Secure OOP with Java

Lecture Unit - 05

Claudia Maderthaner <claudia.maderthaner@fh-hagenberg.at>

The Object-Oriented Paradigm

Object-Oriented Paradigm

- Based on the concept of **objects**,
- which are instances of **classes**
- that encapsulate **data** and **behavior**

Real World

⇩

Model

⇩

Implementation

"divide et impera - divide and rule"

Object-Oriented Decomposition

- Individuals (= **objects**) work together to accomplish a common task.
- Therefore, they communicate with each other (by passing **messages**).
- Objects with common attributes and behaviour are described through **classes**.

Objects

An object is a thing in the real world.

- Identity
- State
- Behaviour
- Lifecycle

Identity

The identity from an object is its being distinct from any other object, regardless of the values of the object's properties.

State

State is

- what objects are
- what objects have

→ object properties

Behaviour

Behavior is

- what objects do
- which messages a object understands

→ object methods

Lifecycle

Object creation

through a constructor call

```
Person person = new Person("Jane", "Doe");
```

Object destruction

through garbage collector

All object which are no longer referenced in the running programm are eligible for garbage collection.

Classification

- Detection of patterns among characteristics

Classes

- Blueprints for classes
- Classes contain the actual code

Compile- vs Runtime

Compile-Time

- Source code to byte code
- Check syntax and semantics
- Detect errors without program execution
- Bugfixing

⇒ Classes

```
javac MyClass.java
```

Runtime

- Time between start and end of running code in runtime environment
- Actually execute the code
- Detect errors after execution
- Fixing errors means going back to code

⇒ Objects

```
java MyClass
```

The Pillars of Object-Orientation

Four Pillars of OOP

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Abstraction

- Simplify reality
- Reduce complexity
- Focus on characteristics relevant in a specific **context**
- Only show essential features

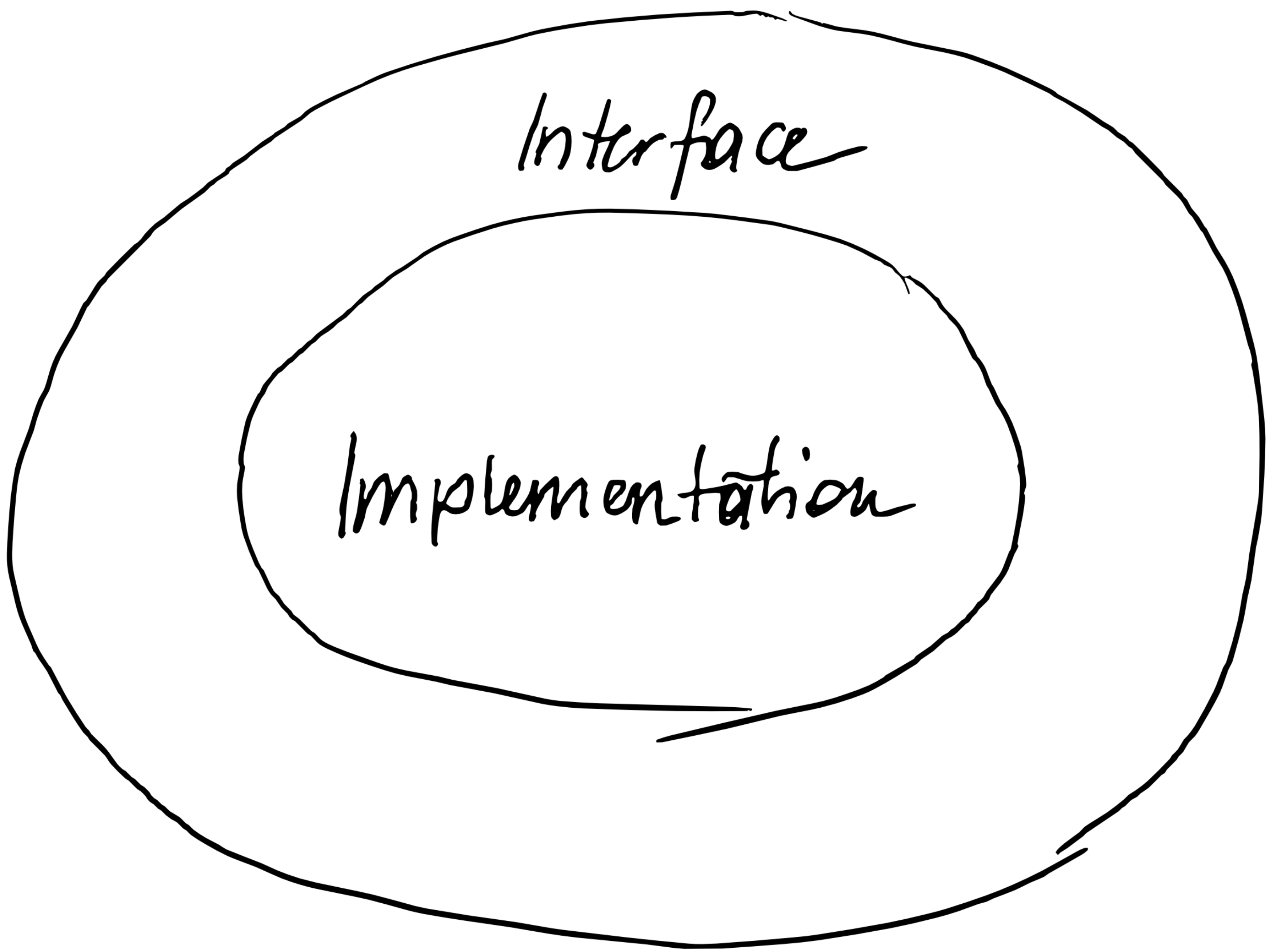
Think of a person.

