Architecture

CSE 403 Software Engineering

Winter 2026

Today’s Outline

Architecture

1. What do we mean by architecture
2. How does it differ from design
3. What are some common architecture patterns used in software
4. What to consider as you create your architecture

See readings posted on the calendar for more software architecture examples

Short video on merge-conflict material is in Panopto (01-21-26 lecture)

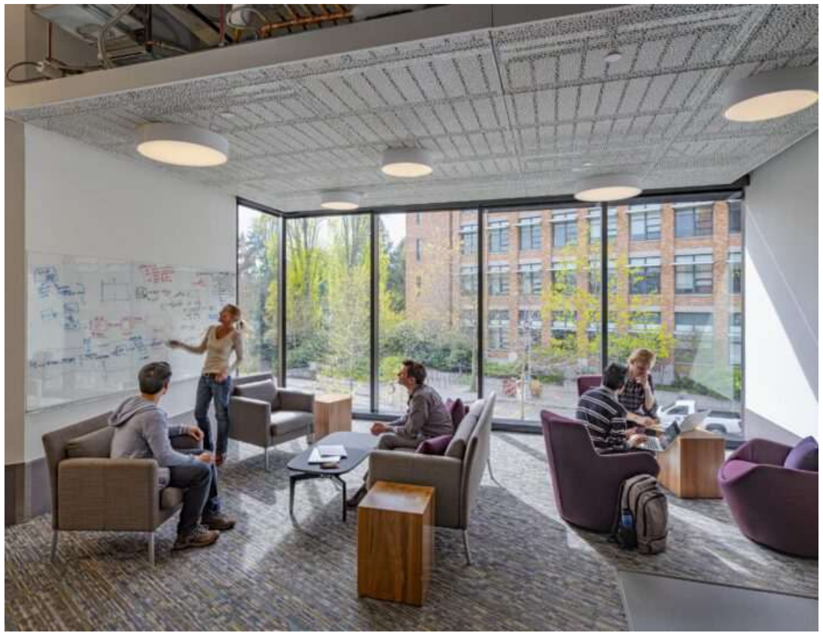
What does “Architecture” make you think of?

MIT Stata Center by Frank Gehry

Paul G. Allen Center by LMN Architects

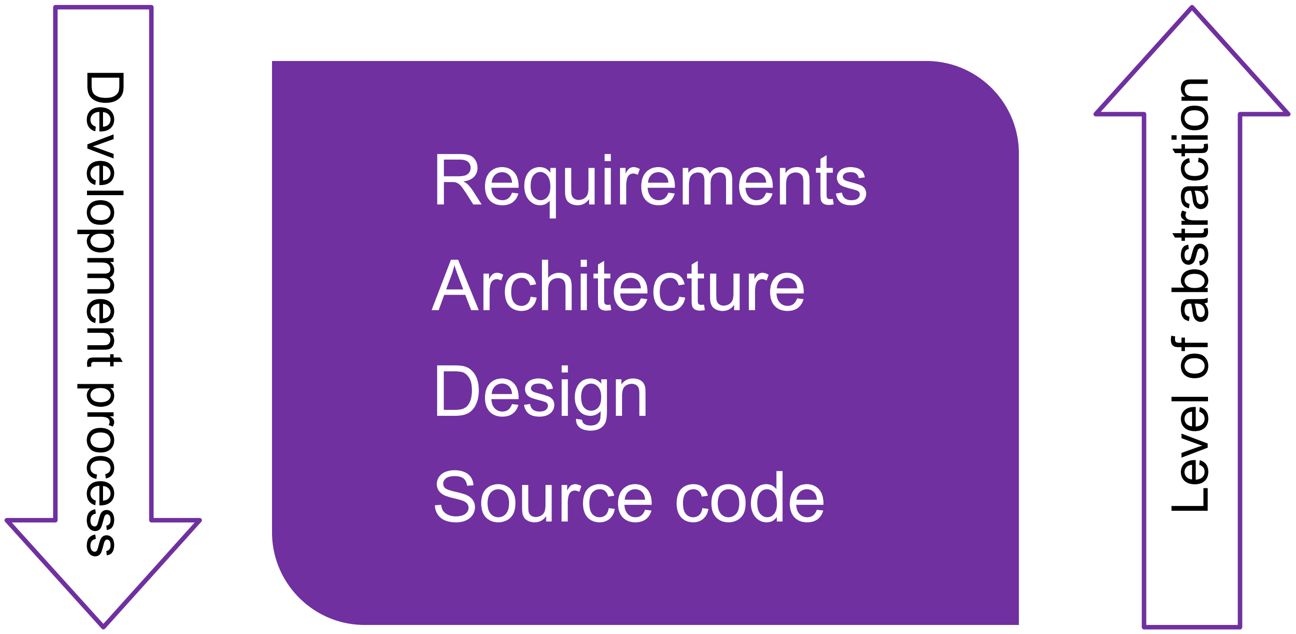
In contrast, what comes to mind with "Design"?

Here’s another example close to home



Bill & Melinda Gates Center for UW CSE - LMN

Let’s transition the ideas to software engineering



The level of abstraction is key

With both architecture and design, we’re building an **abstract representation** of reality

- Ignoring (insignificant details)
- Focusing on the most important properties
- Considering modularity (separation of concerns) and interconnections

High level definitions

Software Architecture (what components and inter-connections are needed)

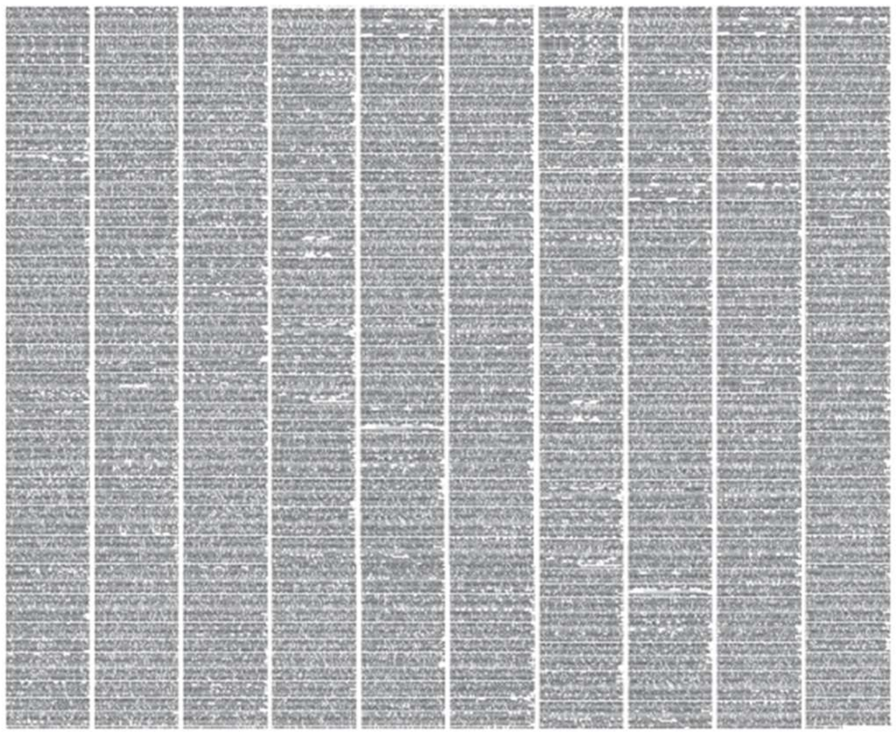
- **Set of structures needed to reason about a software system**
- **Functions as the blueprints for the system and its development**
- Provides a high-level view of the overall system:
  - What are the components
  - What are the connections and/or protocols between components

Software Design (how the components are developed)

- Considers one component at a time
  - Data representation
  - Interfaces, class hierarchy

Case study - abstraction - Linux kernel

Source code

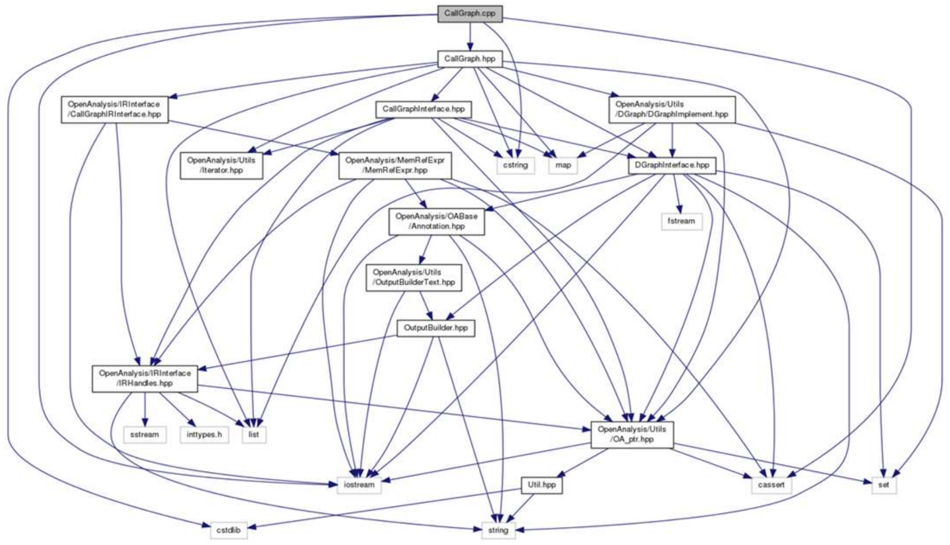


Suppose you want to add a feature
16 million lines of code!
Where would you start?

