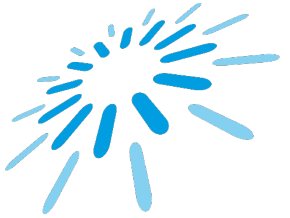
Software Engineering

Prof. Dr. Jóakim von Kistowski



TH Aschaffenburg
university of applied sciences

Check-In

Why is it important to deal with design?

Disciplines in software engineering

Basic topics

Configuration management | Documentation |
Knowledge management | People in the SWE process and digital ethics | Tools

Development

Requirements
- Context analysis
- Requirements Engineering

Design
- Course granular design: Architecture
- Detailed design

Implementation

QualityMgt.

Quality assurance and testing
- Test, inspection, metrics

Processes and procedure models
- Improvement, process model, maturity levels

Evolution
- Roll-Out
- Operation
- Maintenance
- Further development
- Reuse
- Reengineering
- Change management

Management
- Strategy
- Economy
- Team
- Dates
- Risks
- Customer, client/contractor
- Innovation

TH Aschaffenburg
university of applied sciences

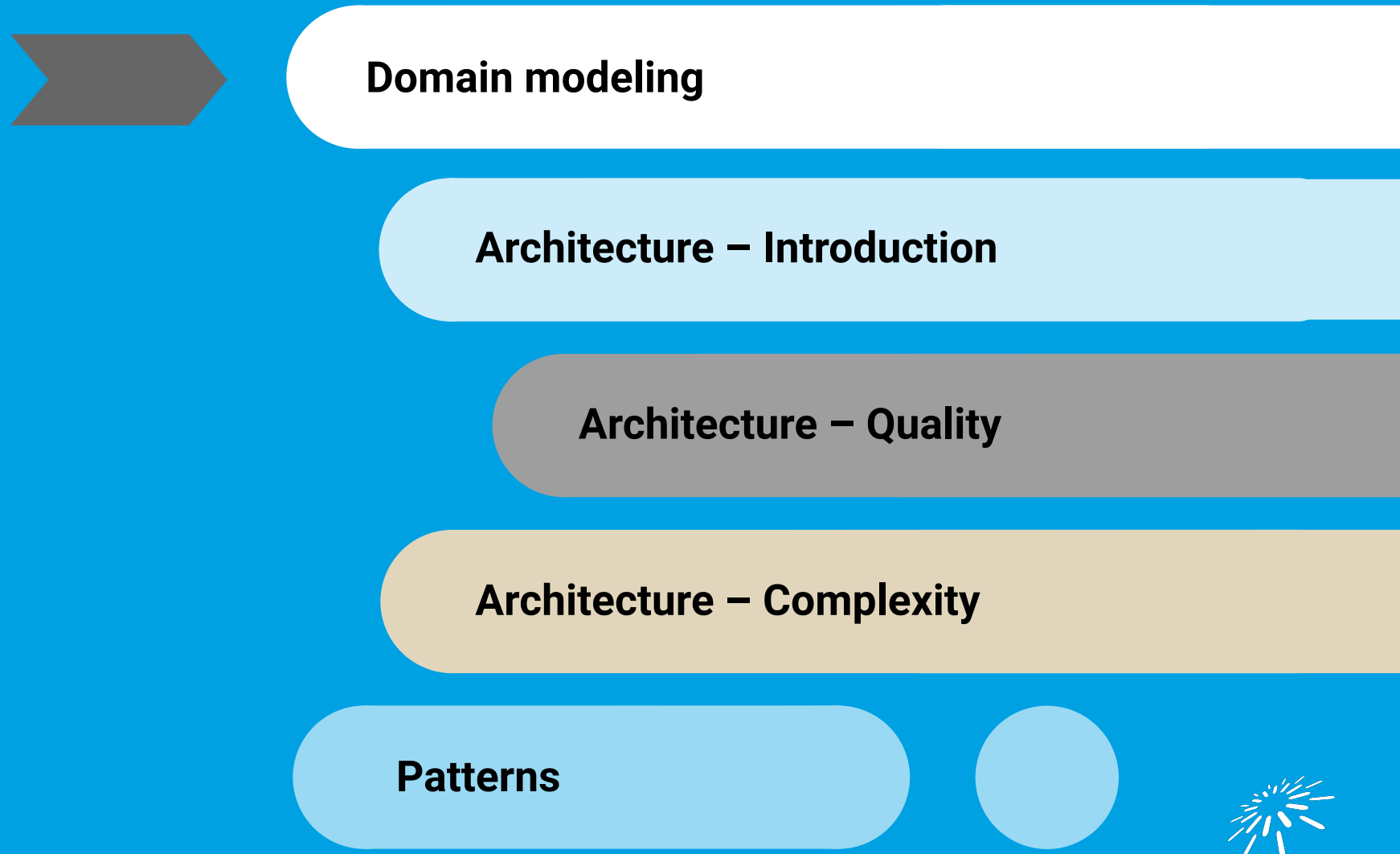
Domain modeling

Architecture – Introduction

Architecture – Quality

Architecture – Complexity

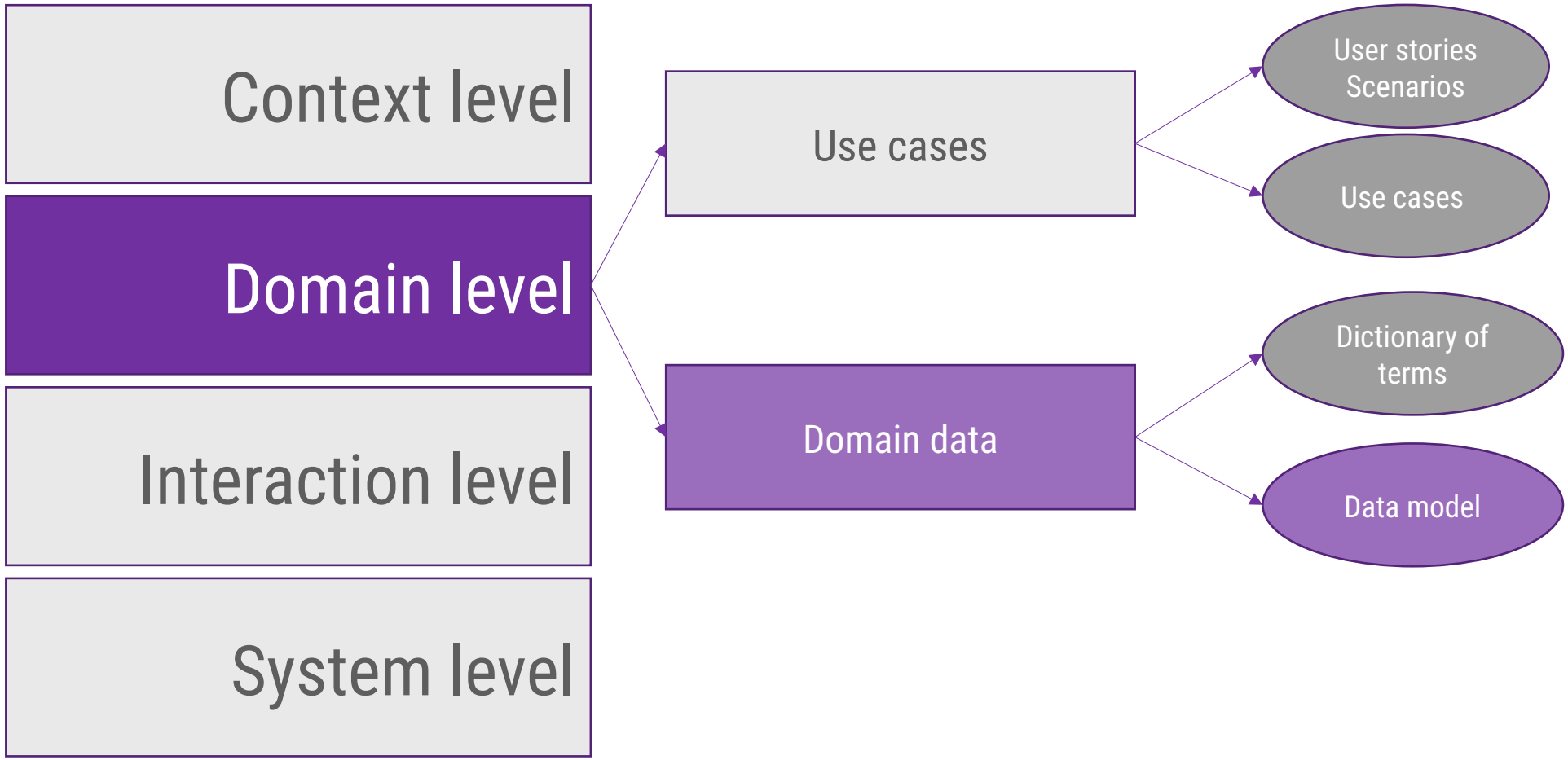
Patterns



Learning Objectives

- You can create an initial data model (**domain data model**).

Requirements exist at different **levels of abstraction**



Domain data

→ Describe at a high level of abstraction the data that is relevant for the system to be developed

- Domain data describes **entities** (things and concepts) that are important in the context. They explain the terms used in the client's descriptions.
  - Are usually described by **simple class diagrams** (attributes and operations do not need to be fully captured at this stage)
  - **Cardinalities** and relationships are important
  - **Glossary** often also sufficient (or supplementary)

Domain data

→ Describe at a high level of abstraction the data that is relevant for the system to be developed

- Domain data describes **entities** (things and concepts) that are important in the context. They explain the terms used in the client's descriptions.
  - Are usually described by **simple class diagrams** (attributes and operations do not need to be fully captured at this stage)
  - **Cardinalities** and relationships are important
  - **Glossary** often also sufficient (or supplementary)
- The goal is to understand the basic entities of the real world and their **relationships** that are necessary **to understand the task at hand.**
- Historically: ER diagrams (ERD) were used for data description in database development (Peter Chen, 1976)

TH Aschaffenburg
university of applied sciences

Identify domain data

→ Linguistic analysis for the identification of data

- Which **nouns** have been used in the descriptions so far?
- Which of the nouns are **synonymous**? → Eliminate them and decide on ONE term. Record your decision in the **dictionary** as a synonym that should not be used.
- Specify vague terms

Use cases

Identify nouns in use cases

Name: Withdraw **money from an ATM**.

Actors: **bank customer**, **banking system**

Precondition: Bank customer has money in the account, ATM is operational.

Triggering event: Person wants to withdraw money.

Main scenario:

1. **Bank customer** inserts **card** into the ATM
2. Banking system asks bank customer for **PIN**
3. **Bank customer** enters **PIN**
4. **Bank system** validates the card and authenticates the bank customer
5. **Bank system** asks for **amount** to be withdrawn.
6. **Bank customer** selects a predefined **amount**. → Alternative: Bank customer enters amount.
7. **Banking system** issues card and then the **money**

TH Aschaffenburg
university of applied sciences

Identify domain data

→ Linguistic analysis for the identification of data

Customer explains what the system should do: "*Peter Winter should be able to use the online conference system alongside other people at the TH Aschaffenburg. The system should work via SSO ...*"

What questions do you need to clarify in more detail?

Too general?

Too specific?

Synonyms?

Nouns?

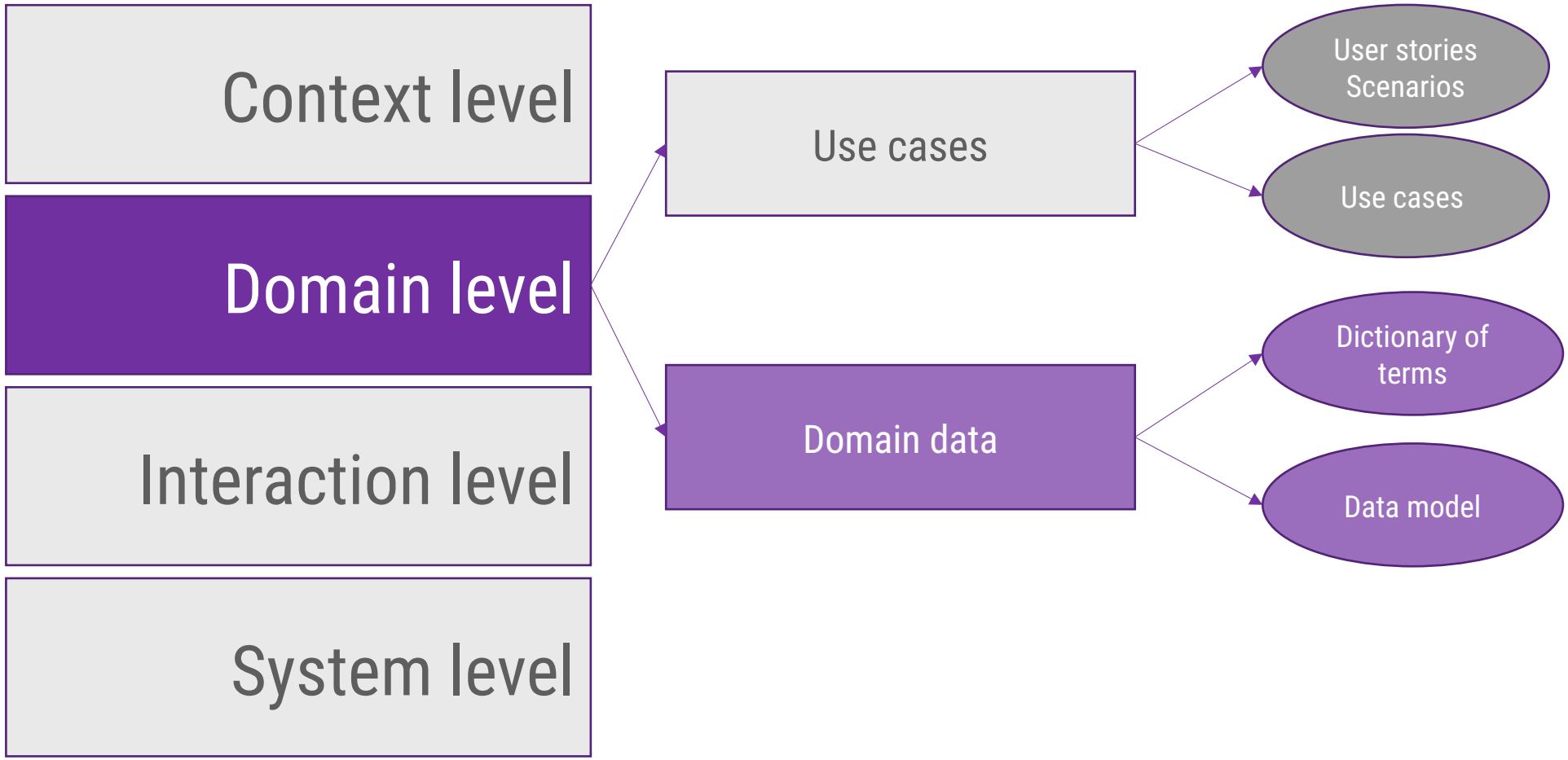
Identify domain data

→ Linguistic analysis for the identification of data

- Which **nouns** have been used in the descriptions so far?
- Which of the nouns are **synonymous**? → Eliminate them and decide on ONE term. Record your decision in the **dictionary** as a synonym that should not be used.
- Concretize vague terms

Customer explains what the system should do: "*Peter Winter should be able to use the **online conference system** alongside other **people** at the TH Aschaffenburg. The **system** should work via SSO ...*"

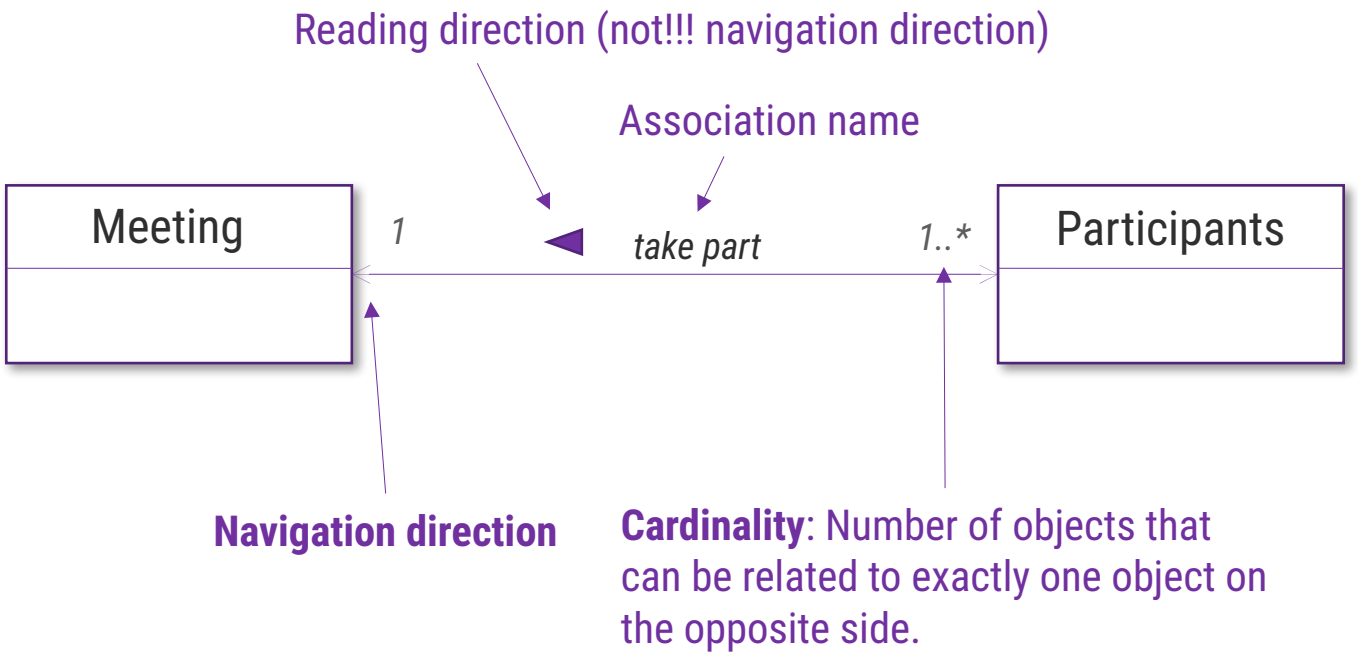
Requirements exist at different **levels of abstraction**



