# L03 System Design I

## Modularity and Data Management

Prof. Pramod Bhatotia
Systems Research Group
https://dse.in.tum.de/

# Today's learning goals

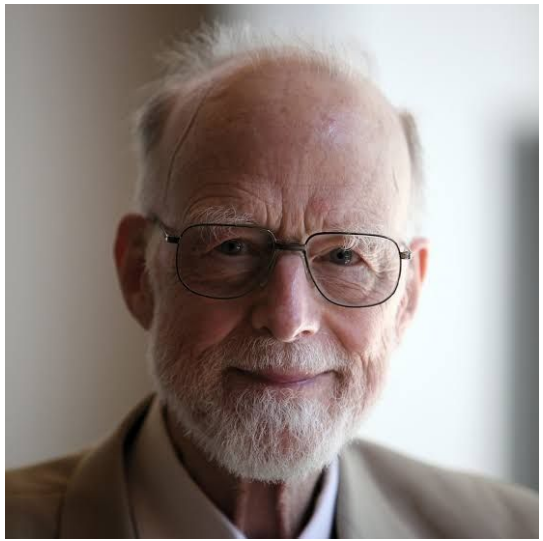
- **Part I:** System design challenges
- **Part II:** Modularity
  - Subsystem decomposition: Modules
  - Differentiate between coupling and cohesion
  - Design pattern: Facade pattern
  - Interface design
- **Part III:** Software architecture: Layered architectures
  - Open vs closed layered architectures
  - Different layers, different abstraction
  - Pulldown the complexity downward/upward
  - Ubiquitous adoption of layered architectures in systems
- **Part IV:** Data management
  - Key Value (KV) store, Filesystems, Shared log, Databases
- **Part V:** Pattern implementation (MVC pattern)

# Outline

- **Part I: System design challenges**
  - Software complexity and the quest for simplicity
  - System design goals
  - System design trade-offs
  - Hints for computer system designs
- **Part II:** Modularity
- **Part III:** Software architecture: Layered architectures
- **Part IV:** Data management
- **Part V:** Pattern implementation

# Design is difficult

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



Sir Tony Hoare
Turing Award Winner, 1980

# Why design is so difficult?

- **Requirements engineering:** focuses on the **application domain**
- **System 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
  - The mapping of hardware-software in the cloud makes it even more challenging
- **Design window:** time in which design decisions have to be made
  - Often short and limited budget

# The quest for simplicity

- Engineering always starts with a **“good system design”**
  - The ability to recognize and avoid complexity is a crucial design skill
  - **Always aim for a simpler system design**

- Complexity is anything related to the structure of a software system that makes it hard to **understand** and **modify** the system

“The Sciences of the Artificial”, Herb Simons, 1969
https://en.m.wikipedia.org/wiki/The_Sciences_of_the_Artificial

Explained "the principles of modeling complex systems, particularly the human information-processing system that we call the mind."

# Symptoms of complexity

- **Change amplification**
  - A seemingly simple change requires code modifications in many different places
- **Cognitive load**
  - Refers to how much a developer needs to know in order to complete a task
- **Unknown unknowns**
  - It is not obvious which pieces of code must be modified to complete a task

# Causes of complexity

1. **Dependencies**
   - A dependency exists when a given piece of code cannot be understood and modified in isolation
   - Dependencies are fundamental in software systems, and can't be completely eliminated – **Aim to minimize them!**

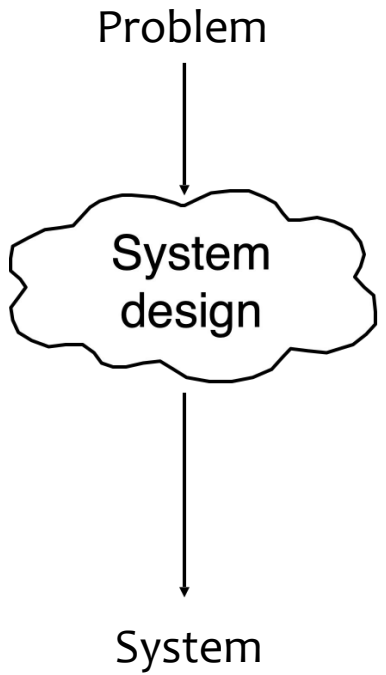
2. **Obscurity**
   - Obscurity occurs **when important information is not obvious**
   - Inadequate documentation, and complex assumptions

# The scope of system design

- Bridge the gap between a problem and a system in a manageable way
- The system design should address both
  - **Functional requirements**
  - **Non-functional requirements**

Collectively also known as **FURPS requirements**

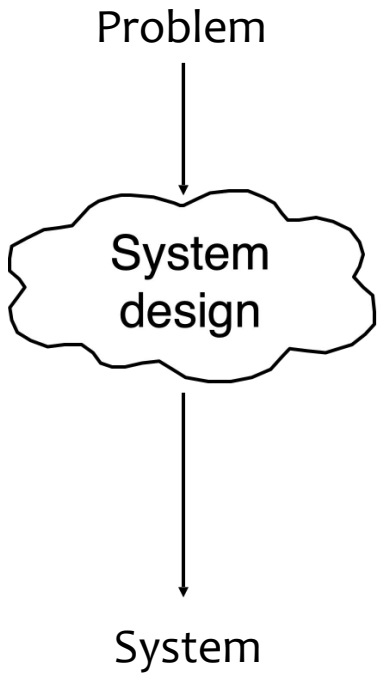
**F**unctionality
**U**sability
**R**eliability
**P**erformance
**S**upportability




FURPS is a broader taxonomy of functional and non-functional requirements:
https://en.wikipedia.org/wiki/FURPS
We have covered important non-functional requirements for the cloud systems in L03

# How do we approach system design?

- Understand the **functional requirements first**
- Next step:
  - Identify **non-functional requirements (or design goals)**



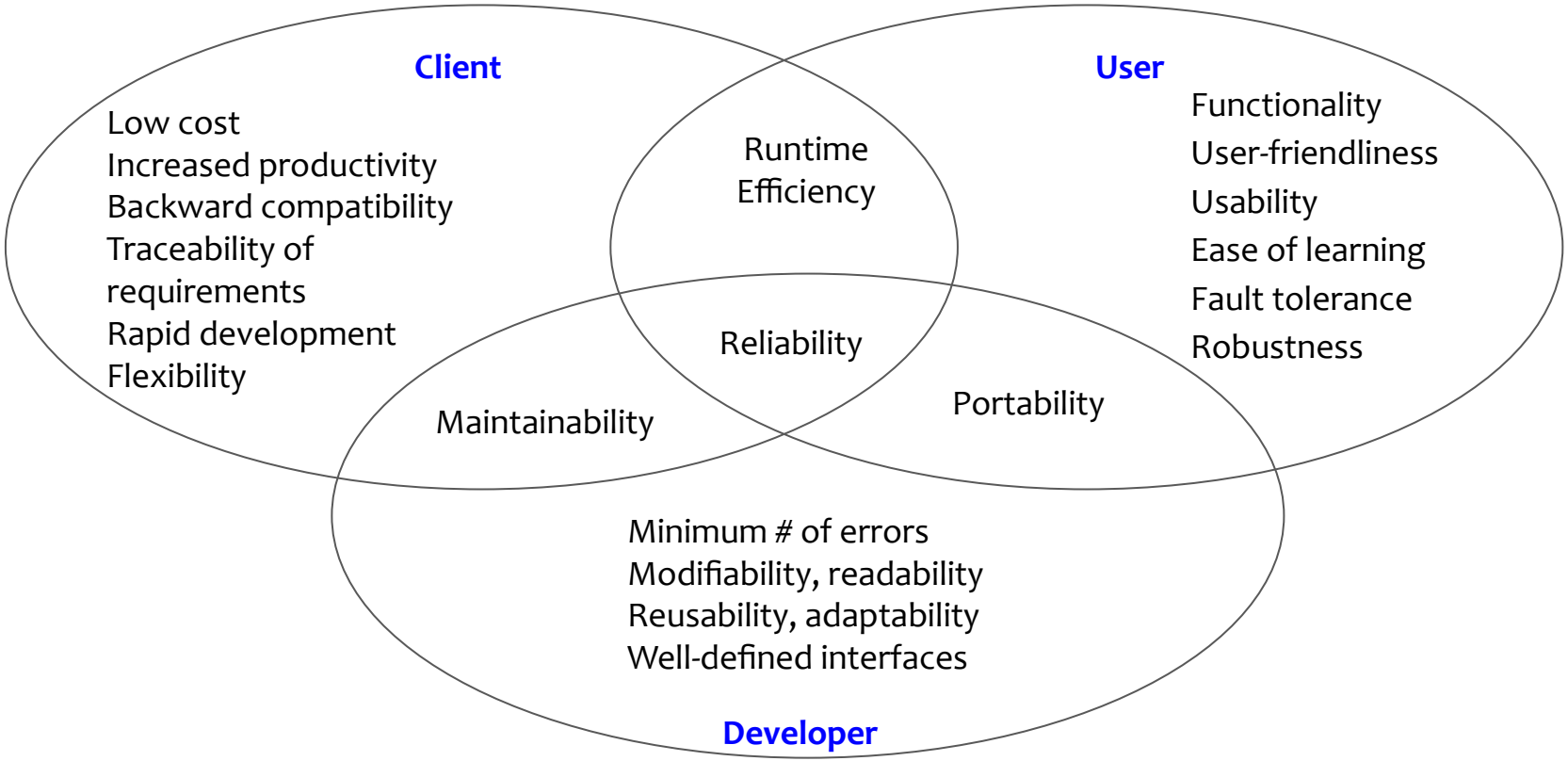
# Design goals

- **Design goals** govern the system design activities

- As a start, **any non-functional requirement** is a design goal
  - See **non-functional requirements in the cloud from L03**:
    - Scalability, performance, security, reliability, etc.

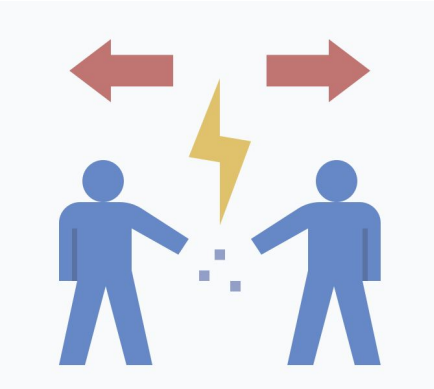
- **Design goals *often* conflict** with each other
  - Typical **design goal trade-offs**
  - Strike a balance between **the major design** goals

# Different types of design goals (Example)



# Typical design goal trade-offs (examples)

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



# Hints for system design

| Why? | *Functionality* Does it work? | *Speed* Is it fast enough? | *Fault-tolerance* Does it keep working? |
|---|---|---|---|
| **Where?** | | | |
| *Completeness* | Separate normal and worst case | Shed load<br>End-to-end<br>Safety first | <br>End-to-end<br> |
| *Interface* | Do one thing well:<br>Don't generalize<br>Get it right<br>Don't hide power<br>Use procedure arguments<br>Leave it to the client<br>Keep basic interfaces stable<br>Keep a place to stand | Make it fast<br>Split resources<br>Static analysis<br>Dynamic translation | End-to-end<br>Log updates<br>Make actions atomic |
| *Implementation* | Plan to throw one away<br>Keep secrets<br>Use a good idea again<br>Divide and conquer | Cache answers<br>Use hints<br>Use brute force<br>Compute in background<br>Batch processing | Make actions atomic<br>Use hints |

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

# A three-part series: System design in our course

**L03: Design I**

- **Modularity**
  - How to design modular systems?
- **Data management**
  - How to manage your data?