- **What does the code do?**

Case study – Linux kernel

Call graph

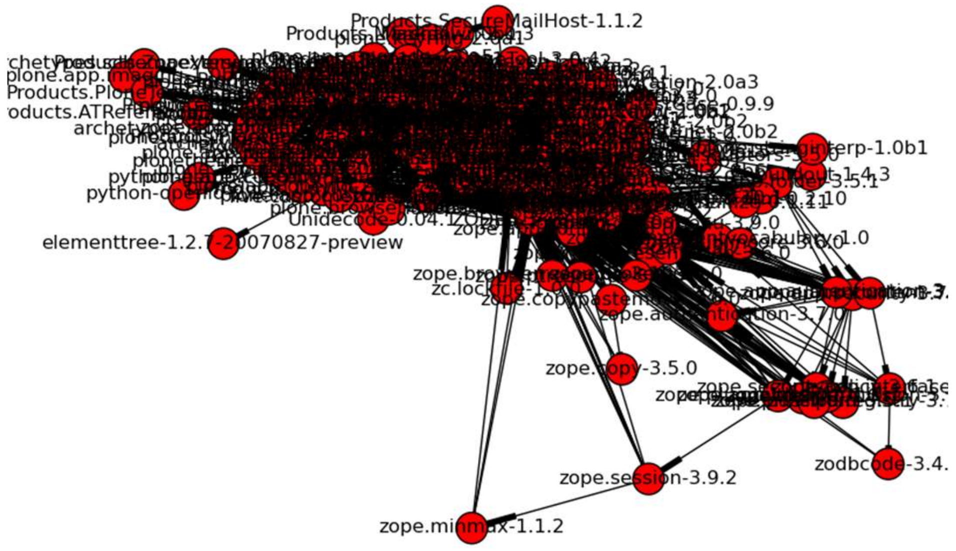

CallGraph.cpp
CallGraph.hpp
OpenAnalysis/IRInterface /CallGraphIRInterface.hpp
CallGraphInterface.hpp
OpenAnalysis/Utils /DGraph/DGraphImplement.hpp
OpenAnalysis/Utils /Iterator.hpp
OpenAnalysis/MemRefExpr /MemRefExpr.hpp
cstring
map
DGraphInterface.hpp
OpenAnalysis/OABase /Annotation.hpp
fstream
OpenAnalysis/Utils /OutputBuilderText.hpp
OutputBuilder.hpp
OpenAnalysis/IRInterface /IRHandles.hpp
sstream
inttypes.h
list
OpenAnalysis/Utils /OA_ptr.hpp
iostream
Util.hpp
cassert
set
cstdlib
string

Suppose you want to add a feature
16 million lines of code!
Where would you start?

- **What does the code do?**

Case study – Linux kernel

Dependency graph

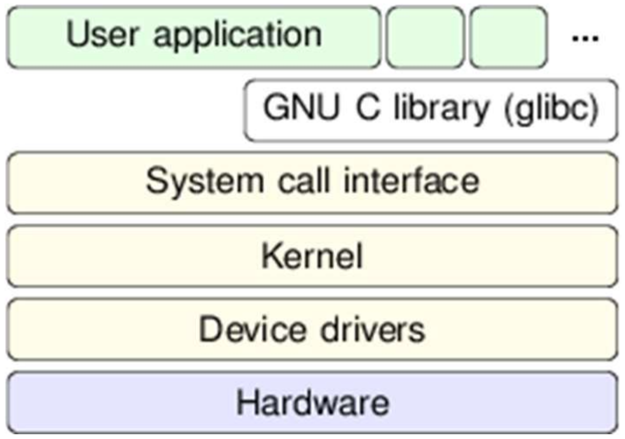


Suppose you want to add a feature
16 million lines of code!
Where would you start?

- What does the code do?
- **Are there dependencies?**

Case study – Linux kernel

Architecural Layer diagram

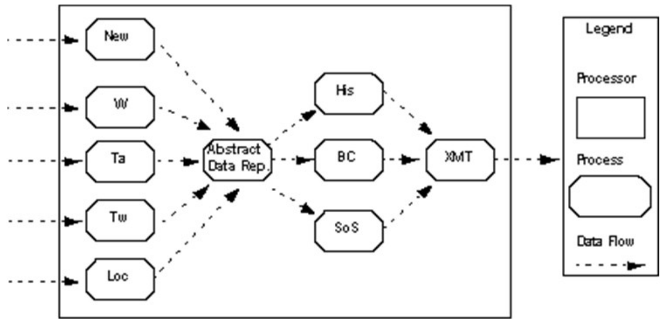


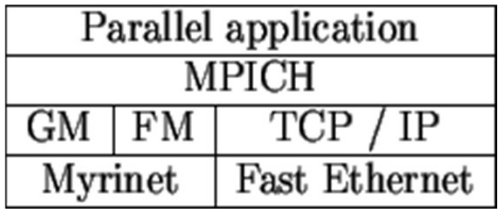
Suppose you want to add a feature
16 million lines of code!
Where would you start?

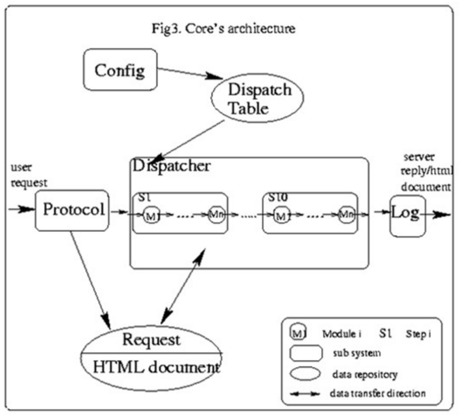
- What does the code do?
- Are there dependencies?
- **What are the different components?**

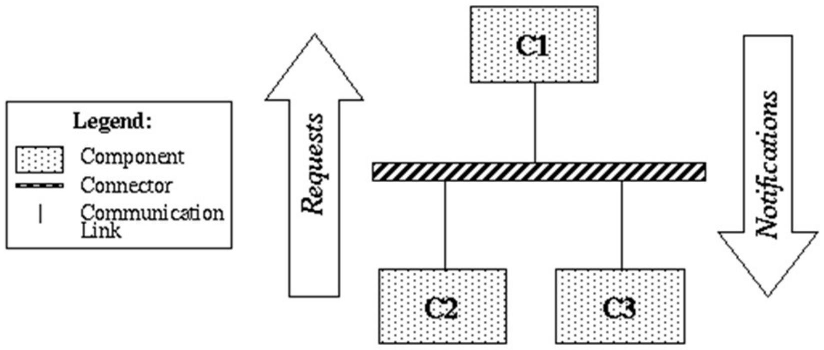
Practically speaking, what does an architecture diagram look like?

Architectures are generally described with box-and-arrow diagrams

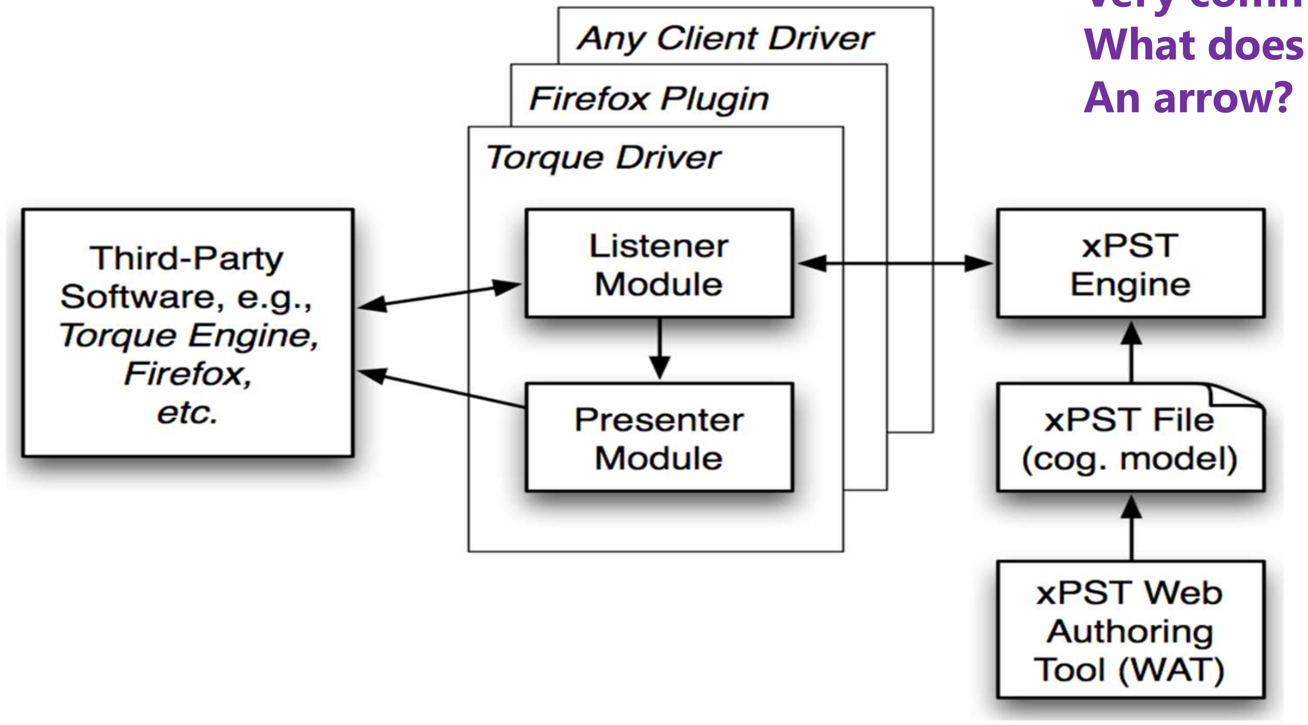








Another box and arrow diagram



Very common and extremely valuable!
What does a box represent?
An arrow?