Relationships between data

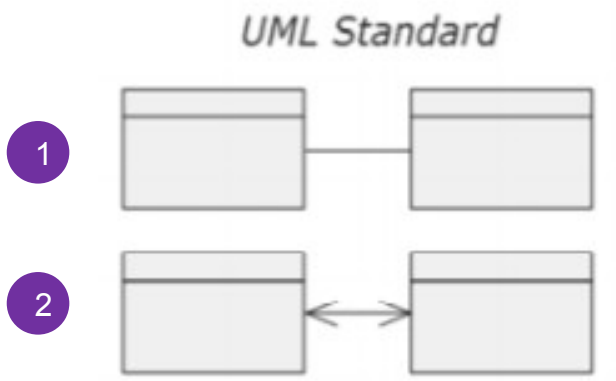
→ Association

- **Associations** between classes model possible object relationships between the class instances



Associations

→ Navigability

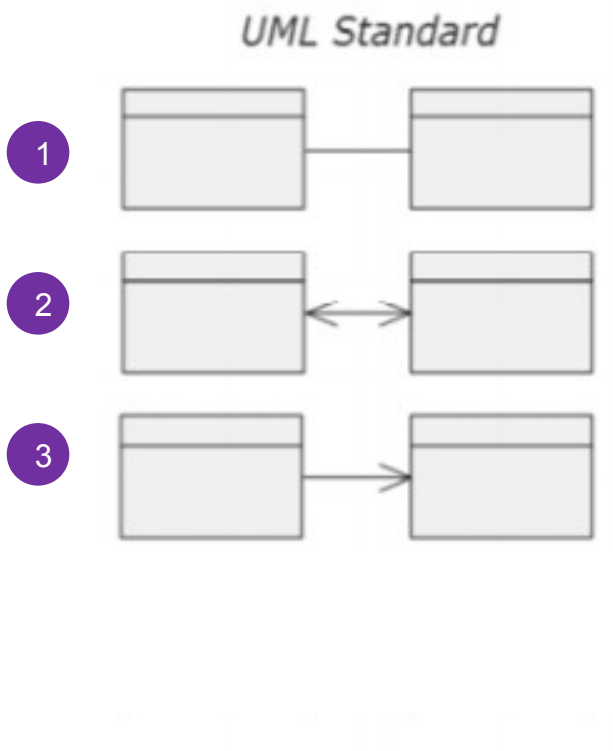


UML standard:
Association direction not yet defined. In practice often equivalent to 2

Bidirectional navigability

Associations

→ Navigability



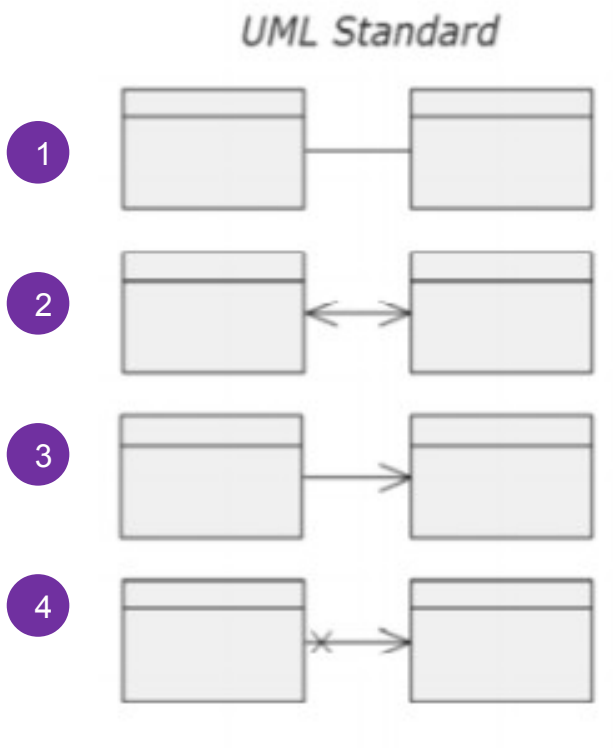
UML standard:
Association direction not yet defined. In practice often equivalent to 2

Bidirectional navigability

Unidirectional navigability from "left to right", not defined in the other direction.

Associations

→ Navigability



UML standard:
Association direction not yet defined. In practice often equivalent to 2

Bidirectional navigability

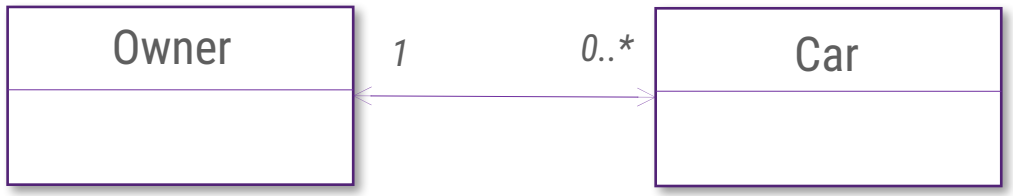
Unidirectional navigation from "left to right", not defined in the other direction.

Navigable from left to right, but reverse navigation is not possible.

Associations

→ Cardinalities, roles

Exactly 1	1
>= 0:	* or 0..*
0 or 1_	0...1 or 0,1
Fixed Number (e.g., 3):	3
Range (e.g., >= 3):	3..*
Range (e.g., 3 to 6):	3..6
Enumeration	3,6,7,8,9 or 3, 6..9

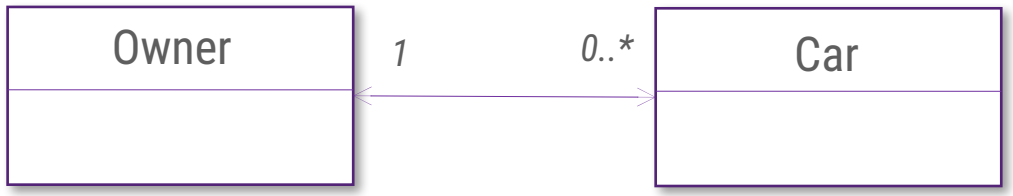


A car has exactly one owner, an owner can have none or several cars

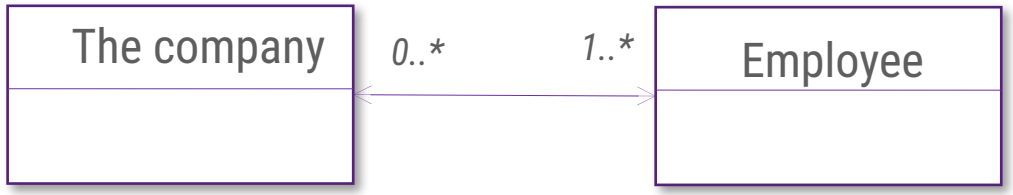
Associations

→ Cardinalities, roles

Exactly 1	1
>= 0:	* or 0..*
0 or 1_	0...1 or 0,1
Fixed Number (e.g., 3):	3
Range (e.g., >= 3):	3..*
Range (e.g., 3 to 6):	3..6
Enumeration	3,6,7,8,9 or 3, 6..9



A car has exactly one owner, an owner can own none or several cars

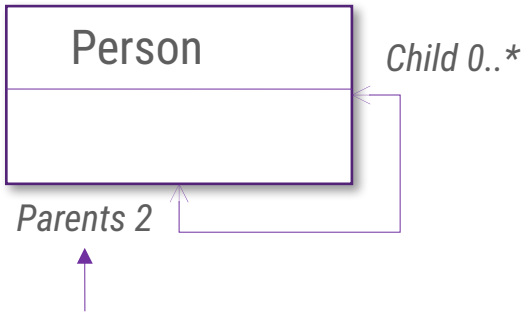


At least one employee works in a company; an employee can work in several companies.

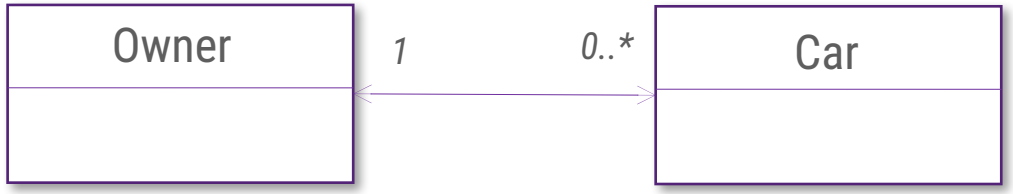
Associations

→ Cardinalities, roles

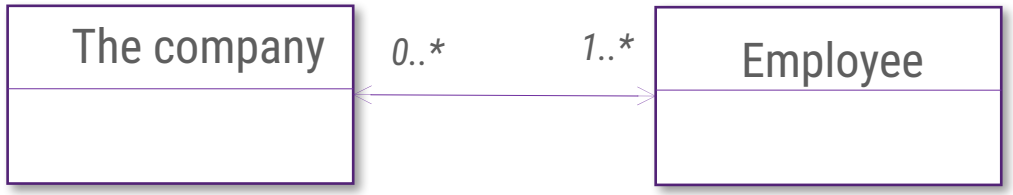
Exactly 1	1
>= 0:	* or 0..*
0 or 1_	0...1 or 0,1
Fixed Number (e.g., 3):	3
Range (e.g., >= 3):	3..*
Range (e.g., 3 to 6):	3..6
Enumeration	3,6,7,8,9 or 3, 6..9



Role in a relationship



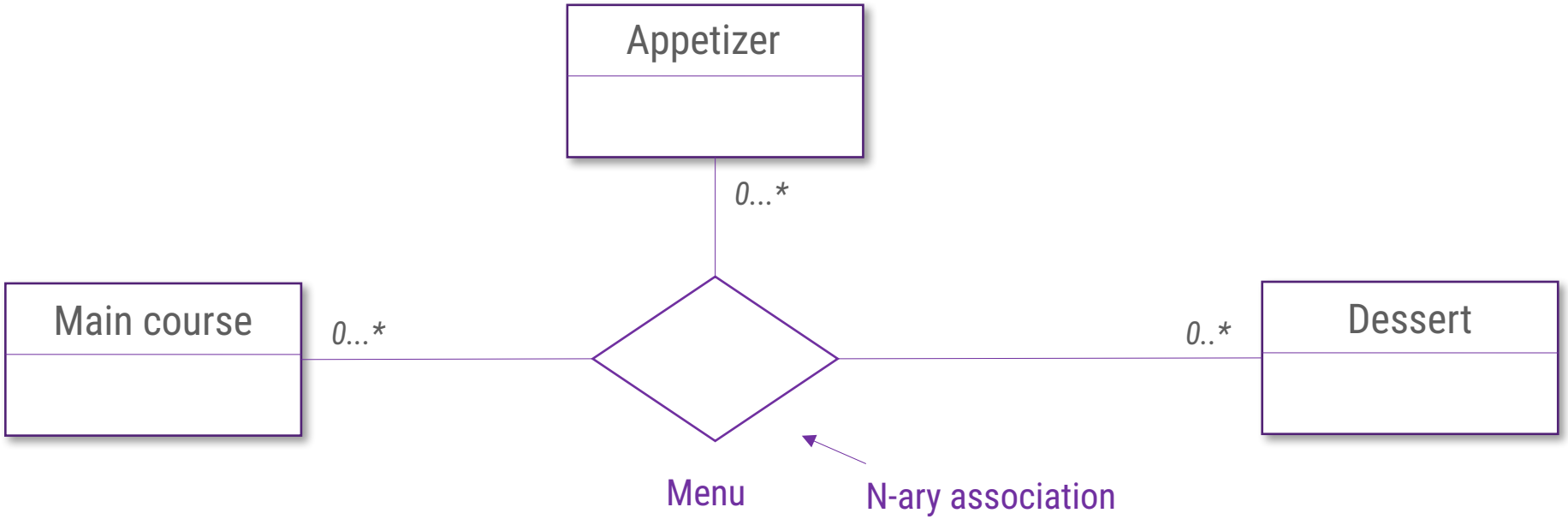
A car has exactly one owner, an owner can own none or several cars



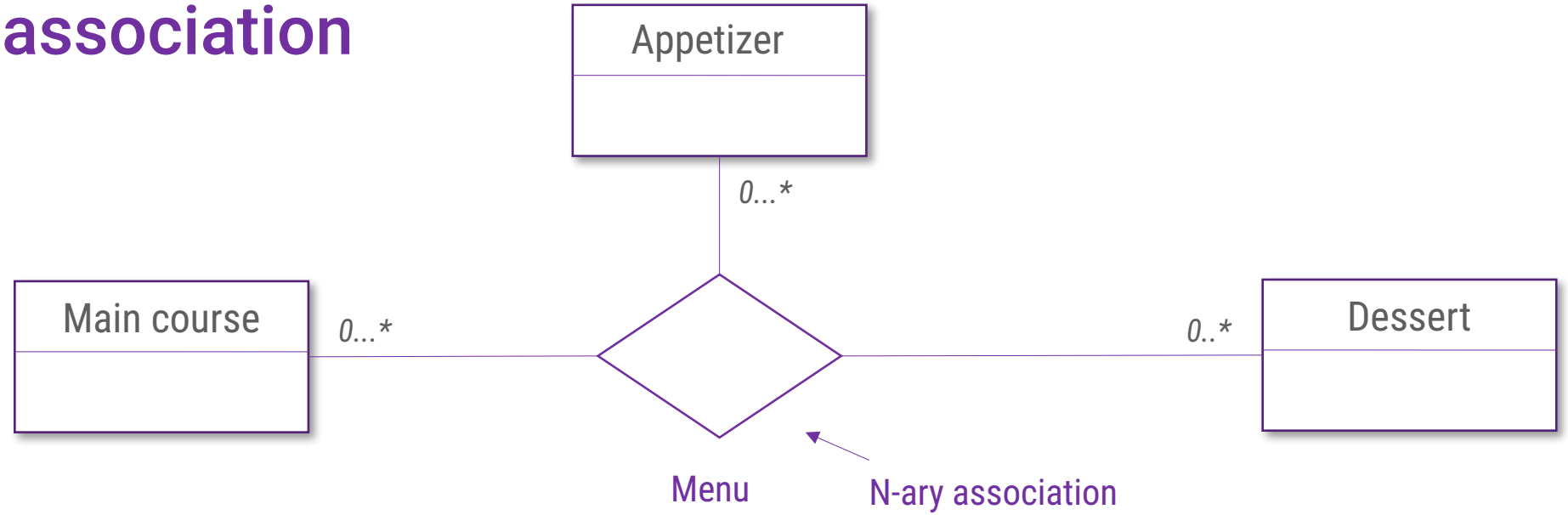
At least one employee works in a company; an employee can work in several companies.

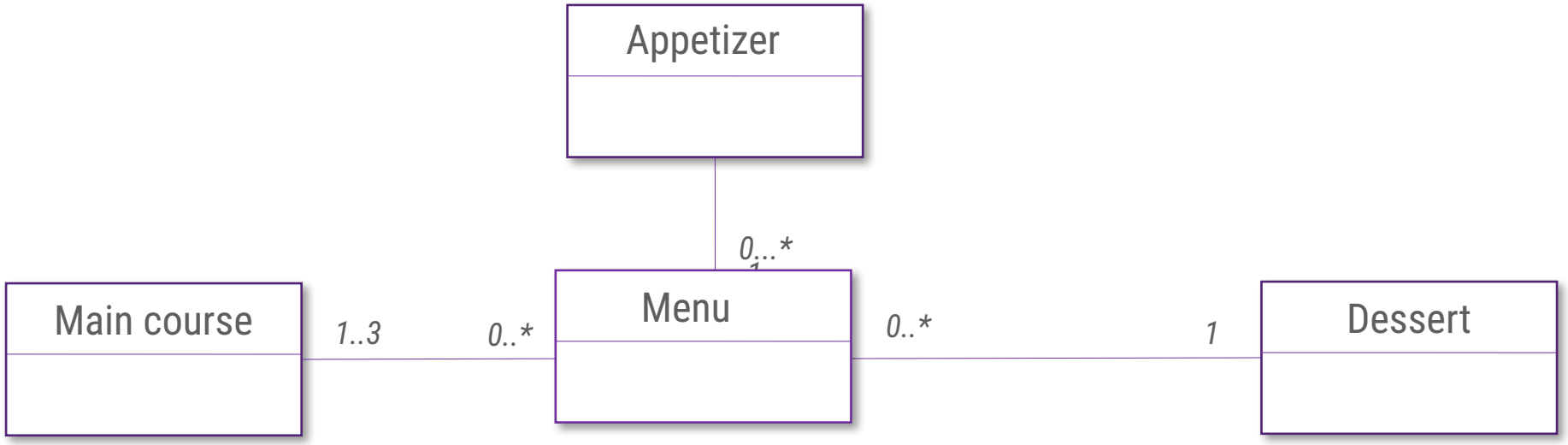
TH Aschaffenburg
university of applied sciences

N-ary association



N-ary association





Relationships between data: Aggregation

- Special form of association
- "**is-part-of**" relationship: There is a superordinate whole that contains a subordinate whole.
- UML distinguishes between two types of aggregations
  - Weak aggregation (shared aggregation)
  - Strong aggregation - composition (composite aggregation)

