Software Architecture

CITS4401 Software Requirements and Design

Week 8

Recap

1. Modular Design (Decomposition of the system)

2. Coupling and Cohesion of the modular systems

Goal of This Week

- **Software Architecture**
  - What is software architecture
  - Introduction of the commonly used software architectures
- **Design Rationale**
  - Using the design rationale to document the system

Software Architecture

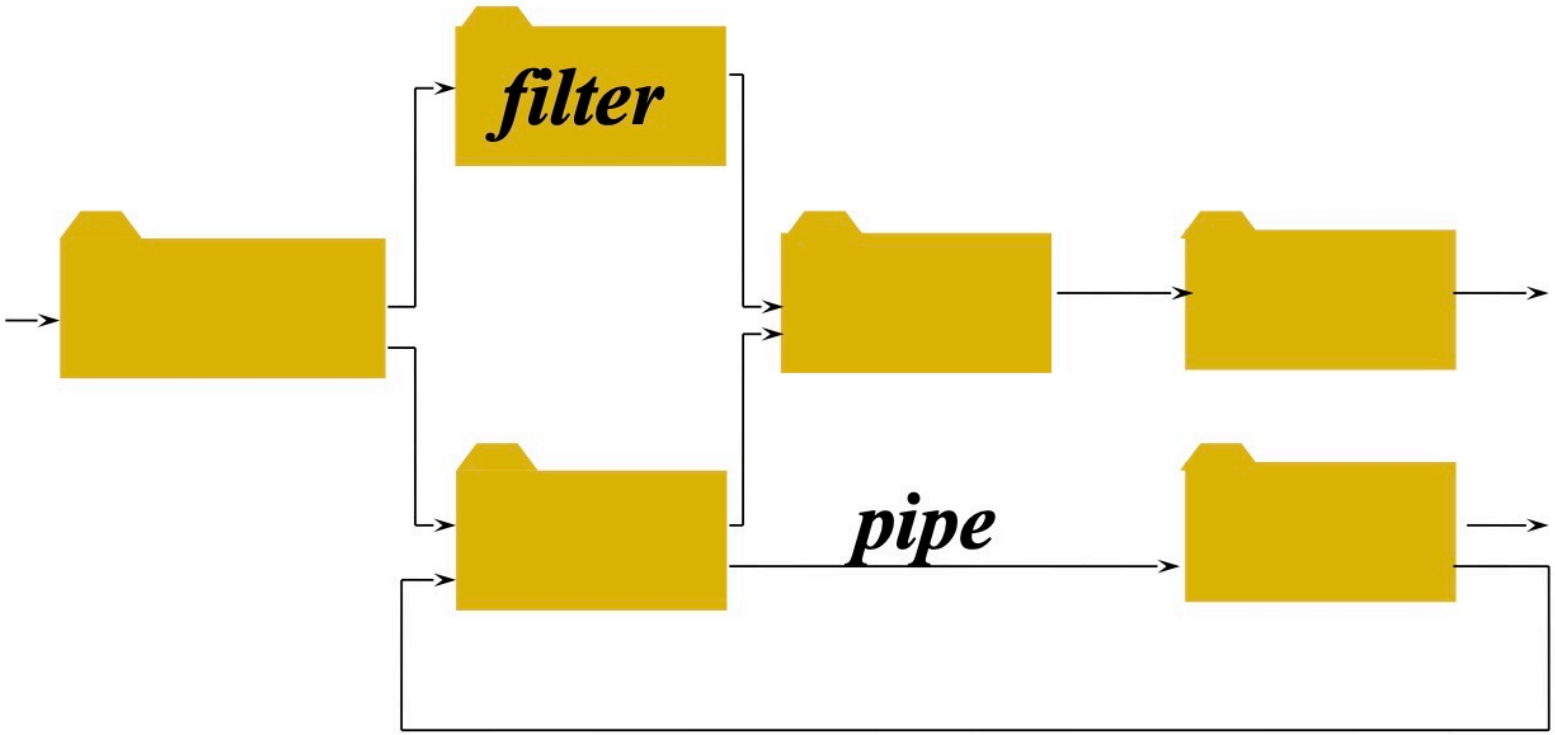
Why Study Software Architecture

- recognize common paradigms so you can understand
  - ✓ high level relationships among systems
  - ✓ build new systems as variations on existing ones
- getting the right architecture is often crucial to the success of a design
  - ✓ the wrong architecture can lead to disastrous results enables principled choices among design alternatives
- architectural system representation is often essential to analysis and description of high-level properties of a complex system

Common Framework

- All software architectures have 3 common elements:
- Components a software architecture consists of a collection of computational components
