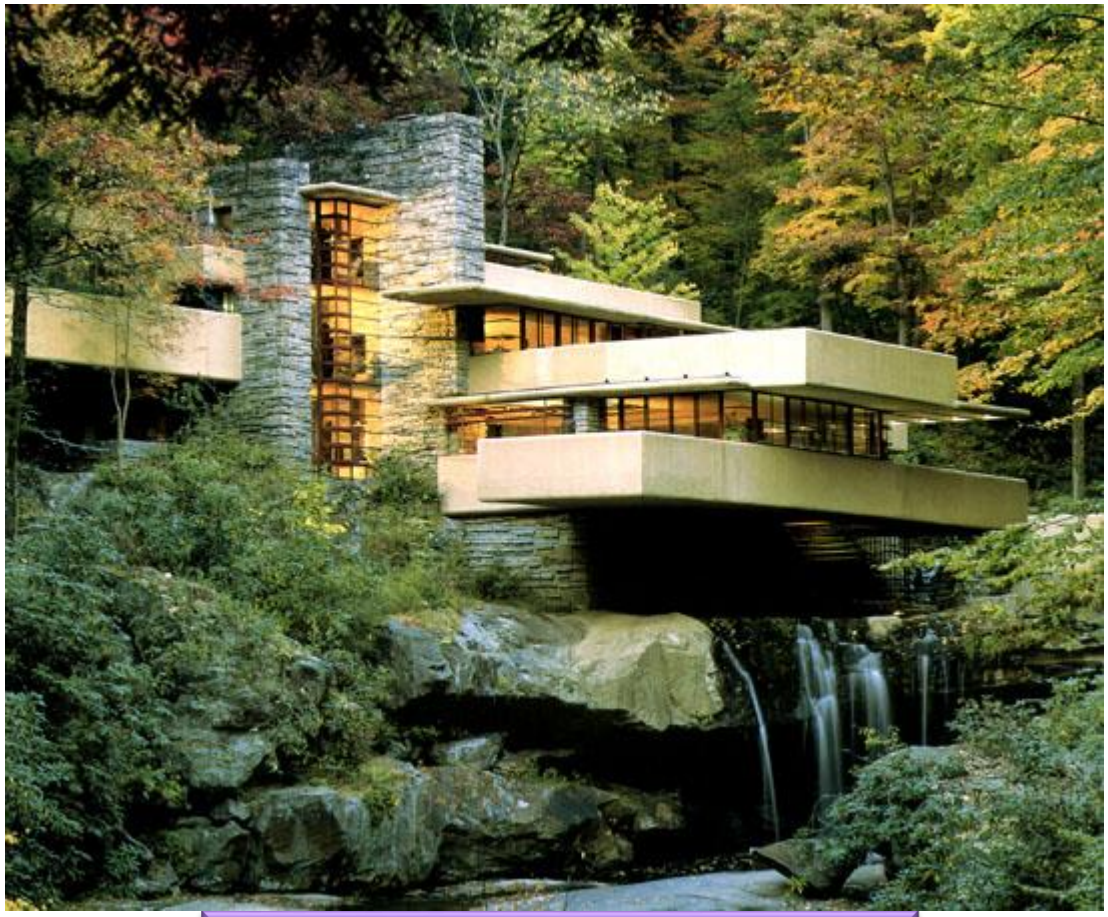
# Architecture

## CSE 403



Fallingwater by Frank Lloyd Wright

# Outline

- What is a software architecture?
- What does an architecture look like?
- What is a good architecture?
- Properties of architectures
- Example architectures

# Why architecture?

"Good software architecture makes the rest of the project easy."

Steve McConnell, Survival Guide

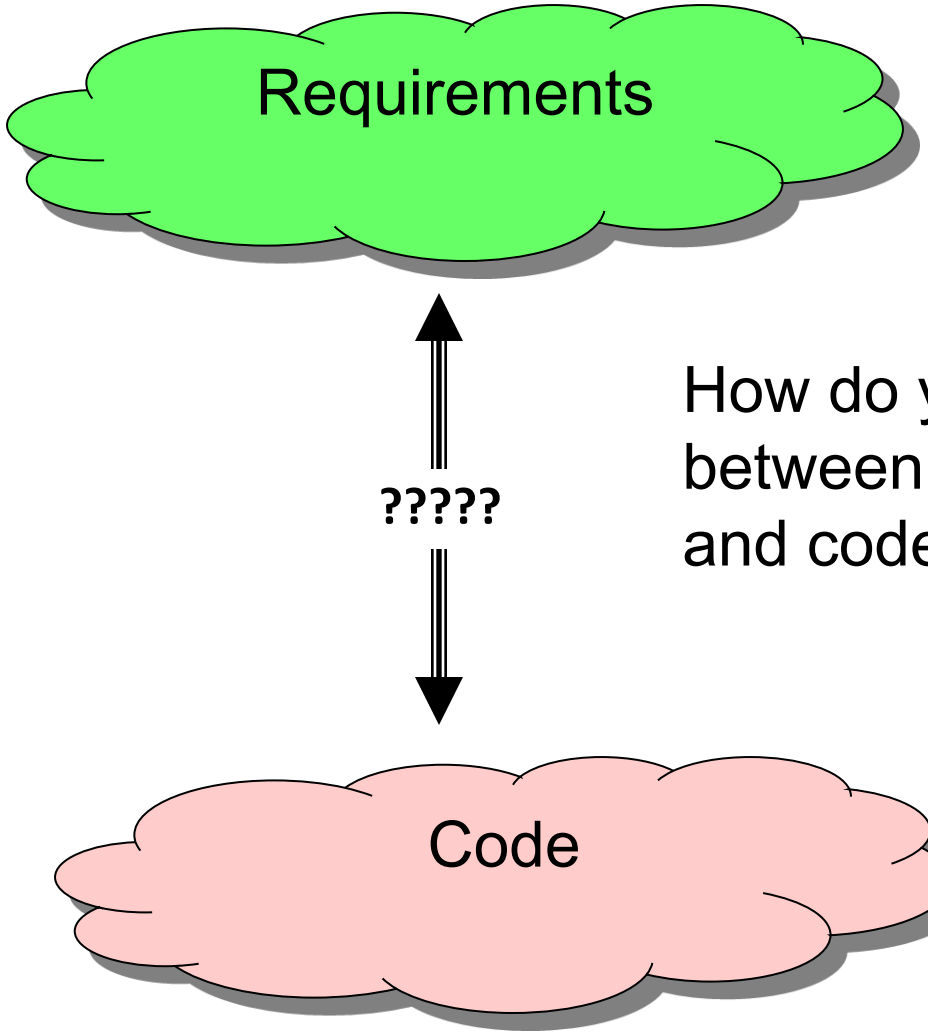
"There are two ways of constructing a software design:

one way is to make it so simple that there are obviously no deficiencies;

the other is to make it so complicated that there are no obvious deficiencies."