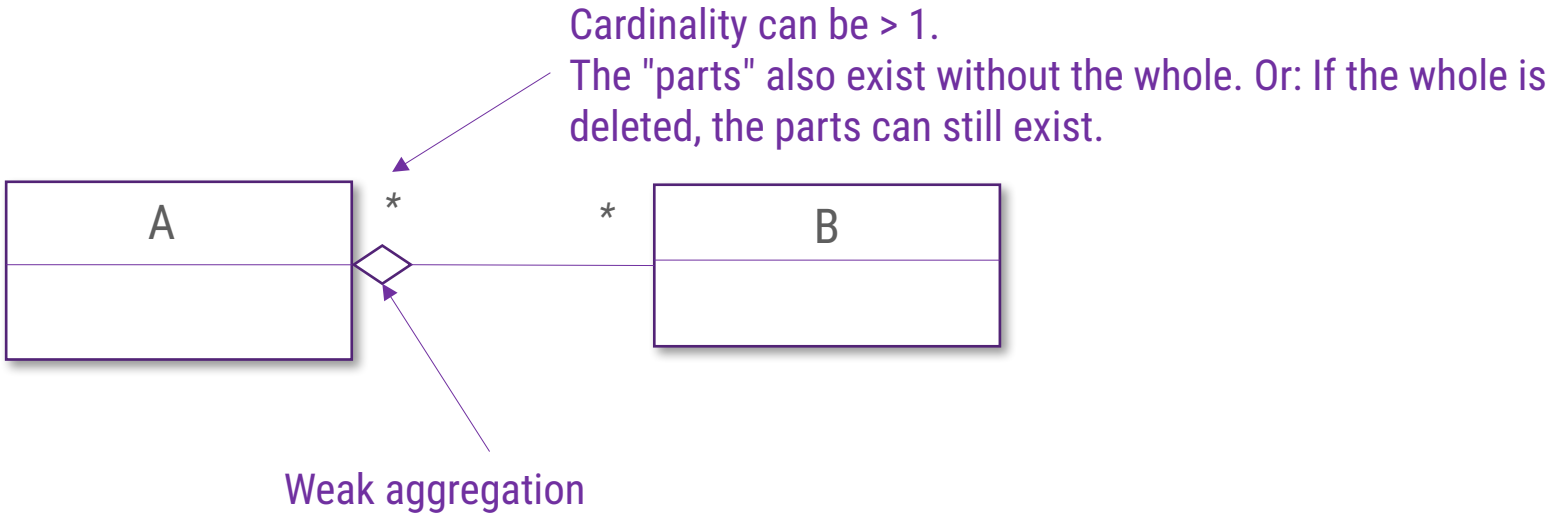
Relationships between data: Aggregation

- Special form of association
- "**is-part-of**" relationship: There is a superordinate whole which contains a subordinate whole.
- UML distinguishes between two types of aggregations
  - Weak aggregation (shared aggregation)
  - Strong aggregation - composition (composite aggregation)
- Properties
  - **Transitivity**: If C is part of B and B is part of A, then C is also part of A. (*If the cooling system is part of the engine and the engine is part of the car, then the cooling system is also part of the car.)*
  - **Anti-symmetry**: If B is part of A then A cannot be part of B. (*If the engine is part of the car, then the car is not part of the engine)*

TH Aschaffenburg
university of applied sciences

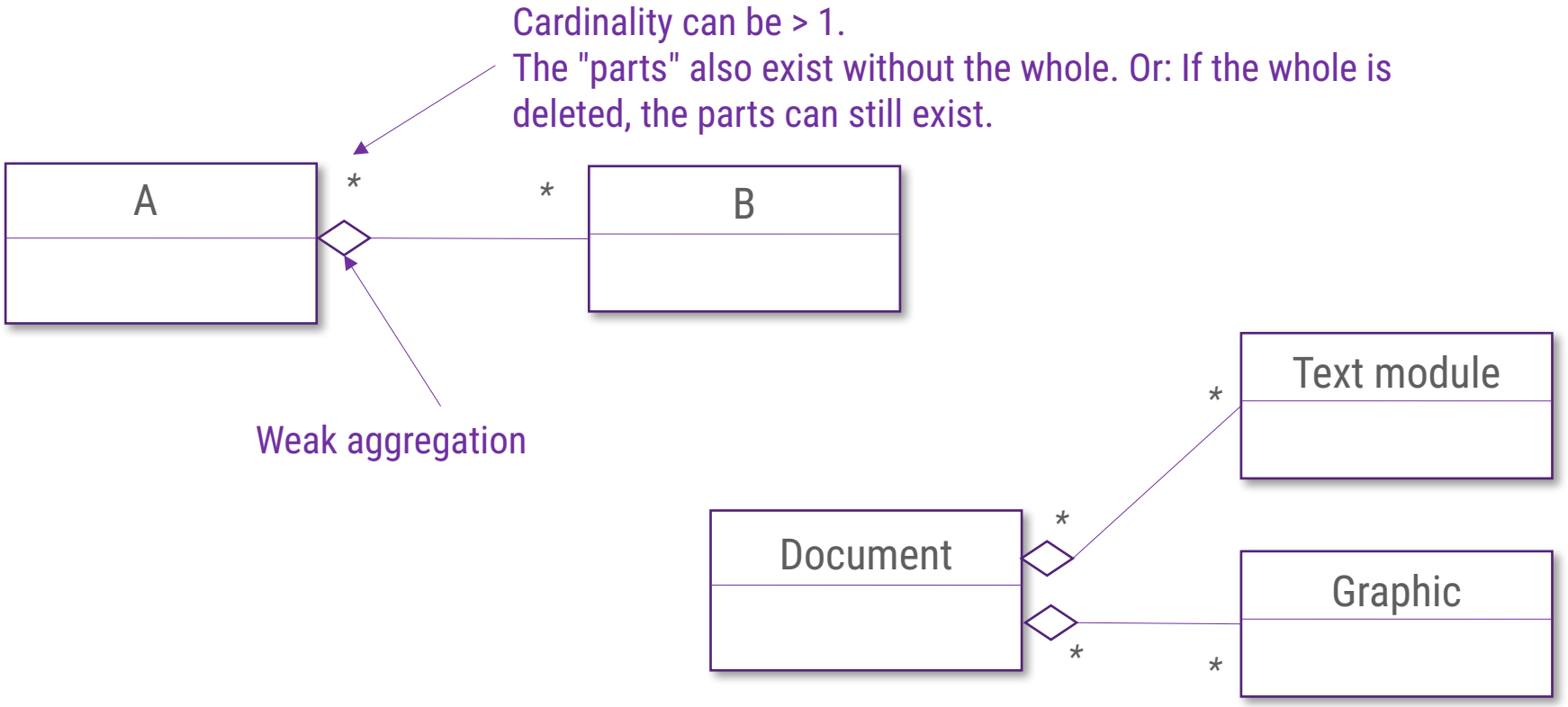
Weak aggregation

→ Parts independent of the whole



Weak aggregation

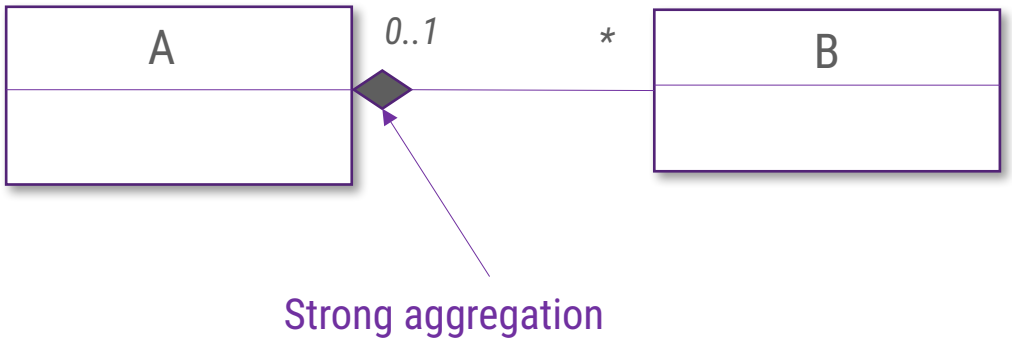
- Parts independent of the whole



Strong aggregation

→ Composition

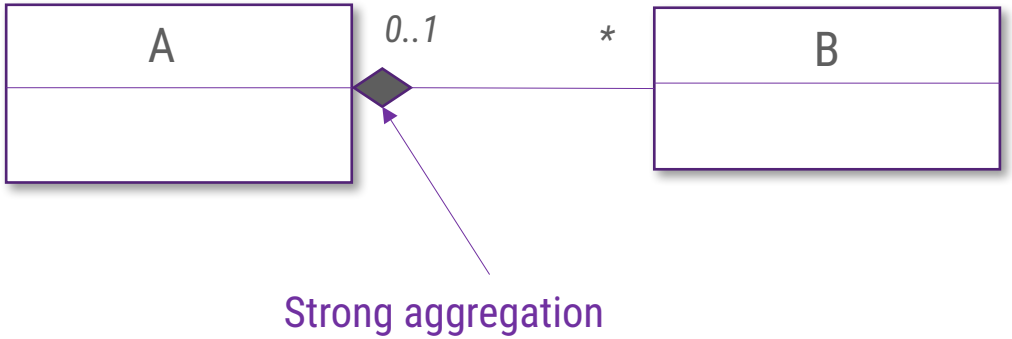
A part can only be part of a whole at any given time. Cardinality MAX 1.
If the whole is deleted, the parts are also deleted.
Cardinality 1 can also be found in the literature (→ existence dependency). The part cannot exist without the whole.

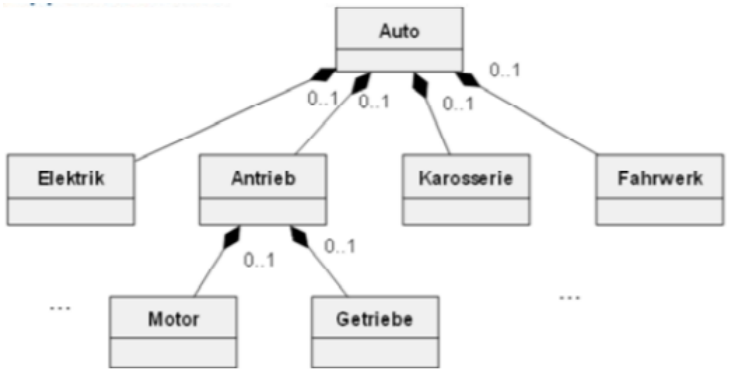


Strong aggregation

→ Composition

A part can only be part of a whole at any given time. Cardinality MAX 1.
If the whole is deleted, the parts are also deleted.
Cardinality 1 can also be found in the literature (→ existence dependency). The part cannot exist without the whole.

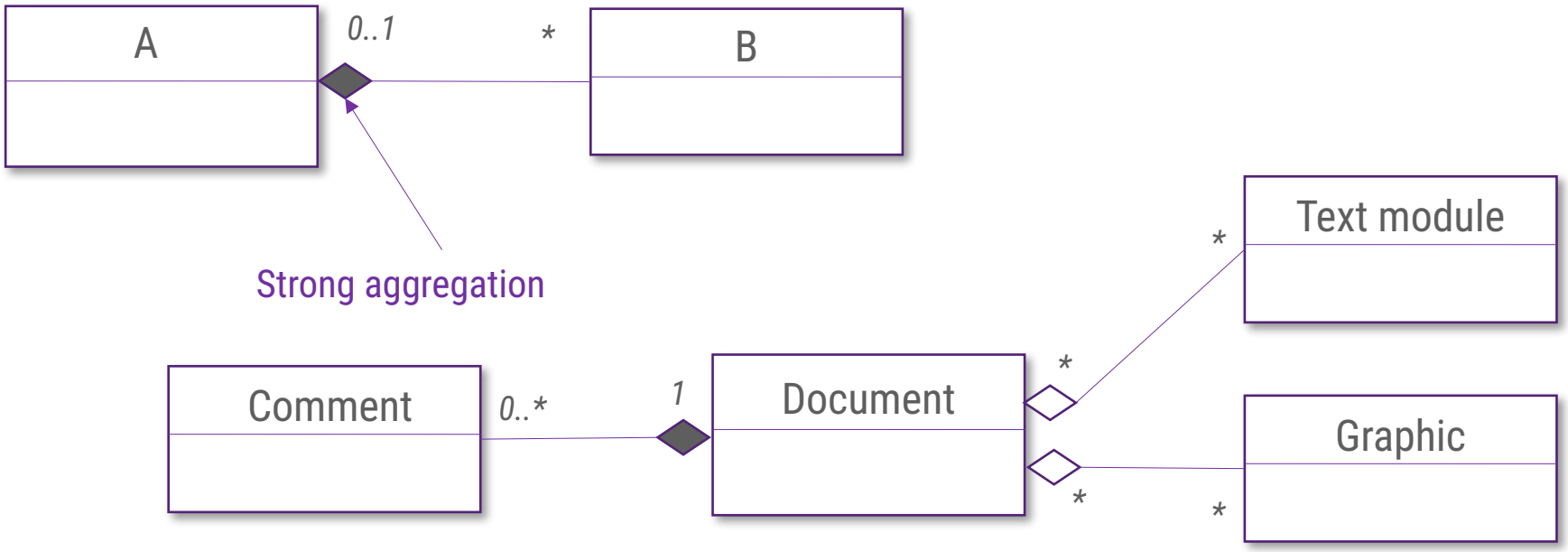




Strong aggregation

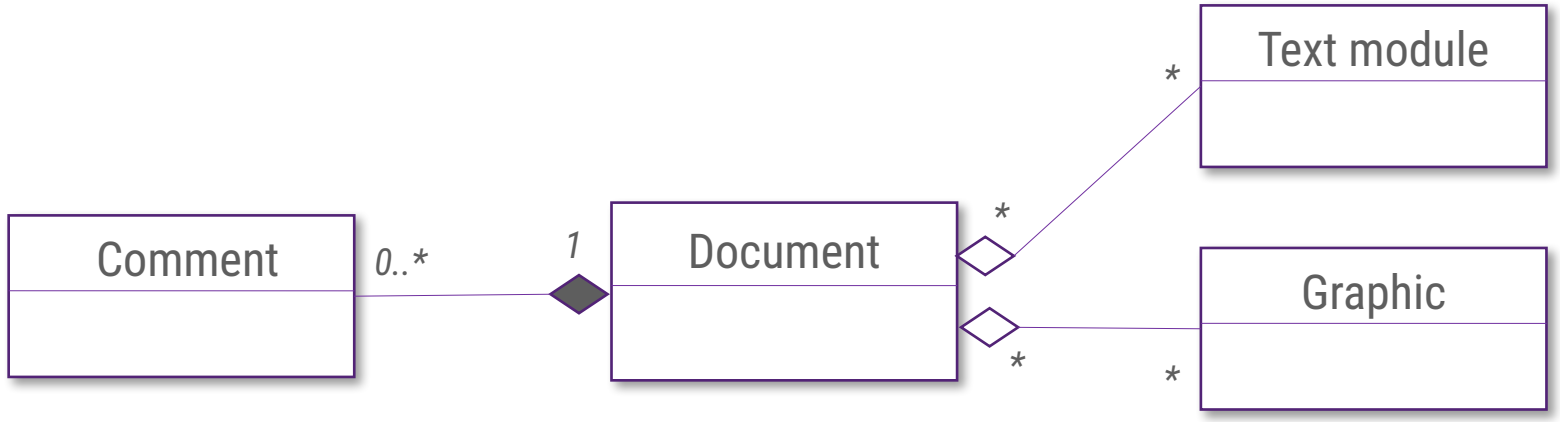
→ Composition

A part can only be part of a whole at any given time. Cardinality MAX 1.
If the whole is deleted, the parts are also deleted.
Cardinality 1 can also be found in the literature (→ existence dependency). The part cannot exist without the whole.



Strong vs. weak aggregation

→ Key questions



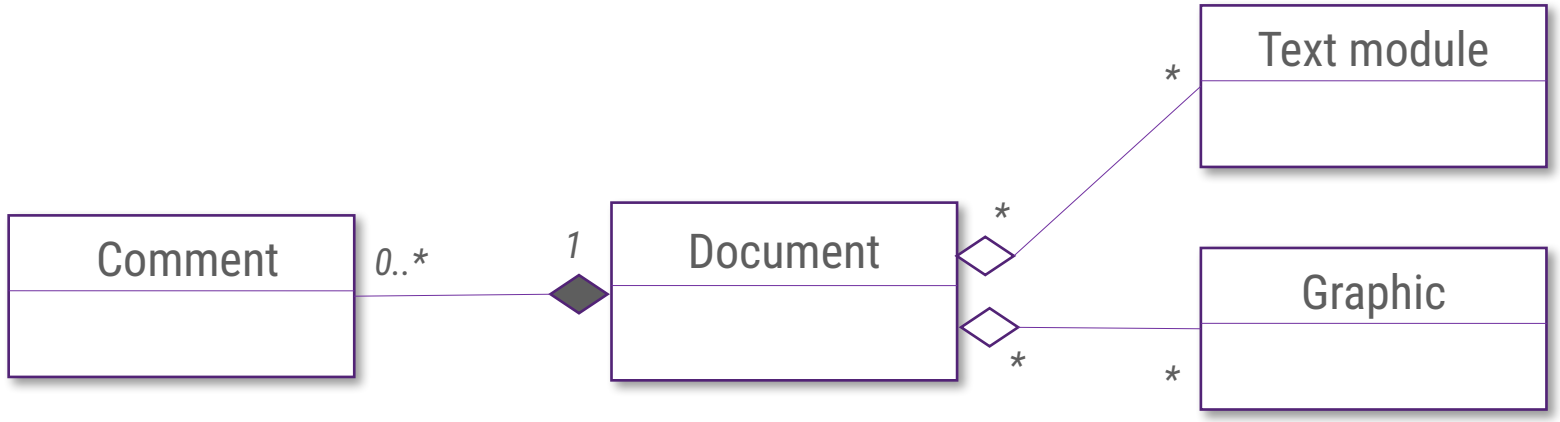
Visibility: Is the part only visible to the whole?