- Connectors the interactions between those components (e.g., procedure call, event broadcast, database queries or pipes)
- Constraints A software architecture might have some constraints imposed on it. e.g. topological constraint of having no cycles

Pipe and Filter Architecture structural pattern



Pipe and Filter Architecture Structural Pattern

- each component (filter)
  - reads streams of data on its inputs
  - applies a local transformation
  - produces streams of data on its outputs
- computation is incremental
  - output begins before all inputs are consumed
- connecters (pipes)
  - transmit outputs of one filter to inputs of another

Pipe and Filter Architecture Structural Pattern

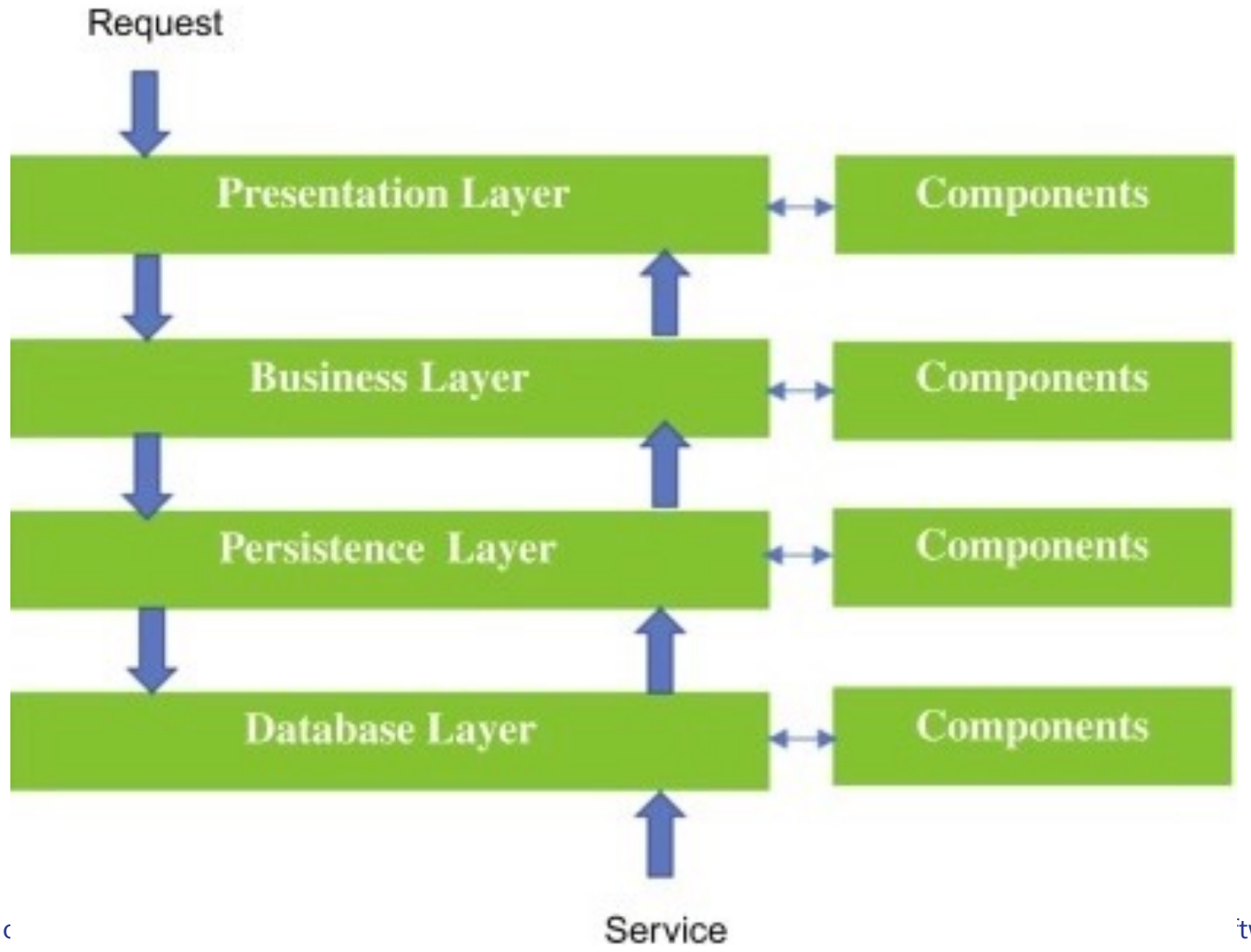
- filters must be independent entities
  - they do not share state with other filters
- filters do not know the identity of other
  - upstream and downstream filters
- filter specifications can
  - restrict what appears on the input pipes or
  - make guarantees about what appears on the output pipes

