Architecture and Design

CSE 403 Software Engineering

Today's Outline

Architecture

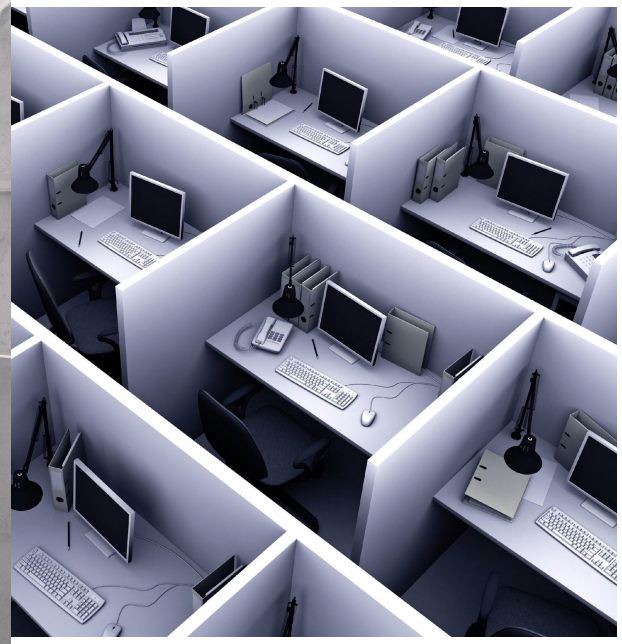
1. What is architecture
2. How does it differ from design
3. What are some common architectures used in software

What does “Architecture” make you think of?

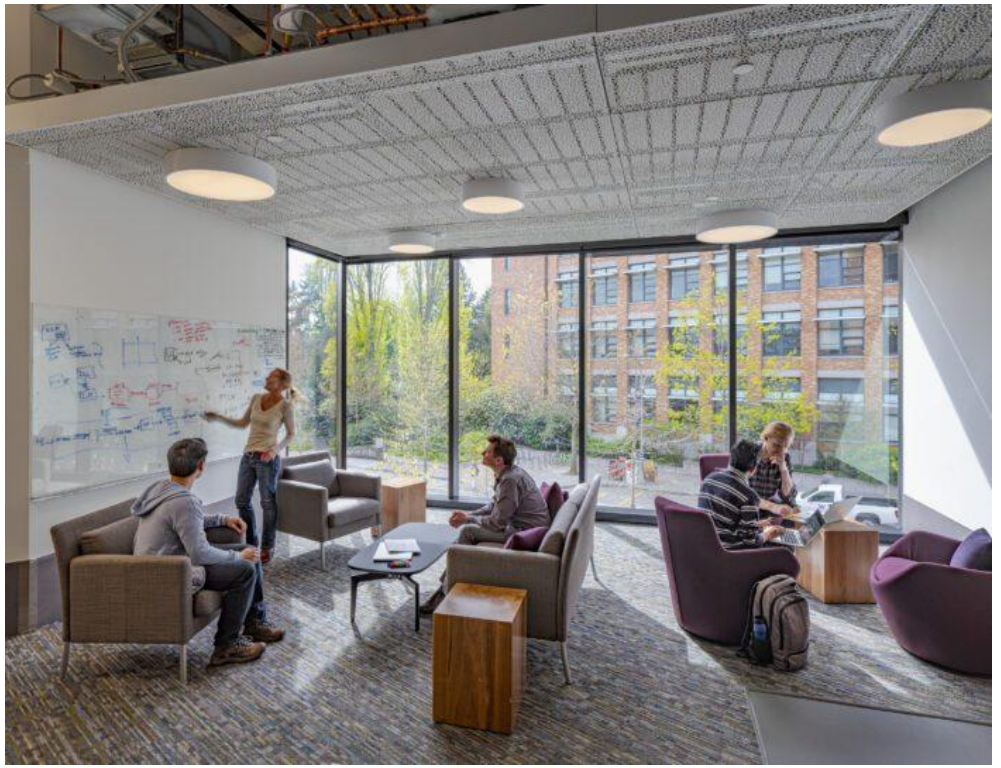
MIT Stata Center by Frank Gehry

Paul G. Allen Center by LMN Architects

In contrast, what comes to mind for “Design”?

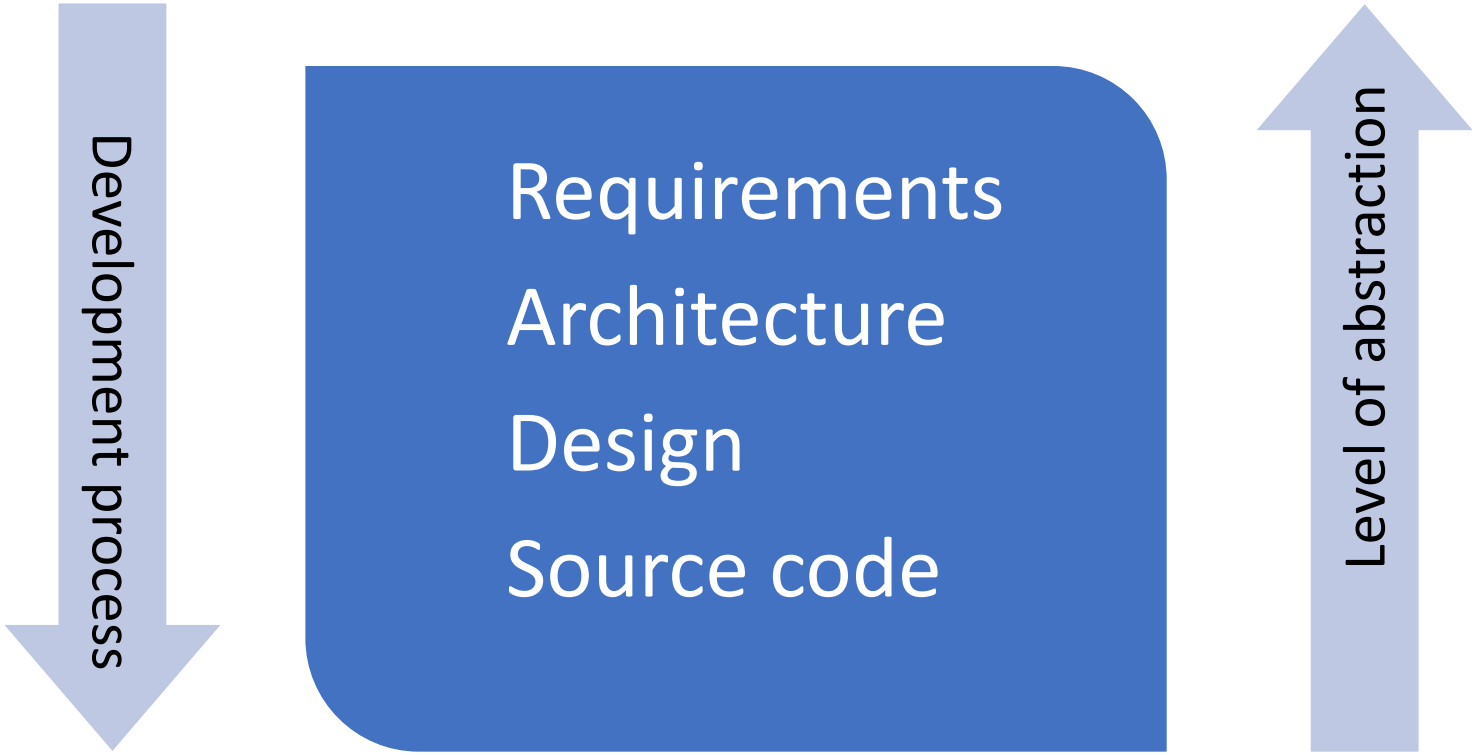


Here’s another example close to home



Bill & Melinda Gates Center for UW CSE - LMN

Where do architecture and design fit in?



Definitions

Architecture (what components are needed)

- High-level view of the overall system:
  - What **components** exist?
  - What are the **connections** and/or protocols between components?

Design (how the components are developed)

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

The level of abstraction is key

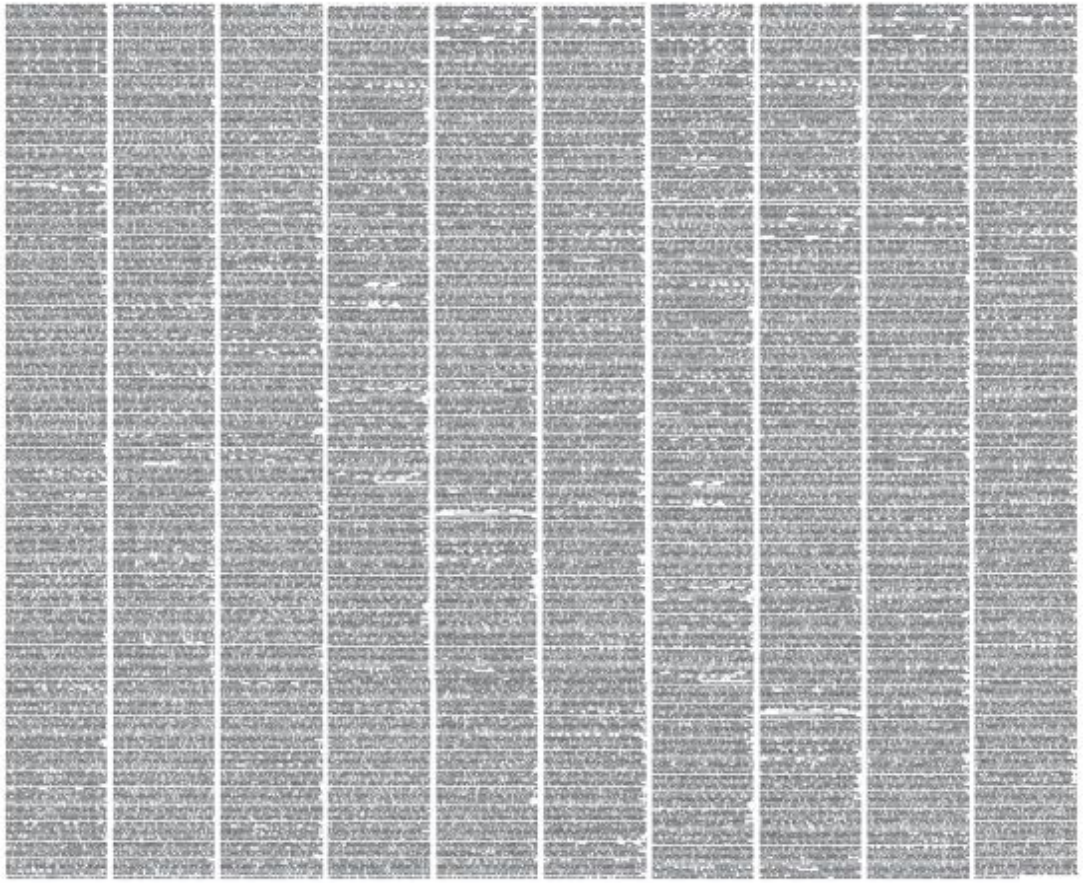
Both architecture and design build an

abstract representation of reality

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

Case study – Linux kernel

Source code

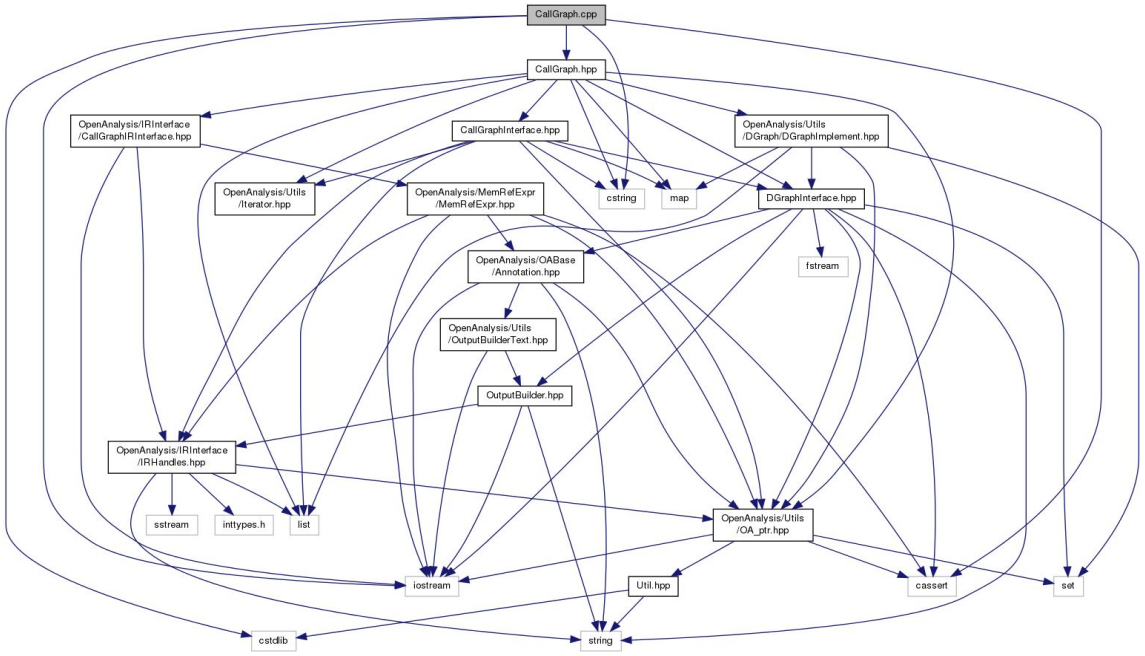


Suppose you want to add a feature
40 million lines of code!
Where would you start?