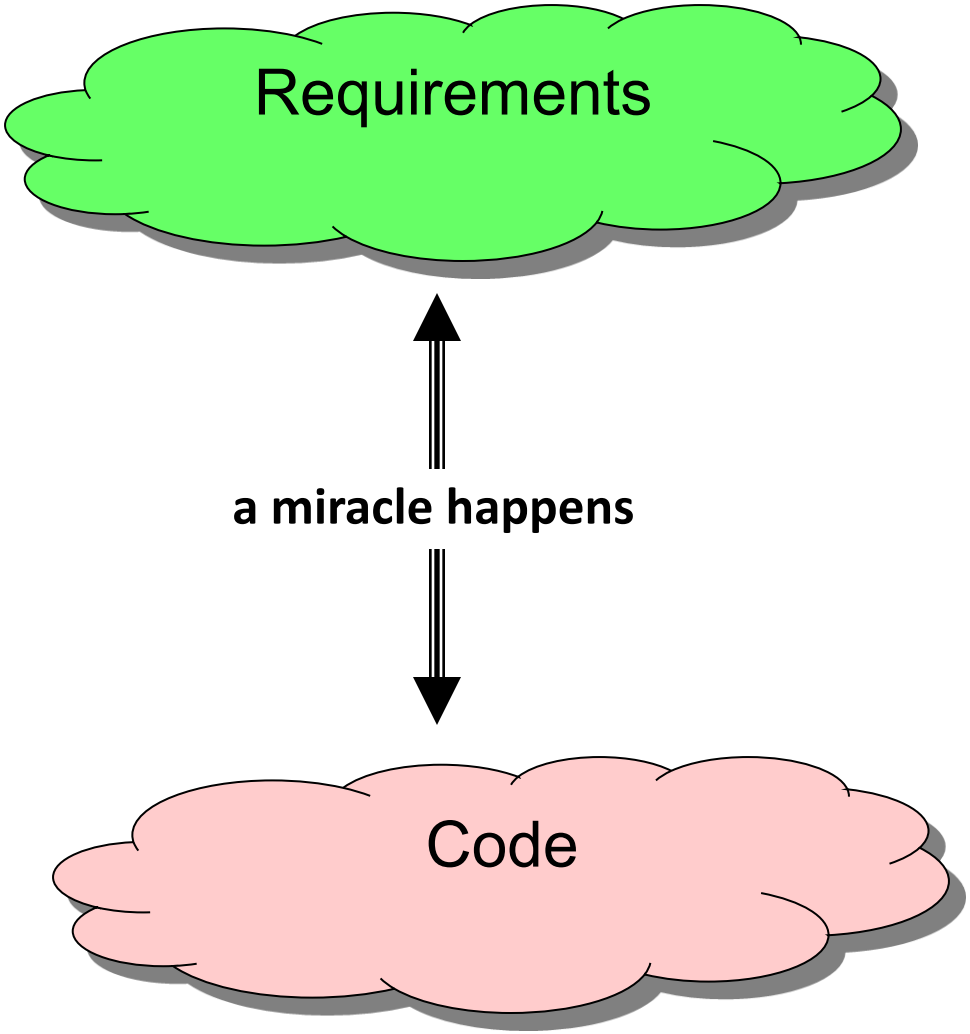
C.A.R. Hoare (1985)

# The basic problem: From requirements to code

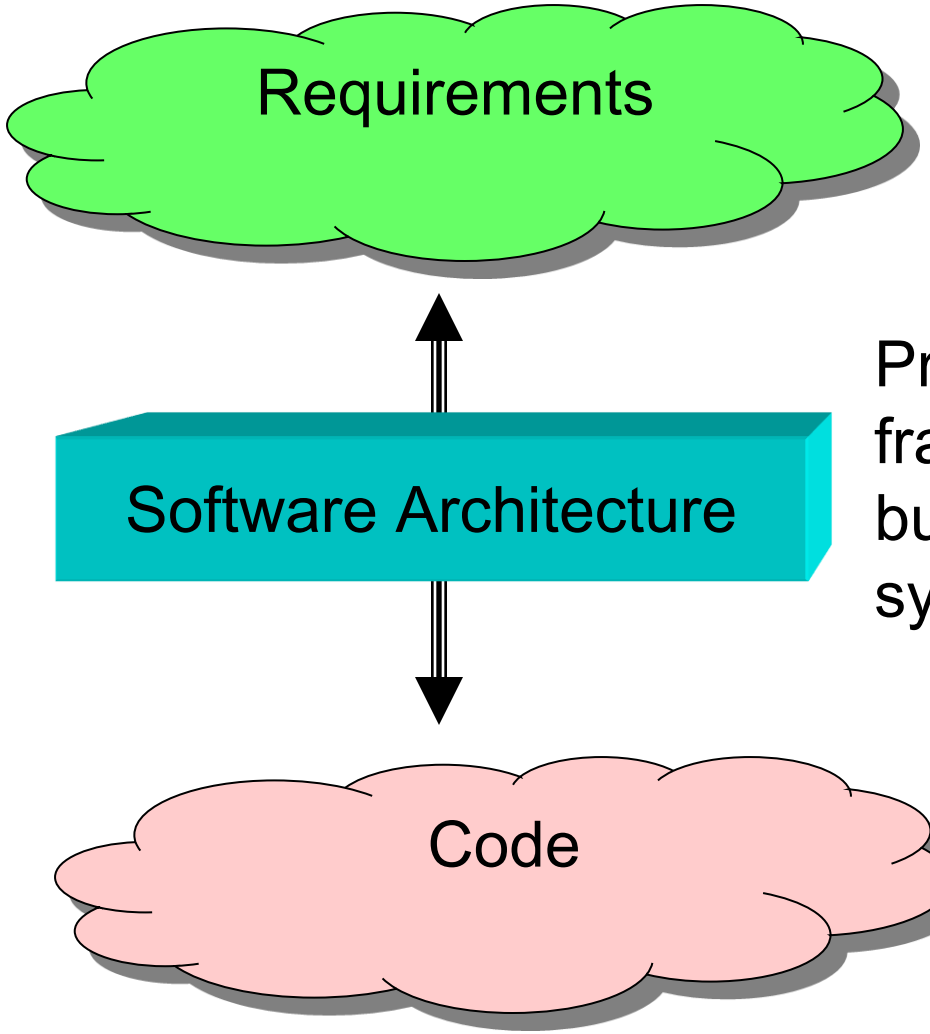



How do you bridge the gap between requirements and code?

