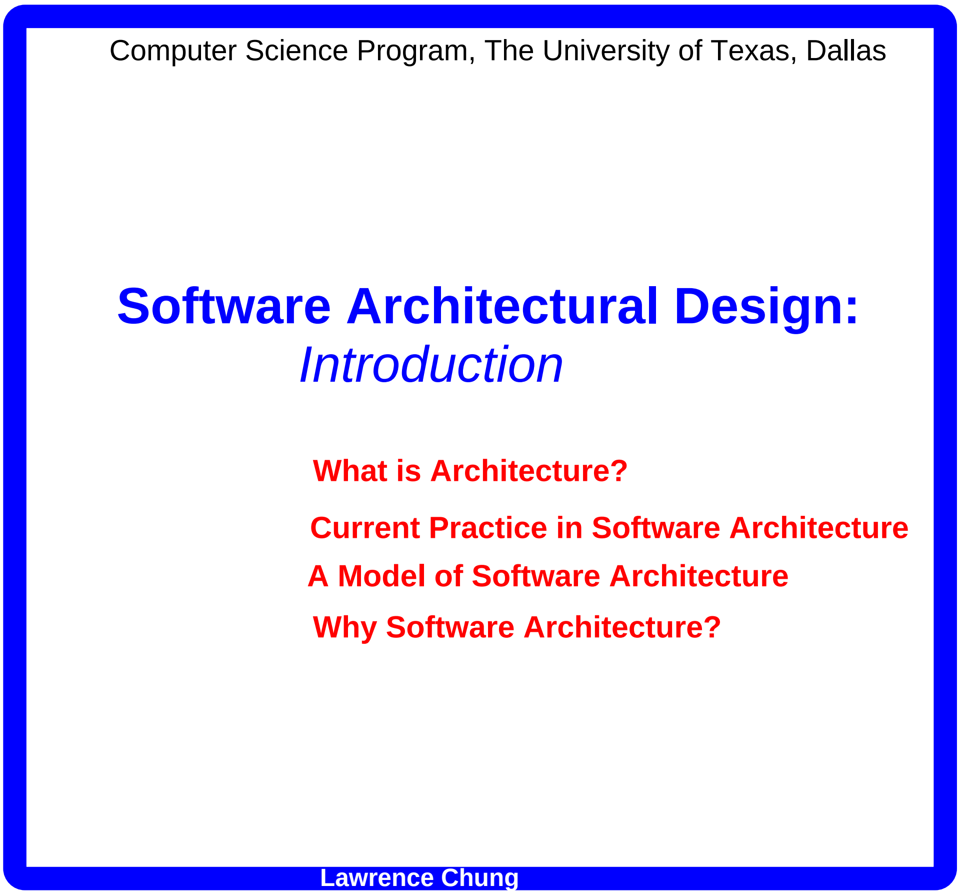


Computer Science Program, The University of Texas, Dallas

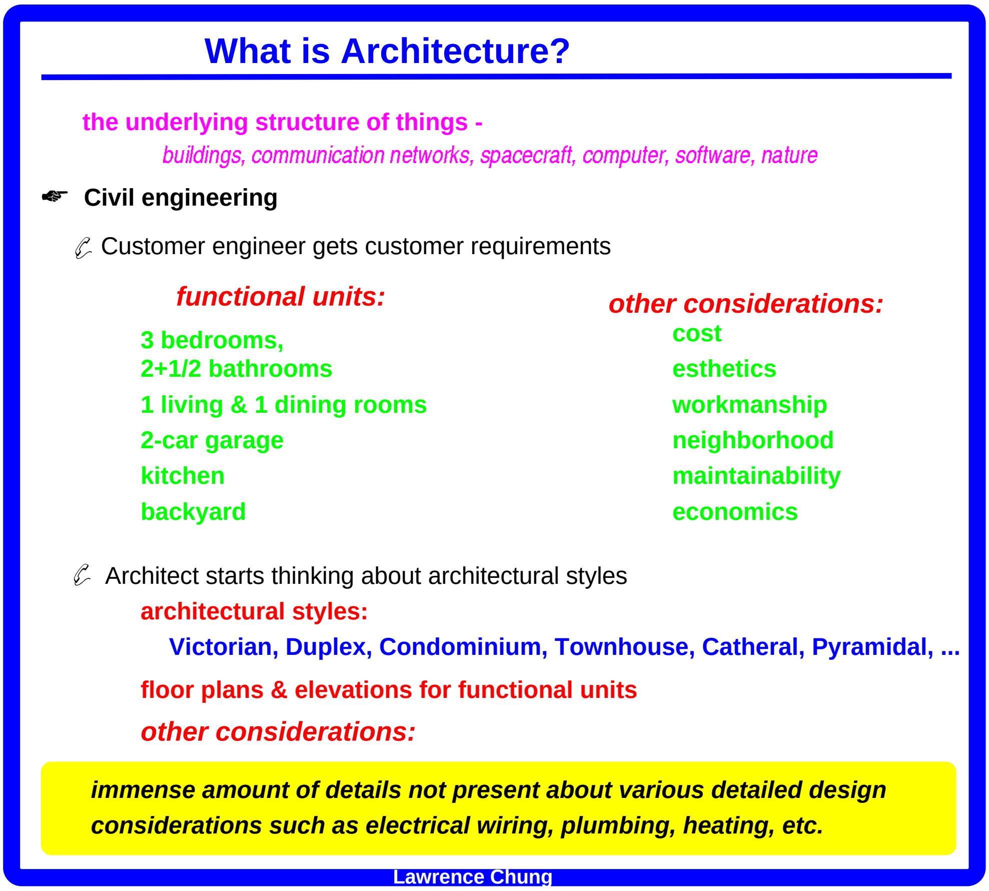
Software Architectural Design:
Introduction