Pipe and Filter Architecture Structural Pattern

```
interface Filter {
    process(data)
}

// Define concrete filter implementations
class Filter1 implements Filter {
    process(data):
        // Processing logic for Filter 1
        return processed_data}


class Filter2 implements Filter {
    process(data):
        // Processing logic for Filter 2
        return processed_data}

// Define a Pipeline class to manage filters
class Pipeline {List<Filter> filters

    addFilter(filter):
        // Add filter to the pipeline
        filters.add(filter)

    process(data):
        // Process data through each filter in the pipeline
        for filter in filters:
            data = filter.process(data)
        return data
}
```

```
// Main program

// Create input data
data = ...

// Create filters
filter1 = new Filter1()
filter2 = new Filter2()

// Create pipeline and add filters
pipeline = new Pipeline()
pipeline.addFilter(filter1)
pipeline.addFilter(filter2)

// Process data through pipeline and obtain result
result = pipeline.process(data)

// Output the result
print(result)
```

Layered Architecture - Structure



Layered Architecture - Structure

- **Focus**: The Layering Architecture focuses on organizing components into distinct layers based on their responsibilities
- **Layer Interaction**: Components within each layer interact with components in the same layer or adjacent layers.
- **Separation of Concerns**: The architecture emphasizes separation of concerns by dividing functionality into separate layers. Each layer has a specific responsibility, making the system easier to understand, maintain, and scale.