→ Yes: strong aggregation (composition)

→ No: weak aggregation

Strong vs. weak aggregation

→ Key questions



Visibility: Is the part only visible to the whole?

→ Yes: strong aggregation (composition)

→ No: weak aggregation

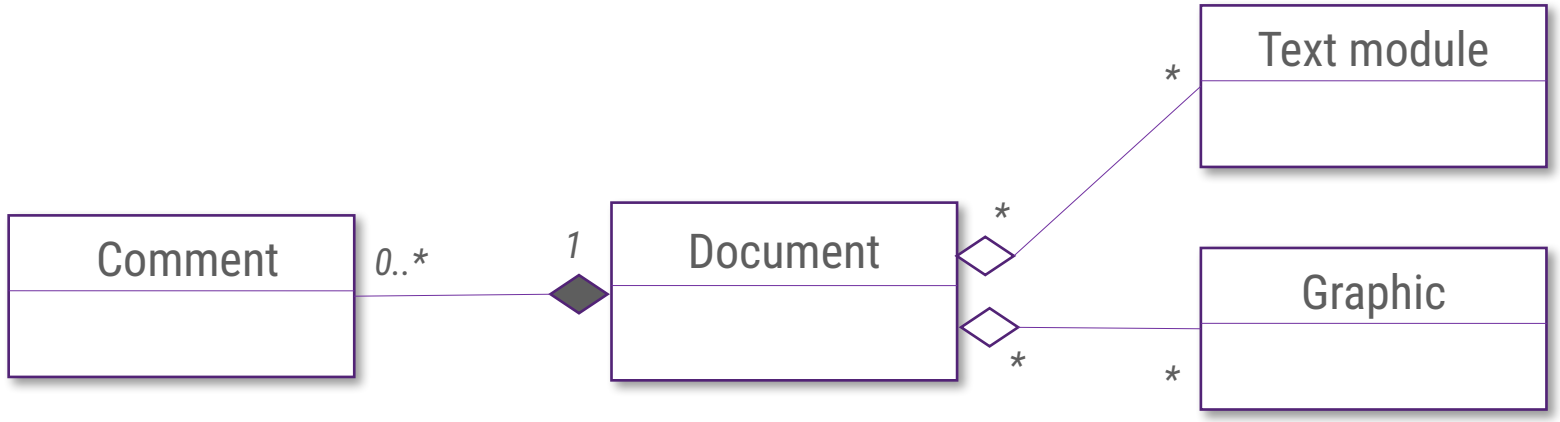
Lifetime: Does the part exist when the whole thing is deleted?

→ No: strong aggregation (composition). The whole creates and deletes the parts.

→ Yes: weak aggregation

Strong vs. weak aggregation

→ Key questions



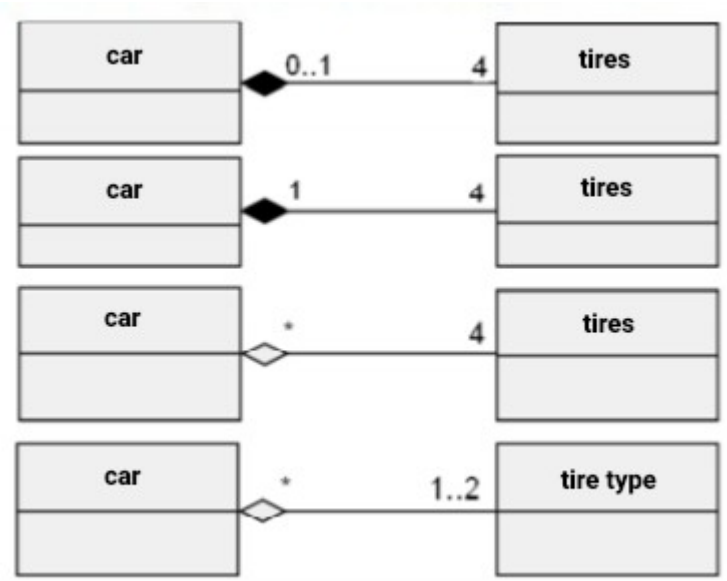
Visibility: Is the part only visible to the whole?
- → Yes: strong aggregation (composition)
- → No: weak aggregation

Lifetime: Does the part exist when the whole thing is deleted?
- → No: strong aggregation (composition). The whole creates and deletes the parts.
- → Yes: weak aggregation

Copy: What happens when the whole thing is copied?
- → Strong aggregation (composition). The whole and the parts are copied
- → Yes: Only references to the parts are copied.

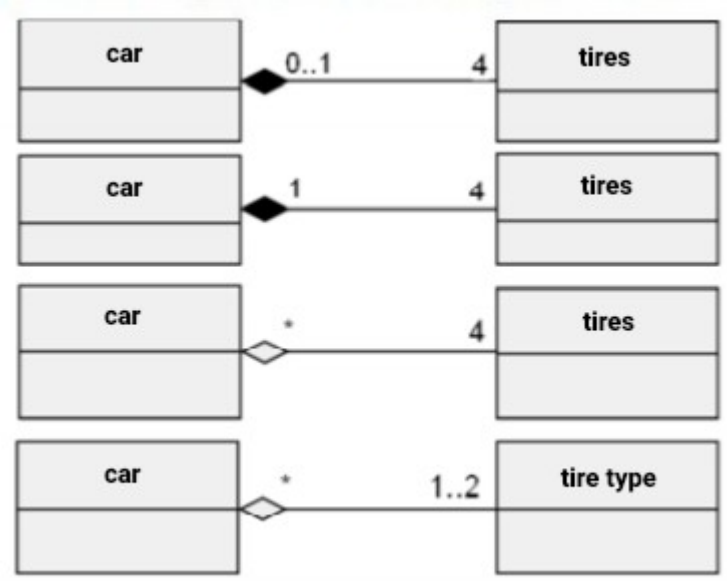
Aggregation: Example

→ Which of the following relationships applies?



Aggregation: Example

→ Which of the following relationships applies?



A car has exactly four tires. 4 tires are mounted to zero or one car at any one time. → Correct

A car has exactly four tires. 4 tires are mounted on exactly one car at a time. → Cardinality **incorrect**, tires do not have to be mounted

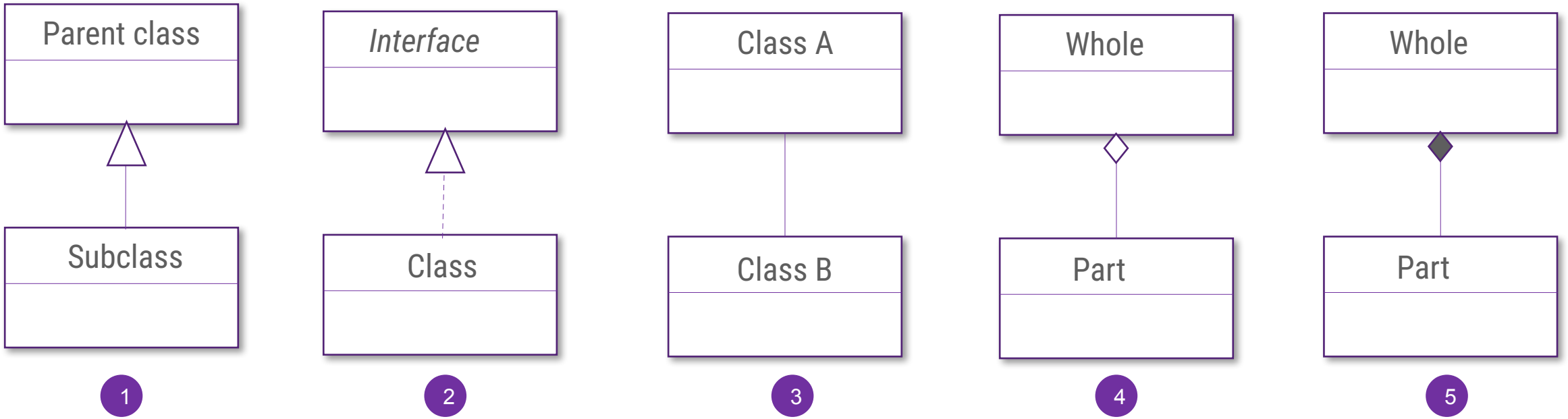
A car has exactly four tires. 4 tires are mounted to any number of cars at any one time. → **Wrong**

A car has exactly one/two tire types. A tire type can be mounted to any car. The same tire types must be mounted on the axles.

Relationships between data

→ What is what?

Implementation Association Aggregation (strong) Generalization Aggregation (weak)



"consists of" Relationship

Inheritance **"is-a"** relationship

"has a" Relationship

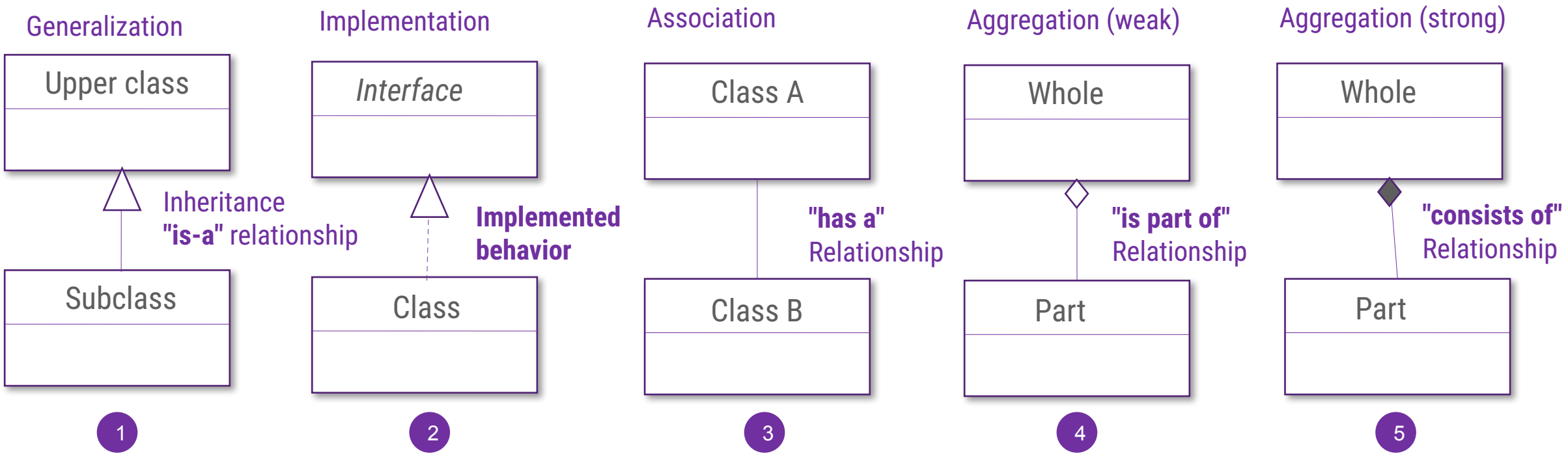
Implemented behavior

"is part of" Relationship

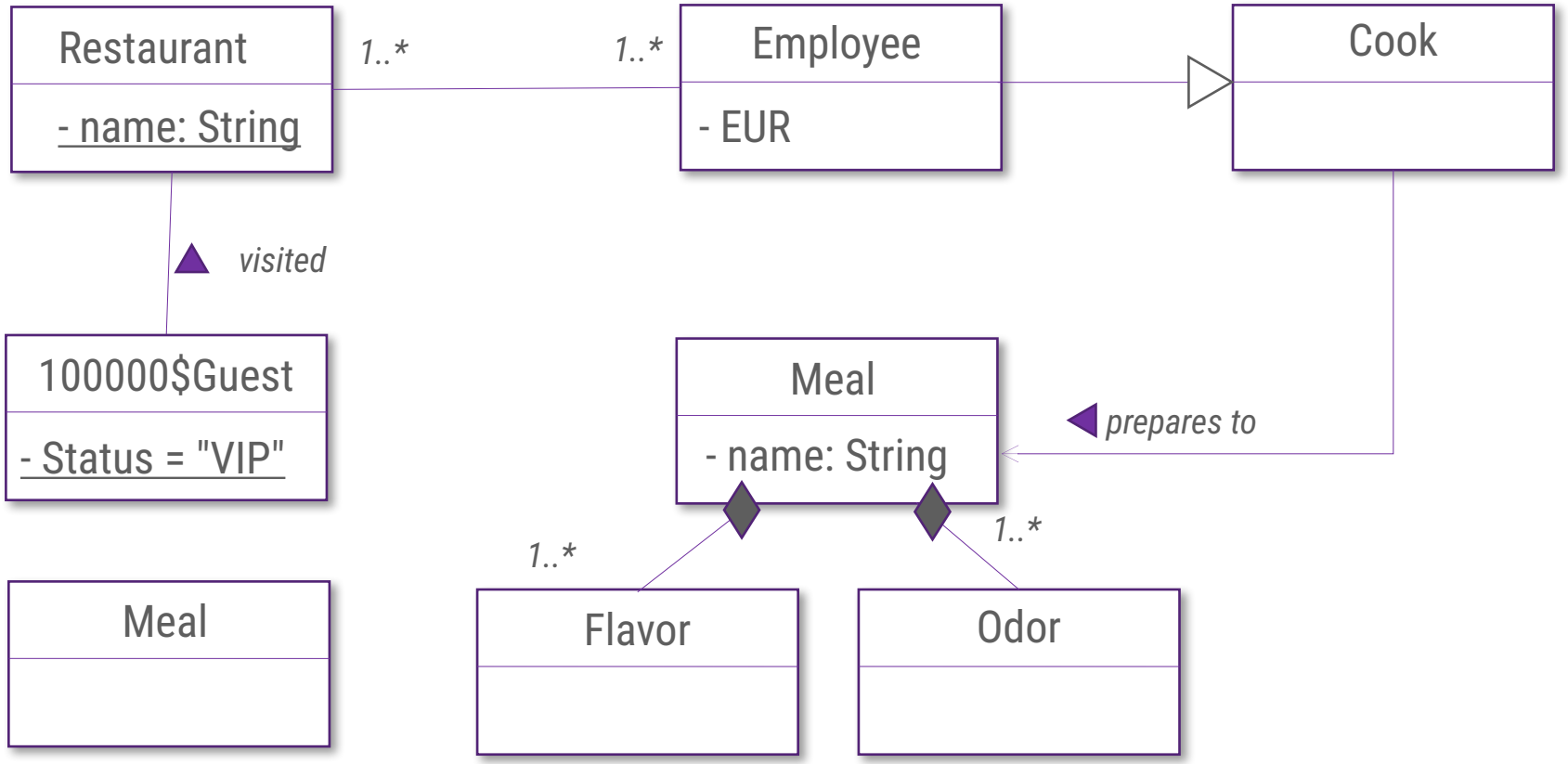
TH Aschaffenburg
university of applied sciences

Relationships between data

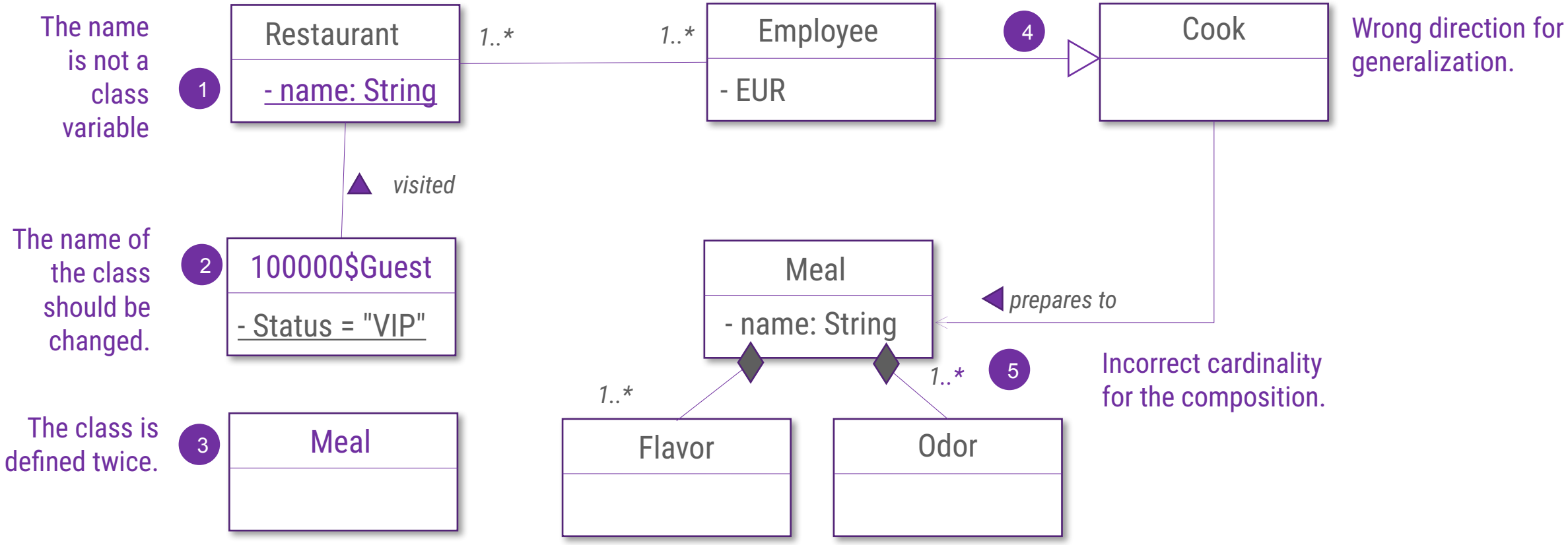
→ In our case between classes (in UML representation)



What errors do you see in the following class diagram?

