Introduction to Software Engineering

04 System Design I

Pramod Bhatotia, Stephan Krusche

17 May 2022
Technical University of Munich
https://ase.in.tum.de

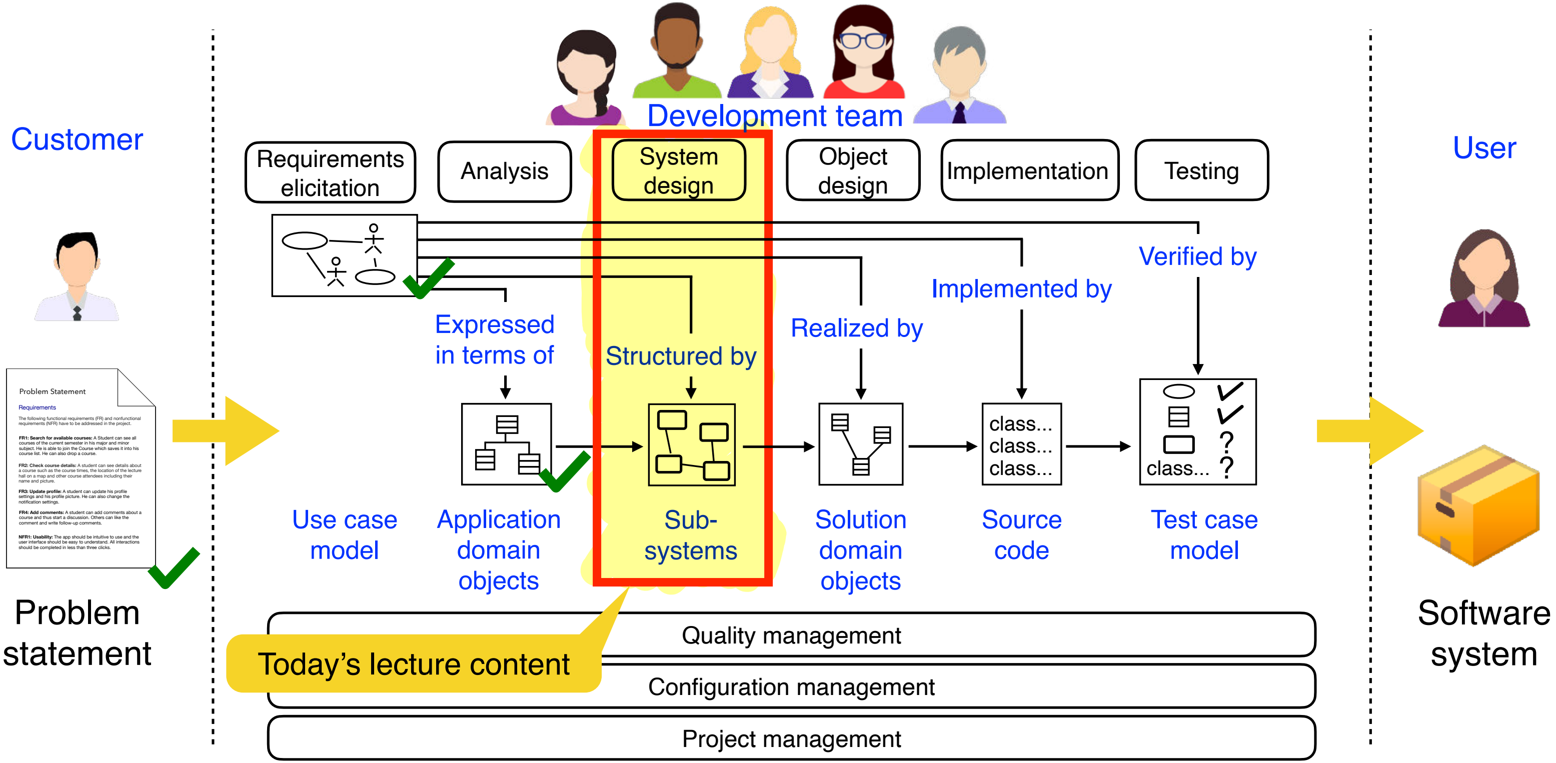
Roadmap of the lecture

- **Context and assumptions**
  - You have a good understanding of requirements elicitation and analysis
  - You know the most important activities of model-based software engineering
  - You can explain Scrum, UML diagrams, JavaFX, Gradle, and usability
- **Learning goals: at the end of this lecture you are able to**
  - Explain the 8 most important issues in system design
  - Differentiate between nonfunctional requirements and design goals
  - Create an initial subsystem decomposition
  - Differentiate between coupling and cohesion
  - Apply the architectural styles: layered, client-server, REST

Course schedule (Garching)

#	Date	Subject
1	26.04.22	Introduction
2	03.05.22	Model-based Software Engineering
3	10.05.22	Requirements Analysis
4	17.05.22	System Design I
5	24.05.22	System Design II
6	31.05.22	Object Design I
	07.06.22	**Holiday (no lecture, no tutor groups)**
7	14.06.22	Object Design II
8	21.06.22	Testing
	28.06.22	**no lecture, no tutor groups**
9	05.07.22	Software Lifecycle Modeling
10	12.07.22	Software Configuration Management
11	19.07.22	Software Quality Management
12	26.07.22	Project Management

Overview of model based software engineering



Outline

- **Overview of system design**
- Design goals
- Hints for system design
- Subsystem decomposition
- Façade pattern
- Architectural styles
  - Layered architecture
  - Client server architecture
  - REST architectural style
- UML component diagrams

Design is difficult

- There are two ways of constructing a **software design** (Antony Hoare)
  - One way is to make it so **simple** that there are obviously no deficiencies 缺陷
  - The other way is to make it so **complicated** that there are no obvious deficiencies

Sir **Antony Hoare,** *1934
- Quicksort
- Hoare logic for verification
- CSP (Communicating sequential processes): modeling language for concurrent processes

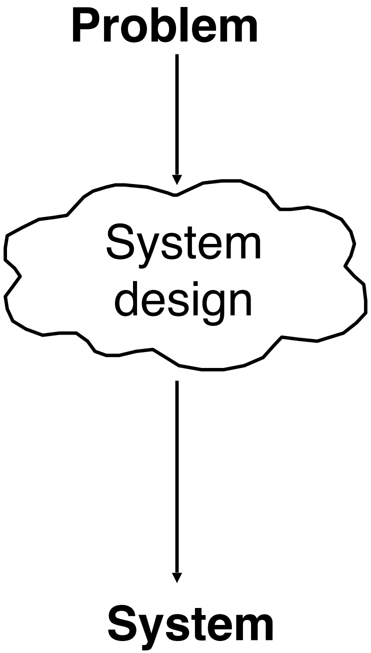
Why is design so difficult?

- Analysis: focuses on the application domain 域
- Design: focuses on the solution domain
  - Designing a computer system is different from designing an algorithm
  - The external interface (that is, the requirement) is less precisely defined, more complex, and subject to change
  - The system has a more complex internal structure, and hence many internal interfaces
  - The measure of success is less clear
- Design window: time in which design decisions have to be made

范围

The scope of system design

- Bridge the gap between a **problem** and a **system** in a manageable way
- **How?**
- Use divide & conquer
  - Identify **design goals**
  - Model the new system design as a set of **subsystems**
  - Address the major **design goals** first



System design: 8 issues

System design

1. Design goals
- Additional nonfunctional requirements
- Design trade-offs

2. Subsystem decomposition
- Layers vs. partitions
- Architectural style
- Cohesion & coupling

3. Concurrency 并发性
- Identification of parallelism (processes, threads)

4. Hardware/ software mapping
- Identification of nodes
- Special purpose systems
- Buy vs. build
- Network connectivity

cloud

5. Persistent data management
- Storing persistent objects
- Filesystem vs. database

6. Global resource handling
- Access control
- ACL vs. capabilities
- Security

access security

7. Software control
- Monolithic
- Event-driven
- Conc. processes

8. Boundary conditions
- Initialization
- Termination
- Failure

Network ...

