Lecture 3

Software Architectures
Roadmap of the course

- What is software architecture?
- Designing Software Architecture
  - Requirements: quality attributes or qualities
  - How to achieve requirements: tactics
- Today:
  - How do tactics lead to architectural styles
  - Case studies on architectural styles
    - to also observe the achieved qualities
Elements of Architectural Descriptions

• The architectural definition of a system selects
  • **Components**: define the locus of computation
    • Examples: filters, databases, objects, ADTs
  • **Connectors**: mediate interactions of components
    • Examples: procedure calls, pipes, event broadcast
  • **Properties**: specify info for construction & analysis
    • Examples: signatures, pre/post conditions, RT specifications

• An architectural style defines a family of architectures constrained by
  • Component/connector vocabulary
  • Topology rules
  • Semantic constraints
Some Architectural Styles

• Data flow systems
  • Batch sequential
  • Pipes and filters

• Call-and-return systems
  • Main program & subroutines
  • Hierarchical layers
  • OO systems

• Virtual machines
  • Interpreters
  • Rule-based systems

• Independent components
  • Communicating processes
  • Event systems

• Data-centered systems (repositories)
  • Databases
  • Blackboards
Questions we ask of each style

• What is the design vocabulary?
• What are the allowable structural patterns?
• What is the underlying computational model?
• What are the essential invariants of the style?
• What are some common examples of its use?
• Advantages and disadvantages of using the style
• Common specializations of the style
Data Flow Systems

- The availability of data controls the computation
- The structure of the design is dominated by the orderly motion of data from process to process
- The pattern of data flow is explicit
- No other interaction between processes
- Types
  - Batch sequential
  - Pipes and filters
  - Feedback loop
Control flow vs Data flow

- Control flow
  - *Dominant question*: how the locus of control moves throughout the execution
  - Data may accompany control but is not dominant
  - *Reasoning is* on the computation order

- Data flow
  - *Dominant question*: how data moves through a collection of (atomic) computations
  - As data moves, control is activated
  - *Reasoning is* on data availability, transformation, latency
Batch Sequential Systems

- Processing steps are independent programs
- Each step runs to completion before next step starts
  - Atomic computations
- Data transmitted as a whole between steps
- Typical applications
  - Classical data processing
  - Program developments

validate → sort → update → report
Pipes and Filters

- Each component has
  - Set of inputs (read)
  - Set of outputs (produced)

- Filter
  - Incrementally transforms some amount of the data at inputs to data at outputs
    - Stream-to-stream transformations
    - Local transformation
  - Uses little local context in processing stream
  - *Independent, shares no state with other filters*
  - *Does not know identity of incoming and outgoing streams*
Pipes and Filters cont’d

• Pipe
  • Move data from a filter output to a filter input
  • Pipes form data transmission graphs

• Overall computation
  • Run pipes and filters (non-deterministically) until no more computations are possible
  • Correctness of p&f network’s output should not depend on ordering

• Specializations
  • Pipelines, bounded pipes, typed pipes
  • Batch sequential systems: degenerated pipeline
Pipe and filter examples

- Programs written in Unix shells
  - Filters: Unix processes
  - Pipes: runtime mechanisms for implementing them
- Compilers (exp of pipeline)
  - (phases often not incremental)
  - Lexical analysis, parsing, semantic analysis, code generation
- Signal processing domains
- Parallel programming
- Distributed programming
Advantages and disadvantages

- Adv
  - Overall I/O is a composition of the behavior of independent filters
  - Support reuse
  - Maintainability, improvement
  - Allow analysis (throughput, deadlock analysis)
  - Support concurrency

- Disadv
  - Often lead to a batch organization of processing
  - Not good at handling interactive applications
  - Quite complex
  - Not so performant
Call-and-Return systems

- The control moves from a module to another and back
- Different from *monolithic systems* or *pipe and filter systems*
- Abstraction made possible
- Types
  - Main program and subroutine
  - OO, ADT
  - (Hierarchical) layers
Main Program and Subroutines

- Hierarchical decomposition
  - Based on definition-use relationship
- Single thread of control
  - Supported directly by programming languages
- Subsystem structure implicit
  - Subroutines typically aggregated to modules
- Hierarchical reasoning
  - Correctness of a subroutine depends on the correctness of the subroutines it calls
Modularization Problems

- Access to internal representation:
  > Vulnerability: Visible representations can be manipulated in unexpected, undesired, and dangerous ways
  > Nonlocality: If the way something is used depends on how it’s implemented, you must find all uses to make a change

- Forced distribution of knowledge:
  > Non-uniform referents: Syntax may reveal structure (If you export a data structure, how does its user iterate through it?)

