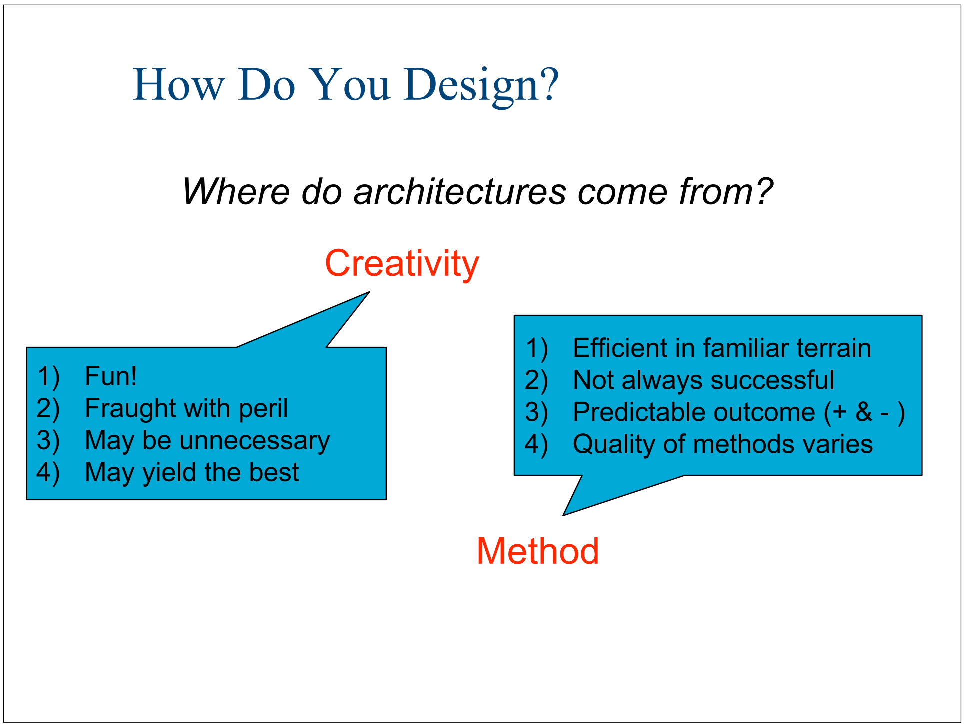


How Do You Design?

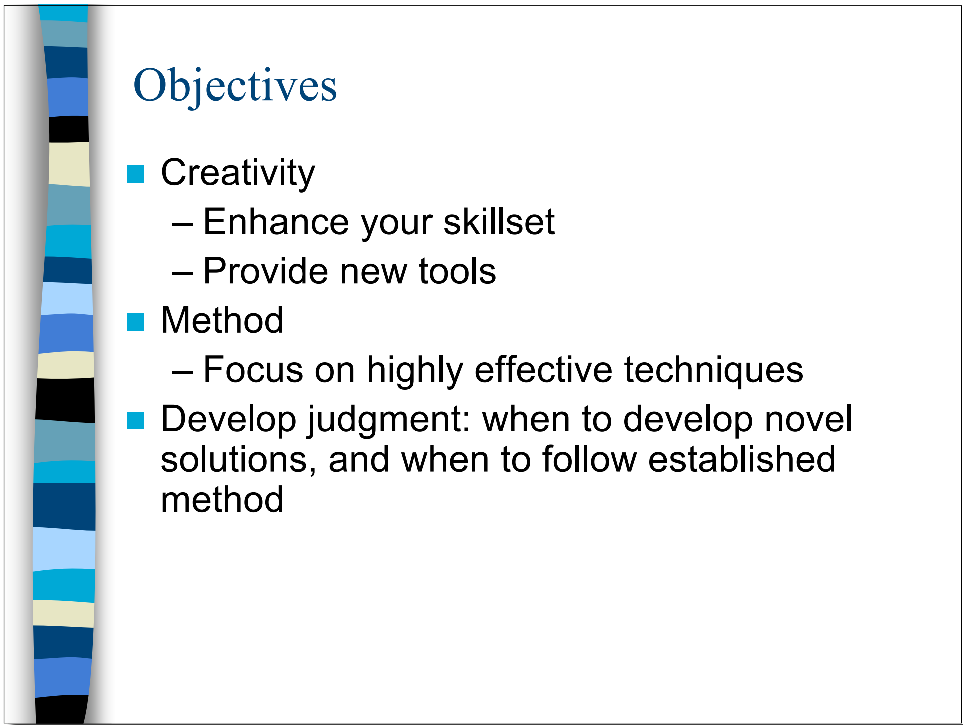
Where do architectures come from?

Creativity

1) Fun!
2) Fraught with peril
3) May be unnecessary
4) May yield the best

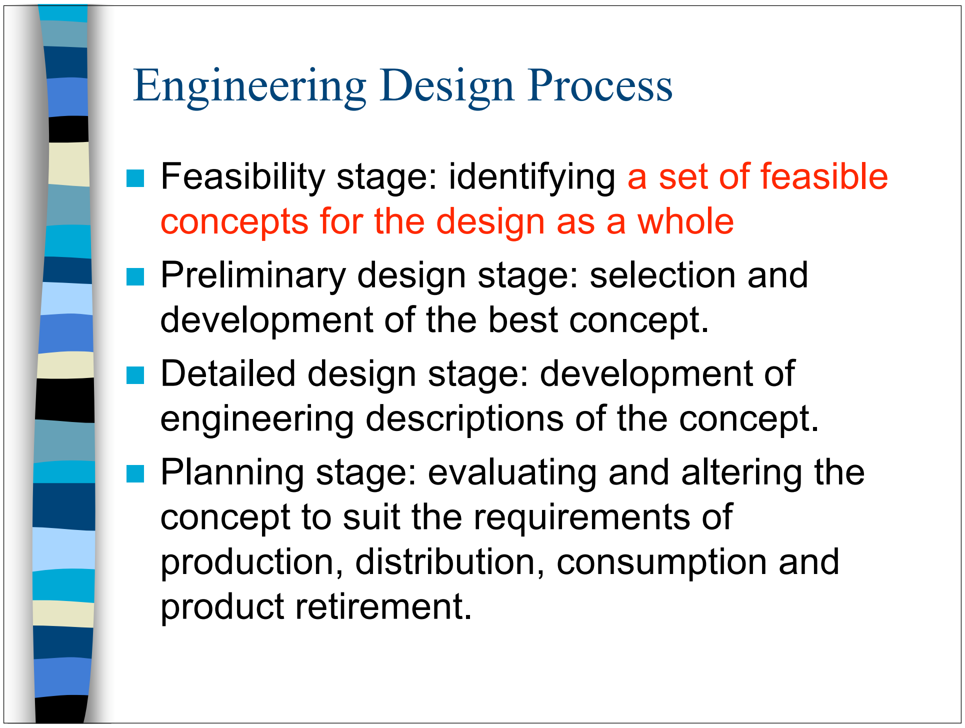
1) Efficient in familiar terrain
2) Not always successful
3) Predictable outcome (+ & -)
4) Quality of methods varies

Method



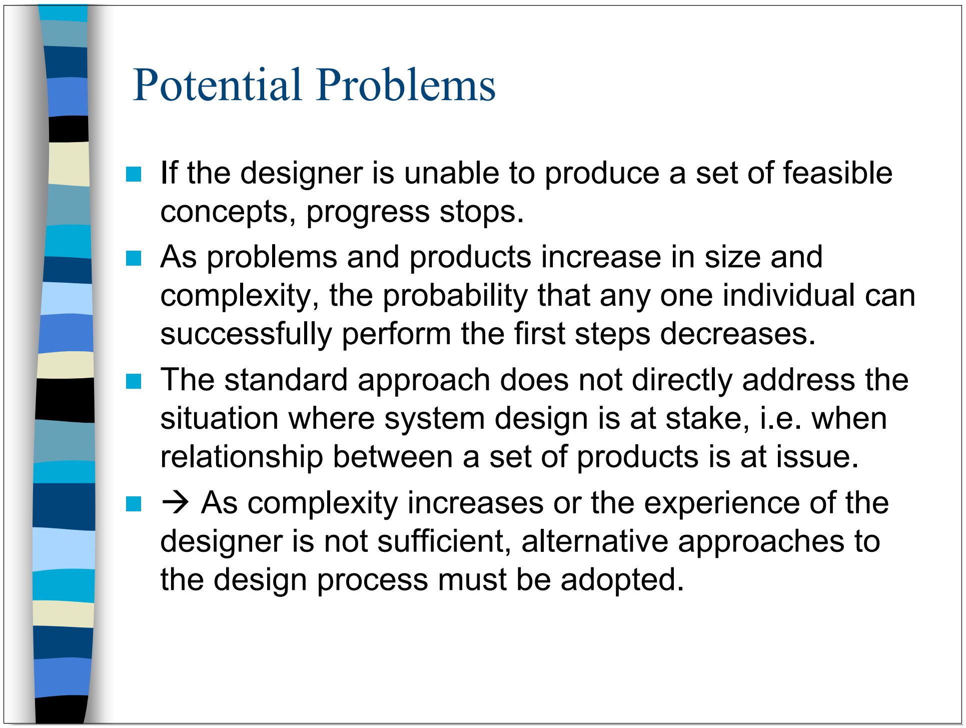
Objectives

- Creativity
  - Enhance your skillset
  - Provide new tools
- Method
  - Focus on highly effective techniques
- Develop judgment: when to develop novel solutions, and when to follow established method



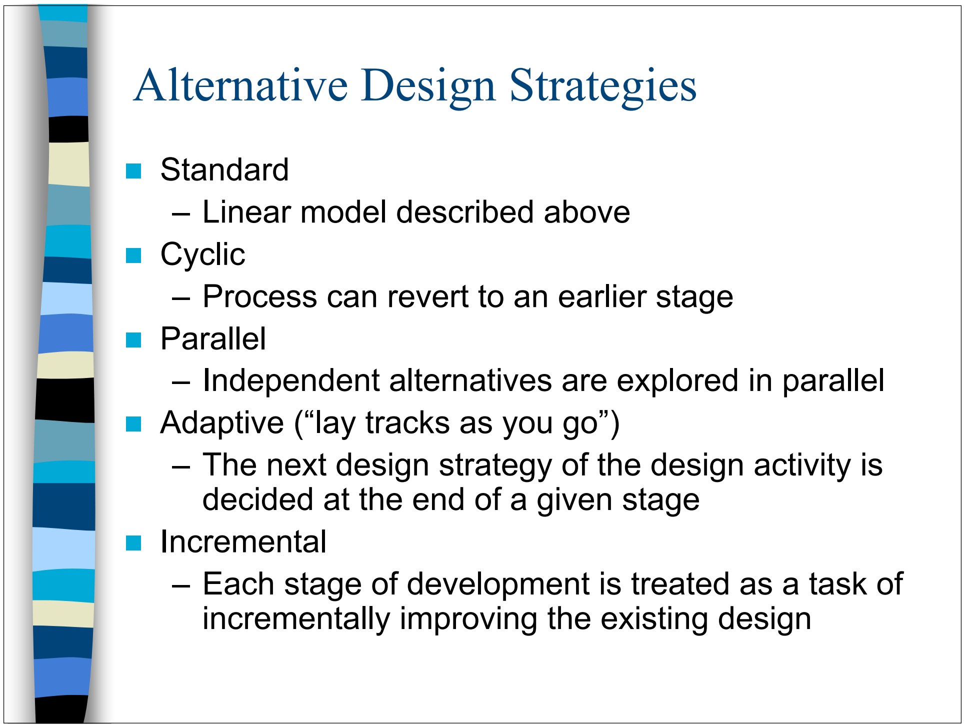
Engineering Design Process

- Feasibility stage: identifying a set of feasible concepts for the design as a whole
- Preliminary design stage: selection and development of the best concept.
- Detailed design stage: development of engineering descriptions of the concept.
- Planning stage: evaluating and altering the concept to suit the requirements of production, distribution, consumption and product retirement.