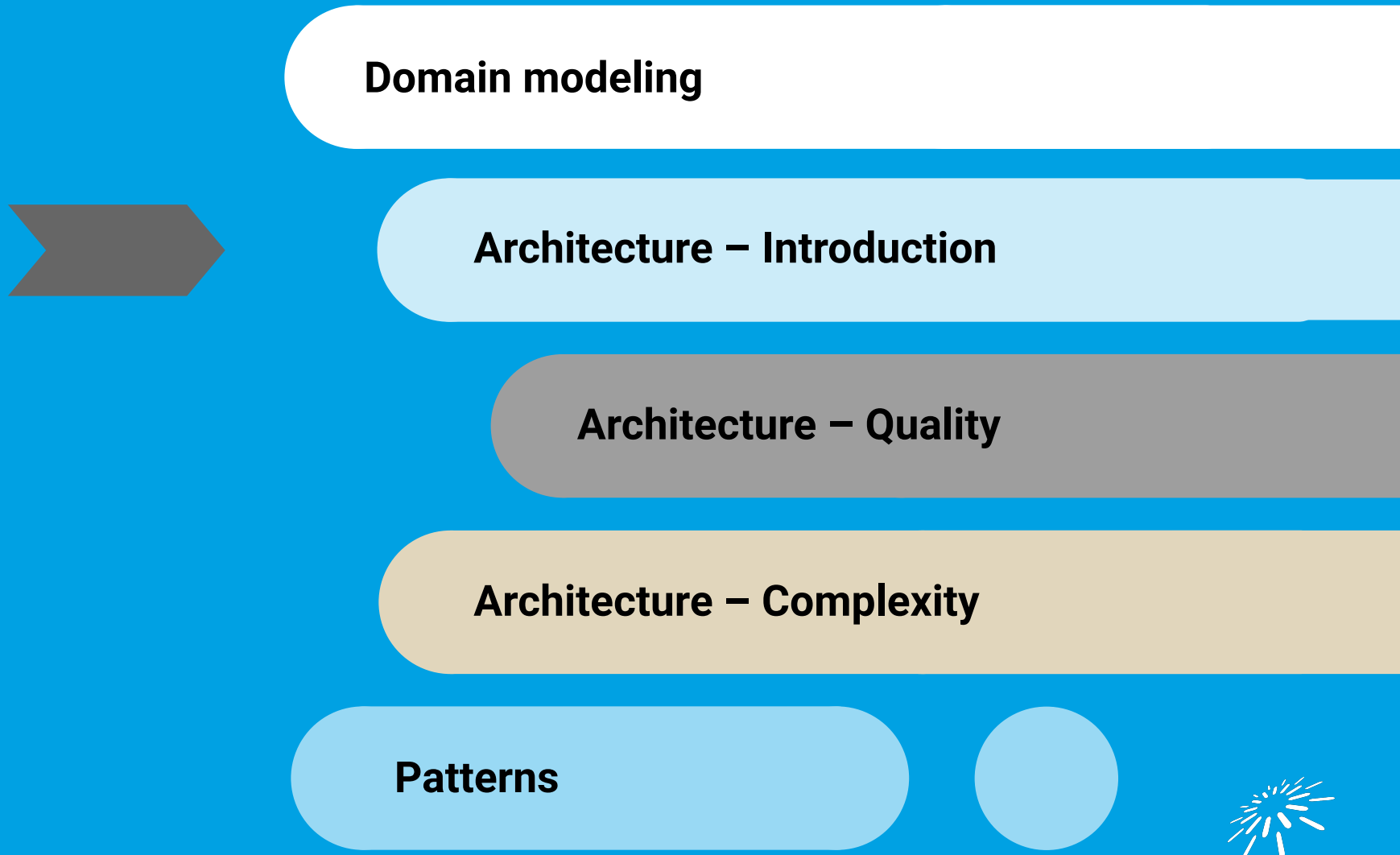


What errors do you see in the following class diagram?



To Go: Summary **context and domain level**

- It is important to understand the **context** of the users when designing the user experience.
- Product vision and goals, user stories, tasks and use cases, domain data, roles and persona characterize this context.
- This ensures that **the right tasks** are supported. Care is taken not to introduce technical considerations unnecessarily early on.
- Domain data describes entities in the real world and their **relationships** with each other.
- A simplified **UML class diagram** can be used for the graphical representation.



Disciplines in software engineering

Basic topics

Configuration management | Documentation |
Knowledge management | People in the SWE process and digital ethics | Tools

Development

Requirements
- Context analysis
- Requirements Engineering

Design
- Course granular design: Architecture
- Detailed design

Implementation

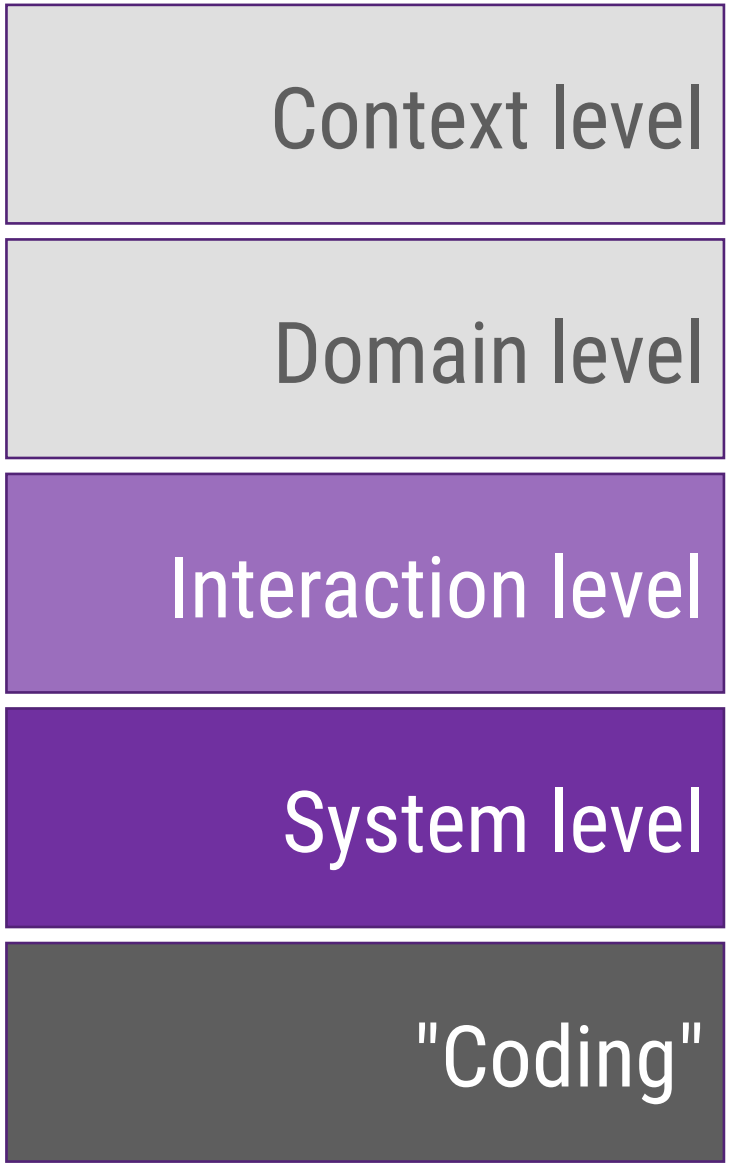
QualityMgt.

Quality assurance and testing
- Test, inspection, metrics

Processes and procedure models
- Improvement, process model, maturity levels

Evolution
- Roll-Out
- Operation
- Maintenance
- Further development
- Reuse
- Reengineering
- Change management

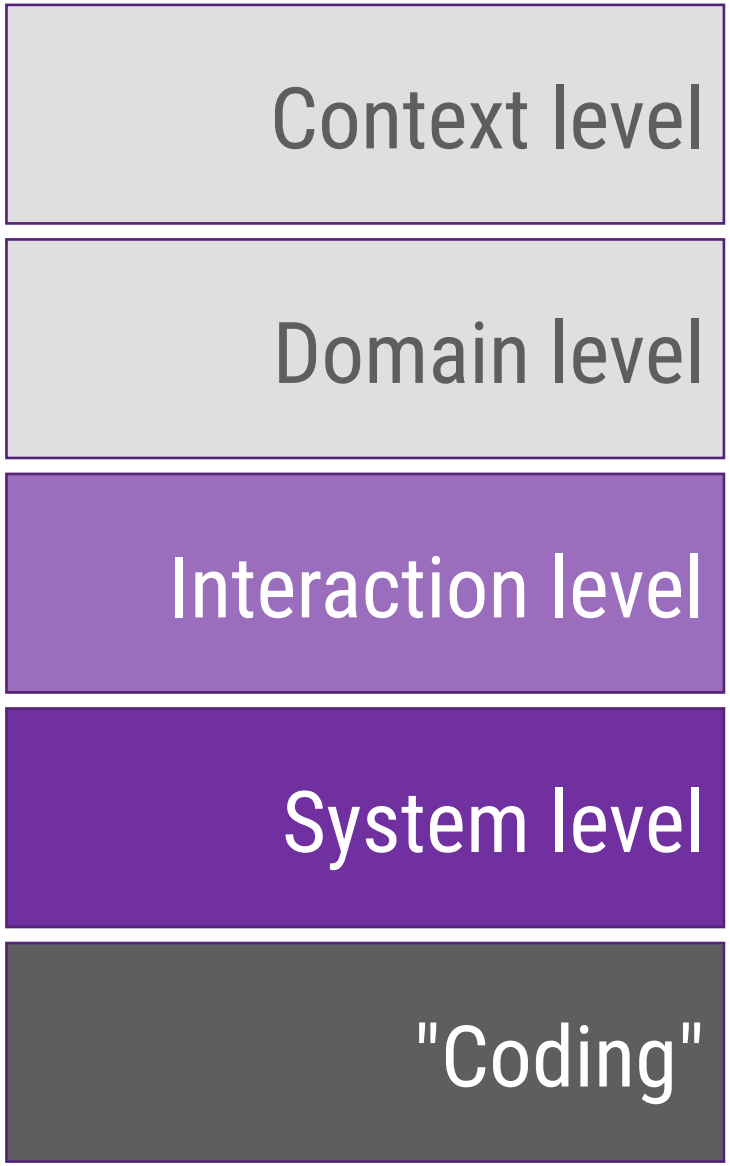
Management
- Strategy
- Economy
- Team
- Dates
- Risks
- Customer, client/contractor
- Innovation



Determine and model design

What happens in between?

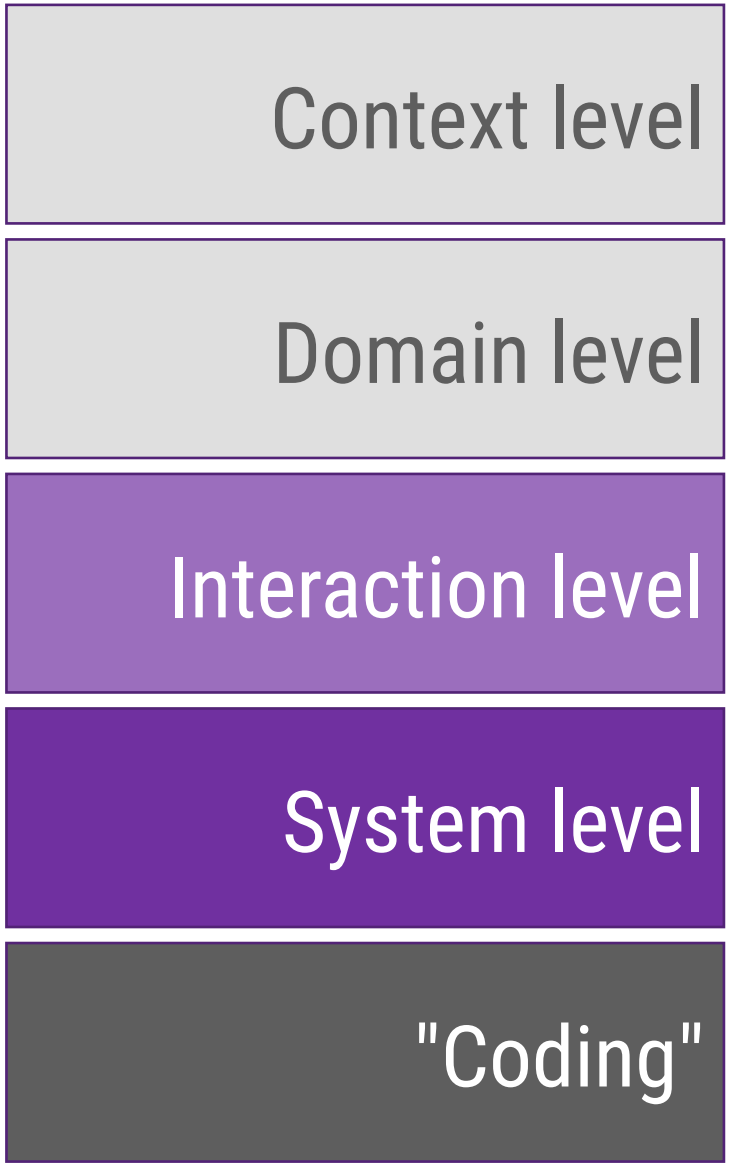
Implement design



Determine and model design

Gradual refinement of the system, different views here too

Implement design



Goal of the design

Division of the system into manageable units, course granular design (architecture in the narrower sense), fine granular design

Check-In

Why is it important to think about architecture?

Design, Architecture

design - *(1) The process of defining the architecture, components, interfaces, and other characteristics of a system or component.*
(2) The result of the process in (1).

IEEE Std 610.12 (1990)

architecture - *The fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.*

IEEE Std 1471 (2000)

Architecture describes the **result**, design **both** the activity and the result.
A software architecture therefore describes the basic organization of a software system with all its components and their relationships with each other and their environment. It also describes the underlying principles that guide design and development.

TH Aschaffenburg
university of applied sciences

Design, Architecture

→ Components

- A component is a **part of a system.**
- The architecture determines which components a system should consist of.
- A component is **atomic** or more **refined**; it offers its environment a **set of services** that can be used via a well-defined **interface**.

Design, Architecture

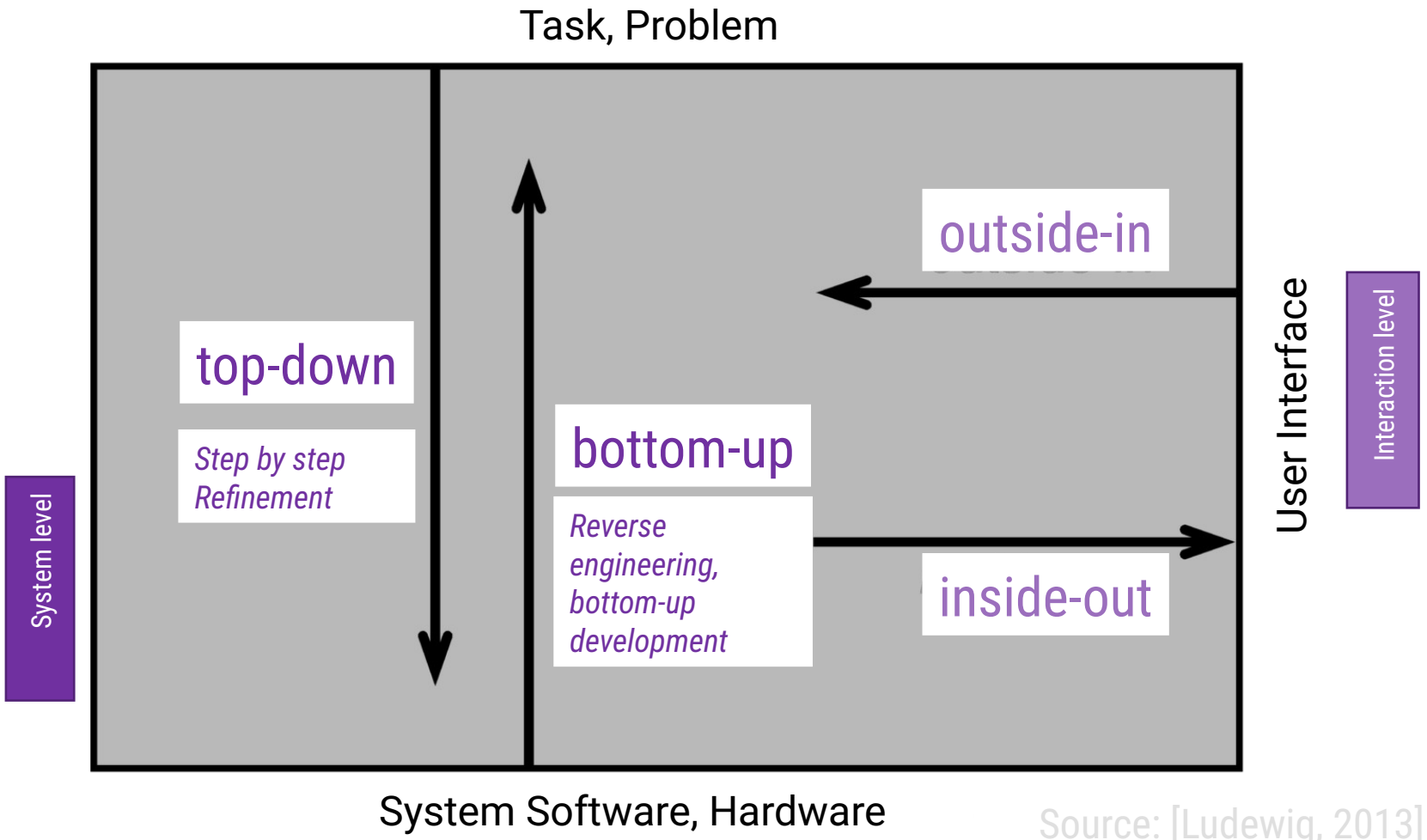
→ Components

- **Definition**: “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies. It can be deployed independently and is subject to composition by third parties.” [WCOP 96]

