Fachhochschule Nordwestschweiz
Hochschule für Technik

Software Architecture

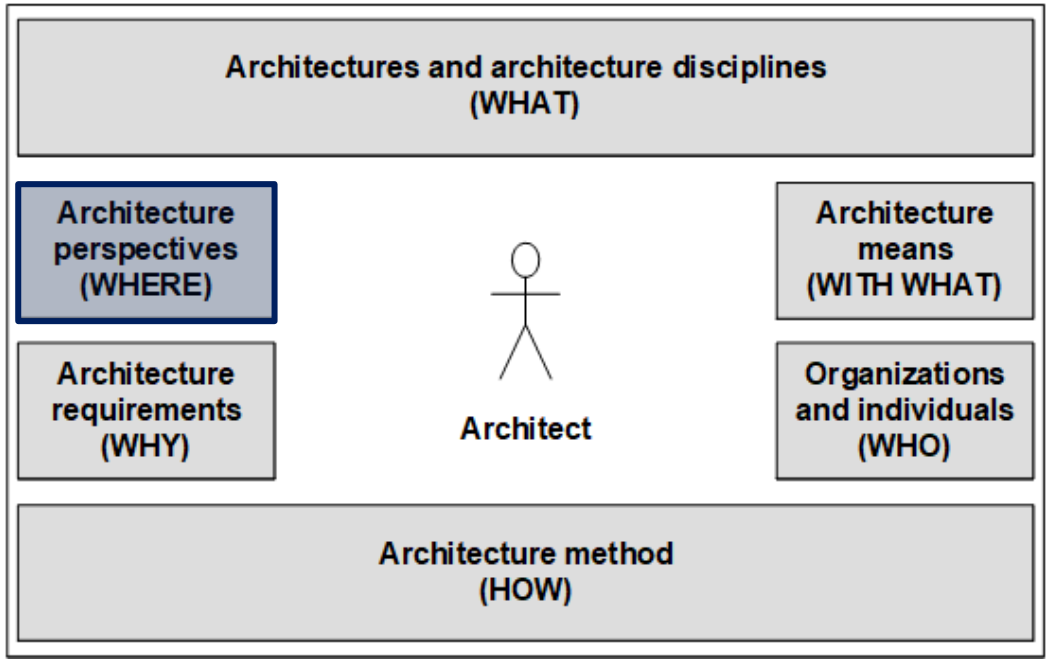
Logical View (object-oriented design)

BSc

Ingo Arnold

Lecture Opening
Architecture Orientation Framework

Architecture WHERE deals with the **representation of architecture** in general as well as software architecture in particular [Vogel, Arnold et al 2011].



Lecture Opening
Motivation

When designing and programming software, you edit components that are mainly visible in the development View. However, the **actual goal is the effect the system causes when it is executed**.

In the object oriented paradigm this **effect is produced by the interaction of objects**. Kruchten's **logical view** is the vehicle supporting a respective perspective.

Software structures are to be modelled in close relation to those of the subject matter domain.

Goal is a better and more understandable representation of the inner logic of software systems. In principle terms this is stated by the ***low representational gap* principle**.