What is Architecture?

Current Practice in Software Architecture

A Model of Software Architecture

Why Software Architecture?



What is Architecture?

the underlying structure of things -

buildings, communication networks, spacecraft, computer, software, nature

☛ **Civil engineering**

✎ Customer engineer gets customer requirements

functional units:	***other considerations:***
3 bedrooms, 2+1/2 bathrooms	**cost**
1 living & 1 dining rooms	**esthetics**
2-car garage	**workmanship**
kitchen	**neighborhood**
backyard	**maintainability**
	economics

✎ Architect starts thinking about architectural styles

architectural styles:

Victorian, Duplex, Condominium, Townhouse, Catheral, Pyramidal, ...

floor plans & elevations for functional units

other considerations:

immense amount of details not present about various detailed design considerations such as electrical wiring, plumbing, heating, etc.

What is Architecture?

the underlying structure of things -

buildings, communication networks, spacecraft, computer, software, nature

☛ **Civil engineering**

- Designers/Contractors think about detailed design considerations

  electrical wiring, plumbing, heating, air-conditioning, carpeting, etc.

- Sub-contractors/Construction Workers:

  electricians, plumbers, furnace installers, carpenters, locksmith, brick layers, bathtub technicians, etc.

Reading Assignment: Chapter 1

Current Practice in Software Architecture

❞ Camelot is based on the client-server model and uses remote procedure calls both locally and remotely to provide communication among applications and servers.

client-server model ***-> what? -> clients (applications) & server(s) as components***

RPCs both locally and remotely ***-> what -> communication/interaction mechanism***

But,

Why client-server model?

distributed data? distributed processing? cooperative processing?

What's a client like?

a terminal emulator? + a domain-specific application?

What's a server like?

a file server? a db server? a transaction server? a CGI server? a groupware?

Why RPCs?

why not sockets? why not MOMs? why not events?

What's communicated?

data? metadata? control? process? object? multimedia? agent?

Any constraint?

like passive Web browser? like client-centric Java?
like server-centric CGI? like CORBA? like OLE(2)/(D)COM?
uni-/bi-directional communication? multi-paradigm?
multi-platform?

Good software developers have often adopted one or several architectural patterns but informally and often only implicitly

Current Practice in Software Architecture

”

Abstraction layering and system decomposition provide the appearance of system uniformity to clients, yet allow Helix [distributed file system] to accommodate a diversity of autonomous devices. The architecture encourages a client-server model for the structuring of applications.

abstraction layering and system decomposition
client-server model ***-> what? -> clients (applications) & server(s) as components***

uniform appearance, accommodate a diversity of autonomous devices
for the structuring of applications.

-> why-> rationale

But,

Why client-server model?
distributed data? distributed processing? cooperative processing?

What's a client like?
a terminal emulator? + a domain-specific application?

What's a server like?
a file server? a db server? a transaction server? a CGI server?
a groupware?

What's the communication mechanism?

sockets? MOMs? events?

What's communicated?
data? metadata? control? process? object? multimedia? agent?

Any constraint?
like passive Web browser? like client-centric Java?
like server-centric CGI? like CORBA? like OLE(2)/(D)COM?
uni-/bi-directional communication? multi-paradigm?
multi-platform?

Good software developers have often adopted one or several architectural patterns
but informally and often only implicitly

Current Practice in Software Architecture

”

We have chosen a distributed, object-oriented approach to managing information.

Observations

- Software architectures are indeed used, very often but without even knowing it

  It's natural to use software architectures!

- carries some, and more often than not a lot of, information

  Care must be exercised!