**L04: Design II**

- **Performance**
  - How to design performant systems?
- **Concurrency (Scale-up)**
  - How to scale-up systems?
- **Scalability (Scale-out)**
  - How to scale-out systems?

**L05: Design III**

- **Security**
  - How to secure your systems?
- **Fault tolerance**
  - How to make systems **reliable** & **available**?

**System implementation**

# Outline

- ~~**Part I:** System design challenges~~

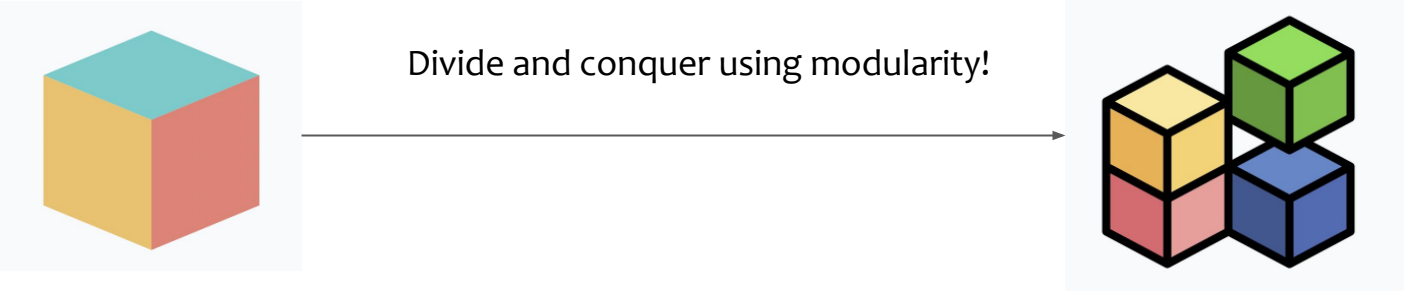
- **Part II:** **Modularity**
  - **Subsystem decomposition: Modules**
    - Create an initial subsystem decomposition
    - Differentiate between coupling and cohesion
  - **Pattern implementation:**
    - Facade
  - **Interface design**
    - Shallow vs deep modules: The trade-offs between interface and functionality
    - General purpose modules are deeper
    - Information hiding (and leakage) principle
- **Part III:** Software architecture: Layered architectures

- **Part IV:** Data management

- **Part V:** Pattern implementation

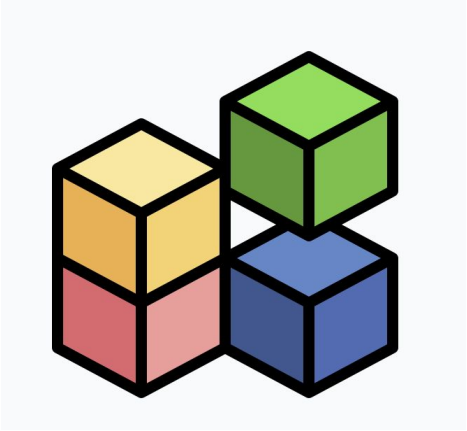
# How to approach system design?

- Given a complex “problem statement”, how do we design systems?
  - **Use divide & conquer**
    - Model the **new system design as a set of subsystems**
    - Address **the major design goals first**



# Modularity

- In a modular design, a software system is decomposed into a set of **modules/subsystems**
  - Modularity can take many forms, such as classes, methods, subsystems, or services
  - In other words, **modularity can applied at different abstraction levels**



- A **modular design helps in managing software complexity**
  - Microservices is an example of a modular design, where we divide a large monolithic service into a set of microservices

# Challenges of a modular design

- **Ideally,** each module would be completely independent of the others
  - A developer could work on a module without knowing anything about other modules

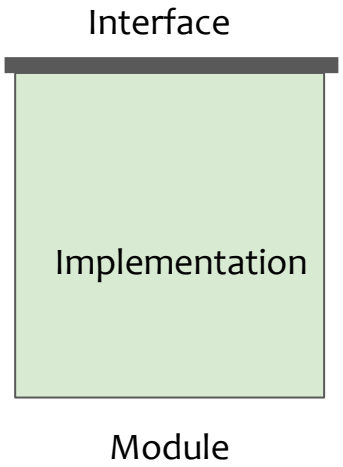
- **In practice,** modules are dependent
  - they rely on each other to provide some functionality or service
  - A root cause of complexity

A good system design aims to **minimize the dependencies between modules**

# Terminology: Interface and implementation

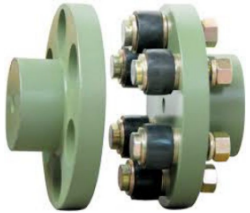
Each module provides two parts:

- An **interface**
  - The **interface** consists of everything that a developer working in a different modules must know to use the given module,
  - Interface describes **"what" the module does**, but **"not how" it does it!**
  - **Think of interface as a module's promise!**
- An **implementation**
  - The **implementation** consists of the code that carries out the promises made by the interface
  - A developer working on a module must understand both the interface and implementation of the module, plus the interfaces of any other dependent modules



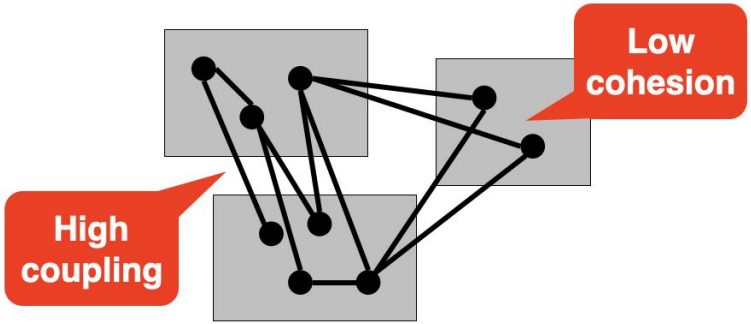
# Coupling and cohesion of modules

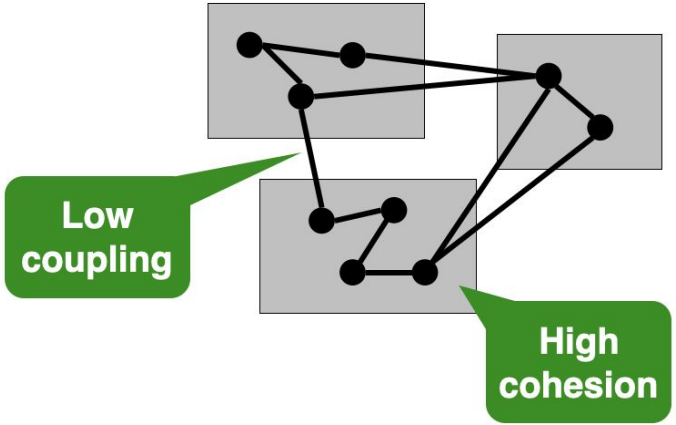
- **Our goal:** Reduce system complexity while allowing dependencies
- **Cohesion** measures dependency between classes within one module
  - **High cohesion:** the classes in the module perform similar tasks and are related to each other via many associations
  - **Low cohesion:** lots of miscellaneous and auxiliary classes, almost no associations
- **Coupling** measures dependency between multiple modules
  - **High coupling:** changes in one module will have a high impact on the other modules
  - **Low coupling:** a change in one module will have minimal impact on other dependent modules

# A good system design

- A good system design aims to achieve **high cohesion** and **low coupling**
  - High cohesion strives for tightly dependent objects in one module
  - Low coupling strives to minimize interdependence of objects between different modules



# How to achieve high cohesion and low coupling?

- **High cohesion** can be achieved if most of the interaction is within modules, rather than across module boundaries
- **Low coupling** can be achieved if a calling module does not need to know anything about the internals of the called module

# Achieving high cohesion and low coupling

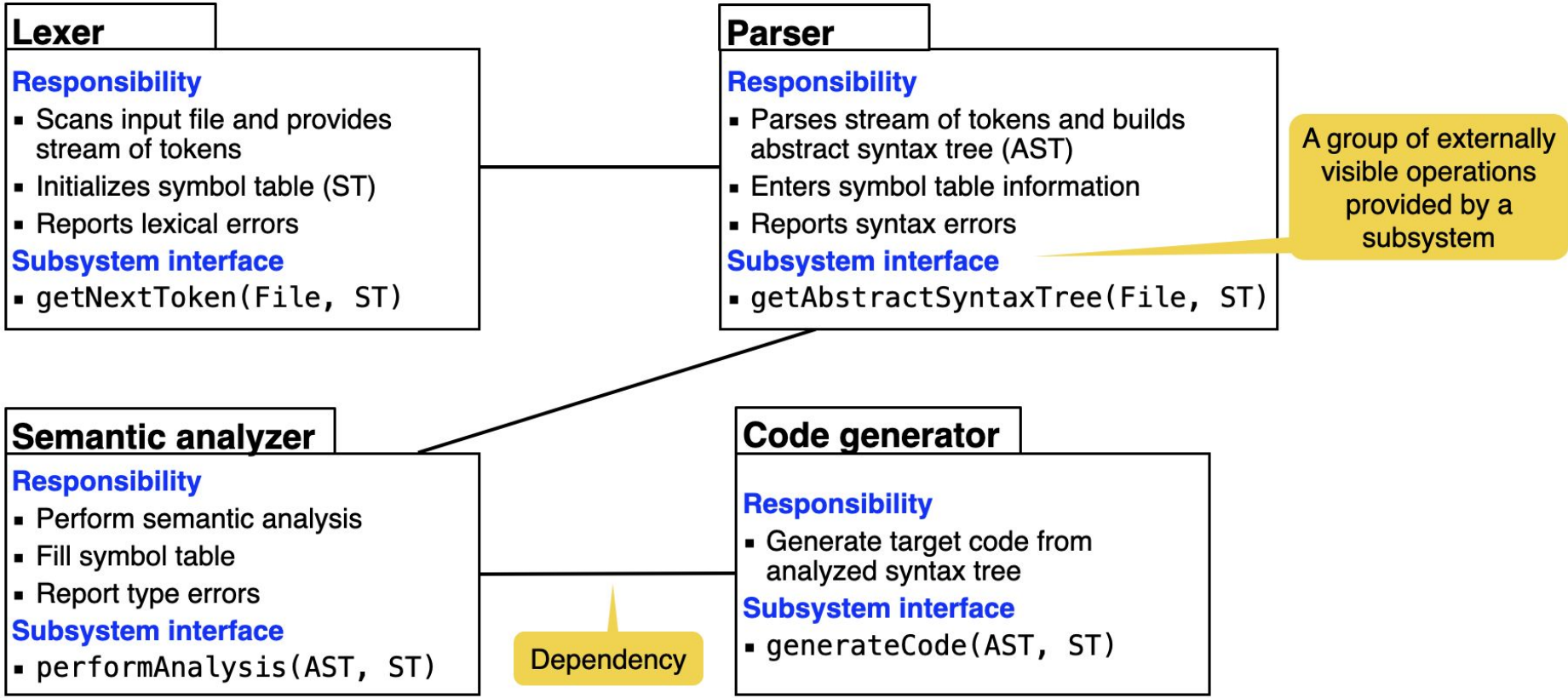
## High cohesion

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

## Low coupling

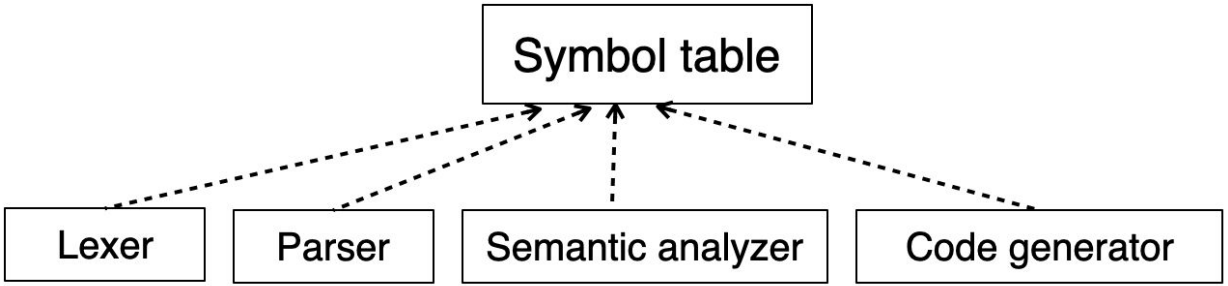
- Small interfaces
- Information hiding principle
- No/minimal global data
- Interactions are mostly within the module rather than across module boundaries