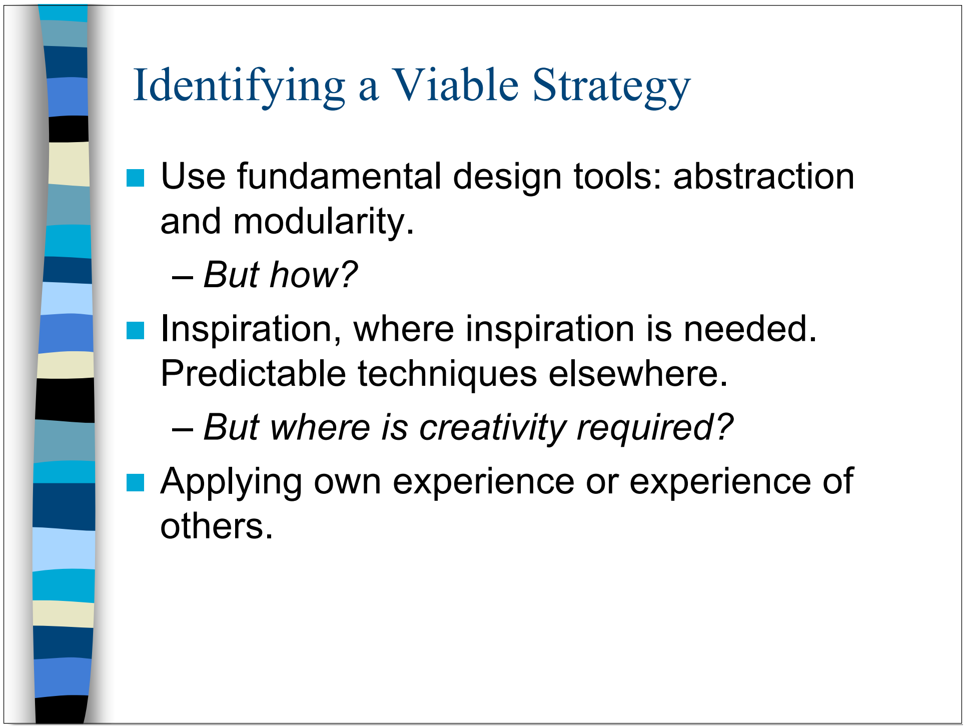
Potential Problems

- If the designer is unable to produce a set of feasible concepts, progress stops.
- As problems and products increase in size and complexity, the probability that any one individual can successfully perform the first steps decreases.
- The standard approach does not directly address the situation where system design is at stake, i.e. when relationship between a set of products is at issue.
- → As complexity increases or the experience of the designer is not sufficient, alternative approaches to the design process must be adopted.



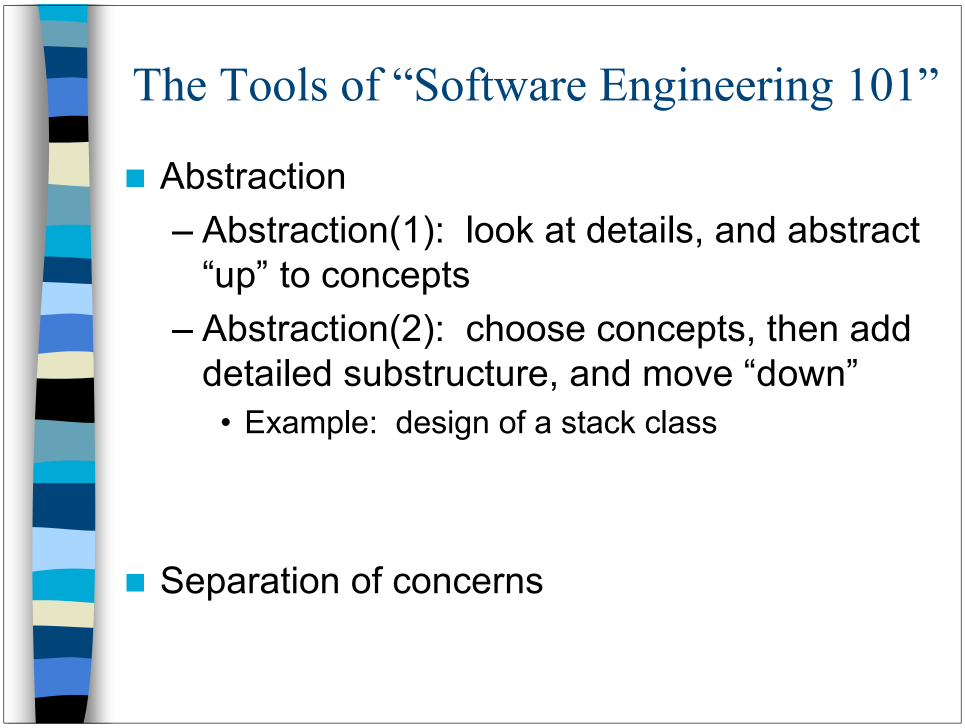
Alternative Design Strategies

- Standard
  - Linear model described above
- Cyclic
  - Process can revert to an earlier stage
- Parallel
  - Independent alternatives are explored in parallel
- Adaptive (“lay tracks as you go”)
  - The next design strategy of the design activity is decided at the end of a given stage
- Incremental
  - Each stage of development is treated as a task of incrementally improving the existing design



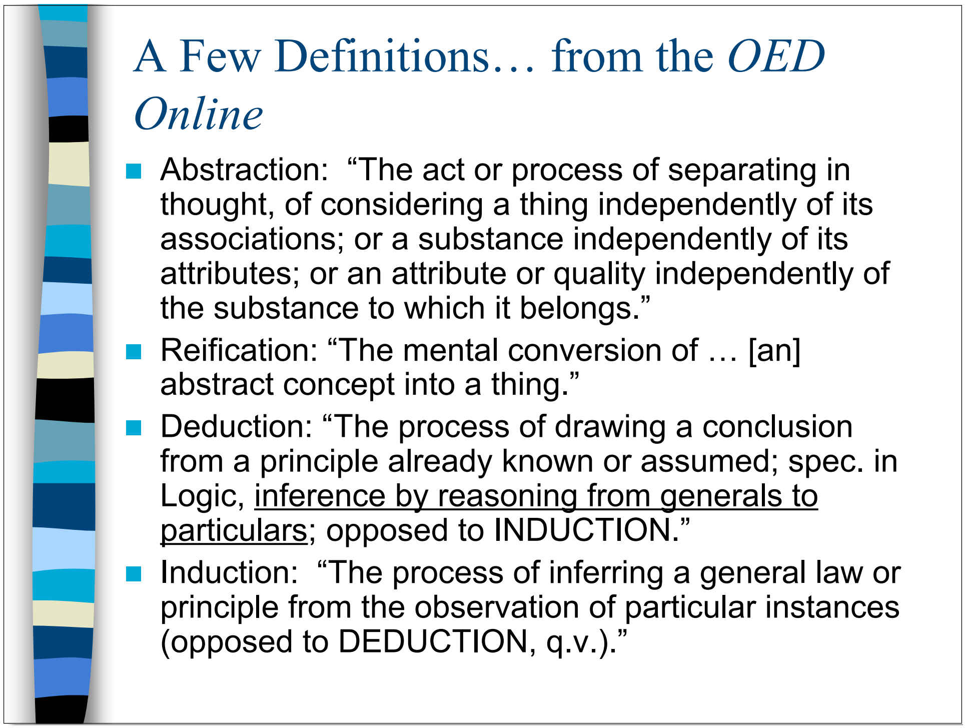
Identifying a Viable Strategy

- Use fundamental design tools: abstraction and modularity.
  - *But how?*
- Inspiration, where inspiration is needed. Predictable techniques elsewhere.
  - *But where is creativity required?*
- Applying own experience or experience of others.



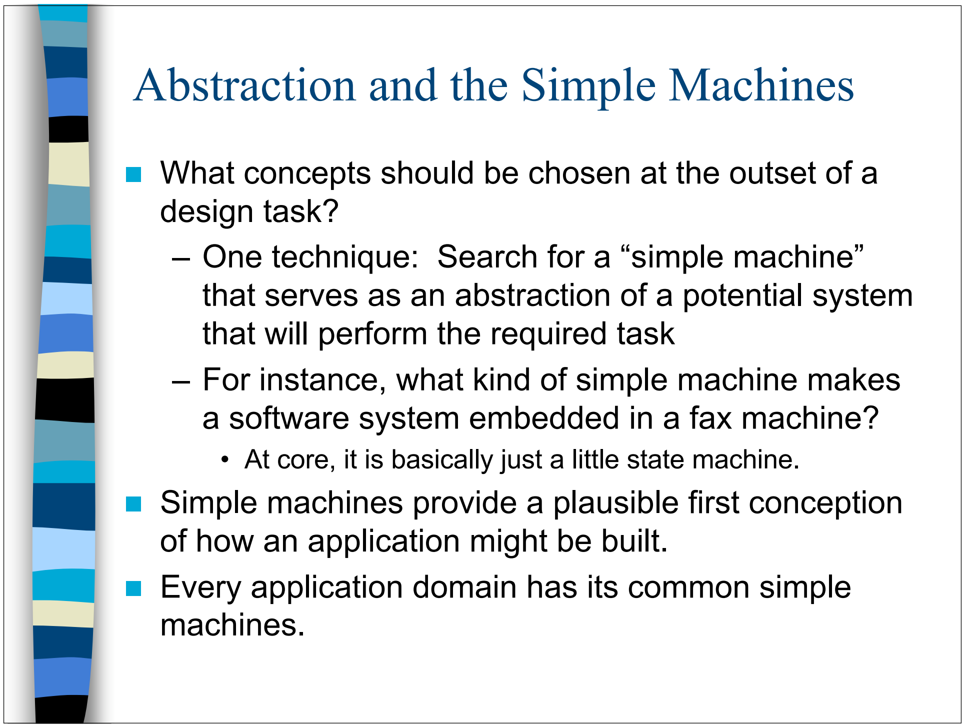
The Tools of “Software Engineering 101”

- Abstraction
  - Abstraction(1): look at details, and abstract “up” to concepts
  - Abstraction(2): choose concepts, then add detailed substructure, and move “down”
    - Example: design of a stack class
- Separation of concerns



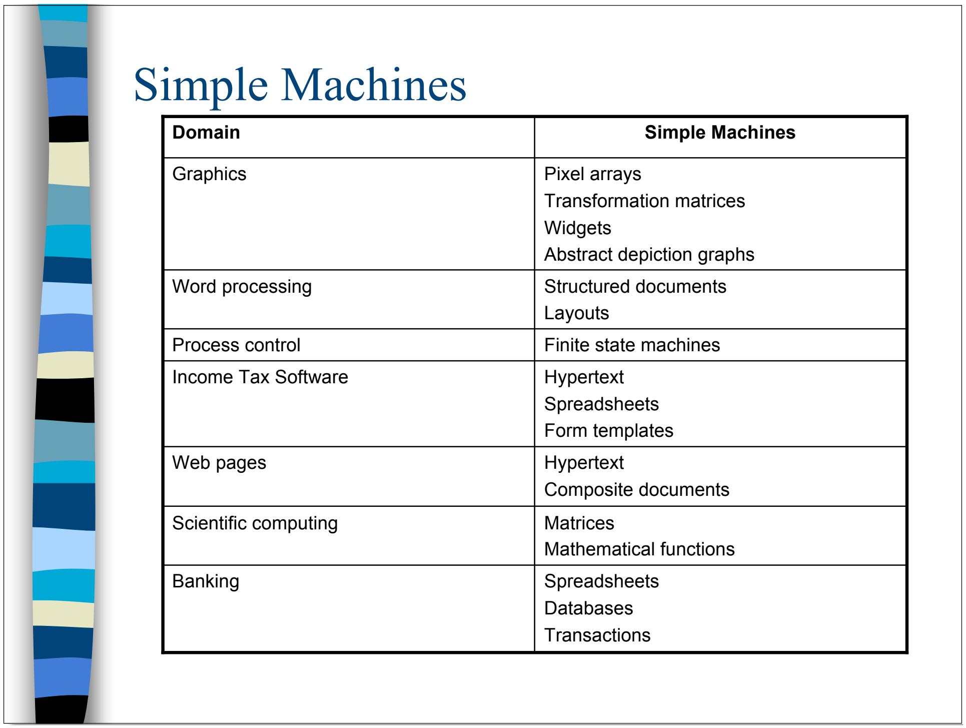
A Few Definitions… from the *OED Online*

- Abstraction: “The act or process of separating in thought, of considering a thing independently of its associations; or a substance independently of its attributes; or an attribute or quality independently of the substance to which it belongs.”
- Reification: “The mental conversion of … [an] abstract concept into a thing.”
- Deduction: “The process of drawing a conclusion from a principle already known or assumed; spec. in Logic, inference by reasoning from generals to particulars; opposed to INDUCTION.”
- Induction: “The process of inferring a general law or principle from the observation of particular instances (opposed to DEDUCTION, q.v.).”