- **What questions would you ask?**

Case study – Linux kernel

Call graph



Suppose you want to add a feature

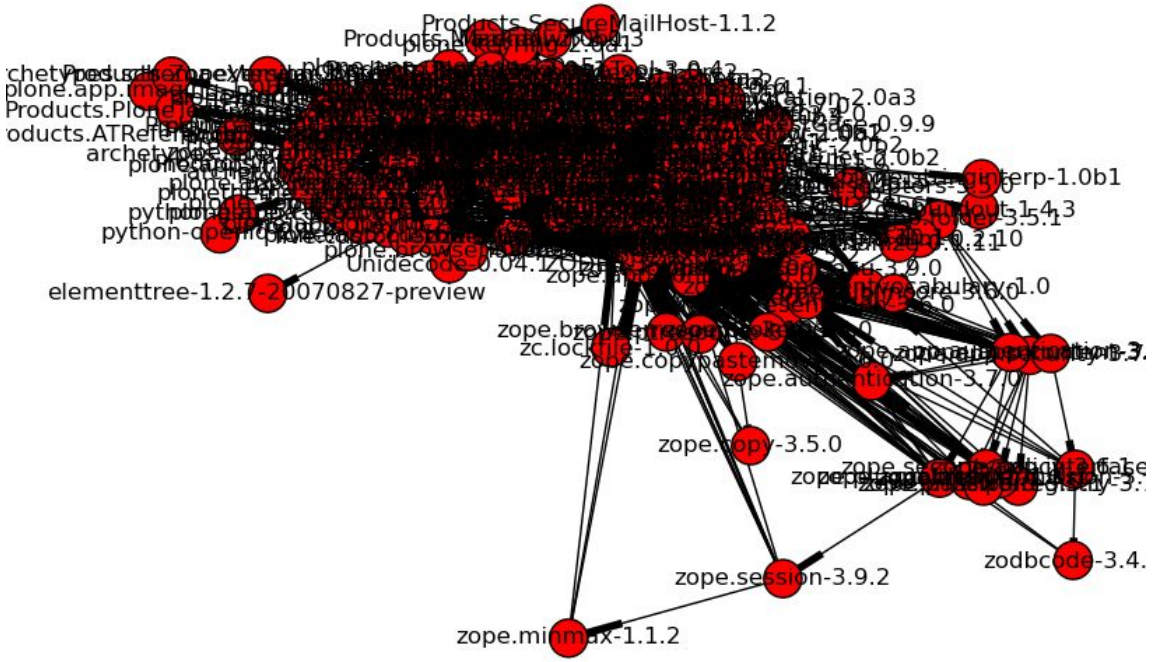
40 million lines of code!

Where would you start?

- **What is the flow of control?**

Case study – Linux kernel

Dependency graph



Suppose you want to add a feature

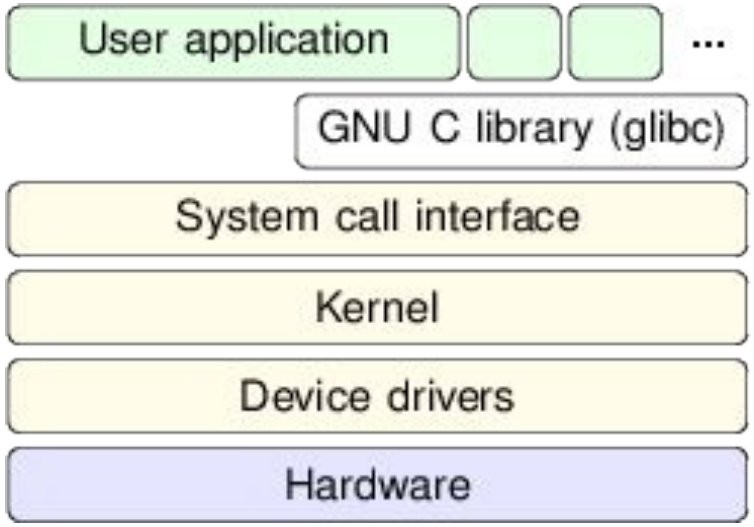
40 million lines of code!

Where would you start?

- What is the flow of control?
- **What components know about one another?**

Case study – Linux kernel

Layer diagram



Suppose you want to add a feature

40 million lines of code!

Where would you start?

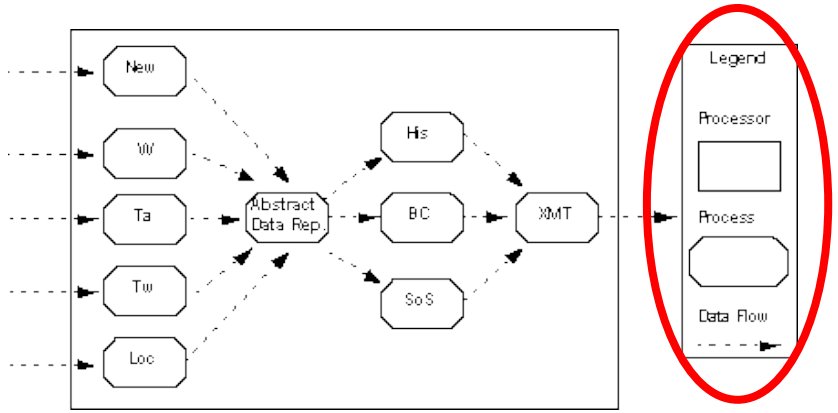
- What is the flow of control?
- What components know about one another?
- **How is the code organized into parts?**

The parts of an architecture

- Components
- Connections

How would you represent them in a diagram?

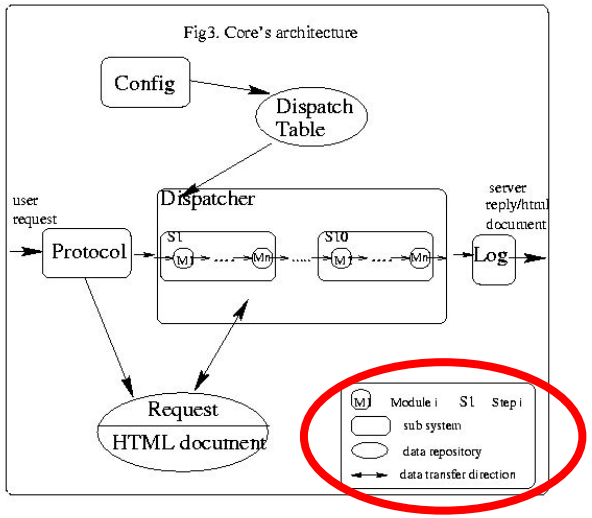
Box-and-arrow diagrams

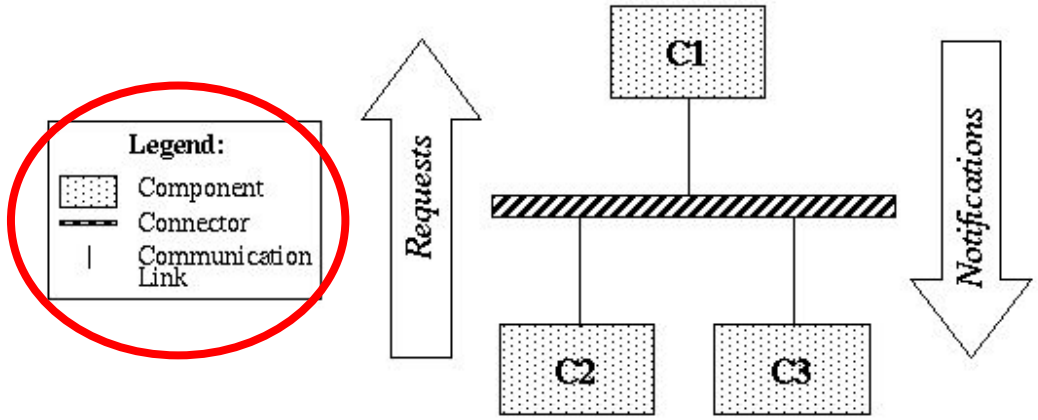


Very common and highly valuable.

You must **define** the parts of the diagram:

- Box
- Arrow
- Layer
- Adjacent boxes

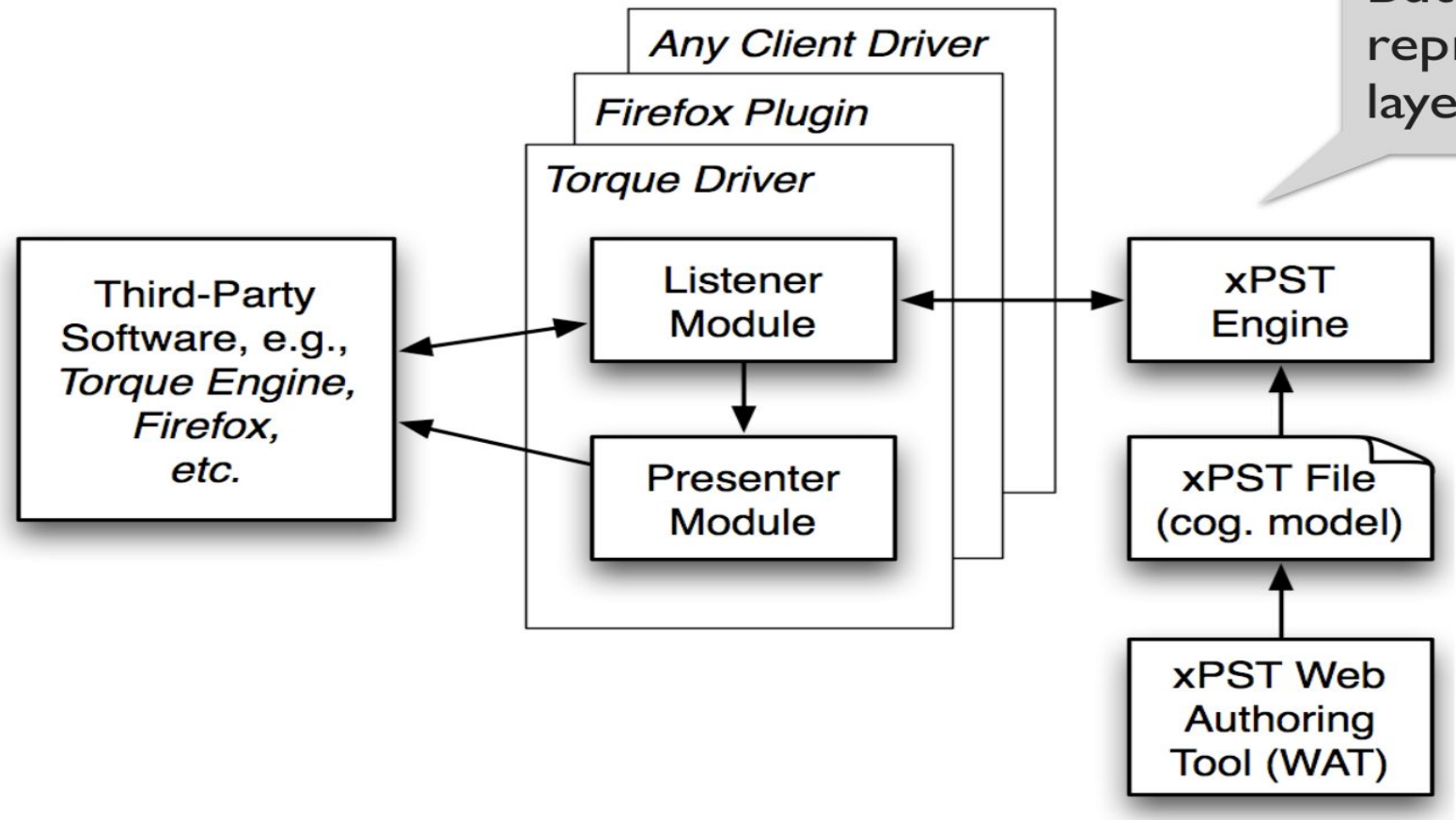




Anoher box and arrow diagram

Very common and highly valuable.

But what does a box represent? An arrow? A layer? Adjacent boxes?



An architecture: components and connectors

- ***Components*** have behavior and perform computations
  - abstract data type, filter, database, web browser, etc.
- ***Connectors*** define the interconnections between components
  - procedure call, event announcement,
    asynchronous message sends, socket/file write/read, etc.
- They may sometimes share behavior
  - Ex: A connector might (de)serialize data, but can it perform other, richer computations?