From analysis to system design

Nonfunctional requirements

Functional model

Dynamic model

Object model

Functional model

Dynamic model

Main influence of requirements analysis artifacts to system design

Requirements analysis		**System design**
Nonfunctional requirements	➡	1. Design goals
Functional model	➡	2. Subsystem decomposition 8. Boundary conditions
Object model	➡	4. Hardware/software mapping 5. Persistent data management
Dynamic model	➡	3. Concurrency 6. Global resource handling 7. Software control

This overview shows the main influence

From analysis to system design

Nonfunctional requirements

1. Design goals
- Additional nonfunctional requirements
- Design trade-offs

Functional model

2. Subsystem decomposition
- Layers vs. partitions
- Architectural style
- Cohesion & coupling

Dynamic model

3. Concurrency
- Identification of parallelism

Object model

4. Hardware/ software mapping
- Identification of nodes
- Special purpose systems
- Buy vs. build
- Network connectivity

5. Persistent data management
- Storing persistent objects
- Filesystem vs. database

Dynamic model

7. Software control
- Monolithic
- Event-driven
- Conc. processes

6. Global resource handling
- Access control
- ACL vs. capabilities
- Security

Functional model

8. Boundary conditions
- Initialization
- Termination
- Failure

Outline

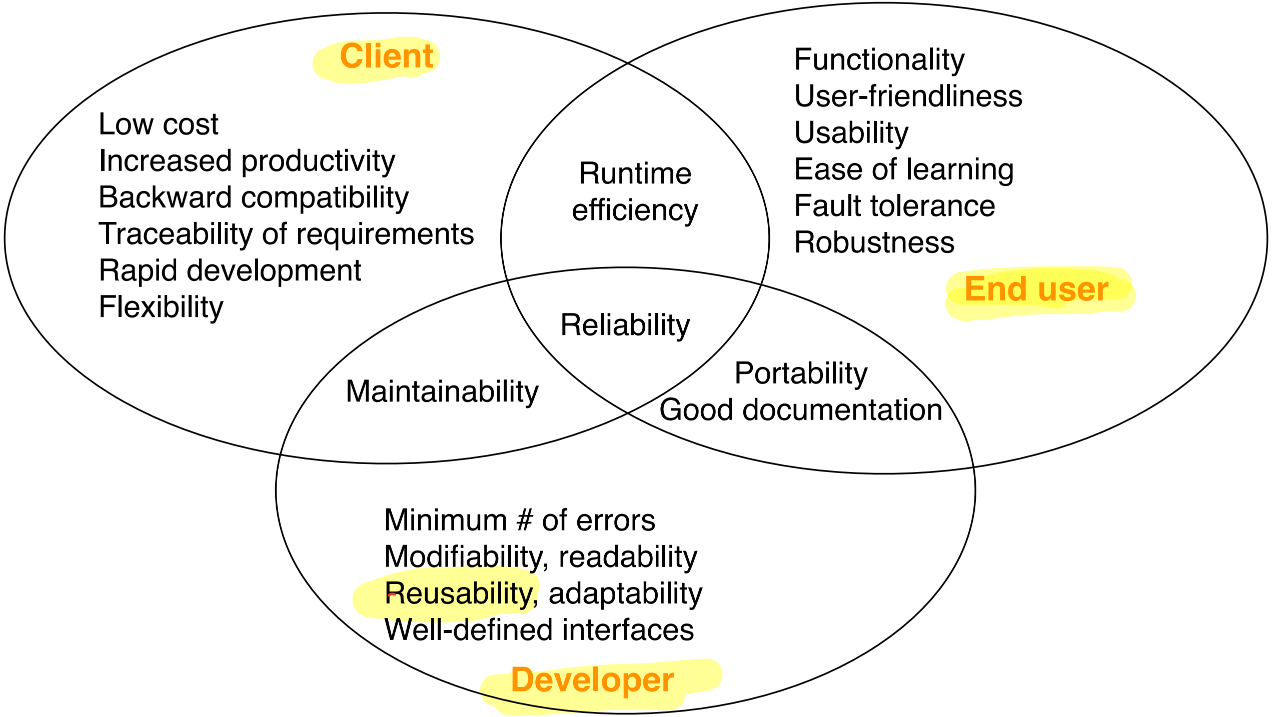
- Overview of system design
- **Design goals**
- Hints for system design
- Subsystem decomposition
- Façade pattern
- Architectural styles
    - Layered architecture
    - Client server architecture
    - REST architectural style
- UML component diagrams

Definition: design goal

- Design goals govern the system design activities
- As a starter: any **nonfunctional requirement** is a **design goal**
- Additional design goals are identified with respect to
  - Design **methodology**
  - Design metrics
  - Implementation goals
- Design goals often **conflict** with each other

  → Typical **design goal trade-offs**

Different types of design goals (examples)



Typical design goal trade-offs

- Functionality vs. usability
- Cost vs. robustness
- Efficiency vs. portability
- Rapid development vs. functionality
- Cost vs. reusability
- Backward compatibility vs. readability

Functionality vs. usability

Example: is a system with 100 functions usable?

Cost vs. reusability

- Assume you model the association between two classes with **one-to-one** multiplicity
  - Easy to code, low cost tests, not very reusable
- Moving from **one-to-one** multiplicity to a **many-to-many** multiplicity
  - Additional coding and testing costs

With design patterns, this trade-off is no longer that painful.
You can get reusability with low cost if you use design patterns!

More details in **Lecture 07** on **Object design**

Outline

- Overview of system design
- Design goals
- **Hints for system design**
- Subsystem decomposition
- Façade pattern
- Architectural styles
  - Layered architecture
  - Client server architecture
  - REST architectural style
- UML component diagrams

Hints for system design

- Functionality
  - Meeting the specification via an interface
- Speed (performance)
  - Build high performance systems
- Fault-tolerance (reliability)
  - Fault are the norms, not an exception

Butler Lampson
Turing Award 1992

Hints for Computer System Design
Original version: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/acrobat-17.pdf
Revised version: https://eecs.ceas.uc.edu/~wilseypa/research/lampson-20.pdf

Functionality

- How to obtain the right functionality from a system?
  - Define **an interface** that separates **an implementation** of some abstraction from **the clients** who use the abstraction
- Interface
  - Defining interfaces is the most important part of system design
  - Three conflicting requirements
    1. Simple
    2. Complete (满足 specification)
    3. Admit a small and fast implementation

Hints: interface design

- **Keep it simple**
  - Capture minimal essentials of an abstraction
  - Do one thing at a time, but do it well
  - Don’t generalize (leads to large, slow and complicated design)
  - Make it fast, rather general or powerful
- **Keep basic interfaces stable**
  - Interfaces embody frequently shared assumptions
  - It is desirable **not** to change the interface
- **Handle normal and worst cases separately as a rule**
  - The normal case must be fast
  - The worst case must make some progress

Hints: performance fast

- **Split resources** in a fixed way if in doubt, rather than sharing them
  资源冲突
- **Cache answers** to expensive computations, rather than doing them over
  (提前储存)
- Use **hints to speed up** normal execution
- **Compute in background** when possible
- Use **batch processing** if possible

Hints: fault-tolerance

- **End-to-end** argument in system design
  - Functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level
- **Log updates** to record the truth about the state of an object

  (保存错时)
  - An append-only efficient data structure to make systems fault tolerance
- **Safety first** resource allocation than to attain an optimum
  - Shed load to control demand, rather than allowing the system to become overloaded
- Make actions **atomic**
  - An atomic action (or a transaction) is one that either completes or has no effect

End-to-end argument in system design:
https://web.mit.edu/Saltzer/www/publications/endtoend/endtoend.pdf

Outline

- Overview of system design
- Design goals
- Hints for system design
- **Subsystem decomposition**
- Façade pattern
- Architectural styles
  - Layered architecture
  - Client server architecture
  - REST architectural style
- UML component diagrams

Definitions: subsystem and service

- **Subsystem**
  - Collection of classes, associations, operations, events that are closely interrelated with each other
  - The classes in the analysis object model are the **seeds** for subsystems
- **Service**
  - A group of externally visible operations provided by one subsystem (also called subsystem interface)
  - The use cases in the functional model provide the **seeds** for services