What characteristics are relevant in following contexts

- School
- Webshop
- Health Insurance Company

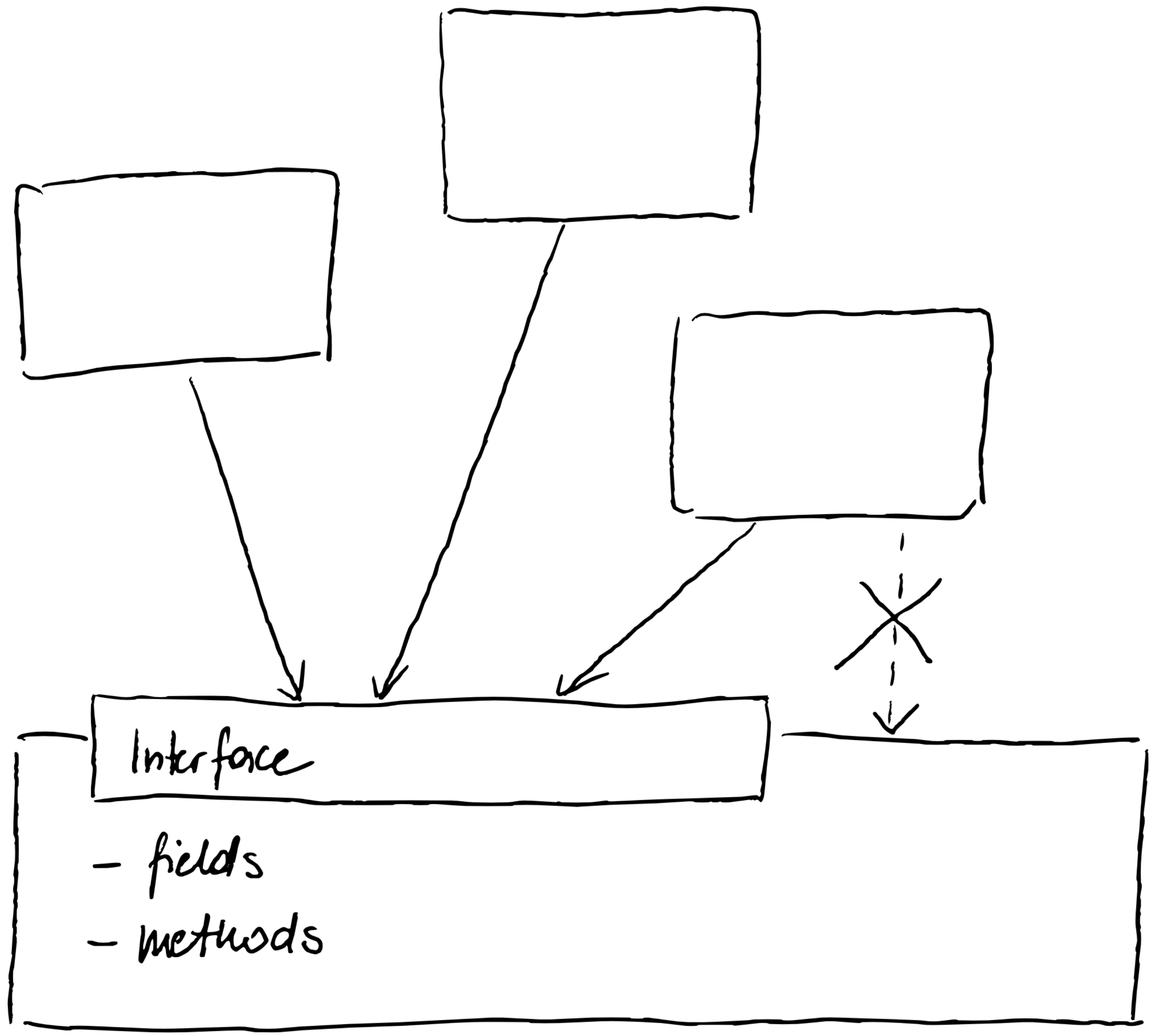
Encapsulation

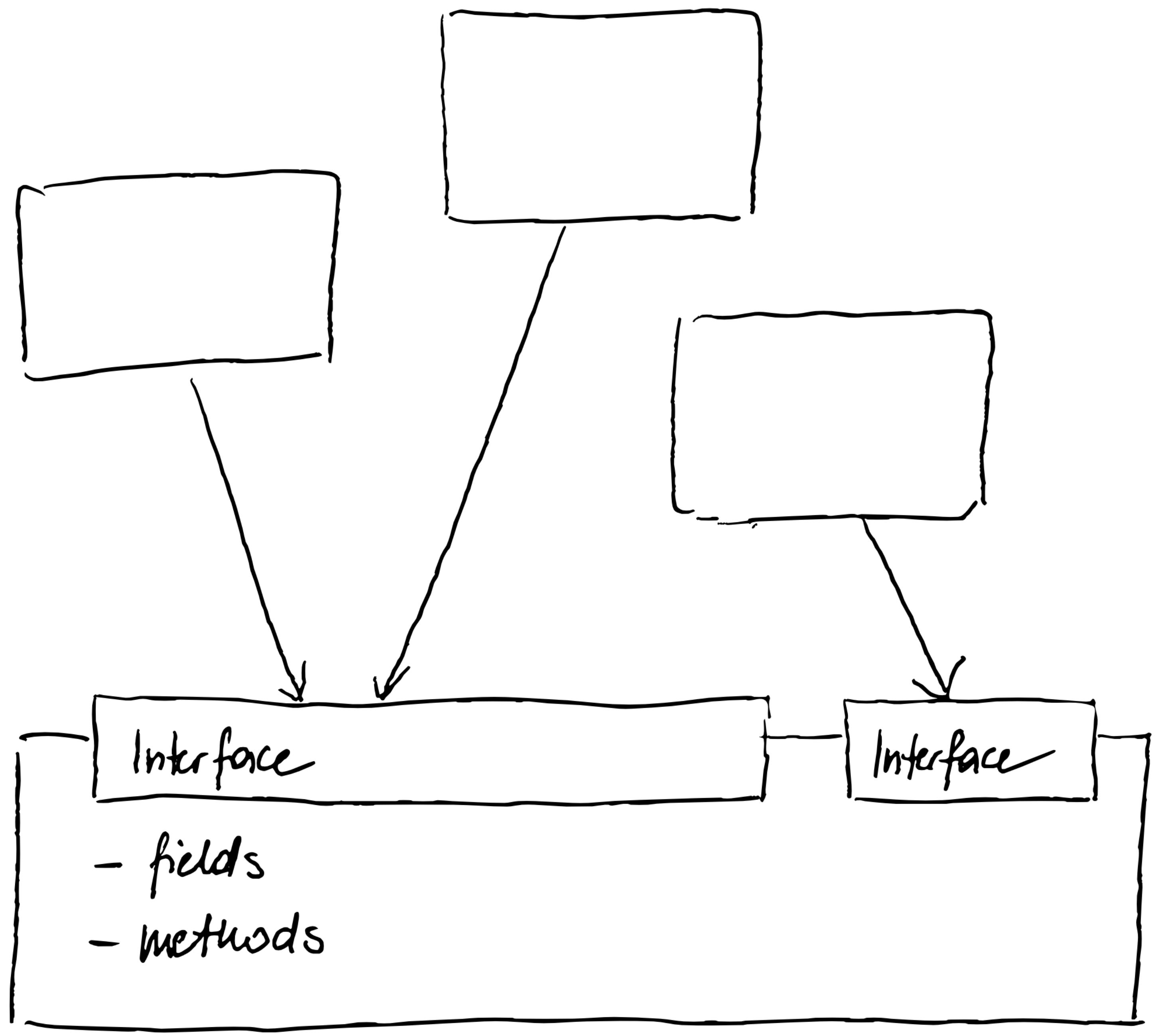
- Hide the inner workings
- Data does not flow freely
- Data is wrapped in objects
- Objects bundle data with the related behaviour
- There are restrictions if and how data may be accessed


Interface
Implementation

Information Hiding

- Prevent certain aspect of a class to be accessible
- Provide a stable interface
- Protect the remainder of the implementation