# Example: Subsystem decomposition of a compiler

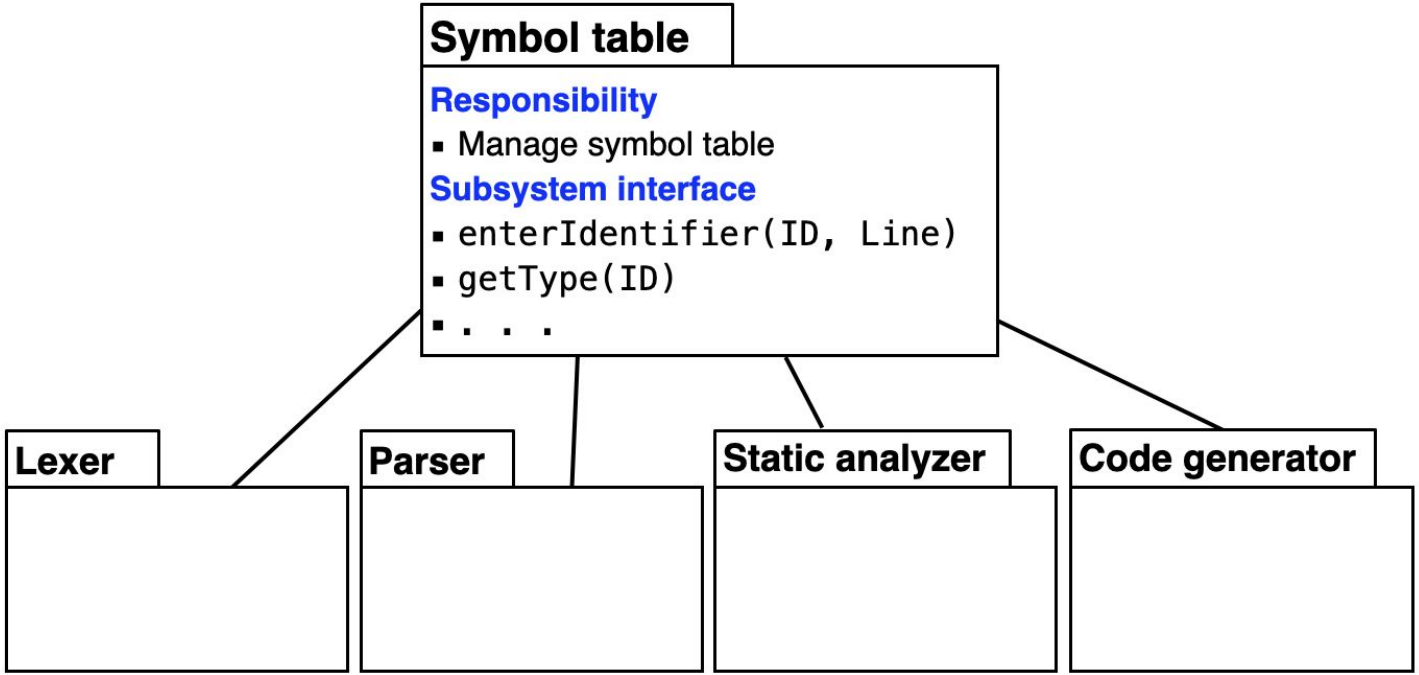


# Coupling and cohesion in the compiler example

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



# Compiler design w/ better cohesion and coupling



# Outline

- ~~**Part I:** System design challenges~~

- **Part II: Modularity**
  - ~~Subsystem decomposition: Modules~~
  - **Design pattern:**
    - Facade pattern
  - **Interface design**
    - Shallow vs deep modules: The trade-offs between interface and functionality
    - General purpose modules are deeper
    - Information hiding (and leakage) principle

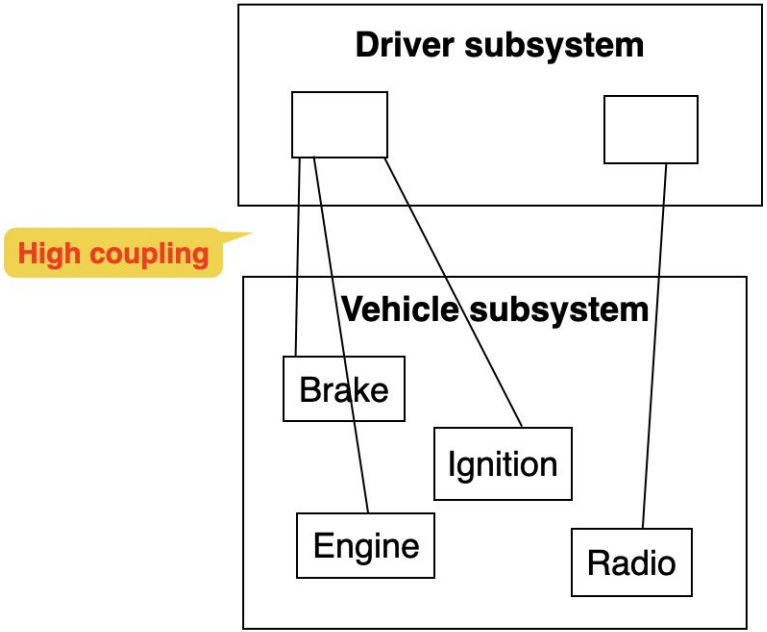
- **Part III:** Software architecture: Layered architectures

- **Part IV:** Data management

- **Part V:** Pattern implementation

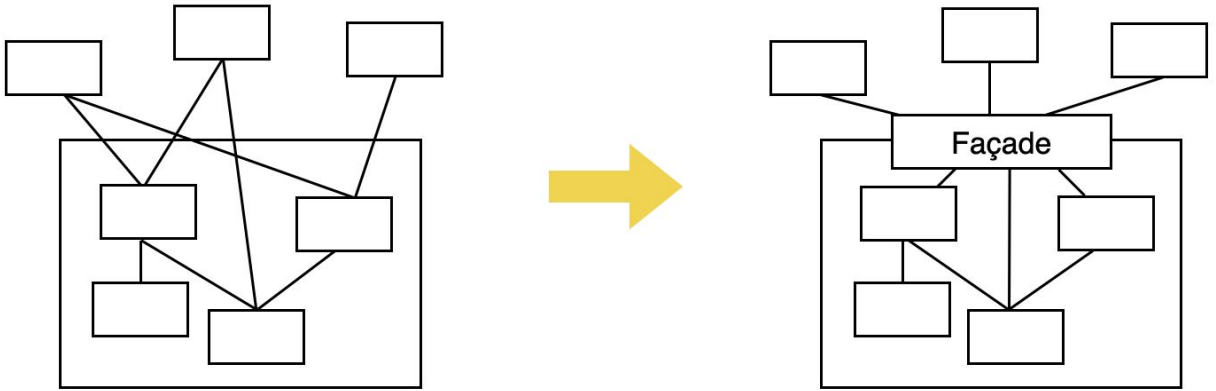
# Facade pattern

- Another example: **Driving system**
  - The driver modules can call any class operation in the vehicle subsystem
- **Problem:**
  - High coupling between modules
  - Lack of proper interface
- **Solution:**
  - **Facade design pattern**



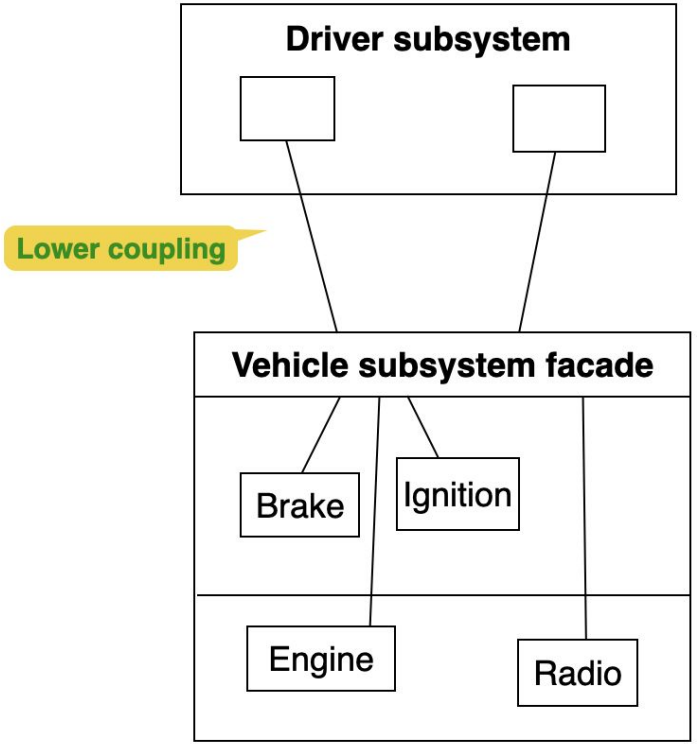
# Facade design pattern: Reduces coupling

- Provides **a unified interface for a module/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



# Opaque architecture with a facade

- The vehicle subsystem decides exactly how it is accessed with the vehicle **subsystem façade**
- **Advantages**
  - **Reduced complexity**
  - 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



# Outline

- ~~**Part I:** System design challenges~~

- **Part II:** **Modularity**
  - ~~Subsystem decomposition: Modules~~
  - ~~Design pattern:~~
  - **Interface design**
    - Shallow vs deep modules: The trade-offs between interface and functionality
    - General purpose modules are deeper
    - Information hiding (and leakage) principle