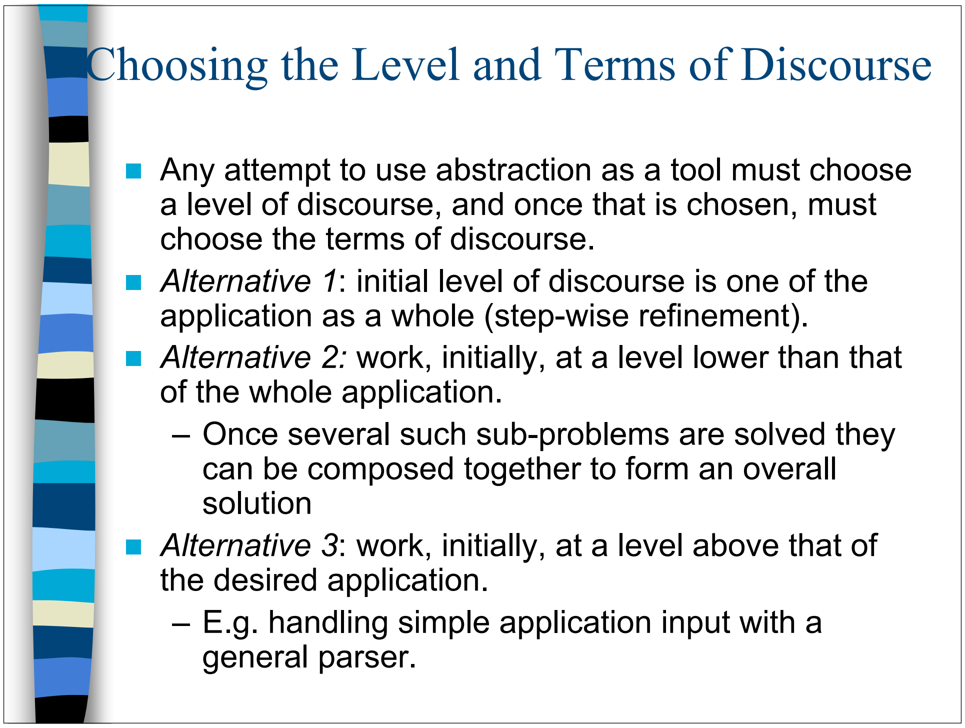
Abstraction and the Simple Machines

- What concepts should be chosen at the outset of a design task?
  - One technique: Search for a “simple machine” that serves as an abstraction of a potential system that will perform the required task
  - For instance, what kind of simple machine makes a software system embedded in a fax machine?
    - At core, it is basically just a little state machine.
- Simple machines provide a plausible first conception of how an application might be built.
- Every application domain has its common simple machines.



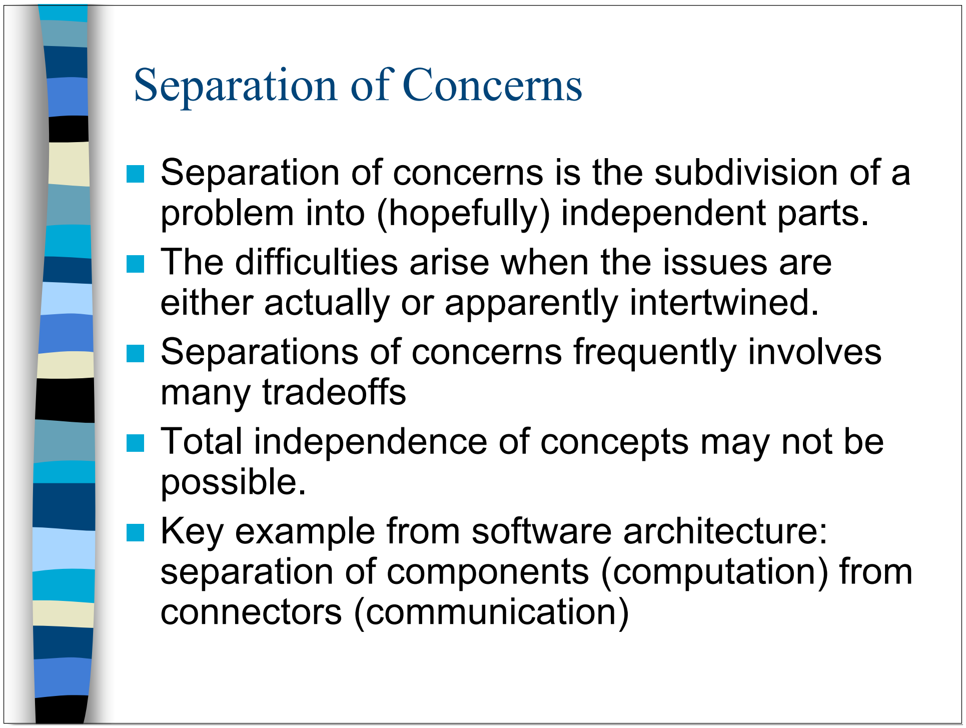
Simple Machines

Domain	**Simple Machines**
Graphics	Pixel arrays Transformation matrices Widgets Abstract depiction graphs
Word processing	Structured documents Layouts
Process control	Finite state machines
Income Tax Software	Hypertext Spreadsheets Form templates
Web pages	Hypertext Composite documents
Scientific computing	Matrices Mathematical functions
Banking	Spreadsheets Databases Transactions



Choosing the Level and Terms of Discourse

- Any attempt to use abstraction as a tool must choose a level of discourse, and once that is chosen, must choose the terms of discourse.
- *Alternative 1*: initial level of discourse is one of the application as a whole (step-wise refinement).
- *Alternative 2:* work, initially, at a level lower than that of the whole application.
  - Once several such sub-problems are solved they can be composed together to form an overall solution
- *Alternative 3*: work, initially, at a level above that of the desired application.
  - E.g. handling simple application input with a general parser.



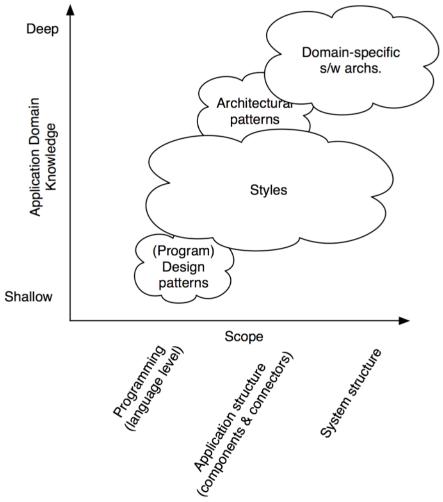
Separation of Concerns

- Separation of concerns is the subdivision of a problem into (hopefully) independent parts.
- The difficulties arise when the issues are either actually or apparently intertwined.
- Separations of concerns frequently involves many tradeoffs
- Total independence of concepts may not be possible.
- Key example from software architecture: separation of components (computation) from connectors (communication)

The Grand Tool: Refined Experience

- Experience must be reflected upon and refined.
- The lessons from prior work include not only the lessons of successes, but also the lessons arising from failure.
- Learn from success and failure of other engineers
  - Literature
  - Conferences
- Experience can provide that initial feasible set of "alternative arrangements for the design as a whole".

Patterns, Styles, and DSSAs


Deep
Application Domain Knowledge
Shallow
Domain-specific s/w archs.
Architectural patterns
Styles
(Program) Design patterns
Scope
Programming (language level)
Application structure (components & connectors)
System structure

Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Domain-Specific Software Architectures

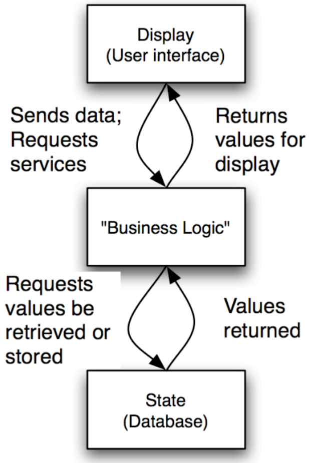
- A DSSA is an assemblage of software components
  - specialized for a particular type of task (domain),
  - generalized for effective use across that domain, and
  - composed in a standardized structure (topology) effective for building successful applications.
- Since DSSAs are specialized for a particular domain they are only of value if one exists for the domain wherein the engineer is tasked with building a new application.
- DSSAs are the pre-eminent means for maximal reuse of knowledge and prior development and hence for developing a new architectural design.

Architectural Patterns

- An architectural pattern is a set of architectural design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears.
- Architectural patterns are similar to DSSAs but applied “at a lower level” and within a much narrower scope.

State-Logic-Display: Three-Tiered Pattern

- Application Examples
  - Business application
  - Multi-player games
  - Web-based applicati

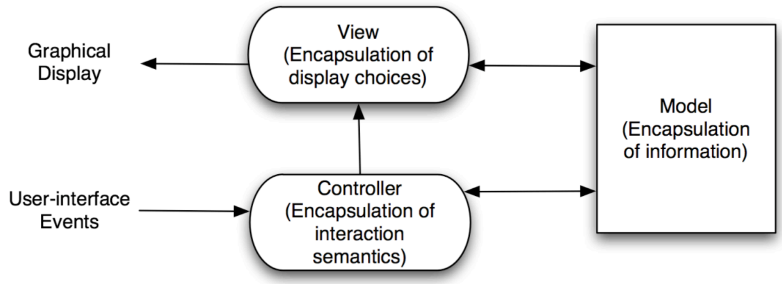


Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Model-View-Controller (MVC)

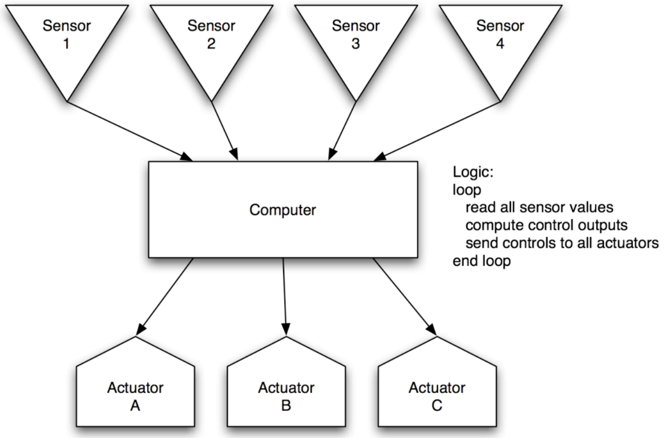
- Objective: Separation between information, presentation and user interaction.
- When a model object value changes, a notification is sent to the view and to the controller. So that the view can update itself and the controller can modify the view if its logic so requires.
- When handling input from the user the windowing system sends the user event to the controller; If a change is required, the controller updates the model object.

Model-View-Controller


Graphical Display
View (Encapsulation of display choices)
Model (Encapsulation of information)
User-interface Events
Controller (Encapsulation of interaction semantics)

Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Sense-Compute-Control



```
Logic:
loop
  read all sensor values
  compute control outputs
  send controls to all actuators
end loop
```

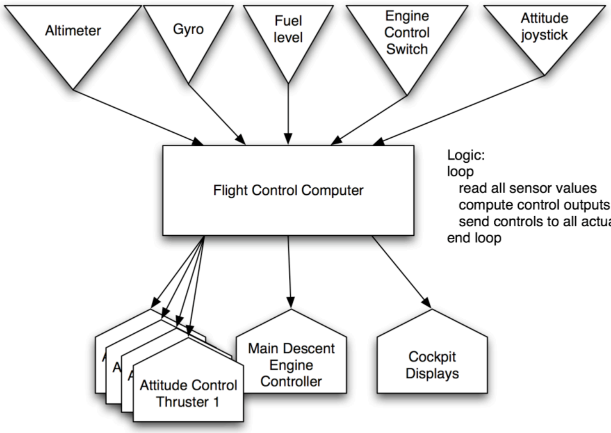
Objective: Structuring embedded control applications

Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

The Lunar Lander: A Long-Running Example

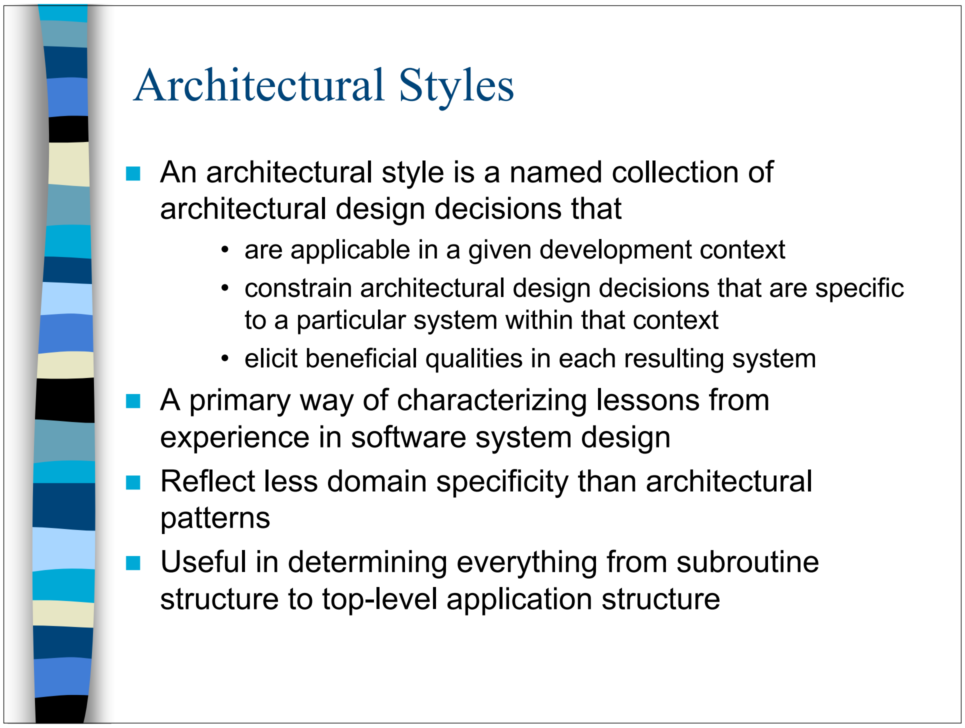
- A simple computer game that first appeared in the 1960's
- Simple concept:
  - You (the pilot) control the descent rate of the Apollo-era Lunar Lander
    - Throttle setting controls descent engine
    - Limited fuel
    - Initial altitude and speed preset
    - If you land with a descent rate of < 5 fps: you win (whether there's fuel left or not)
  - "Advanced" version: joystick controls attitude & horizontal motion