# One answer:
# Solve with inspiration



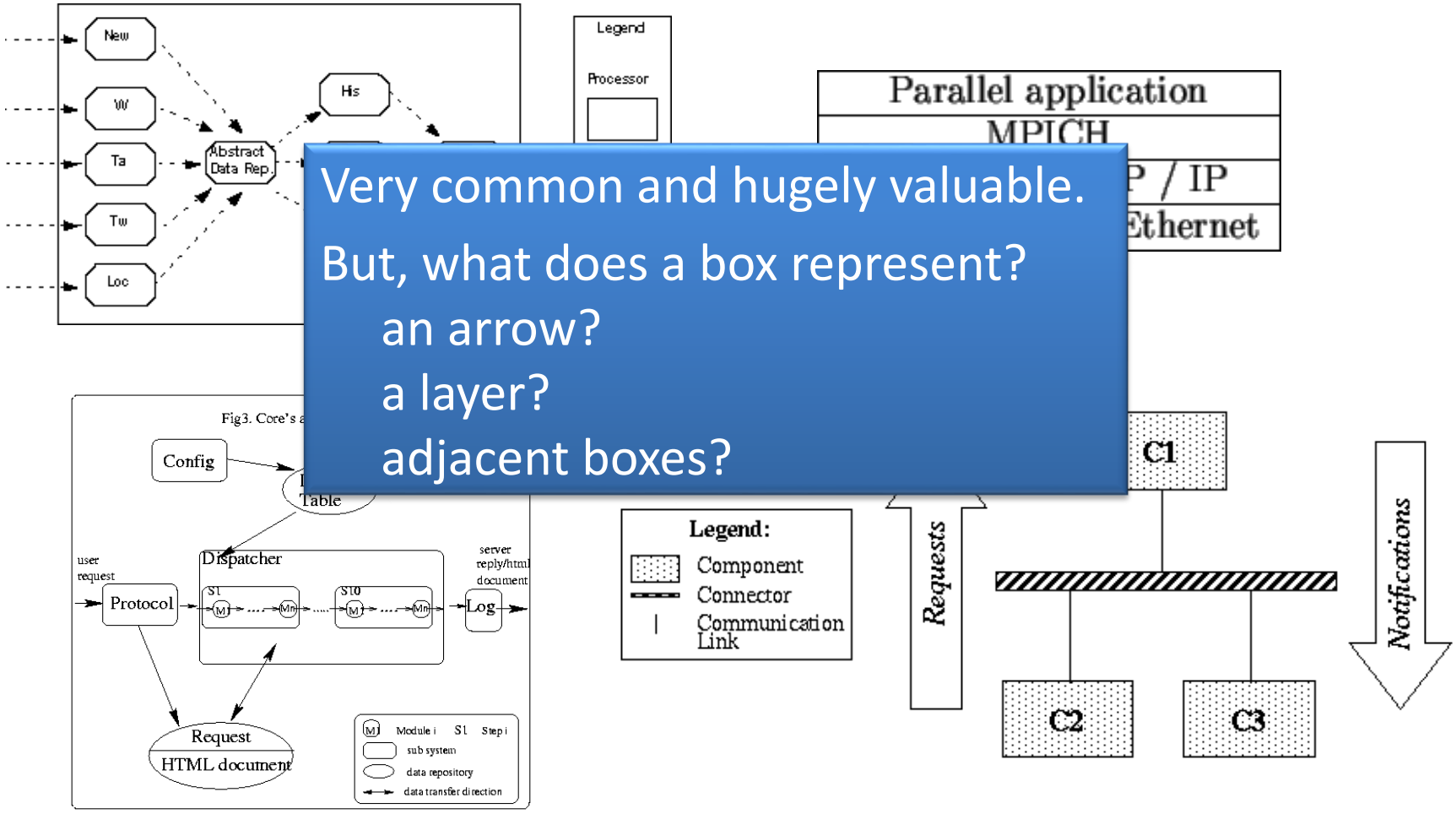
# A better answer: Solve with engineering




Provides a high-level framework to build and evolve the system

# What does an architecture look like?

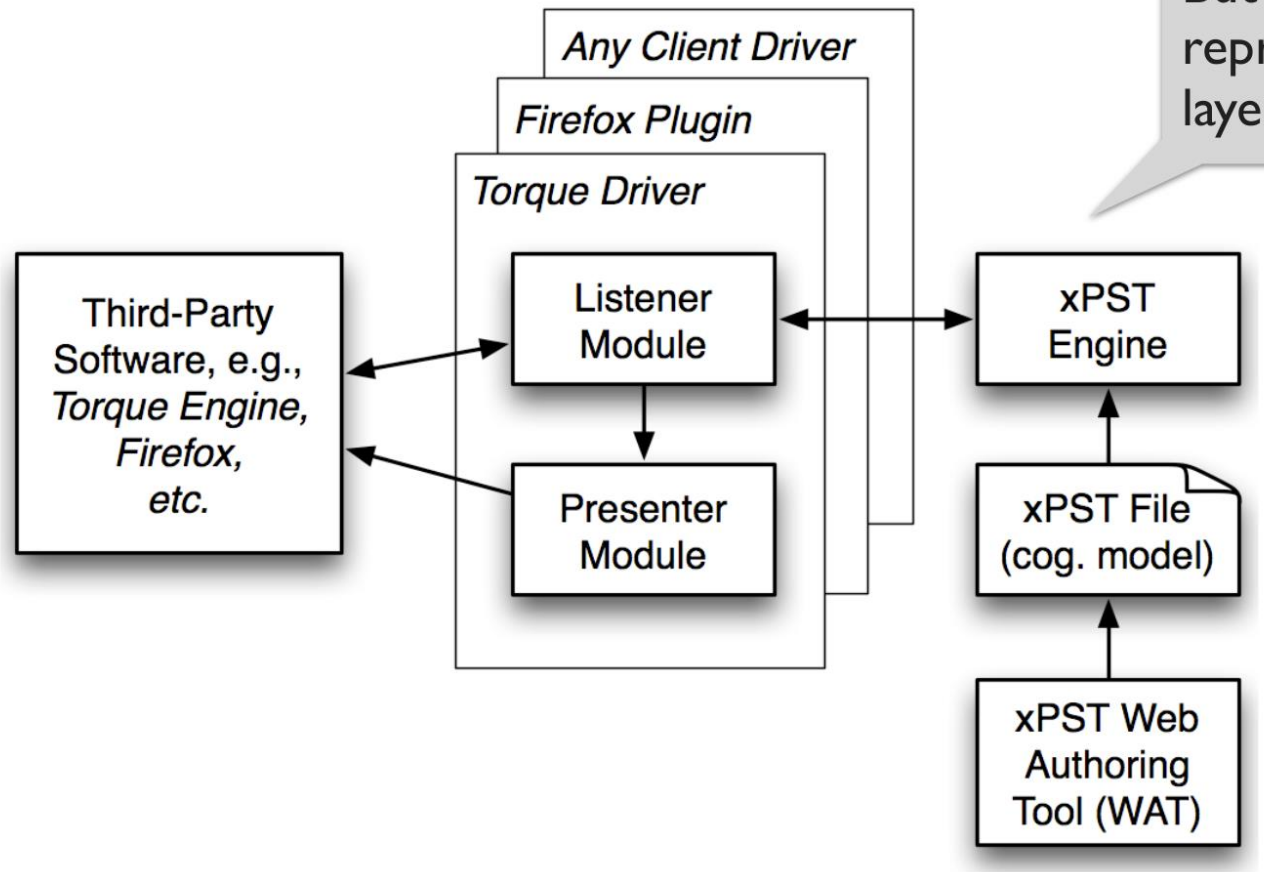
# Box-and-arrow diagrams



# Box and arrow diagrams redux

Very common and highly valuable.