Lecture Opening
Learning Objectives

You …

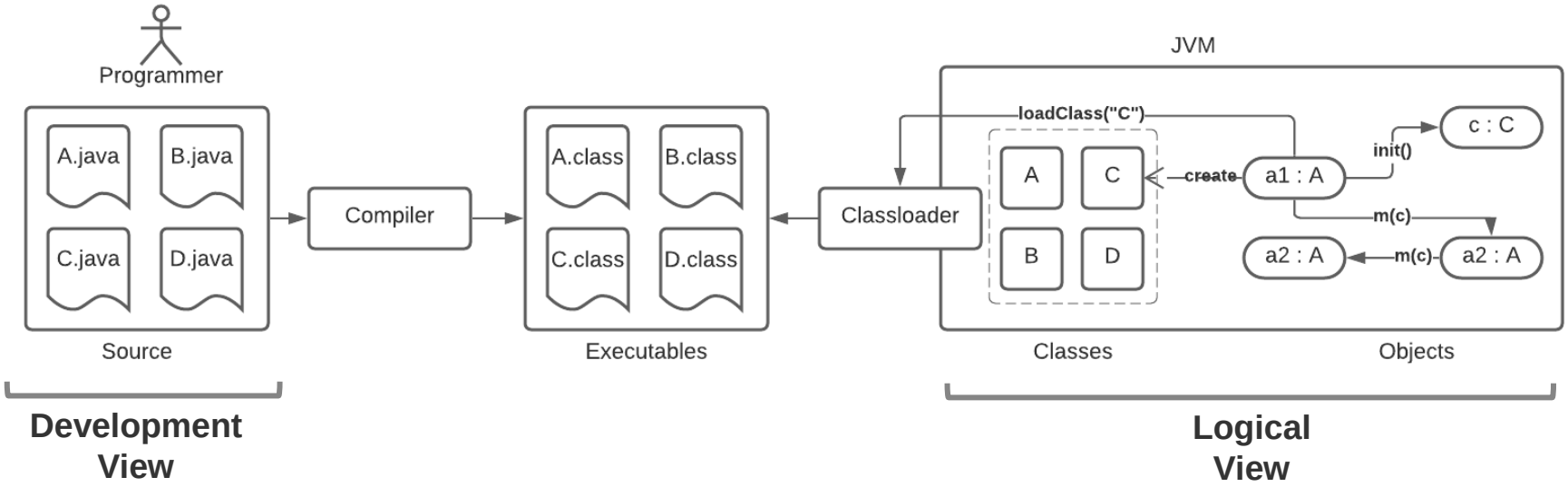
- can distinguish between the Kruchten's development view and logical view and explain how each of them influence the behavior or effect of software.
- will be able to explain the principle of the low representational gap (LRG) and the basic features of object-oriented analysis and design (OOA, OOD).
- Are able to describe an object-oriented algorithmic structure for a functional requirements scenario.

Lecture Agenda

- Development View to Logical View
- Low Representational Gap
- Behaviour and Methods in the Logical View

Development View to Logical View
Overview

The path from programming to the effect of executed software comprises several intermediate steps, which we are not always fully conscious of. Therefore it is worthwhile to mind the fundamental procedure (i.e., from the development view to the logical view).



Development View to Logical View
Overview

A **software system aims at causing an effect** (e.g., in the computer's memory, at the user interface level, or at interfaces to other software).

This effect is caused **by the behavior of objects** in memory and their interaction.

The computer (i.e., java virtual machine) has embedded **a factory for such objects**. It consists of the *classLoader* and class descriptors with methods for creating new objects of a respective class (e.g., *new*, *forName*).

The object factory **uses construction blueprints (i.e., *.class* files) in factoring objects**. The **compiler creates *.class* files from the source files** (i.e., *.java* files) developed during programming.

To be able to estimate or consciously control the effect caused at runtime, you must understand the run time structures and create the design-time structures (classes) in such a way create that the necessary object structures can be produced.

You therefore **begin the design of systems with the *logical view***.

Lecture Agenda

- Development View to Logical View
- Low Representational Gap
- Behaviour and Methods in the Logical View

Low Representational Gap
Overview

The gap between different representations of real-world objects, models and software should be as small as possible. The object oriented paradigm uses object-oriented models and finally object-oriented programs in order to achieve a gap as small as possible.

Real World	Object Model	Class Model	Code Model
 Four wheels are mounted on the car: front left, front right, rear left, rear right.	?	?	?

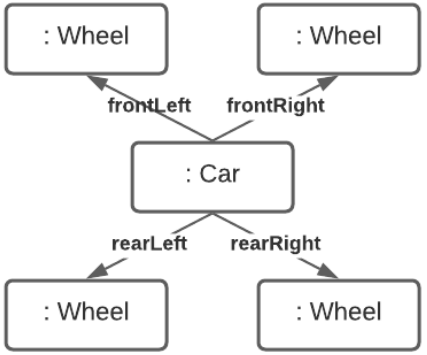
Low Representational Gap
Overview

The gap between different representations of real-world objects, models and software should be as small as possible. The object oriented paradigm uses object-oriented models and finally object-oriented programs in order to achieve a gap as small as possible.