- **WARNING**: „Components are for composition, much beyond that is unkown.“ [after Szyperski]

Different directions

→ Iterative process that encompasses all directions

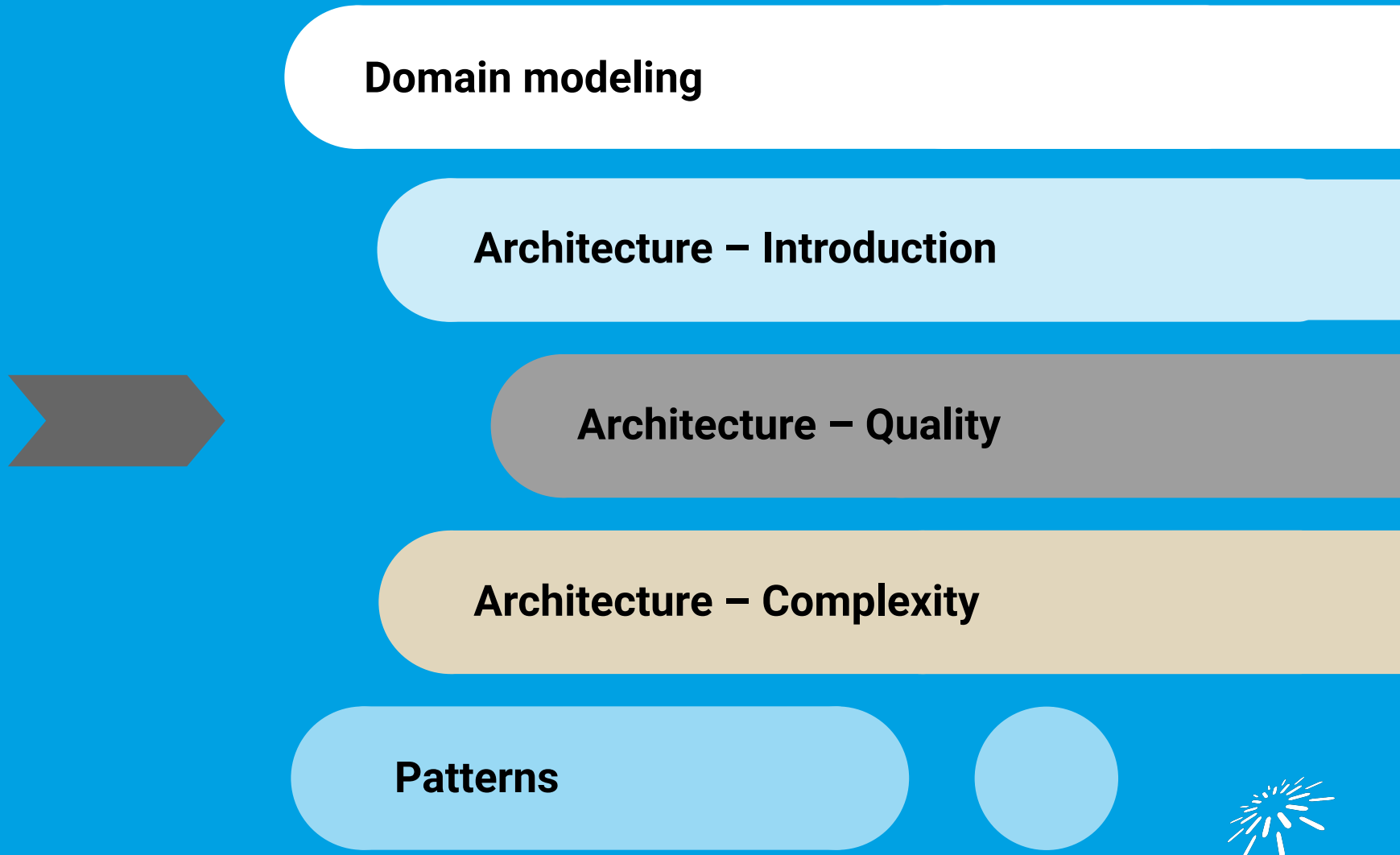


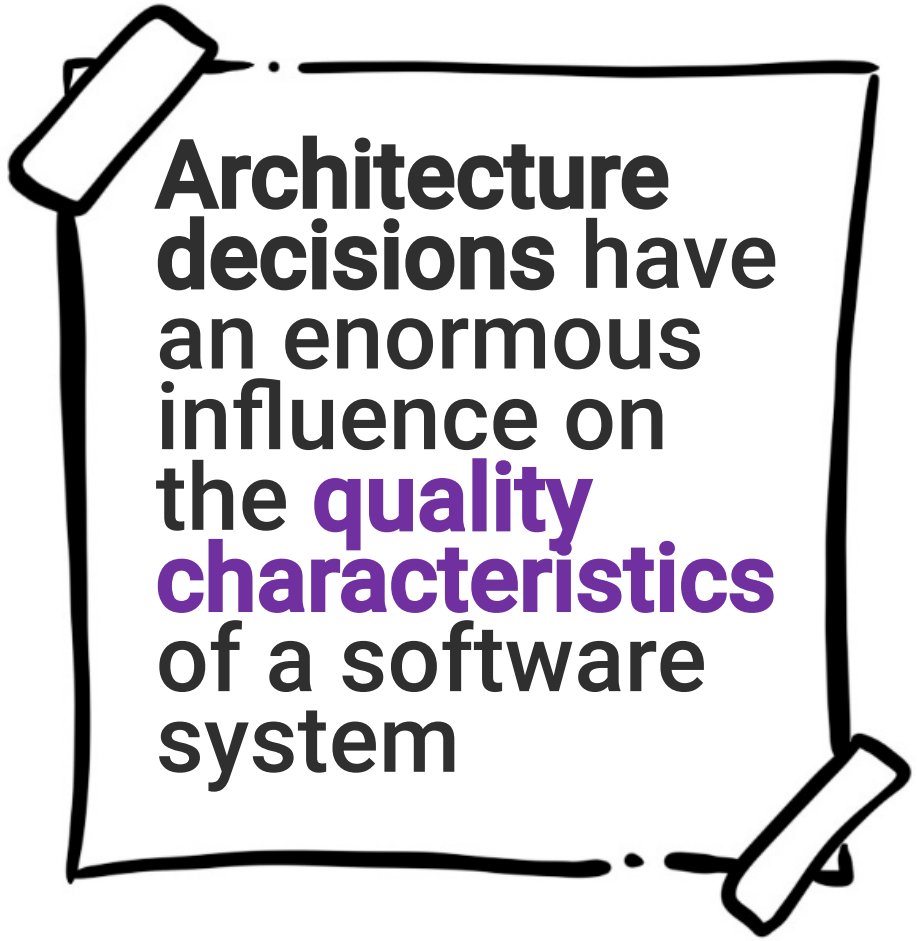
Source: [Ludewig, 2013]

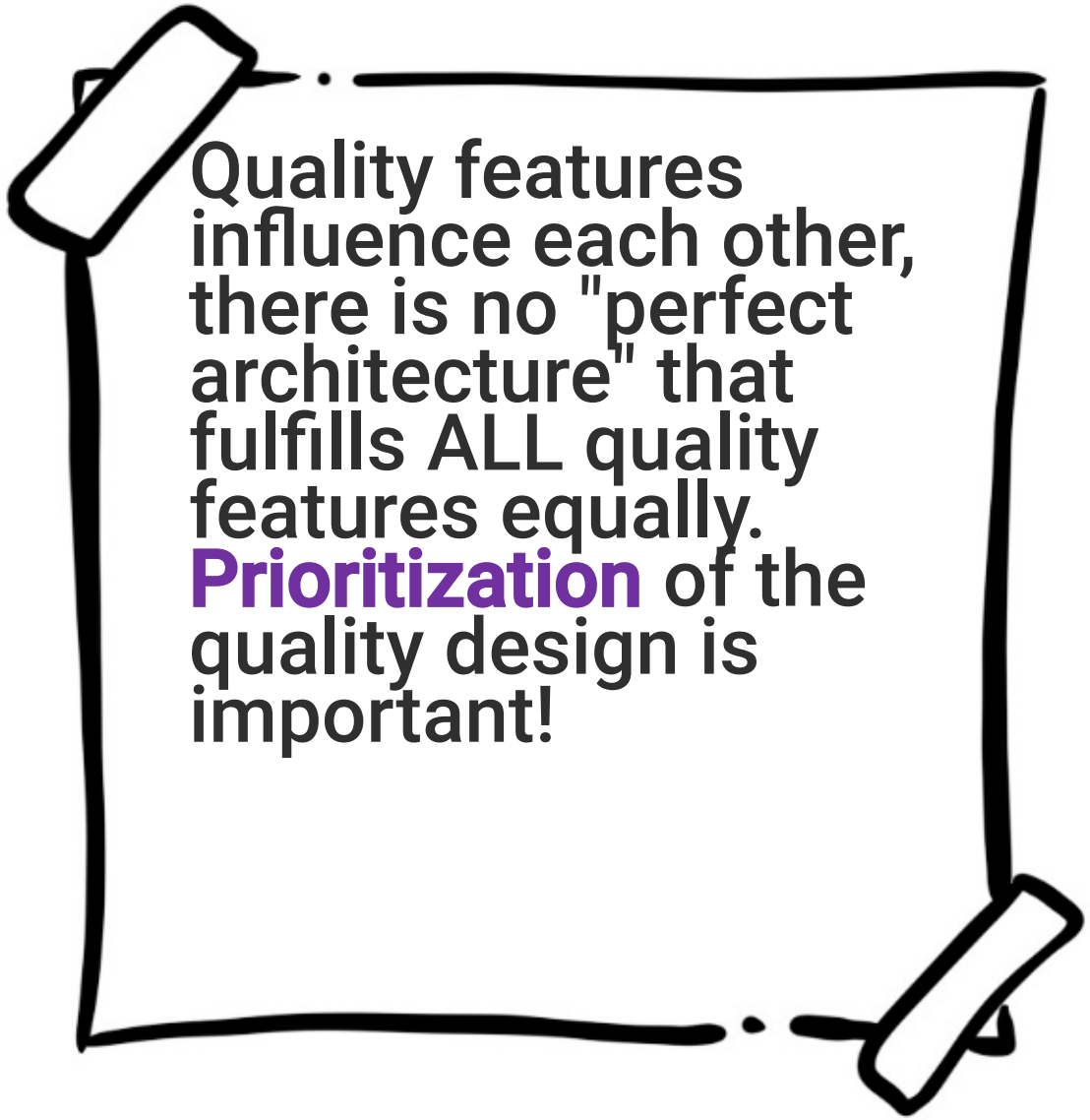
How do I derive a good software architecture?

[Ludewig, 2013]

- In general: A software architecture is good if the **functional and non-functional requirements** can be fulfilled.
- There is no design method that guarantees a good software architecture, but there are a number of **proven design principles and design patterns** that help to answer the following questions, among others:
  - What **criteria** should be used to divide the system into components?
  - Which aspects should be **summarized** in components?
  - Which services should components offer to the **outside world** at their interface, which aspects must be **protected?**
  - How should the components **interact** with each other?
  - How should components be **structured** and **refined**?

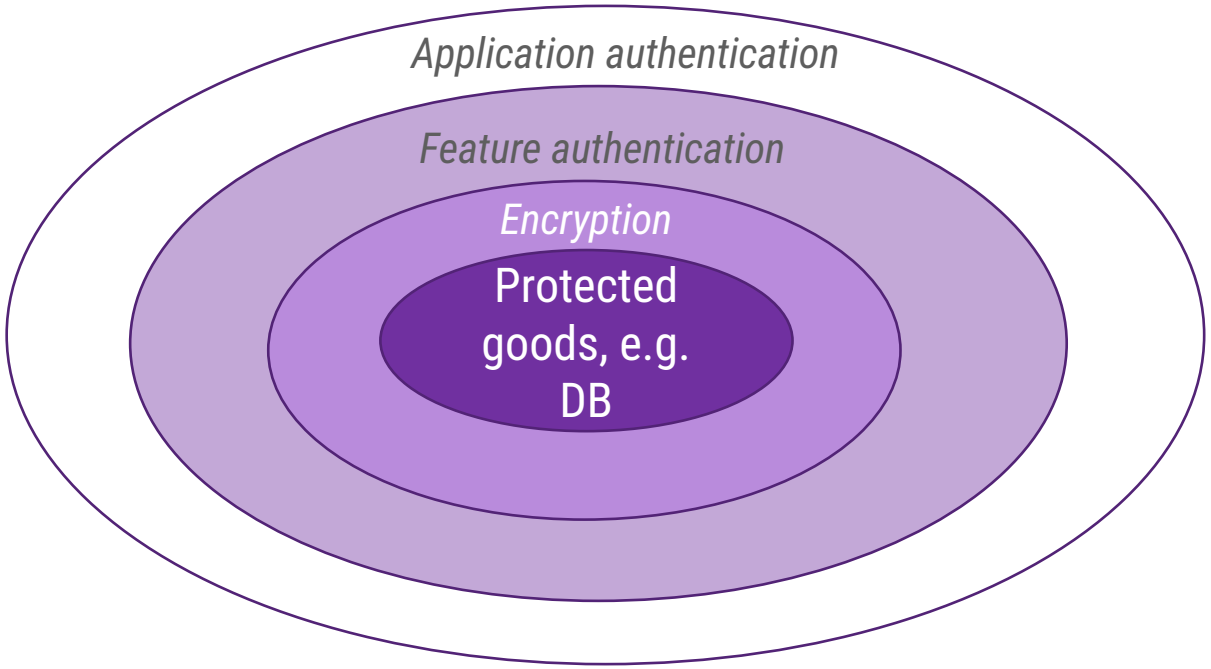



Architecture decisions have an enormous influence on the quality characteristics of a software system


Quality features influence each other, there is no "perfect architecture" that fulfills ALL quality features equally.
Prioritization of the quality design is important!

Example: Security vs. usability

- Use of multiple authentication levels → Increases security
- Affects usability → Users use weak passwords that they can remember, do not log out, use the same passwords, etc.



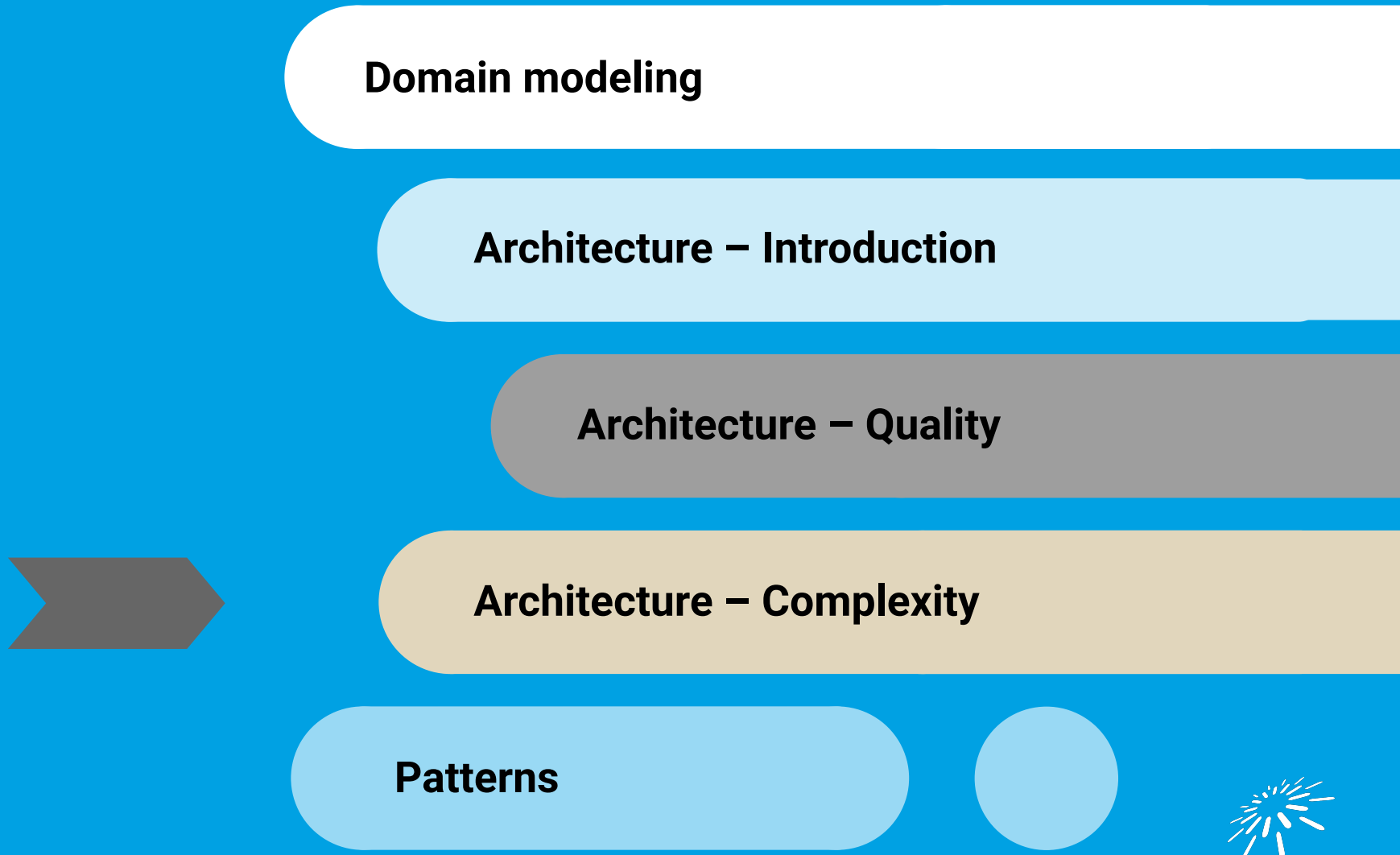
Example: Reliability vs. maintainability

- Availability of a system
  - % of the time the system is available.
- A system that should be available 99.9% of the time
  - system is available 86313 out of 86400 seconds a day.