UML diagrams

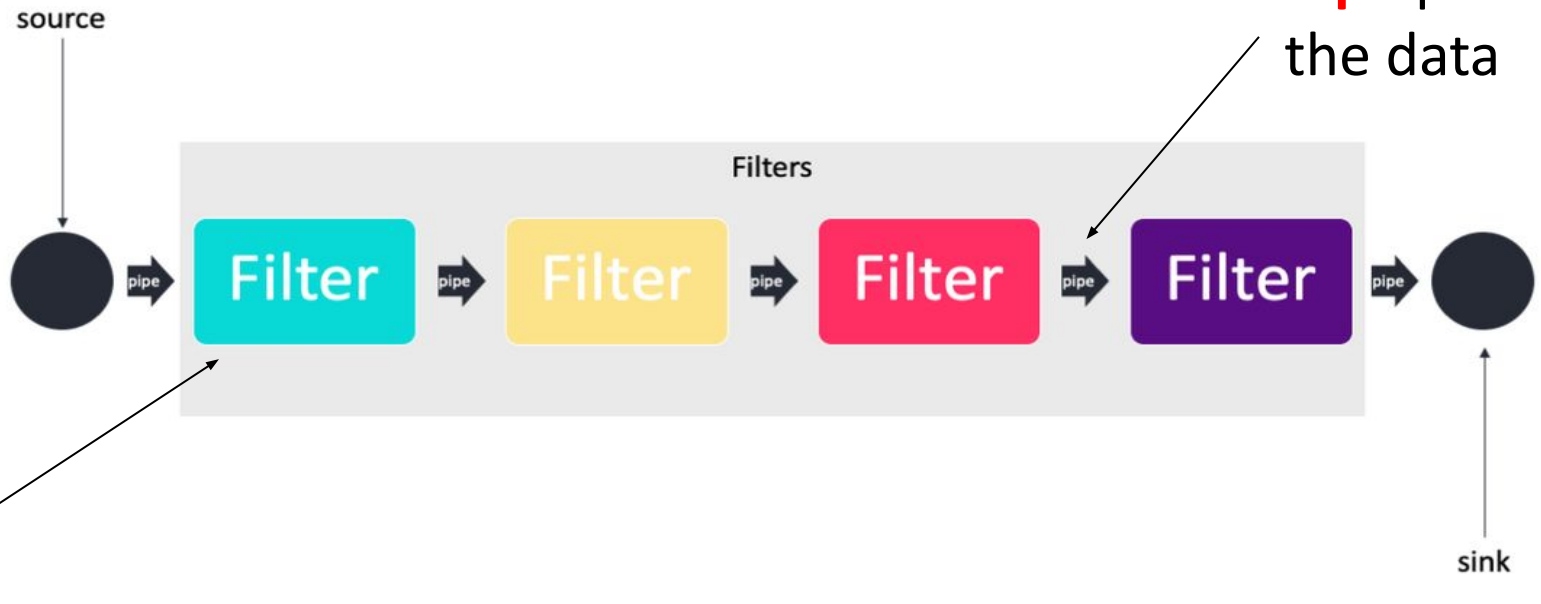
- UML = universal modeling language
- A standardized way to describe (draw) architecture
  - Also implementation details such as subclassing, uses (dependences), and much more
- Widely used in industry
- Not the topic of this lecture
- Compare to design patterns

- Critical advice about syntax:
  - Use consistent notation: one notation per kind of component or connector

Examples of software architectures

SW Architecture #1 – Pipe and filter

An architecture determines the structure of the **components** and how they **connect**.



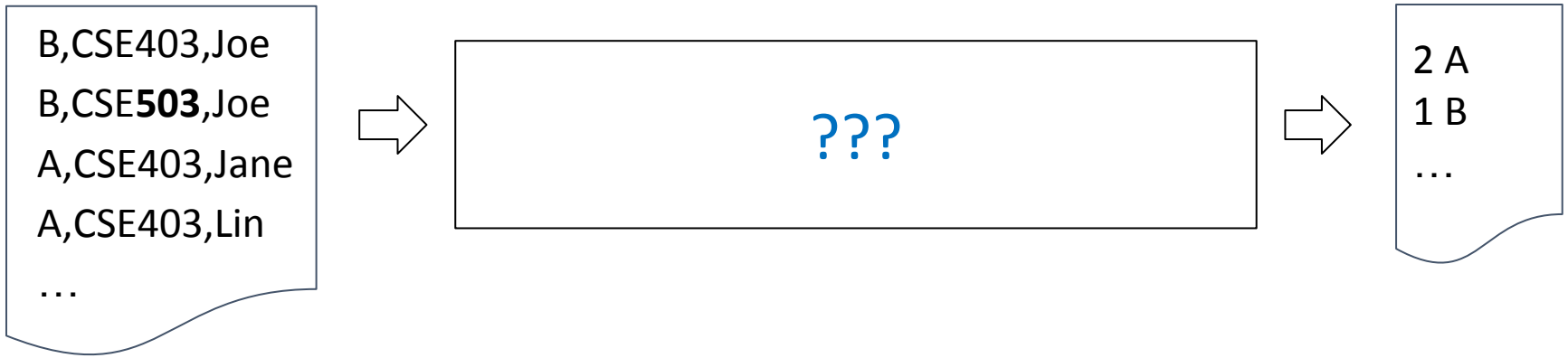
Pipe passes the data

Filter computes on the data

- It doesn't specify the design or implementation details of the individual components (the filters)
- What about the format of the pipe data?

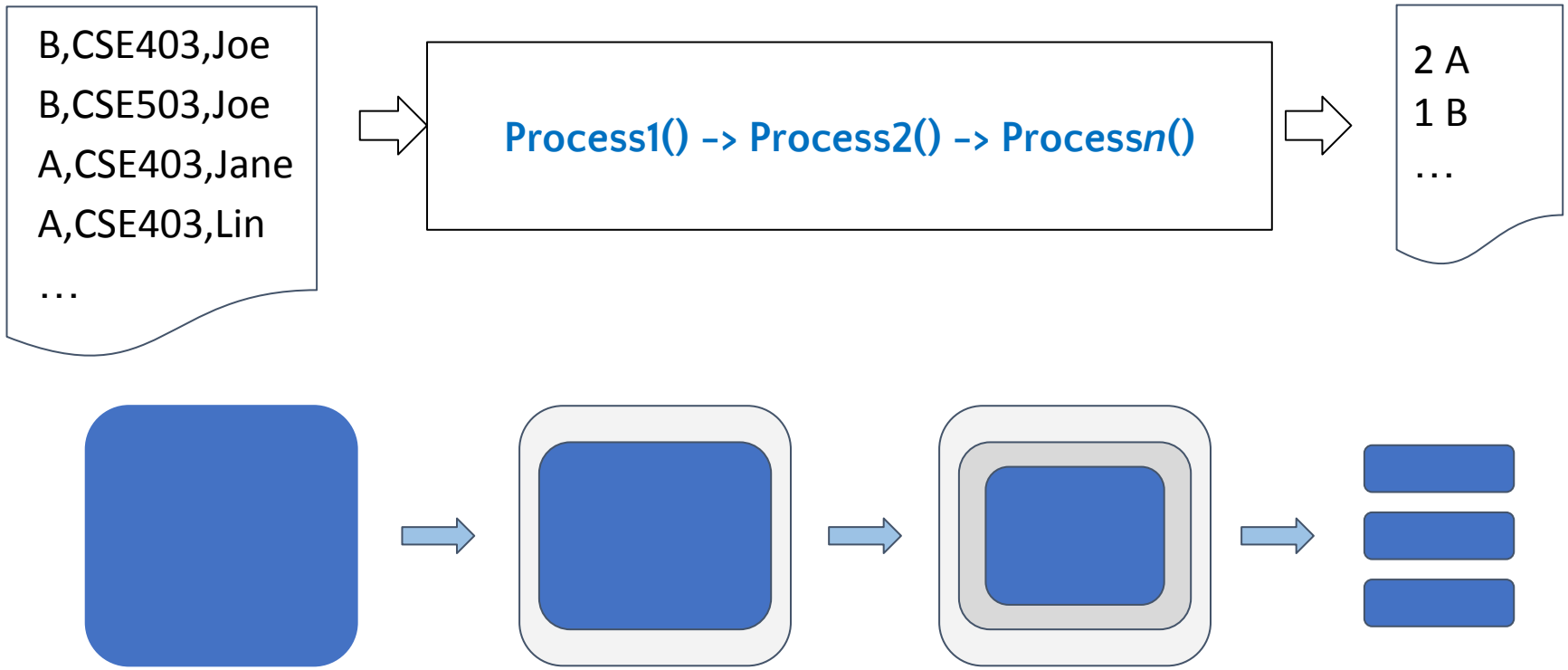
Pipe and filter – let's try it

Goal: a histogram of the CSE 403 letter grades



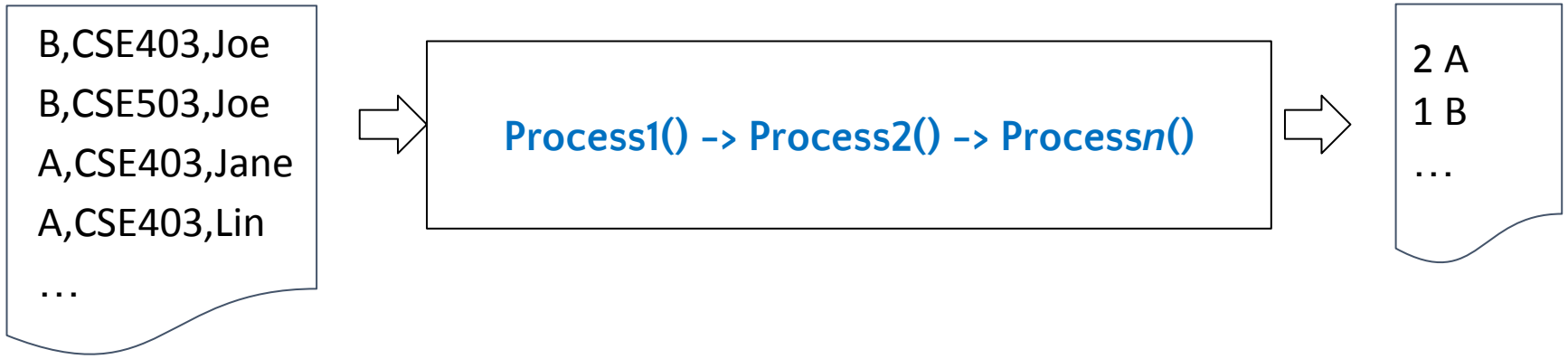
SW Architecture #1 – Pipe and filter

The **architecture** consists of **components** and **successive filtering.**



SW Architecture #1 – Pipe and filter

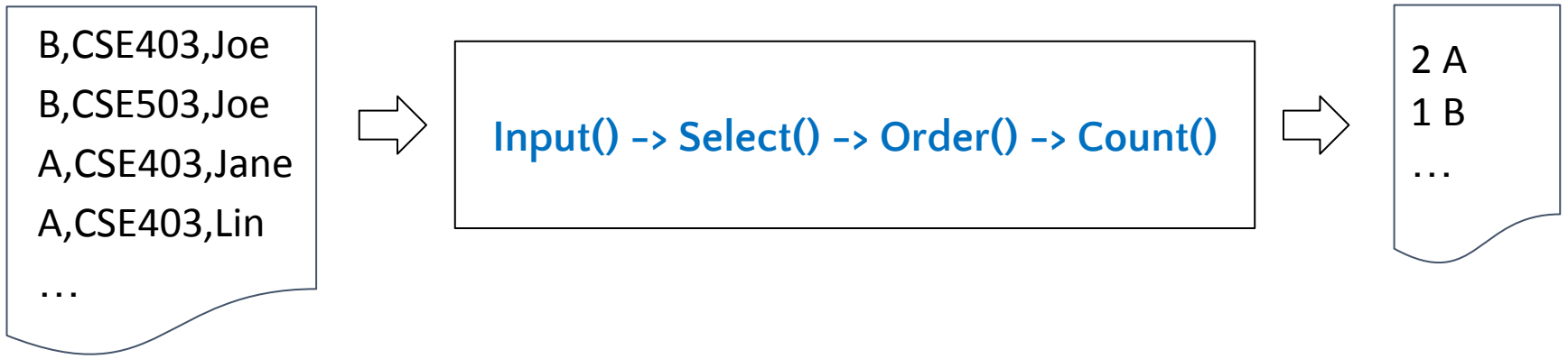
The **architecture** consists of **components** and **successive filtering**.

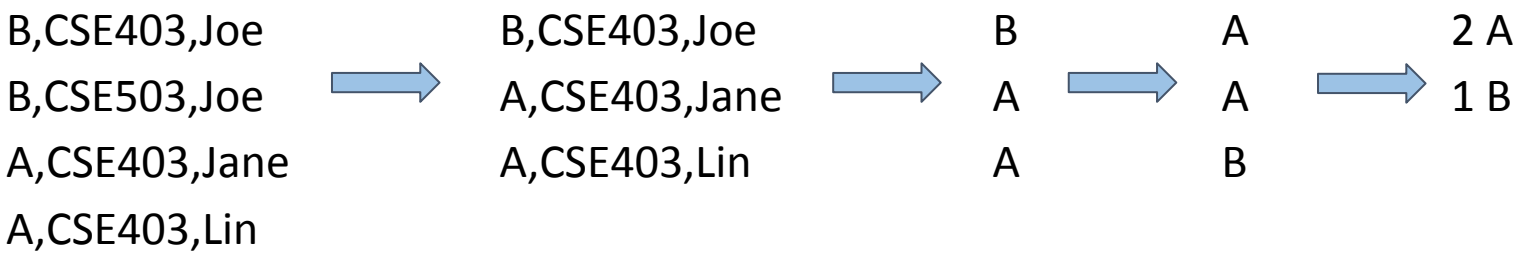


Let's design a Unix pipeline to perform this task.

