following model:



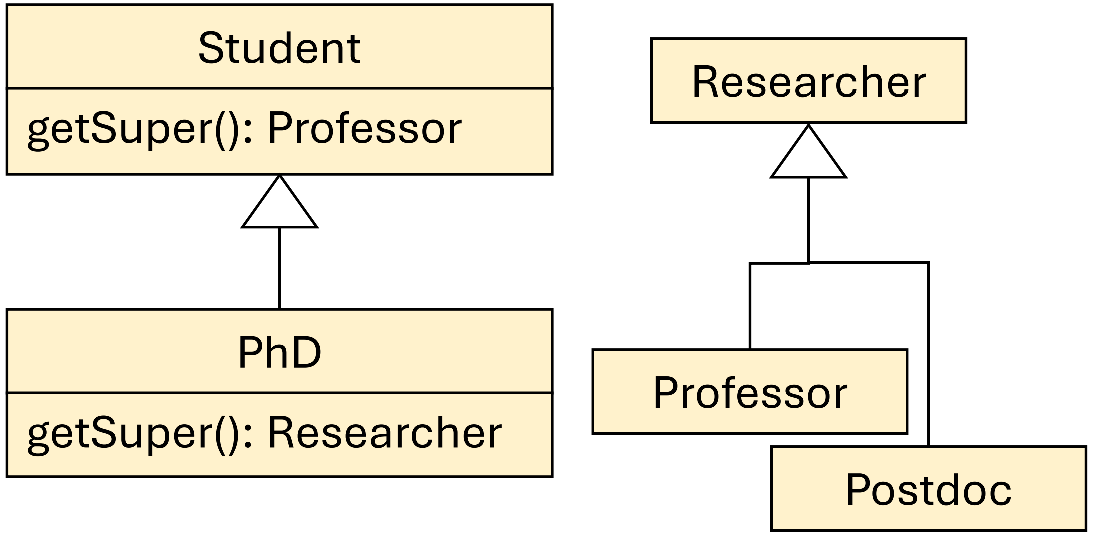
```
Student adrian = new Student(...);

Researcher r = adrian.getSuper();
//Researcher
```
✓

```
Student adrian = new PhD(...);

Researcher r = adrian.getSuper();
//Professor
```
✓

We can say that overriding method's return types can be subtypes of their counterparts in the supertype. Return types are covariant (with the subtype relation).

Return Types

Given the following model:



```
Student adrian = new Student(...);

Professor p = adrian.getSuper();
//Professor
```
✓

```
Student adrian = new PhD(...);

Professor p = adrian.getSuper();
//Researcher
```
✗

We can say that overriding method's return types cannot be supertypes of their counterparts in the supertype.

Liskov Substitution Principle Extended Edition

Type `B` is subtype of type `A`

$\leftrightarrow$

Instances of `B` can always substitute instances of `A`

$\leftrightarrow$

Instances of `B` need at most and provide at least as much as instances of `A`

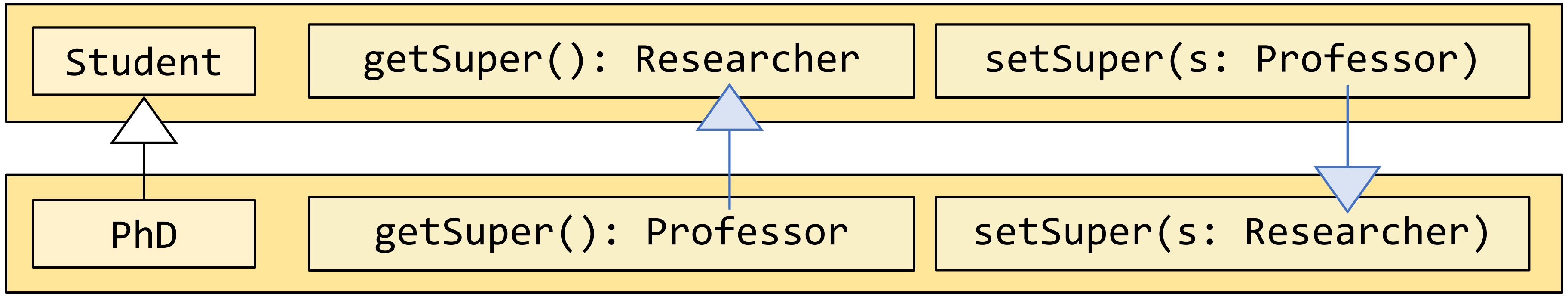
$\leftrightarrow$

Methods of `B` have contravariant parameter types (they are supertypes of their counterparts in `A`) and covariant return types (they are subtypes of their counterparts in `A`).

LSP rules

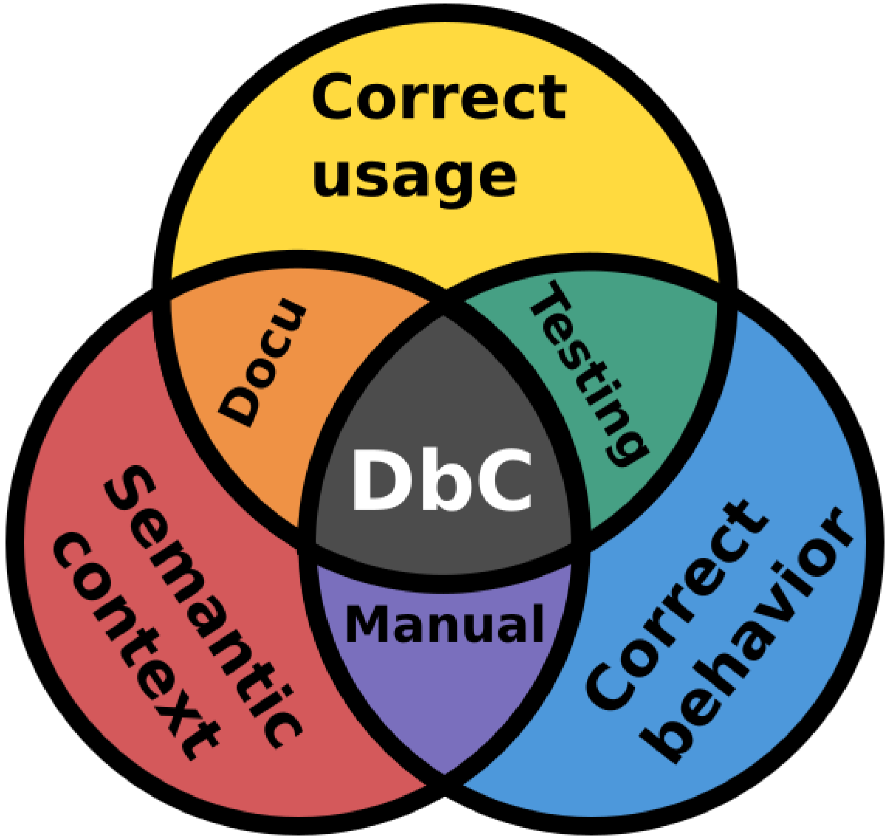
When a method overrides another method in the classes' supertype,

- parameter types must be contravariant
- return types must be covariant.



Attention! In Java, parameter types must always be equal. Otherwise, the method would not override, but overload.

Questions?



Design by Contract

Design By Contract

In design by contract, we make the assumptions of the program explicit by declaring them as logical constraints.

The contract is the set of all these logical constraints that tell us how to partners interact.

- Client: user of a method / class.
- Contractor: used method / class.

In DbC, there are three types of constraints:

- Invariants $\longrightarrow$ legal states.
- Preconditions $\longrightarrow$ legal behavior.
- Postconditions $\longrightarrow$ legal behavior.