Interface
- fields
- methods

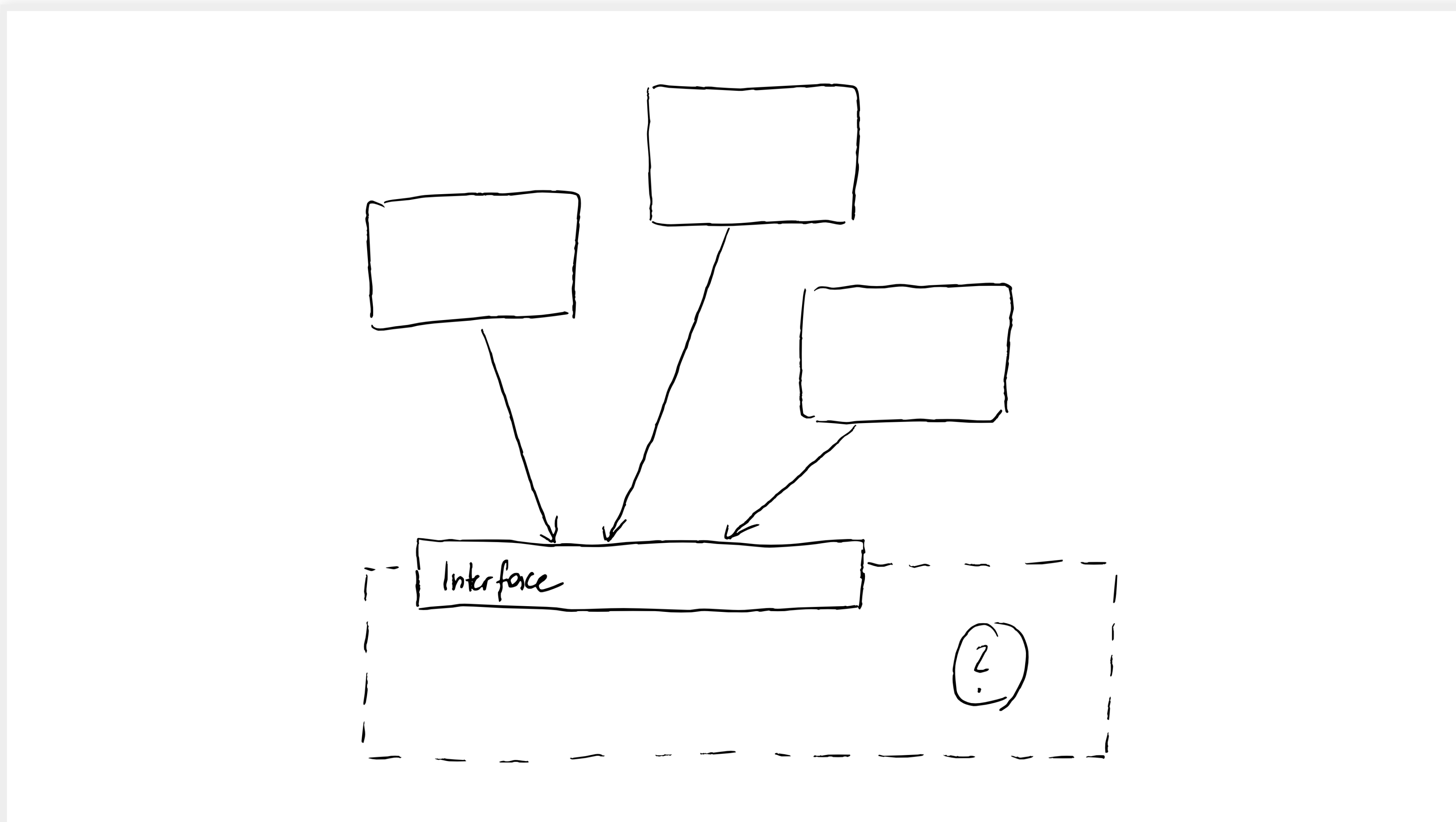

Interface
Interface
- fields
- methods

```java
public class Account {
    public double balance;
}
```

```java
Account accountA = new Account();
accountA.balance = 100.0;

Account accountB = new Account();
accountB.balance = 20.0;

//correct transfer 20 from A to B
accountA.balance -= 20;
accountB.balance += 20;

// incomplete transfer 60 from B to A
accountB.balance -= 60;
```

