• Lecture goals
  – Understand importance of s/w architecture
  – Create a vocab. for talking about system structure
The Architecture of Software

• Notional definition:
  – A software system’s architecture is the set of principal design decisions about the system

• Software architecture is the blueprint for a software system’s construction and evolution

• Design decisions encompass every facet of the system under development
  – Structure
  – Functional behavior
  – Interaction
  – Non-functional properties
  – Implementation

Software Architecture Elements

• One view: s/w arch.’s have 3 types of elems
  – Components
  – Connectors
  – Configuration
Components

- Elements that encapsulate processing and data in a system’s architecture
- Definition: A *software comp.* is an arch. entity that
  - Encapsulates system functionality and/or data
  - Restricts access via explicitly defined interfaces
  - Has explicitly defined dependencies on its required execution context
- Components typically provide application-specific services

Connectors

- Notional definition:
  - A *software connector* is an architectural building block that manages component interactions
- Connectors often simple proc. calls or shared data accesses, but much more sophisticated connectors are used in practice
- Connectors typically provide application-independent interaction facilities
Configurations

• Components and connectors are composed in a specific way in a given system’s architecture to accomplish that system’s objective

• Notional definition:
  – An architectural configuration, or topology, is a set of specific associations between the components and connectors of a software system’s architecture

More on Software Connectors

• Connectors often overlooked in architecture discussions
• Connectors manage more than just control flow
  – Data movement
  – Checking interaction protocol
  – Services: persistence, transactions, load balancing, etc.
Some Connector Examples

• Most common connectors
  – Procedure calls
  – Shared memory connectors

• Other types include
  – Message passing connectors
  – Streaming connectors
  – Distribution connectors
  – Wrapper/adaptor connectors

Software Connector Roles

• Connectors can play multiple roles
  – Communication
  – Coordination
  – Conversion
  – Facilitation
Connectors as Communicators

• Main connector role. Supports data transmission:
  – Different communication mechanisms
    • e.g. procedure call, RPC, shared data access, message passing
  – Constraints on communication structure/direction
    • e.g. pipes
• Separates communication from computation
• May influence non-functional system characteristics
  – e.g. performance, scalability, security

Connectors as Coordinators

• Implement transfer of control
• Separates control from computation
• Orthogonal to other roles:
  – Elements of control combined with communication, conversion and facilitation roles
Connectors as Converters

- Enable interaction of independently developed, mismatched components
- Mismatches based on interaction
  - Type
  - Number
  - Frequency
  - Order
- Examples of converters
  - Conversion of data format
  - Wrappers for legacy components

Connectors as Facilitators

- Enable interaction of components intended to interoperate
  - Optimize or streamline interactions
- Ensure proper performance profiles
  - Load balancing or scheduling
- Synchronization mechanisms
  - Monitors
Discussion

• Connectors are fundamental aspect of s/w arch.
• Allow arbitrarily complex interactions
• Connector libraries let developers to focus on application-specific issues
• Connector flexibility aids system evolution & non-functional properties
  – Component addition, removal, replacement, reconnection, migration
  – Reliability, scheduling, availability
Domain-Specific Software Architectures (DSSA)

- 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.
- DSSAs allow maximal reuse of knowledge and prior development when developing a new architectural design.
- Since DSSAs are specialized for a particular domain they are only of value if they closely match application needs.
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 design patterns but applied “at a much higher level”

State-Logic-Display: Three-Tiered Pattern

- Application Examples
  - Business applications
  - Multi-player games
  - Web-based applications
Model-View-Controller (MVC)

- Objective: Separation between information (M), presentation (V) and user interaction (C).
- When a model object value changes, a notification is sent to the view and to the controller.
  - Thus, 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.
Objective: Structuring embedded control applications

Architectural Styles

• Architectural style: a named collection of architectural design decisions that
  – are applicable in a given development context
  – constrain specific architectural design decisions
  – elicit beneficial qualities in the resulting system
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 elements have well-defined meanings

Some Common Styles

- Traditional, lang.-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
  - Mobile code
- Implicit invocation
  - Event-based
  - Publish-subscribe
- Peer-to-peer
The Lunar Lander Example

- A simple computer game from the 1960’s
- Simple concept:
  - You control the descent rate of a 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
- www.frontiernet.net/~imaging/lunar_lander_game.html
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
  - Malleable object internals
  - System decomposition into sets of interacting agents
- Disadvantages
  - Objects must know identities of servers
  - Side effects in object method invocations
Layered Style

• Hierarchical system organization
  – 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 layers “below”

• Connectors are protocols for layer interaction

• Virtual machine style results from fully opaque layers

Advantages of Layered Style

• Increasing abstraction levels
• Evolvability
• Changes in one layer can only affect the adjacent two layers
  – Promotes reuse
• Different layer implementations allowed as long as interface is preserved
• Standardized layer interfaces for libraries and frameworks
Disadvantages of Layered Style

- Not universally applicable
  - Performance
- Layers may have to be skipped
  - Hard to determine the correct abstraction level
Client-Server Style

- Components: clients & servers
- Servers don’t know number/identities of clients
- Clients know server’s identity
- Connectors are based on network protocols

Client-Server LL

Diagram showing the interaction between client, server, and procedure calls.
Data-Flow Styles

- Batch-sequential
- Pipe and Filter

Batch Sequential

- The “Granddaddy of Styles”
  - Separate programs are executed in order; data is passed as an aggregate from one program to the next.
  - Connectors: file accessors
  - Data Elements: Explicit, aggregate elements passed from one component to the next upon completion of the producing program’s execution.
- Typical uses: transaction procession
Batch-Sequential LL

Not a recipe for a successful lunar mission!