SW Architecture #1 – Pipe and filter

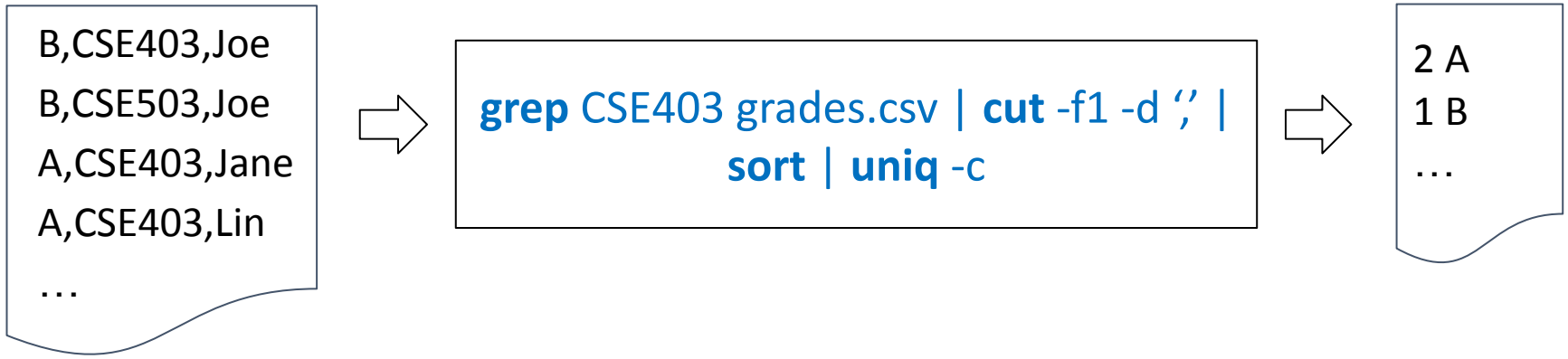
The **design** specifies the **components' inputs and outputs.**

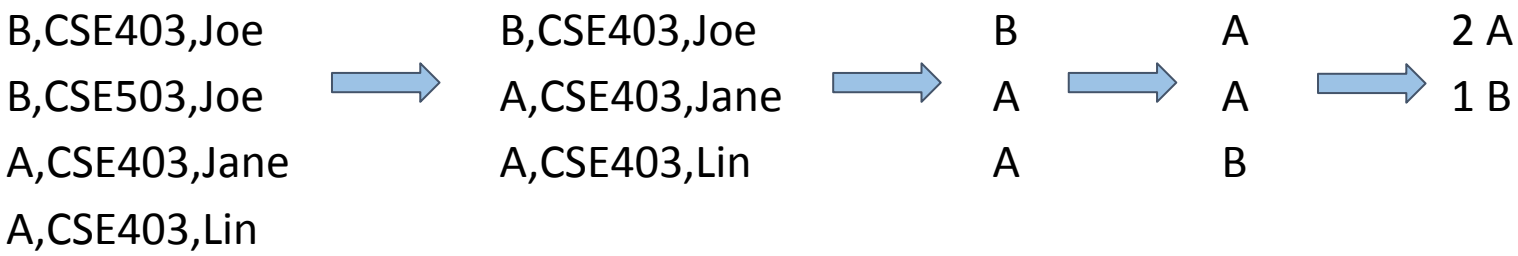




SW Architecture #1 – Pipe and filter

Finally, you get to **code**

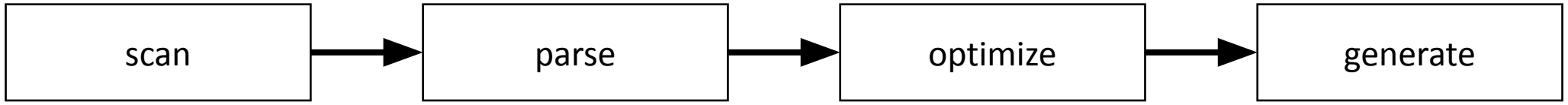




What are the pros and cons of this design?

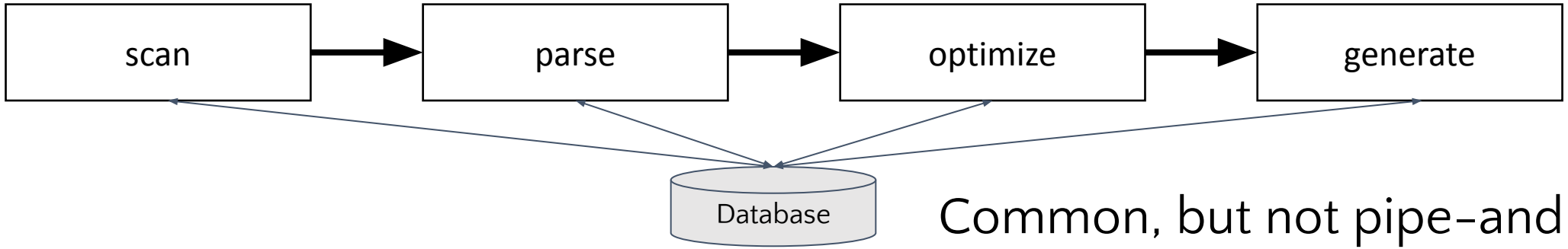
An architectural style imposes constraints

- Pipes & filters
  - Pipes must compute **local transformations**
  - Filters must **not share state** with other filters
  - There must be **no cycles**
- If these constraints are violated, it's not a pipe & filter system
  - One can't tell all of this from a picture
  - One can formalize these constraints



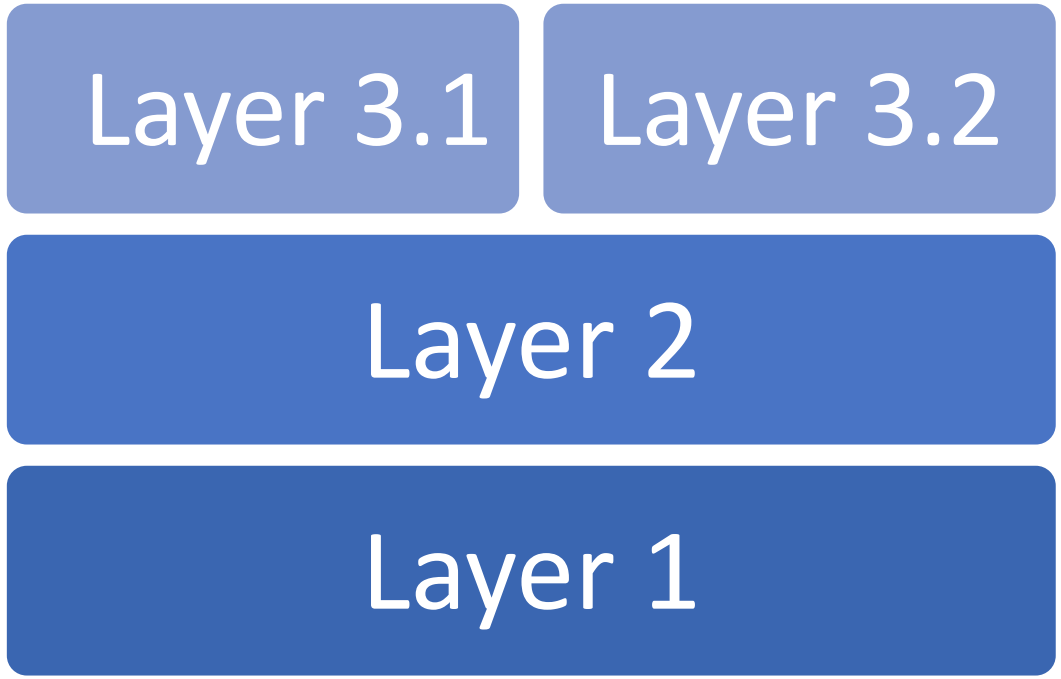
An architectural style imposes constraints

- Pipes & filters
  - Pipes must compute **local transformations**
  - Filters must **not share state** with other filters
  - There must be **no cycles**
- If these constraints are violated, it's not a pipe & filter system
  - One can't tell all of this from a picture
  - One can formalize these constraints



Common, but not pipe-and-filter

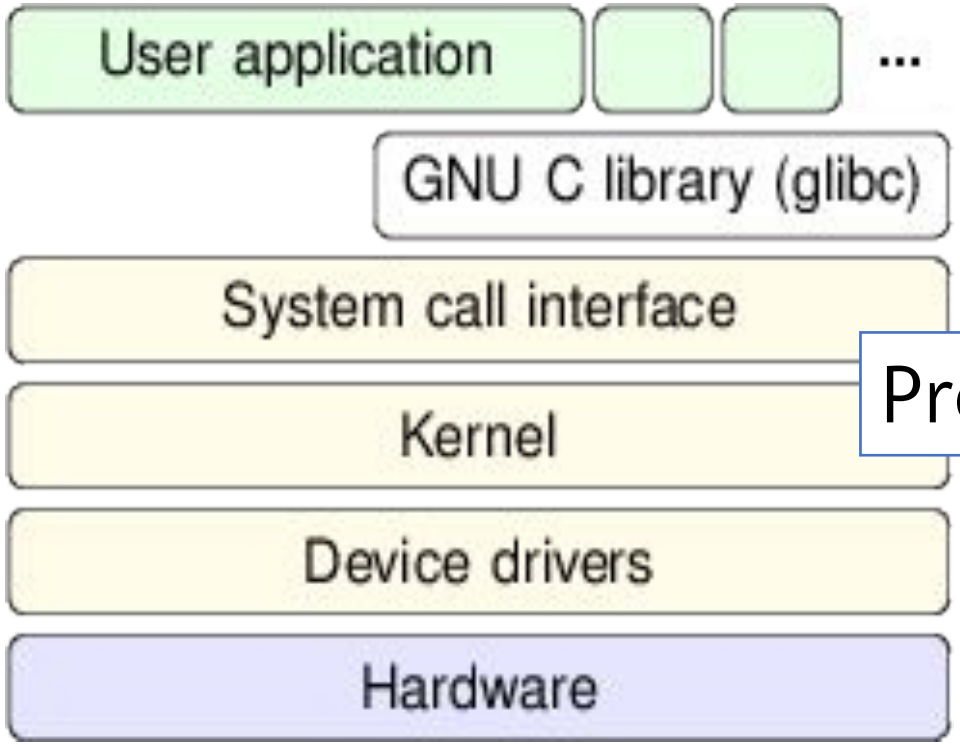
SW Architecture #2 – Layered (n-tier)



- Layers use services provided (only) by the layers directly below them
- Layers of isolation – limits dependencies
- Good modularity and separation of concerns