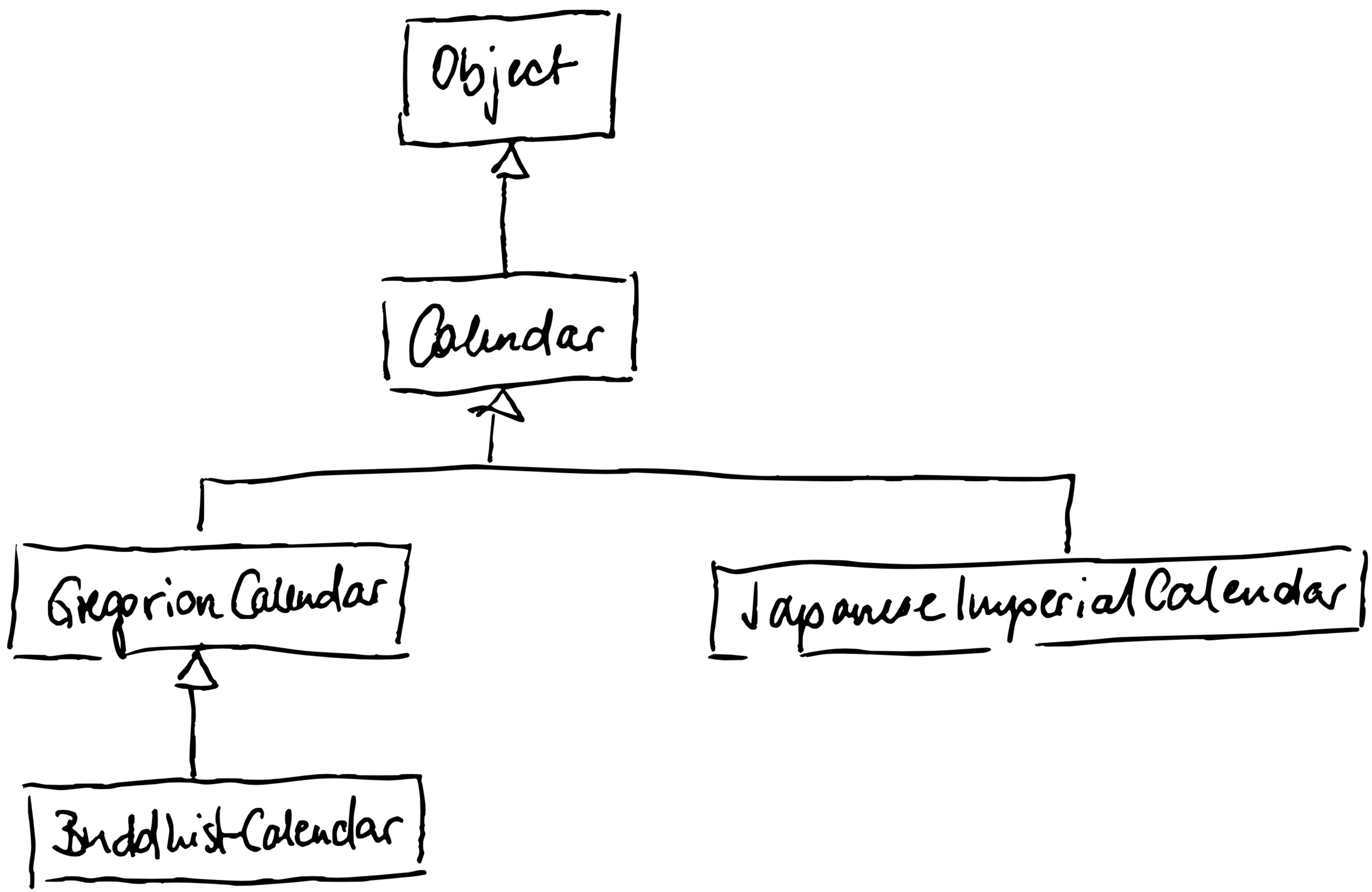
```java
public class Account {
    private double balance;

    public Account(double balance) {
        this.balance = balance;
    }

    public void deposit(Account from, double amount) {
        if (from.balance >= amount) {
            this.balance += amount;
            from.balance -= amount;
        }
    }

    public void withdraw(Account to, double amount) {
        to.deposit(this, amount);
    }
}
```