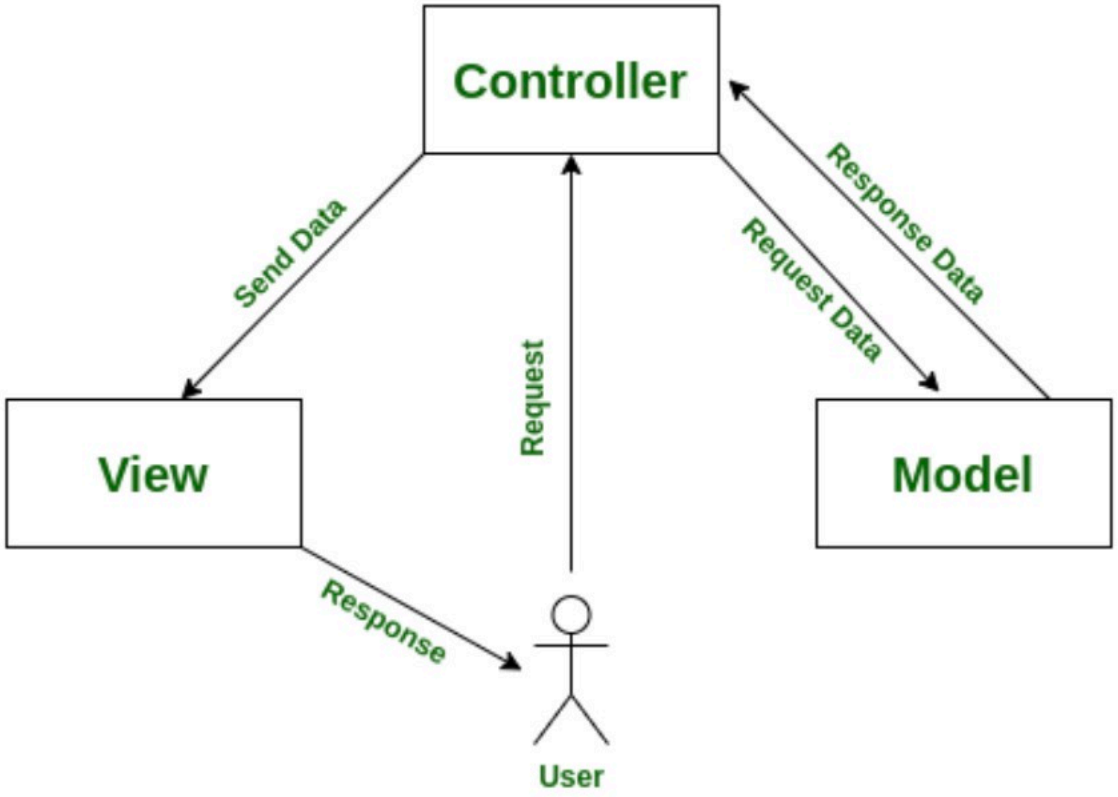
Layering Architecture - Advantages

- Design based on increasing levels of abstraction complex problem becomes sequence of incremental steps.
- Supports enhancement changes to one layer affect at most two other layers.
- Supports reuse different implementations of the same layer can be used interchangeably provided they support the same interfaces

Layering Architecture – Disadvantages

- Not all systems are easily structured in layers
- Performance considerations
  - may need close coupling between logically high-level functions
  - and their low-level implementations
- Can be difficult to find right level of abstraction e.g. protocols which bridge several OSI layers

MVC Architecture – Structure



MVC Architecture

- MVC stands for model-view-controller. Here's what each of those components mean:

- **Model**: The backend that contains all the data logic
- **View**: The frontend or graphical user interface (GUI)
- **Controller**: The brains of the application that controls how data is displayed