Preconditions

A precondition defines the constraints for a successful execution of a method.

- Defined in the beginning of a method.
- Example: Square root of a number:
    - Signature: `squareRoot(int input)`
    - Precondition: `input >= 0`

The contractor defines the precondition. The client is responsible to adhere to the precondition.

Postcondition

The postcondition describes the result and side effects of the method in a declarative manner.

- Defined as last statement before return.
- Example: Square root of a number:
  - Signature: `squareRoot(int input)`
  - Postcondition: `input = result * result`

The contractor defines and adheres to the postcondition.

Note that in a postcondition, we always know that the precondition is true.

Invariant

The invariant describes the legal states of an object.

- A constraint that is true for all instances of a class.
- Example: Student record.
    - Invariant: The total grade is always the weighted average of all module grades.

All state-altering operations of a class are responsible to adhere to the invariant.

Mind that fulfilling the invariant is only necessary after a method has been executed. Within a method of the class, violating the invariant is allowed.

Object Constraint Language

OCL is the UML-way to define constraints, including DbC.

```
context Hashtable inv: numElements >= 0
```

```
context Hashtable::put(key,entry) pre: !containsKey(key)
```

```
context Hashtable::put(key,entry) post: containsKey(key) and get(key) = entry
```

The context is either a class or a specific object.

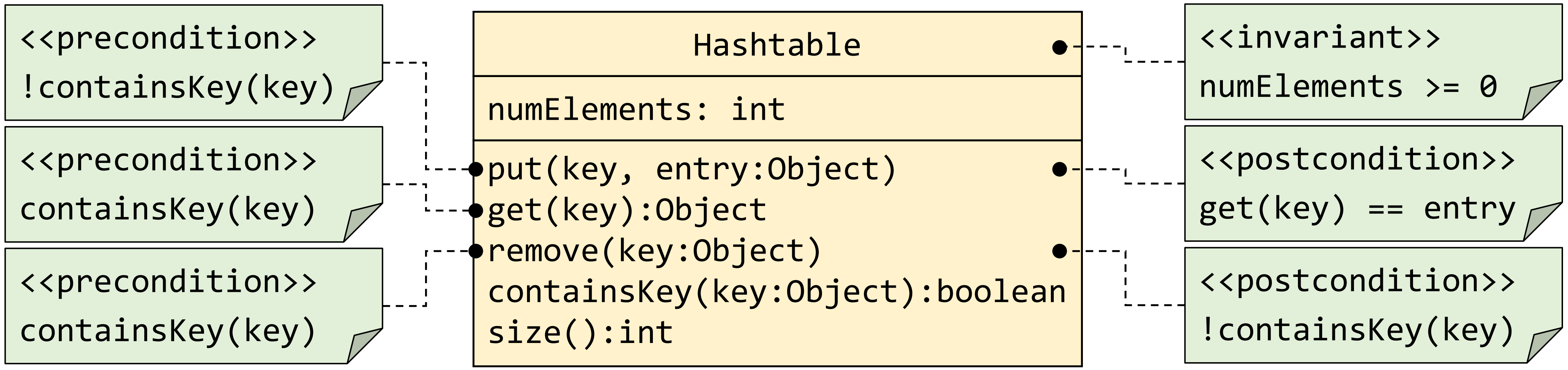
For pre and post condition, we need to specify the method with its parameters.

Indicator of type of constraint.

Contraint(s).

Object Constraint Language

OCL constraints can be added to diagrams via notes.



Design by Contract: Language Support

- Java: `assert` since v1.4 and Java Modeling Language (JML).
- Kotlin: "Kotlin Contracts".
- Eiffel: Full support.
- C++: via attribute syntax.
- Other languages: Make contracts explicit in documentation!

The idea behind DbC plays a crucial role in the creation of correct object-oriented software (substitution).

DbC in JML

```
class Edge {
   //@invariant first != null && second != null
   Node first, second;
   Edge(Node first, Node second) {
      this.first = first;
      this.second = second;
   }
   /* @requires e != null;
    * @ensures \result <==> first.equals(e.first) && @second.equals(e.second); */
   boolean equals(Edge e) {
      return first.equals(e.first) && second.equals(e.second);
   }
   public static void main(String[] args) {
      Node a = new Node();
      Node b = new Node();
      System.out.println(new Edge(a, b).equals(null));
   }
}
```

Stronger Preconditions and Weaker Postconditions

In statically typed languages

- Postcondition = return types
- Preconditions = parameter types.

These conditions can be statically checked by the compiler.

In this context, stronger preconditions (= more specific parameter types) and weaker postconditions (more general return types) do not yield a correct subtype and are thus forbidden in most OO languages.

This is extended by design by contract to additionally have constraints.

Questions?

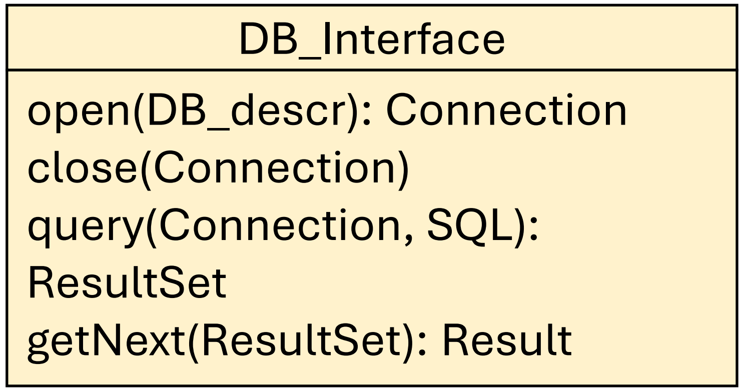
https://www.welt.de/debatte/kommentare/article140710698/Die-Zeit-der-Oberlehrer-ist-nun-wirklich-vorbei.html

Behavior Protocols

Behavior Protocol

Interfaces still do not tell us in which order certain methods may be executed.

In the example: It is not possible to post a database query before the connection to the database is established.



```
DB_Interface
open(DB_descr): Connection
close(Connection)
query(Connection, SQL): ResultSet
getNext(ResultSet): Result
```