Programming against Interfaces



Inheritance

- Expresses a "is a" relation
- Facilitates code reuse (but it is by far not the only way to accomplish code reuse)


Object
Calendar
GregorianCalendar
JapaneseImperialCalendar
BuddhistCalendar

Polymorphism

Static Type

- Type of declaration (= variable)
- May not change during runtime
- Assures a certain interface

Dynamic Type

- Type at runtime (= object)
- May change during runtime
- Concrete behaviour may change

"one name with many forms"

Static vs. Dynamic Binding

- Binding determines which implementation is executed when calling a method
- Static binding happens during compilation (compile-time polymorphism)
- Dynamic bindung is done at runtime (runtime polymorphism)

```java
public interface B {
    void c();
}
```

```java
public class A implements B {
    public A() { ... }
    public A(int x) { ... }

    public void c() { ... }

    public void s() { ... }

    public void m() {
        c();
        s();
    }

    public static void z() { ... }
}
```

```java
class TestA {
    public static void main(String[] args) {
        B myB = new A();
        myB.c();

        A myA = (A) myB;
        myA.s();
        myA.m();

        A.z();
    }
}
```

```
$ javap -c binding/TestA.class
Compiled from "TestA.java"
public class binding.TestA {
  public binding.TestA();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: new           #7                  // class binding/A
       3: dup
       4: invokespecial #9                  // Method binding/A."<init>":()V
       7: astore_1
       8: aload_1
       9: invokeinterface #10,  1           // InterfaceMethod binding/B.c:()V
      14: aload_1
      15: checkcast     #7                  // class binding/A
      18: astore_2
      19: aload_2
      20: invokevirtual #15                 // Method binding/A.s:()V
      23: aload_2
      24: invokevirtual #18                 // Method binding/A.m:()V
      27: invokestatic  #21                 // Method binding/A.z:()V
      30: return
}
```