- Coupling:
  > Instance dependence: When multiple instances of a given structure are active, they must remain independent

- Families of definitions:
  > Dynamic binding: If shared definitions involve type variants, function variants must be chosen at run-time

Recurring Course Theme: Criteria for Modularization

- What is a module?
  > Common view: a piece of code. Too limited.
  > Compilation unit, including related declarations and interface
  > Parnas: a unit of responsibility. OK, but ...

- Why modularize a system, anyway?
  > Management: Partition the overall development effort
    » divide and conquer
  > Evolution: Decouple parts of a system so that changes to one part are isolated from changes to other parts
  > Understanding: Permit system to be understood as composition of mind-sized chunks

- Key issue: what criteria to use for modularization

Module Decomposition

- Parnas
  > Hide secrets. OK, what’s a “secret”?
    » Representation of data
    » Properties of a device, other than required properties
    » Implementation of world models
    » Mechanisms that support policies
  > Try to localize future change
    » Hide system details likely to change independently
    » Expose in interfaces assumptions unlikely to change
  > Use functions to allow for change
    » They’re easier to change than visible representation

10-Nov-11
Abstract data types and OO

- "If you get the data structures right, the rest of the program much simpler"
Object-orientation

- **Object**: collection of data and operations
- **Class**: description of set of objects
- **Subclass**: class with additional properties
  - More restrictive than class => fewer members
- **Instance**: object of a class
- **Method**: procedure body implementing operation
  - Dynamically bound
- **Message**: procedure call; request to execute method
- **Properties**: intuitive, support REUSE
Object architectures

- Encapsulation
  - Restrict access to certain information
- Object responsible for preserving integrity of its representation (invariant)
- Inheritance
  - Share one definition of shared functionality
- Dynamic binding
  - Determine actual operation to call at runtime
- Management of many objects
  - Provide structure on large set of definitions
- Reuse and maintenance
  - Exploit encapsulation and locality
Inheritance

- **Thing**
  - mass, location
  - metabolism, reproduction

- **Alive**
  - chlorophyll

- **Not-alive**
  - hemoglobin

- **Vegetable**

- **Animal**
  - fur
  - feathers
  - scales
  - walking upright

- **Mammal**
  - gnawing teeth

- **Bird**

- **Fish**

- **Rodent**

- **Primate**
  - pouch
  - mature live young

- **Kangaroo**

- **Human**
  - student loan
  - home loan

- **You**

- **Me**

- **Software Architectures**
Advantages and disadvantages

- **Advantages**
  - Maintainability: modifiability of method bodies
  - Architecture anticipates (some) changes
  - Reuse

- **Disadvantages**
  - Identity of interacting objects need to be known ->
    and needs to be changed in all objects interacting
    with an object whose identity was modified
  - Side effect problems
  - Managing many objects (additional structuring needed)
Layered Systems

- Each layer provides certain facilities
  - Hides part of lower layer
  - Provides well-defined interfaces
- Serves various functions
  - **Kernels**: provide core capability, often set of procedures
- Various scoping regimes
  - Opaque versus translucent layers
Layered Pattern

Usually procedure calls lead to Useful Systems, which provide Basic Utility at the Core Level. Composites of various elements interact with Users.
Advantages and disadvantages of layered systems

• Advantages
  • Abstraction (deals with complexity)
  • Modifiability
    • Changing one layer influences only the two adjacent layers
  • Reuse
    • Different implementations easy to substitute
    • Interfaces

• Disadvantages
  • Not all systems suitable for this
  • Performance may require other coupling
  • Abstraction quite hard
Interpreters