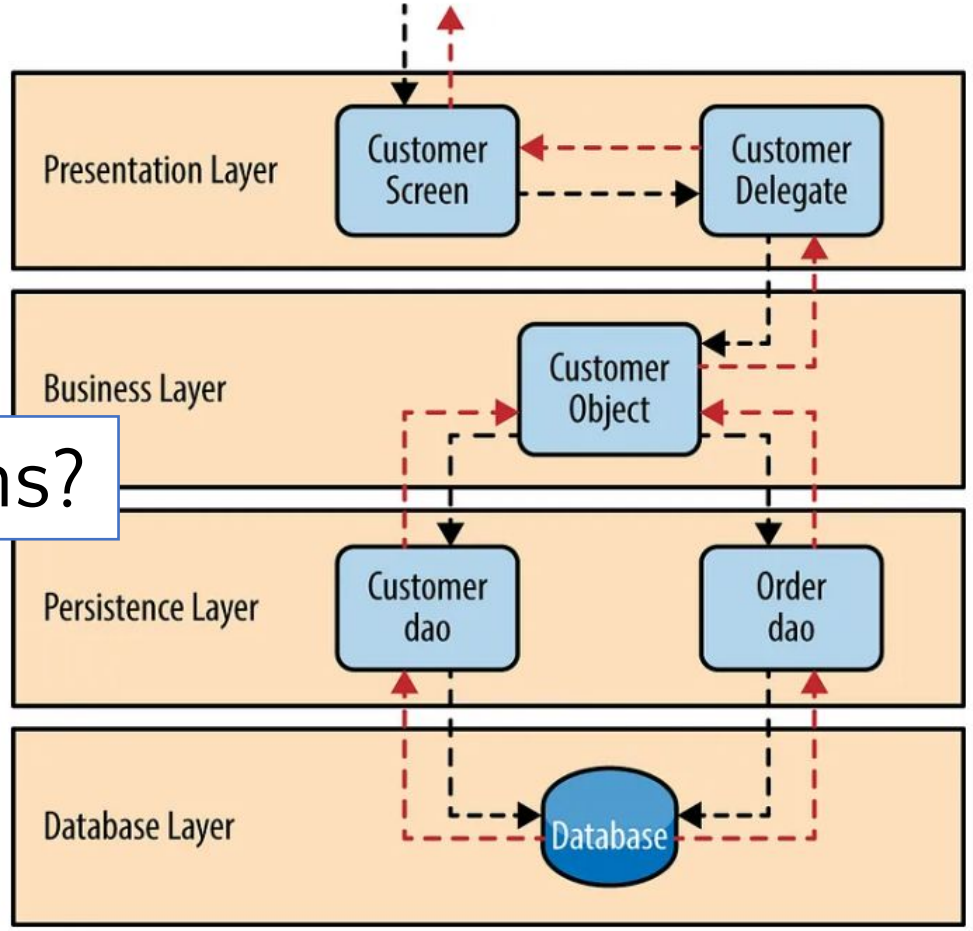
SW Architecture #2 – Layered

Linux Architecture



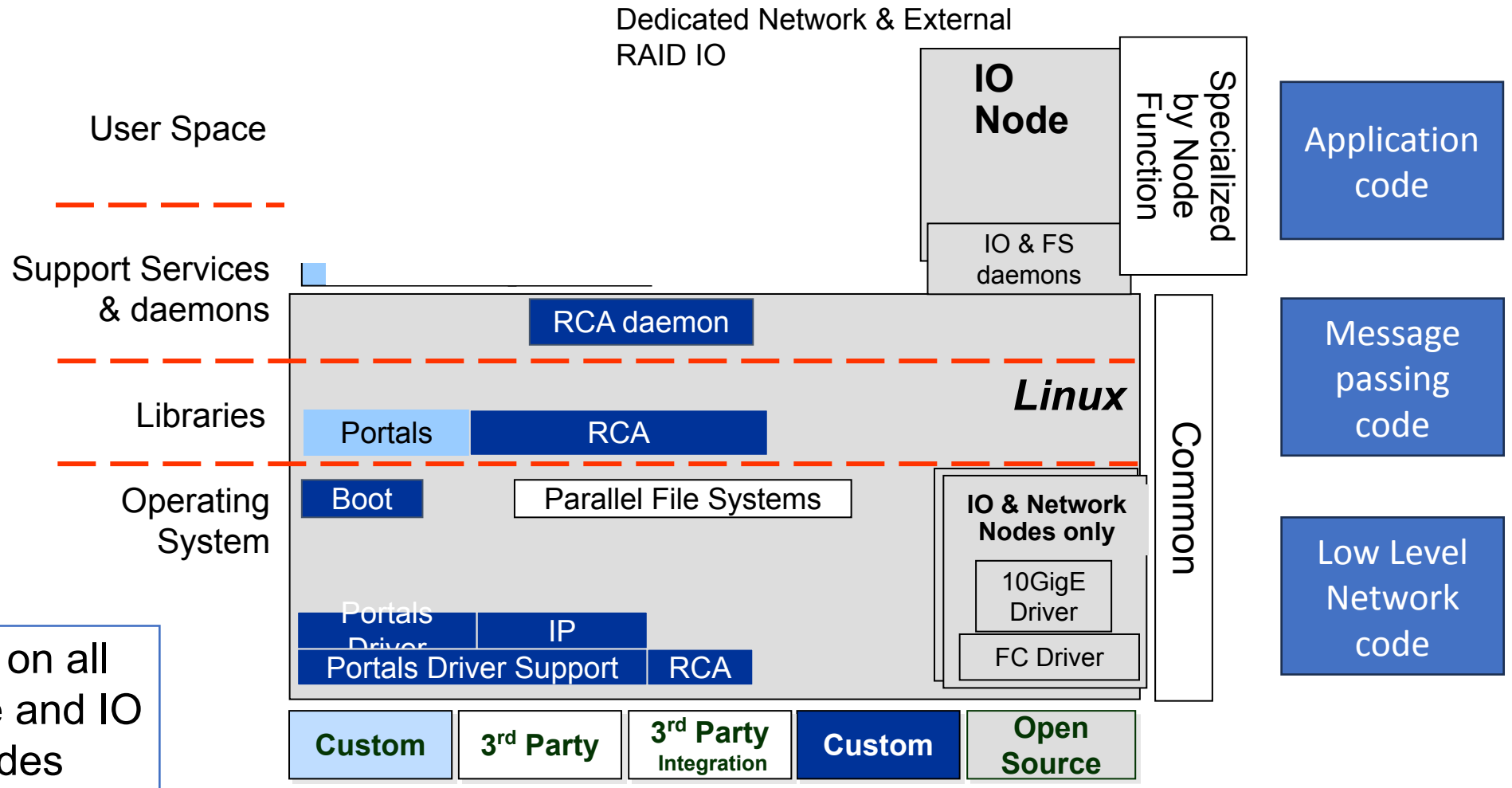
Pros / cons?

Enterprise System Architecture



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

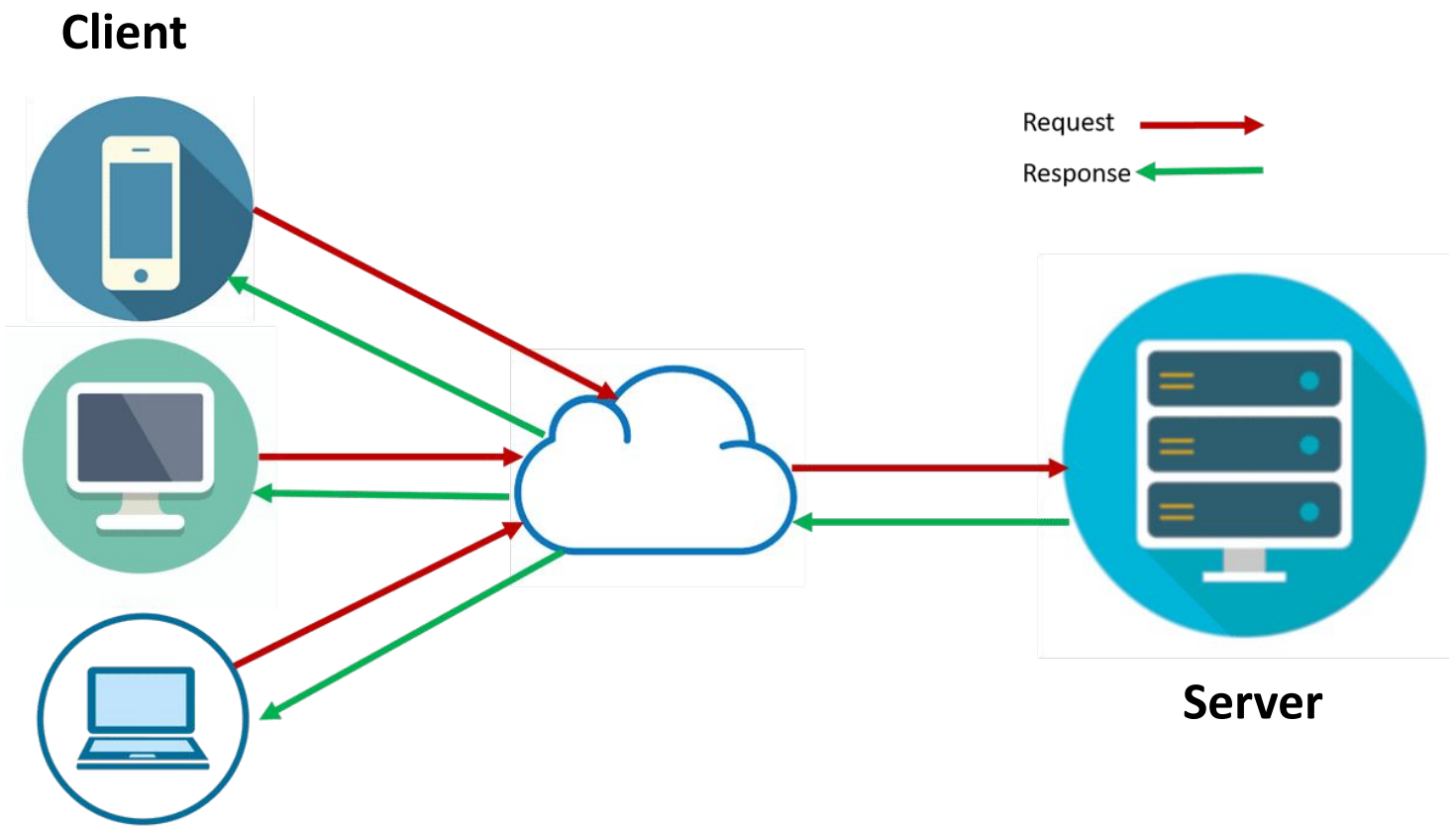
SW Architecture #2 – Layered



Runs on all Service and IO Nodes

SW Architecture #3 – Client-Server

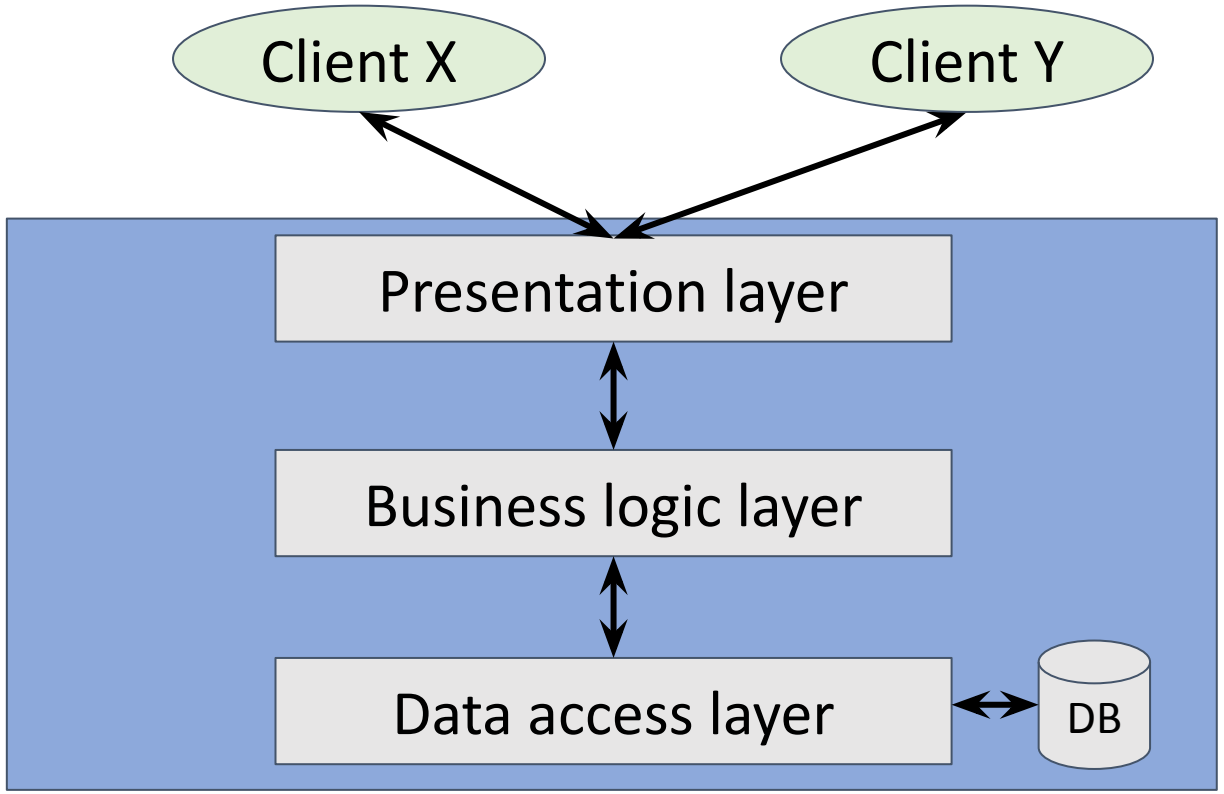
What are pros and cons? How can you avoid the cons?



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

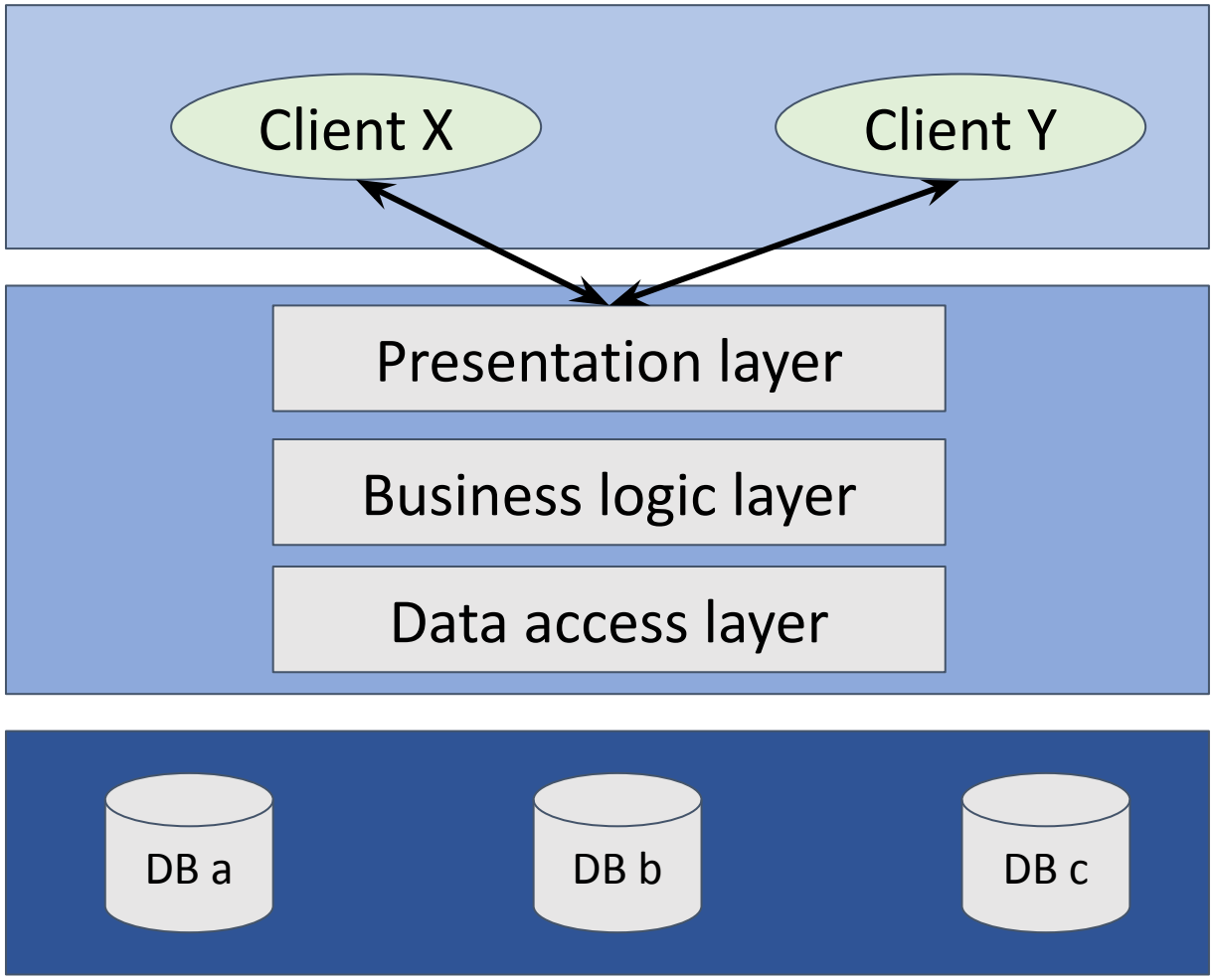
SW Architecture combinations!