- **Part III:** Software architecture: Layered architectures

- **Part IV:** Data management

# Interface design

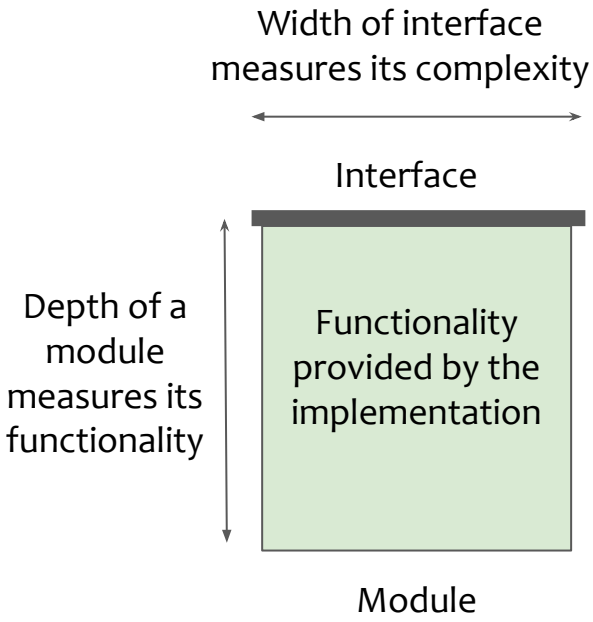
- **An interface** is a boundary that separates different subsystems and defines how they interact with each other
  - An Interface **abstracts out the implementation details**
  - For e.g., a subsystem/module may expose an Application Programming Interface (API)
- **Interface parts:** An interface usually have two parts
  - **Formal parts:** Explicitly parts specified in the code
    - Specified as the API/method signature
    - For e.g., names/types of arguments, return types, exceptions
  - **Informal parts:** Not explicitly specified in the code, but captures the high-level behavior
    - Usually, specified in the comments
    - For e.g., they may assume certain side-effect (a file would be deleted after invoking this interface), or assumes certain constraints to be met before invoking the interface
- **A developer needs to know both formal and informal parts** of the interface before invoking/using them

# Two principles for better interface design

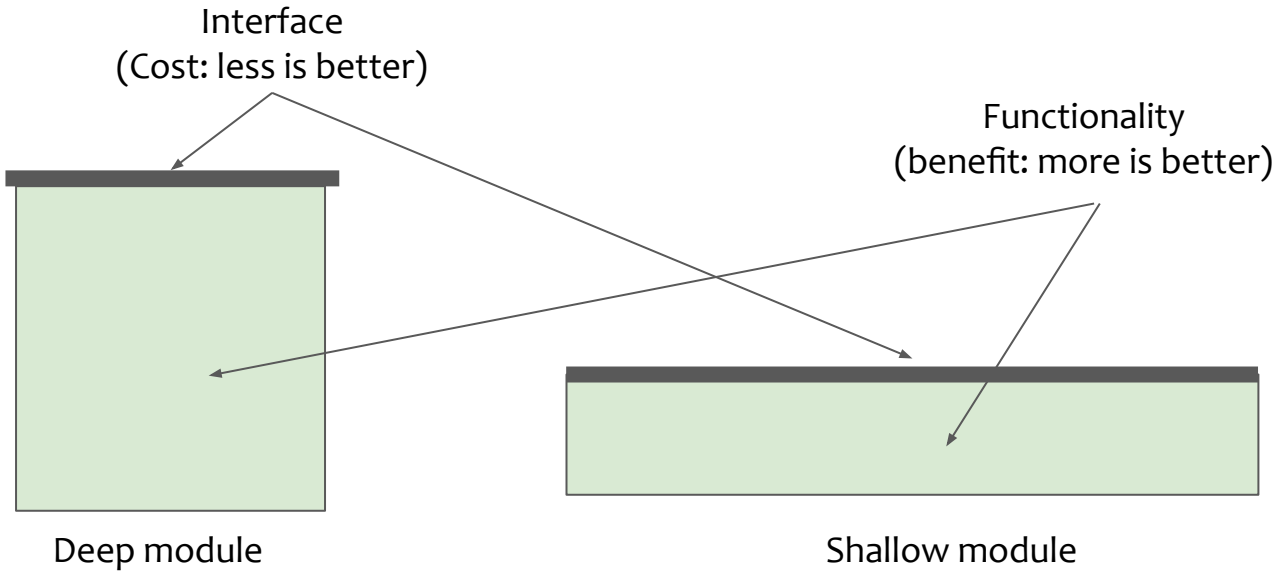
1. Deep vs shallow modules
    a. Modules should be deep
    b. General purpose modules are deeper
2. Information hiding (and leakage)

# #1: Shallow Vs deeper modules

- **Deep modules:** They provide powerful functionality yet have simple interfaces
  - They provide good abstraction because only a small internal complexity is visible to users
- Module design as a **cost vs benefit trade-off**
  - Cost: The complexity using the interface
  - Benefits: Provided functionality of the module
- **Shallow modules:** Provide huge interface, but less functionality
  - More dependencies with other modules leading to high coupling



# # 1.a Modules should be deep




- **The best modules are deep**
  - They allow a lot of functionalities to be accessed through a simple interface
- **A shallow modules** in one with a relatively complex interface, but not much functionality
  - It doesn't hide much complexity

# #1.a: An example of deep module

- File I/O in Linux primarily primarily only five basic system calls to access a file
  - **Simple yet powerful interface**
    - open() – Opens a file
    - read() – Reads a file in a buffer
    - write() – Writes a buffer in a file
    - seek() – Enables random access in a file
    - close() – Closes a file
  - **Hides a lot of implementation/functionality details**
    - How files are represented on a storage medium/disk?
    - How directories are stored?
    - How data is accessed read/written?
    - How permission are managed?
    - How to cache data for improved performance?
    - How to enable concurrent accesses in a safe manner?
    - And, the list goes on…

As a filesystem user we prefer a simpler interface!

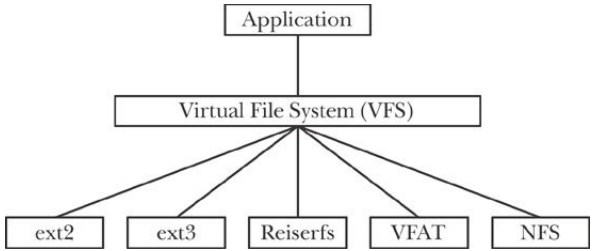
We don't care much about the low-level implementation details!

# #1.b: General purpose modules are deeper

- **Generalization vs specialization:** How general should you design your subsystem?
  - In general, **general purpose modules/subsystems** lead to simpler, cleaners, and easier to understand interfaces
  - One needs to think carefully: **how to provide different services through a common/simpler interface**
- Learn to differentiate **interface** with the **implementation mechanisms**
  - A cleaner interface can incorporate different implementation choices
- **CAUTION:** Over-generalization is also slippery-slope
  - Try to make subsystems **"some-what general"**
  - **Strategic mind-set for programming** (long-term view of the system) will help you!

# #1.b: An example of general-purpose module

- **Linux virtual filesystem (VFS)**
  - Provides a simple interface, yet supports a range of different filesystems
    - Ext3/4
    - BTRFS
    - ZFS
    - And the list is really long?
- General design of VFS led to simpler interface, yet provide rich functionalities



# #2: Information hiding (and leakage)

- **Basic idea:** Each module should encapsulate a few pieces of knowledge about design decisions, and the knowledge is embedded in the implementation, but not visible in its interface
  - The design decisions are implementation choices to achieve the functionality
  - Hiding low-level implementation details
- **Information hiding leads to good interface design and reduce complexity**
  - **Simple interface design** by abstracting away the low-level functionalities
  - Also, it helps in **long-term evolution of the system**
    - The **interface remains stable**, while the implementation can change, e.g., to suit a new hardware technology

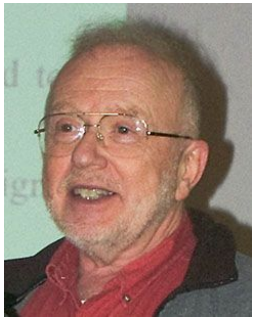
# #2: Avoid information leakage

- The opposite in **information leakage**: Information leakage occurs when a design decision is reflected in multiple modules
  - This **creates a dependency between the modules:**
    - Any leaked information will be visible by the interface, which will lead to increased dependencies between modules
  - When designing a module, you should think carefully to **hide as much as low-level implementation details**

# #2: An example of information hiding principle

- In **object-oriented programming:** 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



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

**Understanding David Parnas' Information Hiding and System Modularization:**
https://mendelbak.medium.com/understanding-david-parnas-information-hiding-and-system-modularization-f491420d2d87

# Outline

- ~~**Part I:** System design challenges~~

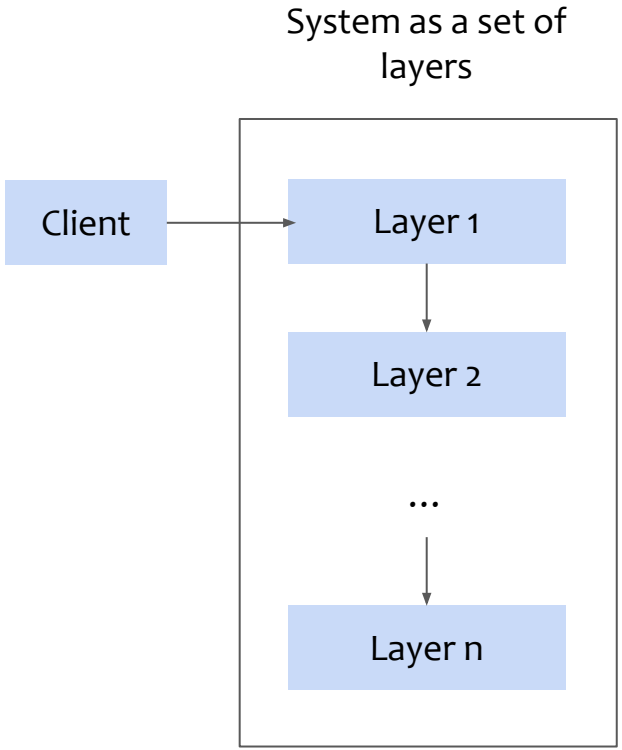
- ~~**Part II:** Modularity~~

- **Part III: Software architecture: Layered architectures**
  - Layered architectures and their ubiquitous adoption in systems
  - Open vs closed layered architectures
  - Different layers, different abstraction
  - Pulldown the complexity downward/upward

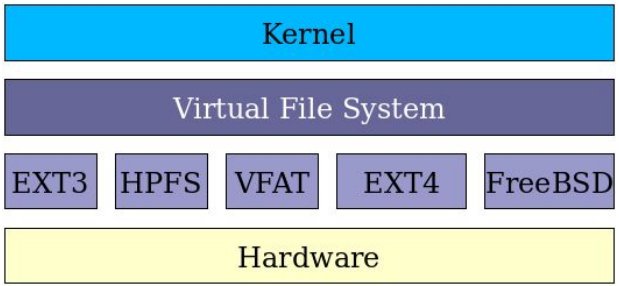
- **Part IV:** Data management

# Layered architecture

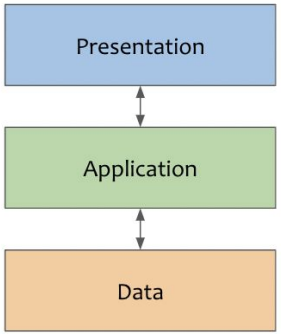
- **A Layered architecture** achieves modularity by dividing the system into distinct layers (subsystems/components)
- A layer is a subsystem that provides a service to another subsystem with the following restrictions
  - A layer only **depends on services from lower layers**
  - A layer **has no knowledge of higher layers**
- This separation of concerns makes the s**ystem more organized**, **easier to maintain**, and promotes better **code reusability**



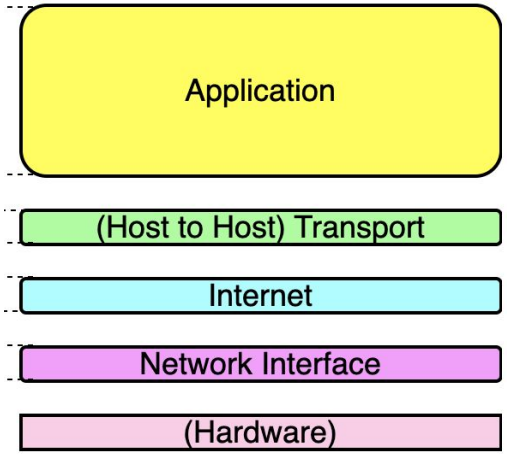
# Layered systems are everywhere!