How do you achieve this?

Example: Reliability vs. maintainability

- Availability of a system
  - % of the time the system is available.
- A system that should be available 99.9% of the time
  - system is available 86313 out of 86400 seconds a day.
- Highly redundant systems
  - E.g. online banking
  - With all the disadvantages of duplicates (maintenance, costs, ...)




Architecture has an influence on the complexity of the software system to be developed!

Check-In

How can complexity be reduced?

Reducing complexity: Coupling and cohesion

[Ludewig, 2013]

- The architect of a concert hall endeavors to build the hall in such a way that the acoustic disturbance from outside is extremely low, the audibility inside the hall is extremely high.

Reducing complexity: Coupling and cohesion

[Ludewig, 2013]

- In the software architect's work, this corresponds to the division into modules in such a way that
  - minimize the coupling (i.e. the width and complexity of the interfaces) between the modules,
  - the cohesion (i.e. the relationship between the parts of a module) is as high as possible.

Reducing complexity: Coupling and cohesion

What does the picture want to tell us in terms of coupling and cohesion?

Reducing complexity - principles

[Sommerville, 2020]

- Separation of concerns
  - Combine relevant architectural concerns and responsibilities into groups of related functions, e.g. authentication, DB management, system monitoring, etc.
  - Single responsibility principle: A component **is responsible only for one task**

Reducing complexity - principles

[Sommerville, 2020]

- Separation of concerns
  - Combine relevant architectural concerns and responsibilities into groups of related functions, e.g. authentication, DB management, system monitoring, etc.
  - A component is **responsible only for one task**
- Don't repeat yourself
  - Do not **duplicate** any functionality, this makes maintenance more difficult

→ Why is that?

Reducing complexity - principles

[Sommerville, 2020]

- Separation of concerns
  - Combine relevant architectural concerns and responsibilities into groups of related functions, e.g. authentication, DB management, system monitoring, etc.
  - A component is responsible only for one task
- Don't repeat yourself
  - Do not duplicate any functionality, this makes maintenance more difficult
- Ensure **stable (small) interfaces**
  - Changes to interfaces mean that all components that implement this interface may have to be changed.

Reducing complexity - Principles - Hierarchical structure

→ Ludewig 2013

- The hierarchical structure is a proven method of reducing complexity.
- A hierarchy is a structure of elements that are ranked by a (hierarchy-forming) relationship.

Reducing complexity - Principles - Hierarchical structure

→ Ludewig 2013

- Aggregation hierarchy
  - Organizes a system into its components; it is also called the "whole-part hierarchy".
- Layer hierarchy
  - Arranges components (layers) in such a way that each layer builds on exactly one layer below it and forms the basis for exactly one layer above it.

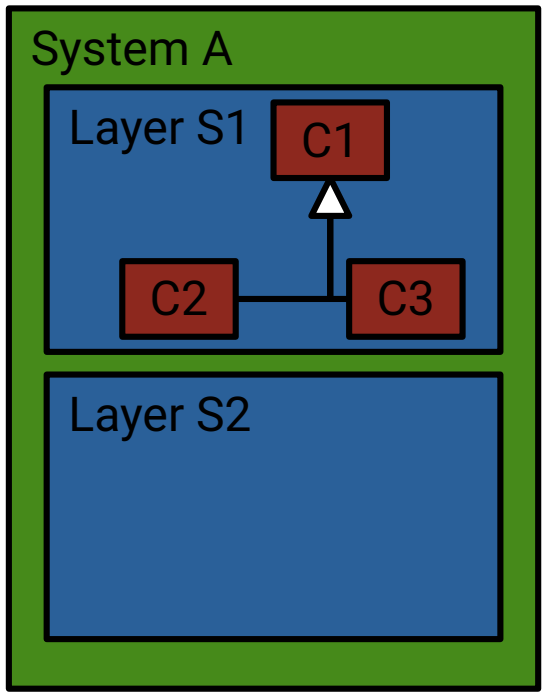
Reducing complexity - Principles - Hierarchical structure

[Ludewig, 2013]

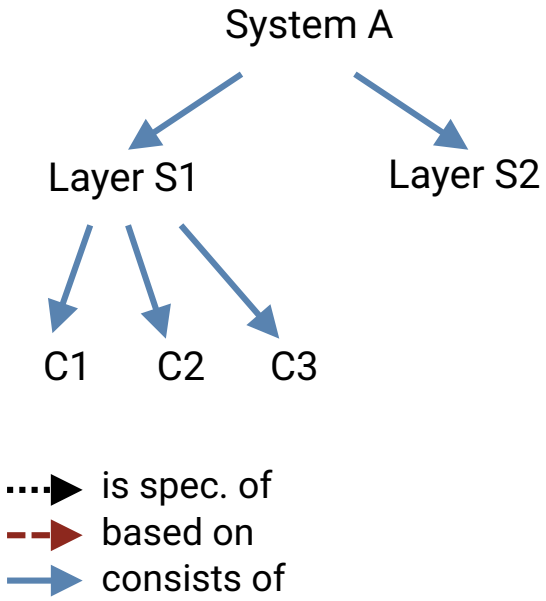
- Generalization hierarchy
  - Organizes components according to characteristics (methods and attributes) by combining fundamental, common characteristics of several components in one universal component.
  - Specialized components derived from these take over these characteristics and add special ones. This means that the fundamental features are only defined once.
  - Object-oriented design focuses on generalization hierarchies of classes and interfaces

Reducing complexity - principles - example of hierarchies

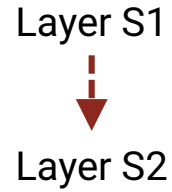
Ludewig, 2013



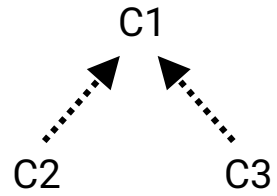
Aggregation Hierarchy



Layer Hierarchy



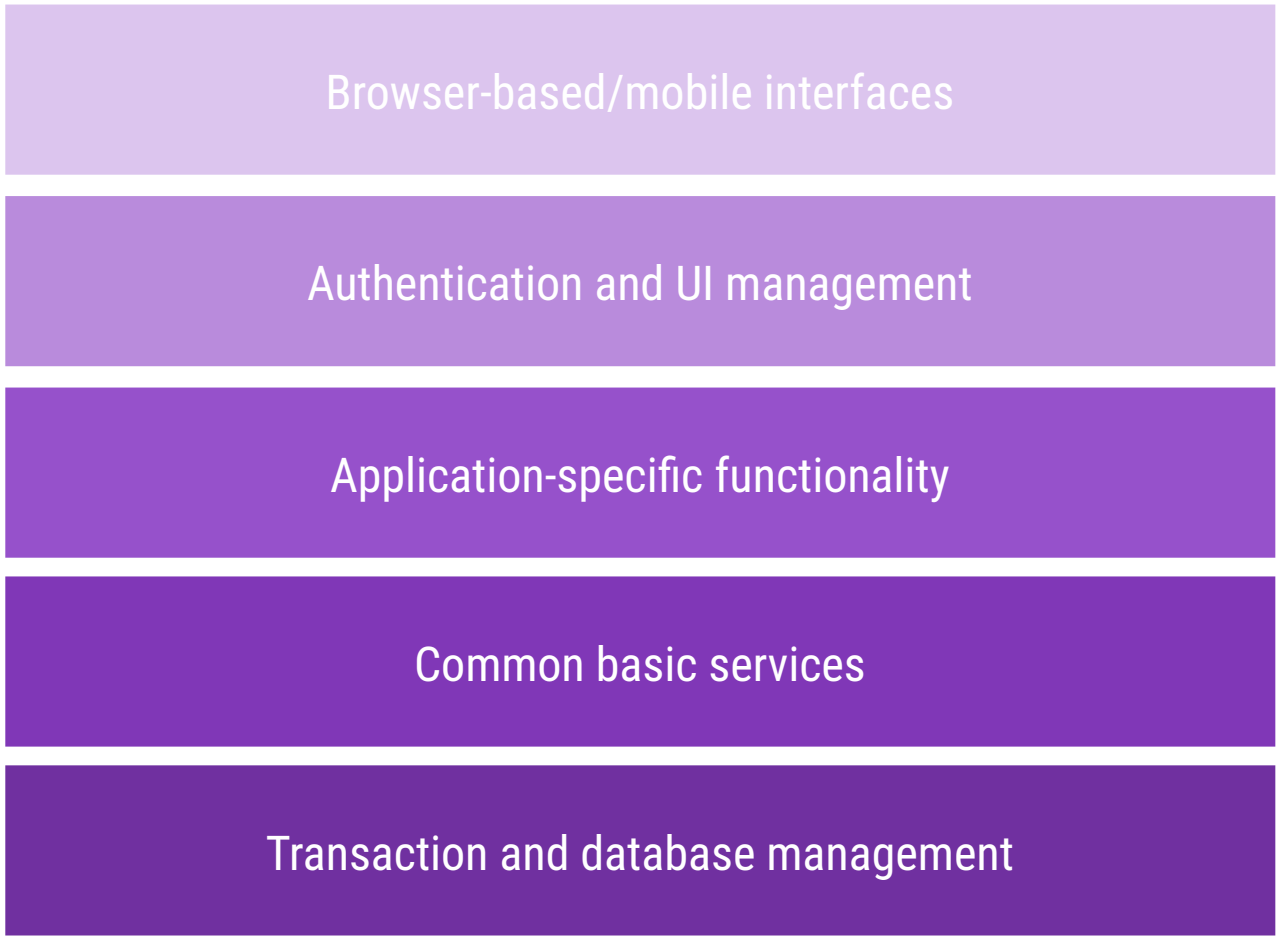
Generalization Hierarchy



Layers

- Domain modeling
- Architecture – Introduction
- Architecture – Quality
- Architecture – Complexity
- Patterns

A general layered architecture for web-based applications



E.g. web browser interfaces, JavaScript components for local actions, e.g. input validation.

Management layer for the user interface, components for authentication, creation of web pages, etc.

Application layer. This is where the "business logic" takes place.

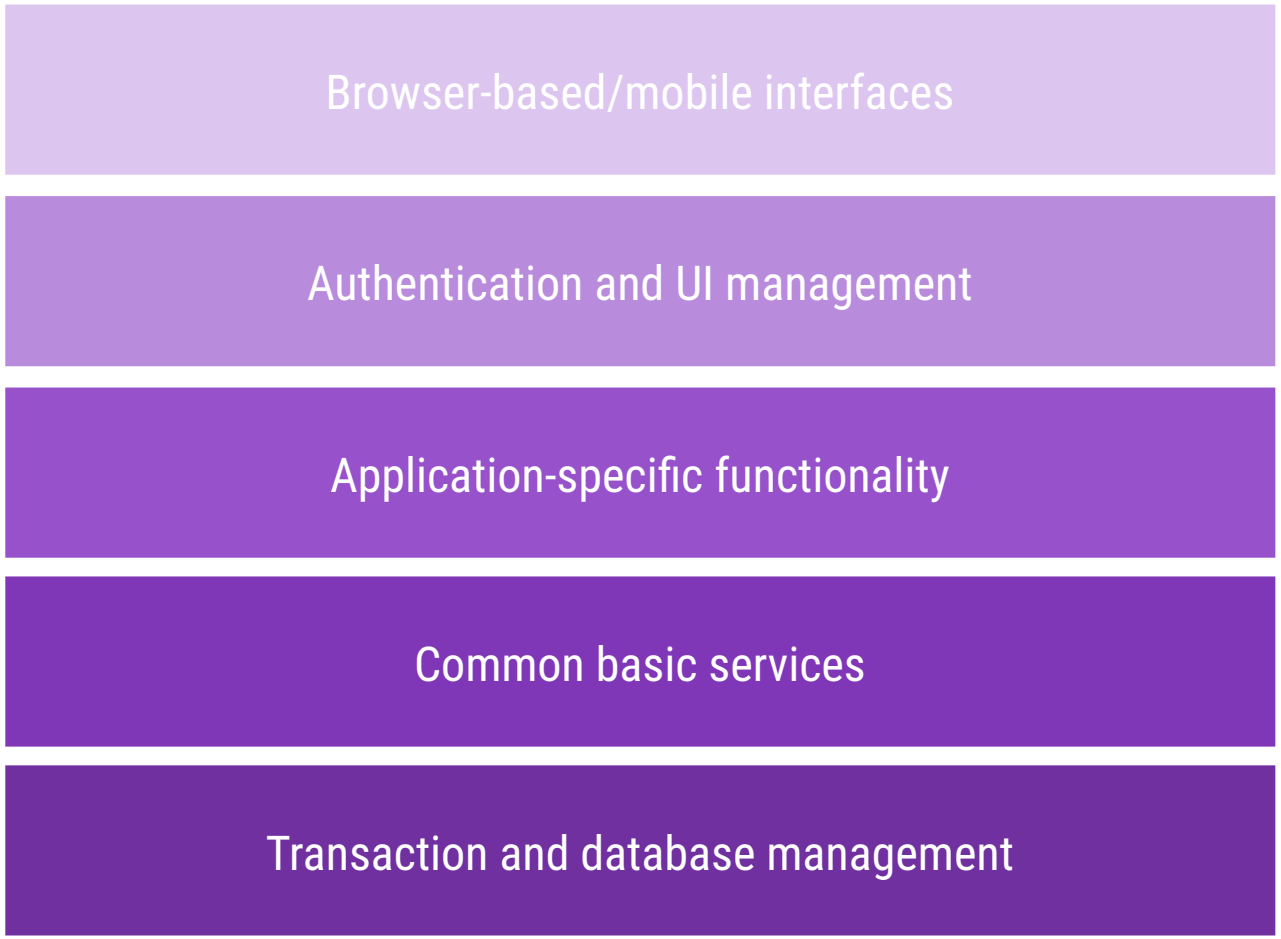
Reuse of basic functionalities required by the application layer.

Database layer that provides services such as transaction management and recovery.

Source: Sommerville, 2020, Ludewig, 2013

TH Aschaffenburg
university of applied sciences

A general layered architecture for web-based applications

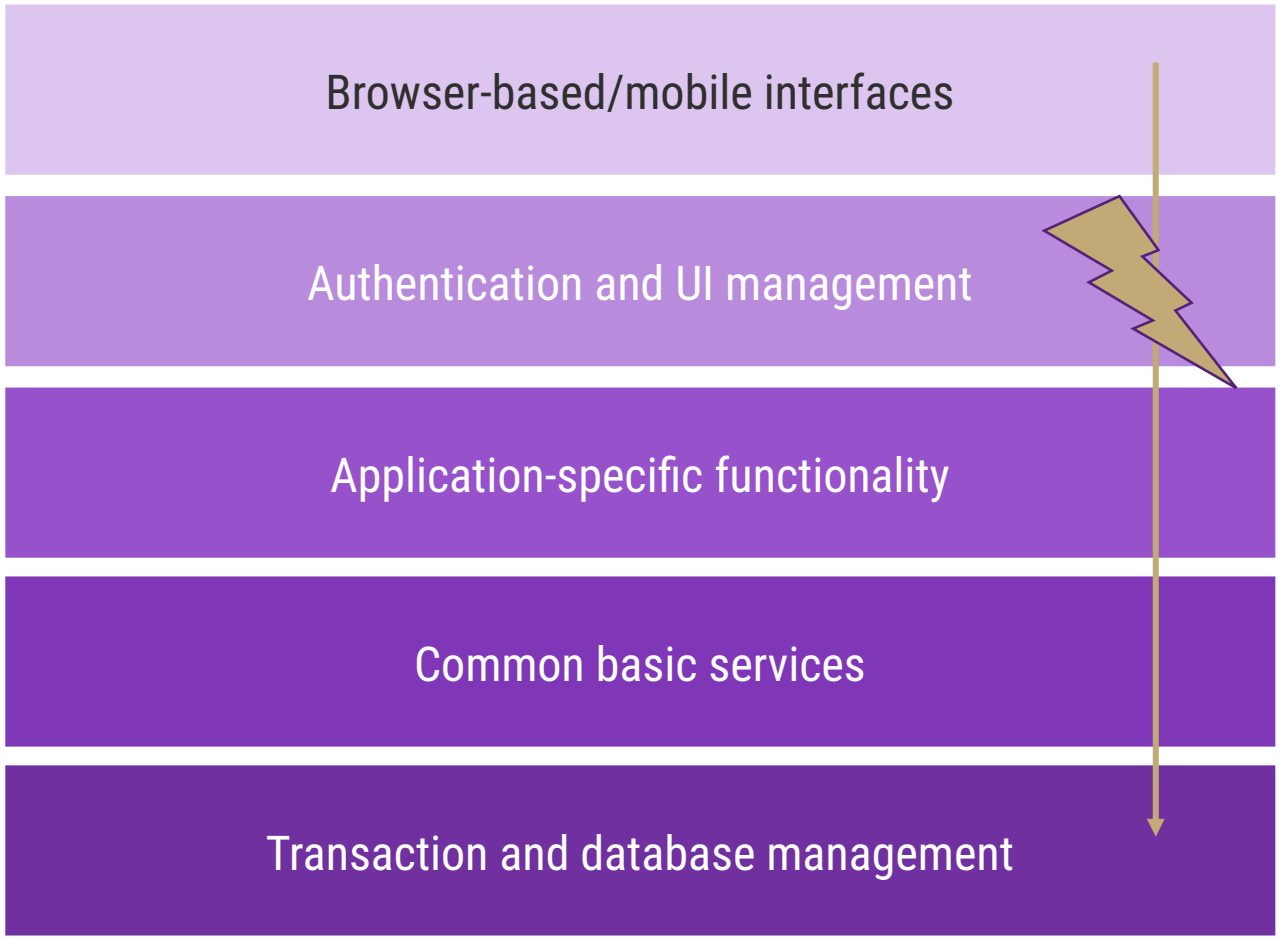


Source: Sommerville, 2020, Ludewig, 2013

- A layer combines components that logically belong together.
- A layer provides services that are offered at the (upper) interface of the layer.
- The services of a layer can only be used by components of the layer directly above it.