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



# An architecture: components and connectors

- *Components* define the basic computations comprising the system and their behaviors
  - abstract data types, filters, etc.
- *Connectors* define the interconnections between components
  - procedure call, event announcement, asynchronous message sends, etc.
- The line between them may be fuzzy at times
  - 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
- Topic of next lecture

# What is 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

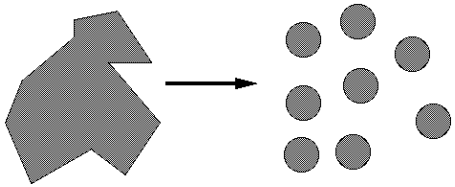
**Evolution**: high-level view shows evolution path

**Management**: helps understand work items and track progress

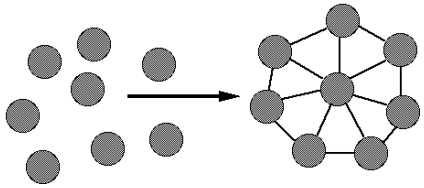
**Communication**: provides vocabulary; a picture says 1000 words

# Qualities of modular software

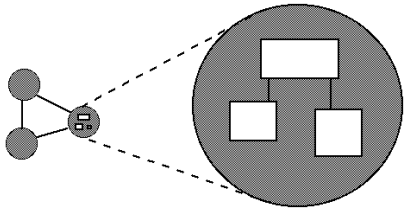
- decomposable
  - can be broken down into pieces



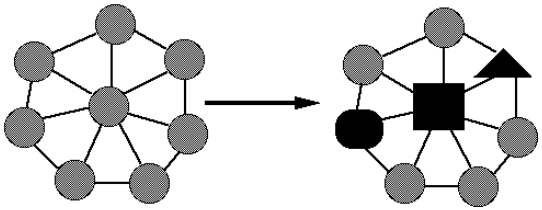
- composable
  - pieces are useful and can be combined



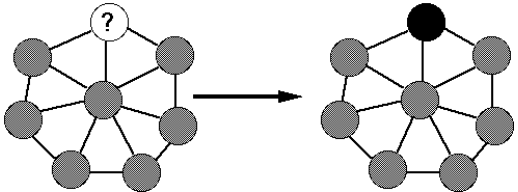
- understandable
  - one piece can be examined in isolation



- 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

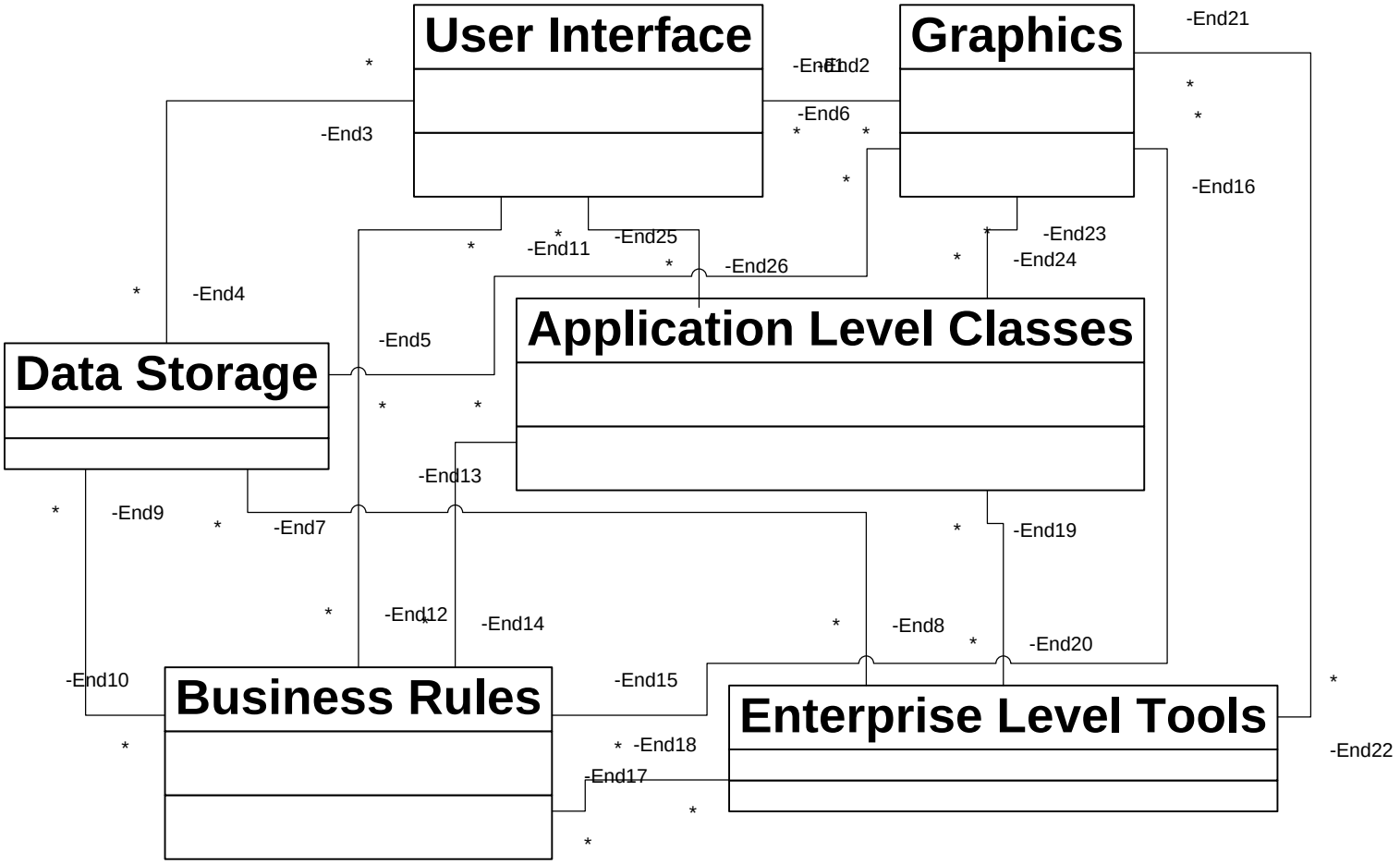
# Properties of architecture

- Coupling
- Cohesion
- Style conformity
- Matching
- Errosion

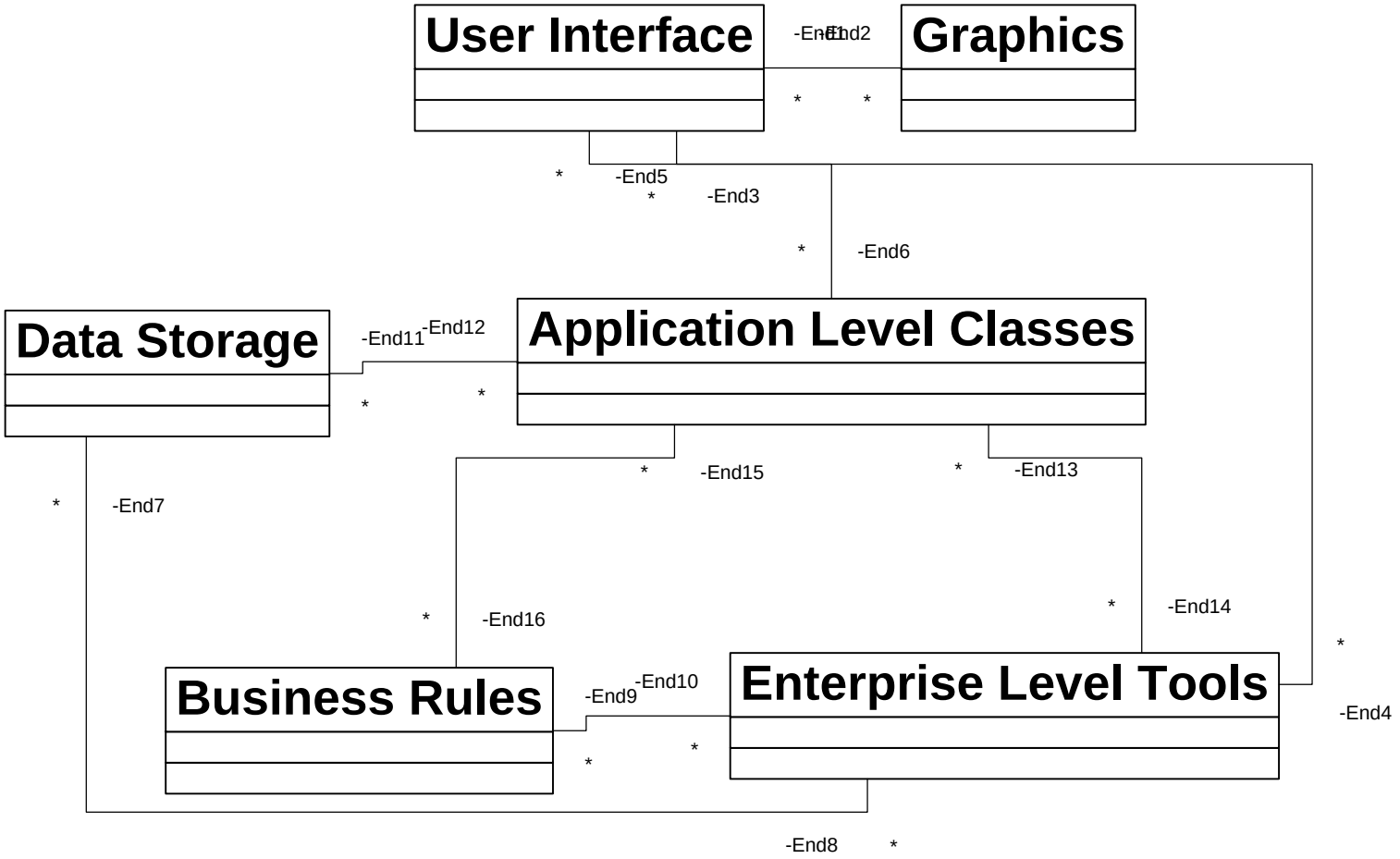
# Coupling (loose vs. tight)

