Institute for Software Research International
School of Computer Science
Carnegie Mellon

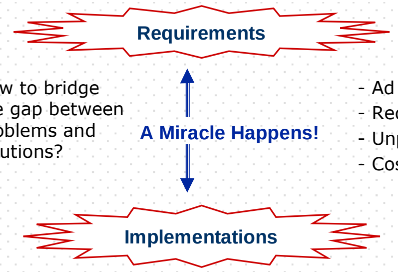
Software Architectures

Presenter: Marwan Abi-Antoun

Slides Courtesy of Professor David Garlan

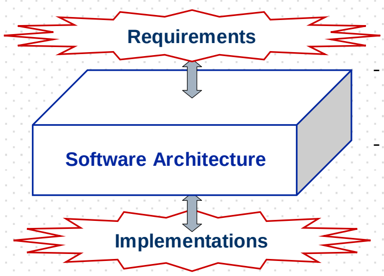
From Requirements to Code…

How to bridge the gap between problems and solutions?



- Ad hoc
- Requires gurus
- Unpredictable
- Costly

The Role of Software Architecture



- High-level abstractions (components, connectors, …)
- Reuse of design idioms

Quick Survey

- What is an architect?

The Role Of The Architect

- Understand business needs and project requirements
- Be aware of a variety of technical approaches to solving the problems presented
- Evaluate tradeoffs between these approaches
- Translate the needs and tradeoffs into a technical architecture that addresses the problems and makes the appropriate tradeoffs
- Guide development team to build system as designed
- “Soft” skills are as important as technical skills

Quick Survey

- What is an architecture?

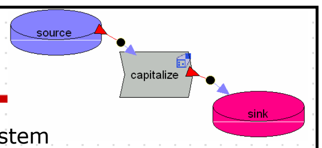
What is an architecture?

- A software architecture is the structure or structures of a system, which comprise elements, their externally-visible properties, and the relationships among them
- But what kinds of structure?
  - *modules*: showing composition/decomposition
  - *runtime*: components at runtime
  - *allocation*: how software is deployed
  - …
- Each is the basis of an Architectural View

Component-and-Connector (C&C) View

- Decomposition of system into **components**…
  - Components: principal units of run-time computation and data stores
    - Examples: client, server
  - Typically hierarchical
- And **connectors**…
  - Connectors: define an abstraction of the interactions between components
    - Examples: procedure call, pipe, event announce
- Using architectural **styles**…
  - Guide composition of components and connectors
- And **constraints** (or invariants)

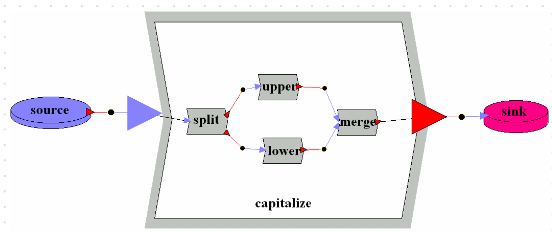
Example: *CaPiTaLiZe*



- Pipe-and-Filter System
  - data source component (*source*)
  - a data sink component (*sink)*
  - a filter component (*capitalize*)
  - two connectors (character pipes)
- Component *capitalize*
  - Receives ASCII characters from *source*
  - Converts characters alternatively to upper or lower case
  - Sends characters on to component *sink*

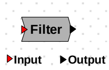
Example: *CaPiTaLiZe* (continued)

- Further decomposed into a sub-architecture consisting of another pipe-and-filter system



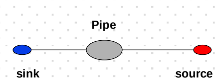
C&C Components

- Components have one or more interfaces ("ports")
- Each interface:
  - **provides** a set of services that other components may use
  - **requires** (or **uses**) a set of services that other components must provide



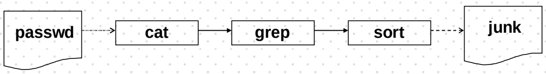
C&C Connectors

- Show pathways of communication
- Represent interactions between components
  - Example: method calls
  - Example: SQL connection
- The interface of a connector is defined as a set of roles (think of it as the "protocol")



Unix Pipe-and-Filter Systems

```
cat /etc/passwd | grep “joe” | sort > junk
```

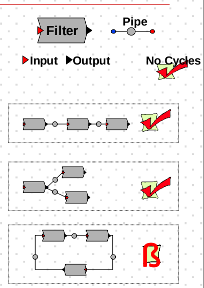


How Does a Software Architecture Help?