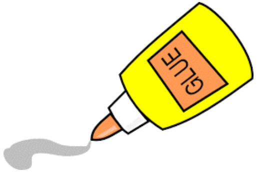
Subsystem interface

- **Subsystem interface**: set of fully typed UML operations
  - Specifies the interaction and information flow from and to subsystem boundaries, but not inside the subsystem
  - 具化 Refinement of service, should be well-defined and small
  - (by Code) Subsystem interfaces are defined during object design
- **Application programming interface (API)**
  - The API is the specification of the subsystem interface in a specific programming language
  - APIs are defined during object design
- The terms subsystem interface and API are often confused with each other

cohesion → coupling

Coupling and cohesion of subsystems 耦合

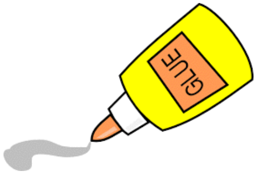
- **Goal:** reduce system complexity while allowing change
- **Cohesion** measures dependency between classes **within one** subsystem
  - **High cohesion**: the classes in the subsystem perform similar tasks and are related to each other via many associations
  - **Low cohesion**: lots of miscellaneous and auxiliary classes, almost no associations



- **Coupling** measures dependency between **multiple** subsystems
  - **High coupling**: changes in one subsystem will have a high impact on the other subsystem
  - **Low coupling**: a change in one subsystem does not affect any other subsystem

Coupling and cohesion of subsystems

- **Goal:** reduce system complexity while allowing change
- Cohesion measures dependency between classes **within one** subsystem
  - High cohesion: the classes in the subsystem perform similar tasks and are related to each other via many associations
  - Low cohesion: lots of miscellaneous and auxiliary classes, almost no associations



- Coupling measures dependency between **multiple** subsystems
  - High coupling: changes in one subsystem will have a high impact on the other subsystem
  - Low coupling: a change in one subsystem does not affect any other subsystem

Good system design

How to achieve **high cohesion**

- **High cohesion** can be achieved if most of the interaction is within subsystems, rather than across subsystem boundaries
- Question: does **class A** from **subsystem 1** always calls **class B** from **subsystem 2** for a specific service?
  - **Yes:** consider moving both classes **A** and **B** into the same subsystem
  - **No:** keep the classes in different subsystems

How to achieve **low coupling**

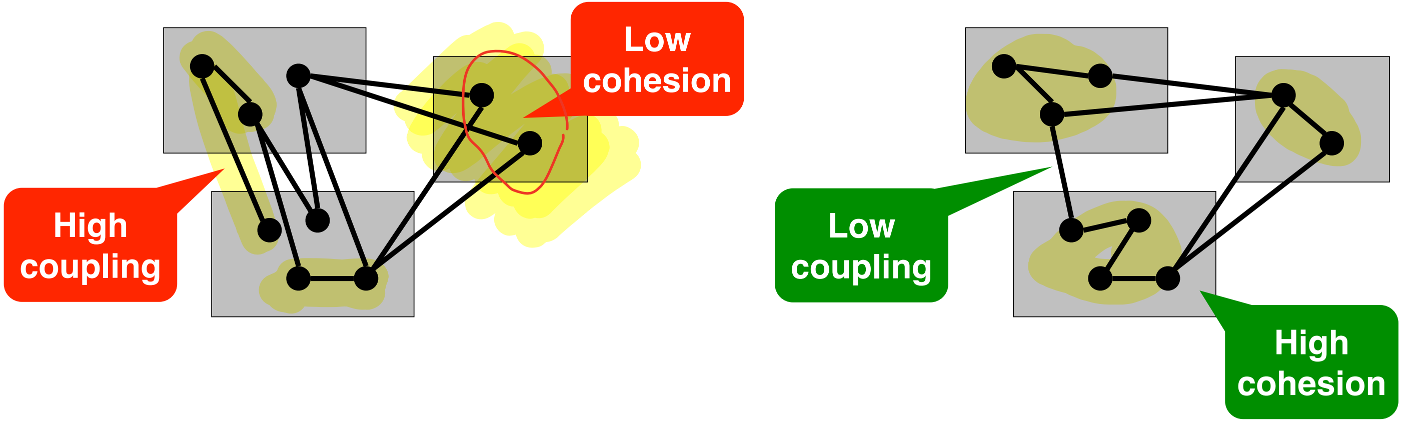
- **Low coupling** can be achieved if a calling class does not need to know anything about the internals of the called class (principle of information hiding, Parnas)
- The same principle can be applied in a larger scope on subsystems / module

每个System尽量
隐藏内部实现
内部方法

David Parnas, *1941,
Developed the concept of
modularity in design

Cohesion and coupling measure interdependence

- Cohesion measures the **interdependence** of objects in **one subsystem**
- Coupling measures the **interdependence** of objects between **different subsystems**



Achieving high cohesion and low coupling

High cohesion

- Operations work on the same attributes
- Operations implement a common abstraction or service

Low coupling

- Small interfaces
- Information hiding
- No global data
- Interactions are mostly within the subsystem rather than across subsystem boundaries

Example: subsystem decomposition of a compiler

Lexer

Responsibility

- Scans input file and provides stream of tokens
- Initializes symbol table (ST)
- Reports lexical errors

Subsystem interface

- `getNextToken(File, ST)`

Parser

Responsibility

- Parses stream of tokens and builds abstract syntax tree (AST)
- Enters symbol table information
- Reports syntax errors

Subsystem interface

- `getAbstractSyntaxTree(File, ST)`

A group of externally visible operations provided by a subsystem

Semantic analyzer

Responsibility

- Perform semantic analysis
- Fill symbol table
- Report type errors

Subsystem interface

- `performAnalysis(AST, ST)`

Dependency

Code generator

Responsibility

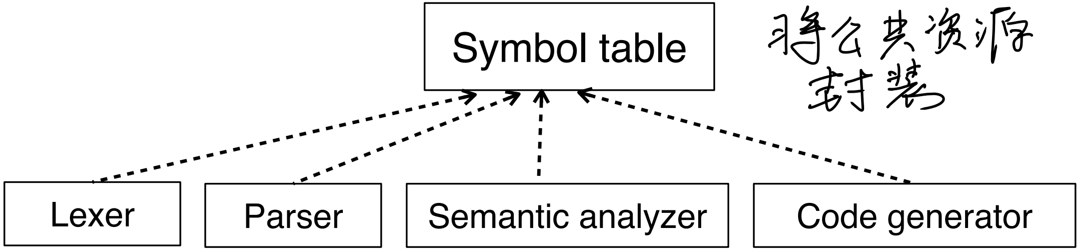
- Generate target code from analyzed syntax tree

Subsystem interface

- `generateCode(AST, ST)`

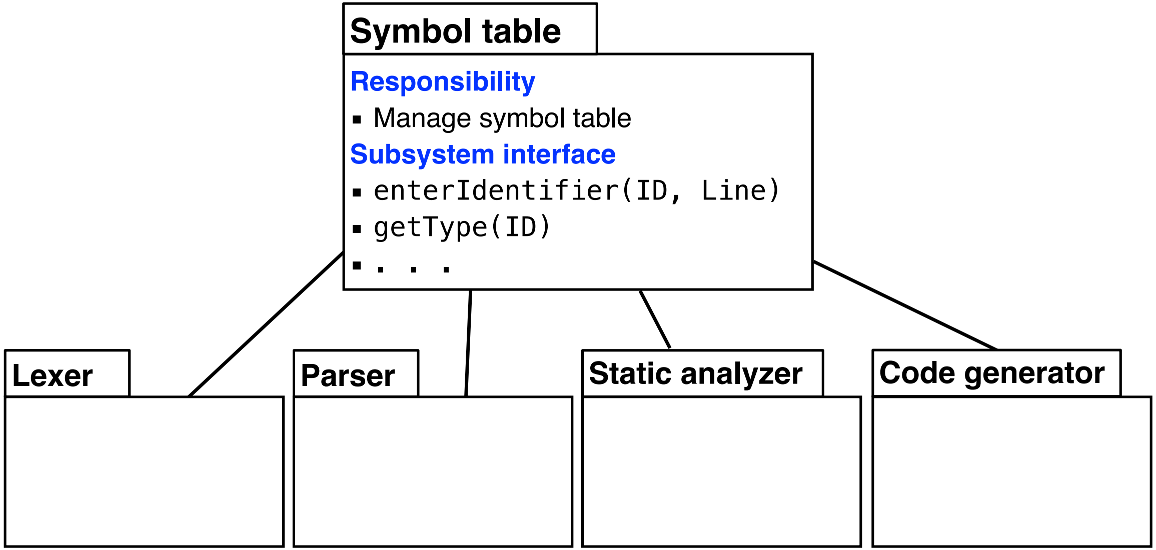
Coupling and cohesion in the compiler example