- *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
- Classes with good abstraction usually have strong cohension
- No schizophrenic classes!

# Strong or weak cohesion?

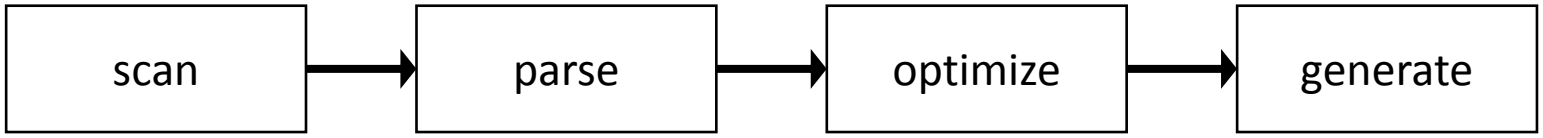
```
class Employee {

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 any architecture in that style)
  - For example: performance, lack of deadlock, ease of making particular classes of changes, etc.

# Styles are not just boxes and arrows

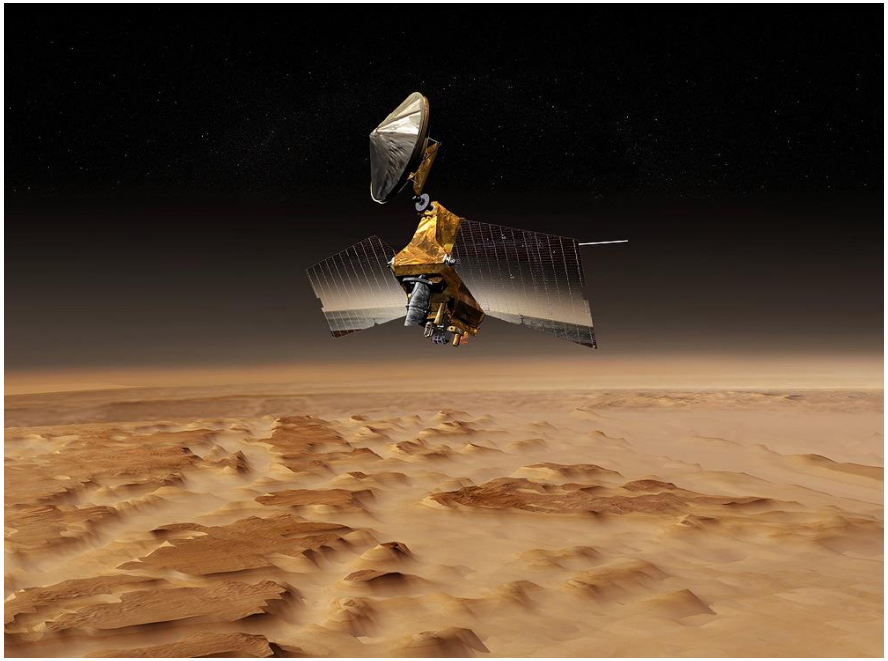
- Consider pipes & filters, for example (Garlan and Shaw)
  - Pipes must compute local transformations
  - Filters must not share state with other filters
  - There must be no cycles
- If these constraints are violated, it's not a pipe & filter system