- Understanding
  - Vocabulary for structure, system constraints
- Construction
  - What behavior must be built-in before actually having running code
- Design Reuse
  - Reuse of styles and selection among alternatives
- Evolution (allowable envelope of change)
  - Limits of scalability and adaptation
- Analysis
  - System-level analysis that exploits structural constraints
  - Performance, reliability, security, fault-tolerance, …

Architectural Styles or Families

- Describe *sets* of related architectures
  - Vocabulary (types) of components, connectors, …
  - Topological constraints that all members of the family must satisfy
- Useful for
  - Style-based analyses
  - Checking conformance to a style



How many Architectural Styles?

- Understand Pure Styles
  - Overall organizational patterns
- Understand specializations and examples
  - What are some examples and common variants?
- Pure styles are rarely found in practice
  - Systems in practice deviate from the academic definitions (for good reasons, hopefully!)
  - Combine several architectural styles simultaneously
- What you need to know:
  - Understand strengths and weaknesses
  - Understand consequences or tradeoffs of deviating from the style

Common Architectural Styles

Call-return

…
Main program/subroutines
Simple Client-server

Hierarchical

…
N-Tiered

Interacting processes

…
Implicit invocation (a.k.a. Publish-Subscribe)

Data Flow

…
Dataflow
Pipe-and-filter

Shared Data/Repository

…
Databases
Blackboard

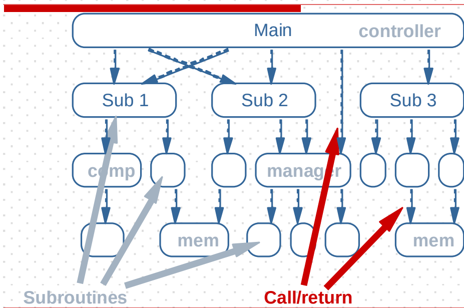
Call-Return Style Semantics

- Functional correctness is hierarchical
- Correctness of one component depends on the correctness of the components on whose services it relies
- Leads naturally to a pre/post-condition style of specification
  - Pre = conditions under which a service may be requested
  - Post = the result of having made a service request

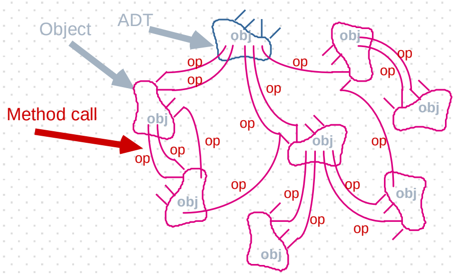
Main Program and Subroutines

- Hierarchical decomposition:
  - Based on definition-use relationship
- Hierarchical reasoning:
  - Correctness of a subroutine depends on the correctness of the subroutines it calls
- Natural correspondence to code structures
  - Code modules are viewed as the corresponding run-time entities
- Subsystem structure implicit:
  - Subroutines typically aggregated into modules

Main Program/Subroutine Pattern



Data Abstraction or Object-Oriented



Problems with Object Approaches

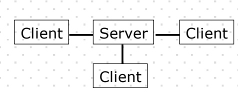
- Managing many objects
  - Vast sea of objects requires additional structuring
- Distributed responsibility for behavior
  - Makes system hard to understand
  - Interaction diagrams now used in design
- Capturing families of related designs
  - Types/classes are often not enough
  - design patterns as an emerging off-shoot

Generalizing the Call-Return Style

- System Topology
  - In general, no restriction (i.e., arbitrary graph)
- Special cases may restrict topology
  - Client-server (Asymmetric)
    - Clients can only talk to servers, not to each other
    - Client knows about servers, but not vice versa
  - Tiers (elaboration on client-server)
    - Hierarchical virtual machines
    - Aggregation into run-time layers

Simple Client-Server

- A client is a system that accesses a service from the server
- A server carries out some task on behalf of a client
- Various topologies
  - Initially star, others emerged later



Client/Server Semantics

- Servers do not know the identities or number of clients that will access it at run time
- Clients know the identity of a server or can find it from another server (that they know the identity of)
- Clients and servers use (or agree) the same protocols to communicate

Client/Server Tradeoffs

- General quality attributes promoted
  - Scale
    - easy to add more clients
    - easy to add more data (provided the structure or access services do not change)
- General quality attributes inhibited
  - reliability?
  - performance?
  - security?
  - higher complexity? (harder to test, harder to maintain