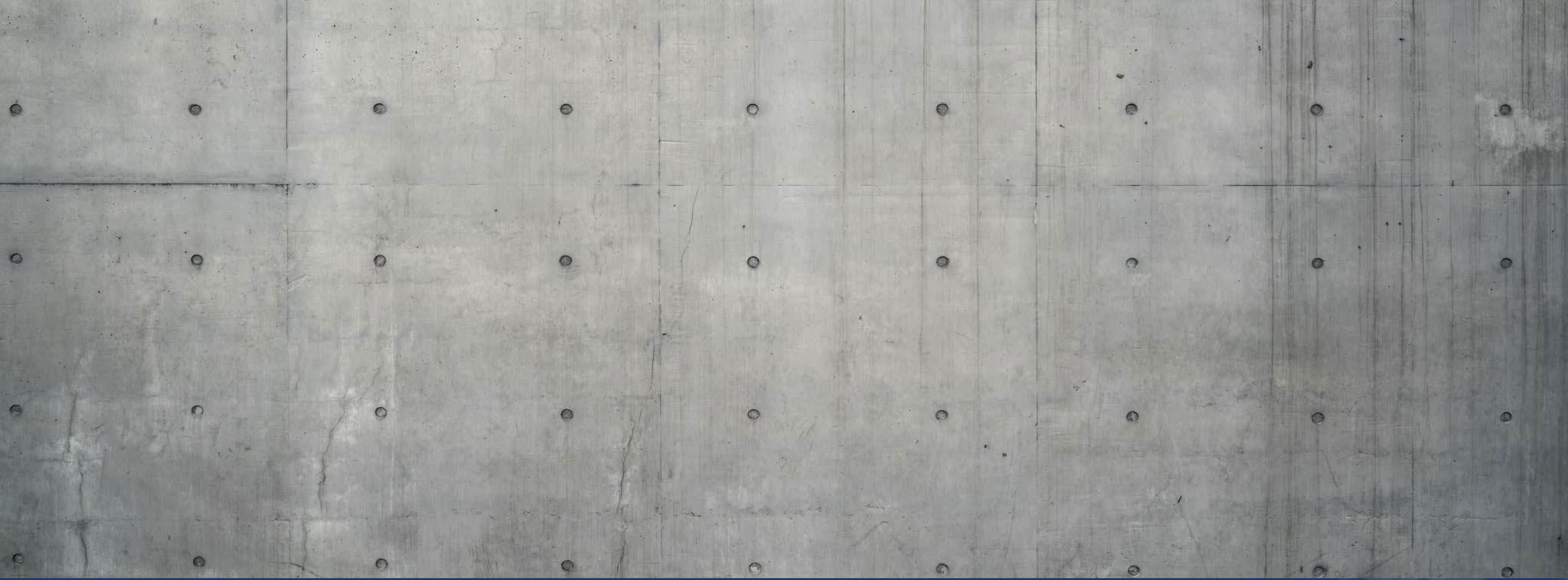
Behavior Protocol

Behavior protocols are regular expressions, defined per type or per set of methods, that define the legal order and number of repetitions of calling methods.

```
DB_Interface
open(DB_descr): Connection
close(Connection)
query(Connection, SQL):
ResultSet
getNext(ResultSet): Result
```

In the example: First, one must establish a connection to the database. Only then, an arbitrary number of queries can be posted. In the end, the connection must be closed again.

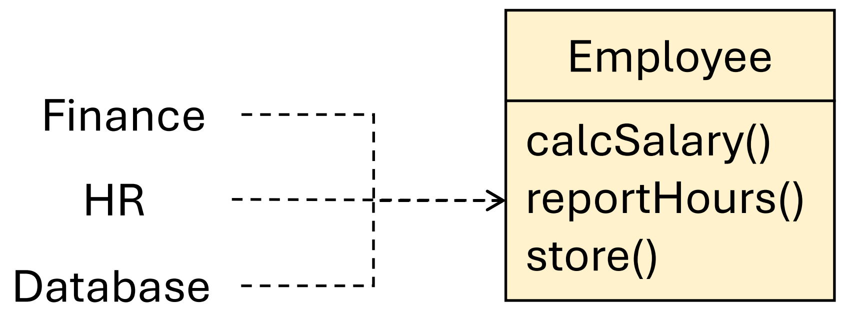
```
protocol DB_Interface_Use =
    open(DB_descr) ,
    (
        query(Connection,SQL) : ResultSet ,
        ( getNext(ResultSet) : Result )*
    )* ,
   close(Connection)
```



SOLID and Co.

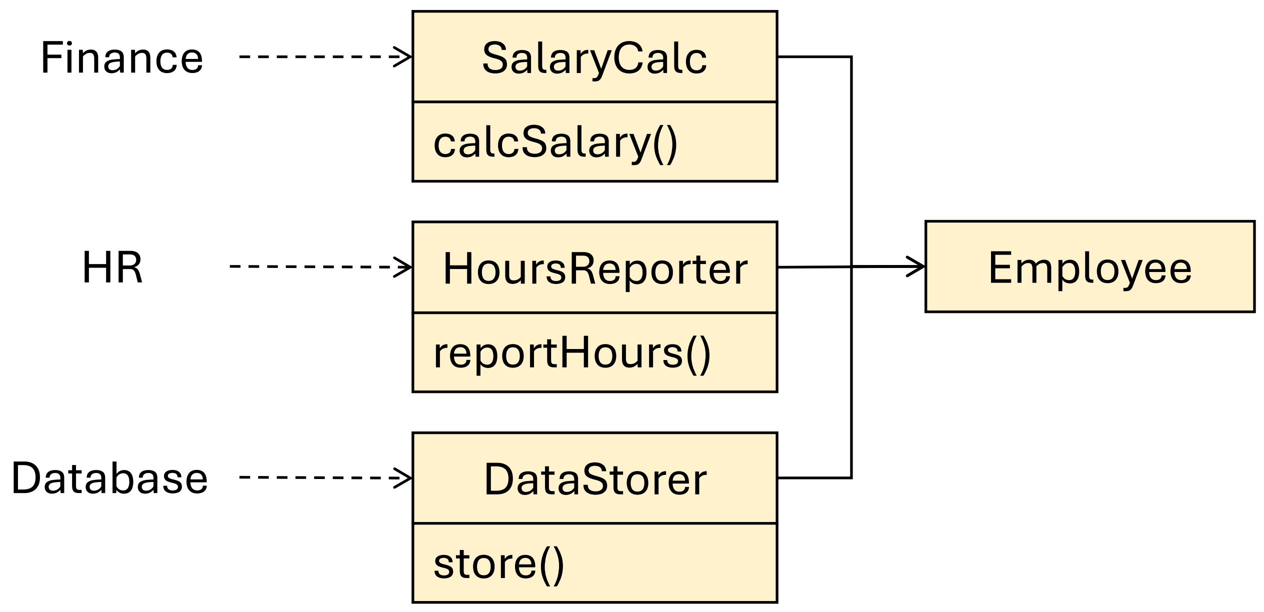
Single Responsibility Principle

Problem: God classes. Functionalities of multiple different aspects used by different actors lie within one class. Changes to code concerned with one actor also affects other actors.



Solutions:

- Move responsibilities to dedicated class.
- Facade: Unified access point, but implementation is in dedicated classes.



Open-Closed Principle

"A software artifact should be open for extension but closed for modification." – Bertrand Meyer

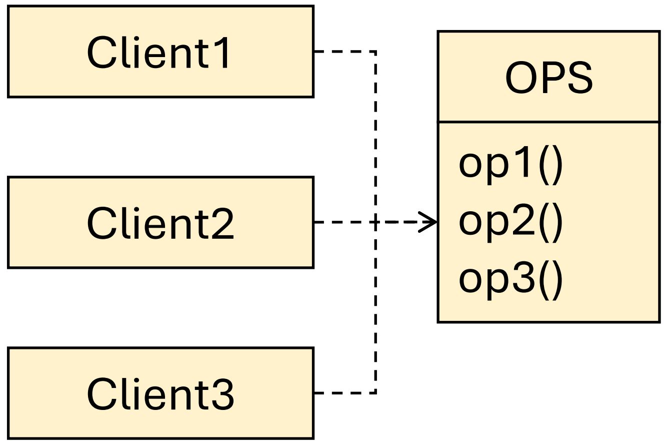
When requirements change, rather extend the code instead of changing existing code.

This can be achieved by separating code that has different reasons to change (SRP) and by organizing dependencies between code units (DIP).