  - One can't tell this from a picture
  - One can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



# Interface mismatch

- Mars orbiter loss

  NASA lost a 125 million Mars orbiter because one engineering team used metric units while another used English units for a key spacecraft operation



# 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 reasoning

# Architectural mismatch

- Garlan, Allen, Ockerbloom tried to build a toolset to support software architecture definition from existing components
  - OODB (OBST)
  - graphical user interface toolkit (Interviews)
  - RPC mechanism (MIG/Mach RPC)
  - Event-based tool integration mechanism (Softbench)
- It went to hell in a handbasket, not because the pieces didn't work, but because they didn't fit together
  - Excessive code size
  - Poor performance
  - Needed to modify out-of-the-box components (e.g., memory allocation)
  - Error-prone construction process
- Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software 12*, 6 (Nov. 1995)
- Architecture should warn about such problems (& identify problems)

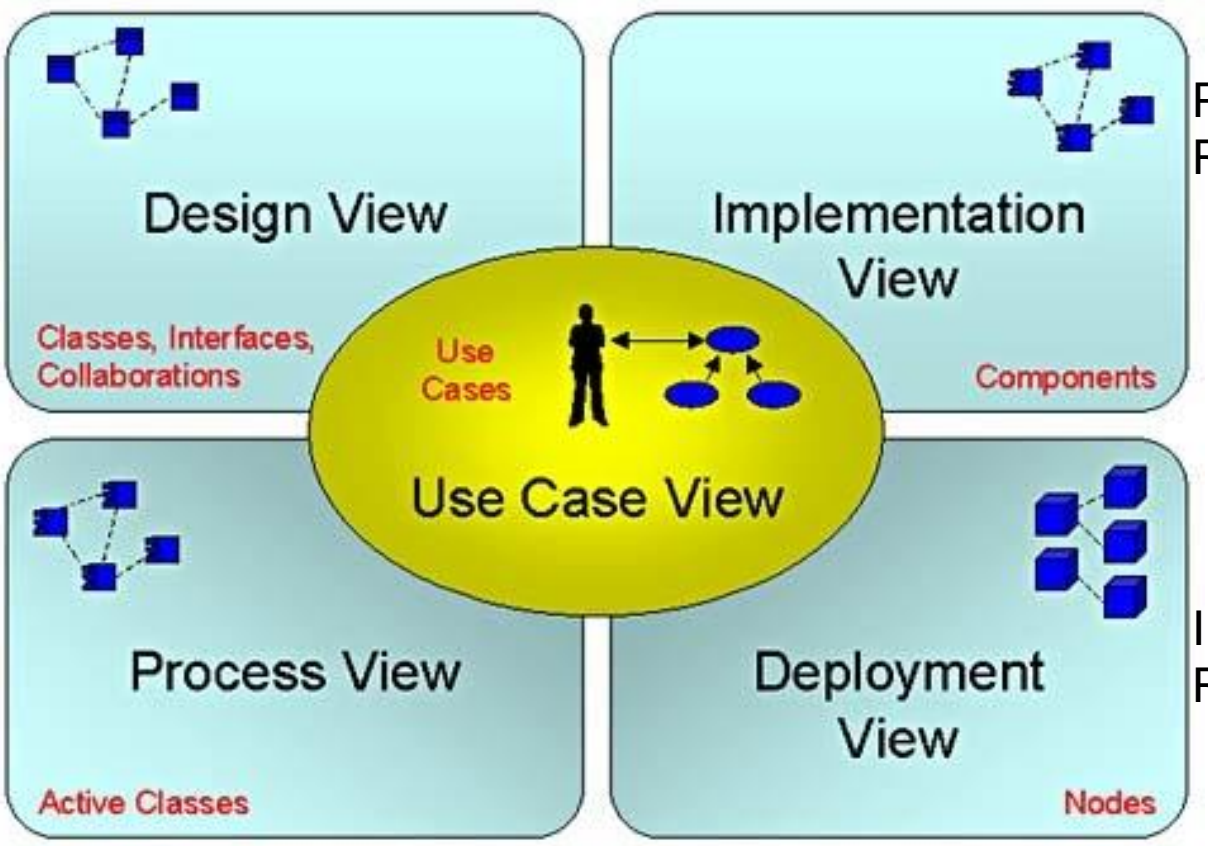
# Views

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

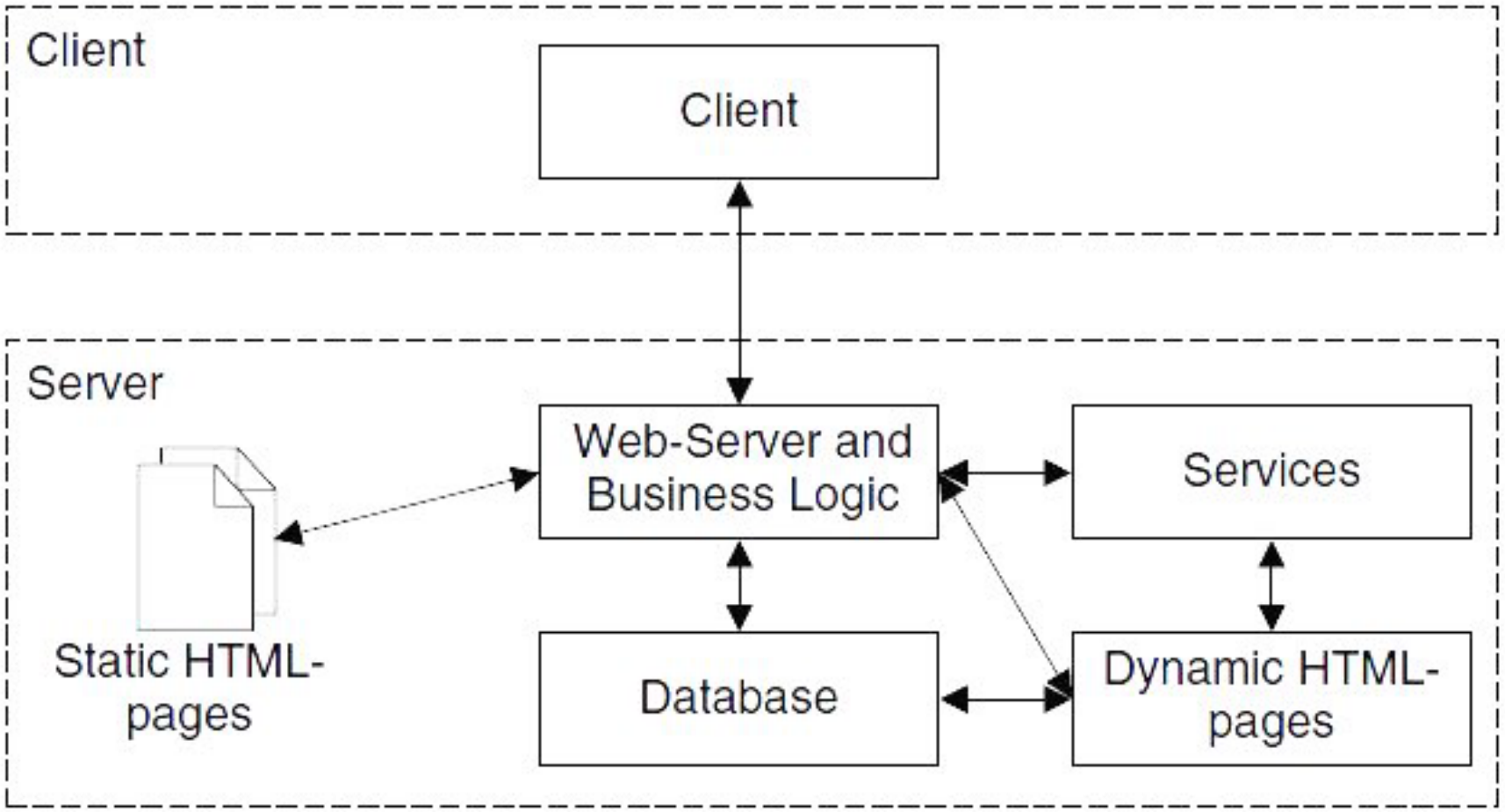
Packaging Requirements

Performance (execution) Requirements

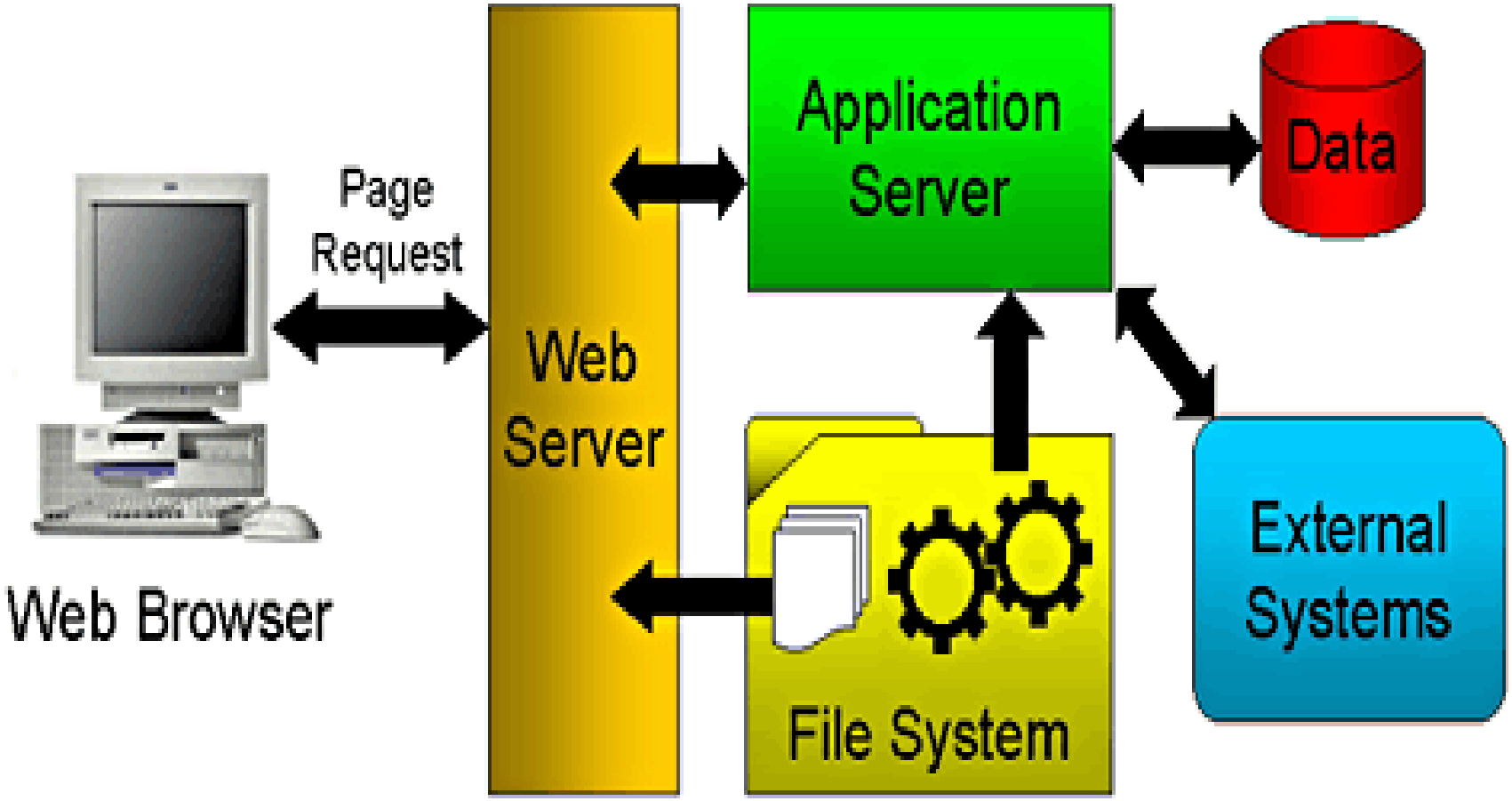
Installation Requirements

Booch

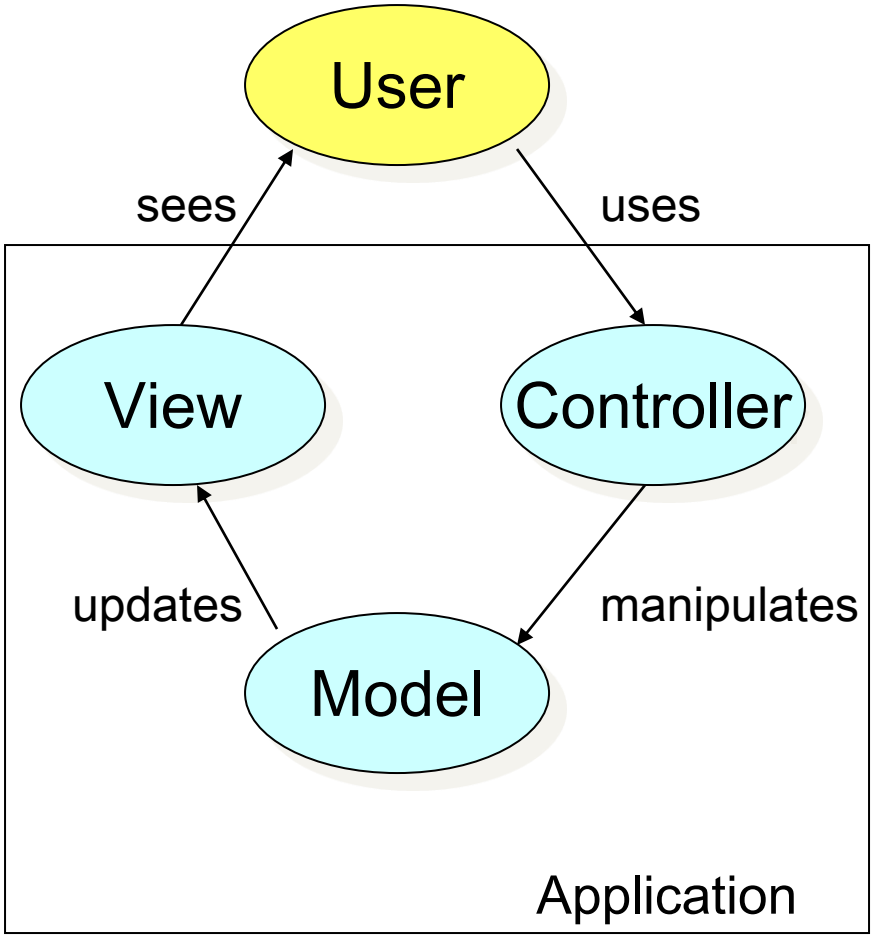
# Client-server architecture



# Web application (client-server)



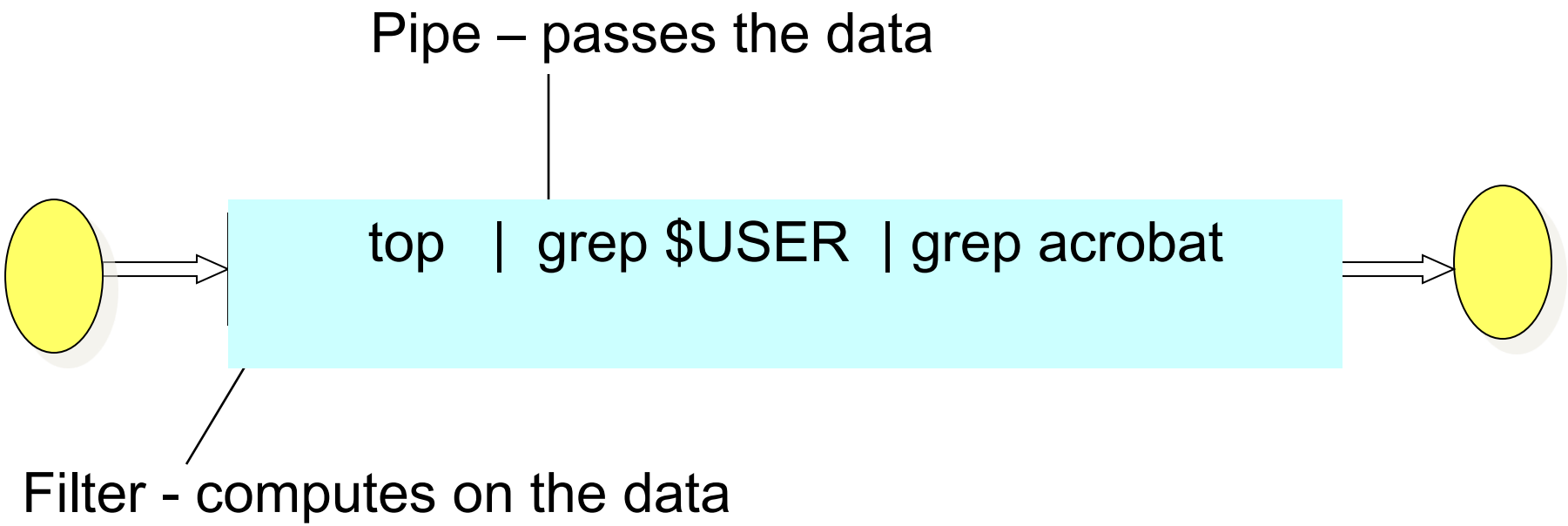
# Model-View-Controller




Separates:

- the application object (model)
- the way it is represented to the user (view)
- the way in which the user controls it (controller).

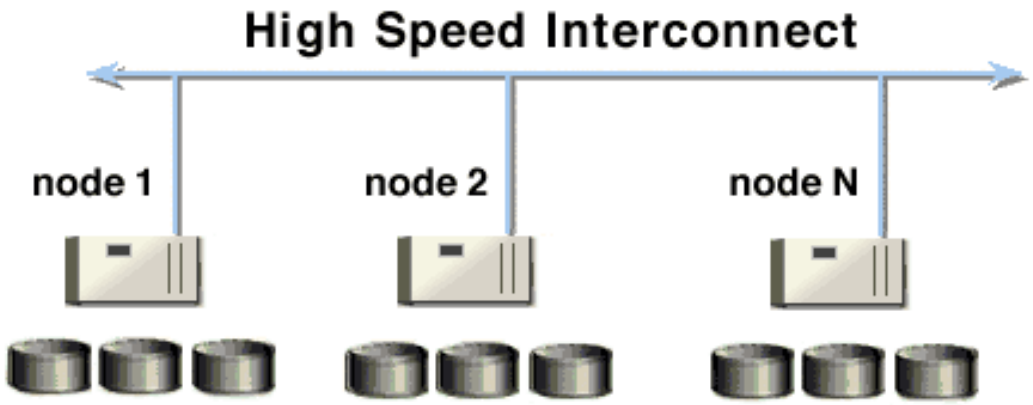
# Pipe and filter




Each stage of the pipeline acts independently of the others.

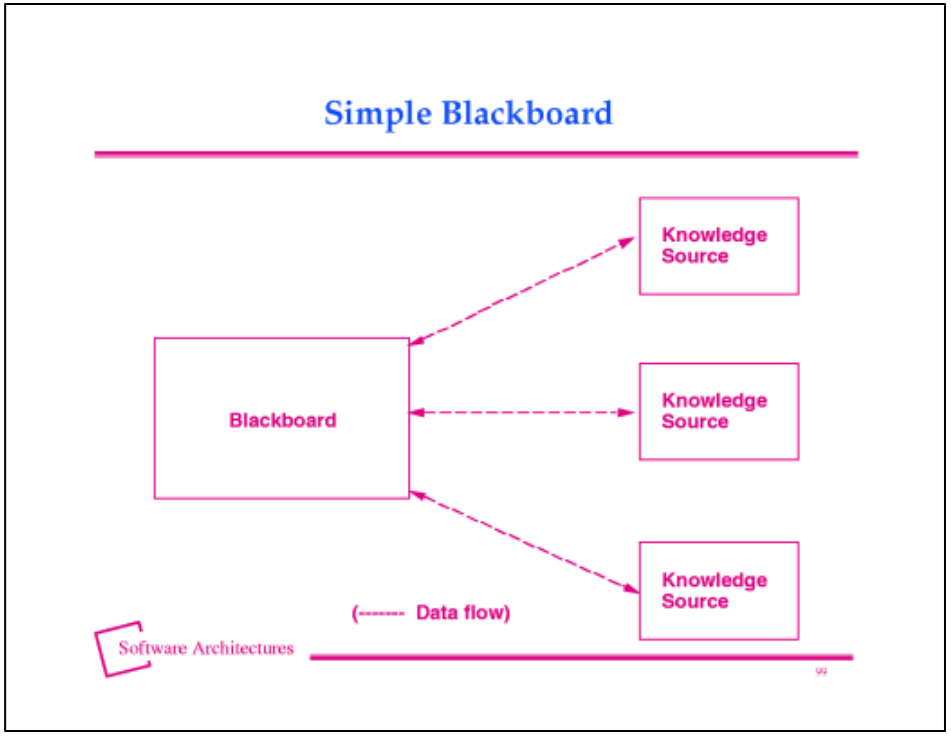
Can you think of a system based on this architecture?

# Shared nothing architecture



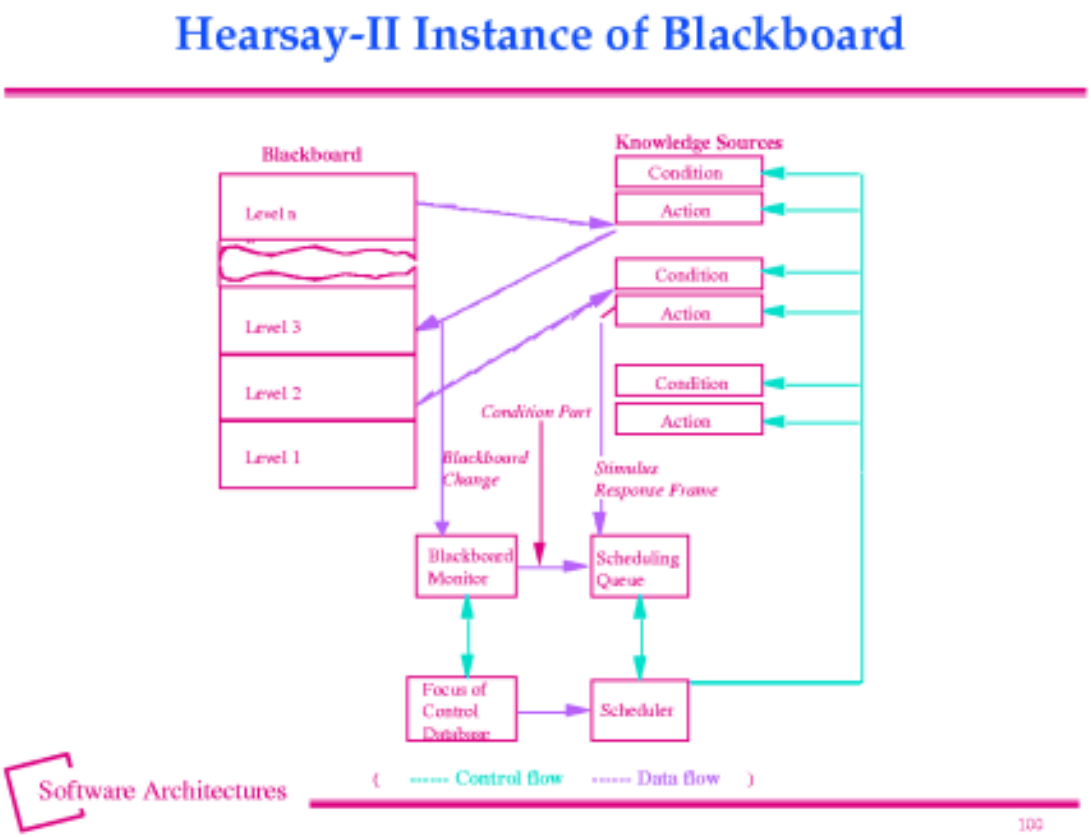
# Blackboard architectures

- *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.




Blackboard systems have traditionally been used for applications requiring complex interpretations of signal processing, such as speech and pattern recognition.

# Hearsay-II: blackboard



# 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.