CSE4006 Software Engineering

08. Architectural Design

Scott Uk-Jin Lee

Department of Computer Science and Engineering
Hanyang University ERICA Campus

1^{st} Semester 2015

Software Architecture

Software Architecture :

- provides fundamental description of a system
  - the components that make up the system
  - the significant collaboration between those components, including the data and control flows of the system
- provides a sound basis for analysis, decision making and risk assessment of both design and performance
- is an asset that constitutes tangible value to the organization that has created it

lab(se);

Purpose of Architecture

- The architecture is **not** the operational software
- The architecture is a representation that enables a software engineer to:
    1. analyze the **effectiveness of the design** in satisfying its stated requirements
    2. consider **architectural alternative** at a stage when making design changes is still relatively easy
    3. **reduce the risks** associated with the construction of the software

Importance of Architecture

- Representations of software architecture enables **communication** between all parties (stakeholders) interested in the development
- The architecture highlights early **design decisions**
  - earlier design decisions have a profound impact on all software engineering work that follows and on the ultimate success of the system
- Architecture provides **models to grasp** how the system is structured and how its components work together

Architectural Descriptions (AD)

- definition : "a collection of products to document an architecture" [IEEE Standard]
  - set of work products reflecting different stakeholder system viewpoints
  - views are representations of a whole system from the perspective of a related set of stakeholder concerns
  - viewpoints are specifications of conventions for constructing and using a view
- Recommended Practice for AD of Software-Intensive System [IEEE-Std-1471-2000]
  - to establish a conceptual framework and vocabulary for the design of software architecture
  - to provide detailed guidelines for representing an AD
  - to encourage sound architectural design practices

Architecture Description Language (ADL)

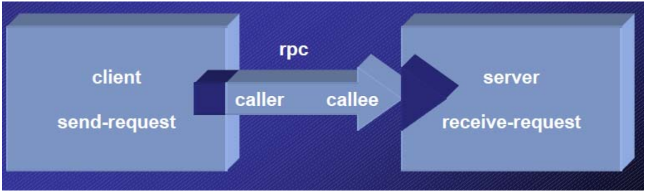
- provides a semantics and syntax for describing a software architecture
- provide the designer with the ability to:
  - decompose architectural components
  - compose individual components into larger architectural blocks
  - define interfaces (connection mechanisms) between components
- examples :
  - ABACUS (developed by UTS)
  - ACME (developed by CMU)
  - Rapide (developed by Stanford University)
  - Wright (developed by CMU)
  - ...

lab(se);

ADL Example (in ACME)

```
System simple_cs= {
    Component client = Port send-request
    Component server = Port receive-request
    Connector rpc= Roles caller, callee
    Attachments : client.send-request to rpc.caller;
                  server.receive-request to rpc.callee
}
```

simple_cs



Architectural Genre

- **Genre** : a specific category within the overall software domain

  e.g. artificial intelligence, communications, devices, financial, games, industrial, legal, medical, military, operating systems, transportation, utilities, ...

- within each category(genre),
  a number of subcategories(general style) also exists

  e.g. within the genre of building, the following general style exists : houses, condos, apartments, offices, warehouses, ...

  - within each general style, more specific styles might apply where each style have a structure that can be described using a set of predictable patterns

Architecture Style

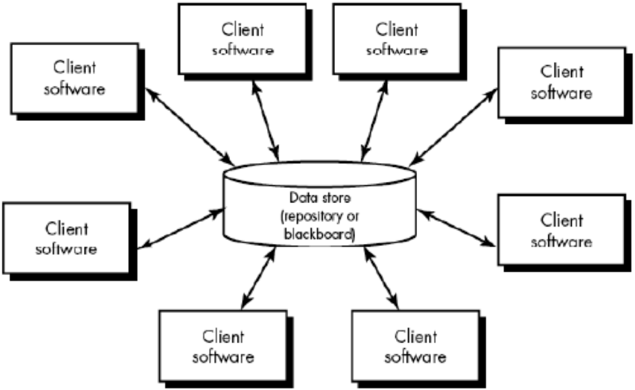
- Each style describes a system category that encompasses:
    1. **components** : a database or computational modules that perform a function required by a system
    2. **connectors** : enable communication, coordination and cooperation among components
    3. **constraints** : define how components can be integrated to form the system
    4. **semantic models** : enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts

Architecture Style

- **Data centered architectures** :
  - data store (file or database) lies at the center and is accessed frequently by other components (clients) that modify data
- **Data flow architectures** :
  - input data is transformed by a series of computational or manipulative components into output data
- **Layered architectures** :
  - several layers are defined, each accomplishing operations that progressively become closer to the machine instruction set
- **Call and return architectures** :
  - program structure decomposes function into control hierarchy with main program invokes several subprograms
- **Object-oriented architectures** :
  - components of system encapsulate data and operations, communication between components is by message passing

lab(se);

Data-Centered Architecture

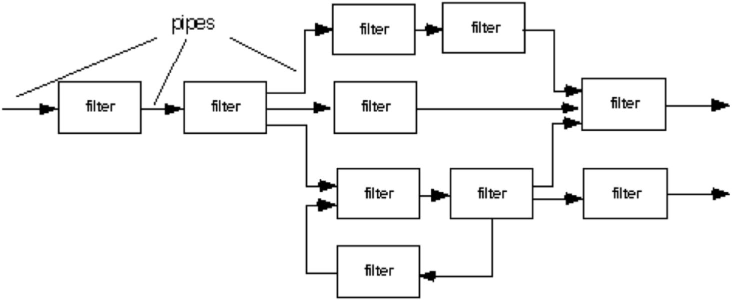


Data-Centered Architecture

- Data repository is
  - Passive
    - client software accesses the data independent of any changes to the data or the action of the other clients
  - Blackboard scheme
    - sends notifications to client software when data of interest to the client changes
- High integrability
  - Client components operate independently

Data Flow Architecture

Pipe-filter pattern



(a) pipes and filters

Batch sequential

(b) batch sequential

Pipes and Filters: Pattern

- A **data-flow architectural** pattern that views the system as a series of transformations on successive pieces of input data
- Pipes : **stateless** and serve as conduits for moving streams of data between multiple filters
- Filters : **stream modifiers**, which process incoming data in some specialized way and send that modified data stream out over a pipe to another filter

lab(se);

Pipes and Filters: Features

- Incremental delivery: data is output as work is conducted
- Concurrent (non-sequential) processing:
  - data flows through the pipeline in a stream, so multiple filters can be working on different parts of the data stream **simultaneously**
- Filters work **independently** and **ignorantly** of one another, and therefore are plug-and-play
- **Maintenance** is again isolated to individual filters, which are **loosely coupled**
- Very good at supporting producer-consumer mechanisms
- Multiple readers and writers are possible

Batch Sequential Data Processing

- Stand-alone programs would operate on data, producing a file as output
- The file would stand as input to another standalone program which would read the file in, process it, and write another file out
- Processing takes place **sequentially**
  - each process in **a fixed sequence** would run to completion, producing an output file in some new format, and then the next step would begin

lab(se);

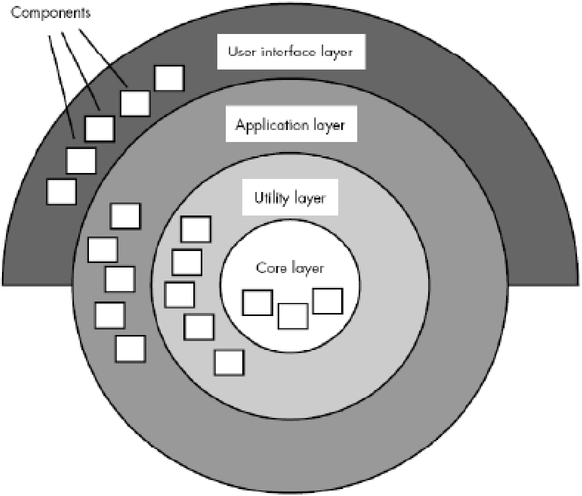
Pipes and Filters: Benefits

- Fairly **simple** to understand and implement
- Simple, **defined interface** reduces complex integration issues
- Filters are substitutable black boxes, and can be plug and played, and thus **reused** in creative ways
- Filters are **highly modifiable**
  - there's no coupling between filters and new filters can be created and added to an existing pipeline
- Filters and Pipes can be hierarchical and can be composed into a mechanism to further simplify client access
- Since filters stand alone, they can be distributed easily and support concurrent execution (the stream is in process)
- Multiple filters can be used to design larger complex highly-modifiable algorithms (by adding/deleting filters)