MVC Architecture

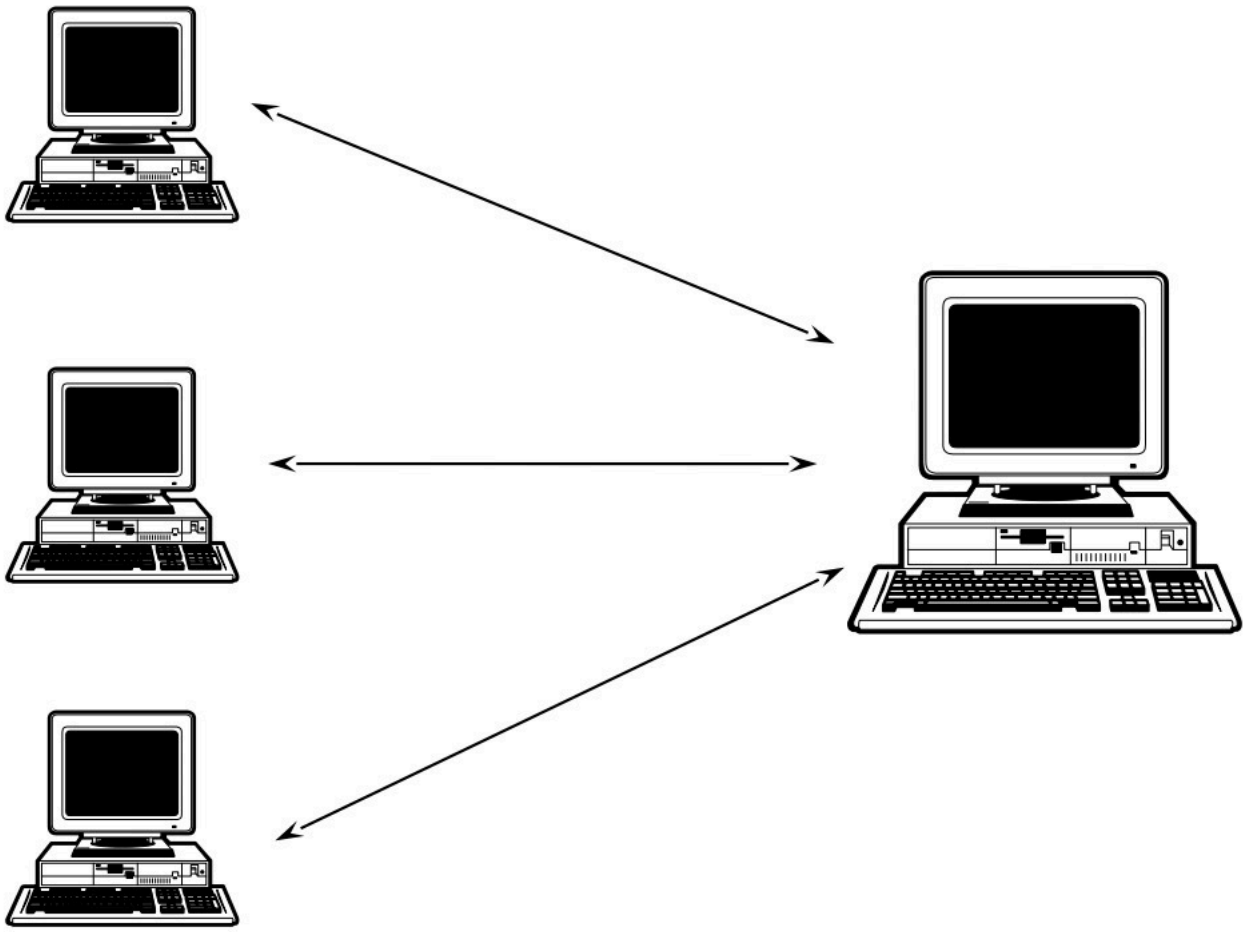
```
// Model
class Model { private String data;
    getData():
        return data
    setData(newData):
        data = newData }

// View
class View {
 displayData(data):
        // Display data to the user

```

```
// Controller
class Controller {
    private Model model;
    private View view;

    Controller(model, view):
        this.model = model
        this.view = view


processData(newData):
        // Update model with new data
```