Filesystems




Three-tier web applications

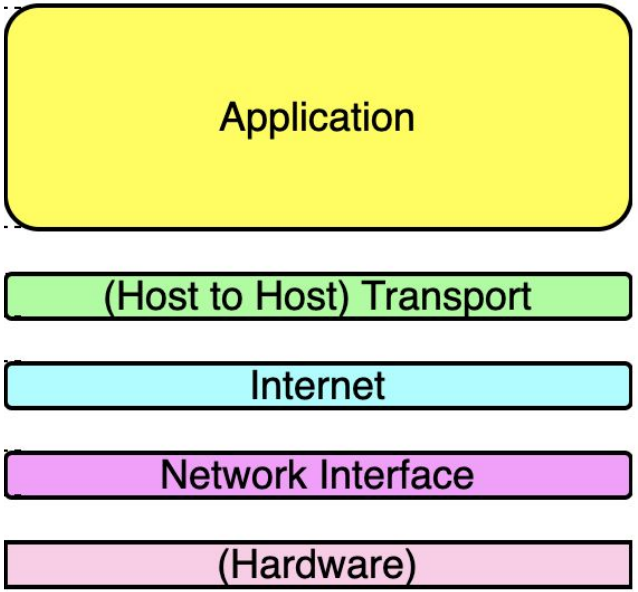



Networked systems

# An example: Network stack as a layered architecture

**Network stack** is usually organized as a layered architecture (e.g., TCP/IP model):

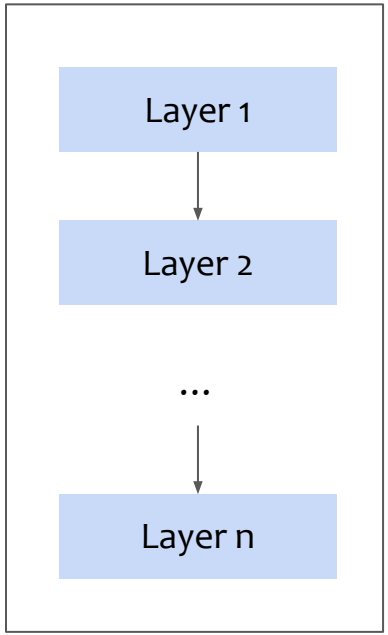
- Layer 1: Network access layer
  - handles the physical layer (Ethernet/Wireless)
- Layer 2: IP layer
  - Routing of the traffic
- Layer 3: Transport layer
  - Reliable data communication
- Layer 4: Application layer
  - Applications accessing the network (RPC/Sockets)



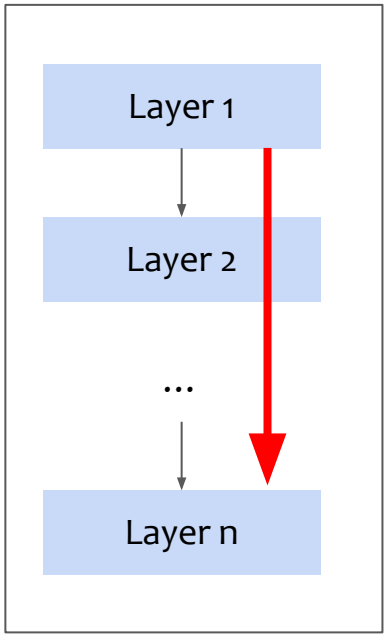

TCP/IP network architecture: https://www.avg.com/en/signal/what-is-tcp-ip

**Vinton G. Cerf and Robert E. Kahn led the design and implementation of the Transmission Control Protocol and Internet Protocol (TCP/IP) that are the basis for the current internet – Both were awarded Turing Award for their contributions!** https://amturing.acm.org/award_winners/kahn_4598637.cfm

# Layered architecture: Open vs closed



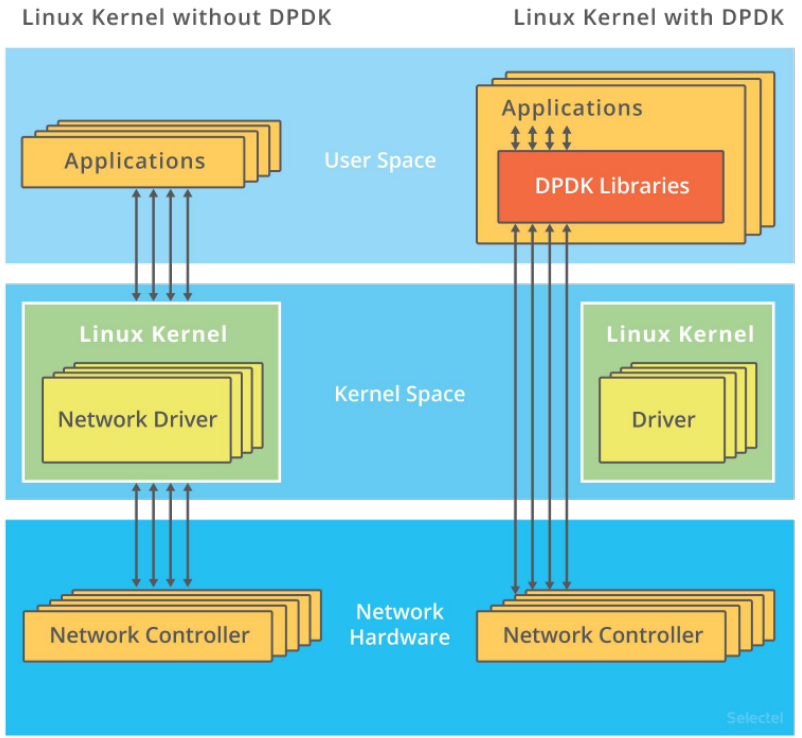

**Closed layered architecture:**
If each layer can only call operations from the layer directly below
**Pros:** Cleaner system interfaces and design, portability, maintainability



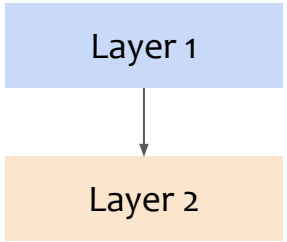

**Open layered architecture:**
A layered architecture is open if a layer can call operations from any layer below
**Pros:** High-performance

# An example of closed vs open layered architecture

- **Network stack can be organized:**
  - **Closed layered architecture (Left)**
    - Applications communicate with network devices (NICs) via the Linux kernel network stack (TCP/IP)
    - Structured communication and APIs
  - **Open layered architecture (Right)**
    - Modern network devices are fast and Linux network stacks slows down communication
    - Userspace network libraries (such as DPDK) allow direct communication between application and network devices, i.e., bypassing Linux network stack, leading to high-performance

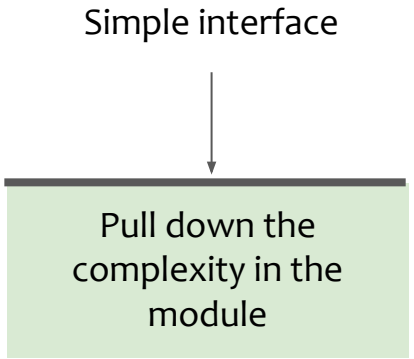


# Design tips for layered architectures




**A:**

Different layer, different abstraction



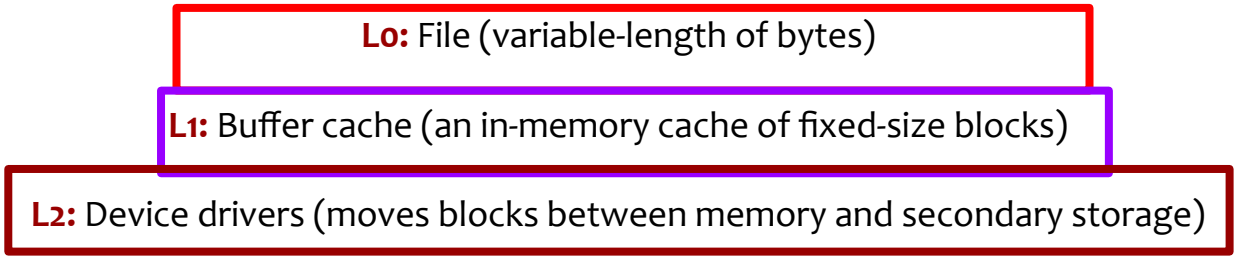

**B:**

Pull complexity downwards

# A. Different layer, different abstraction

- In a well-designed system, **each layer provides a different abstraction** from the layers above or below it
- If **two adjacent layers provide the same abstraction,** it is **NOT a good decomposition** of system into layers

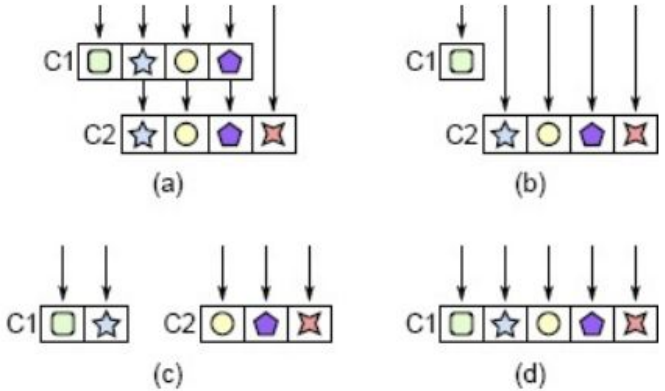
**File system as a layered architecture with three layers (L0, L1, L2):**



# Pass-through methods violate this principle!

- **Pass-through method**
  - When adjacent layers have similar abstractions, the problem often manifests itself in the form of pass-through methods
- A **pass-through method is one that does little except invoke another method!**
  - They don't improve the functionality
- **When is interface duplication OK?**
  - **Each new layer/method should improve the functionality**
  - Even though they work on same signature, **they work on a different level of abstraction**

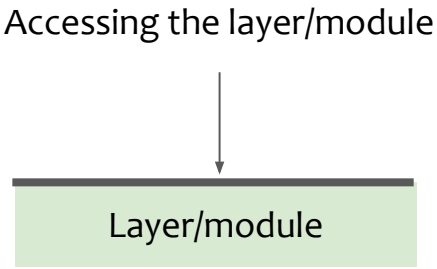
# Pass-through methods




In (a), class C1 contains three pass-through methods, which do nothing but invoke methods with the same signature in C2 (each symbol represents a particular method signature). The pass-through methods can be **eliminated by having C1's callers invoke C2 directly** as in **(b),** by **redistributing functionality** between C1 and C2 to avoid calls between the classes as in **(c),** or by **combining the classes** as in **(d)**.