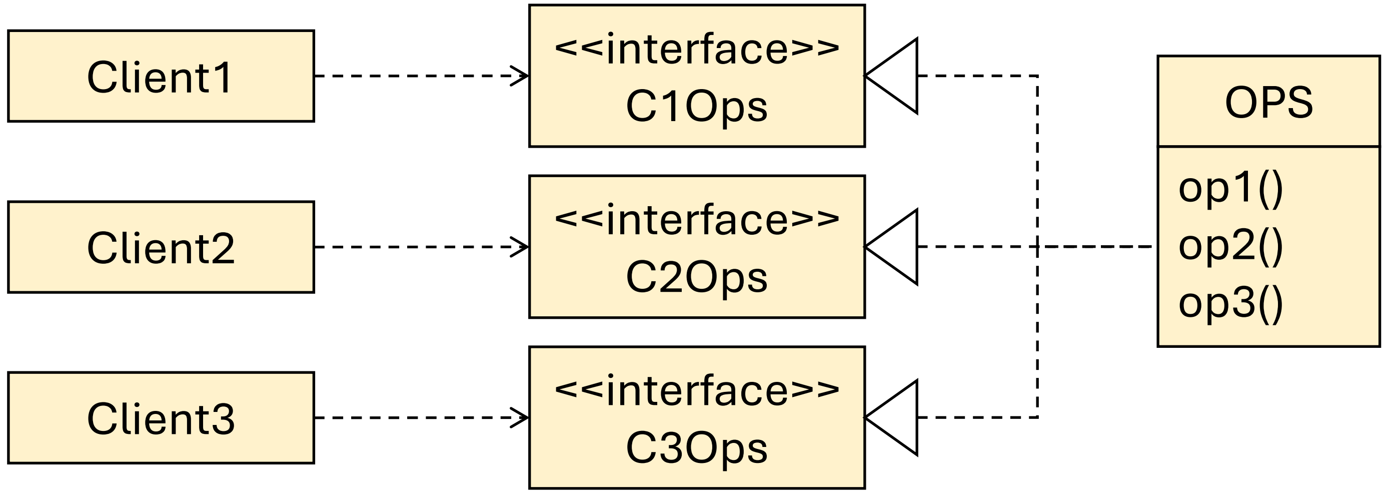
Interface Segregation Principle

Problem: Different clients of the same code only use a part of the interface.

All clients depend on the whole interface though not using it.

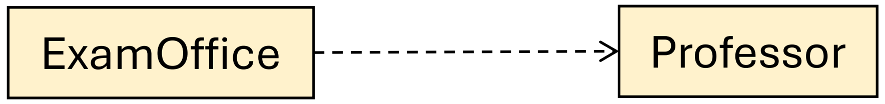


Solution:

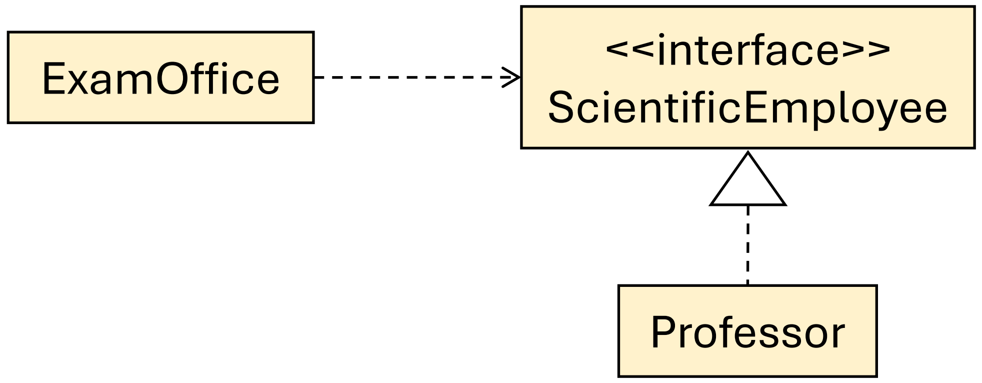


Dependency Inversion Principle

Problem: Concrete modules are subject to change. A module depending on a concrete module becomes subject to change as well.



Solution: Make client and implementation both dependent on an abstraction.



Interface Design

Rules to memorize:

- Using a class name `C` in static type declarations restricts the assignable instances to `C` and its subclasses.
- Using an interface name `I` in static type declarations allows to assign instances of any class that implements `I`.

Using interfaces mainly for static type declarations eliminates concrete class names. In the best-case, concrete types are only used for object creation (see factory method and prototype patterns to also enhance that).

Questions?

https://www.liveabout.com/when-is-it-okay-to-turn-down-a-modeling-job-2379348

Further Modeling Principles / Problems

More Design Principles by RCM

- (Dependency Inversion Principle (DIP): Dependencies only point towards the same or a higher level of abstraction.)
- Stable Abstractions Principle (SAP)
  - More stable units should be more abstract.
  - More instable units should be more concrete.
- Stable Dependencies Principle (SDP): Dependencies only point towards the same or a higher level of stability.
- Acyclic Dependencies Principle (ADP): The dependency graph should not contain any cycles.

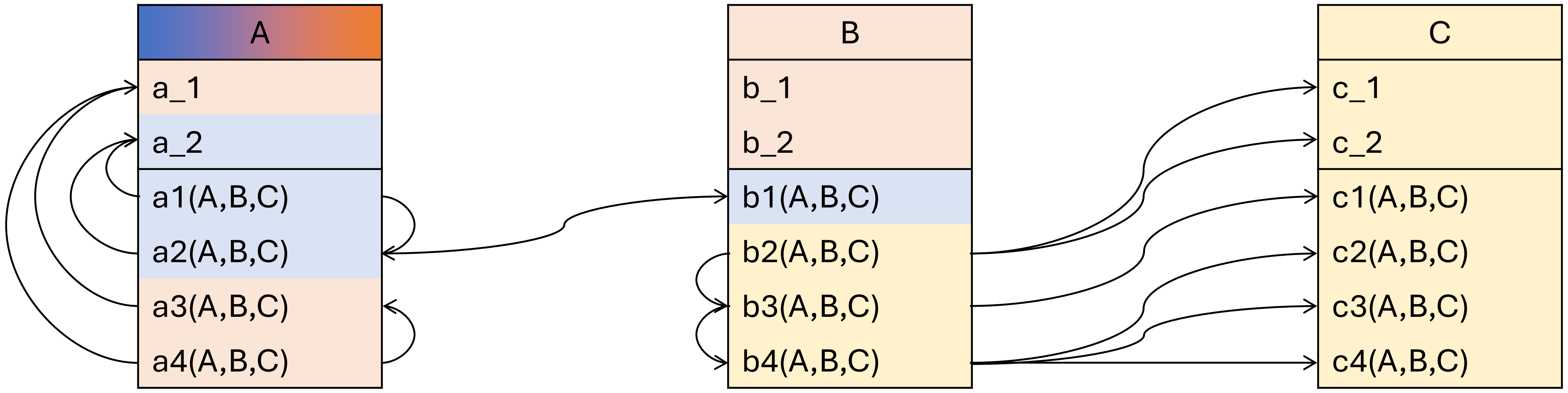
Cohesion and Coupling

Cohesion is the degree of interdependency between classes within a unit (e.g., a subsystem). High cohesion = many dependencies between classes within the unit.

Coupling is the degree of interdependency between units (e.g., between subsystem). High coupling = many dependencies between units.

For well maintainable systems, most dependencies should stay within the same subsystem. A subsystem decomposition should thus strive for high cohesion and low coupling.