Pipes and Filters: Limitations

- A batch processing metaphor is not inherently limiting, but this pattern does **not facilitate highly dynamic responses to system interaction**
  - filters are black boxes, and are ignorant of one another, they cannot intelligently **reorder themselves dynamically**
  - once a pipeline is in progress, it cannot be altered without corrupting the stream
  - Difficult to configure **dynamic pipelines**, where depending on content, data is routed to one filter or another

Layered Architecture



Layers

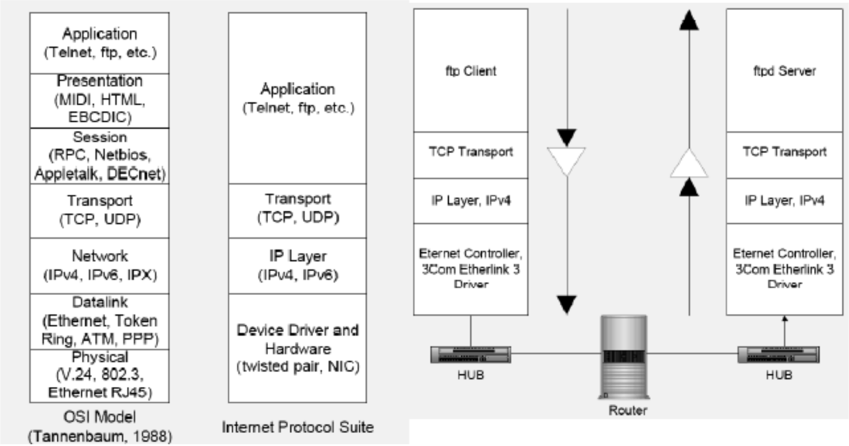
- Architectural layers are collaborating sections of an overall complex system that provide several benefits such as:
  - supporting incremental coding and testing, allowing localization of changes
  - **well-defined interfaces** allow substitution of different layers
  - protection between collaborating layers
  - layers support a responsibility-driven architecture that divides subtasks into groups of related responsibilities

Layers Pattern

- In the pure sense, each layer provides services to the layer **directly above it**, and acts as a client to the layer **directly below it**
- In an "impure" implementation, distanced layers can be "bridged" which allows communication between them but reduces portability and flexibility and plug and play capability
- Each layer provides a defined interface to the layers above and below it
- Higher layers provide increasing levels of **abstraction**

lab(se);

Example: Protocol Stacks



lab(se);

Layered Architecture: Benefits

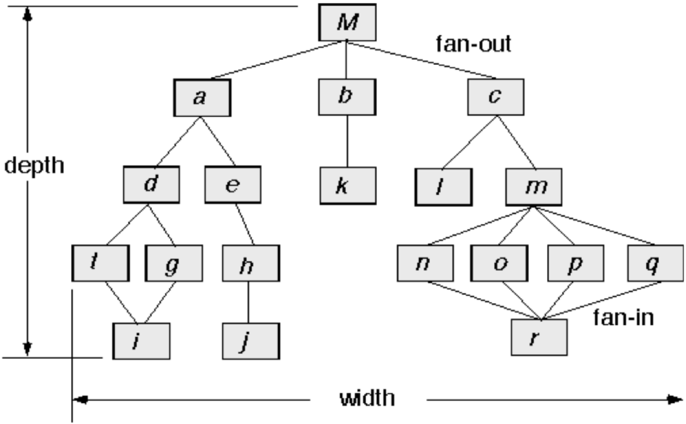
- A layered pattern supports increasing levels of abstraction, thus **simplifying design**
  - allows a complex problem to be partitioned into a sequence of manageable incremental strategies (as layers)
- Like Pipes and Filters, layers are **loosely coupled**
  - maintenance is enhanced because new layers can be added affecting only two existing components (as layers)
- Layers support **plug-and-play (=reusable)** designs
  - As long as the interfaces do not change, one layer can be substituted for another changing the behavior of the layer system