- Execution engine simulated in software
- Data
  - Representation of program being interpreted
  - Data (program state) of program being interpreted
  - Internal state of interpreter
- Control resides in “execution cycle” of interpreter
  - But simulated control flow in interpreted program resides in internal interpreter state
- Syntax-driven design
Rule-based systems

- Store and manipulate knowledge to interpret information in a useful way
- Especially used in artificial intelligence
- Exp
  - Doctor diagnoses based on symptoms
  - Game move deduced
- Components
  - List of rules or rule base
  - An inference engine
  - Temporary working memory
  - User interface
Simple Rule-Based System

- Inputs
- Working memory
- Triggers data
- Rule interpreter
- Updates
- Selected rule
- Rule and data element selection
- Selected data
- Knowledge base
- Rule memory
- Fact memory
- Rules and facts
- Outputs

Software Architectures
Communicating Processes

- Components: independent processes
  - Typically implemented as separate tasks
- Connectors: message passing
  - Point-to-point
  - Asynchronous and synchronous
  - RPC and other protocols can be layered on top
Event Systems

• Components: objects or processes
  • Interface defines a set of incoming procedure calls
  • Interface also defines a set of outgoing events

• Connectors: event-procedure bindings
  • Procedures are registered with events
  • Components communicate by announcing events at “appropriate” times
  • When an event is announced the associated procedures are (implicitly invoked)
  • Order of invocation is non-deterministic
  • In some treatments connectors are event-event bindings
Advantages and disadvantages of event systems

- **Advantages**
  - Support reuse: any component can be introduced in the system simply by registering it for the events of the system
  - Implicit invocation => easy evolution (modifiability)

- **Disadvantages**
  - No control over the overall computation
    - Component does not know what other components respond
    - Component may know who responds but does not know order
  - Data exchange
    - Can provoke performance problems
  - Reasoning about correctness: problematic
Classical Databases

- Central data repository
  - Schemas designed specifically for application

- Independent operators
  - Operations on database implemented independently, one per transaction type
  - Interact with database by queries and updates

- Control
  - Transaction stream drives operation
  - Operations selected on basis of transaction type
Multi-databases

Users

Client-Server

Mediators

Client-Server

Databases
The Blackboard Model

- Knowledge sources
  - World and domain knowledge partitioned into separate, independent computations
  - Respond to changes in blackboard

- Blackboard data structure
  - Entire state of problem solution
  - Hierarchical, nonhomogeneous
  - Only means by which knowledge sources interact to yield solutions

- Control
  - In model, knowledge sources self-activating
Repository (Blackboard)

- Direct access
- Computation
- Memory

Blackboard (shared data)

ks1 -> ks2 -> ks3
ks1 -> ks7 -> ks4
ks1 -> ks8
ks2 -> ks3
ks3
ks4
ks5
ks6
ks7
ks8
Some case studies
1. KWIC

- In his paper (1972) David Parnas proposed the following problem

  The KWIC (Key Word in Context) index system accepts an ordered set of lines; each line is an ordered set of words, and each word is an ordered set of characters. Any line may be “circularly shifted” by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

(On the Criteria to be Used in Decomposing Systems into Modules - discusses the important design decisions that impact how we modularize our software systems)
Architectural solutions for KWIC

- Shared data
- Abstract Data Types
- Implicit invocation
- Pipe-and-filter
KWIC design issues

- Changes in the processing algorithm
  - Line shifting on each line after it is read, on all lines after they are read or on demand
- Changes in the data representation
- Enhancement to system function
- Performance
- Reuse
Main Program/Subroutine with shared data

- Problem decomposed according to 4 basic functions
  - Input, shift, alphabetize, output
- Components coordinated by main program that sequences through them
- Data in shared storage
- Communication: unconstrained read-write protocol
  - Coordinator ensures sequential access to data
KWIC – shared data solution

**FIGURE 3.1** KWIC: Shared Data Solution
Shared data, pro and cons

- Advantages
  - Data can be represented efficiently
  - Intuitive appeal

- Disadvantages
  - Modifiability
    - Change in data format affects all components
    - Change in overall processing algorithm
    - Enhancements to system function
    - Reuse not easy to do
Abstract data types (ADT)

- Similar set of five modules, with interfaces
- Data is not shared by computational components
  - Accessed via interfaces
KWIC, ADT solution
ADT, pro and cons