Cohesion and Coupling: Example

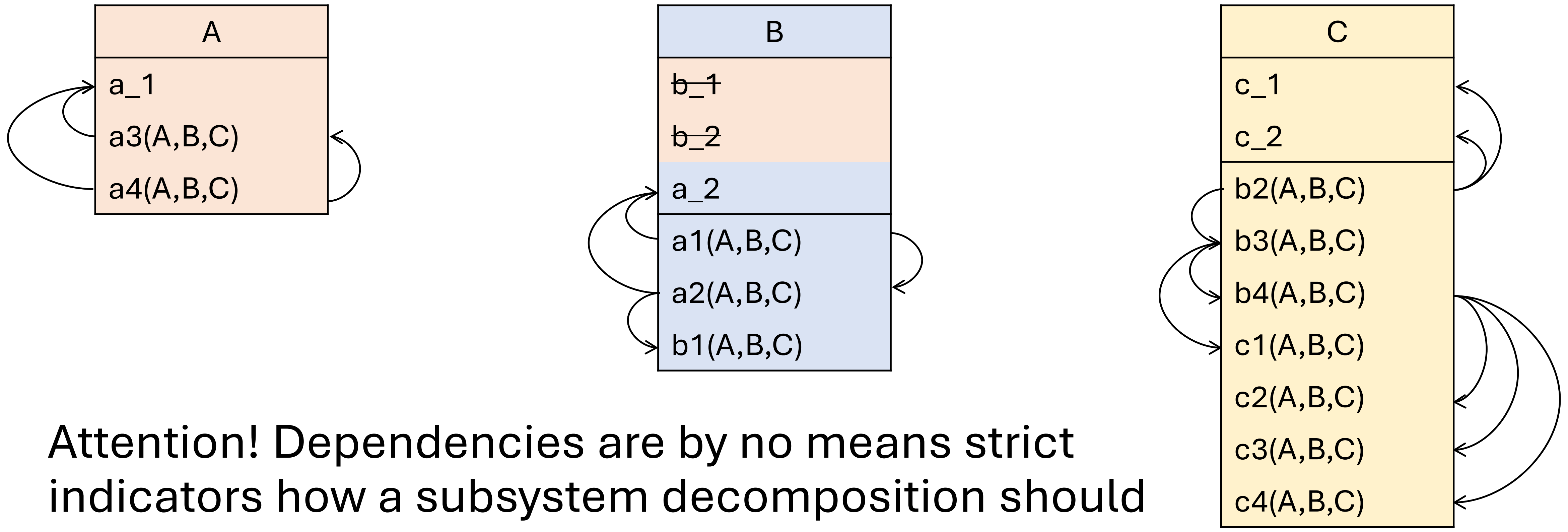


2 units of cohesion.
→ split into 2 classes!

Low cohesion.
→ b_1, b_2 unused.
→ b1 belongs to A!
→ b2 - b4 belong to C!

No cohesion.
B cares more about C elements than C.

Cohesion and Coupling: Example enhanced



Attention! Dependencies are by no means strict indicators how a subsystem decomposition should be done! The main goal should be to group elements of the system in logical units. Dependencies can help to decide between variants of decompositions.

Information Hiding

Reduce dependencies by hiding internals of an encapsulation unit.

- Layer 0: Pure code lines: no hiding possible.
- Layer 1: Methods: hide the implementation of operations.
- Layer 2: Classes: access to attributes only through methods.
- Layer 3 and 4: Packages, Subsystems: Facade.

The law of Demeter is a famous principle that realizes information hiding.

Law of Demeter

"Who am I allowed to talk to?"

- Object with itself.
- Object as argument.
- Object as attribute.
- Objects in collections.
- Objects created by methods.
- Global objects.

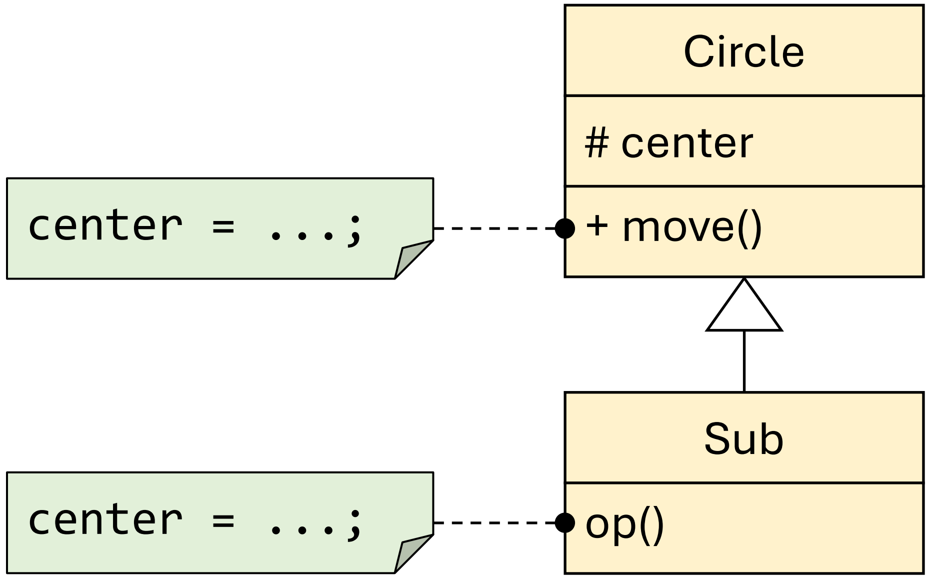
```
Public class ClassB extends ClassA {
   public ClassX faster = new ClassX();
   public ClassY[] y;
   public int method(ClassC better) {
      ClassZ daft = new ClassZ();
      stronger = y[3];

      this.method2();
      harder.workIt();
      better.makeIt();
      faster.doIt();
      stronger.makesUs();
      daft.punk();
      …
      return i;
   } … }
```

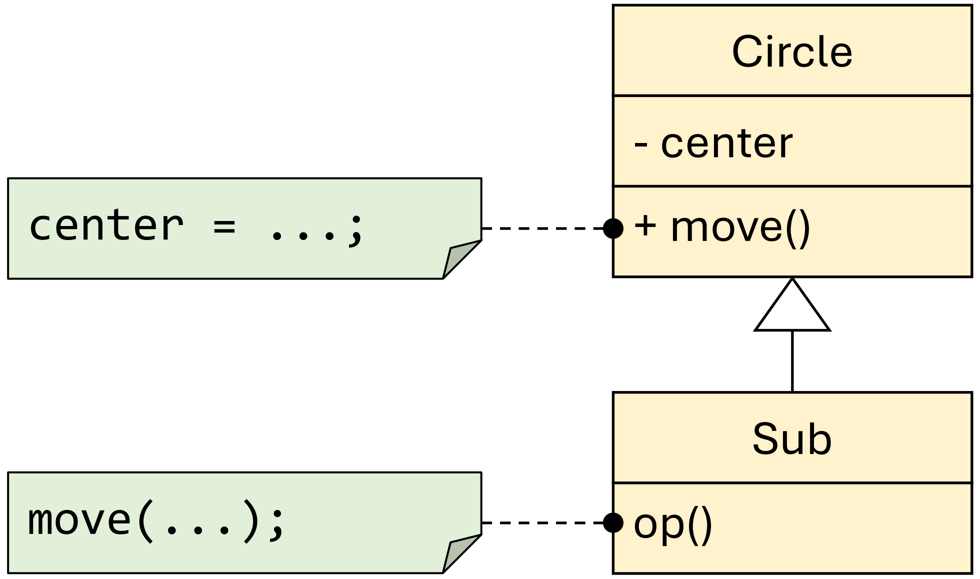
```
#include<stdio.h>
extern vector v;
…
int method() {
    v.doSomething()
    ;
    …
    return i;
}
```

Dependency on Inherited Attributes

Rather use getter and setter on inherited attributes instead of direct access.

This is in accordance with the strong law of Demeter.





Simple Design Principles: Understandability

- Principle of separate understandability: Each module should be understandable on its own.
- Rule of Explicitness: Make implicit assumptions explicit.
- Principle of least surprise: Develop modules in a way that others would expect it to function.
- Easy to use and hard to misuse: The obvious way to use a module is the right way. Misusing the module takes considerably more effort.

Simple Design Principles: Simplicity

- Murphy's Law: "*Whatever can go wrong, will go wrong*". Keep the system simple, eliminate sources of exceptions.