- Coupling
  - Small subsystem interfaces
  - **Coupling is ok**
- Cohesion
  - All subsystems read and update the **symbol table** shared data
  - Any change in the symbol table representation affects all subsystems
  - **Cohesion is bad**
- We improve the compiler design by introducing a separate subsystem symbol table



将公共资源封装

Compiler design example with improved cohesion



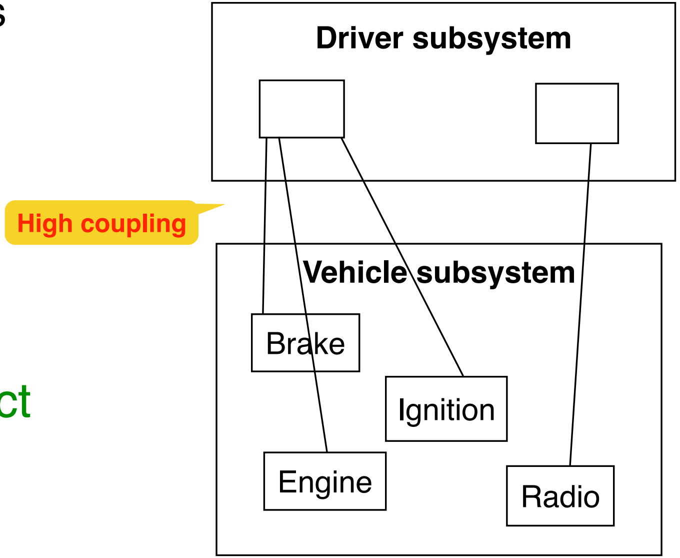
Outline

- Overview of system design
- Design goals
- Hints for system design
- Subsystem decomposition
- **Façade pattern**
- Architectural styles
  - Layered architecture
  - Client server architecture
  - REST architectural style
- UML component diagrams

Another design example

The **driver subsystem** can call any class operation in the **vehicle subsystem**

➡ **Problem: Spaghetti design!**



Solution: subsystem interface object

Subsystem interface object

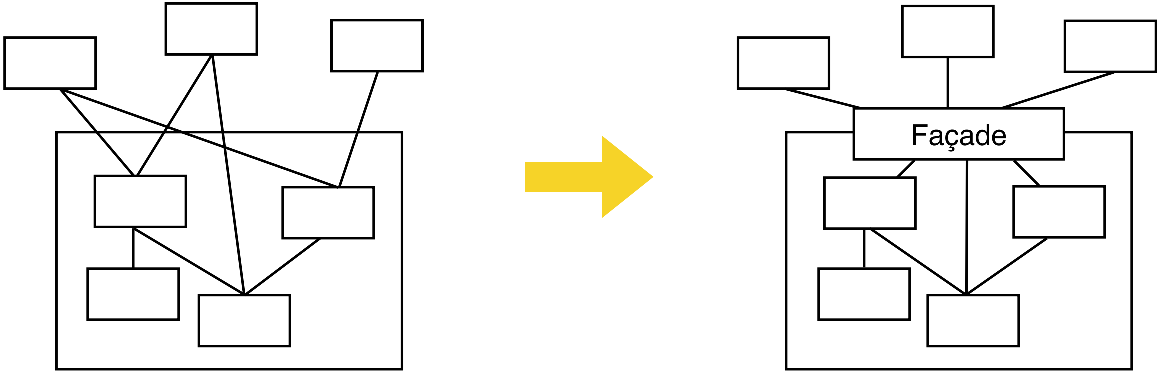
- The set of **public operations** provided by a subsystem
- The subsystem interface object describes **all services** of the subsystem interface
- A subsystem interface object can be realized with the **façade design pattern**

Façade design pattern: reduces coupling

统一

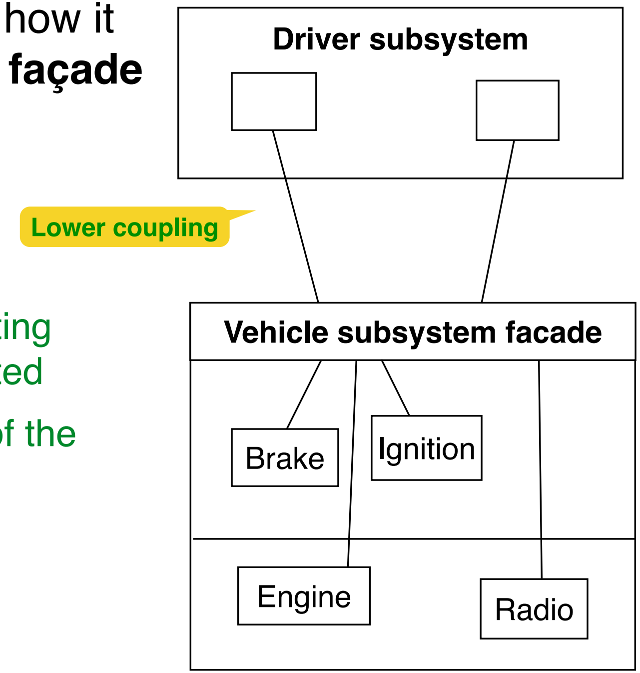
- Provides a **unified interface** for a subsystem
  - Consists of a set of public operations
  - Each public operation is delegated to one or more operations in the classes behind the façade
- Defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts away the gory details)
- Allows to **hide** design spaghetti from the caller

More details on **design patterns** in **Lecture 07** on **Object design**



Opaque architecture with a façade

- The **vehicle subsystem** decides exactly how it is accessed with the **vehicle subsystem façade**
- **Advantages**
  - Reduced complexity
  - Fewer recompilations
  - A façade can be used during integration testing when the internal classes are not implemented
  - Possibility of writing mock objects for each of the public methods in the façade



More details in **Lecture 08** on **Testing**

L04E02 Facade Pattern

Not started yet.

Start exercise

Easy

Due Date: End of today (AoE)

10 min

5 pts

- **Problem statement**
  - Reduce the coupling between the two subsystems **store** and **ecommerce**
  - Implement the façade pattern to provide a unique interface for the **ecommerce** subsystem

Design principle:
low coupling

Outline

- Overview of system design
- Design goals
- Hints for system design
- Subsystem decomposition
- Façade pattern
- **Architectural styles**
  - **Layered architecture** 连接层
  - Client server architecture
  - REST architectural style
- UML component diagrams

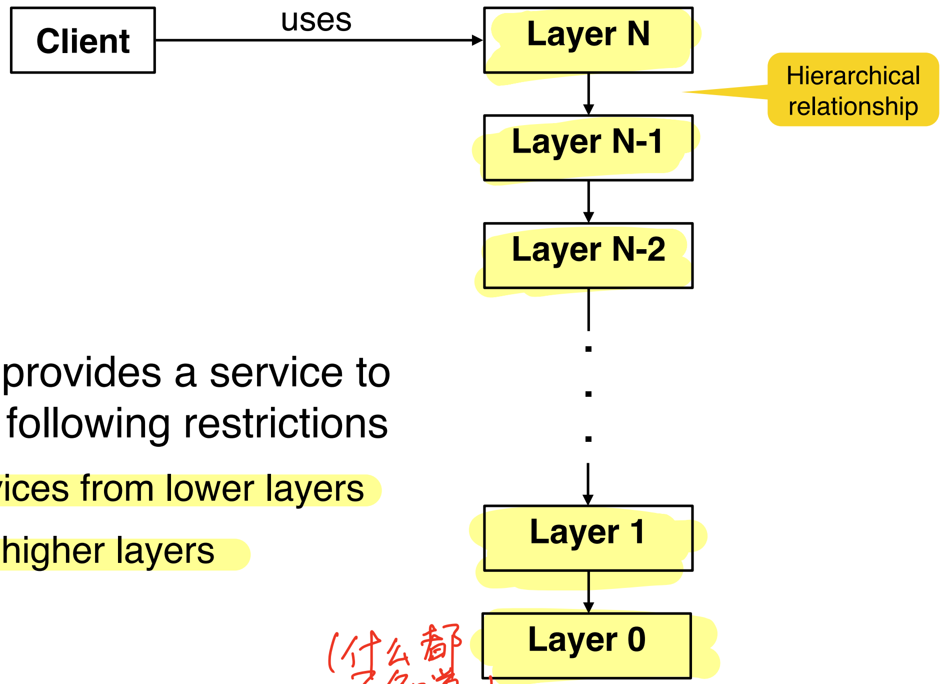
Architectural style vs. architecture

- **Subsystem decomposition:** identification of subsystems, services, and their relationships to each other