Sense-Compute-Control LL



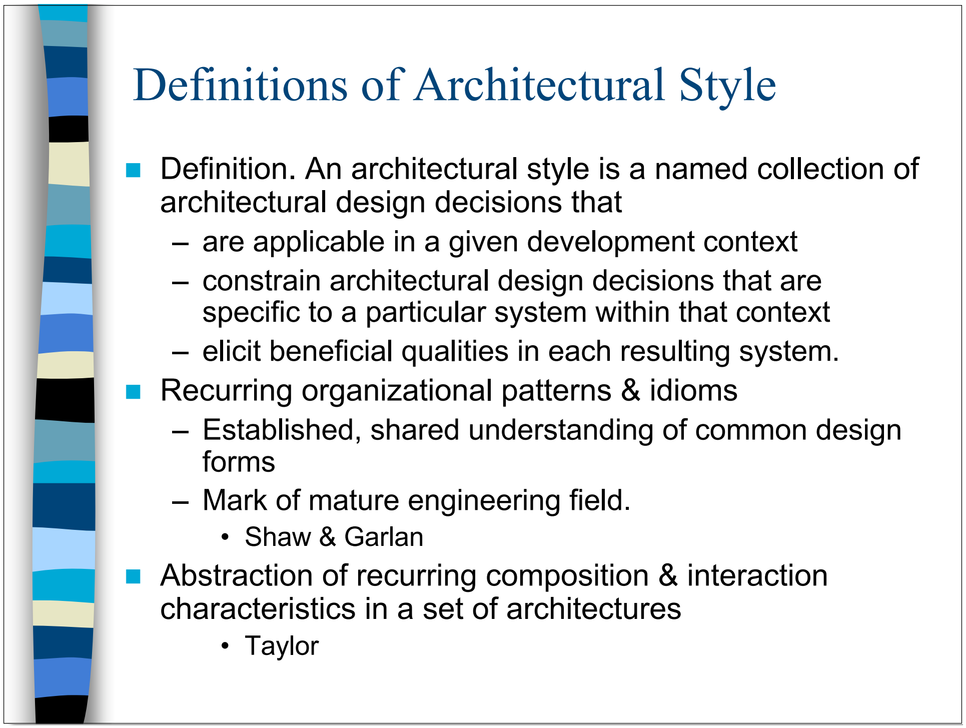
```
Logic:
loop
   read all sensor values
   compute control outputs
   send controls to all actuators
end loop
```

Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.



Architectural Styles

- An architectural style is a named collection of architectural design decisions that
    - are applicable in a given development context
    - constrain architectural design decisions that are specific to a particular system within that context
    - elicit beneficial qualities in each resulting system
- A primary way of characterizing lessons from experience in software system design
- Reflect less domain specificity than architectural patterns
- Useful in determining everything from subroutine structure to top-level application structure



Definitions of Architectural Style

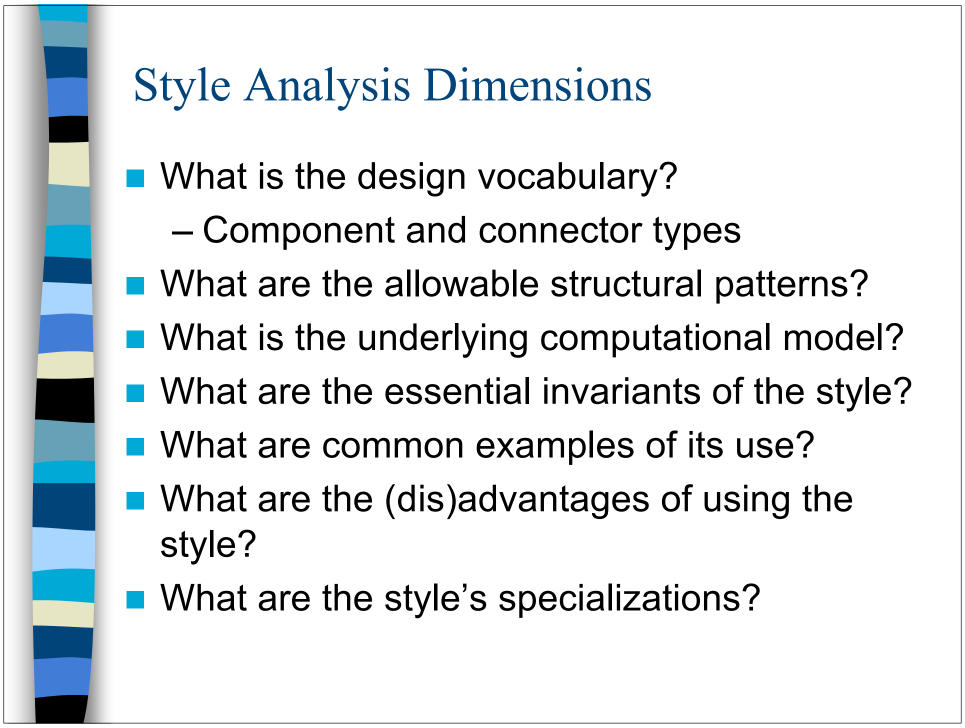
- Definition. An architectural style is a named collection of architectural design decisions that
    - – are applicable in a given development context
    - – constrain architectural design decisions that are specific to a particular system within that context
    - – elicit beneficial qualities in each resulting system.
- Recurring organizational patterns & idioms
    - – Established, shared understanding of common design forms
    - – Mark of mature engineering field.
        - Shaw & Garlan
- Abstraction of recurring composition & interaction characteristics in a set of architectures
    - Taylor

Basic Properties of Styles

- A vocabulary of design elements
  - Component and connector types; data elements
  - e.g., pipes, filters, objects, servers
- A set of configuration rules
  - Topological constraints that determine allowed compositions of elements
  - e.g., a component may be connected to at most two other components
- A semantic interpretation
  - Compositions of design elements have well-defined meanings
- Possible analyses of systems built in a style

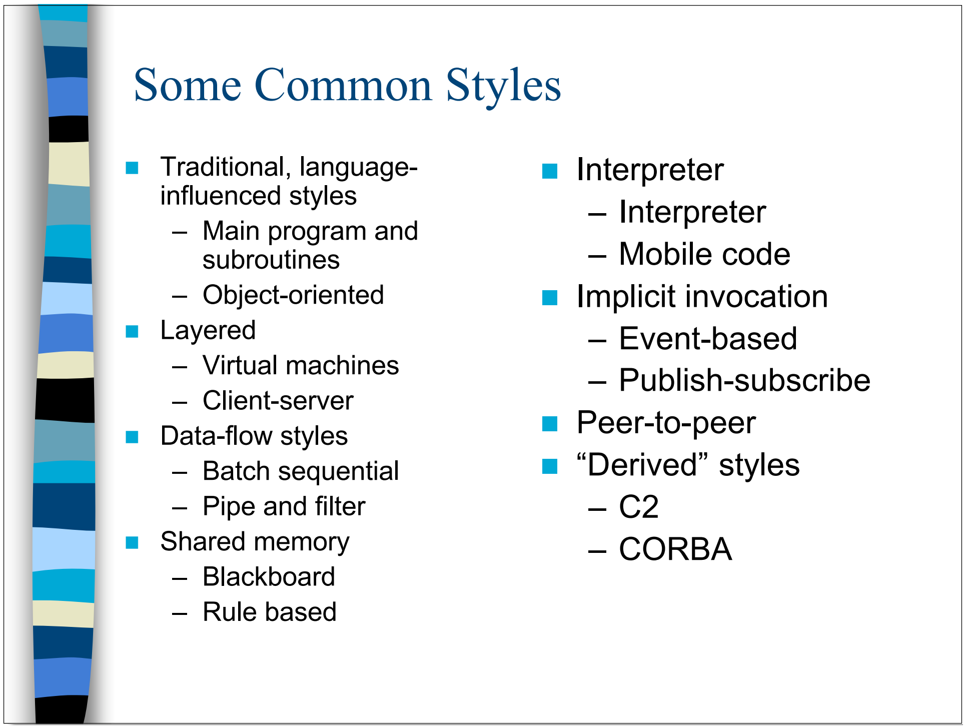
Benefits of Using Styles

- Design reuse
  - Well-understood solutions applied to new problems
- Code reuse
  - Shared implementations of invariant aspects of a style
- Understandability of system organization
  - A phrase such as “client-server” conveys a lot of information
- Interoperability
  - Supported by style standardization
- Style-specific analyses
  - Enabled by the constrained design space
- Visualizations
  - Style-specific depictions matching engineers’ mental models



Style Analysis Dimensions

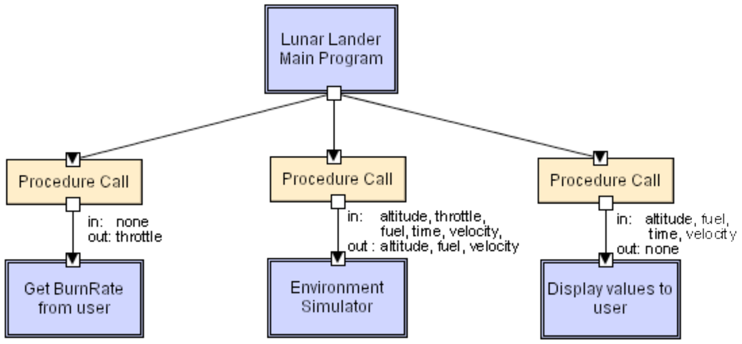
- What is the design vocabulary?
  - Component and connector types
- What are the allowable structural patterns?
- What is the underlying computational model?
- What are the essential invariants of the style?
- What are common examples of its use?
- What are the (dis)advantages of using the style?
- What are the style’s specializations?



Some Common Styles

- Traditional, language-influenced styles
  - Main program and subroutines
  - Object-oriented
- Layered
  - Virtual machines
  - Client-server
- Data-flow styles
  - Batch sequential
  - Pipe and filter
- Shared memory
  - Blackboard
  - Rule based
- Interpreter
  - Interpreter
  - Mobile code
- Implicit invocation
  - Event-based
  - Publish-subscribe
- Peer-to-peer
- “Derived” styles
  - C2
  - CORBA

Main Program and Subroutines LL

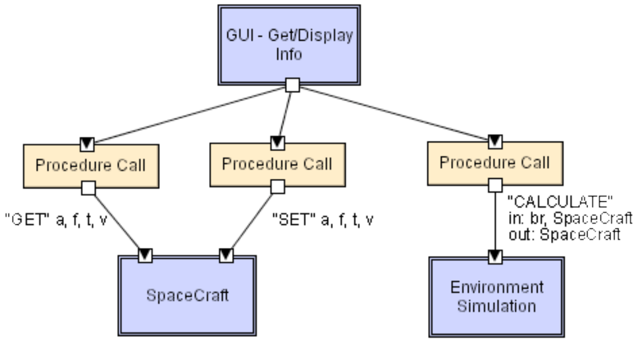


Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Object-Oriented Style

- Components are objects
  - Data and associated operations
- Connectors are messages and method invocations
- Style invariants
  - Objects are responsible for their internal representation integrity
  - Internal representation is hidden from other objects
- Advantages
  - “Infinite malleability” of object internals
  - System decomposition into sets of interacting agents
- Disadvantages
  - Objects must know identities of servers
  - Side effects in object method invocations

Object-Oriented LL



GUI - Get/Display Info

Procedure Call

Procedure Call

Procedure Call

"GET" a, f, t, v

"SET" a, f, t, v

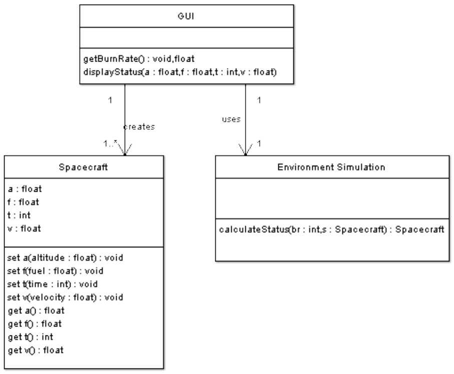
"CALCULATE"
in: br, SpaceCraft
out: SpaceCraft

SpaceCraft

Environment Simulation

Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

OO/LL in UML



GUI

getBurnRate() : void,float
displayStatus(a : float,f : float,t : int,v : float)