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



SW Architecture combinationsˆ2

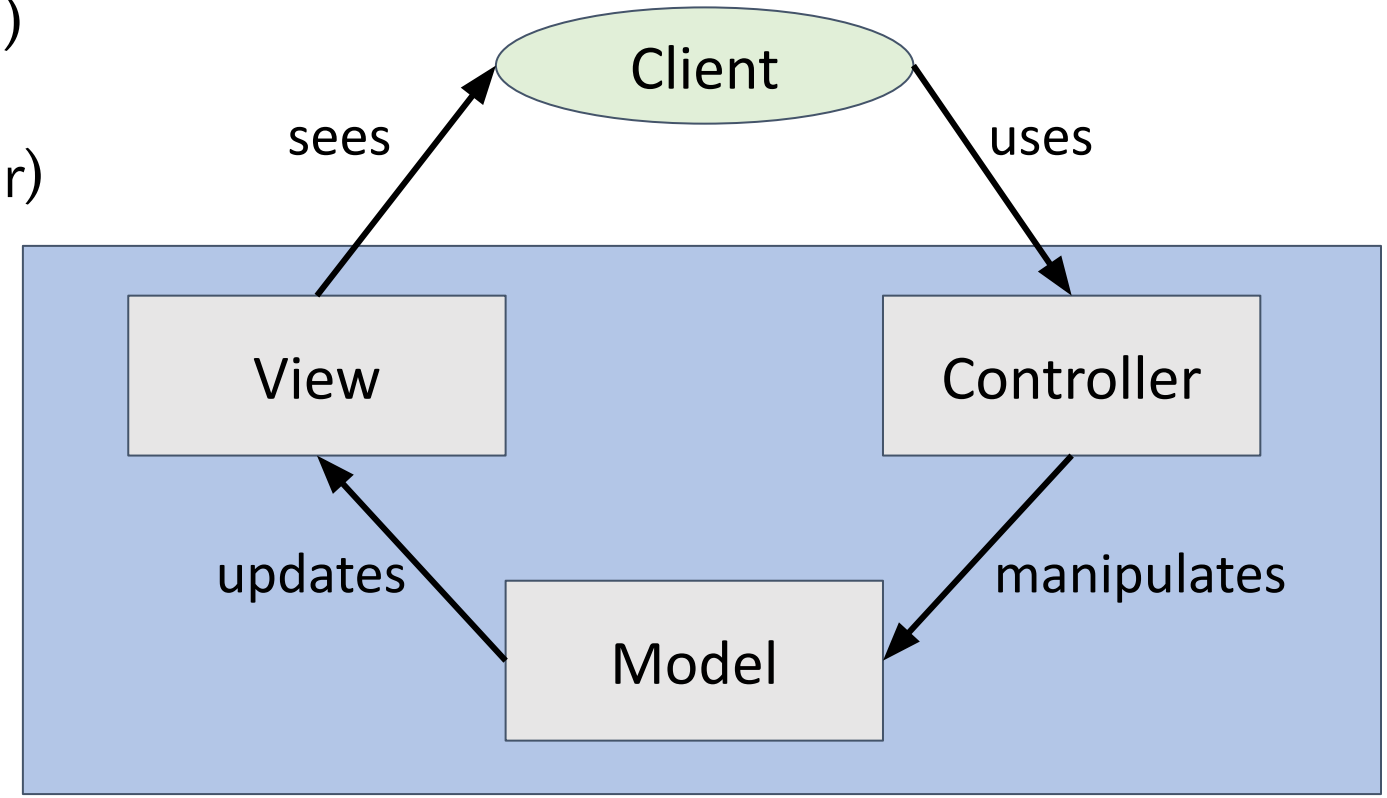
How detailed should an architecture description be?



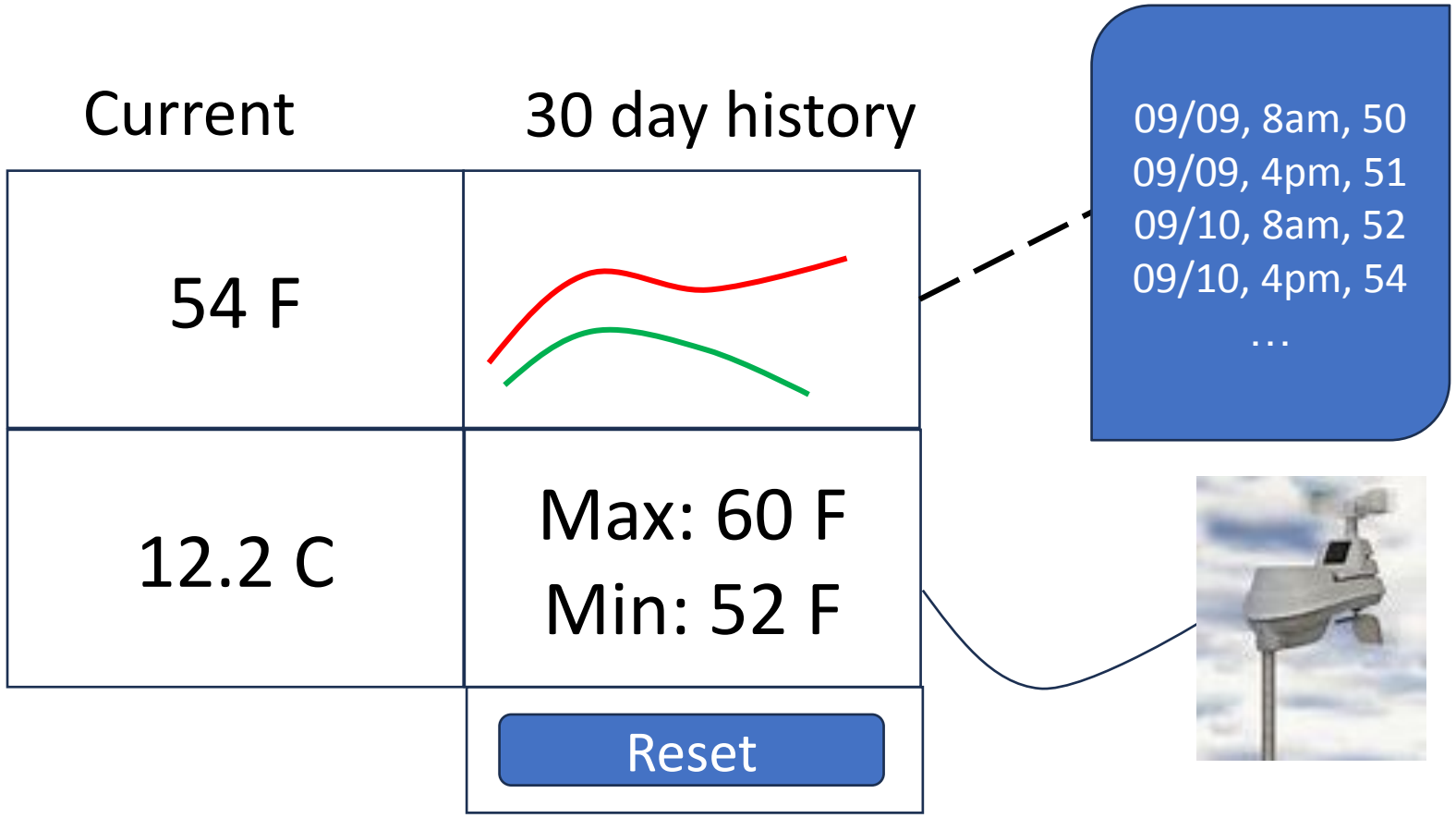
SW Architecture #4 – Model-View-Controller (MVC)

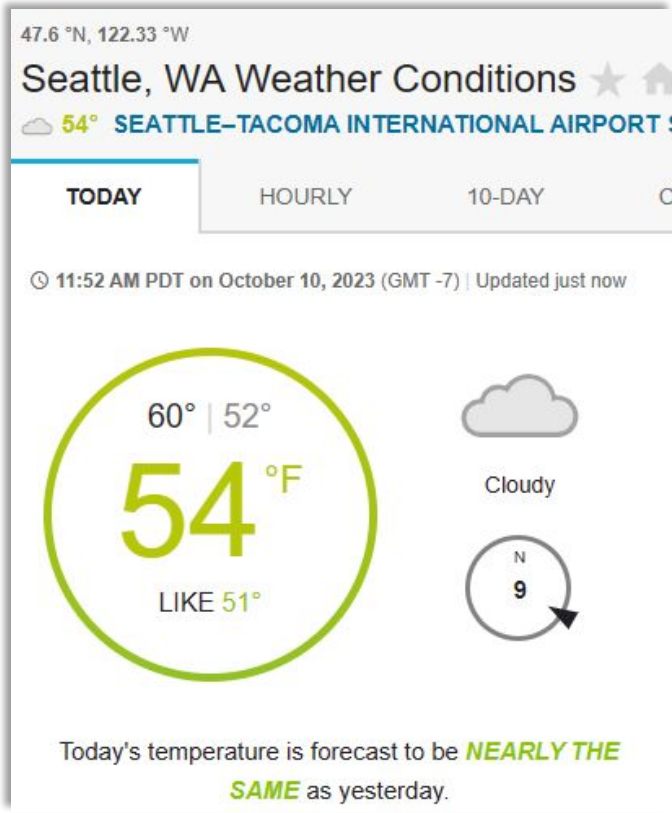
Separates

- data representation (Model)
- visualization (View)
- client interaction (Controller)