- **Architectural style:** a pattern for a subsystem decomposition

- **Software architecture:** instance of an architectural style

Layered architectural style

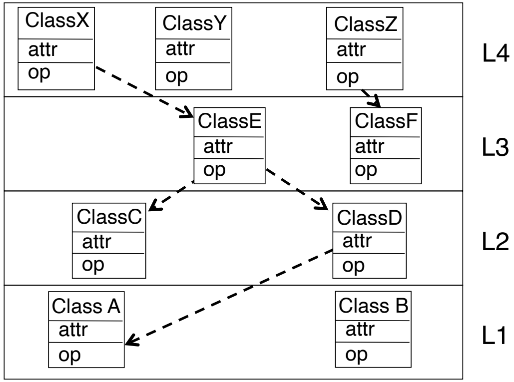


(什么都不知道)

- A **layer** is a subsystem that provides a service to another subsystem with the following restrictions
  - A layer only depends on services from lower layers
  - A layer has no knowledge of higher layers

Closed architecture (opaque layering)

A **layered architecture is closed**, if each layer can only call operations from the layer directly below (also called "direct addressing")

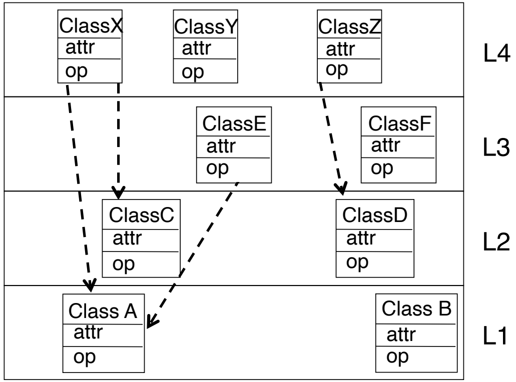


Design goals: maintainability, flexibility, portability

more portable ➡ **low coupling** 😄, but potentially a **bottleneck**

Open architecture (transparent layering)

A **layered architecture is open** if a layer can call operations from any layer below (also called “indirect addressing”)



Design goals: high performance, real-time operations support

more **efficient** ➡ **high coupling** 😞

3 layered architectural style

- Often used for the development of web applications
- **Example**

1) The **web browser** implements the user interface
2) The **web server** serves requests from the web browser
3) The **database** manages and provides access to the persistent data

chrome server

database

Layer vs. tier

- 3 layered architectural style: an **architectural style** where an application consists of 3 hierarchically ordered layers
- 3 tier architecture: a **software architecture** where the 3 layers are allocated on 3 separate hardware nodes
- **Note:** Layer is a type (e.g. class, subsystem) and tier is an instance (e.g. object, hardware node)
- In practice, the terms layer and tier are often used interchangeably (when blurring the distinction between type and instance is admissible)

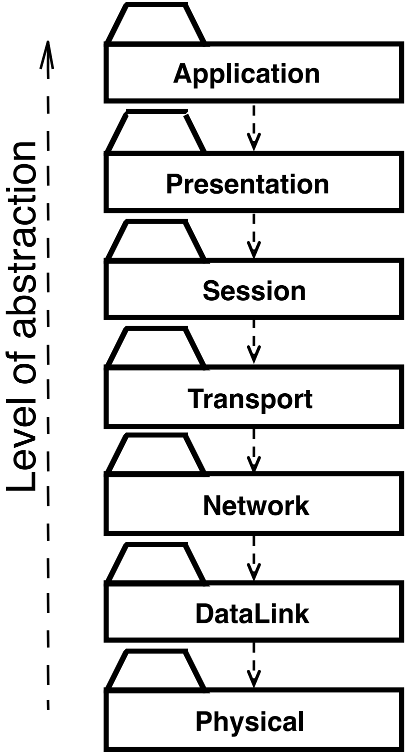
4 layered architectural style

- Hierarchically ordered layers
- **Example**
  1) A **web browser** provides the user interface
  2) A **web server** serves static HTML requests
  3) An **application server** provides session management (for example the contents of an electronic shopping cart) and processes dynamic HTML requests
  4) A **database** manages and provides access to the persistent data
     - Usually a relational database management system (RDBMS)

➡ If these layers reside on different hardware nodes, then it is a 4 tier architecture

7 layered architectural style

- ISO's OSI Reference Model
  - ISO = International Standard Organization
  - OSI = Open System Interconnection
- The reference model defines 7 layers and communication protocols between the layers



Outline

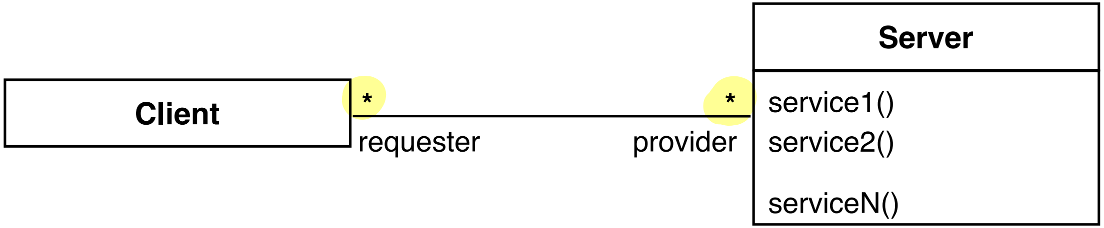
- Overview of system design
- Design goals
- Hints for system design
- Subsystem decomposition
- Façade pattern
- Architectural styles
  - Layered architecture
  - **Client server architecture**
  - REST architectural style
- UML component diagrams

Client server architecture 也是一种layer

- Often used in the design of database systems
  - **Client**: user application
  - **Server**: database access and manipulation (cloud)
  - **Client** requests a service from the **server**
- Functions performed by the client 有些locally cached
  - Input by the user (customized user interface)
  - Sanity checks of input data
- Functions performed by the server
  - Centralized data management
  - Provision of data integrity and database consistency
  - Provision of database security

Client server architectural style

- One or more servers provide services to clients
- Each client calls a service offered by the server
  - **Server** performs service and returns result to client
  - **Client** knows interface of the server
  - **Server** does not know the interface of the client
- Response is typically immediate (i.e. less than a few seconds)
- End users interact only with the client



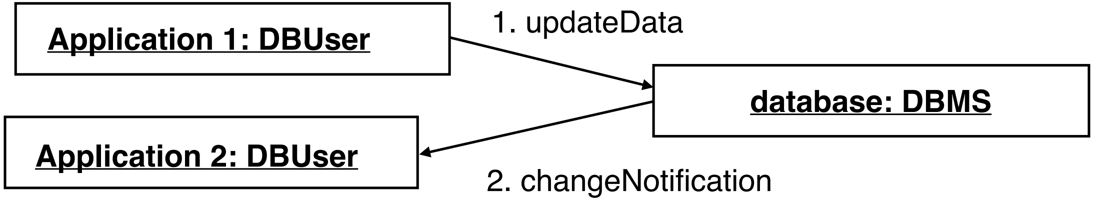
Design goals for client server architectures

Portability	Server runs on many operating systems and many networking environments
High performance	Client optimized for interactive display-intensive tasks Server optimized for CPU-intensive operations
Scalability	The server can handle large amounts of clients
△ 规模化	
Flexibility	The user interface of the client supports a variety of end-devices (phone, laptop, smart watch)
	QQ：任何平台
Reliability	Server should be able to handle client and communication problems

Problems with the client server architectural style

- Client server systems use a request-response protocol
- Peer to peer communication is often needed
- **Example:** a database must process queries from application 1 and should be able to send notifications to application 2 when data in the database has changed