• Advantages
  • Logical decomposition into processing modules similar to shared data
  • Algorithms/data can be changed in individual modules w/o affecting others
  • Better reuse (module has fewer assumptions about other modules)

• Disadvantages
  • Enhancing the function
    • Modify existing modules -> bad for simplicity, integrity
    • Add new modules -> performance penalties
Implicit invocation

- Shared data as the integration mechanism
- More abstract data interfaces
  - Data accessed as a list/set
- Computations invoked implicitly when data is modified
  - Line added -> event to shift module
  - Circular shifts produced in another shared data store -> event to alphabetizer, invoked
KWIC, implicit invocations
Implicit invocation, pro and cons

- Advantages
  - Functional enhancements easily
  - Data changes possible
  - Reuse

- Disadvantages
  - Difficult to ctrl processing order of implicitly invoked modules
  - Data representation uses more space
Pipe and filter

- Four filters
  - Input, shift, alphabetize, output
  - Process data and send it to the next
- Distributed ctrl
- Data sharing
  - Only the one transmitted on pipes
KWIC, pipes and filters
Pipe and filter, pros and cons

- **Advantages**
  - Maintains intuitive flow of processing
  - Reuse supported
  - New functions easily added
  - Amenable to modifications

- **Disadvantages**
  - Impossible to modify design to get interactive system
  - Data is copied between filters → space used inefficiently
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Shared data</th>
<th>ADT</th>
<th>Impl. invocation</th>
<th>Pipe and filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changes in the processing</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>algorithm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Changes in the data</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>representation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enhancement to system function</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Performance</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Reuse</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>
2. Instrumentation software

- Develop a reusable system architecture for oscilloscopes
  - Rely on digital technology
  - Have quite complex software
- Reuse across different oscilloscope products
  - Tailor a general-purpose instrument to a specific set of users
- Performance important
  - Rapid configuration of software within the instrument

=> Domain-specific software architecture
The Arena: Oscilloscopes

signals → waveforms → traces and measurements

Oscilloscope
Object-oriented model of software domain

- Clarified the data types used for oscilloscopes
  - Waveforms, signals, measurement, trigger modes, ...
- No overall model to explain how the types fit together
- Confusion about partitioning of functionality
  - Should measurements be associated with types of data being measured or represented externally?
  - Which objects should the user interface interact with?
Oscilloscope: O-O Approach

First attempt was an Object-Oriented Decomposition

- waveform
  - max-min wvfm
  - x-y wvfm
  - accumulate wvfm

waveform
- w: time-> voltage
- max: -> voltage
- min: -> voltage
- invert: ...
- add: ...

Result: Hundreds of classes, little structure, no overall pattern
Layered model of the oscilloscope

- Well-defined grouping of functions
- Wrong model for the application domain
  - Layer boundaries conflicted with the needs of the interaction among functions
    - The model suggest user interaction only via Visualization, but in practice this interaction affects all layers (setting parameters, etc)
Pipe-and-filter model

- Signal transformers used to condition external signals
- Acquisition transformers derive digitized waveforms from these signals
- Display transformers convert these waveforms into visual data
- Oscilloscope functions were viewed as incremental transformers of data
- Corresponds well with the engineers’ view of signal processing as a dataflow problem

Main problem:
- How should the user interact?
Oscilloscope: Pipe-Filter Approach

Third attempt was a Pipe-Filter Architecture

Result: A better model, but not clear how to model user input.
Modified pipe-and-filter model

- Each filter was associated with a control interface
  - Provides a collection of settings to be modified dynamically by the user
  - Explains how the user can make incremental adjustments to SW
  - Decouples signal-processing from user interface
- Signal-processing SW and HW can be changed without affecting the user interface as long as the control interface remains the same
Oscilloscope: Extended Pipe-Filter Approach

Fourth attempt was Pipe-Filter Architecture with Parameterized Filters.