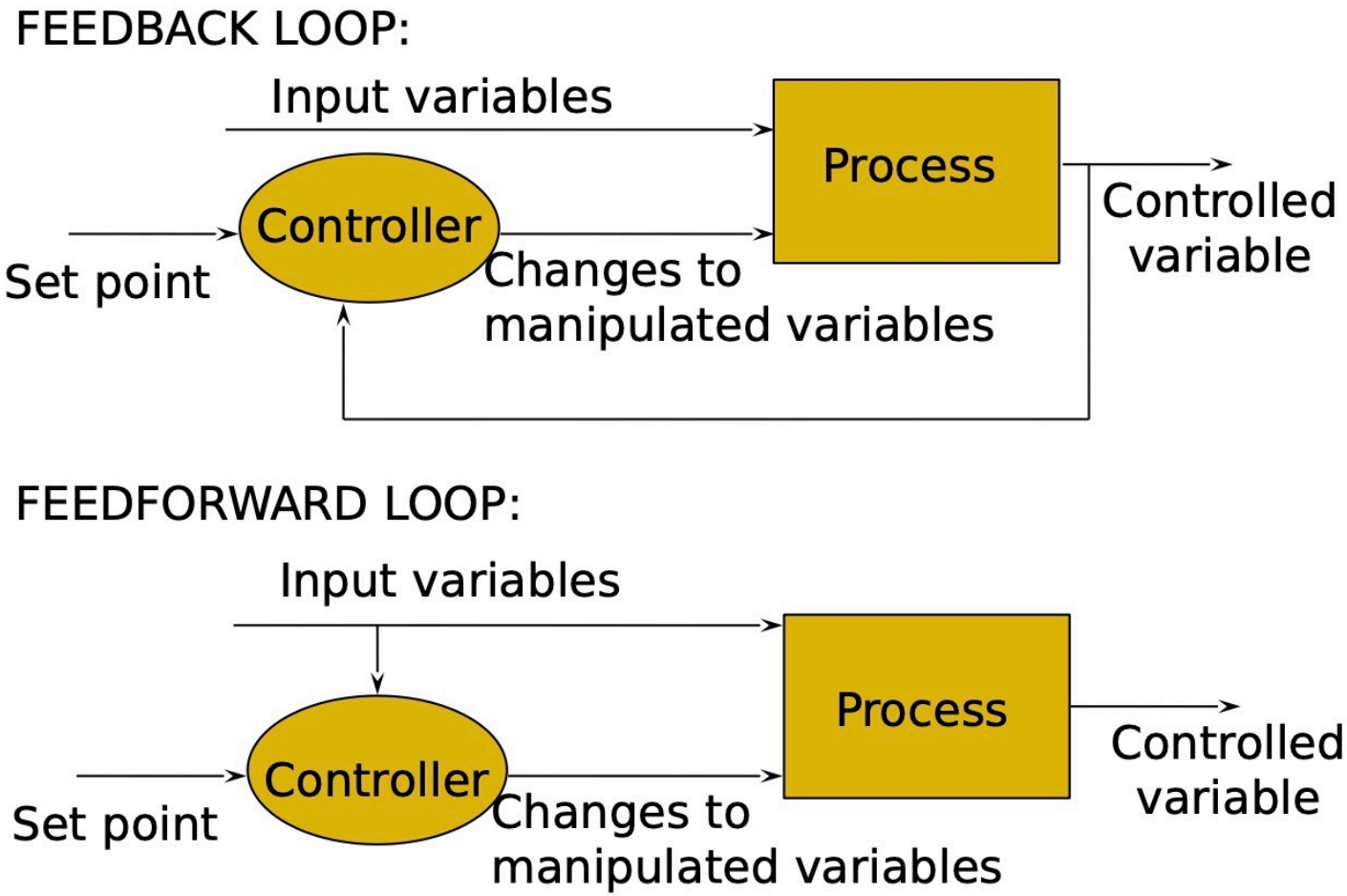
```
// Main program
main:
    // Create instances of model, view, and controller
    model = new Model()
    view = new View()
    controller = new Controller(model, view)

    // Process data through controller

controller.processData("New Data")

    // Display data using controller
    controller.displayData()
```

- **Some other architectures…**

Client and Server Architecture



Process Control Architecture



Some Other Architectures

And a lot of more….