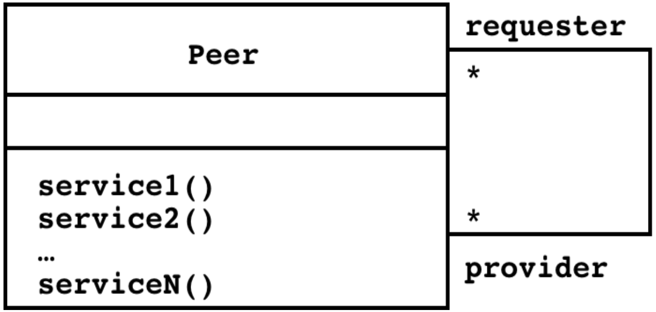
Spotify：点对点传输
（迅雷）



Peer to peer architectural style

P2P

- Generalization of the client server architectural style
  - Clients can be servers and servers can be clients
- Introduction of a new abstraction: **Peer**



Outline

- Overview of system design
- Design goals
- Hints for system design
- Subsystem decomposition
- Façade pattern
- Architectural styles
  - Layered architecture
  - Client server architecture
  - **REST architectural style**
- UML component diagrams

REST as an architectural style

- Context: large number of distributed clients access shared resources and services
- Problem: how can we manage a set of shared web resources and services, and still make them **highly modifiable**, **reusable**, **scalable** and **available**?
- **Solution: REST** provides an abstraction that models the structure and behavior of the world wide web
  - REpresentational State Transfer (REST): **combination of client server and layered architecture style** (originally called the “HTTP object model”)
  - Defines a set of constraints on distributed architectures to simplify the exchange of resources between distributed components

Elements of a REST architecture

- Web resource: any entity that can be accessed online
  - Identified by their URI (Uniform Resource Identifier)
  - The most common form of URI is the URL used to identify documents or files
- RESTful web service: offers access to textual representations of web resources with a predefined set of operations

Stateful
Stateless

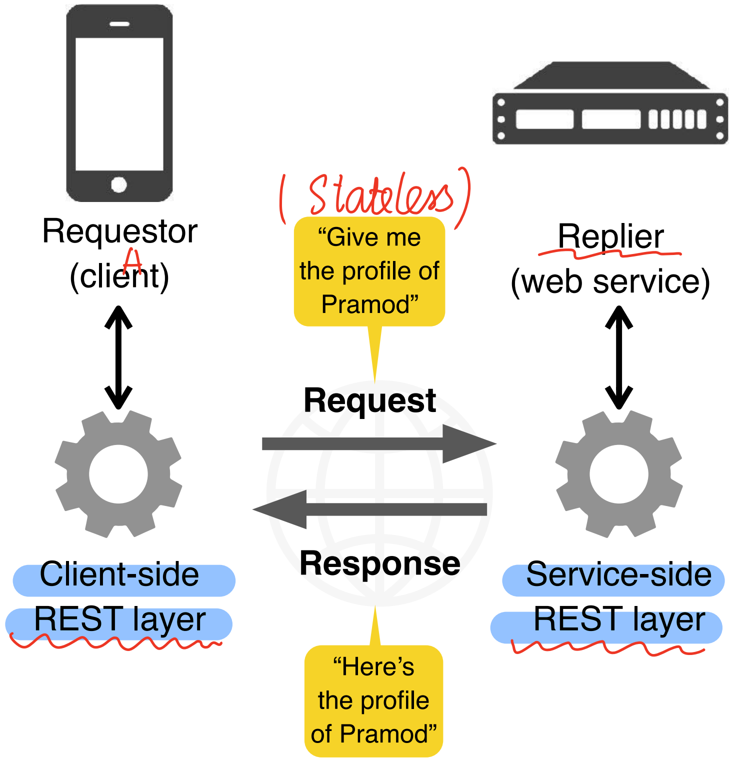
- Client: component that invokes methods on a RESTful web service
- Connector: **stateless** request-response protocol with a message encoded in XML / JSON

Stateless: no additional state information between two requests

Stateless request-response protocol

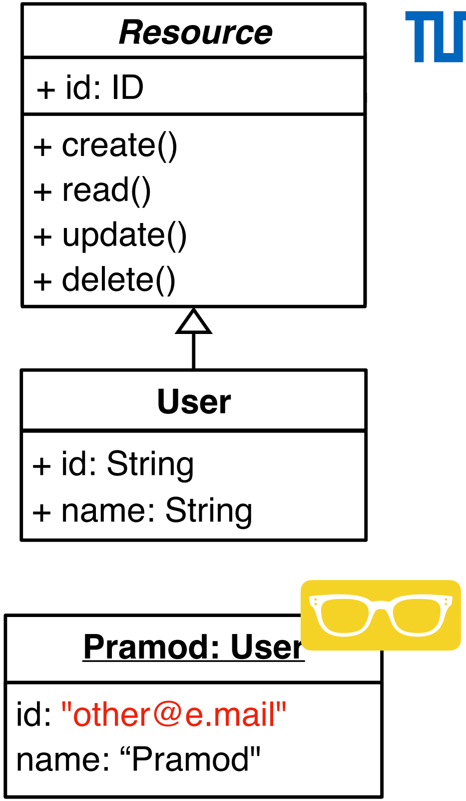
1. **Requestor** sends a message to **replier**
2. **Replier** receives and processes the **request**
3. **Replier** returns a message in **response**

- Stateless: the replier does not keep a history of old requests
- Request and replier use multiple layers to handle requests and responses



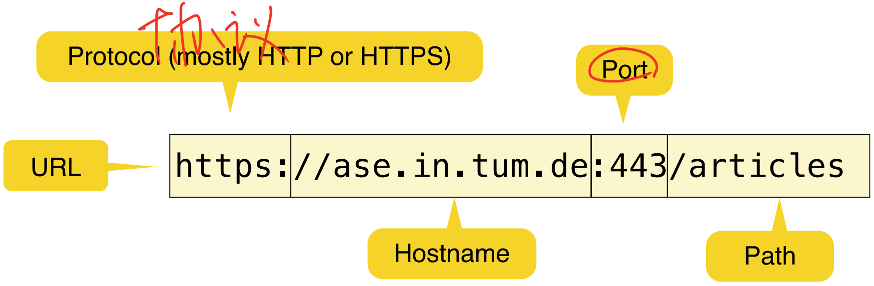
Performing methods on resources

- REST uses the four basic methods that can be performed on any resource
- Example resource Pramod:
  `{ id = "an@e.mail", name = “Pramod" }`
  - Create (POST) — Create a new user named Pramod
  - Read (GET) — Give me all information about Pramod
  - Update (PUT) — Update the email of Pramod
  - Delete (DELETE) — Delete the user named Pramod
- These methods are often summarized as **CRUD**



Identifying resources

- URI = Uniform Resource Identifier
  统一
  - A text string used to identify a resource (e.g.: smart objects, …)
- Most often: a Uniform Resource Locator (URL)

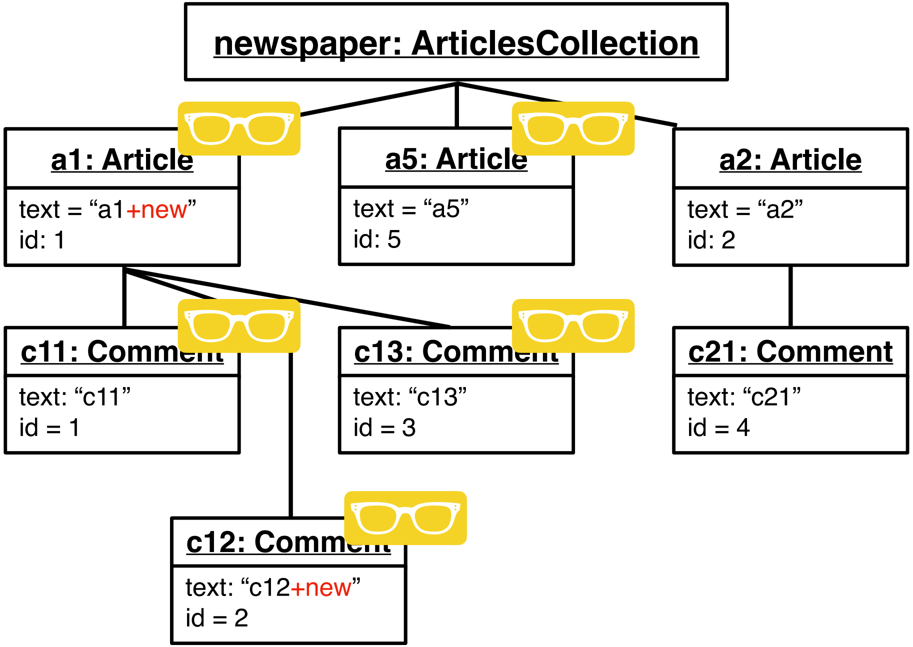


Example: mapping web requests to CRUD operations

Client web request

HTTP Method	URI Path Component
GET	/articles
POST	/articles
PUT	/articles/1
DELETE	/articles/5
GET	/articles/1/comments
POST	/articles/2/comments
PUT	/articles/1/comments/2
DELETE	/articles/1/comments/3