Architecture diagrams include:

Components (boxes)

- Define the basic computations comprising the system and their behaviors
  - Data management, major services, responsible entities, etc.

Connectors (arrows)

- Define the interconnections (communication) between components
  - Procedure call, event announcement, asynchronous message sends, etc.

Consider the set of structures needed to reason about a software system

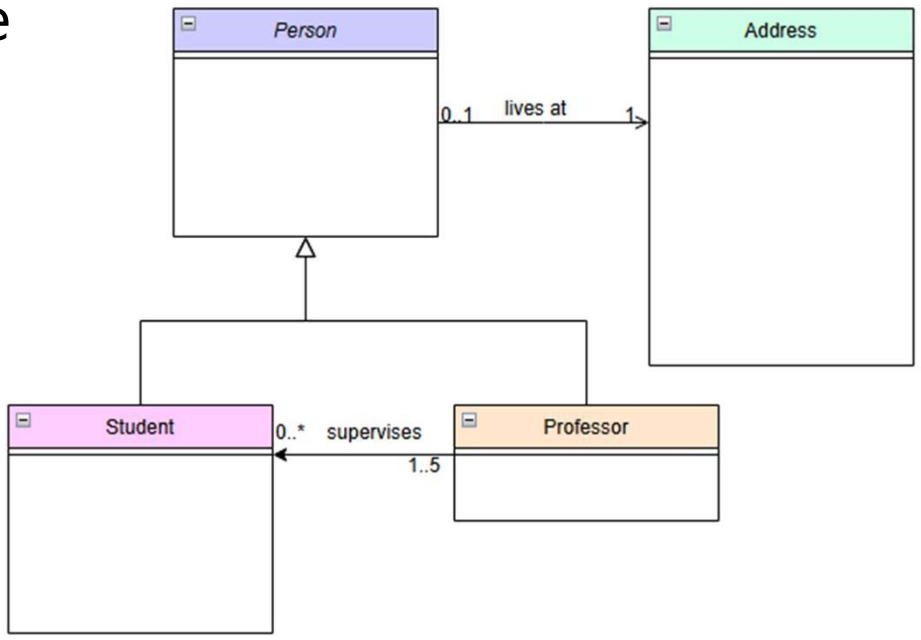
Architecture diagrams using UML

UML = universal modeling language

- A standardized way to describe software architecture and design
- Used in industry
- Not the topic of this lecture

Critical advice about syntax:

- Use consistent notation: one notation per kind of component or connector



Neat free tool: https://www.drawio.com/

Leverage common architecture patterns as you consider how to design one for your system

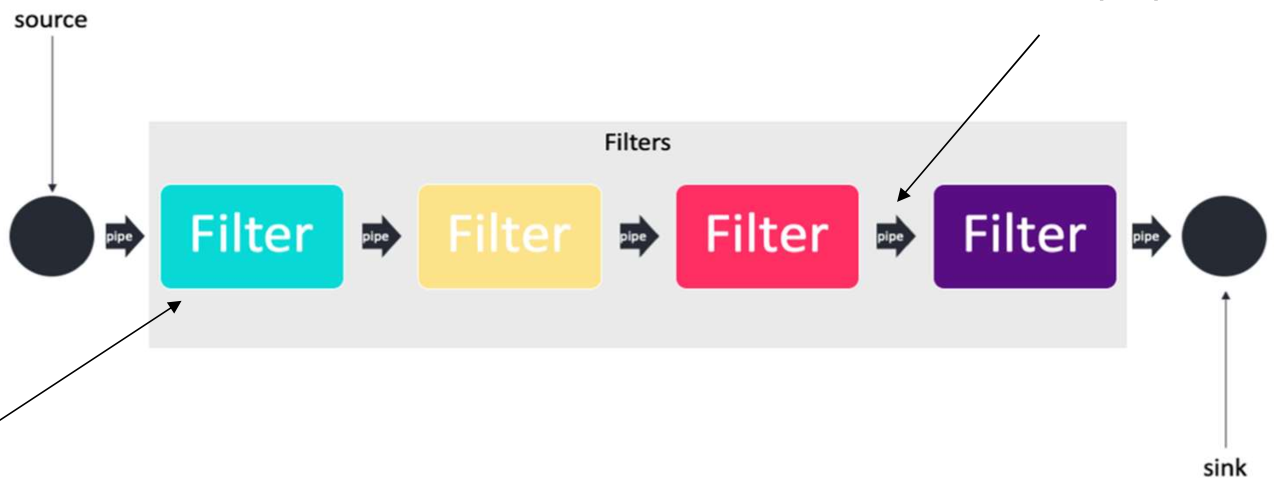
1. Pipe and filter
2. Layered
3. Client-server
4. MVC
5. Micro services

We'll also discuss some overall architectural design principles to consider (these patterns exhibit them!)

SW Architecture #1 – Pipe and filter

The **pipe-and-filter** architecture talks about the main components and the way they connect

Pipe passes the data

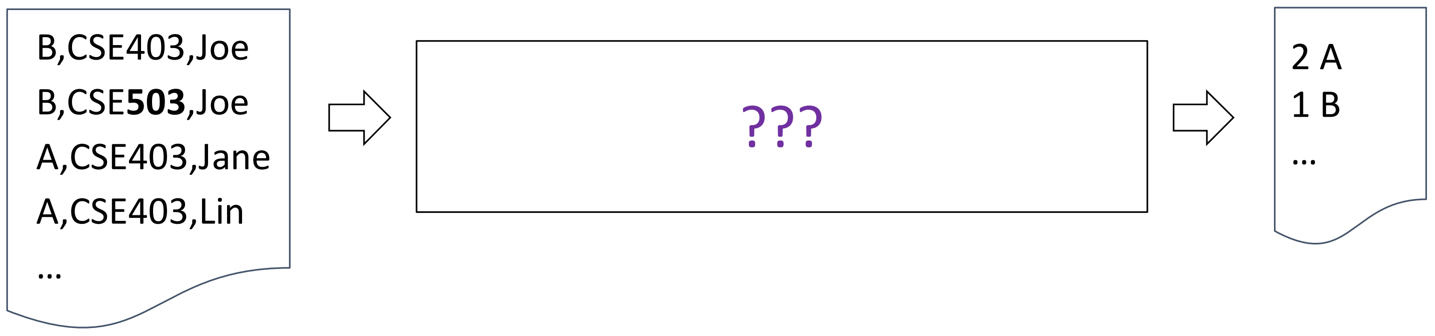


Filter computes on the data

It doesn't specify the design or implementation details of the individual components (the filters)

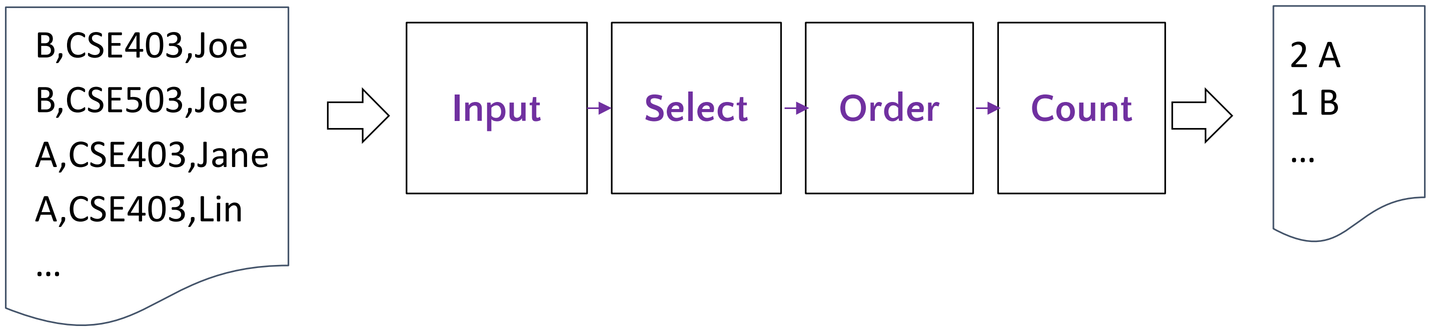
SW Architecture #1 – Pipe and filter

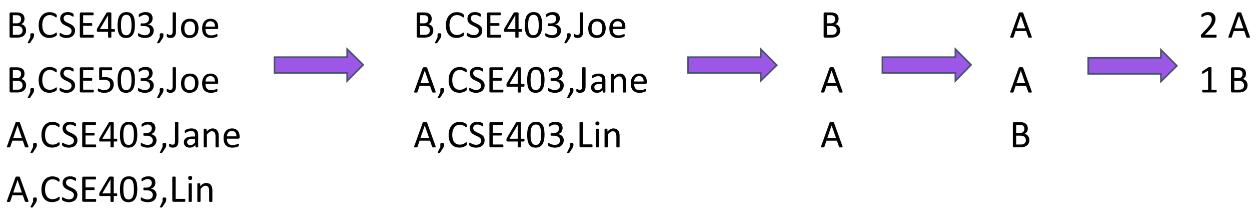
Example: create a histogram of the CSE 403 letter grades