- Keep it simple, stupid (KISS): Prefer simple, ugly solutions to complex ones that no one understands.

- More is more complex: Less complex systems are easier to maintain.

- Need to know: Only allow access to a resource if it is really necessary.

Simple Design Principles: Redundancy

- Don't repeat yourself (DRY): Do not introduce unnecessary redundancies to your codebase.

  There are reasons to have redundancy, e.g., Integrity and Performance.

- Generalization Principle: Generalize solutions so that they solve multiple related problems.

- Locality of Change: You only need to change one thing in one place.

Questions?

Summary

In today's lecture: Got to know (OO) modeling principles.

About object behavior:

- Liskov
- Design by Contract.
- Behavior Protocols

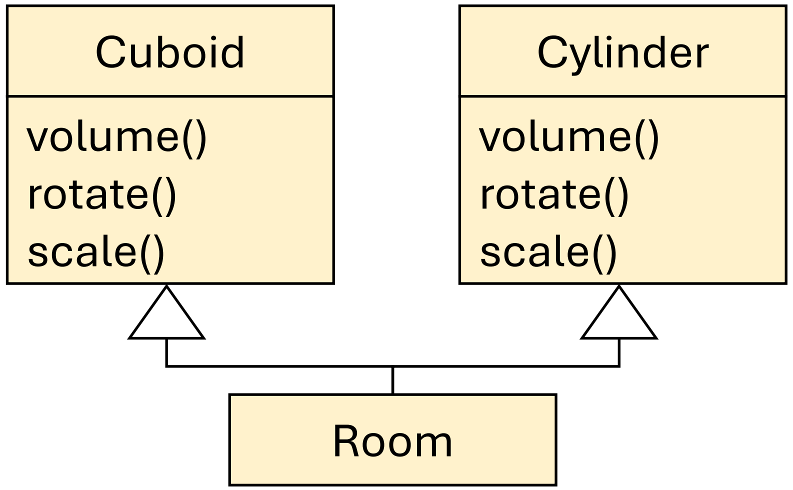
About dependencies:

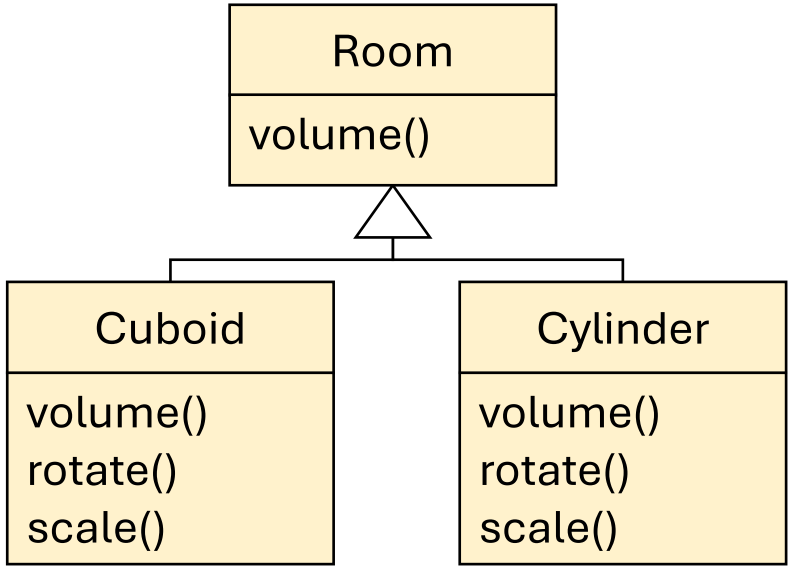
- SO(L)ID.
- Cohesion, Coupling, KISS, DRY and others.
- How NOT to model software.

Bad Object Design

Wrong Application of "Is A"

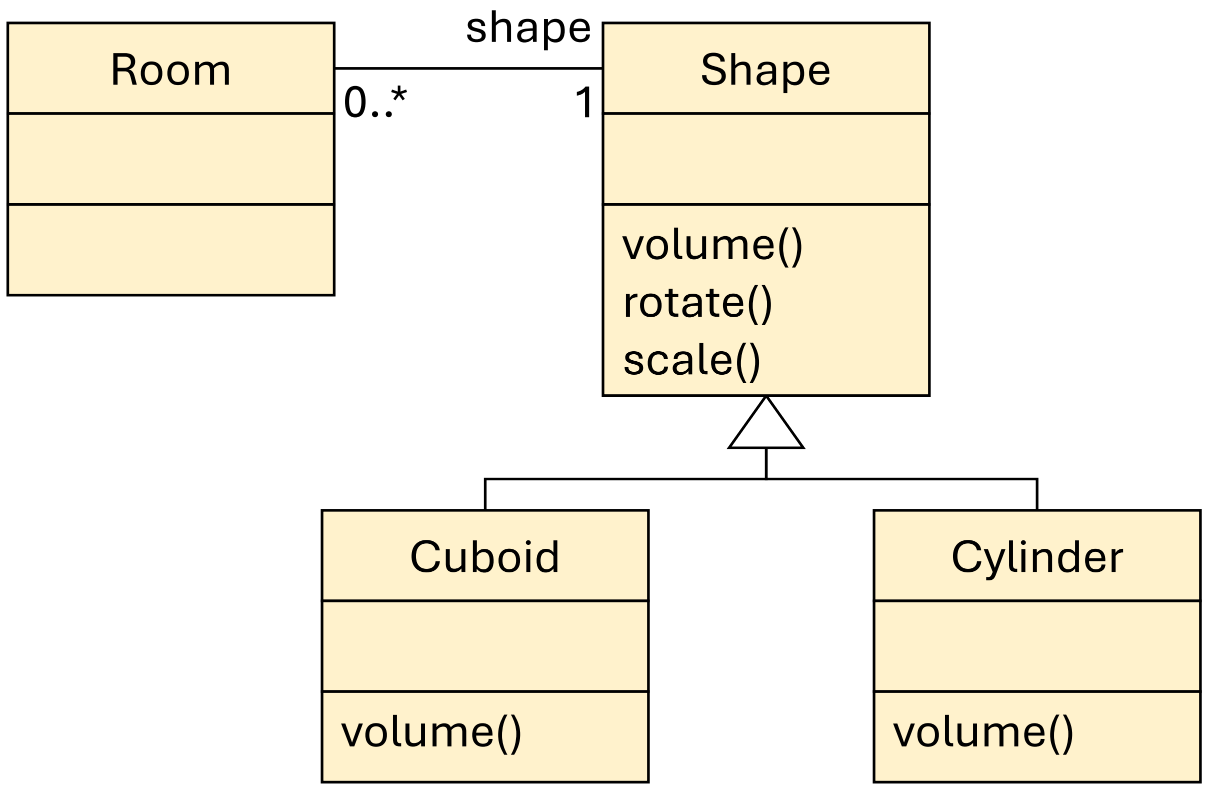
- Problem: Illogical substitution. Rotating, scaling, etc. are not sensible for rooms.
- Problem: Ambiguous substitution
  - How can Room be a cuboid and a cylinder?
  - From whom does Room inherit volume()?
- Problem: Lost substitution
  - Scale, rotate,... not visible from Room.