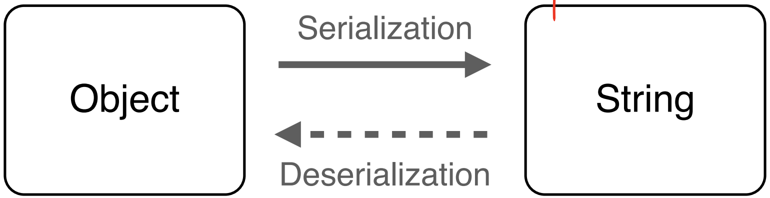
Web service state



Serialization

- The message body in requests and responses is arbitrary data
- To send objects over the network, they must be mapped to data
  ➔ serialized
- On the receiving side, the data needs to be mapped back to objects
  ➔ deserialized
- There are several ways of representing objects as data, e.g. using UTF-8 strings and JSON as serialization/deserialization technique

protocol Map



JavaScript object notation (JSON)

- Open-standard format
- Uses human-readable text to transmit objects consisting of **key value pairs**
- Language-independent data format

Movie
- imdbID: String - title: String - simplePlot: String - genres: [String] - year: Int - rating: String

Object

<u>theGodfather: Movie</u>
imdbID = "tt0068646" title = "The Godfather" simplePlot = "The …" genres = [Crime, Drama] year = 1972 rating = "9.2"

JSON

key value

```
{
    "imdbID" : "tt0068646",
    "title" : "The Godfather",
    "simplePlot" : "The aging ...",
    "genres" : [
        "Crime",
        "Drama"
    ],
    "year" : 1972,
    "rating" : "9.2",
}
```

HTTP status codes

- Purpose: classify responses

No error occurred	An error occurred
2xx Success 200 OK 201 Created (POST) 202 Accepted **3xx Redirection**	**4xx Client Error** 400 Bad Request 401 Unauthorized 403 Forbidden 404 Not Found 405 Method Not Allowed **5xx Web Service Error** 500 Internal Web Service Error 501 Not Implemented

Example: Spring Boot

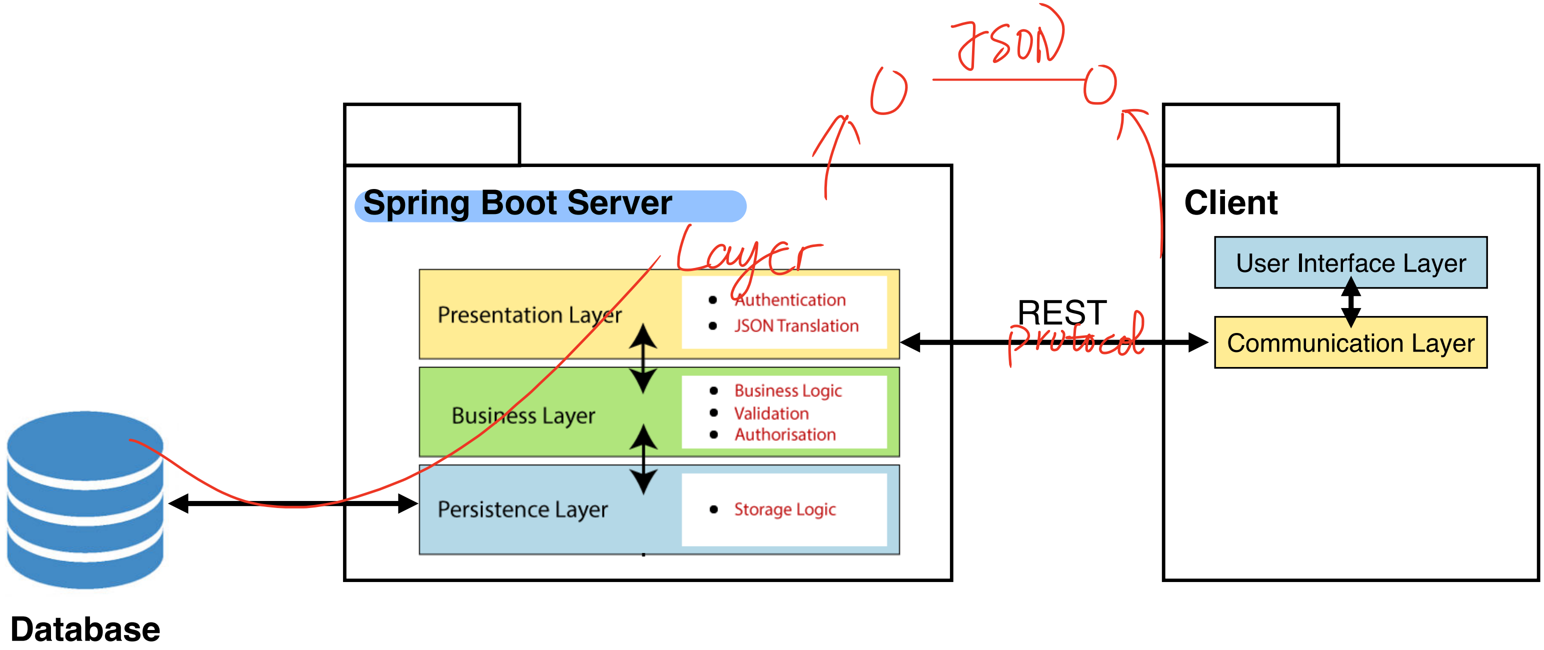
- Opinionated, easy to get-started addition to the **Spring** platform
- Highly useful for creating stand-alone, production-grade applications with minimum effort
- The **Spring** framework provides comprehensive infrastructure support for developing Java applications using features like dependency injection, and out of the box modules like
  - **Spring Data**: provide a familiar and consistent programming model for data access (e.g. database)
  - **Spring MVC**: framework to build web applications based on REST
  - **Spring Security**: powerful and highly customizable authentication and access-control framework
  - **Spring AOP**: programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns
  - **Spring ORM**: object relational mapping provides a high-level data access
  - **Spring Test**: provides several abstract support classes that simplify the writing of integration tests

  → These modules reduce the development time of an application

More about **M**odel **V**iew **C**ontroller (MVC) in **Lecture 05** on **System Design**

https://spring.io/guides/gs/rest-service https://spring.io/projects/spring-boot

Example: Spring Boot architecture

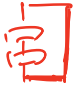


Outline

- Overview of system design
- Design goals
- Hints for system design
- Subsystem decomposition
- Façade pattern
- Architectural styles
  - Layered architecture
  - Client server architecture
  - REST architectural style
- **UML component diagrams**

UML component

- **Building block** of the system (also called subsystem)



- Represented as a rectangle with a tabbed rectangle symbol inside

- Components have different lifetimes
  - Some exist only at **design time**: classes, associations
  - Others exist until **compilation time**: source code, pointers
  - Some exist at link time or only at **runtime**: linkable libraries, executables, addresses

UML component diagram

- Model the top level view of the system design in terms of **components** and **dependencies**
  - Components can be source code, linkable libraries, executables
  - Dependencies are the **connectors** between components
  - The types of dependencies are implementation language specific
- Informally also called “software wiring diagram”
  - They show how the components are wired together in the overall application
- UML component diagrams use **UML interfaces**

UML component diagram

- A UML interface describes a group of operations provided or required by a component
- There are two types of interfaces: provided and required interfaces