Object-oriented Analysis and Design

Analysis

- Generalisation
- Specialisation

Design

- Inheritance
- Association
  - Aggregation
  - Composition

Analysis

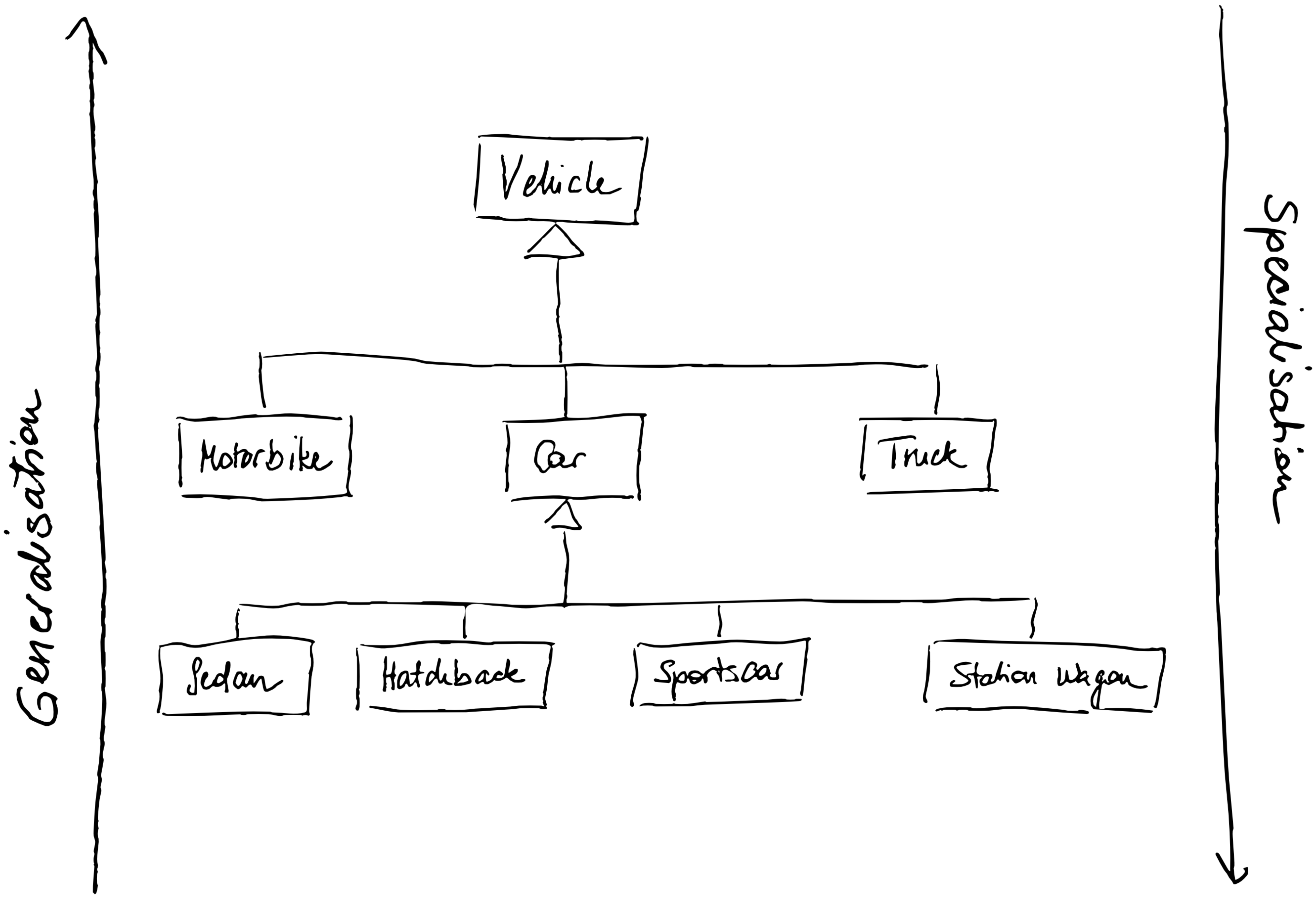
- Identifying the objects and classes needed to solve a problem
- Developing software specifications
  - Object model
  - Object interaction

Generalisation

- Capturing similarities between objects
- Capturing similarities between classes

Specialisation

- Capturing differences among objects in a class


Generalisation
Vehicle
Motorbike
Car
Truck
Sedan
Hatchback
Sportscar
Station wagon
Specialisation

Design

- Applying object-oriented concepts and principles
- Decomposing a problem in smaller, more manageable parts
- Designing classes and objects who represent those parts and interact with each other to solve the problem

Design

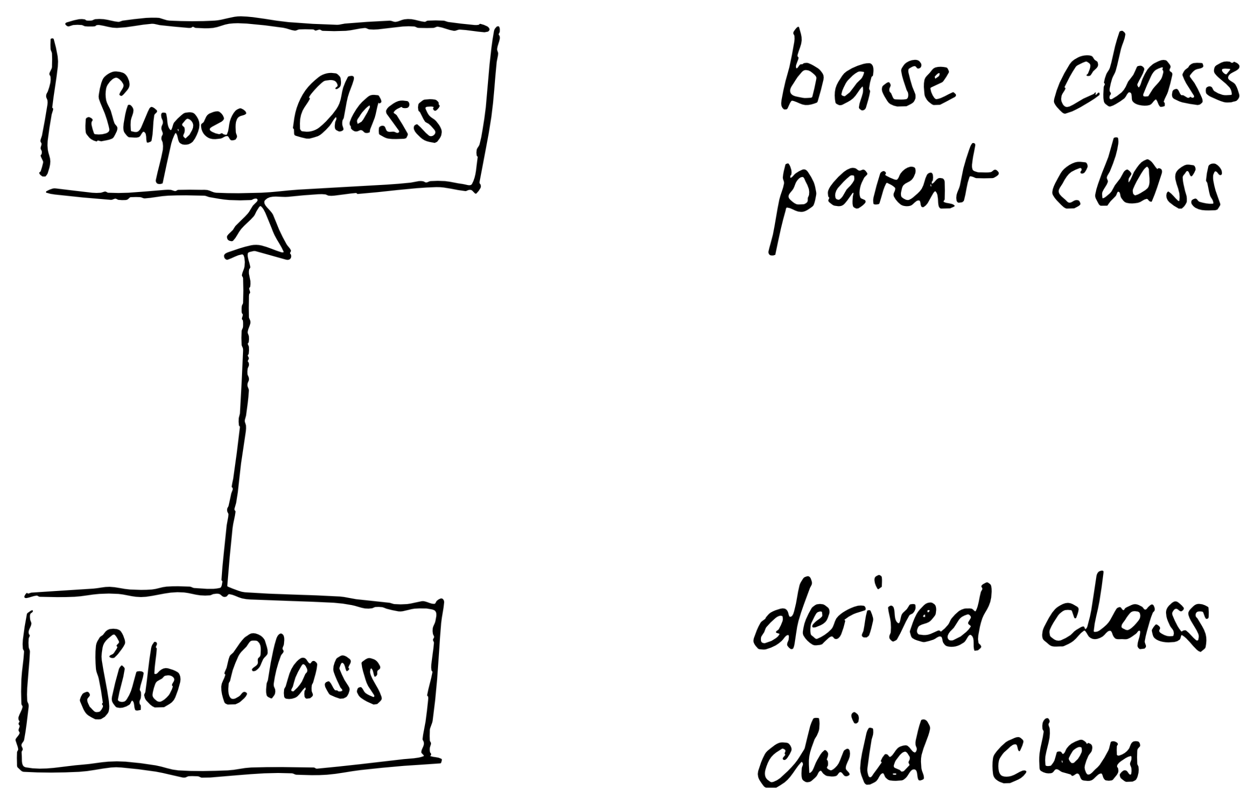
- Map the analysis model onto implementing classes
- Identify constraints
- Design interfaces