1

creates

1..*

uses

1

1

Spacecraft

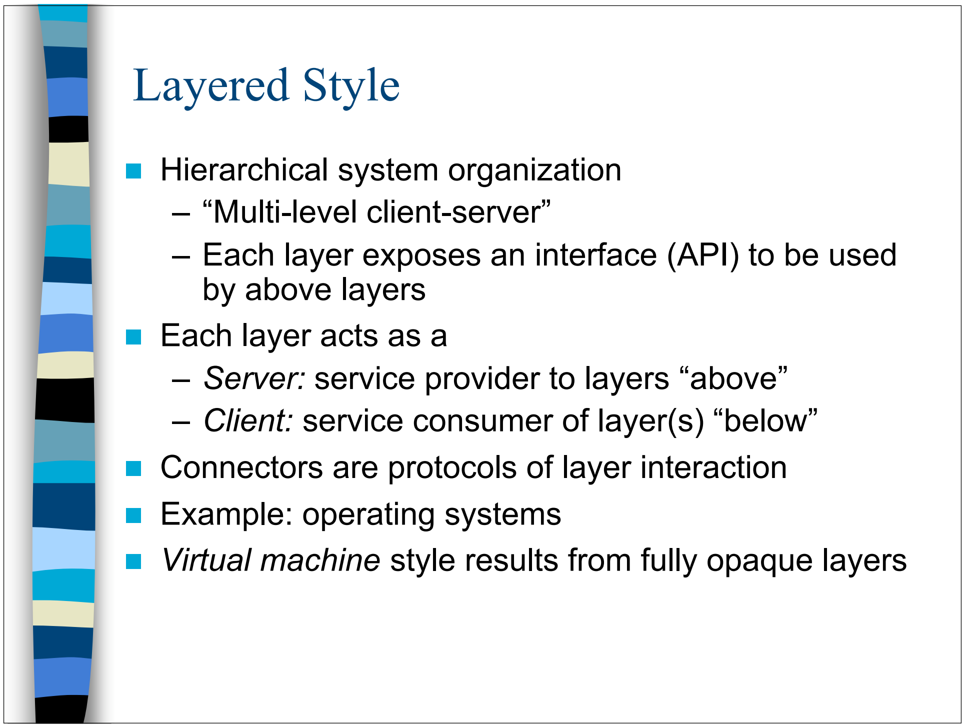
a : float
f : float
t : int
v : float

set a(altitude : float) : void
set f(fuel : float) : void
set t(time : int) : void
set v(velocity : float) : void
get a() : float
get f() : float
get t() : int
get v() : float

Environment Simulation

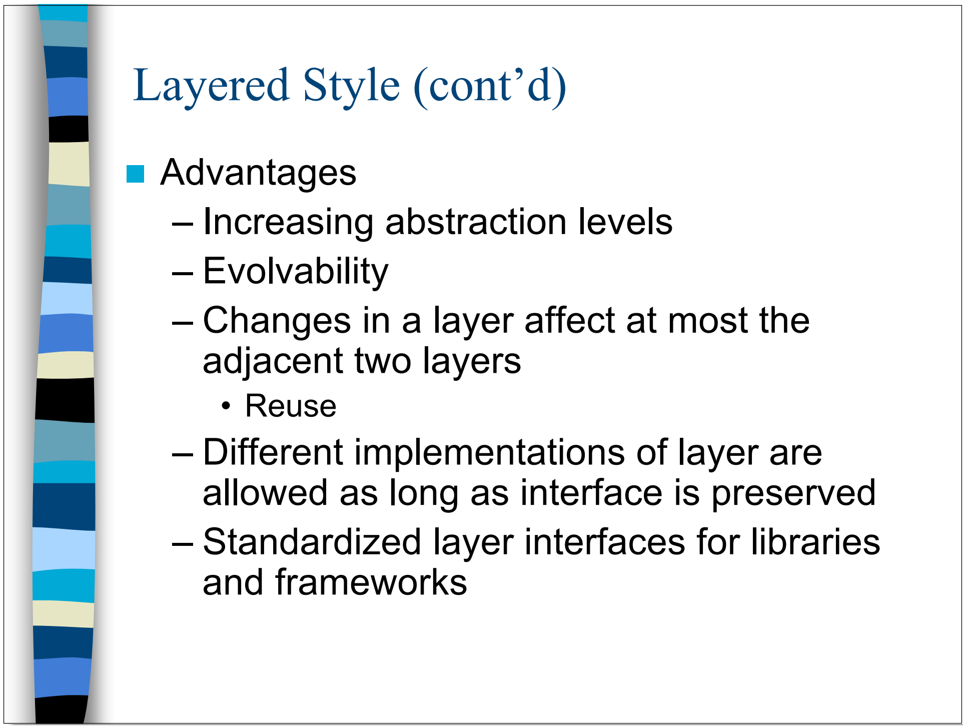
calculateStatus(br : int,s : Spacecraft) : Spacecraft

Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.



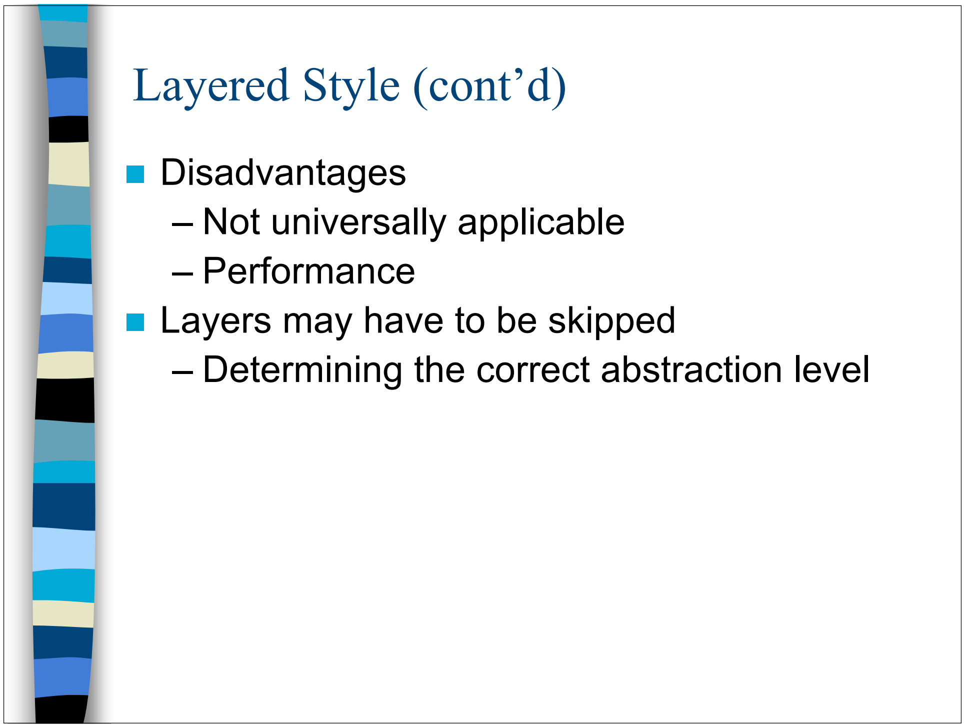
Layered Style

- Hierarchical system organization
  - "Multi-level client-server"
  - Each layer exposes an interface (API) to be used by above layers
- Each layer acts as a
  - *Server:* service provider to layers "above"
  - *Client:* service consumer of layer(s) "below"
- Connectors are protocols of layer interaction
- Example: operating systems
- *Virtual machine* style results from fully opaque layers



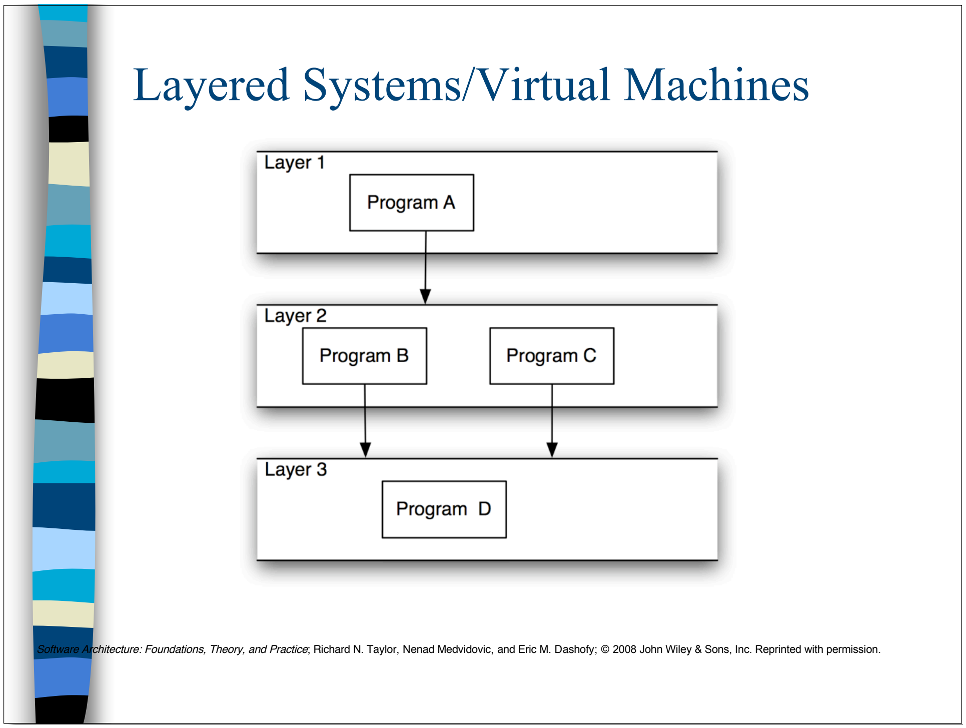
Layered Style (cont'd)

- Advantages
  - Increasing abstraction levels
  - Evolvability
  - Changes in a layer affect at most the adjacent two layers
    - Reuse
  - Different implementations of layer are allowed as long as interface is preserved
  - Standardized layer interfaces for libraries and frameworks



Layered Style (cont'd)

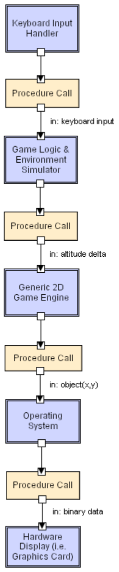
- Disadvantages
  - – Not universally applicable
  - – Performance
- Layers may have to be skipped
  - – Determining the correct abstraction level



Layered Systems/Virtual Machines

Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

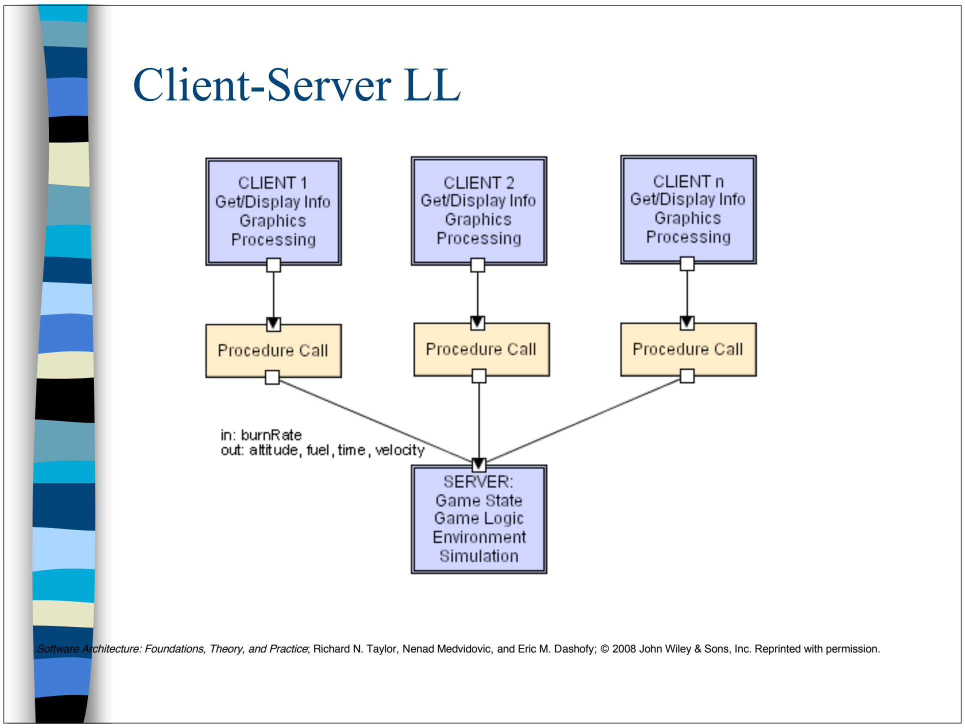
Layered LL



Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

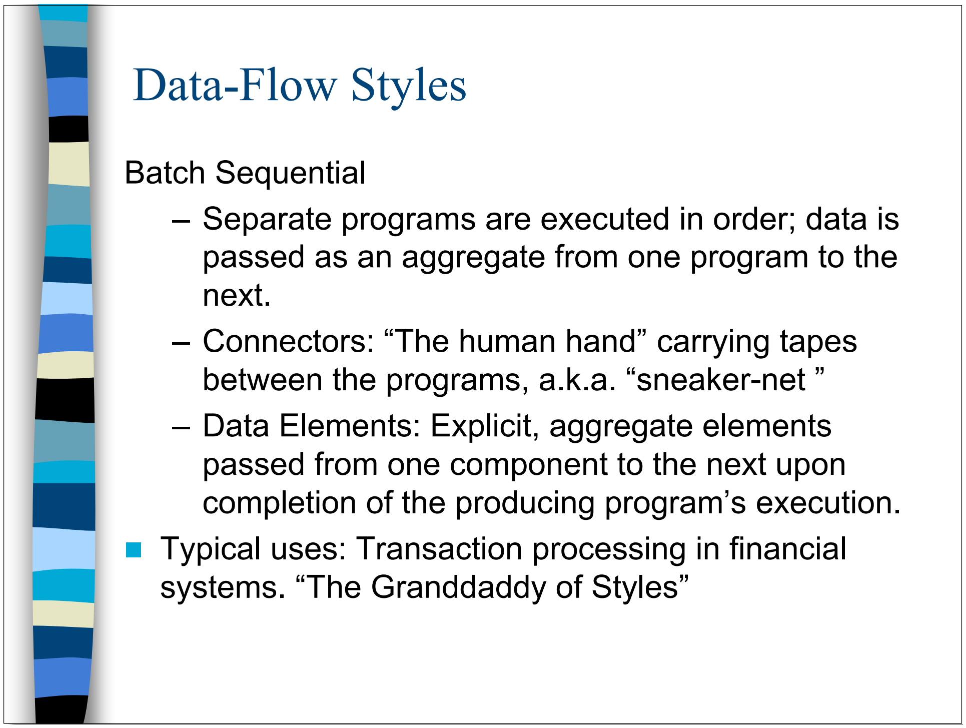
Client-Server Style

- Components are clients and servers
- Servers do not know number or identities of clients
- Clients know server's identity
- Connectors are RPC-based network interaction protocols