# B: Pull down complexity

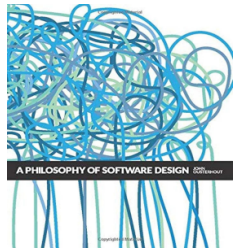
- **Hide complexity:** If you have a module/layers that introduces complexity to the system due to the functionality it provides, it is better to hide that functionality inside that module
  - **Most modules have more users than developers**, so it is better to hide the functionality to have the developers deal with it
- **Example:** config parameters
  - They **could obviously be useful and provide flexibility**
  - However, **they can also be misused and pass the responsibility of configuration to the users** when as a developer you could have handled it dynamically
- **Ideally,** each module should solve a problem completely and config parameters can indicate an incomplete solution!



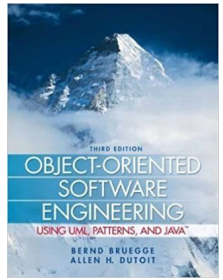

Hide the complexity internally, while keeping the interface easier!

# References

- Book: “A philosophy of software design”
  By John Ousterhout
  - Chapter 4: Modules should be deep
  - Chapter 5: Information hiding (and leakage)
  - Chapter 6: General purpose modules are deeper
  - Chapter 7: Different layer, different abstraction
  - Chapter 8: Pull down complexity
  - https://milkov.tech/assets/psd.pdf




- Book: “Object-Oriented Software Engineering Using UML, Patterns, and Java”
  By Bernd Bruegge and Allen H. Dutoit
  - Chapter 6 (System Design: Decomposing the System)
  - Chapter 7 (System Design: Addressing Design Goals)



# Today's learning goals

- ~~**Part I:** System design challenges~~
- ~~**Part II:** Modularity~~
- ~~**Part III:** Software architecture: Layered architectures~~
- **Part IV:** **Data management**
  - Key Value (KV) stores
  - Filesystems
  - Shared log
  - Databases
- **Part V:** Pattern implementation

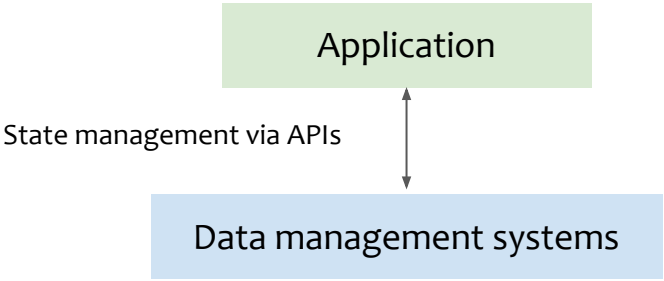
# Data management

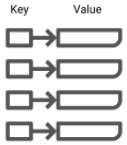
- **State management:** An application developer models the data as objects or data structures, and uses APIs to manipulate those data structures

- A **data management system** offers
  - **Data model:** An abstraction for storing and retrieving data objects
  - **Application programming interface (APIs):** Interface for manipulating data objects

- Data management systems present **different trade-offs** in terms of interface, performance, programmability, reliability, and security properties
  - A complex application may make use of different data management systems

# Data management systems




**A:** Key-value stores

**B:** Filesystems


**C:** Shared logs


**D:** Databases

# Lecture structure for each data management system

- **Single-node abstraction**
  - Usage model
  - APIs
  - Illustrative systems
- **Distributed system architecture**
  - For **scalable data management in the cloud**
  - A **single machine can't store and serve** large amounts of data (or " Big Data")!

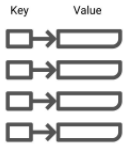
**Covered in the Scalability lecture!**

# Data management systems

Application

State management via APIs

Data management systems




**A:** Key-value stores

**B:** Filesystems


**C:** Shared logs


**D:** Databases

# A: Key-value store (KVS)

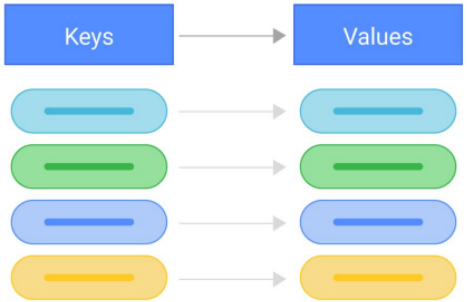
- **KVS** stores data as a set of unique identifiers, each of which have an associated value
  - **Key:** a unique identifier
  - **Value:** An arbitrary string

| Key (Name) | Value (Age) |
|---|---|
| Pramod | 37 |
| Martin | 38 |

- **APIs:**
  - Get(key) → Value
    - Retrieves the value for a given key
  - Put(key, value)
    - Stores the new value for a given key

# KVS usage

- **Caching**
  - Using cache to accelerate application responses, i.e., when your application needs to handle lots of small continuous reads and writes
- **Simple access model**
  - For applications that don't require to support complex queries, simple get/put interface
- **Scalable data management system**
  - NoSQL systems: Relatively simple to scale out system (caveat for read-only workloads)



# Types of KVS

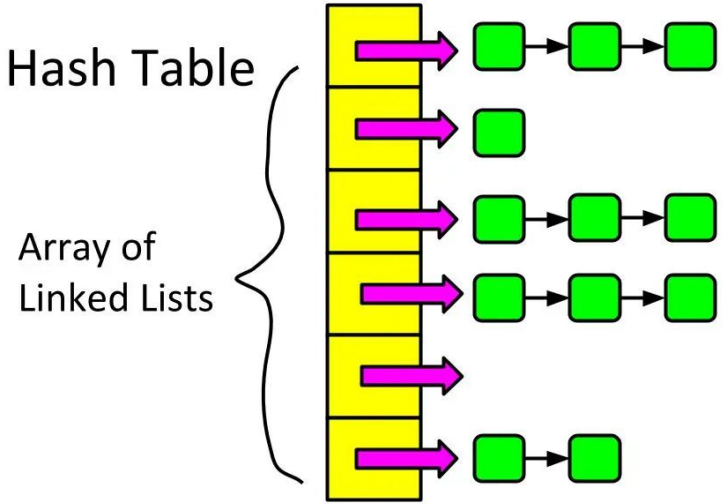
- **In-memory KVS systems:**
  - Maintain the entire state in the main memory
  - Volatile state (no fault tolerance)
  - Primarily for fast look-ups on the read path by caching values
  - E.g., Memcached: https://memcached.org/


- **Persistent KVS systems:**
  - Maintain the state in memory (fast) and on disk/state (non-volatile)
  - Better suited for fault-tolerance
  - E.g., RocksDB: https://rocksdb.org/

# MemCached: An In-memory KVS

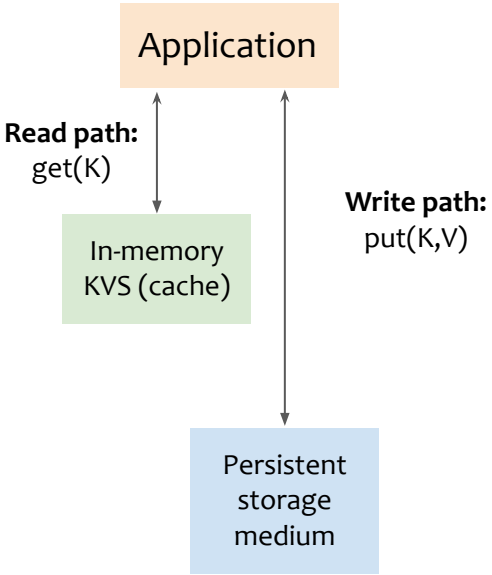
- **Memcached:** A Single-node, in-memory KVS
  - Widely deployed for caching objects (fast lookups)
  - Serves mostly read-only workloads
  - Writes require updating the state in persistent storage medium
- **Design:** Core index data structure
  - **Hash table with chaining**
  - Each node in the linked list stores a KV pair

    **Put/Get operations:** Hash (key) to find an entry in the hash table Traverse the linked-list to read/write nodes



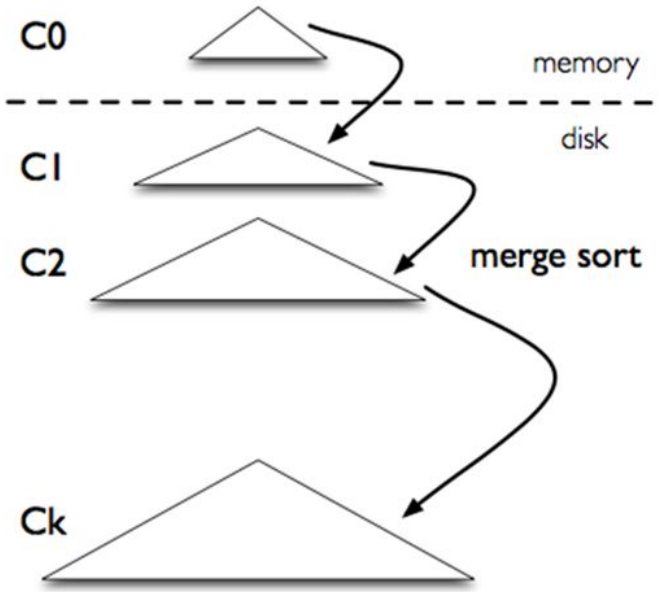
# Example use-case of an in-memory KVS

- **KVS is used for caching on the read path**
  - Read heavy workloads require fast access
  - Heavy hitter objects are cached in-memory KVS
  - **Read path:**
    - Look-up in the cache
    - If cache hit, serve the read request
    - Else, fetch the data from the persistent storage
    - Populate the cache (for the next read request)
    - Return the result
  - **Write-path:**
    - Invalidate the cached entry
    - Write the data in the persistent storage



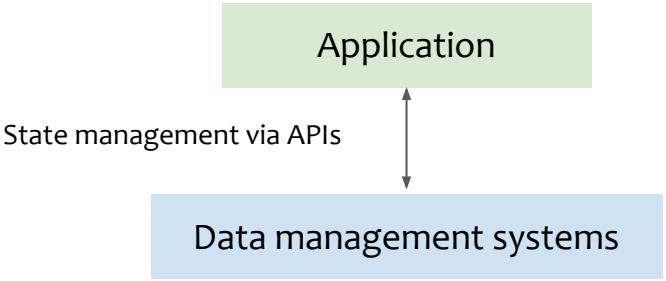
# RocksDB: A persistent KVS

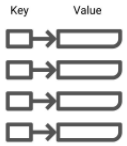
- **Persistent KVS** keep the data fault-tolerant
  - Data is persistent across reboots/crashes
  - Provides KV interface for unstructured data
- **RocksDB design:** LSM-based data structure
  - Log-structured merge trees (or LSM trees) keep the data in memory (fast lookups) and also on persistent storage (persistency)
  - LSM uses a hybrid data structures, spanning main memory (MemTable) and SSDs (SSTables)




- Under the Hood: Building and open-sourcing RocksDB, Engineering @ Meta
  https://www.facebook.com/notes/10158791582997200/
- LSM-Tree, the Underlying Design of NoSQL Database
  https://medium.com/@qiaojialinwolf/lsm-tree-the-underlying-design-of-nosql-database-cf30218e82f3