Object Relations

- Inheritance
- Association
- Aggregation
- Composition

Inheritance

- "is-a" relationship



Association

- Any kind of relationship
- Objects "know" each other
- Associations may be directed

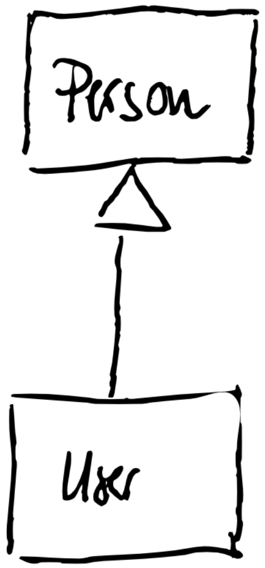
Aggregation

- "has-a" relationship
- Related objects may have different lifecycles
- Assemble parts to a bigger construct
- A member may be related to different owners

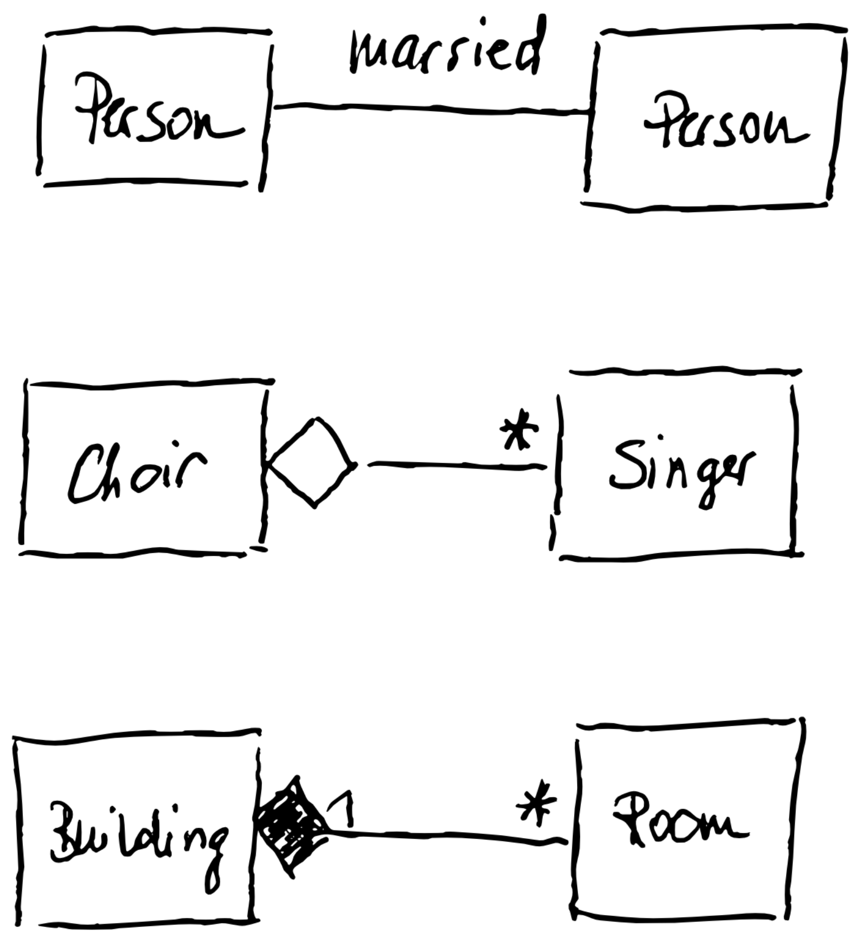
Composition

- "belongs-to" relationship
- Lifecycle is tied to owner object
  ⇒ if the owner object is destroyed, all members will be destroyed to
- A member can only belong to one owner