Result: Elegant model, but not directly useful to implementors.
Modified pipe-and-filter model, more

• Further specialization
  • Pipe-and-filter lead to poor performance
    • Problems with internal storage and data exchange between filters
      • Waveforms have large internal storage => not practical for filters to copy waveforms every time they process them
    • Filters may run at radically different speeds
      • Not good to slow faster filter just to keep the pace with slower ones
  • Solution: several types of pipes (distinct colours)
    • Some allowed data processing w/o copying
    • Slow filters allowed to ignore incoming data when already processing other data
    • => the pipe/filter computations more tailorable
Oscilloscope: Solution

Pipe-Filter Architecture with Parameterized Filters and Colored Pipes

Result: Elegant model, and implementable.
Instrumentation software summary

- Case study shows
  - Some issues for developing architectures for industrial SW
  - Different styles => different effects on solution
- Software must be typically adapted from pure forms to specialized styles (domain specific)
- Here the result depended on properties of pipe-and-filter architecture adapted to satisfy the needs of the product family
3. Mobile Robotics

- The system controls a manned or partially manned vehicle
  - Car, submarine, space vehicle, ...
- Build software to control the robot
  - External sensors and actuators
  - Real-time
- Input provided by sensors
- Control the motion
- Plan the future path
Mobile Robotics con’t

- Complicating factors
  - Obstacles may block the path
  - Imperfect sensor input
  - Robot might run out of power
  - Accuracy in movement
  - Manipulation with hazardous material
  - Unpredictable events might lead to need of rapid response
Mobile Robotics con’t

- Consider four (4) architectural designs
  - Control loop
  - Layered design
  - Implicit invocation
  - Blackboard
Mobile Robotics con’t

- Design considerations
  - Req 1: deliberative and reactive behaviour
    - Coordinate robot actions with environment reactions
  - Req 2: uncertainty
    - The robot needs to act based on incomplete and unreliable information
  - Req 3: account for dangers
    - Fault tolerance, safety, performance
  - Req 4: flexibility
    - Application development requires experimentation and reconfiguration
Mobile Robotics con’t

- Requirements of different kind, application depends on complexity and predictability
  - Robot in another planet => fault tolerance
- The four requirements guide the evaluation of the four architectural alternatives
Solution 1: control loop

- A mobile robot uses a closed-loop paradigm
  - The controller initiates robot actions and monitors their consequences, adjusting plans
Solution 1 con’t

- The four requirements?
  - Req 1: deliberative and reactive behaviour
    + simplicity of paradigm
    - simplicity a problem in unpredictable environments
      - Implicit assumption: continuous changes in environment require continuous reaction
      - Robots face discrete events
      - Switch between behaviour modes - how to change between modes?
    - How to decompose the software into cooperating components?
Solution 1 con’t

- The four requirements?
  - Req 2: uncertainty
    - A trial-and-error process
  - Req 3: account for dangers
    + simplicity makes duplication easy
  - Req 4: flexibility
    + the major components (supervisor, sensors, motors) separate and replaceable
Solution 1 con’t

• Summary:
  • Paradigm appropriate for simple robotics
  • Can handle only a small number of external events
  • No really for complex decomposition of tasks
Solution 2: layered architecture

- Eight (8) levels:
  - Level 1: Robot control routines (motors, joints, …)
  - Levels 2&3: input from the environment
    - Sensor interpretation and integration
  - Level 4: robot’s model of the real world
  - Level 5: navigation
  - Levels 6&7: scheduling and planning of robot actions
    - Dealing with problems in level 7
  - Level 8: user interface and supervisory functions
Solution 2 con’t

The four requirements?

- Req 1: deliberative and reactive behaviour
  - More components to delegate tasks to
  - + indicates concerns that must be addressed
    - Sensor integration
  - + defines abstraction levels to guide the design
    - Robot ctrl vs navigation
  - - does not fit the actual data and control-flow patterns
  - - does not separate the data hierarchy (1-4) from the control hierarchy (1,5-8)
Solution 2 con’t

• The four requirements?
  • Req 2: uncertainty
    • + abstraction layers manage this
  • Req 3: account for danger
    • + managed by the abstraction mechanism: data and commands are analysed from different perspectives
      • Fault tolerance and passive safety ok; active safety not ok
  • Req 4: flexibility
    • - interlayer dependencies an obstacle
    • - complex relationships between layers can become difficult to manage
Solution 2 con’t