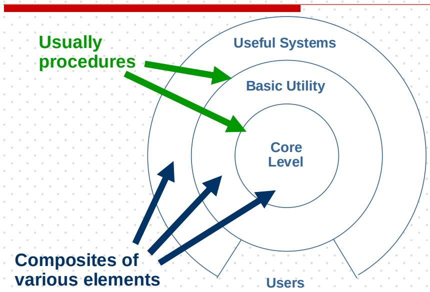
Tiers

- Clients and Servers are organized into levels, called *tiers*
- Each tier provides a set of services to the tiers above it
- Each tier relies on services from the tiers below it
- A tier encapsulates a set of services and the lower-level implementations that it relies on.
  - Often a lower tier acts as a "virtual machine" for the tier above

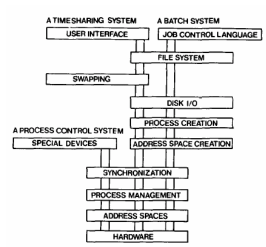
Note: Tiered vs. Layered

- Layered
  - Applies to module view (i.e., static source code organization)
- Tiered
  - Applies to C&C view (i.e., runtime view)
- It's important to not mix the two in the same diagram
  - More on this later

Hierarchical: Virtual Machine

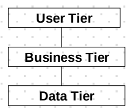


Tiered Operating System



The 3-Tiered Client-Server

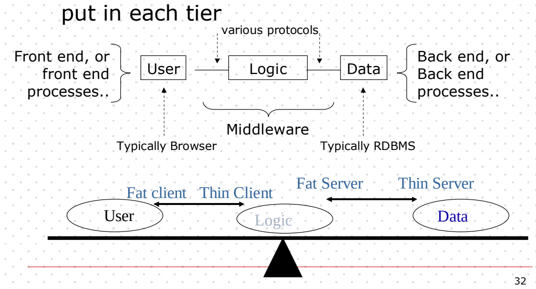
- Generalized client-server
- Further promotes scalability and modifiability
- Addresses some of the shortfalls
  - performance, availability, security
- Generally, a 3-tiered style has:
  - User Tier
  - Logic Tier
  - Data Tier



31

Variations on the 3-Tier Style

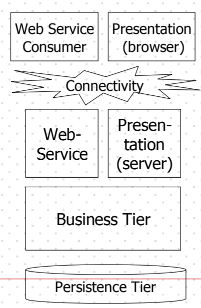
- Many variations in how much functionality you put in each tier



32

The *n*-Tier Architectural Style

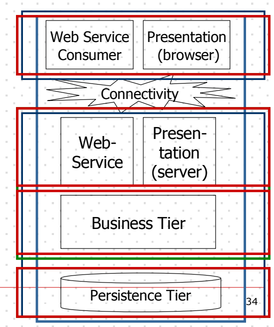
- Responsibilities partitioned into tiers
- Now a standard way to build a web application
  - Presentation
  - Connectivity
  - Web Service/Integration
  - Business
  - Persistence/Data



33

Runtime View vs. Allocation View

- Many ways to map runtime components and connectors to hardware and network topologies



34

Tiered Style Semantics

- Every component is assigned to exactly one tier
- A component in a tier is allowed to require services from components in {any lower, next lower} tier
- A component in a tier {is, is not} allowed to use services from components in same tier

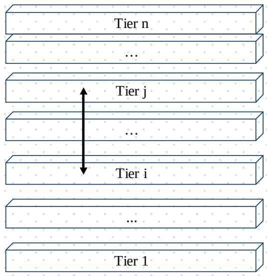
35

Tiered Style Tradeoffs

- Advantages:
  - Supports design based on increasing levels of abstraction
  - Supports enhancement—changes of one layer affects at most the one above & below
  - Supports reuse, portability, …
- Disadvantages:
  - Can be difficult to determine which functionality should go in which tier
  - Performance considerations may require "tunneling" through layers
  - Can be quite difficult to find the right level of abstractions
  - Computations may not fit smoothly into the layers

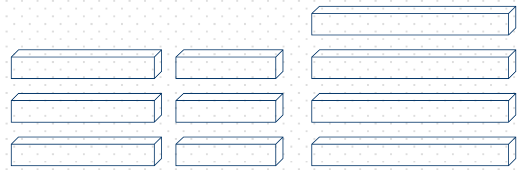
36

Tiered Style: Tunneling/Wormholes



Tiered Style Variations (many)

- Segmented Tiers
  - Dividing a tier into segments, with *allowed-to-use* relations between the segments within a tier and segments between tiers



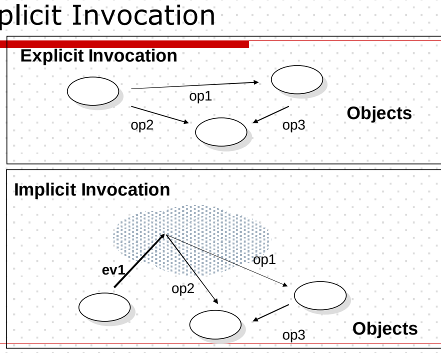
Layered Style (cont'd.)