SW Architecture #1 – Pipe and filter

The **architecture** specifies the functional components and their connections

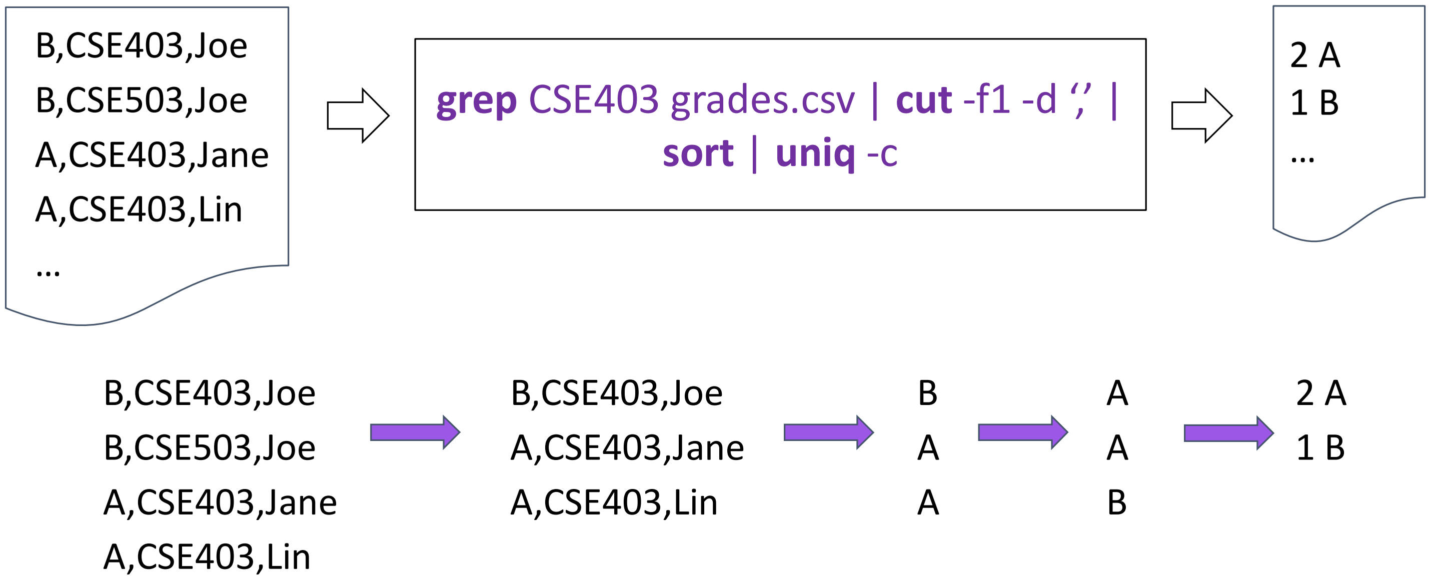




SW Architecture #1 – Pipe and filter

The **architecture** abstraction eventually gets lowered to code

What is a pro and con of pipe and filter architecture?



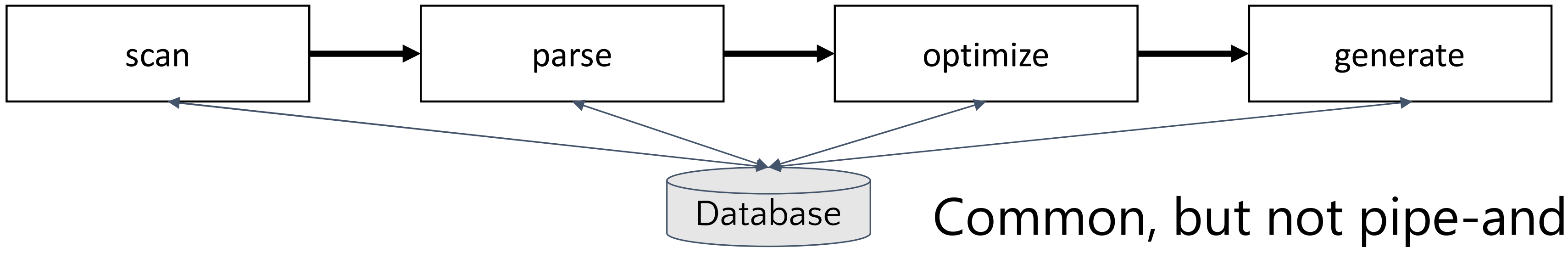
B,CSE403,Joe
B,CSE503,Joe
A,CSE403,Jane
A,CSE403,Lin
…

```
grep CSE403 grades.csv | cut -f1 -d ‘,’ |
sort | uniq -c
```

2 A
1 B
…

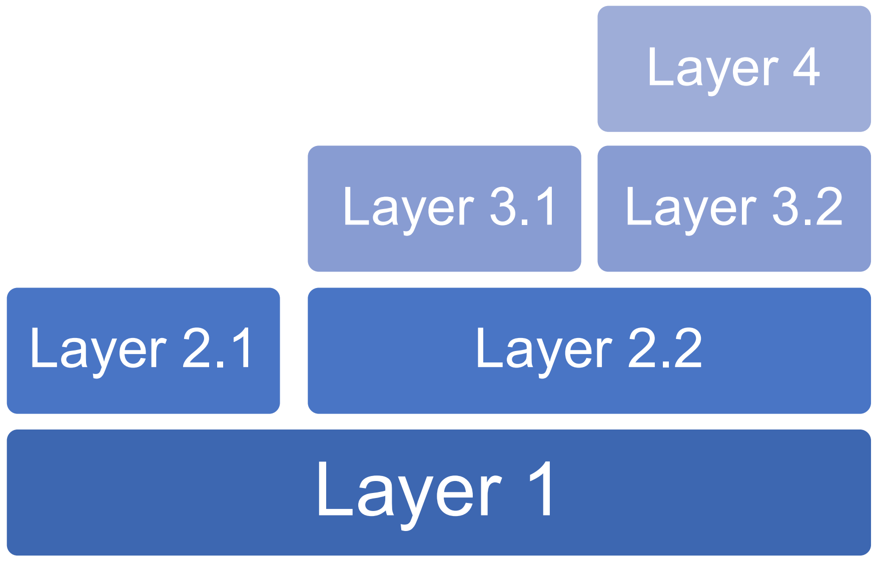
An architectural style imposes constraints

- Pipe & filter
  - Filters must compute local transformations
  - Filters must not access or share state with other filters
  - There must be no cycles in the pipeline
- If these constraints are violated, it's not a pipe & filter system
- Is this pipe and filter?



Common, but not pipe-and-filter

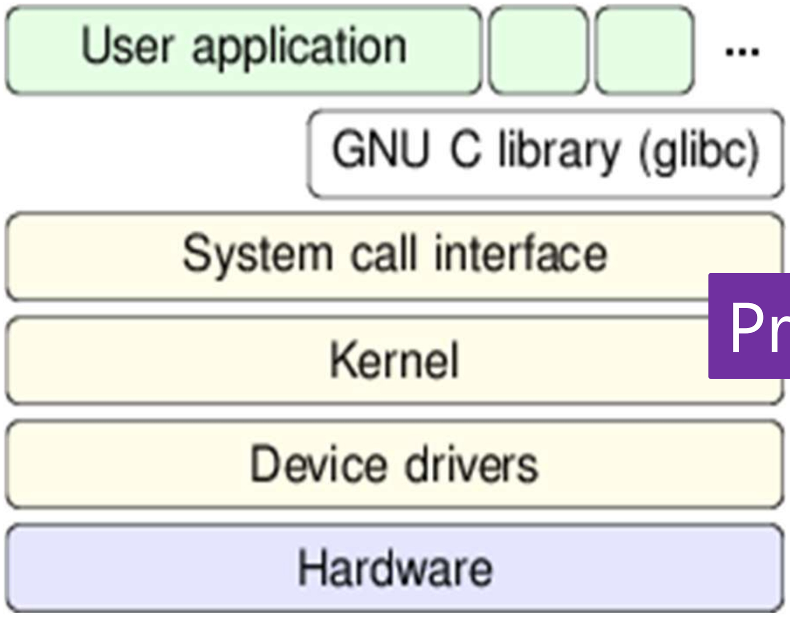
SW Architecture #2 – Layered (n-tier)