# Data management systems




**A:** Key-value stores

**B:** Filesystems


**C:** Shared logs


**D:** Databases

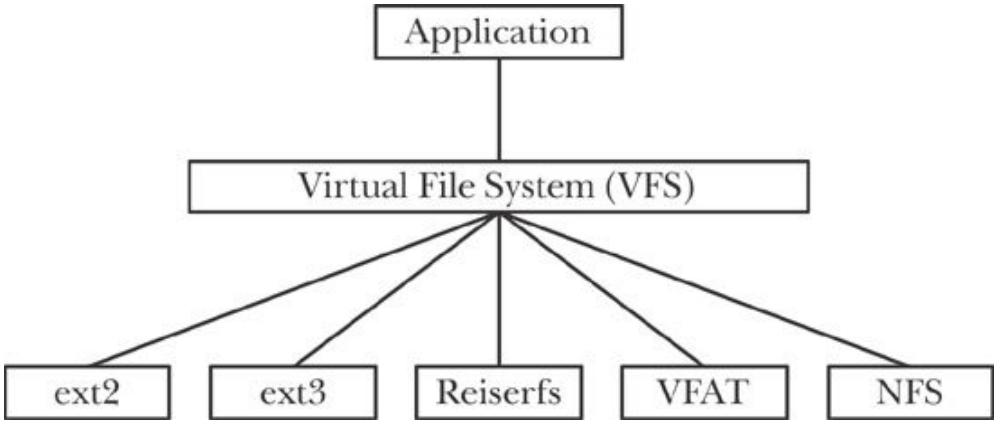
# Filesystems

- A **filesystem** is a data structure that stores data and information on storage devices (hard drives, SSDs, etc.), making them easily retrievable and persistent across system reboots/crashes
-
- Different OS's use **different file systems**, but **they all have similar features**
  - Persistent data management
  - **Namespace:** Hierarchical directory structure with directories and files
  - **Metadata:** file sizes, last modified, etc.
  - **APIs for access the filesystem:** creating, manipulating, reading, writing, deleting, etc.
  - **Security:** Access control lists (ACLs)

Introduction to Linux filesystems:
https://opensource.com/life/16/10/introduction-linux-filesystems

# Linux filesystem

- Linux uses a virtual file system mechanism to provide a unified filesystem interface for applications




Anatomy of the Linux filesystem: https://developer.ibm.com/tutorials/l-linux-filesystem/

# Filesystem APIs

- **Open and close**
  - Opening and closing a file
- **Read, write and position**
  - Reading, writing, and seeking a location for random access
- **Metadata management**
  - File metadata (last access, size, access permissions, etc.)

**File APIs in C programming language:**

- **File management APIs**
  https://www.gnu.org/software/libc/manual/html_node/File-System-Interface.html
- **Input/Output:**
  https://www.gnu.org/software/libc/manual/html_node/I_002fO-Overview.html

# Usage of filesystems

- **Unstructured data store:** A persistent storage medium for raw text
  - A file system is primarily used as an unstructured data store

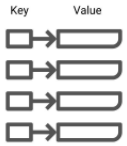
- **General abstraction/storage medium for data managements system:**
  - File systems provide a general abstraction/storage layer for most data management system, i.e., most databases/KVS/Shared log systems are built on top of the general data storage services provided by file systems

# Data management systems

Application

State management via APIs

Data management systems




**A:** Key-value stores

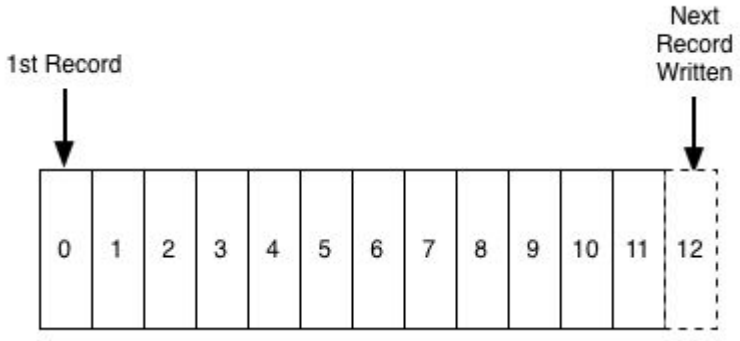
**B:** Filesystems


**C:** Shared logs


**D:** Databases

# Shared logs

- **A log** is an append-only file, totally-ordered sequence of records ordered by time
- **Records are appended** to the end of the log, and reads proceed left-to-right
- Each entry is assigned **a unique sequential log** entry number




Shared logs:
https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

# Shared log APIs

- **Append(X)**
  - Append an entry X and return the log position X occupies
- **Read(P)**
  - Return entry at log position P
- **Trim(P)**
  - Indicate that no valid data exists at log position P
  - Garbage collection works in the background

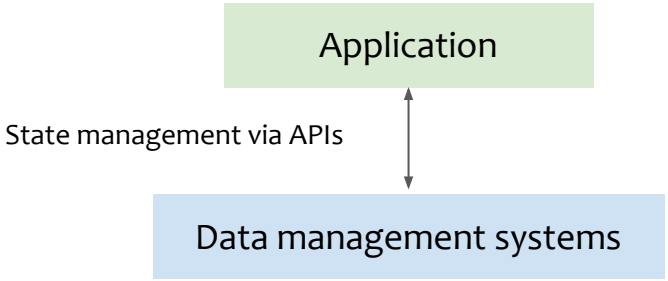
# Usage of shared logs

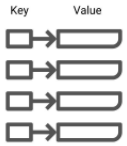
- Shared logs provide **fast sequential read/write performance**

- They are widely used for **storing a stream of bytes**
  - Monitoring systems
  - Debug info/logging
  - Event streams (click streams, events form IoT)

- Large-scale data analytics systems also use shared logs
  - Distributed systems
  - Stream processing
  - IoT applications
  - Database systems (for fault tolerance)

# Data management systems




**A:** Key-value stores

**B:** Filesystems


**C:** Shared logs


**D:** Databases

# Databases

- **A relational database** is a collection of information that organizes data in predefined relationships where data is stored in one or more tables (or "relations") of columns and rows

- **Relationships are a logical connection** between different tables, established on the basis of interaction among these tables, **using a common attribute**


Relational databases: https://cloud.google.com/learn/what-is-a-relational-database

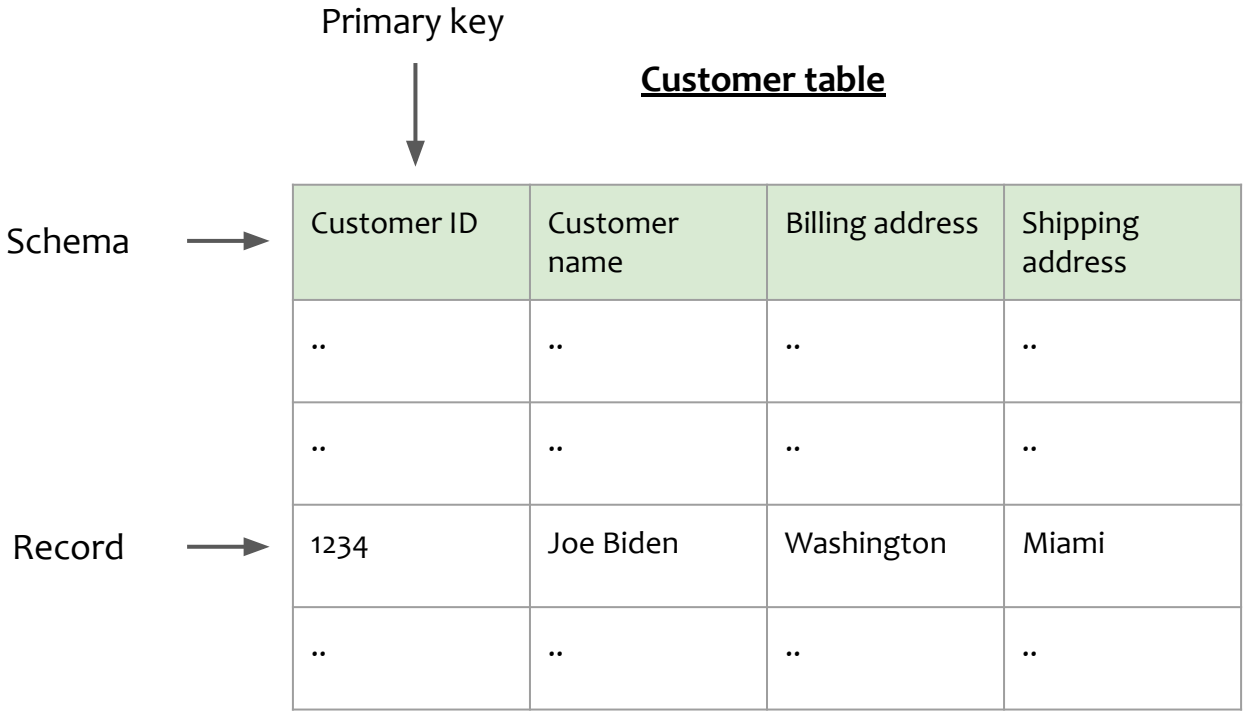
# The relational database model

- Simply, a **collection of "tables"** to organize structured data

- Each table stores information in **columns (attributes)** and **rows (records or tuples)**
  - Attributes (columns) specify a data type
  - Each record (or row) contains the value of that specific data type

- **Tables in a relational database** have an attribute known as the **primary key**, which is a unique identifier of a row, and each row can be used to create a relationship between different tables using a **foreign key**—a reference to a primary key of another existing table

# Relational model



# Data model

**Customer table**

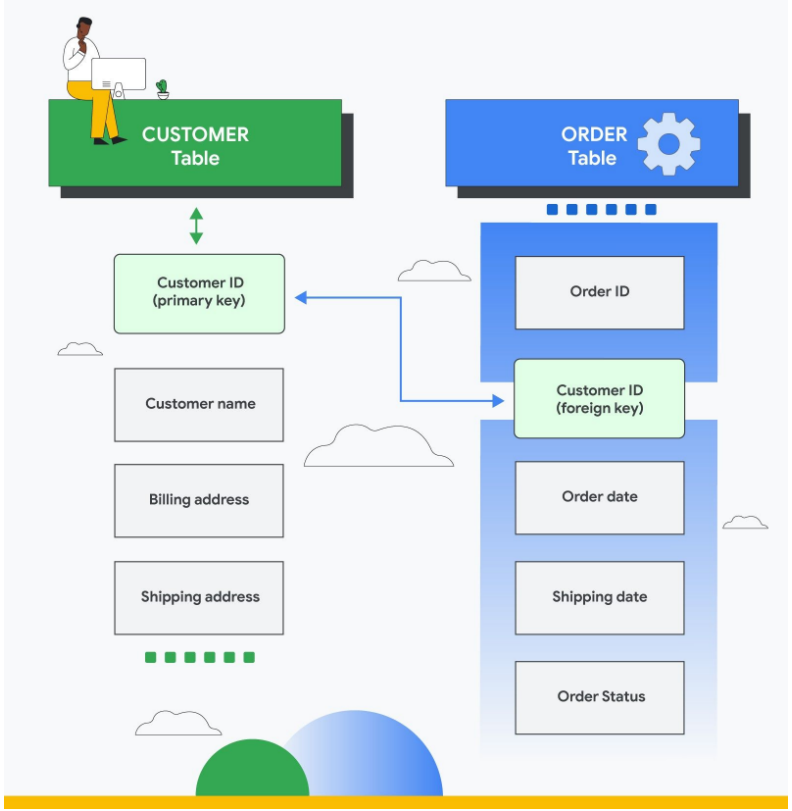
| Customer ID | Customer name | Billing address | Shipping address |
|---|---|---|---|