- What it's for
  - Portability
  - Fielding subsets, incremental development
  - Separation of concerns
- Variations (many)
  - Segmented layers: Dividing a layer into segments (or sub-modules), with *allowed-to-use* relations between the segments within a layer and segments between layers.

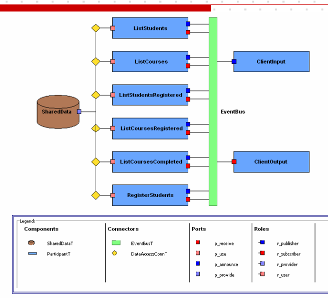
Publish-Subscribe Style

- Components communicate
  - Announcing events
  - Registering for events of interest
- Loose coupling
  - The correctness of a component does not depend on the correctness of any components that receive events it has announced.
  - There may be 0, 1, or many receivers of an event
- Specialization
  - Implicit Invocation: register procedures with events

Implicit Invocation



Implicit Invocation System

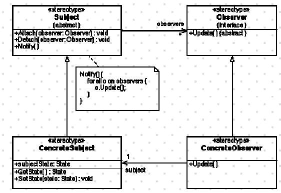


Event Considerations

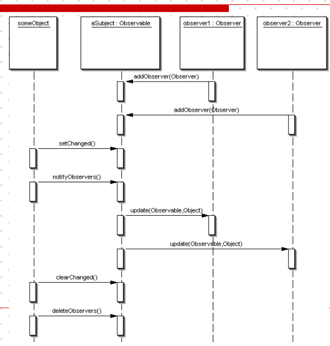
- Event Declarations
  - Who should declare events and where?
- Event Structure
  - How should events be parameterized?
- Event Bindings
  - How/when should events be bound to procedures?
- Event Announcement
  - How should events be announced and dispatched?
- Concurrency
  - Can components operate concurrently?

Recap: Observer

- Applicability
  - When an abstraction has two aspects, one dependent on the other, and you want to reuse each
  - When change to one object requires changing others, and you don't know how many objects need to be changed
  - When an object should be able to notify others without knowing who they are
- Consequences
  - Loose coupling between subject and observer, enhancing reuse
  - Support for broadcast communication
  - Notification can lead to further updates, causing a cascade effect



Implementing Implicit Invocation



Implicit Invocation Tradeoffs

- Advantages:
  - Strong support for reuse
  - Ease of system evolution—components can be replaced without affecting the interfaces of other components
- Disadvantages:
  - Components relinquish control over the computation performed by the system
  - Exchange of data sometimes relies on a shared repository—performance & resource management becomes a serious issue
  - Reasoning about correctness can be problematic

Data Flow Styles

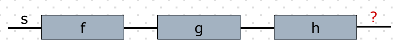
- A data flow system is one in which:
  - the structure of the design is determined by the motion of data from component to component
  - the availability of data controls the computation
  - the pattern of data flow is explicit
  - this is the only form of communication between components
- There are variety of variations on this theme:
  - how control is exerted (e.g., push versus pull)
  - degree of concurrency between processes
  - incrementality of computation
  - topological restrictions (e.g., pipeline)

Data Flow Styles – Elements

- Components: Data Flow Components
  - Interfaces are input ports and output ports
    - Input ports read data; output ports write data
  - Computational model: read data from input ports, compute, write data to output ports
    - Corollary: components do not know the identity of upstream/downstream producers/consumers
- Connectors: Data Streams
  - Unidirectional (usually asynchronous, buffered)
  - Interfaces are reader and writer roles
  - Computational model: transport data from writer roles to reader roles

Data Flow Styles – Elements

- Systems
  - Arbitrary graphs
  - Computational model: Availability of data controls the computation
    - Pick a component that has input and execute it
    - Overall data transformation is the "functional composition" of individual transformations



h(g(f(s)))

Control Flow vs. Data Flow

- Control Flow
  - question is how locus of control moves through the program or system
  - data may follow control, but data is not the driving force of the architecture
- Data Flow
  - dominant question is how data moves through a collection of computational units
  - as data moves, control (and computation) is activated
  - Important note: Data Flow architectural styles are NOT the same as data flow diagrams of traditional structured analysis