Layered Architecture: Disadvantages

- Close coupling of juxtaposed layers lowers maintainability
- Each layer must manage all data marshaling and buffering
- Lower runtime efficiency
- Sometimes difficult to establish the granularity of the various layers (10 layers or 4?)

lab(se);

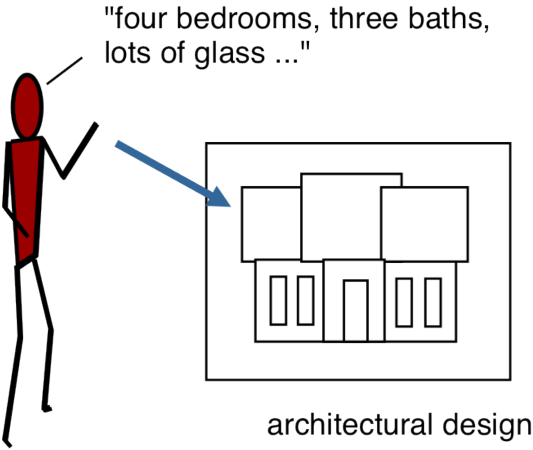
Call and Return Architecture



- main program / subprogram architecture
- remote procedure call architecture

An Architectural Design Method

customer requirements

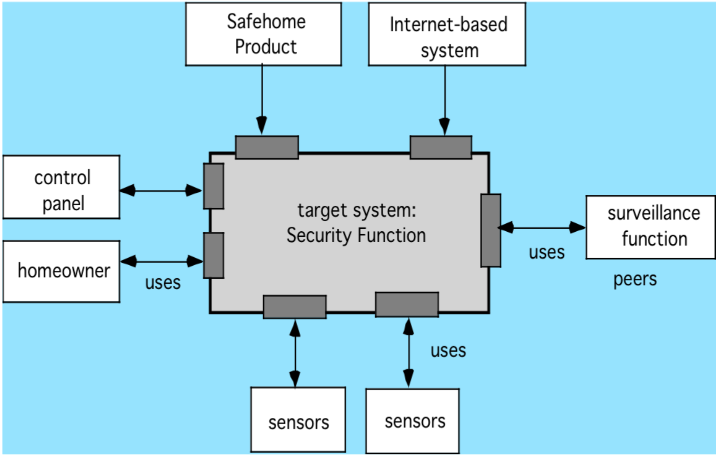


Architectural Design

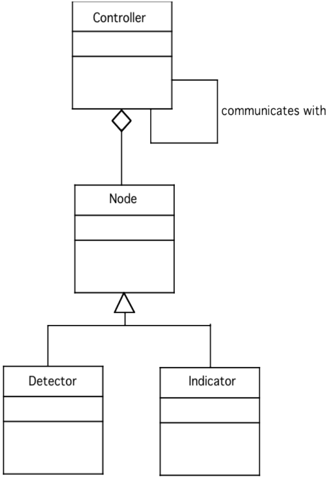
- The software must be placed into **context**
  - the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction (interface)
- A set of architectural archetypes should be identified
  - An **archetype** is an abstraction (similar to a class) that represents one element of system behavior
- The designer specifies the structure of the system by defining and refining software **components** that implement each archetype
- Continue the process iteratively until a complete architectural structure has been derived

lab(se);