**Customer ID (primary key)**

**Order table**

| Order ID | Customer ID | Order date | Shipping date | Order status |
|---|---|---|---|---|

**Order ID (primary key)**
**Customer ID (foreign key)**



# Advantages of relational model

- **Databases offer structured data management**
  - Complex analytics via **SQL query language**
    - A rich declarative query interface
  - Built-in **ACID transactions**
    - Concurrency control for multiple clients

# SQL

- **SQL (Structured Query Language)** is a relational query language that is used for interacting with the database
  - SQL is used for accessing, storing, and manipulating data in a relational database
- SQL provides a declarative interface using a wide range of operators
  - **Data definition (setting up database):** Create, delete, alter, rename
  - **Data query (querying database):** Select
  - **Data manipulation (manipulating database):** Insert, delete, update
  - **Data control (administration):** Grant, revoke
  - **Transactions (set of tasks in a single unit):** Commit, rollback

What is SQL: https://aws.amazon.com/what-is/sql/

# SQL query example

**SQL query structure:**

**SELECT** *<column-name>*

**FROM** *<table-name>*

**WHERE** *<condition predicate>*

**An example:**

**SELECT** *customer-name*

**FROM** *customer-table*

**WHERE** *Billing-address=”Munich”*

**Customer table**

| Customer ID | Customer name | Billing address | Shipping address |
|---|---|---|---|

SQL basics and operators: https://www.w3schools.com/sql/

# Transaction processing

- In **transaction processing,** work is divided into individual, indivisible operations, called transactions (TXs)
- **Advantages:** TXs shield programmers from low-level management of
  - Concurrent processing of records
  - Integrity of data
  - Manages the prioritization of transaction execution

# ACID properties of transactions

- **Atomicity**
  - All changes to data are performed as if they are a single operation
  - That is, all the changes are performed, or none of them are
- **Consistency**
  - Data is in a consistent state when a transaction starts and when it ends
- **Isolation**
  - The intermediate state of a transaction is invisible to other transactions
  - As a result, transactions that run concurrently appear to be serialized
- **Durability**
  - After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure

https://www.ibm.com/docs/en/cics-ts/5.4?topic=processing-acid-properties-transactions

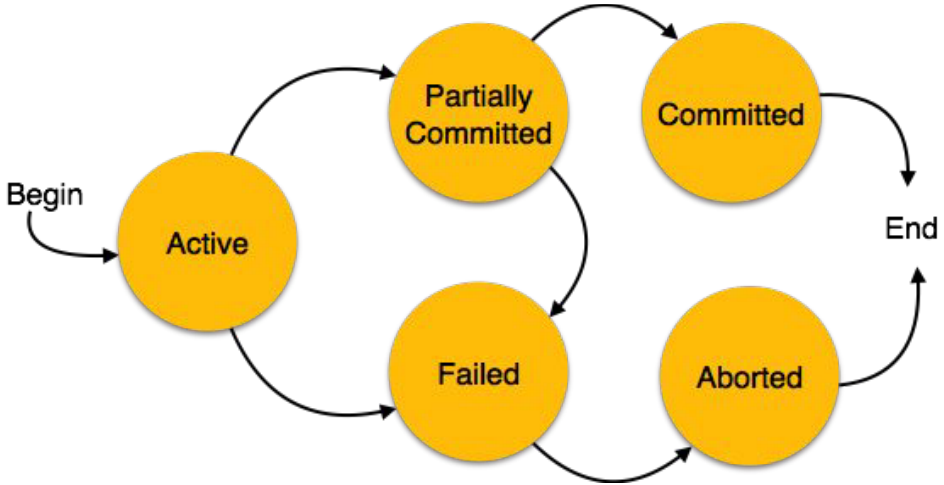
# A transaction example

- Bank transfer transaction of 100 EURs from Alice to Bob

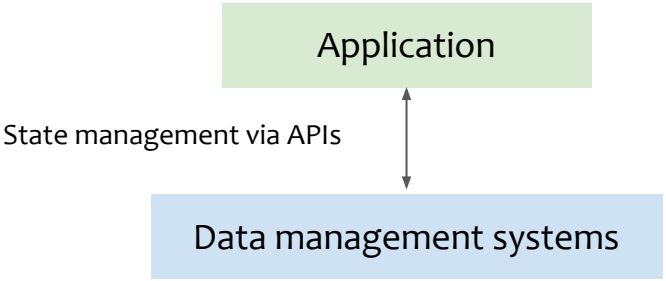
**TX-START**

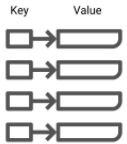
- Withdraw(Alice, 100 EUR)
- Credit(Bob, 100 EUR)

**TX-COMMIT**



# Data management systems




**A:** Key-value stores

**B:** Filesystems


**C:** Shared logs


**D:** Databases

# Outline

- ~~**Part I:** Performance~~
- ~~**Part II:** Concurrency (or Scale Up!)~~
- ~~**Part II:** Scalability (or Scale Out!)~~
- **Part IV: Pattern implementation**
  - MVC pattern

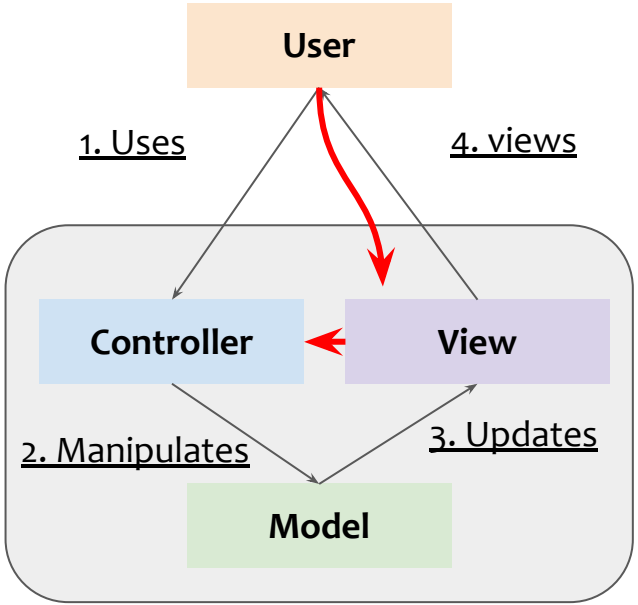
# Today's learning goals

- ~~**Part I:** System design challenges~~
- ~~**Part II:** Modularity~~
- ~~**Part III:** Software architecture: Layered architectures~~
- ~~**Part IV:** Data management~~
- **Part V:** Pattern implementation
  - MVC pattern

# Model-view-controller (MVC)

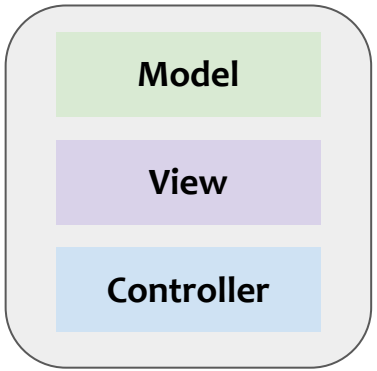
- **Model-View-Controller (MVC)** is a design pattern that separates an application into three main logical components:
  - the model, the view, and the controller

- **The key idea:** Each of these components are built to handle specific aspects of an app
  - *Some of your code holds **the data of your app (model)**, some of your code makes your **app look nice (view)**, and some of your code controls **how your app functions (control)***



# Three parts of MVC – A similar view to a three-tier app!

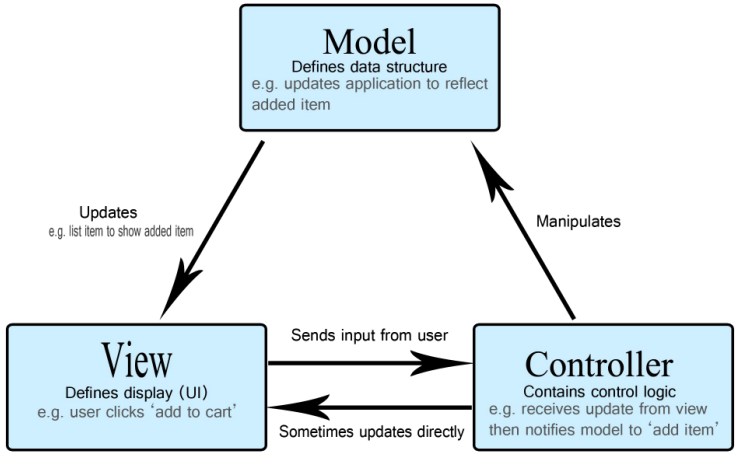
- **Model:** Manages data and business logic
  - If the state of this data changes, then the model will usually notify the view (so the display can change as needed) and sometimes the controller (if different logic is needed to control the updated view)
- **View:** Handles layout and display
  - The view defines how the app's data should be displayed
- **Controller:** Routes commands to the model and view parts
  - The controller contains logic that updates the model and/or view in response to input from the users of the app




**Application**

# An example app: Shopping cart

1. Users updates the shopping cart **– View**
   - Clicks “Add to the cart”
2. **Controller** receives the updates
   - A new item is added to the cart
3. **Controller** notifies to the **Model**
   - Updates the state (e.g., database) to reflect the inventory
4. **Model** updates the **view**
   - The shopping cart list shows the newly added item



# Push vs pull-based MVC

- **Pull notification variant:** view and controller obtain the data from the model
  - Pulling the new updates “on-demand”
  - **Pros:** Saves bandwidth
  - **Cons:** Missing intermediate updates
- **Push notification variant:** the model sends the changed state to view and controller
  - Pushing the updates for “all changes”
  - **Pros:** Not missing any updates
  - **Cons:** Wastes bandwidth

Select the variant that suits your application requirements!

# References for MVC

- Note that there are several variants of MVC, such as
  - Hierarchical model–view–controller (HMVC), model–view–adapter (MVA), model–view–presenter (MVP), model–view–viewmodel (MVVM), and others that adapted MVC to different contexts
- However, the key idea remains the same, the application is usually divided in three components
  - View – User-facing component
  - Controller – Application logic
  - Model – Application state
- https://developer.mozilla.org/en-US/docs/Glossary/MVC
- https://www.codecademy.com/article/mvc
- https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

# Summary

- **Part I:** System design challenges
- **Part II:** Modularity
  - Subsystem decomposition: Modules
  - Differentiate between coupling and cohesion
  - Design pattern: Facade pattern
  - Interface design
- **Part III:** Software architecture: Layered architectures
  - Open vs closed layered architectures
  - Different layers, different abstraction
  - Pulldown the complexity downward/upward
  - Ubiquitous adoption of layered architectures in systems
- **Part IV:** Data management
  - Key Value (KV) store, Filesystems, Shared log, Databases
- **Part V:** Pattern implementation (MVC pattern)