- Each layer has a certain responsibility
- Layers only communicate with neighboring layers
- Layers use (depend on) services provided by the layers directly below them
- Layers of isolation – limits dependencies
- Good modularity and separation of concerns

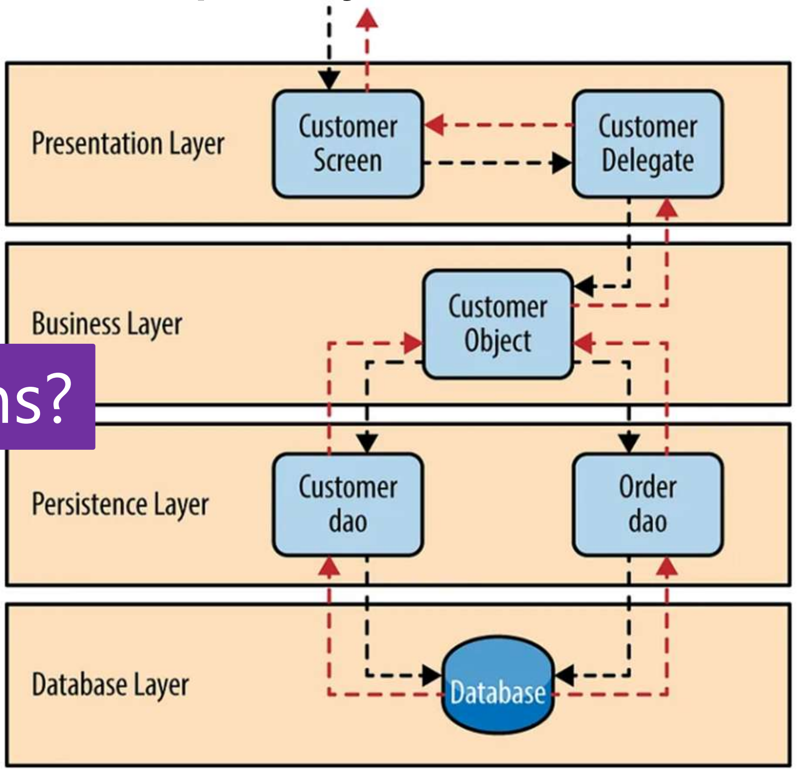
SW Architecture #2 – Layered

Linux Architecture



Pros / cons?

Enterprise System Architecture

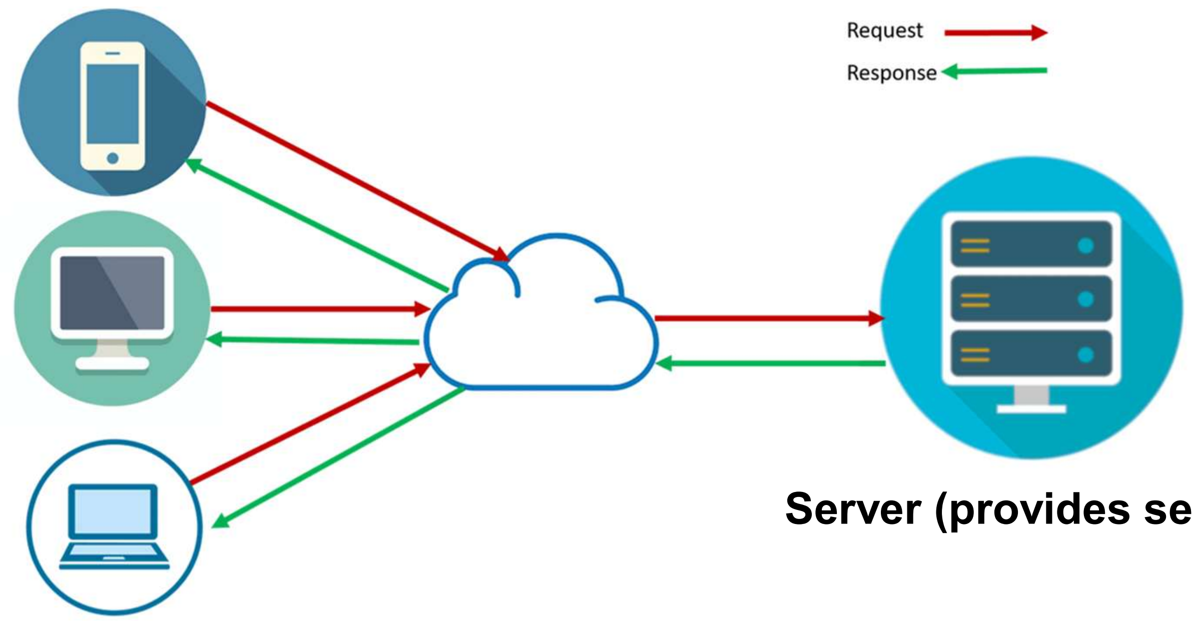


Source: https://www.oreilly.com/ideas/software-architecture-patterns/page/2/layered-architecture

SW Architecture #3 – Client Server

What might be a con of this and how might it be avoided?

Client (requests service)

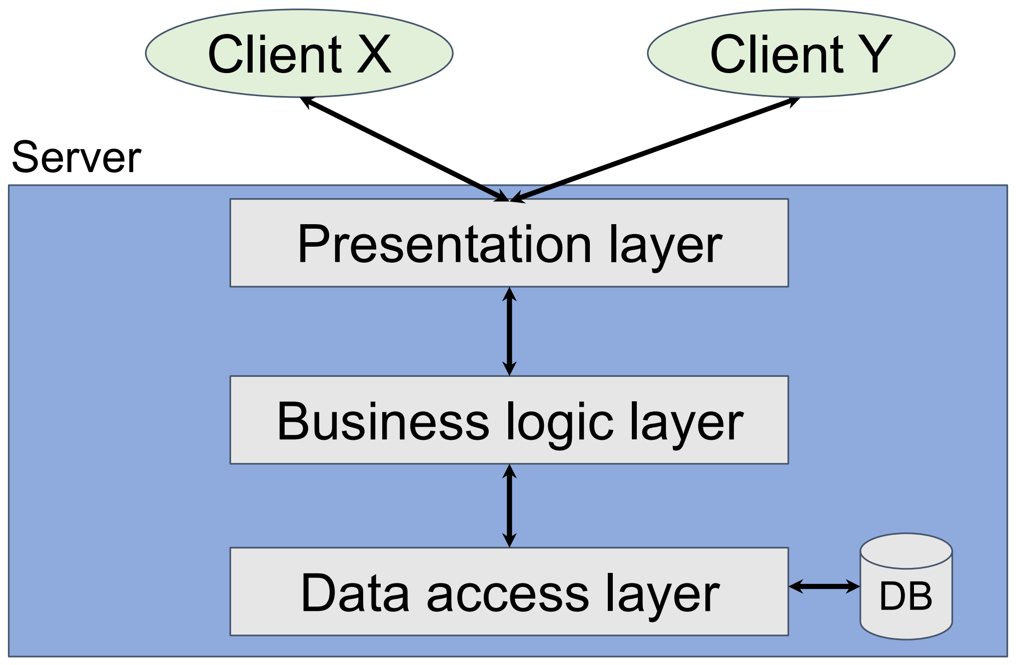


Server (provides service)

Clients can be software that depends on a shared database/service

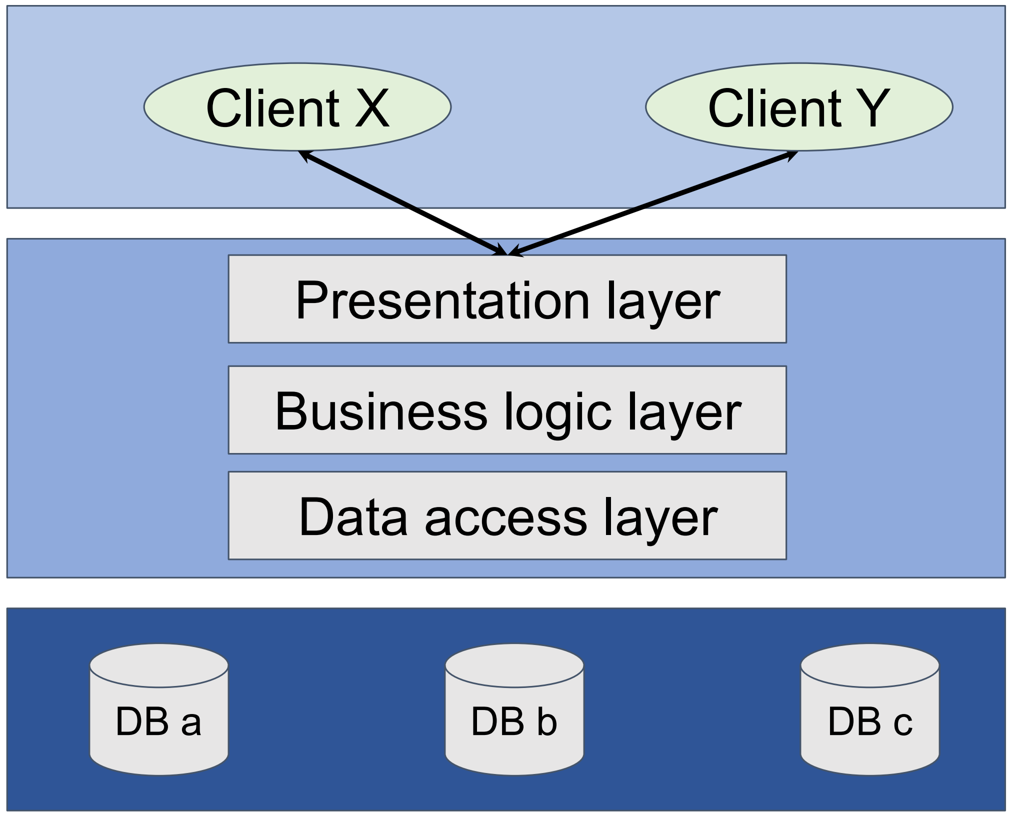
SW Architecture combinations!