Client-Server LL

Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

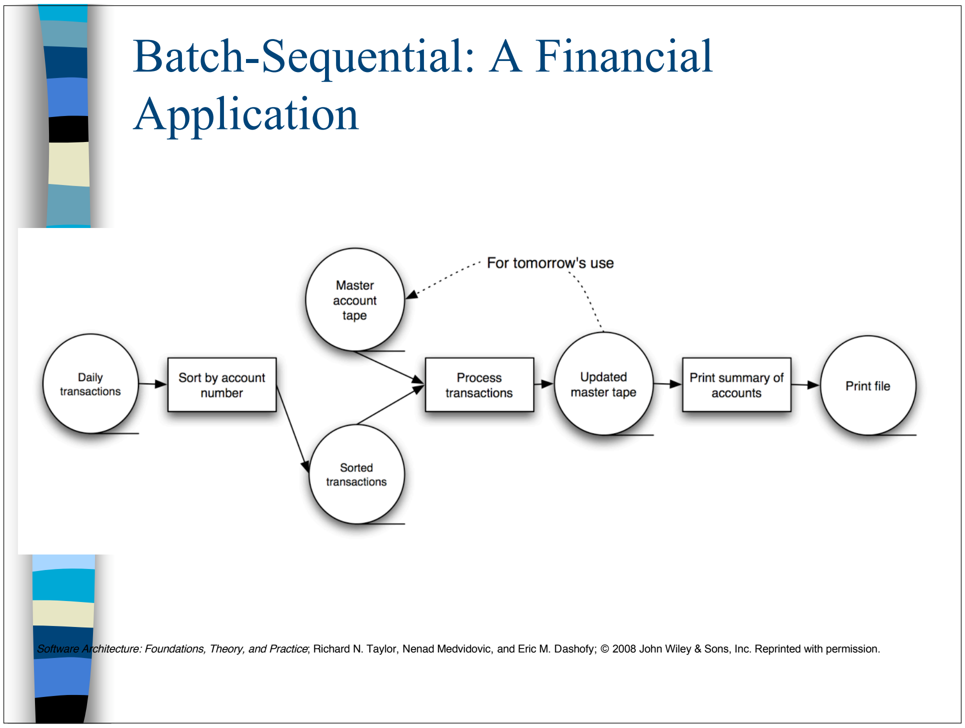


Data-Flow Styles

Batch Sequential

- – Separate programs are executed in order; data is passed as an aggregate from one program to the next.
- – Connectors: “The human hand” carrying tapes between the programs, a.k.a. “sneaker-net ”
- – Data Elements: Explicit, aggregate elements passed from one component to the next upon completion of the producing program’s execution.

- Typical uses: Transaction processing in financial systems. “The Granddaddy of Styles”



Batch-Sequential: A Financial Application

For tomorrow's use

Daily transactions

Sort by account number

Master account tape

Sorted transactions

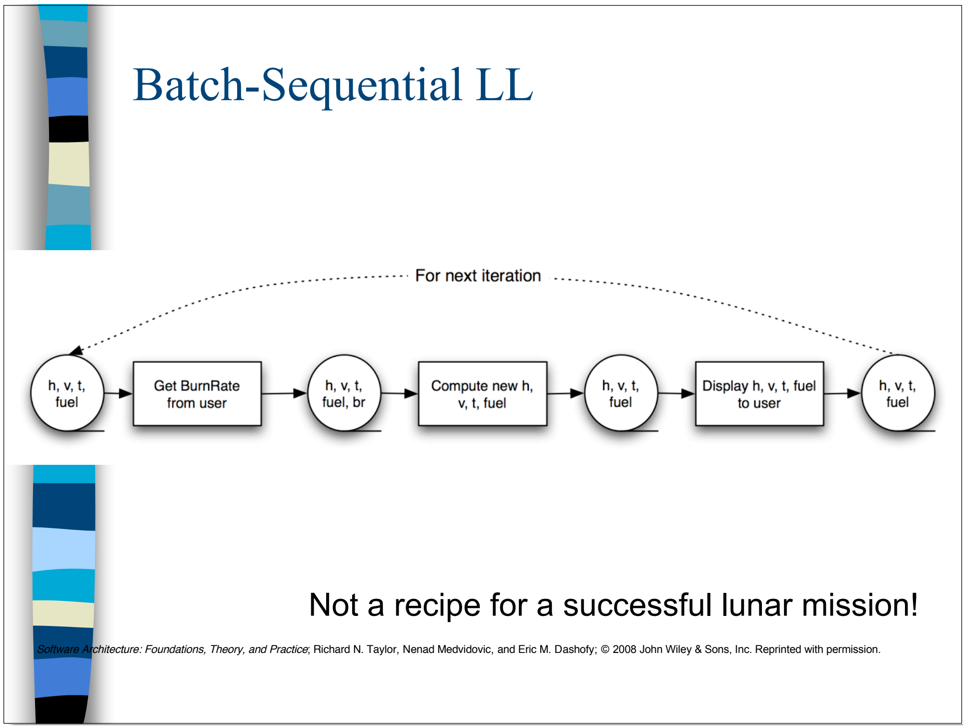
Process transactions

Updated master tape

Print summary of accounts

Print file

Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.



Batch-Sequential LL

For next iteration

h, v, t, fuel

Get BurnRate from user

h, v, t, fuel, br

Compute new h, v, t, fuel

h, v, t, fuel

Display h, v, t, fuel to user

h, v, t, fuel

Not a recipe for a successful lunar mission!

Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Pipe and Filter Style

- Components are filters
  - Transform input data streams into output data streams
  - Possibly incremental production of output
- Connectors are pipes
  - Conduits for data streams
- Style invariants
  - Filters are independent (no shared state)
  - Filter has no knowledge of up- or down-stream filters
- Examples
  - UNIX shell signal processing
  - Distributed systems parallel programming
- Example: `ls invoices | grep -e August | sort`

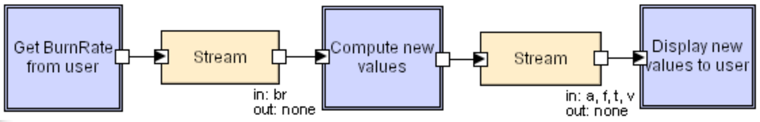
Pipe and Filter (cont'd)

- Variations
  - Pipelines — linear sequences of filters
  - Bounded pipes — limited amount of data on a pipe
  - Typed pipes — data strongly typed
- Advantages
  - System behavior is a succession of component behaviors
  - Filter addition, replacement, and reuse
    - Possible to hook any two filters together
  - Certain analyses
    - Throughput, latency, deadlock
  - Concurrent execution

Pipe and Filter (cont'd)

- Disadvantages
  - Batch organization of processing
  - Interactive applications
  - Lowest common denominator on data transmission

Pipe and Filter LL

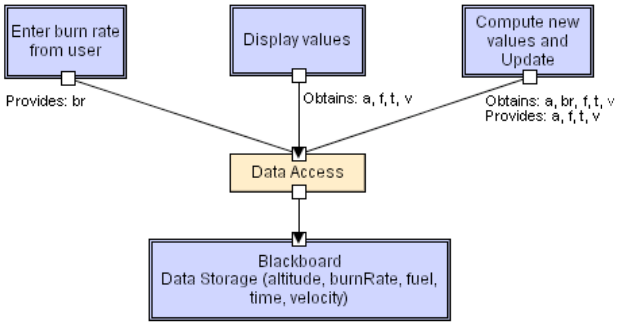


Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

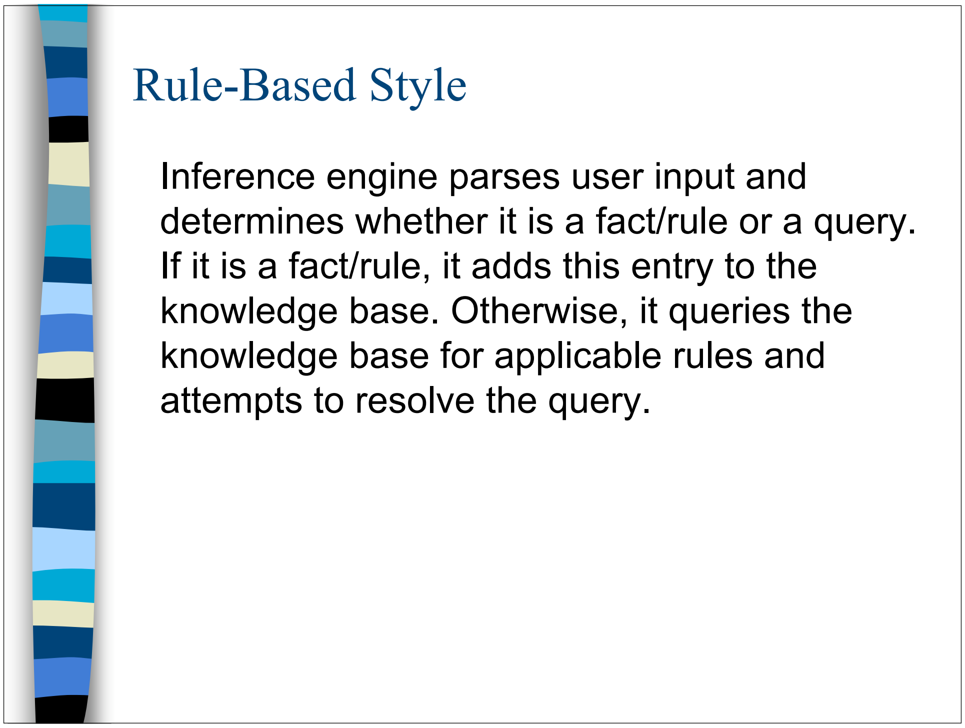
Blackboard Style

- Two kinds of components
  - Central data structure — blackboard
  - Components operating on the blackboard
- System control is entirely driven by the blackboard state
- Examples
  - Typically used for AI systems
  - Integrated software environments (e.g., Interlisp)
  - Compiler architecture

Blackboard LL

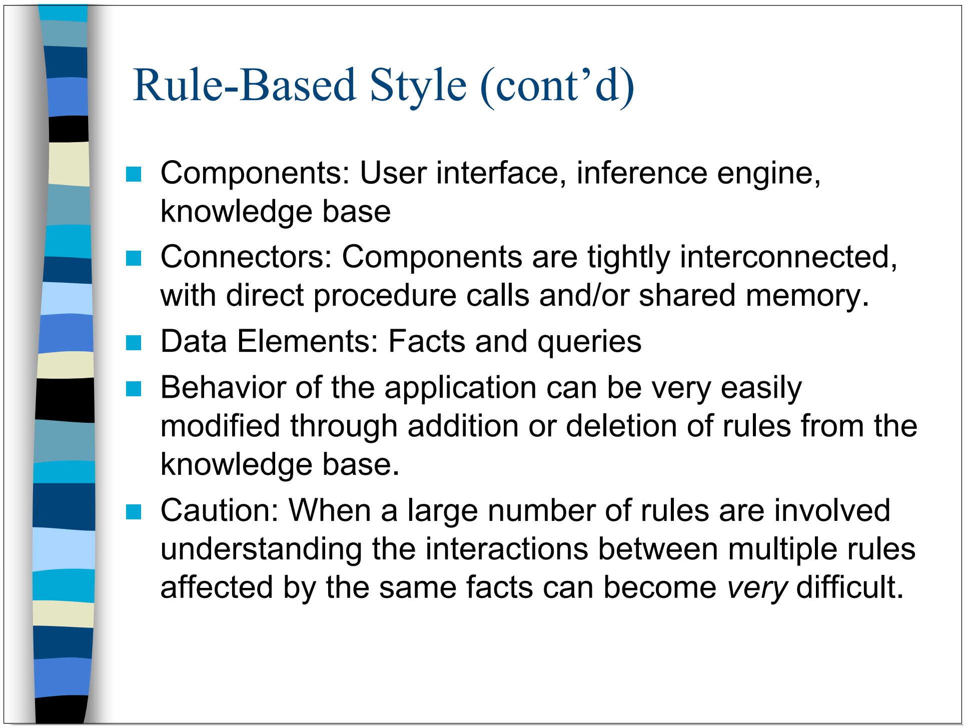

Enter burn rate from user
Display values
Compute new values and Update
Provides: br
Obtains: a, f, t, v
Obtains: a, br, f, t, v
Provides: a, f, t, v
Data Access
Blackboard
Data Storage (altitude, burnRate, fuel, time, velocity)

Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.