- Summary:
  - Provides a framework for organizing components
    - Precise about roles of layers
  - Problems when adding detail at implementation level
    - The communication pattern in a robot will not follow the scheme of the architecture
Solution 3: implicit invocation

• Task-control architecture
  • Based on hierarchies of tasks
    • Task trees
    • Parent tasks initiate child tasks
    • Software designer can define temporal dependencies between tasks
    • Dynamic reconfiguration of task trees at run time
  • Uses implicit invocation to coordinate interaction between tasks
    • Tasks communicate by multicasting messages via a message server (redirects msgs to tasks registered to handle them)
Solution 3 con’t

- TCA’s implicit invocation of msgs supports:
  - Exceptions: exception handling override tasks
    - Change processing mode
    - Can abort or retry tasks
    - Better suited for managing spontaneous events
  - Wiretapping: intercept messages by superimposed tasks
    - Safety-checks of outgoing commands
  - Monitors: read information and execute actions
    - Fault-tolerance issues using agents to supervise the system
Solution 3 con’t

• The four requirements?
  • Req 1: deliberative and reactive behaviour
    • + Separation of action and reaction via the task trees and exceptions, wiretapping and monitors
    • + concurrency explicit: multiple actions can proceed simultaneously and independently
      • - though in practice limited by the central message server
  ⇒ - relies on a central control point
Solution 3 con’t

• The four requirements?
  • Req 2: uncertainty
    • - not explicitly in the model
      • Maybe via task trees and exceptions
  • Req 3: dangers
    • + exception, wiretapping, monitors
    • + fault tolerance by redundancy
      • Multiple handlers registered for same signal, one fails another takes over
      • Multiple occurrences of same request: concurrently
  • Req 4: flexibility
    • + implicit invocation allows incremental development and replacement of components
      • Often sufficient to register new handlers in central server
Solution 3 con’t

- Summary:
  - TCA offers a comprehensive set of features for coordinating tasks
  - Appropriate for complex robot projects
Solution 4: blackboard

- Based on the following components:
  - **Captain**: overall supervisor
  - **Map navigator**: high-level path planner
  - **Lookout**: monitors the environment
  - **Pilot**: low-level path planner and motion controller
  - **Perception subsystem**: raw input from sensors and its integration to an interpretation
Solution 4 con’t

• The four requirements?
  • Req 1: deliberative and reactive behaviour
    • + components interact via the shared repository
    • - control flow must be coerced to fit the database mechanism
      • Components do not communicate directly
  • Req 2: uncertainty
    • + blackboard the means for resolving conflicts and uncertainties
      • All data available in the database
Solution 4 con’t

• The four requirements?
  • Req 3: account for dangers
    • + communication via a central service, the database
      • Exception handling, wiretapping, monitors can be implemented by adding modules that watch the database for certain signs of problematic situations
  • Req 4: flexibility
    • + Supports concurrency
    • + Decouples senders from receivers
      • Facilitates maintenance
Solution 4 con’t

• Summary:
  • The architecture is capable of modelling the cooperation of tasks
    • Coordination
    • Resolving uncertainty
  • Slightly less powerful than TCA
  • Not the only possibilities for robotics …
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Control loop</th>
<th>Layers</th>
<th>Impl. invocation</th>
<th>Blackboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task coordination</td>
<td>+ -</td>
<td>-</td>
<td>++</td>
<td>+</td>
</tr>
<tr>
<td>Dealing with uncertainty</td>
<td>-</td>
<td>+ -</td>
<td>+ -</td>
<td>+</td>
</tr>
<tr>
<td>Fault tolerance</td>
<td>+ -</td>
<td>+ -</td>
<td>++</td>
<td>+</td>
</tr>
<tr>
<td>Safety</td>
<td>+ -</td>
<td>+ -</td>
<td>++</td>
<td>+</td>
</tr>
<tr>
<td>Performance</td>
<td>+ -</td>
<td>+ -</td>
<td>++</td>
<td>+</td>
</tr>
<tr>
<td>Flexibility</td>
<td>+ -</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
Today’s take away

- A basic collection of architectural styles is necessary
- Small case studies are useful for illustrative purposes
- Same (rather basic) ideas come back in our field
  - Dropbox
  - Cloud computing
  - Wikipedia