Pipe and Filter

- **Components**: 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
Pipe and Filter (cont’d)

- **Advantages**
  - System behavior is the composition of component behaviors
  - Filter addition, replacement, and reuse
    - Possible to hook any two filters together
  - Concurrent execution

Pipe and Filter (cont’d)

- **Disadvantages**
  - Batch organization of processing
  - Interactive applications
  - Lowest common denominator on data transmission
Shared Memory Styles

- Blackboard
- Rule-based
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
  - Speech recognition
  - Planner component for RADRSAT-1 satellite
**Rule-Based Style**

- Inference engine parses user input and determines whether it is a fact/rule or a query
- If a fact/rule, add this entry to the knowledge base
  - Rules often take if (condition) then {action} form
- Otherwise, query the knowledge base for applicable rules and attempt 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
Rule-Based Style (cont’d)

• Pro: can easily modify system behavior through addition or deletion of rules
• Con: as number of rules grows understanding overall system behavior becomes very difficult
Interpreter Style

- Basic Interpreter
- Mobile Code

Basic 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
Basic Interpreter Style

- Highly dynamic behavior possible, where the set of commands is dynamically modified.
- Good for end-user programming; supports dynamically changing set of capabilities
- System architecture remains constant while new capabilities can be added
- Examples: Microsoft Excel, Postscript
Mobile-Code Style

- Data elements become processing comps
- 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

Mobile Code LL

Scripting languages (i.e. JavaScript, VBScript), ActiveX controls, embedded Word/Excel macros, cloud computing
Implicit Invocation

- Publish-Subscribe
- Event-Based

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 explicit or implicit in response to events
- Style invariants
  - “Announcers” unaware of events’ effects
  - No assumption about processing in response to events
Implicit Invocation (cont’d)

- Advantages
  - Component reuse
  - System evolution

- 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
- Connectors:
  - Network protocols. Content-based subscription requires sophisticated connectors.
- Data Elements:
  - Subscriptions, notifications, published information
- Topology:
  - Subscribers connect to publishers or receive notifications from intermediaries

Pub-Sub LL
Event-Based Style

- Independent components asyn. emit and receive events 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

Event-Based Style (cont’d)

- 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
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 to node failure
- Scalable in terms of resources and comp. power
- Resource discovery is key concern
- Security – detecting/handling malicious peers
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?

SW Architecture Examples
Architecture in Action: WWW

• This is the Web

Architecture in Action: WWW

• So is this
Architecture in Action: WWW

• And this

Early WWW Architecture

• Requirements
  – Decentralized hypermedia application
  – Multi-user
  – Multi-owner
  – Heterogeneity: platform and data
  – Scale is a dominating concern

• Representational State Transfer (REST) Arch.
REST Principles

• [RP1] The key abstraction of information is a resource, named by an URL. Any information that can be named can be a resource
• *Resources are logical entities*

REST Principles (cont’d)

• [RP2] The representation of a resource is a sequence of bytes, plus representation metadata to describe those bytes. The particular form of the representation can be negotiated between REST components
• *Client/server model*
• *Representations, not resources, transferred between components*
• [RP3] All interactions are context-free: each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it
• *Interactions are stateless*

• [RP4] Components perform only a small set of well-defined methods on a resource producing a representation to capture the current or intended state of that resource and transfer that representation between components
• *Very strong emphasis on uniform comp. interfaces*
REST Principles (cont’d)

• [RP5] Idempotent operations and representation metadata are encouraged in support of caching and representation reuse.

• Caching is important for scalability

REST Principles (cont’d)

• [RP6] The presence of intermediaries is promoted. Filtering or redirection intermediaries may also use both the metadata and the representations within requests or responses to augment, restrict, or modify requests and responses in a manner that is transparent to both the user agent and the origin server.

• Components are strictly layered
Data Elements

• The Web is a collection of uniquely-named resources. Each resource corresponds to some logical information.
• URL’s are resource IDs used to where the representation of the resource can be found.
• Representations are data + representation metadata + resource metadata + control data.

Connectors

• client  libwww, libwww-perl
• server  libwww, Apache API, NSAPI
• cache  browser cache, Akamai cache network
• resolver  bind (DNS lookup library)
• tunnel  SOCKS, SSL after HTTP CONNECT
Components

- User agent
  - e.g., browser
- Origin server
  - e.g., Apache Server, Microsoft IIS
- Proxy
  - Selected by client
- Gateway
  - Squid, CGI, Reverse proxy
  - Controlled by server

WWW’s Architecture

- Web’s architecture separate from its code
- No single piece of code implements the arch.
- Multiple implementations of a given component
  - E.g., different browsers, servers, proxies, etc.
WWW’s Architecture (cont’d)

- One of the world’s most successful applications is only understood adequately from an architectural vantage point.

Families of Related Products
The Necessity and Benefit of PLs

• Building each of these TVs from scratch would be expensive and error-prone
• Better to reuse structure, behaviors, and component impls across multiple products
  – Simplifies software development
  – Reduces development time and cost
  – Improves system reliability
• Depends on recognizing and exploiting commonality and variability across products

Product-Line Architecture