Rule-Based Style

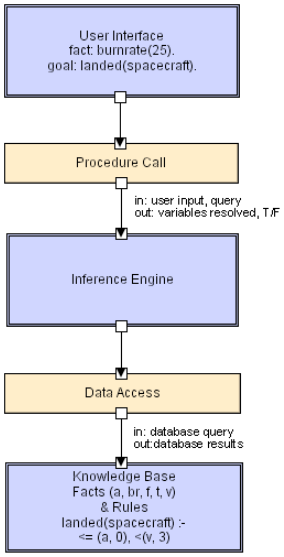
Inference engine parses user input and determines whether it is a fact/rule or a query. If it is a fact/rule, it adds this entry to the knowledge base. Otherwise, it queries the knowledge base for applicable rules and attempts to resolve the query.



Rule-Based Style (cont’d)

- Components: User interface, inference engine, knowledge base
- Connectors: Components are tightly interconnected, with direct procedure calls and/or shared memory.
- Data Elements: Facts and queries
- Behavior of the application can be very easily modified through addition or deletion of rules from the knowledge base.
- Caution: When a large number of rules are involved understanding the interactions between multiple rules affected by the same facts can become *very* difficult.

Rule Based LL

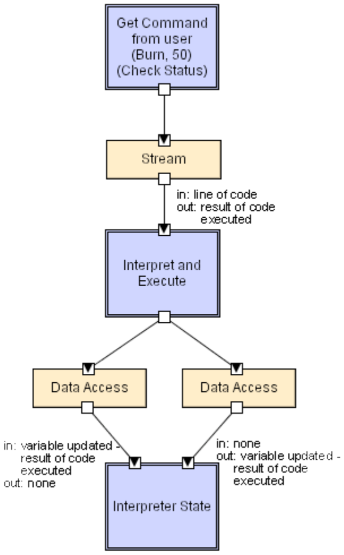


Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Interpreter Style

- Interpreter parses and executes input commands, updating the state maintained by the interpreter
- Components: Command interpreter, program/interpreter state, user interface.
- Connectors: Typically very closely bound with direct procedure calls and shared state.
- Highly dynamic behavior possible, where the set of commands is dynamically modified. System architecture may remain constant while new capabilities are created based upon existing primitives.
- Superb for end-user programmability; supports dynamically changing set of capabilities
- Lisp and Scheme

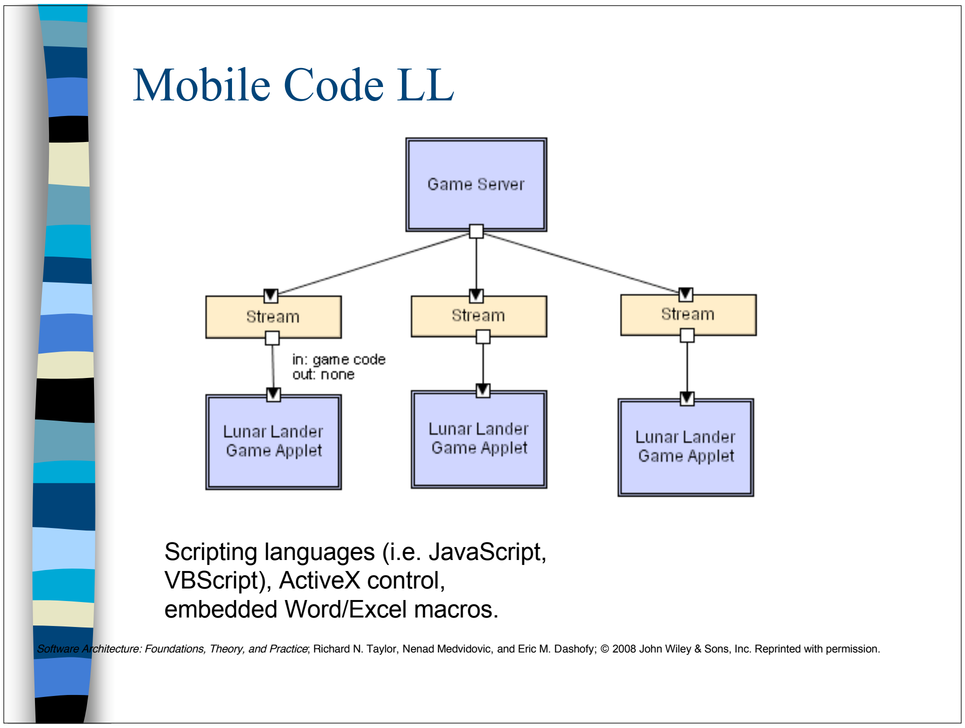
Interpreter LL


Get Command from user (Burn, 50) (Check Status)
Stream
in: line of code
out: result of code executed
Interpret and Execute
Data Access
Data Access
in: variable updated - result of code executed
out: none
in: none
out: variable updated - result of code executed
Interpreter State

rchitecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Mobile-Code Style

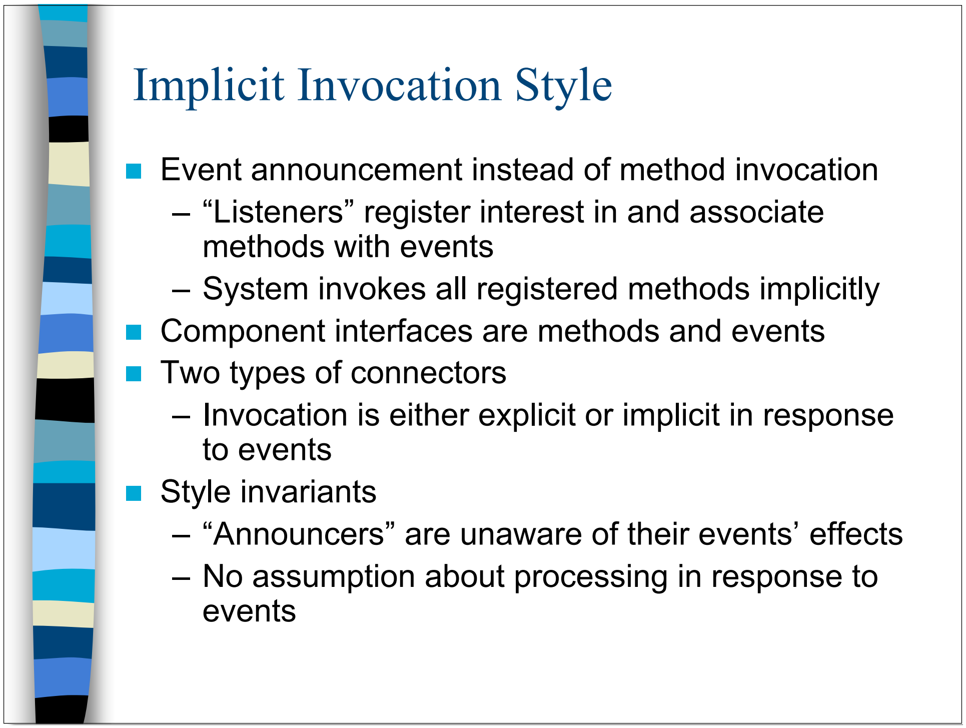
- Summary: a data element (some representation of a program) is dynamically transformed into a data processing component.
- Components: “Execution dock”, which handles receipt of code and state; code compiler/interpreter
- Connectors: Network protocols and elements for packaging code and data for transmission.
- Data Elements: Representations of code as data; program state; data
- Variants: Code-on-demand, remote evaluation, and mobile agent.



Mobile Code LL

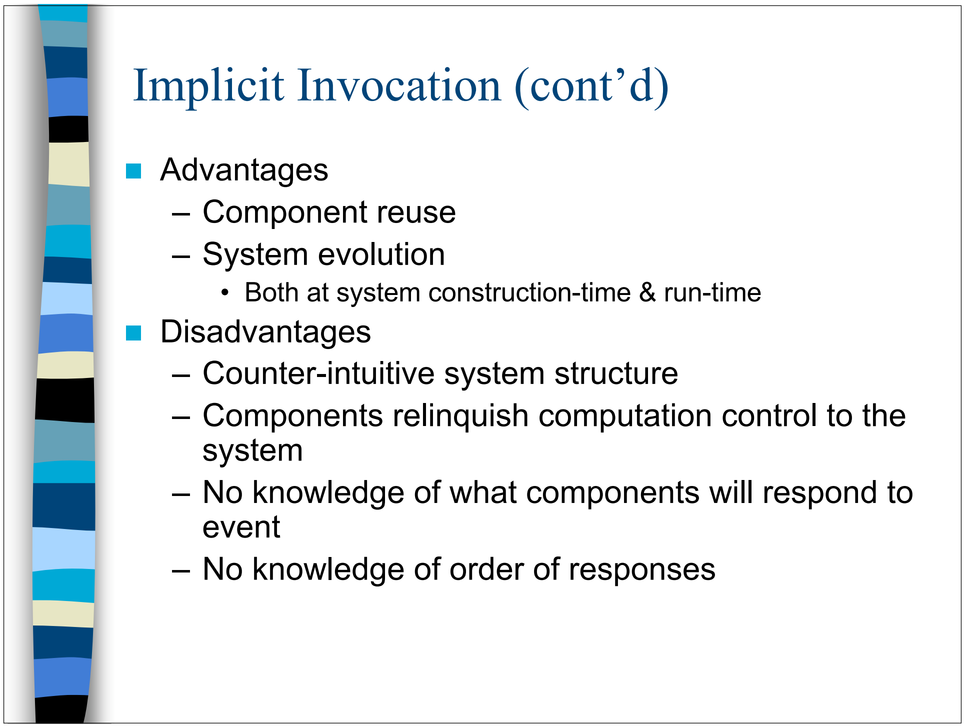
Scripting languages (i.e. JavaScript,
VBScript), ActiveX control,
embedded Word/Excel macros.

Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.



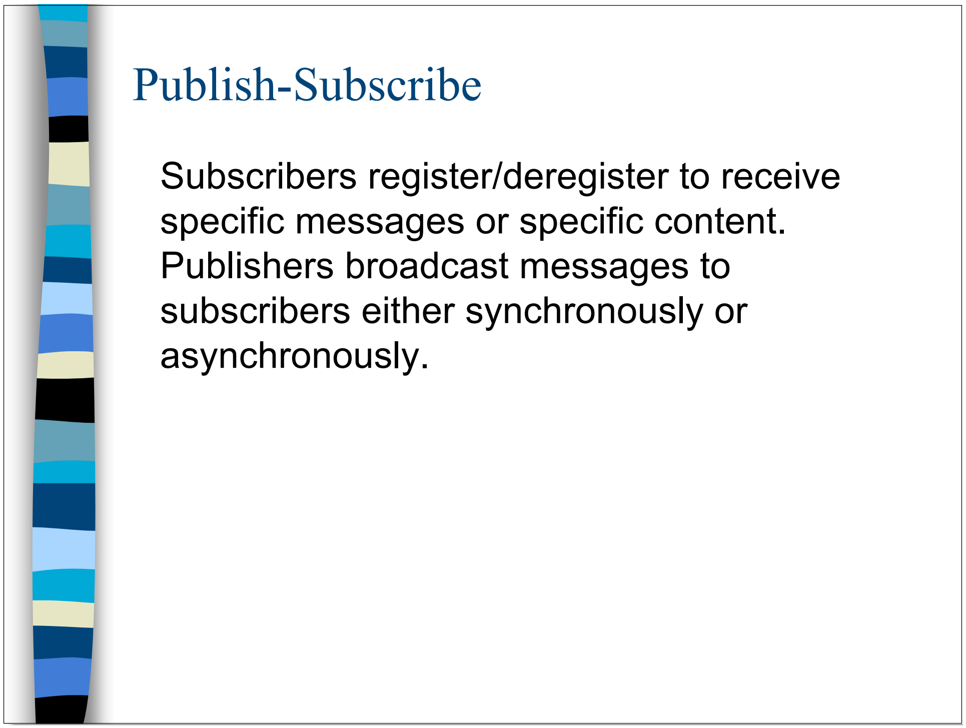
Implicit Invocation Style

- Event announcement instead of method invocation
  - “Listeners” register interest in and associate methods with events
  - System invokes all registered methods implicitly
- Component interfaces are methods and events
- Two types of connectors
  - Invocation is either explicit or implicit in response to events
- Style invariants
  - “Announcers” are unaware of their events’ effects
  - No assumption about processing in response to events