is - a



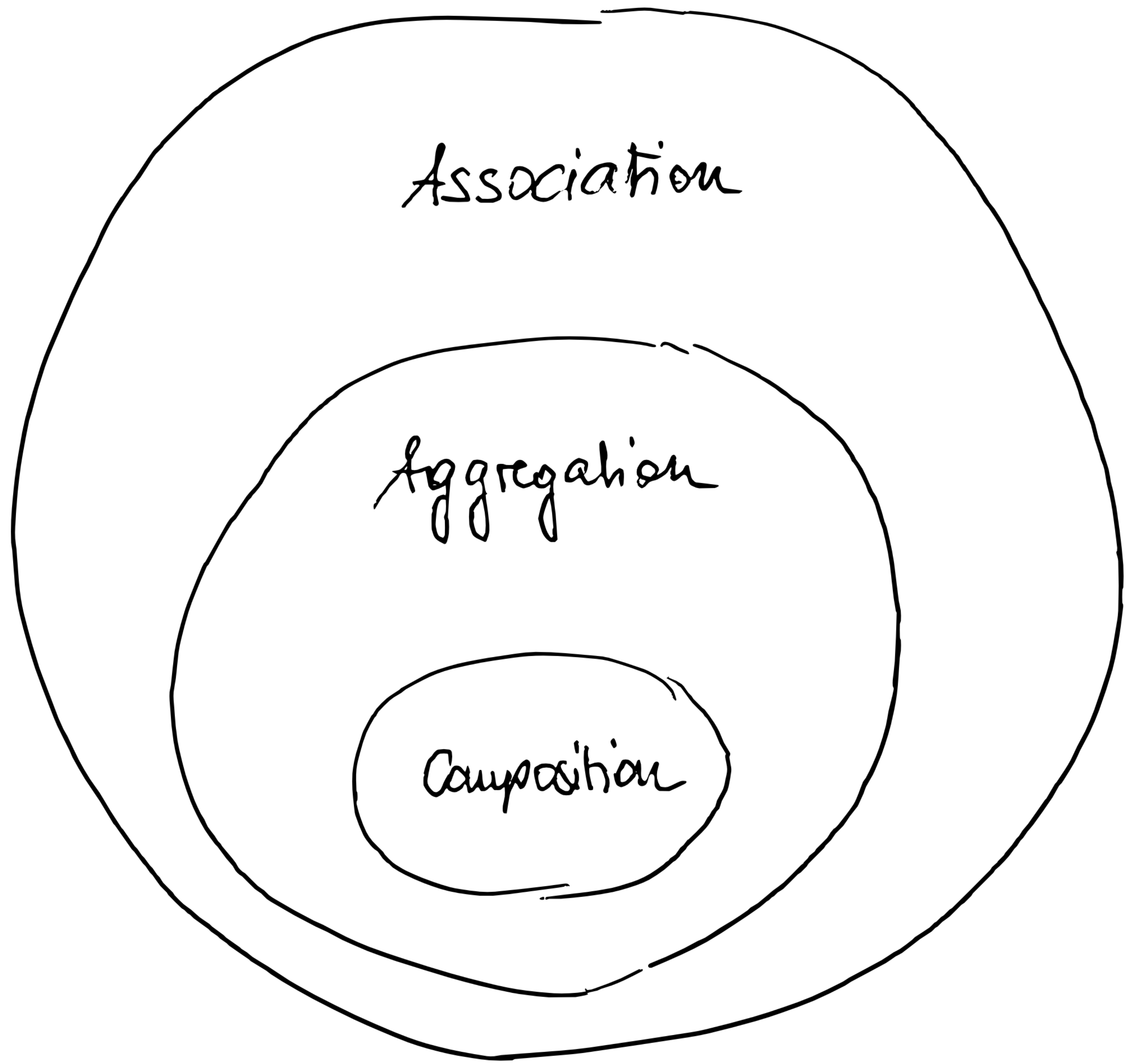
has - a



Association

Aggregation

Composition


Association
Aggregation
Composition

SOLID Principles

- **S**ingle Responsibility
- **O**pen/Closed
- **L**iskov Substitution
- **I**nterface Segregation
- **D**ependency Inversion

Single Responsibility

- A class should only have one responsibility.
- It should only have one, and only one, reason to change.

Benefits

- Easier to understand
- Simpler and more efficient testing
- Lower coupling
- Organization

Open/Closed

- Open for extension
  - Subclasses/implementing classes may override or enhance existing behaviour
  - Add new functionality without changing (or breaking) the existing code
- Closed for modification
  - The original interface is stable and will not change

> The principle does not apply to fixing bugs.

Liskov Substitution

"If class B is a subtype of class A, it should be possible to replace A with B without disrupting the behaviour of the program."

Design by Contract

- Preconditions cannot be strengthened in the subtype.
- Postconditions cannot be weakened in the subtype.
- Invariants must be preserved in the subtype.

Interface Segregation

- Split large interface into several smaller ones.
- Classes should not be forced to depend upon interfaces that they not use
- Implementing classes only need to be concerned about the methods of interest to them

Dependency Inversion

- Decoupling of modules
- High-level modules should not depend on low-level modules
- Both should depend on abstractions
- Abstractions should not depend on details.
- Details should depend on abstractions.

Good Practices

Creating OO Models

1. Identify the objects
2. Organize the objects
3. Identify the object interaction
4. Describe the properties of the objects
5. Describe the behavior of the objects

Class Design Hints

- Always keep data private
- Always initialize data
- Don't use too many basic types in a class
- Not all fields need individual field accessors and mutators
- Break up classes that have to many responsibilities
- Make the names of your classes and methods reflect their responsibilities
- Prefer immutable classes

Inheritance Design Hints

- Place common operations and fields in the superclass
- Don’t use protected fields
- Use inheritance to model a "is-a" relationship
- Don’t use inheritance unless all inherited methods make sense
- Don’t change the expected behavior when you override a method
- Use polymorphism, not type information
- Don’t overuse reflection

Benefits of Object-oriented Paradigm

- Transform complex scenarios in the real world to simpler models
- Reuse of components

Contact

Moodle Discussion Board

claudia.maderthaner@fh-hagenberg.at