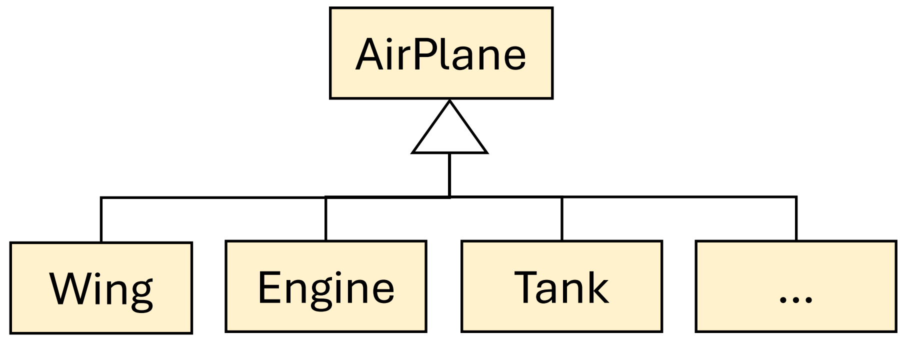
Wrong Application of "Is A"

Solution: Room has a shape.

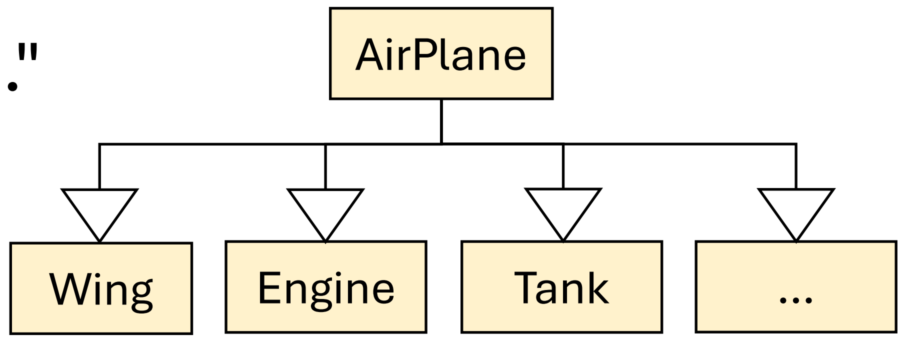


Confusion of "Is A"

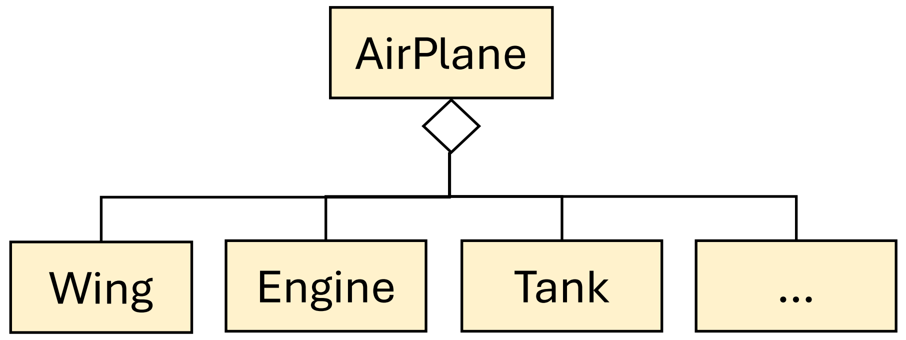
- "Wing is a kind of airplane"



- "An airplane is wings, an engine, a tank,..."

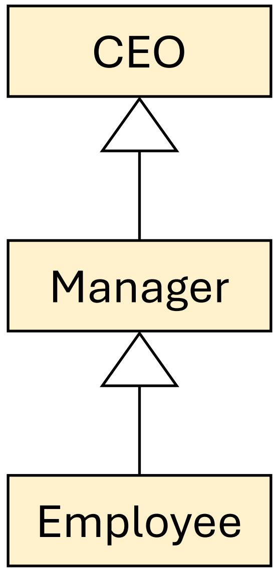


- Actual intention:

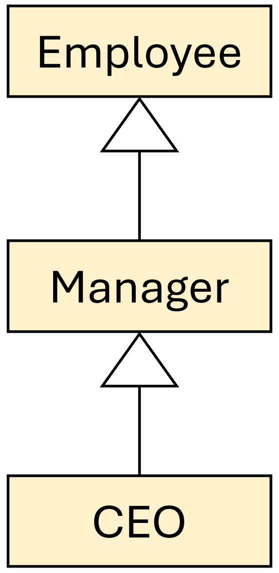


Inverted Inheritance Hierarchy

Problem: Tendency to implement inheritance according to orgchart.



Actual: Inverted.

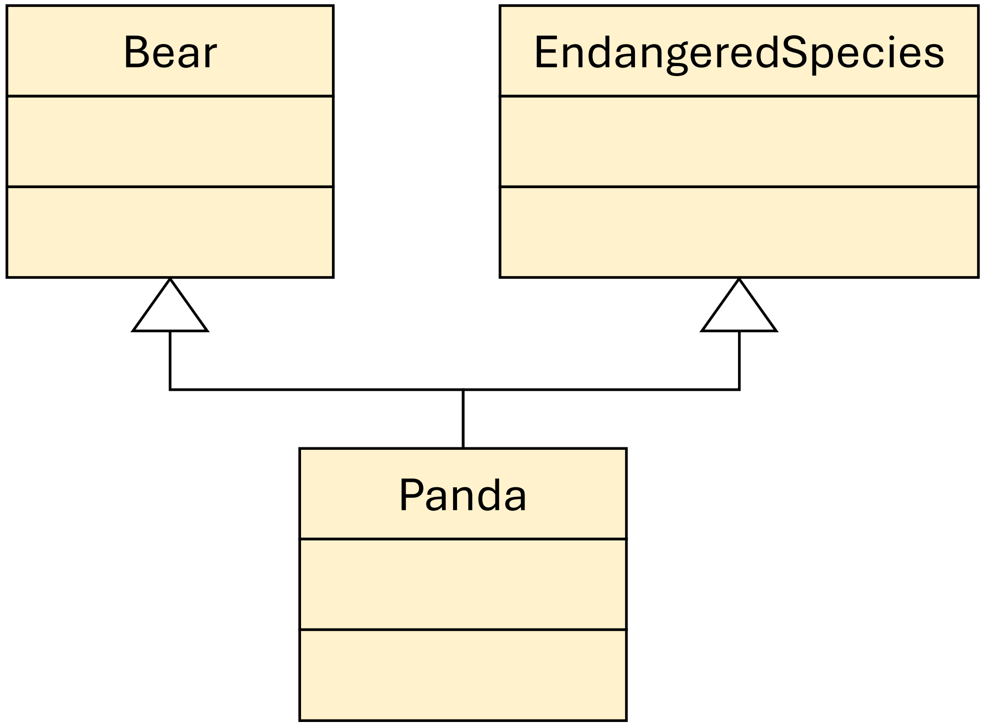


Confusion of Instance and Class

Natural language often does not differentiate between classes and instances, e.g., "student" as class and "student" as a single instance, or "iPhone" for the whole product line or your "instance".

```
Panda miou = new Panda();
EndangeredSpecies es = miou;
```

Is a specific panda (miou) really an endangered species?
In actuality, the type Panda is an instance to the EndangeredSpecies class.