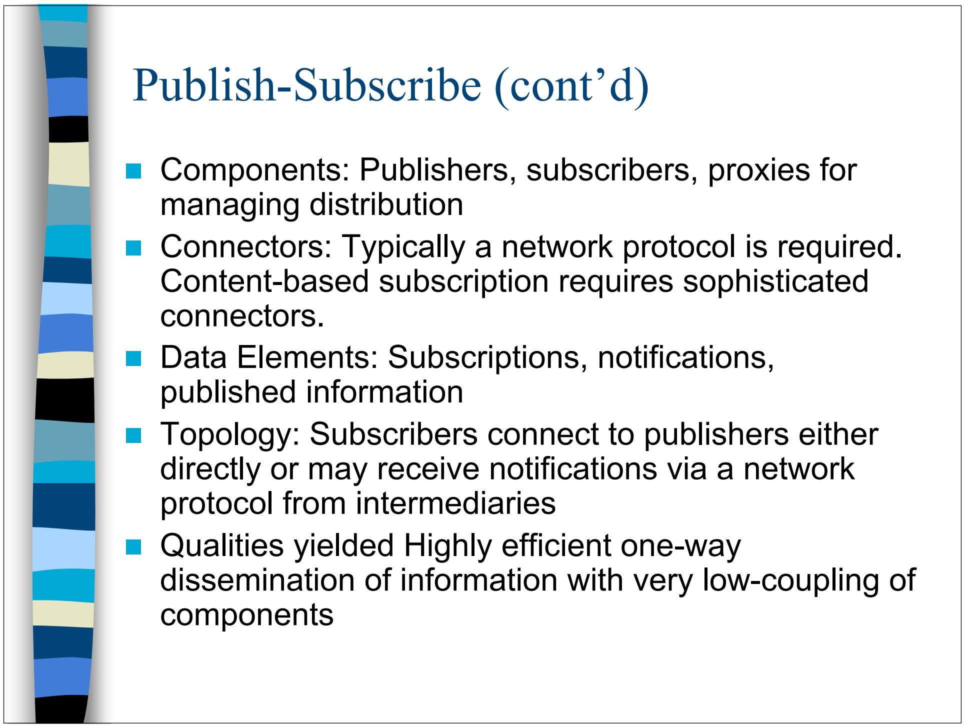
Implicit Invocation (cont'd)

- Advantages
  - Component reuse
  - System evolution
    - Both at system construction-time & run-time
- Disadvantages
  - Counter-intuitive system structure
  - Components relinquish computation control to the system
  - No knowledge of what components will respond to event
  - No knowledge of order of responses



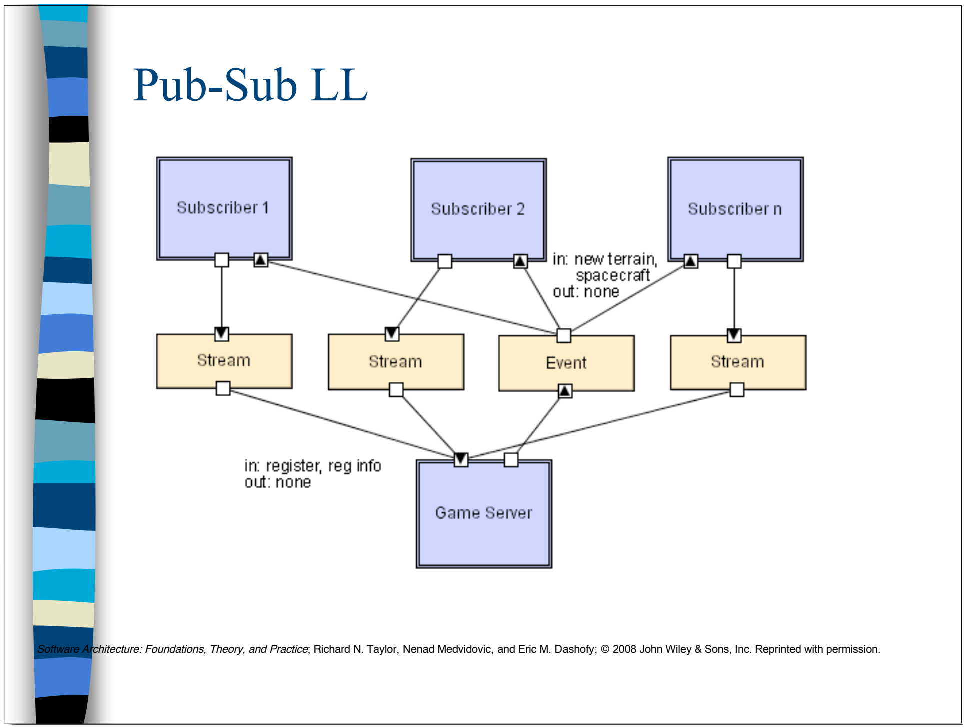
Publish-Subscribe

Subscribers register/deregister to receive specific messages or specific content. Publishers broadcast messages to subscribers either synchronously or asynchronously.



Publish-Subscribe (cont’d)

- Components: Publishers, subscribers, proxies for managing distribution
- Connectors: Typically a network protocol is required. Content-based subscription requires sophisticated connectors.
- Data Elements: Subscriptions, notifications, published information
- Topology: Subscribers connect to publishers either directly or may receive notifications via a network protocol from intermediaries
- Qualities yielded Highly efficient one-way dissemination of information with very low-coupling of components



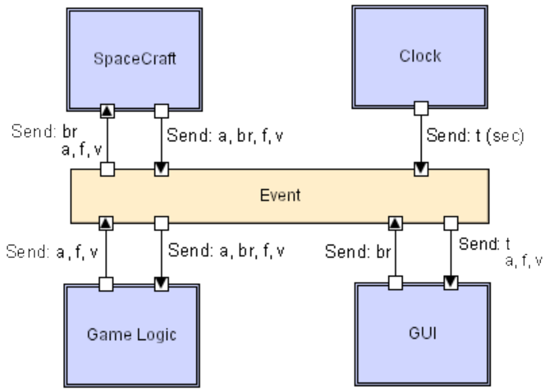
Pub-Sub LL

Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

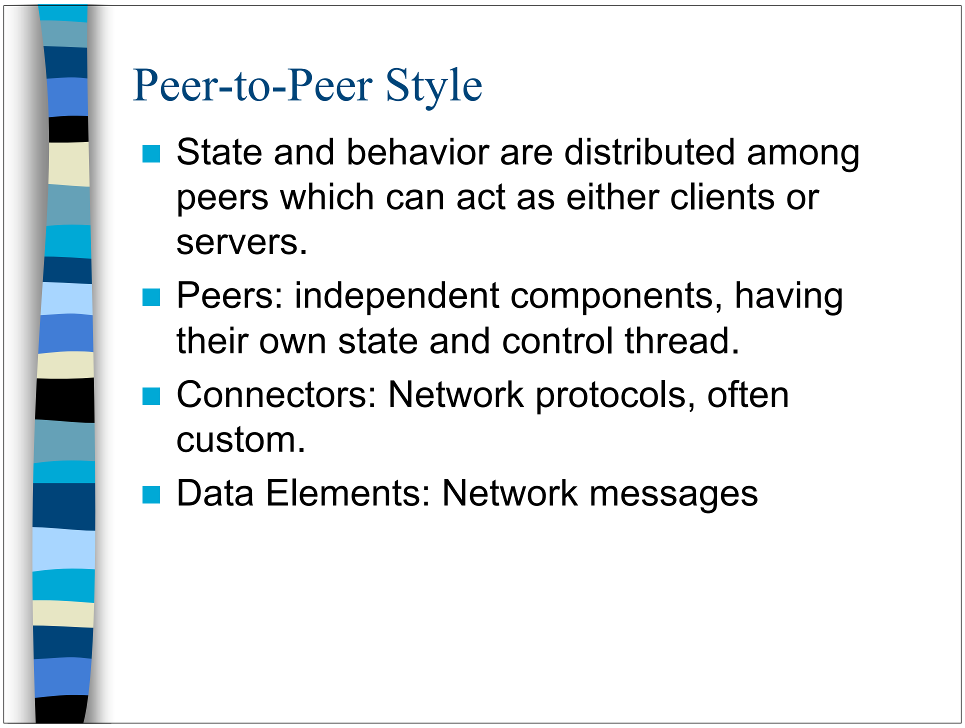
Event-Based Style

- Independent components asynchronously emit and receive events communicated over event buses
- Components: Independent, concurrent event generators and/or consumers
- Connectors: Event buses (at least one)
- Data Elements: Events – data sent as a first-class entity over the event bus
- Topology: Components communicate with the event buses, not directly to each other.
- Variants: Component communication with the event bus may either be push or pull based.
- Highly scalable, easy to evolve, effective for highly distributed applications.

Event-based LL

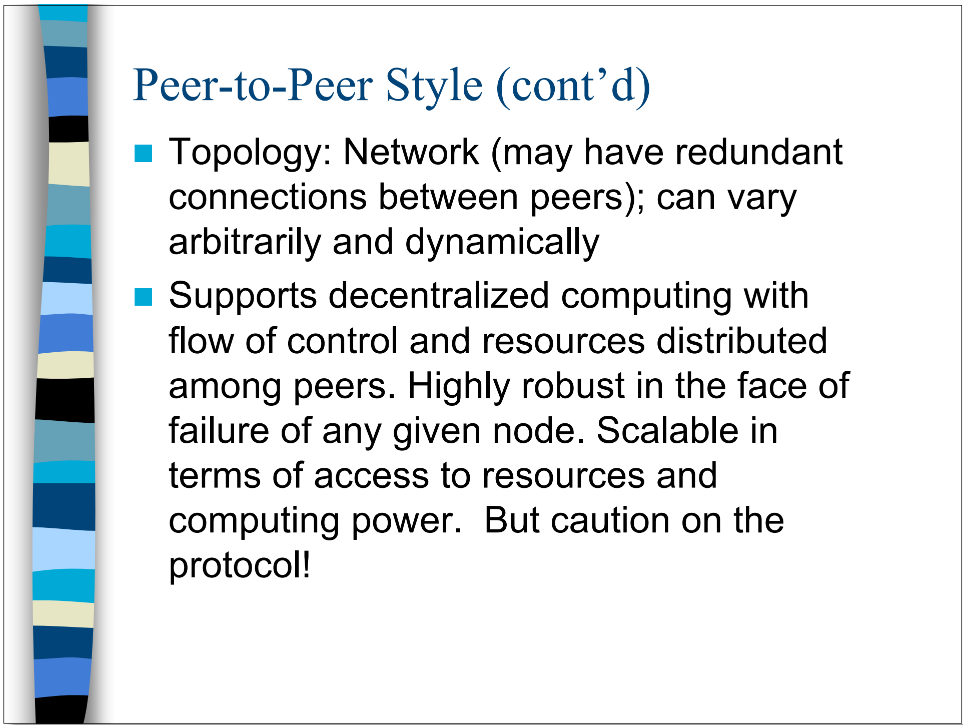


Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.



Peer-to-Peer Style

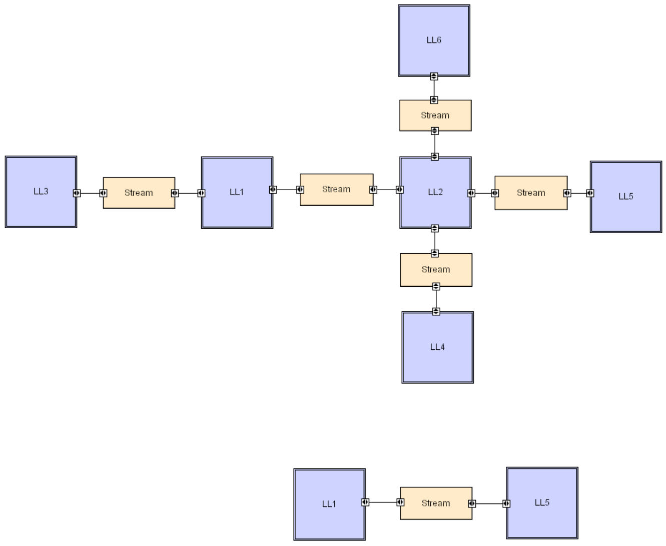
- State and behavior are distributed among peers which can act as either clients or servers.
- Peers: independent components, having their own state and control thread.
- Connectors: Network protocols, often custom.
- Data Elements: Network messages



Peer-to-Peer Style (cont'd)

- Topology: Network (may have redundant connections between peers); can vary arbitrarily and dynamically
- Supports decentralized computing with flow of control and resources distributed among peers. Highly robust in the face of failure of any given node. Scalable in terms of access to resources and computing power. But caution on the protocol!

Peer-to-Peer LL



Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.