Client-Server may be too high a level of abstraction for your purpose
Consider combining with other patterns (e.g., layered)



SW Architecture combinations^2

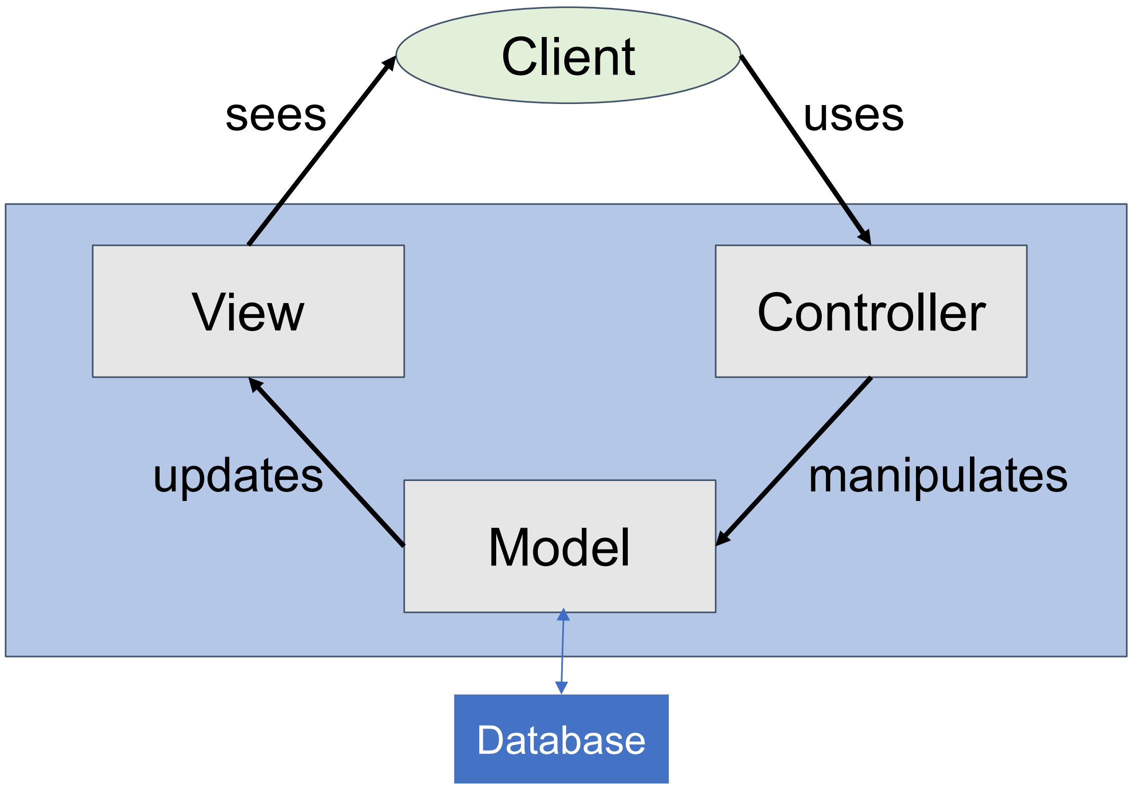
How detailed should an architecture description be?



SW Architecture #4 – Model View Controller

Divides a system into three components:

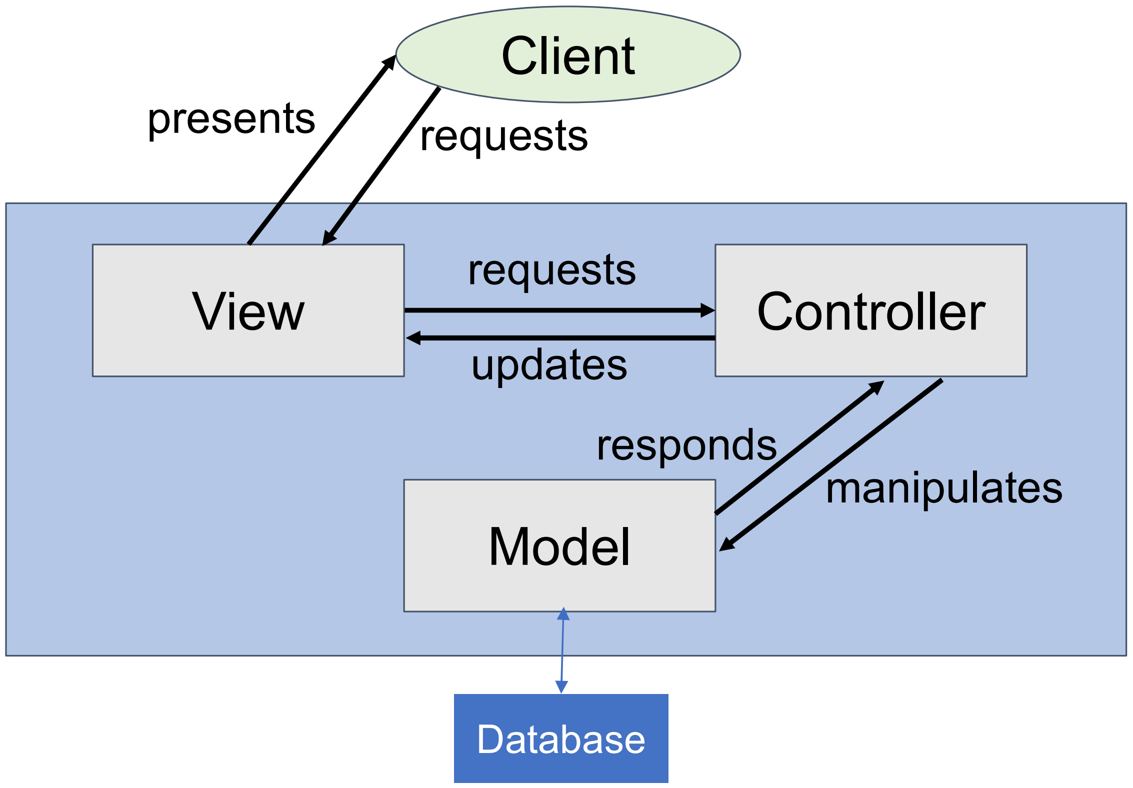
- **Model**
  - Data management
- **View**
  - Presents data to user / provides user interface
- **Controller**
  - Handles control flow / mediates between the view and model



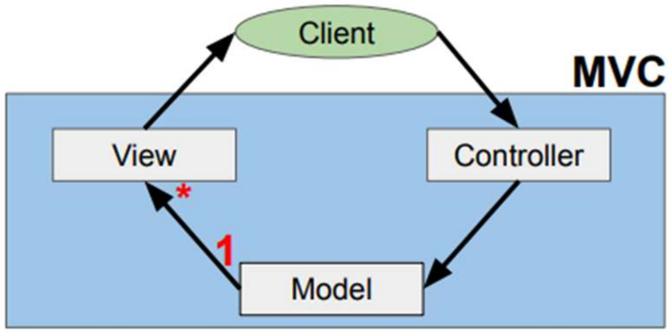
SW Architecture #4 – Model View Controller

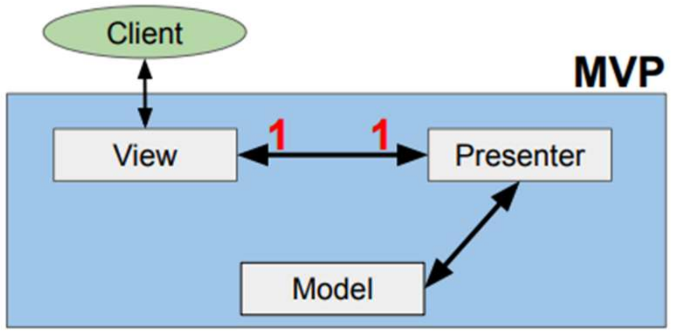
Divides a system into three components:

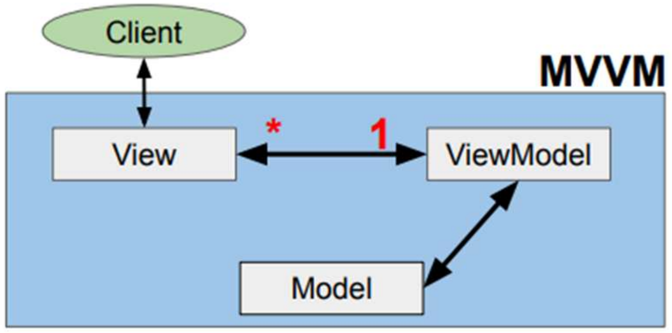
- **Model**
  - Data management
- **View**
  - Presents data to user / provides user interface
- **Controller**
  - Handles control flow / mediates between the view and model