Mixed-Instances Cohesion

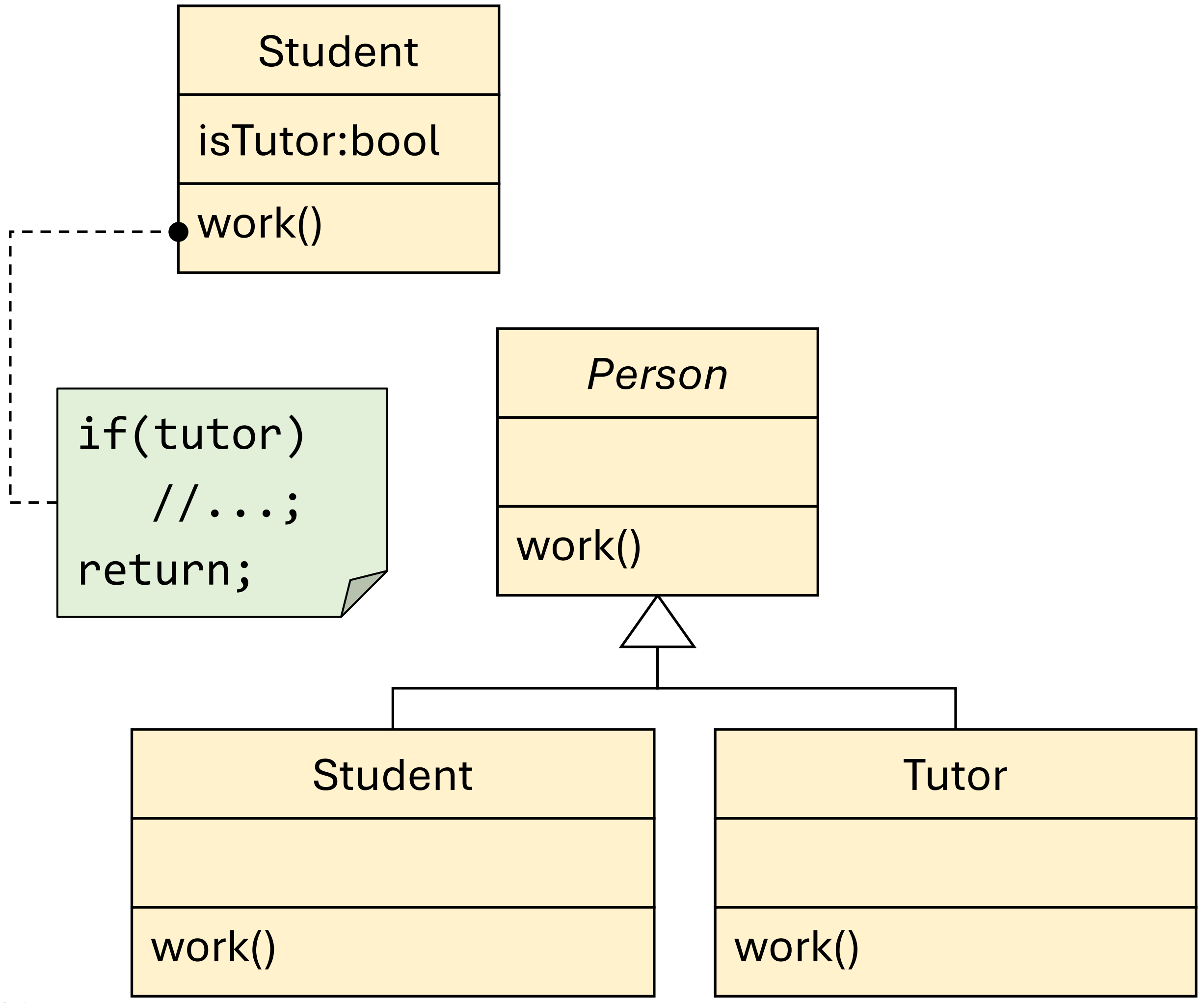
Problem: A class has some features that are undefined for some of its instances.

Consequences:

- Dependence on conditionals.
- Hard to maintain.

Solution:

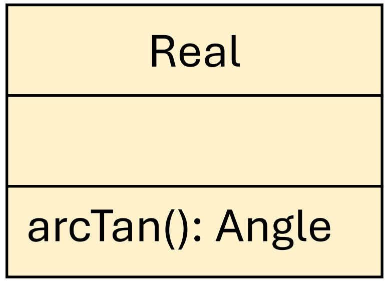
- Separation of Concerns.
- Split class.
- Walter Hürsch: "*Should superclasses be abstract?*"



```
if(tutor)
   //...;
return;
```

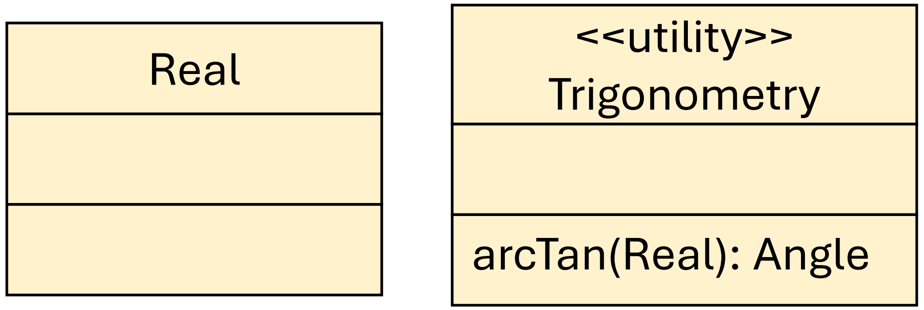
Mixed-Domain Cohesion

Problem: A class contains elements that directly encumber the class with an extrinsic class <u>of a different</u> domain.



Famous saying: "Should a thing know how to print itself?"

Solution: Extract extrinsic parts to own classes or to classes they encumber.

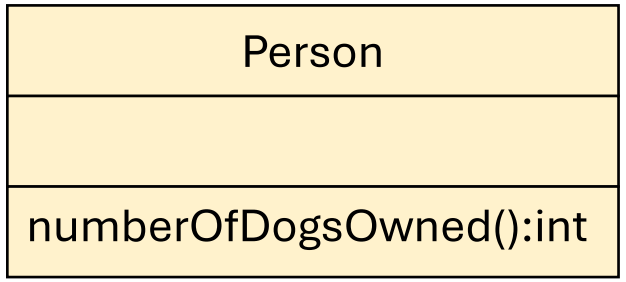


Mixed-Role Cohesion

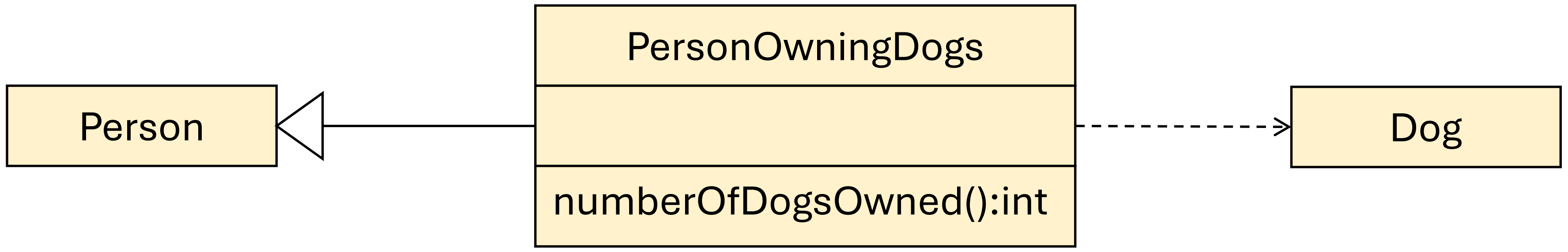
Problem: A class contains an element that directly encumbers the class with an extrinsic class that lies in the same domain as itself.

Is reusing this class sensible?

Why not also incorporate cats, frogs,... and all their getter and setter.



Solution:



There are more Types of Dependencies

- Conventions: `if(order.accountNumber > 0)` // what does that mean?
- Value: Keeping stored values consistent.
- Algorithm assumptions: Inserting order = printing order, Searching and inserting in hash tables use the same hash function,...
- Names:
  - Existence: `int i; i := 7` // depends on declaration of `i`
  - Non-existence: `int i; int j;` // depends on not renaming `i` to `j` or declaring `j` twice.
- Type: `int i; i := j;` // types of `i` and `j` must be compatible.
- Relative Position: `method1` must be declared before `method` 2.

Questions?