Specific Data Flow Styles

- Batch sequential
  - sequential processing steps, run to completion
  - typical of early MIS applications
- Pipe-and-filter
  - incremental transformation of streams
  - typified by Unix
- Process control
  - looping structure to control environment variables

Pipes and Filters

- Components: Filters
  - Incrementally transform some of the source data to sink data
  - Stream-to-stream transformations
  - Preserve no state between instantiations
- Connectors: Pipes
  - Move data from a filter output to a filter input
  - One-way, order-preserving, data-preserving
  - Pipes form data transmission graphs
- Overall Operation
  - Run pipes and filters (non-deterministically) until no more computations are possible.

Pipe-and-Filter: More on Filters

- Stream-to-stream transformations
  - enrich data by computation and adding information
  - refine by distilling data or removing irrelevant data
  - transform data by changing its representation
- Incrementalility
  - data processed as it arrives, not gathered then processed
- Independent entities
  - no external context in processing streams
  - no state preservation between instantiations
  - no knowledge of upstream/downstream filters
  - the correctness of the output should not depend upon the order in which filters execute within pipe-and-filter network, although topology matters

Pipe-and-Filter: More on Pipes

- Pipes move data from a filter output to a filter input (or to a device or file)
  - data moves in one direction
  - pipes form data transmission graphs
  - logically infinitely buffered (in practice usually finitely buffered with flow control, i.e., blocking)
- Overall Operation
  - "do the plumbing"
  - system action is mediated by data delivery

Unix Pipes and Filters

- Filters: Unix processes
  - Built-in ports: "stdin" "stdout" "stderr"
  - Filters usually transform "stdin" to "stdout"
- Pipes: Buffered streams supported by OS
  - Unix pipes can treat files as well as filters as data sources and sinks, but files are passive
  - Unix assumes that the pipes carry ASCII character streams
    - the good news: anything can connect to anything
    - the bad news: everything must be encoded in ASCII, then shipped, then decoded

55

Pipes versus Procedures

	Pipes	Procedures
Arity	Binary	Binary
Control	Asynchronous, data-driven	Synchronous, blocking
Semantics	Functional	Hierarchical
Data	Streamed	Parameter/ return value
Variations	Buffering, end-of-file behavior	Binding time, exception handling, polymorphism

56

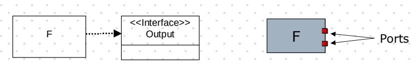
Pipe-and-Filter Style Tradeoffs

- Advantages:
  - Overall input/output behavior is a simple composition of the behavior of individual filters
  - Support reuse
  - Easily maintained and enhanced
  - Permit certain kinds of specialized analysis, e.g., throughput, deadlock analysis
  - Naturally support concurrent executions
- Disadvantages
  - Often lead to batch organization of processing
  - Awkward for handling interactive applications
  - May be hampered by having to maintain correspondences between two separate, but related streams
  - May force a lowest common denominator on data transmission, depending on implementation
    - Loss of performance
    - Increased complexity in writing the filters

57

Module "interface" vs. C&C View Component "interface"

- Consider a filter, F, with two outputs, both of which write characters to a pipe
  - Viewed as a Module both outputs would have the same interface
  - Viewed as a Component the outputs would be different ports, even though their "signatures" are different



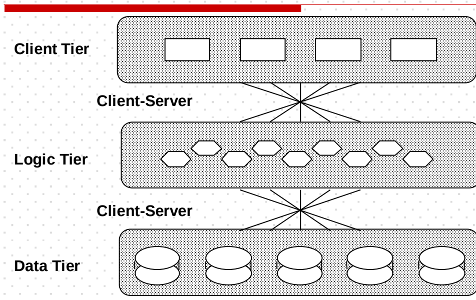
58

Mixing Architectural Styles

- Styles are often used in combination
  Example:
  - Each tier could be defined internally in a different style
  - Each component could have a decomposition in a different style

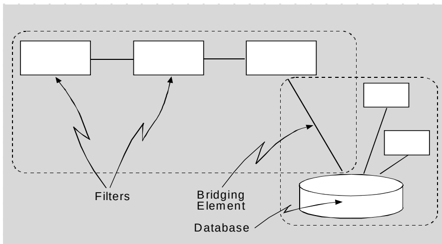
59

Tiered and Client-Server



60

Combining Pipe-Filter with Shared Data with Bridge



Questions?