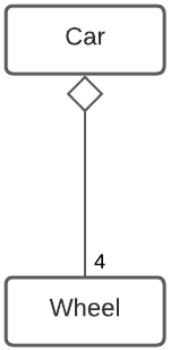
Real World

Four wheels are mounted on the car: front left, front right, rear left, rear right.

Object Model



Class Model



Code Model

```
public class Car {
   Wheel frontLeft, frontRight;
   Wheel rearLeft, rearRight;
}
public class Wheel { }
```

Low Representational Gap
Overview

The LRG principle requires the **representation of the real world in the form of an object-oriented model** (i.e., create a model of the real world).

The second step is to try to **represent the resulting model 1:1 in software**. This cannot always be realized down to the last detail, but the aim serves as a reasonable heuristic. Ideally, this is a process that can be carried out methodically and reproducibly.

Following the LRG principle …

- the objects, their data or states and behavior correspond as closely as possible to real world objects.
- between real objects, data and program structures and finally code there is only an as small as possible gap.
- object and class structures and the code can be understood more systematically and efficiently. This helps to avoid errors and makes it easier for other people to familiarize themselves with the solution later on.

Low Representational Gap
Overview

In order to adopt the LRG principle, the **design must be guided primarily by the structures of the subject matter domain** and less by technical structures.

There are various alternative approaches, like entity-relationship modelling (**ERM**) in relational database design, object-oriented analysis with object-oriented design (**OOA, OOD**), domain-driven design (**DDD**), model-driven architecture (**MDA**).

All approaches have in common that you initially describe the real world, in which the software is to cause something, with a model. The structures of the software are then oriented to the structures of this model.

Low Representational Gap
Exercise

*5-BSc_SWA_ERMvsOOD

Lecture Agenda

- Development View to Logical View
- Low Representational Gap
- Behaviour and Methods in the Logical View

Behaviour and Methods in the Logical View
Overview

In the previous slides, we followed a **data-driven design**. This means that the modelled entities and their relationships were merely derived from data structures.

Operations, performed by individual objects, **were completely disregarded**. This is despite the fact that the encapsulation of data and operations is a pivotal feature of object-oriented programming.

A simple design strategy is to provide (for each object) all operations that seem useful for manipulating the data encapsulated in the object. However, such a **design is entity-driven**. Decisions are made with respect to each individual entity (or more precisely, each entity type) in isolation.

This works as long as the operations are more or less trivial. If **more complex operations** are expected (e.g., performing complex calculations, or having to change a set of attributes together and consistently), it often makes more sense to be **guided by what other objects (in an interaction scenario) require**.

Behaviour and Methods in the Logical View
Overview

Therefore, start from **scenarios that consider the system as a whole** and describe the cooperation between objects that is necessary for them to jointly realize a given scenario.

Such design is **scenario-driven**. It only considers the cooperation between objects and not the internal realization of corresponding methods. Thus, it is a **top-down approach** that abstracts from the details, such as the internal implementation of the objects.

Once it is known what operations the objects must provide because they are called to implement the scenarios, one can directly **derive the publicly visible elements** (e.g., methods, parameters, expected outcomes) from these objects.

The internals of the objects or classes can be left to **detailed design** thanks to **information hiding**.

Behaviour and Methods in the Logical View
Exercise

*i.1-BSc_SWA_GamePlatform
5.2-BSc_SWA_DomainModel
i.2-BSc_SWA_GamePlatform

Bibliography
Lecture

[Vogel, Arnold et al 2011]
Vogel, Oliver; Arnold, Ingo; Chugtai, Arif; Kehrer, Timo, *Software Architecture: A Comprehensive Framework and Guide for Practitioners*, Springer Science and Business Media, Berlin Heidelberg, 2011

Questions