- no explicit description of the structure

  No clear basis for communication or reasoning!

Good software developers have often adopted one or several architectural patterns
but informally and often only implicitly

A Model of Software Architecture

Software architecture:

- ***elements (components/parts):***

  from which systems are built
  e.g., process, data, object, agent

- ***interactions (connections/connectors/glues/relationships):***

  between the elements
  e.g., PCs, RPCs, MOMs, events

- ***patterns:***

  describe layout of elements and interactions, guiding their composition
  e.g., # of elements, # of connectors, order, topology, directionality

- ***constraints:***

  on the patterns (i.e., on components, connectors, layout)
  e.g., temporal, cardinality, concurrency, (a)synchronous, etc.

- ***styles:***

  abstraction of architectural components from various specific architectures.
  (Sometimes interchangeably used with patterns)
  e.g., Unix OS, OSI protocol layer, Onion ring IS structure -> layering

- ***rationale:***

  describe why the particular architecture is chosen

A Model of Software Architecture

Example: Sequential Compiler

elements	*interactions*	*patterns:*
	source code (characters) a+ x * (1-1) +7	connector (stream of data)
Lexer		**process**
	tokens (name table) a plus x mult lParen 1 minus 1 rParen plus 7	(stream of data)
Parser		**process**
	phrases (name table & abstract syntax tree) a plus [x mult [1 minus 1]] plus 7	
Semantic Analyzer		**process**
	correlated phrases (name table & abstract syntax graph) a plus [x mult [1 minus 1]] plus 7	
Optimizer		**process**
	(annotated) correlated phrases (name table & annotated abstract syntax graph) a plus 7	
Coder		**process**
	load a; load 7; add	

A Model of Software Architecture

Example: Sequential Compiler

elements	*interactions*	*patterns:*
	source code (characters) a+ x * (1-1) +7	connector (stream of data)
Lexer		**process**
	tokens (name table) a plus x mult lParen 1 minus 1 rParen plus 7	(stream of data)
Parser		**process**
	phrases (name table & abstract syntax tree) a plus [x mult [1 minus 1]] plus 7	
Semantic Analyzer		**process**
	correlated phrases (name table & abstract syntax graph) a plus [x mult [1 minus 1]] plus 7	
Optimizer		**process**
	(annotated) correlated phrases (name table & annotated abstract syntax graph) a plus 7	
Coder		**process**
	load a; load 7; add	

{connector process}* connector

style: pipe&filter

each element does a local transformation to the input and produces output

each glue serves as a conduit for the data stream,
transmitting outputs of one process to inpts of another

constraints:

processes do not share state with other processes

processes do not know the identity of their upstream and downstream processes
(partial concurrency, or complete degenerate case)

=> Independent processes (other than stream availability)

rationale: simplicity, process independence

A Model of Software Architecture

Points to ponder about:

❐ What are disadvantages (& other advantages) of this architecture?

Time, Space, Reusability, Adaptability, etc.

❐ What alternative architectures are possible?

Lexer + Parser

2 Semantic Analyzers (forward reference)

Shared data + sequential

No Optimizer

Concurrent compiler (semantic analyzer || optimizer || coder)

❐ What are some other instances of this style?

Unix command processing: e.g., ls|sort|pr|lpr

Common Architectural Styles

- ***Dataflow systems*** [topic 5: Data Flow]
  - *Batch sequential*
  - *Pipe & Filter*
- ***Call-and-return systems***
  - *Main program & subroutine* [topic 4: Modular Decomposition Issues]
  - *OO systems* [topic 3: ADT]
  - *Hierarchical layers* [topic 5 & 6 & 10 - Data Flow & Repositories & Middleware]
- ***Independent components***
  - *Communicating processes* [topic 11?: Processes]
  - *Event systems* [topic 4 & 7 - Modular Decomposition Issues & Events]
- ***Virtual machines***
  - *Interpreters*
  - *Rule-based systems*
- ***Client-server*** [topic 9]
- ***Data-centered systems*** [topic 6: Repositories]
  - *Databases*
  - *Hypertext systems*
  - *Blackboards*
- ***Process-control paradigms*** [topic 8: Repositories]

Why Software Architecture?

- *Abstract solution to conquer complexity*

  functionality and performance (Non-functional requirements)

  divide and conquer

- *A shared, semantically-rich vocabulary between SEeers.*

  E.g., instanceOf (X, pipe & filter)

  =>

  X is primarily for stream transformation

  functional behavior of X can be derived compositionally from the behaviors of the constituent filters

  issues of system latency and throughput can be addressed in relatively straightforward ways

- *supports reuse*
- *facilitates (integration) testing*
- *parallel development*
- *system evolvability*

... and many other conceptual reasons