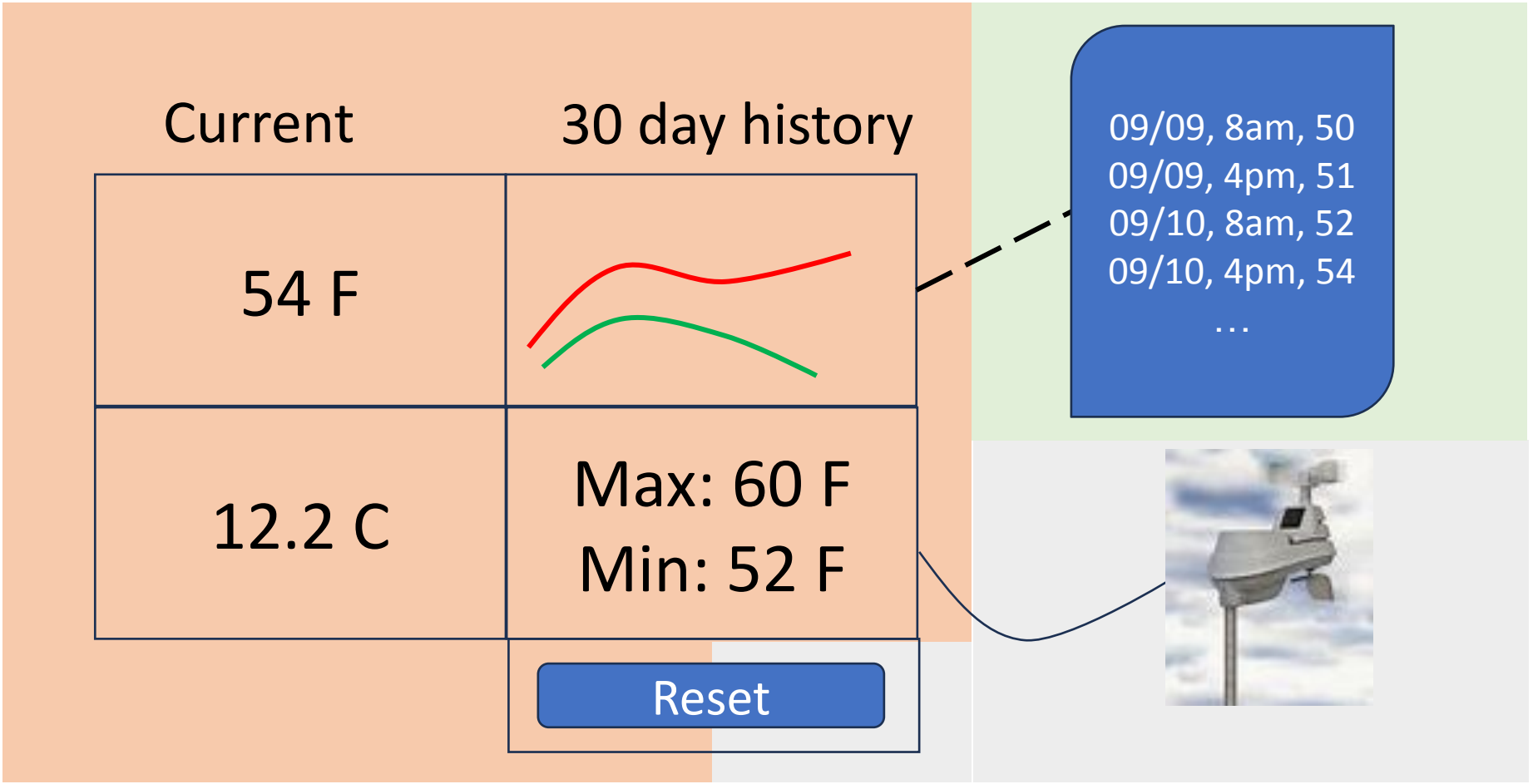


SW Architecture #4 – MVC Example

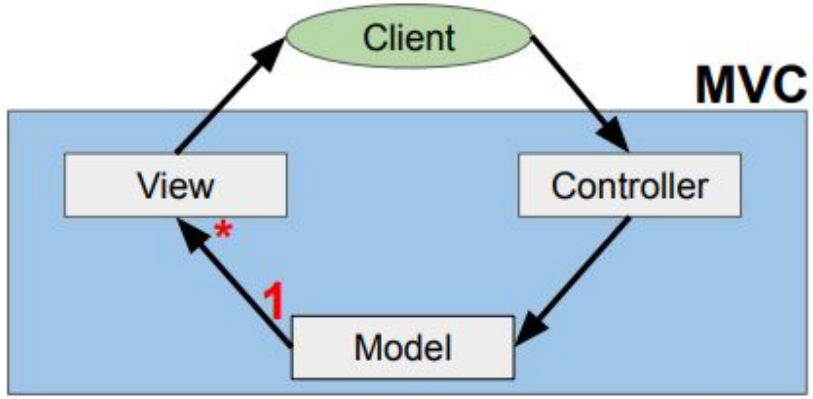


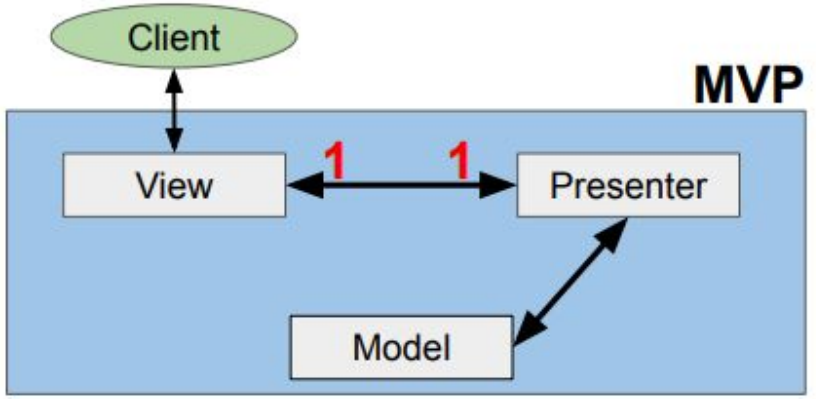


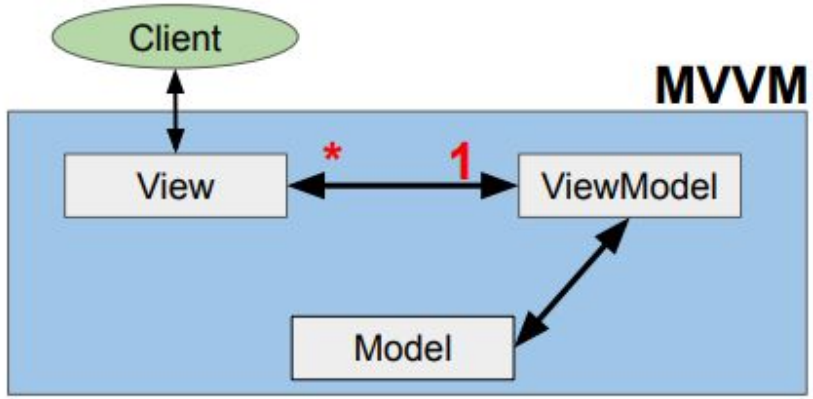
SW Architecture #4 – MVC Example



SW Architecture – many variants of MVC



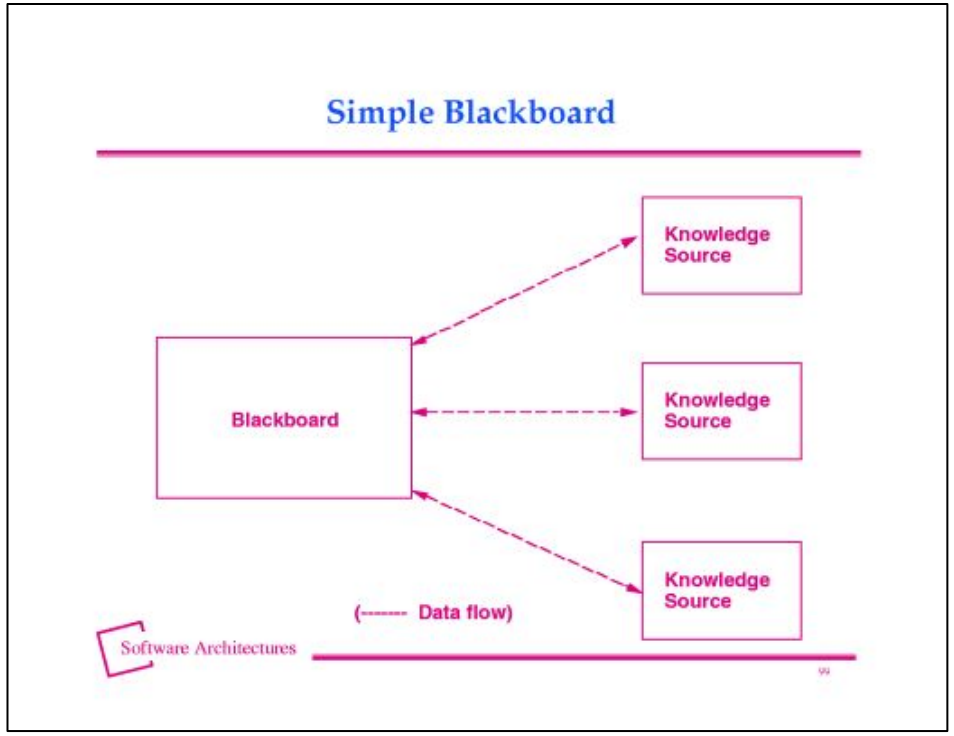




Consider the connections (* == many)

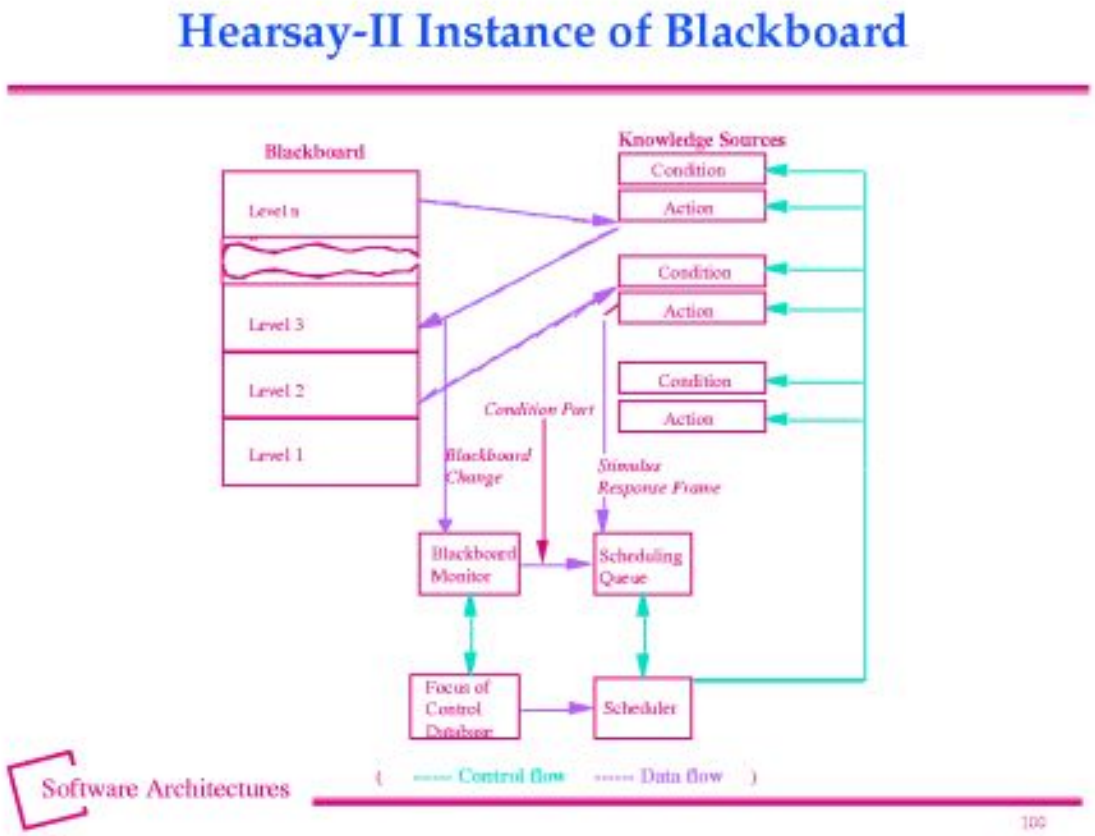
Blackboard architectures

- *The **knowledge sources***: separate, independent units of application dependent knowledge. No direct interaction among knowledge sources
- *The **blackboard** data structure*: problem-solving state data. Knowledge sources make changes to the blackboard that lead incrementally to a solution to the problem.
- ***Control***: driven entirely by state of blackboard. Knowledge sources respond opportunistically to changes in the blackboard.



Useful for applications requiring complex interpretations of data, such as reasoning, speech, and pattern recognition.

Hearsay-II: blackboard



As an architect (and designer), consider …

Level of Abstraction

- Components (modules) and their interconnections (APIs)

Separation of concerns

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

Modularity

- Decomposable designs
- Composable components
- Localized changes (due to requirement changes)
- Span of impact (how far can an error spread)

Properties of a good architecture

- Satisfies functional and performance **requirements**
- Manages **complexity**
- Accommodates future **change**
- Is concerned with
  - reliability, safety, understandability, compatibility, robustness, …

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

An architecture helps with

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

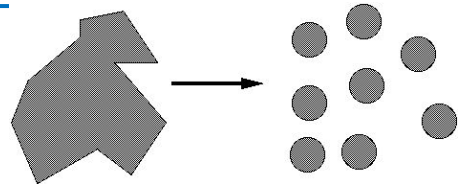
Evolution: high-level view shows evolution path

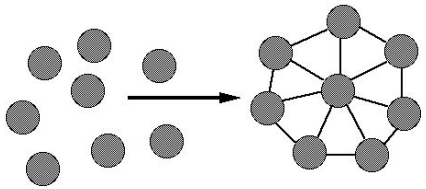
Management: helps understand work items and track progress

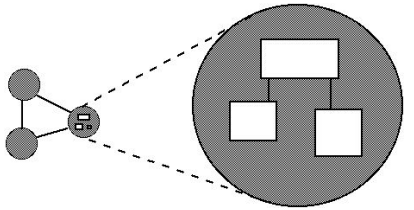
Communication: provides vocabulary; a picture says 1000 words

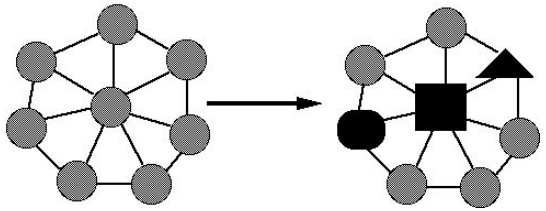
Qualities of modular software

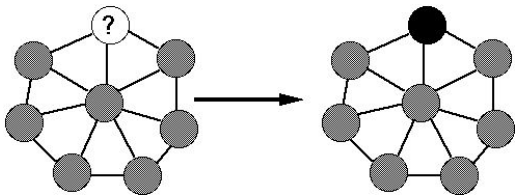
- decomposable
  - can be broken down into pieces
- composable
  - pieces are useful and can be combined
- understandable
  - one piece can be examined in isolation
- has continuity
  - change in reqs affects few modules
- protected / safe
  - an error affects few other modules





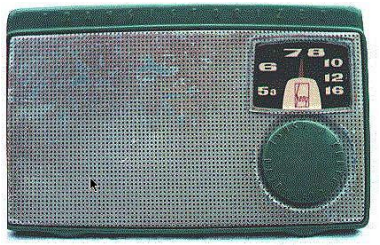






Interface and implementation

- **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

Summary

- An architecture provides a high-level framework to build and evolve a software system.
- Strive for modularity: strong cohesion and loose coupling.
- Consider using existing architectural styles or patterns.