Architectural Context

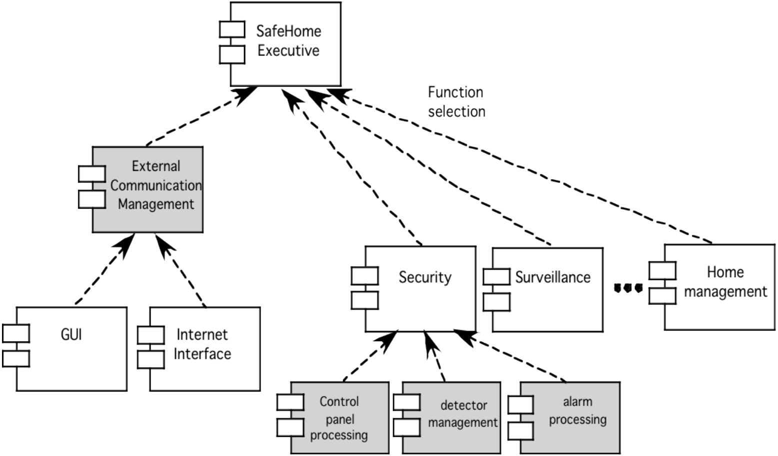


Archetypes

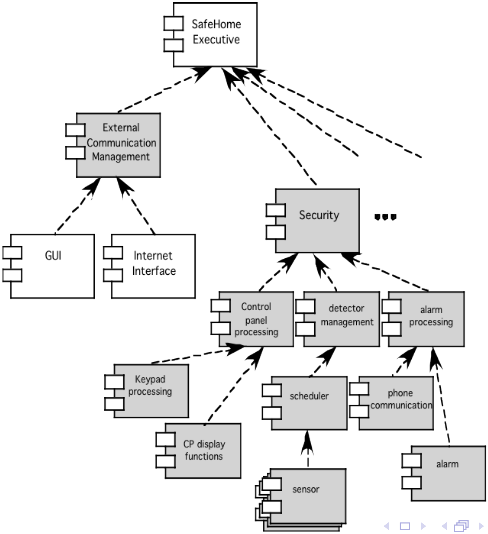


Component Structure

Top-level Component Structure



Refined Component Structure



Architectural Considerations

- **Economy** : the best software is uncluttered and relies on abstraction to reduce unnecessary detail
- **Visibility** : architectural decisions and the reasons for them should be obvious to software engineers who examine the model at a later time
- **Spacing** : separation of concerns in a design without introducing hidden dependencies
- **Symmetry** : architectural symmetry implies that a system is consistent and balanced in its attributes
- **Emergence** : emergent, self-organized behavior and control are often the key to creating scalable, efficient, and economic software architectures

Architectural Decision Documentation

1. Determine which information items are needed for each decision and its best representation
2. Define links between each decision and appropriate requirements
3. Provide mechanisms to change status when alternative decisions need to be evaluated
4. Define prerequisite relationships among decisions to support traceability
5. Link significant decisions to architectural views resulting from decisions
6. Document and communicate all decisions as they are made

Analyzing Architectural Design

Architecture Trade-off Analysis Method (ATAM) by SEI

1. Collect scenarios
2. Elicit requirements, constraints, and environment description
3. Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements:
   - module view / process view / data flow view
4. Evaluate quality attributes by considering each attribute in isolation (reliability, performance, maintainability, etc)
5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style
6. Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5

lab(se);

Architectural Complexity

: assessed by considering the dependencies between components within the architecture

- sharing dependencies
  - dependencies among consumers who use the same resource or producers who produce for the same consumers
- flow dependencies
  - dependencies between producers and consumers of resources
- constrained dependencies
  - constraints on the relative flow among a set of components

Architectural Reviews

- assess the ability of the software architecture to meet the systems quality requirements and identify potential risks
- have the potential to reduce project costs by detecting design problems early
- often make use of experience-based reviews, prototype evaluation, and scenario reviews, and checklists

Pattern-based Architectural Review

1. identify and discuss the quality attributes by walking through the use cases
2. discuss a diagram of system's architecture in relation to its requirements
3. identify the architecture patterns used and match the system's structure to the patterns' structure
4. using existing documentation and past use cases to determine the each pattern's effect on the system's quality attributes.
5. identify all quality issues raised by architecture patterns used in the design
6. develop a short summary of the issues uncovered during the meeting and make revisions to the walking skeleton

Agility and Architecture

- to avoid rework, user stories are used to create and evolve an architectural model (walking skeleton) before coding
- hybrid models which allow software architects contributing users stories to the evolving storyboard
- well run agile projects include delivery of work products during each sprint
- reviewing code emerging from the sprint can be a useful form of architectural review