SW Architecture – many variants of MVC





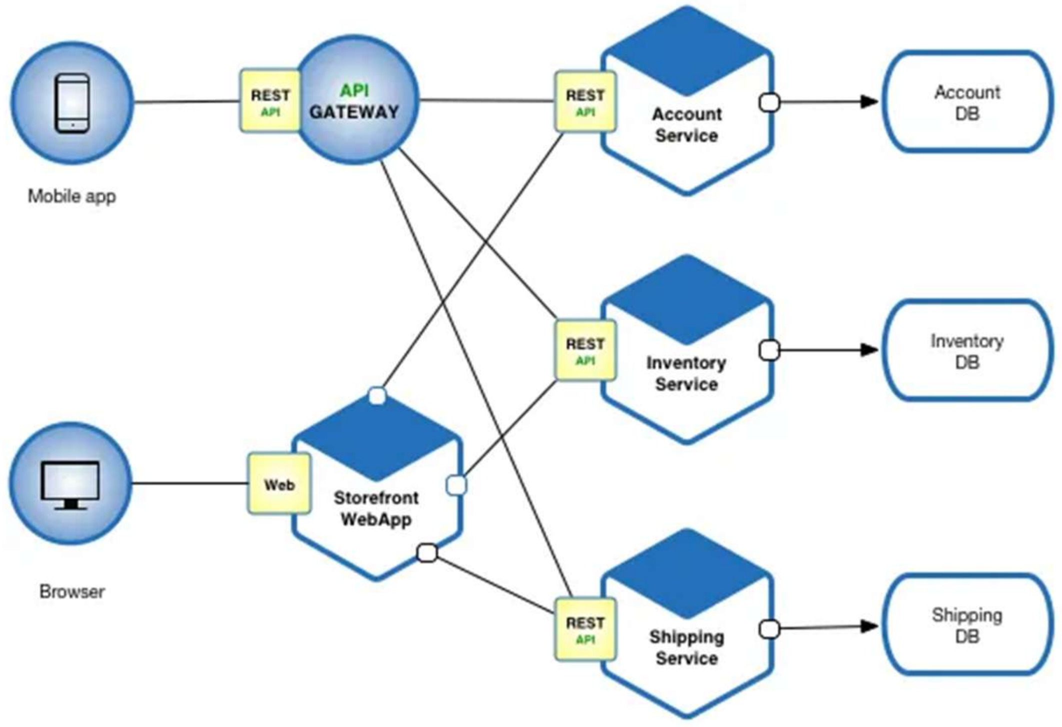


Consider the connections expressed in UML (* == many)

SW Architecture #5: Microservices

Pros / cons?

- Breaks app into small independent modular services
- Each is responsible for specific functionality and communicates with others via apis



https://medium.com/@the_nick_morgan/what-are-the-10-most-common-software-architecture-patterns-faa4b26e8808

What architecture pattern would you choose and why?

https: //PollEv.com/cse403wi

- Weather app like the Weather Channel
- Email service like Outlook or Gmail
- Online banking service like Bank of America
- Online multi-faceted store like Amazon
- Continuous integration tool that supports build>test>commit

What software architecture would you choose?

0 surveys completed

0 surveys underway

Weather app like the Weather Channel

Pipe and filter

Layered

Client-server

Model-View-Controller (MVC)

Microservices

Email service like Outlook or Gmail

- Pipe and filter
- Layered
- Client-server
- Model-View-Controller (MVC)
- Microservices

Online banking service like Bank of America

- Pipe and filter
- Layered
- Client-server
- Model-View-Controller (MVC)
- Microservices

Online multi-faceted store like Amazon

Pipe and filter

Layered

Client-server

Model-View-Controller

Microservices

Continuous integration tool (build>test>commit)

Pipe and filter

Layered

Client-server

Model-View-Controller (MVC)

Microservices

What to consider as you design your architecture

As an architect, consider …

Level of Abstraction

- Components (modules) and their interconnections (communication/apis)
- Decompose to a level that allow you to reason about the software system

Separation of concerns

- High cohesion – tight relationships within a component (module)
- Loose coupling – interconnections between components (module)

Modularity

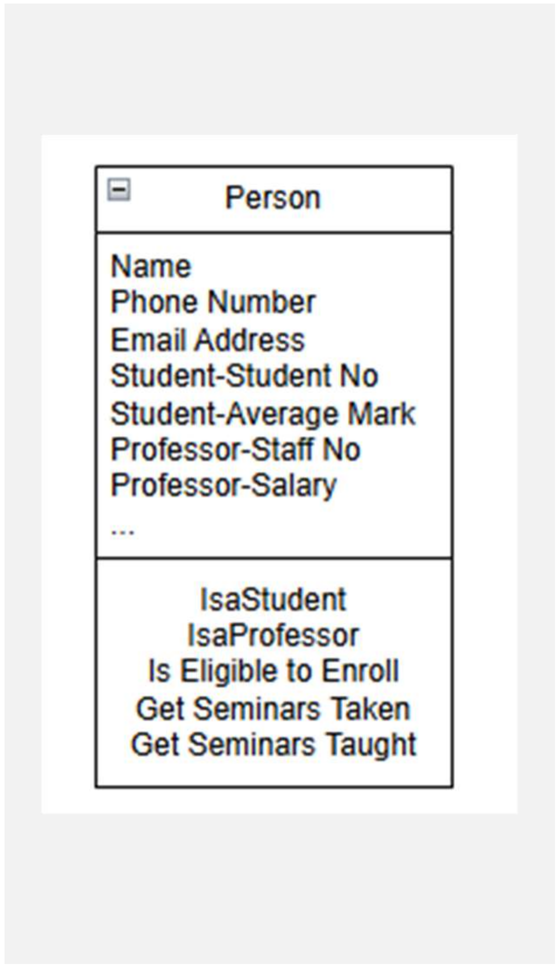
- Decomposable designs (divide and conquer), composable components
- Localized changes (due to requirement changes)
- Span of impact (how far can an error spread)

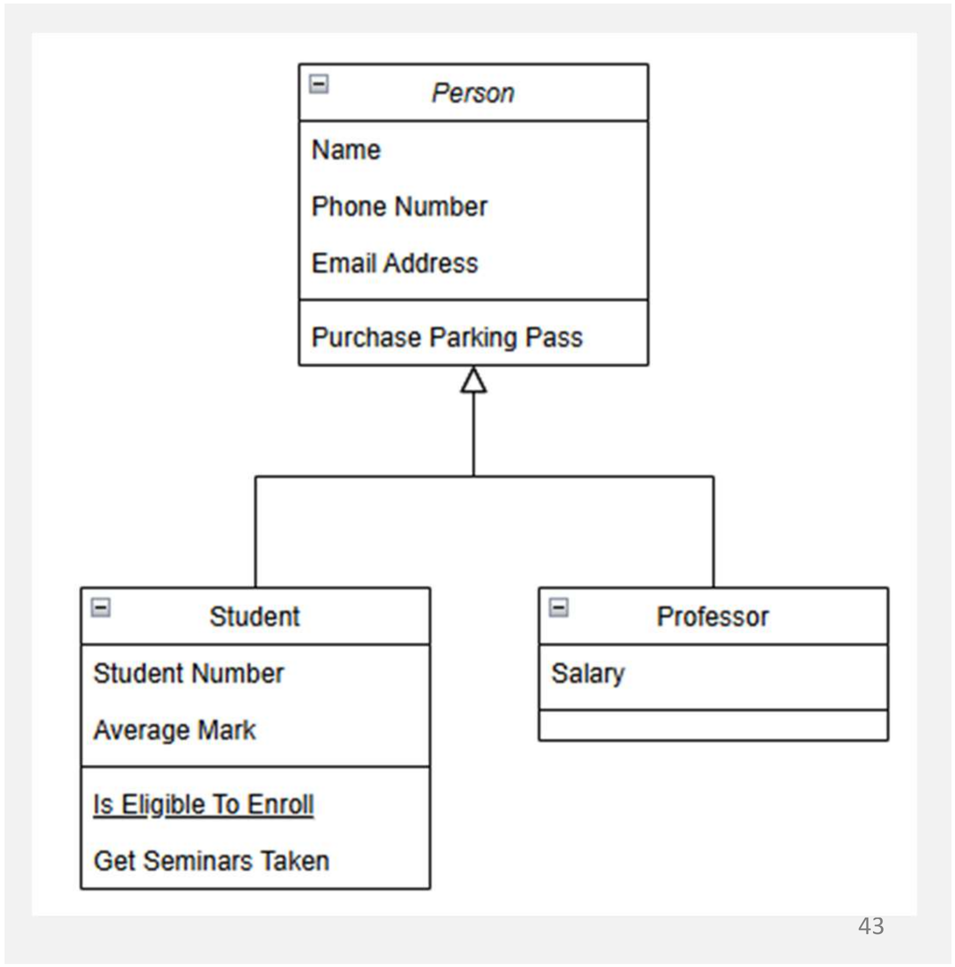
High cohesion (strong cohesion)

Cohesion: how closely the operations in a module are related and belong together

- High cohesion means a component of a system has a clear purpose and scope, and only does one thing well
- Strong relationships within a module improve clarity and understanding
- A module with good abstraction usually has strong internal cohesion

Which would you rather work with?