TH Aschaffenburg
university of applied sciences

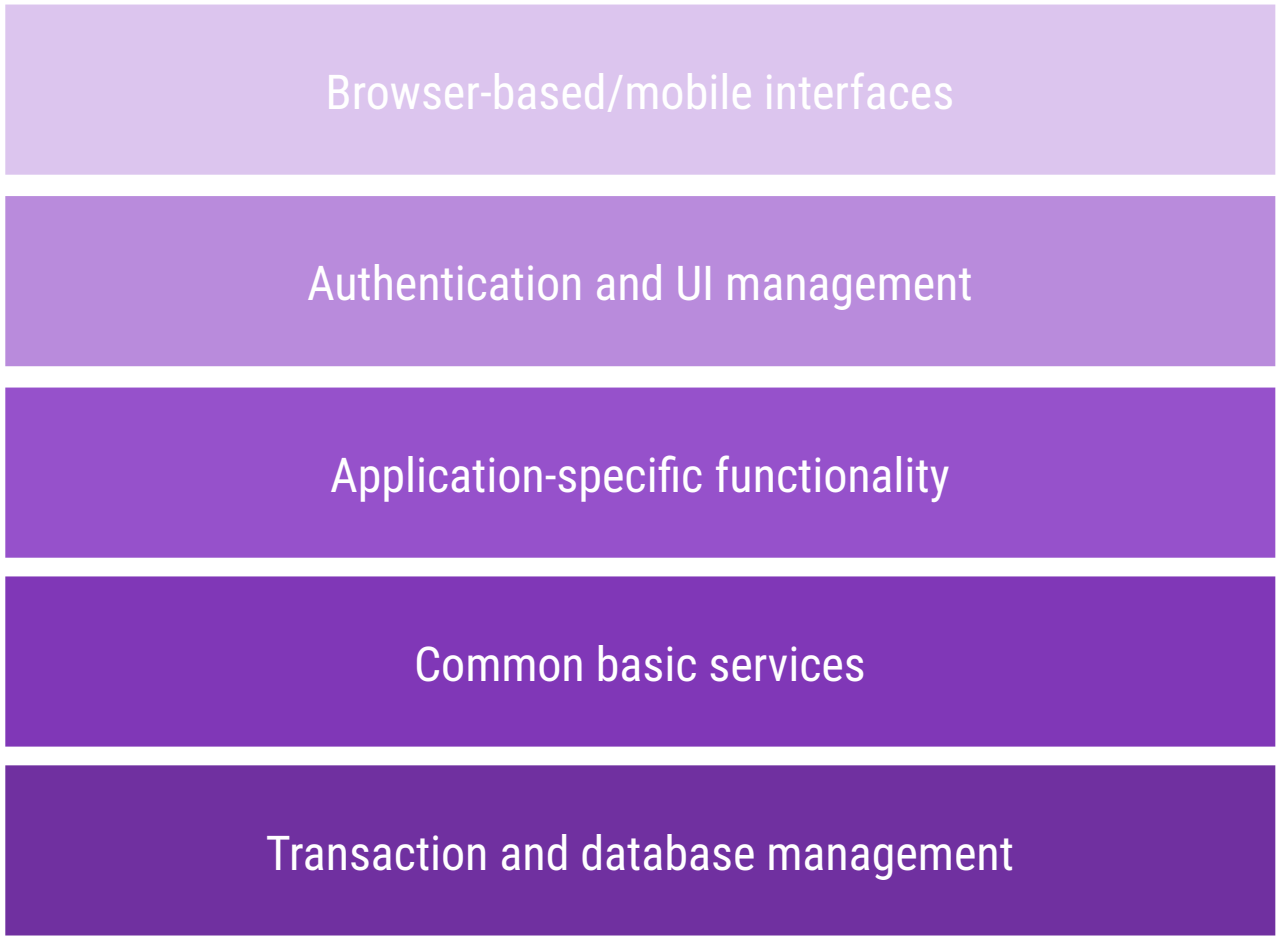
A general layered architecture for web-based applications



Source: Sommerville, 2020, Ludewig, 2013

- A layer combines components that logically belong together.
- A layer provides services that are offered at the (upper) interface of the layer.
- If strictly layered: The services of a layer can only be used by components of the layer directly above it.

A general layered architecture for web-based applications

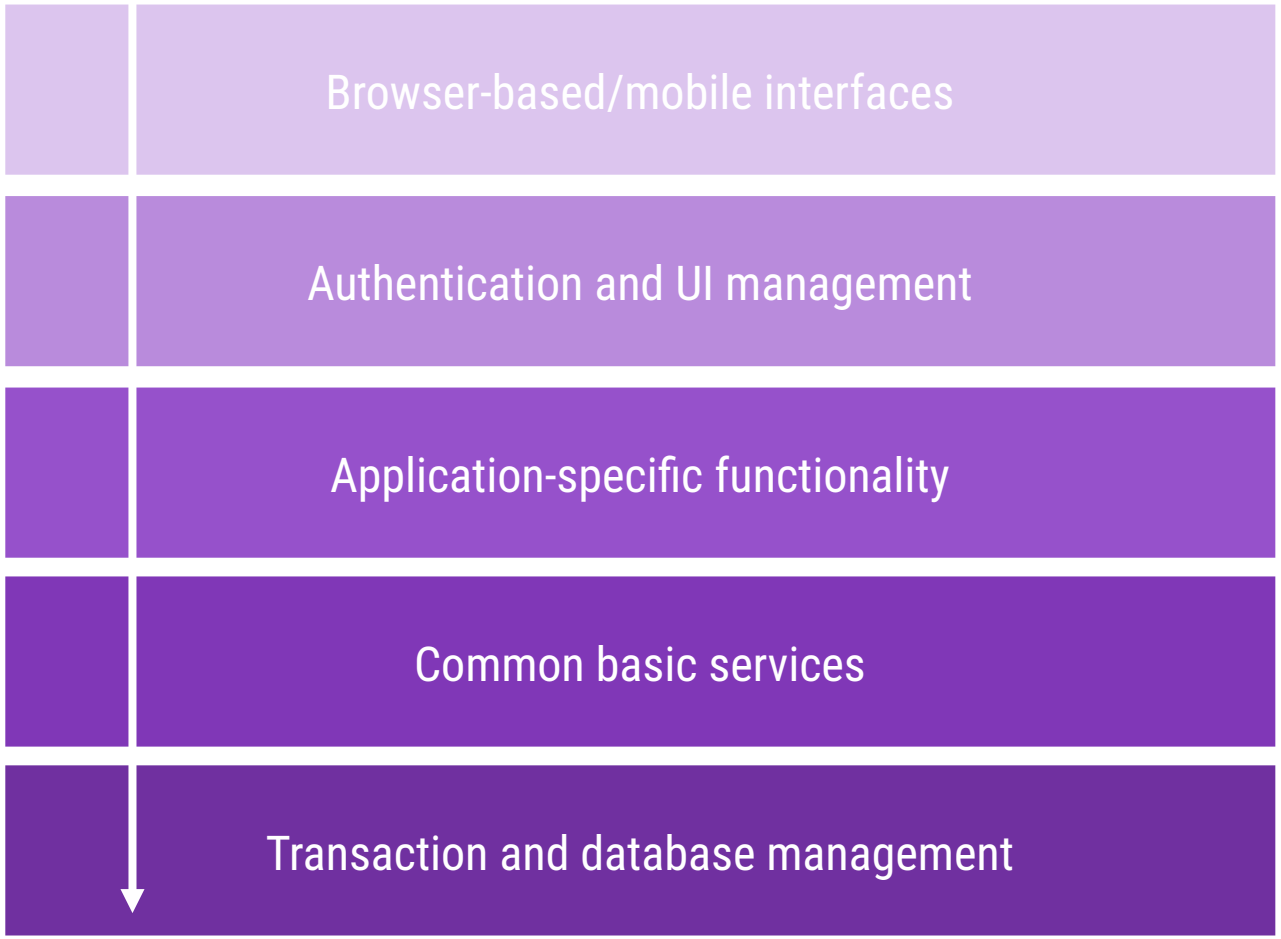


Source: Sommerville, 2020, Ludewig, 2013

- Changes in a layer should only affect the layers above it, but not the layers below.
- If strictly layered: Layers are built (directly) on top of each other; access through several layers is not permitted.
- Layers are only coupled if they are adjacent. However, this coupling is also low due to the encapsulation of the operations. Changes therefore usually only have a local effect (within a layer).

Cross-Cutting Concerns

E.g. security

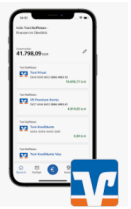


Source: Sommerville, 2020, Ludewig, 2013

- Some responsibilities are "cross-cutting" and must be considered in each layer → Cross-Cutting Concerns (e.g. security, performance, reliability).
- Typical quality features are CCC
- Example: Security attacks can happen at any level → Protection mechanisms must be implemented at every level.

TH Aschaffenburg
university of applied sciences

A general layered architecture for web-based applications



Browser-based/mobile interfaces

Authentication and UI management

Application-specific functionality

Common basic services

Transaction and database management

Source: Sommerville, 2020, Ludewig, 2013

E.g. web browser interfaces, JavaScript components **for local actions**, e.g. input validation, e.g. *input validation amount*.

Management layer for the **user interface**, components for **authentication**, creation of web pages, etc., *authentication, structure of screens, pages*

Application layer. This is where the "**business logic**" takes place, *e.g. transferring money, displaying account balances, creating standing orders, etc.*.

Reuse of basic functionalities required by the application layer, *e.g. customer, account*.

Database layer, which provides services such as transaction management and recovery, *persistence layer*.

TH Aschaffenburg
university of applied sciences

Three-layer architecture

- Many interactive software systems are made up of the following three layers:
  - Presentation layer,
  - Application layer and
  - Data storage layer.
- The presentation layer implements the user interface, displays information and controls the user-system interaction.
- The application layer contains all components that implement the technical functionality. They are designed in such a way that they do not contain any information about the presentation.
- Below this is the data storage layer. It ensures permanent storage, usually in a database. Storage details are hidden in this layer!

Further principles

[Sommerville, 2020]

- KISS
  - Keep it simple and stupid (Originally: Keep it short and simple)

  → What does that mean? Why is it important?

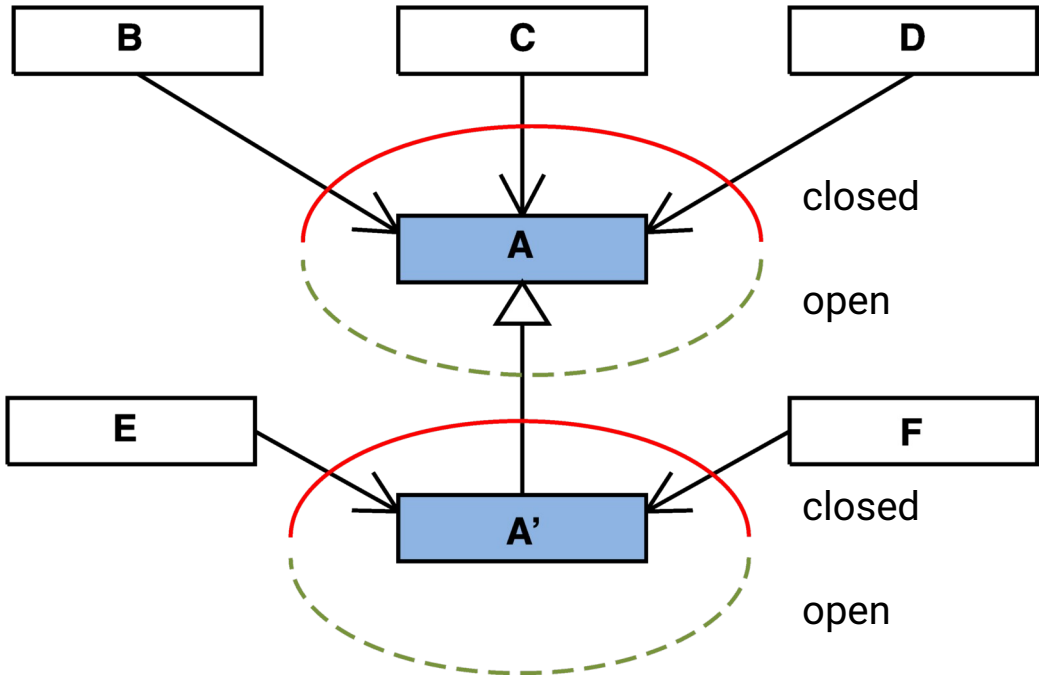
Further principles

[Sommerville, 2020]

- KISS
  - Keep it simple and stupid
- Abstraction
  - The basis of many design principles is abstraction.
  - By creating abstractions, we concentrate on the essentials and ignore the non-essentials.

The open-closed principle

Open for expansion, closed for change



- Extensions can be made without having to change existing code
- What constructs do you know that support this principle?

Source: Ludewig, 2013

Domain modeling

Architecture – Introduction

Architecture – Quality

Architecture – Complexity

Patterns

Design patterns

- Architecture patterns: offer solutions for non-trivial problems at the level of the course granular design (architecture in the narrower sense) of components. (e.g. layer architecture, client-server architecture, MVC model-view-controller)
- Design patterns: offer solutions for non-trivial problems at the level of detailed component design.
- Both are formulated independently of a specific language, but are usually based on object-oriented concepts.

Check-In

What are the advantages of design patterns?

Design patterns

- Advantages
  - The design patterns give us the opportunity to use our experience and implement tried-and-tested solutions.
  - They help us to consider non-functional design, such as maintainability or reusability, in the architectural design.
  - They create a vocabulary for the design and facilitate documentation and communication about architectures.
  - They can be used as an analysis tool when reengineering existing software.
- Warning: Understanding a design pattern is easy. However, you need a lot of design experience to use the patterns sensibly

Source: Ludewig, 2013

Libraries

> *A **class library** consists of a set of classes that are reusable and offer generally usable functionality, i.e. independent of the application context.*
>
> Ludewig, 2013

From the application's point of view, the classes of a library are used directly, or the classes of the application inherit from the library classes.

Frameworks

> *A **framework** is an architecture of class hierarchies that provides a general generic solution for similar problems in a specific context.*
>
> **Züllighoven (2005)**

- If very similar applications are developed repeatedly, the applications should be developed on the basis of a **generic solution.**
- A framework has defined interfaces at which the generic solution can be extended with application-specific code.
- UI frameworks, test frameworks, etc.
- Frameworks rely heavily on the **inversion of control** principle.

Quality and architecture

- The quality of the architecture significantly determines the quality and costs of the developed system, and does so in the long term.
- Therefore:
  - Prioritize quality requirements
  - Carry out architecture reviews
  - Evaluate the extent to which the quality requirements are met
  - Document the WHY of an architectural decision

Object-oriented analysis and design

[Ludewig, 2013]

- Means finding functional units and designing (classes) in such a way that they represent the relevant units and concepts of the application area under consideration (=part of the reality to be modeled).
- Central activities of object-oriented design:
  - Identifying **objects** and classes
  - Defining the **behavior** of objects and classes
  - Identifying **relationships** between the classes
  - Defining the **interfaces** between the classes
- The design is based on the (object-oriented) current state analysis, which aims to
  - identify the technical processes and model them in the form of use cases.
  - Describe the units using an object-oriented conceptual model.

Task

Take a few minutes and answer the following questions.

In writing!

- **Question 1**: Why is it important to minimize the complexity of a system?
- **Question 2**: An architecture that takes security aspects into account can either be based on a centralized model, in which all sensitive information is stored in one place, or on a distributed model, in which information is distributed everywhere and stored in many different places. Write down one advantage and one disadvantage of each solution.
- **Question 3**: What is the Separation of Concerns principle?

- **Architecture decisions** have an enormous influence on the **quality characteristics** of a software system.
- Quality features influence each other, there is no "perfect architecture" that fulfills ALL quality features equally. **Prioritization** of the quality design is important!
- Architecture has an influence on the **complexity** of the software system to be developed!
- In order to reduce complexity, a high degree of **cohesion** within the components (packages, modules, classes) and loose **coupling** between them must be ensured.
- **Principles** provide guidelines on how complexity can be reduced.
- **Design patterns** exist at different levels of abstraction.

Literature

- **[Ludewig 2013]** Ludewig, Lichter: Software Engineering. Basics, people, processes, techniques, dpunkt.verlag.
- **[Paech 2021]** Barbara Paech: Lecture Software Engineering, Uni-Heidelberg.
- **[Balzert, 2009]** Helmut Balzert: Lehrbuch der Softwaretechnik, Basiskonzepte und Requirements Engineering, 3rd edition, 2009, Springer.
- **[Sommerville 2018]** Ian Sommerville: Software Engineering, Pearson.
- **[Rupp, 2021]** Rupp & the SOPHISTs: Requirements Engineering and Management, Hanser Verlag, 7th edition, Hanser Verlag, 2021.

Thank you for your attention!

Software Engineering

Prof. Dr. J. v. Kistowski

TH Aschaffenburg
university of applied sciences