A **provided interface** is modeled using the lollipop notation

Interfaces describe a more formal relationship

A **required interface** is modeled using the socket notation

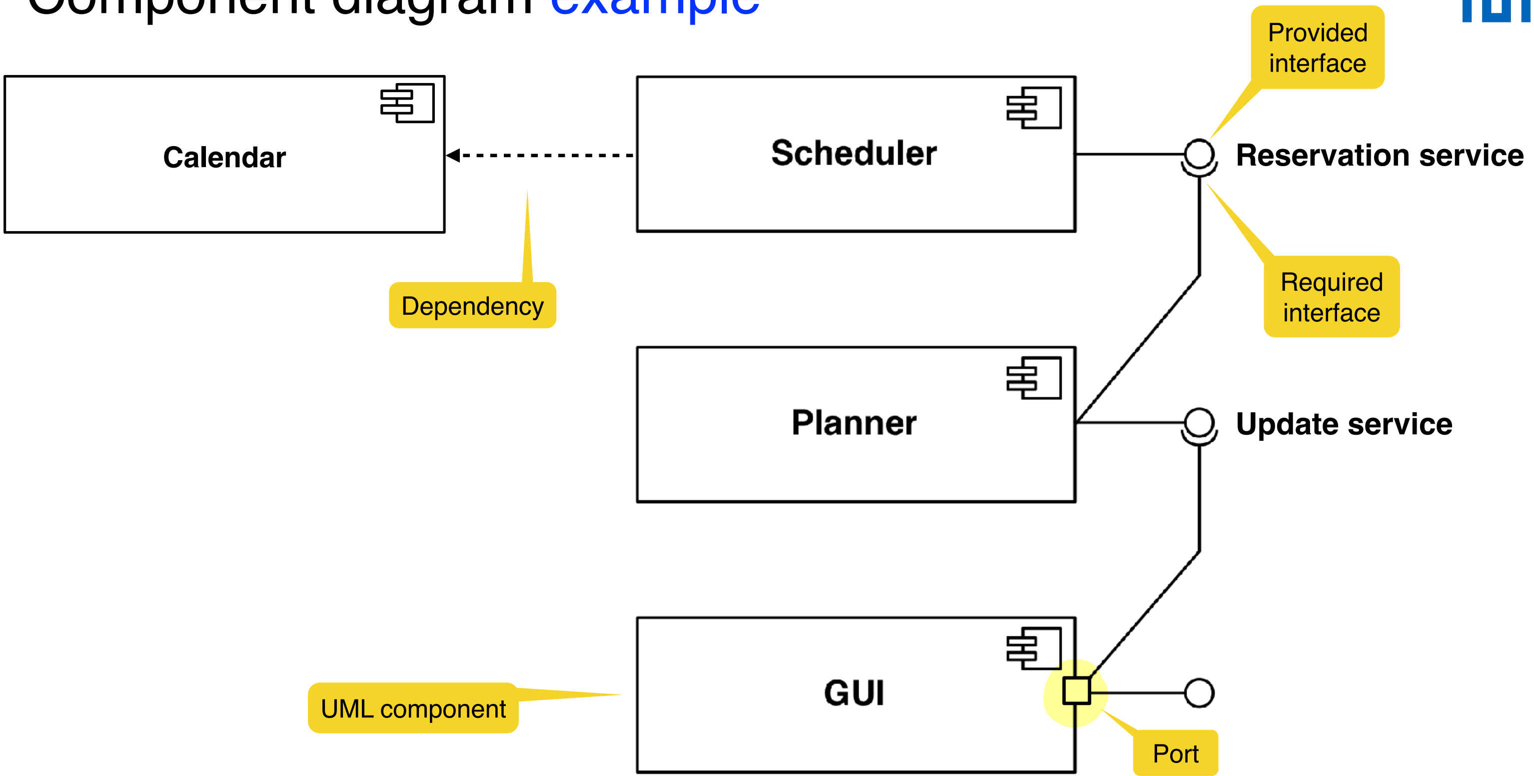
Dependency: one component depends on the implementation of another component

A **port** specifies a distinct interaction point between the component and its environment

- Depicted as small squares on the sides of classifiers
- Allow to group interfaces

Dependencies describe a more informal relationship

Component diagram example



L04E03 Model a Chat System

Not started yet.

Start exercise

Medium

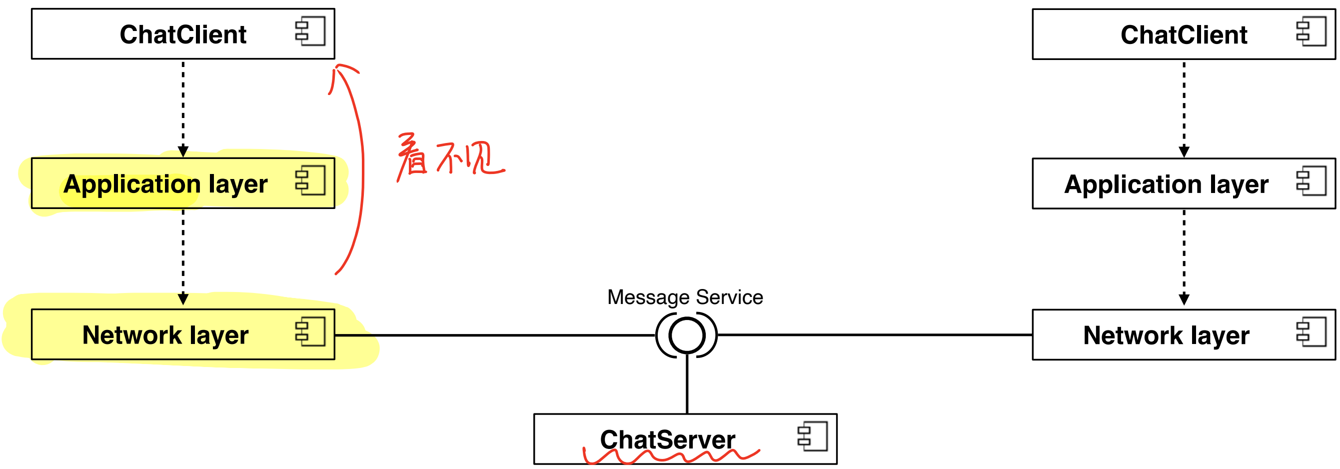
Due Date: End of today (AoE)

5 min

5 pts

- **Problem statement**
  - Model the architecture of a chat system with a layered architectural style
  - 2 computers are using a **ChatClient** with an **application layer** and a **network layer**
  - The network layer communicates with a **ChatServer**

Example solution



L04E03: Model a chat system

- **Change:** the messages should be end-to-end encrypted
- Encryption should be implemented in a new **presentation layer** between the **application layer** and the **network layer**

Homework

- **H04E01** Analysis Models & System Design (text exercise)
- **H04E02** Design Goal Trade-offs (text exercise)
- **H04E03** Subsystem decomposition of an exam management system (modeling exercise)
- Read more about **REST** and **Spring Boot** (see Literature)

→ Due until 1h before the **next lecture**

Summary

- **System design** reduces the gap between a problem and an existing machine
- **Design goals** describe important system qualities
  - Design trade-offs can be used to evaluate alternative designs
- **Subsystem decomposition**
  - Partitioning a system into manageable parts with **low** coupling and **high** cohesion
- Distinction between **architectural style** and **architecture**
  - Layered architecture
  - Client server
  - REST
- **UML component diagrams** model the software architecture

Literature

- E.W. Dijkstra (1968): The structure of the T.H.E Multiprogramming system, Communications of the ACM, 18(8), pp. 453-457
- D. Parnas (1972): On the criteria to be used in decomposing systems into modules, CACM, 15(12), pp. 1053-1058
- J.D. Day and H. Zimmermann (1983): The OSI Reference Model, Proc. IEEE, Vol.71, 1334-1340
- Jostein Gaarder (1991): Sophie's World: A Novel about the History of Philosophy
- Frank Buschmann et al. (1996): **Pattern-Oriented Software Architecture**, Vol 1: A System of Patterns, Wiley
- Roy Thomas Fielding: Architectural Styles and the Design of Network-based Software Architectures: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- REST: https://www.codecademy.com/article/what-is-rest
- Spring Boot: https://spring.io/projects/spring-boot
- Building a RESTful Web Service: https://spring.io/guides/gs/rest-service