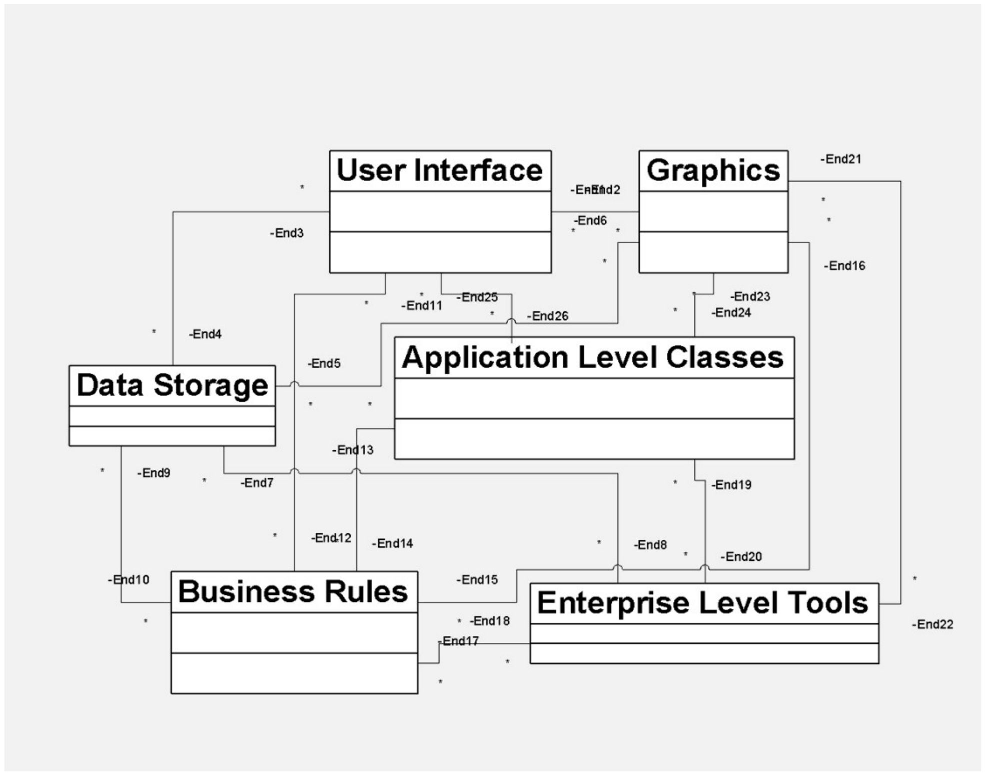


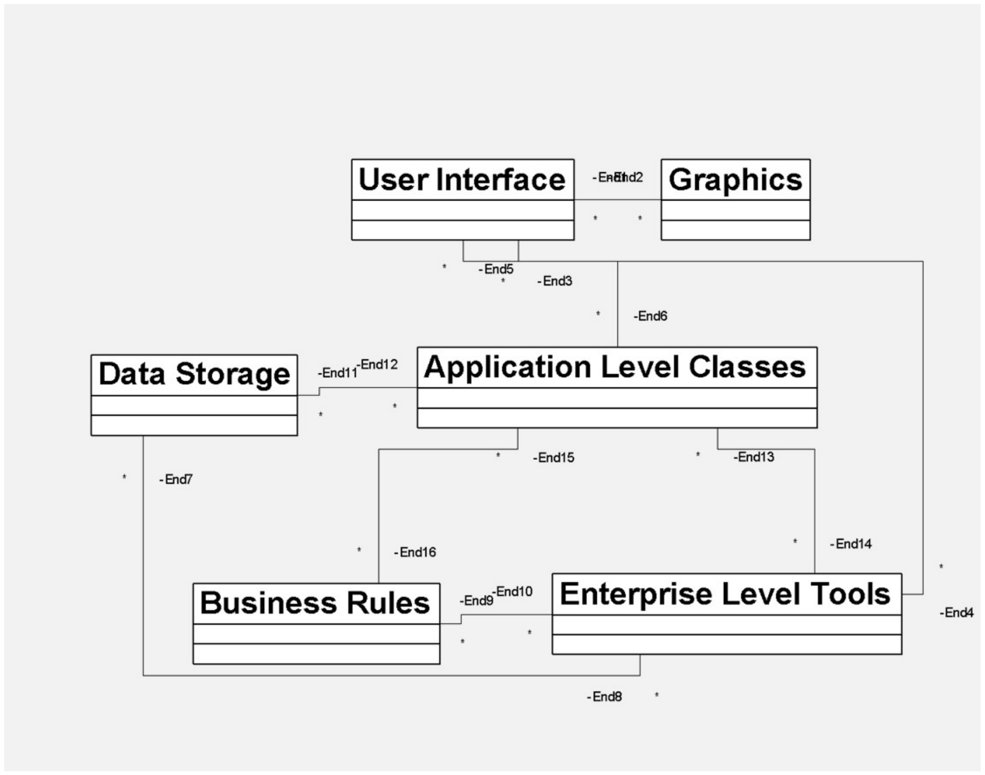
Loose coupling

Coupling: the kind and quantity of interconnections among modules

- Modules that are **loosely coupled** (or uncoupled) are easier to work with
- The more **tightly coupled** two modules are, the harder it is to work with them separately (consider development, testing, …)

Which would you rather work with?





As an architect, consider …

- **System understanding**: interactions between modules
- **Reuse**: high-level view shows opportunity for reuse
- **Construction**: breaks development down into work items; provides a path from requirements to code
- **Evolution**: high-level view shows evolution path
- **Management:** helps understand work items and track progress
- **Communication**: provides vocabulary; a picture says 1000 words

As an architect, don’t lose sight of …

- Satisfying functional and performance **requirements**
- Managing **complexity**
- Accommodating **change**

Summary

Architecture

The set of structures needed to reason about a software system

- An architecture provides a high-level framework, a blueprint, to build and evolve a software system
- Strive for modularity: high cohesion and loose coupling
- Learn from existing architectural styles and patterns

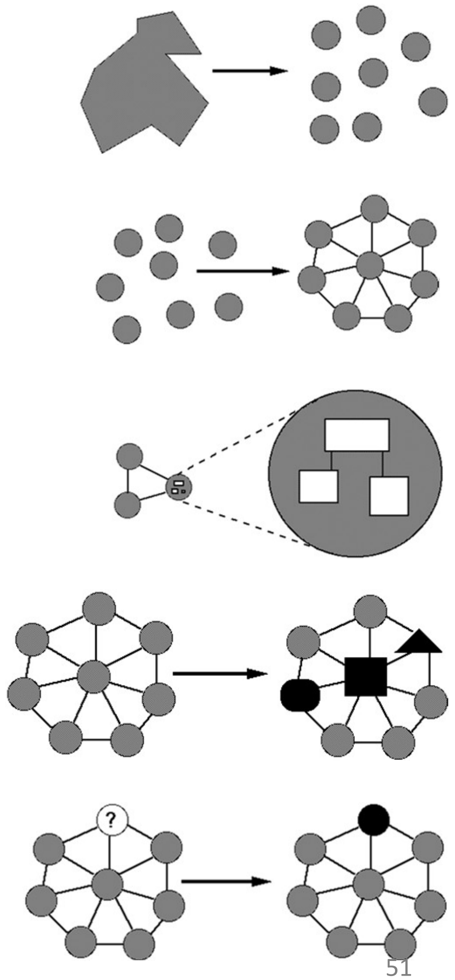
Additional Material

Divide and conquer

- Benefits of decomposition:
  - Decrease size of tasks
  - Support independent testing and analysis
  - Separate work assignments
  - Ease understanding
- Use of abstraction leads to modularity
  - Implementation techniques: information hiding, interfaces
- To achieve modularity, you need:
  - Strong cohesion within a component
  - Loose coupling between components
  - And these properties should be true at each level

Qualities of modular software

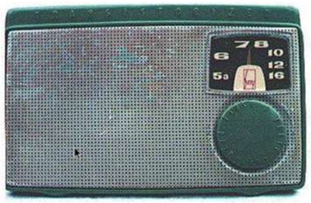
- Decomposable
  - can be broken down into pieces
- Composable
  - pieces are useful and can be combined
- Understandable
  - one piece can be examined in isolation
- Has continuity
  - change in reqs affects few modules
- Protected / safe
  - an error affects few other modules



Interface and implementation

- **public interface**: data and behavior of the object that can be seen and executed externally by "client" code
- **private implementation**: internal data and methods in the object, used to help implement the public interface, but cannot be directly accessed
- **client**: code that uses your class/subsystem

Example: *radio*



  - public interface is the speaker, volume buttons, station dial
  - private implementation is the guts of the radio; the transistors, capacitors, voltage readings, frequencies, etc. that user should not see