Microservices Architecture
Event-Driven Architecture
Service-Oriented Architecture……

Design Rationale

Design Rationale

- Rationale is the justification of decisions
- Rationale is critical in two areas: it supports
  - ✓decision making and
  - ✓knowledge capture
- Rationale is important when designing or updating (e.g. maintaining) the system and when introducing new staff

Design Rationale

- Rationale includes：

✓ the issues that were addressed,
✓ the alternative proposals which were considered,
✓ the criteria used to guide decisions and
✓ the arguments developers went through to reach a decision
✓ the decision made for resolution of the issues

Design Rational Document Example

One fundamental issue in database design was database engine realization. The initial non-functional requirements on the database subsystem insisted on the use of an object-oriented database for the underlying engine. Other possible options include using a relational database, or a file system. An object-oriented database has the advantages of being able to handle complex data relationships and is fully buzzword compliant. On the other hand, object-oriented databases may be sluggish for large volumes of data or high-frequency accesses. Furthermore, existing products do not integrate well with CORBA, because that protocol does not support specific programming language features such as Java associations. Using a relational database offers a more robust engine with higher performance characteristics and a large pool of experience and tools o draw on. Furthermore, the relational data model integrates nicely with CORBA. On the downside, this model does not easily support complex data relationships. The third option was proposed to handle specific types of data that are written once and read infrequently. This type of data (including sensor readings and control outputs) has few relationships with little complexity and must be archived for extended period of time. The file system option offers an easy archival solution and can handle large amounts of data. Conversely, any code would need to be written from scratch, including serialization of access. We decided to use only a relational database, based on the requirements to use CORBA and in light of the relative simplicity of the relationships between the system's persistent data.

Design Rational Document Example

Issue: How to realize database engine?

Proposals:
- P1: use an Object Oriented database
- P2: use a relational database
- P3: use a file system

Arguments:
P1: use an Object Oriented database

A+ is able to handle complex data relationship.
A+ is fully buzzword compliant.
A- may be sluggish for large volumes of data or high-frequency accesses.
A- does not integrate well with CORBA.

Design Rational Document Example

P2: Use a relational DB

A+ offers a more robust engine with high performance characteristics.
A+ offers a large pool of experience and tools to draw on.
A+ integrates well with CORBA.
A- does not easily support complex data relationships.

P3: Use a file system
A+ handles data that are written once and read infrequently (including sensor readings and control outputs which have few relationships).
A+ is suitable for data that must be archived for long period of time.
A+ can handle large amounts of data.
A- needs to write code from scratch.

Design Rational Document Example

Criteria: Requirement to use CORBA

Resolution: Use a relational database (proposal P2), based on the criteria and in light of the relative simplicity of the system's persistent data relationships.

Summary

- Software Architecture Design
- Get familiar with some commonly used software architectures
- Using Design Rationale to document

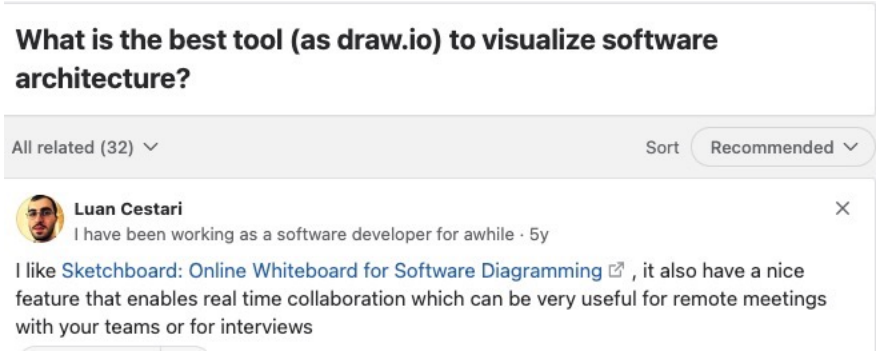
Recommended Reading

Software Engineering by Pressman (different editions)

Chapter: Requirements Modelling

Section: Creating a Behavioural Model

Software Architecture in Practice: Software Architect Practice_c3.
Bass, Len, Paul Clements, and Rick Kazman. Addison-Wesley, 2012



https://www.quora.com/What-is-the-best-tool-as-draw-io-to-visualize-software-architecture