Bonus slides

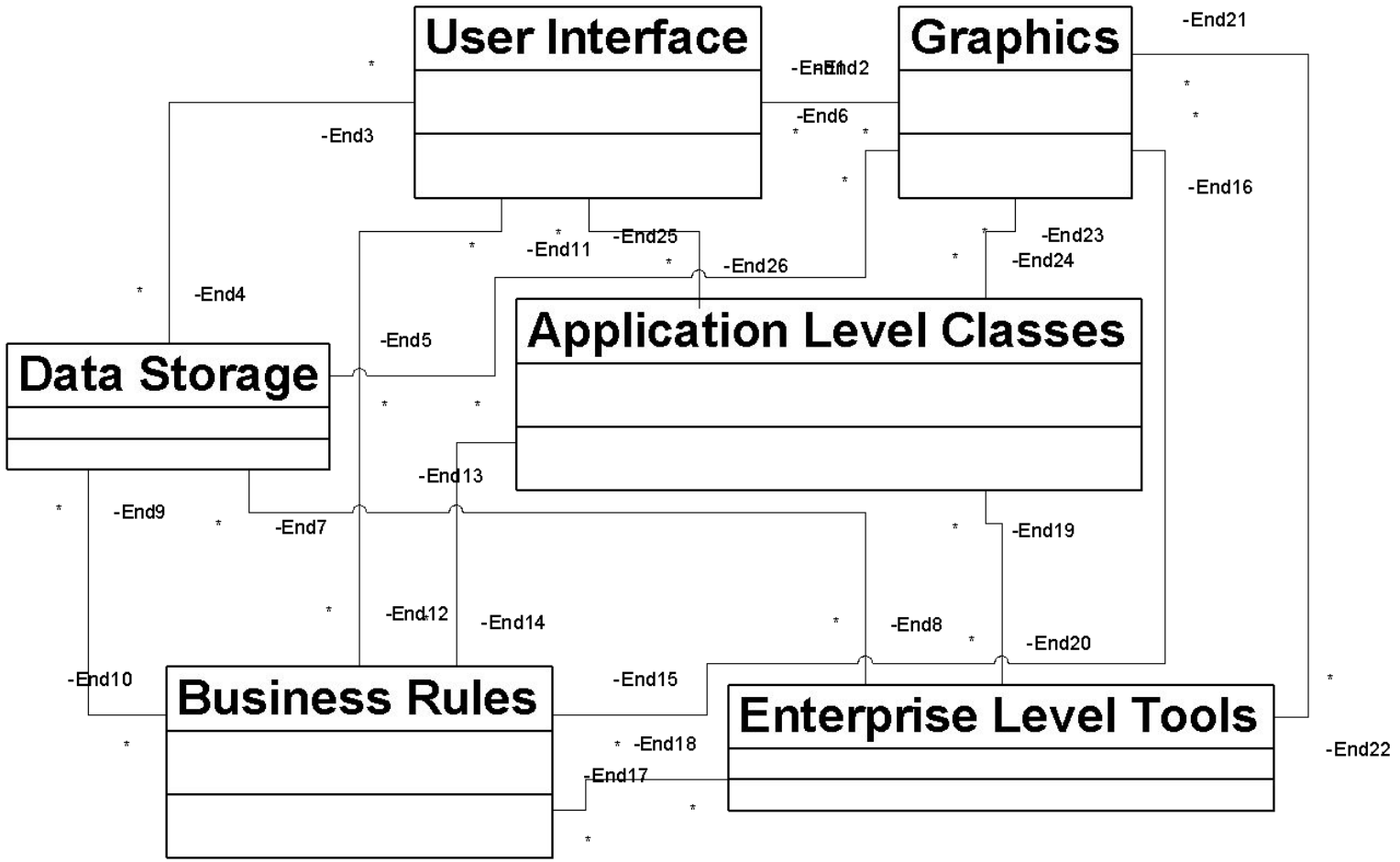
Properties of architecture

- Coupling
- Cohesion
- Style conformity
- Matching

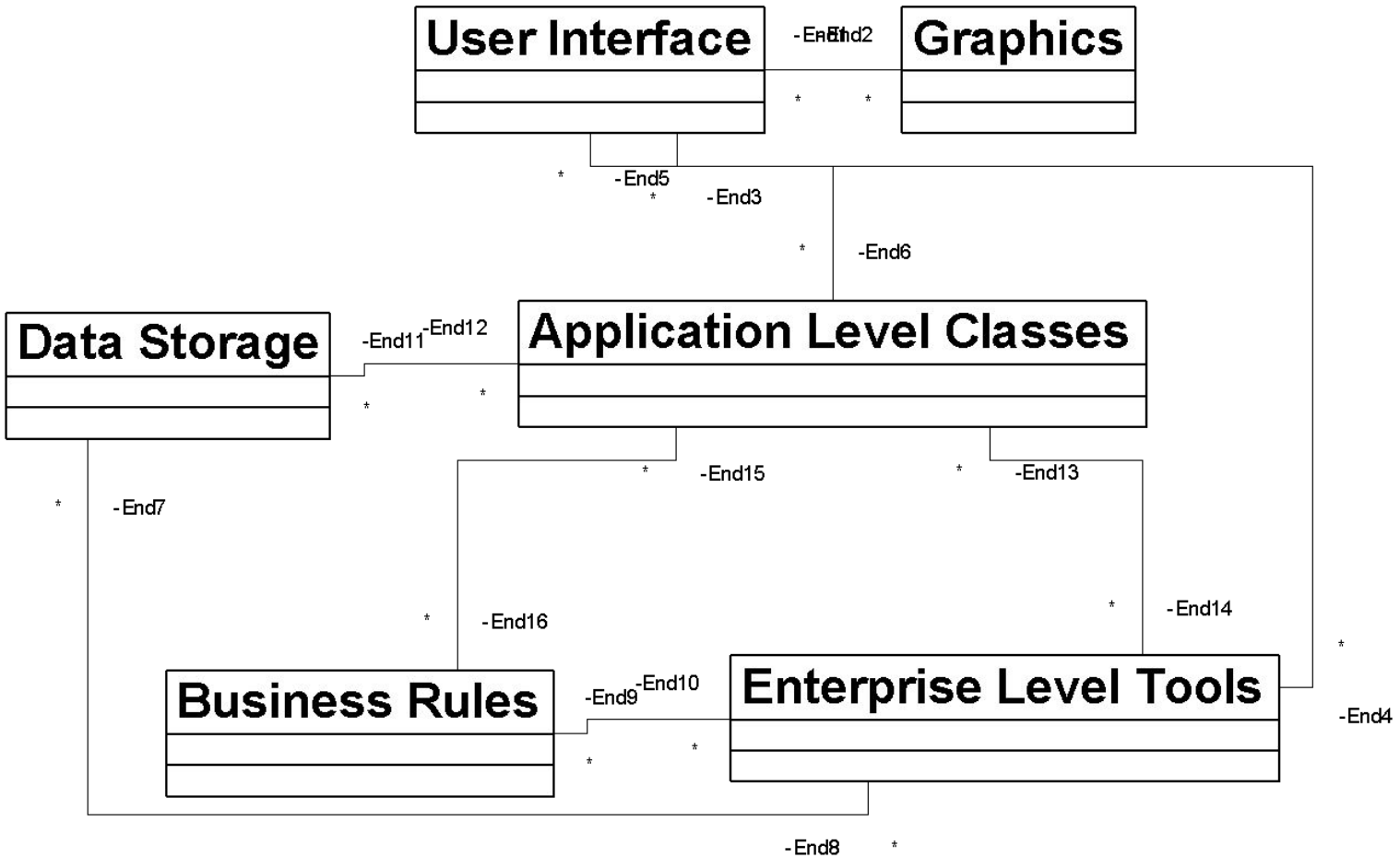
Coupling (loose vs. tight)

- *Coupling*: the kind and quantity of interconnections among modules
- Modules that are loosely coupled (or uncoupled) are better than those that are tightly coupled
- The more tightly coupled two modules are, the harder it is to work with them separately

Tightly or loosely coupled?



Tightly or loosely coupled?



Cohesion (strong vs. weak)

- *Cohesion*: how closely the operations in a module are related
- Tight relationships improve clarity and understanding
- A class with good abstraction usually has strong internal cohension
- No schizophrenic classes!

Strong or weak cohesion?

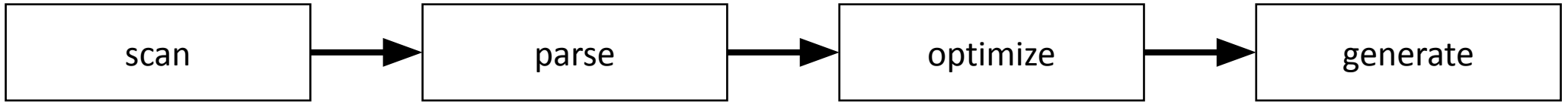
```
class Employee {

public:
  …
  FullName GetName() const;
  Address GetAddress() const;
  PhoneNumber GetWorkPhone() const;
  …
  bool IsJobClassificationValid(JobClassification jobClass);
  bool IsZipCodeValid (Address address);
  bool IsPhoneNumberValid (PhoneNumber phoneNumber);
 …
  SqlQuery GetQueryToCreateNewEmployee() const;
  SqlQuery GetQueryToModifyEmployee() const;
  SqlQuery GetQueryToRetrieveEmployee() const;
 …
}
```

Style conformity: What is a style?

- An **architectural style** defines
  - The vocabulary of components and connectors for a family (style)
  - Constraints on the elements and their combination
    - Topological constraints (no cycles, register/announce relationships, etc.)
    - Execution constraints (timing, etc.)
- By choosing a style, one gets all the known properties of that style (for in that style)
  - For example: performance, lack of deadlock, ease of making particular classes

An architectural style imposes constraints

- Pipes & filters
  - Pipes must compute local transformations
  - Filters must not share state with other filters
  - There must be no cycles
- If these constraints are violated, it's not a pipe & filter system
  - One can't tell this from a picture
  - One can formalize these constraints



The design and the reality

- The code is often less clean than the design
- The design is still useful
  - communication among team members
  - selected deviations can be explained more concisely and with clearer reasonir

Architectural mismatch

- Some components are inherently incompatible
  - Assumptions about memory allocation, vs. custom allocator
  - Use of two frameworks (assumes it is **`main`**)
  - Library wants to operate first or last
  - Data formats
  - Assumed infrastructure

Views

A **view** illuminates a set of top-level design decisions

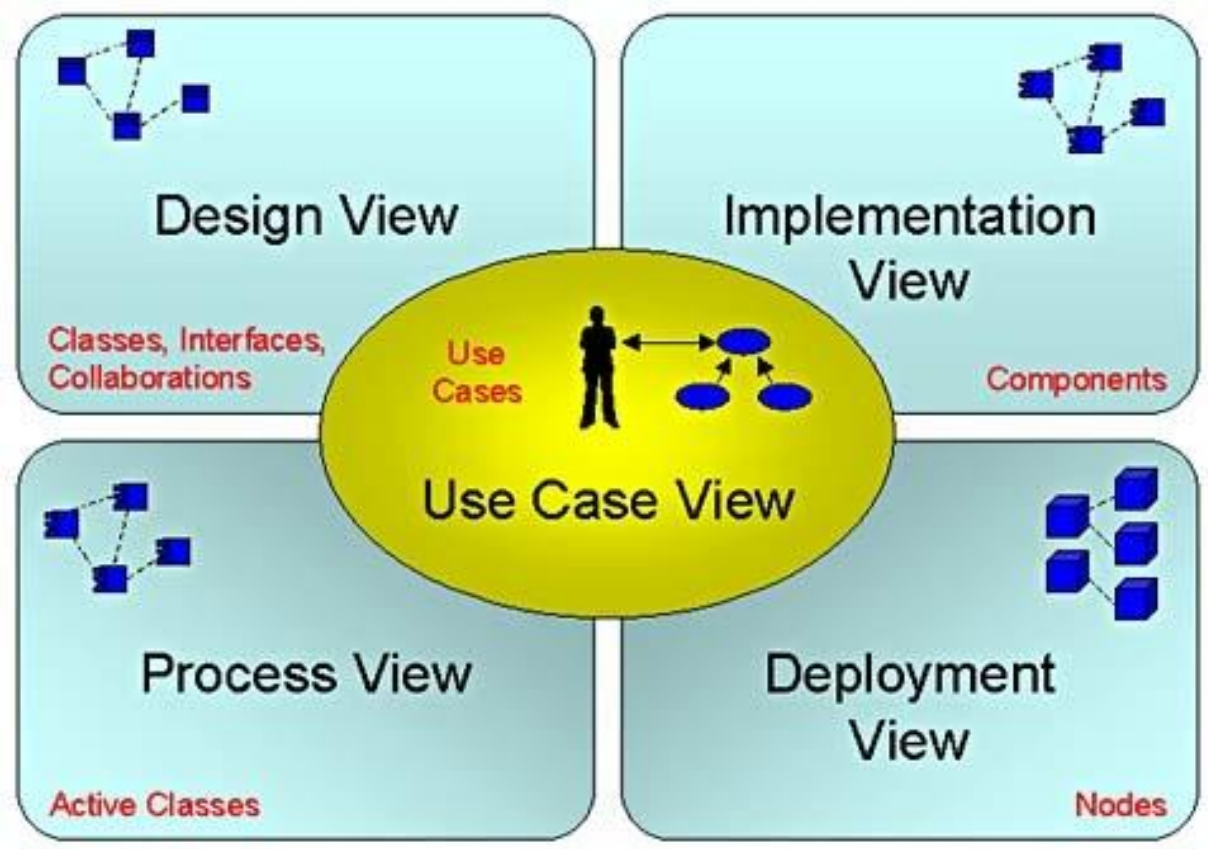
- how the system is composed of interacting parts
- where are the main pathways of interaction
- key properties of the parts
- information to allow high-level analysis and appraisal

Importance of views

Multiple views are needed to understand the different dimensions of systems

Functional Requirements

Performance (execution) Requirements



Packaging Requirements

Installation Requirements

Booch