Software Architecture

- Design patterns focus on “micro structure”
- To build big system we must consider macro structure as well
- Today’s lecture goals
  - Understand importance of software architecture
  - Build vocabulary for talking about architecture
Why Study Software Architecture?

- Analogy: Why do we study building architecture?
  - Buildings are complex and mistakes are hard to repair
  - Good architecture helps in meeting stakeholder needs
  - Architectural knowledge can be abstracted and generalized for use in many buildings
- Similar, but not identical, to issues facing software creators

Some Key Points

- Architectural knowledge is generalizable
  - Good choice reduce cost and improve quality
- Architectural decisions key to achieving certain desirable properties, such as
  - Reliability
  - Extensibility
  - Scalability
Limitations of the Analogy

- We know a lot about buildings, much less about software
- Software is less tangible than buildings
- Software is easier to change than physical materials
- The two “construction industries” are very different
  - Software deployment has no counterpart in building
- Software has more fine-grained dynamism than buildings

Software Architecture

- **Notational 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

• Software architectures have 3 types of elements
  – Components
  – Connectors
  – Configuration

Components

• Elements that encapsulate processing and data in a system’s architecture
• A architecture component is an 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

- For large systems, interaction is often more challenging than functionality
- An architectural connector is an entity that manages component interactions
- Connectors are often just simple procedural calls or shared data accesses,
  - but much more sophisticated connectors exist
- 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
- 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

• You been taught about components, but connectors often overlooked in CS classes
• Connectors manage more than just control flow
  – Data movement
  – Interoperability
  – Enforcing interaction protocols
  – Nonfunctional properties
    • e.g., persistence, transactions, load balancing, code mobility, etc.

Some Connector Dimensions

• Role: What purpose does the connector serve?
• Type: How does the connector operate?
Software Connector Roles

- Connectors can play one or more roles:
  - Communication
  - Coordination
  - Conversion
  - Facilitation

Connectors as Communicators

- Main connector role supports traditional 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
- Often orthogonal to other roles:
  - Elements of control combined with communication, conversion and facilitation roles

Connectors as Converters

- Allow independently developed, but mismatched components to interoperate
- Mismatches based on interaction
  - Type
  - Number
  - Frequency
  - Order
- Examples of converters
  - Conversion of data format
  - Wrappers for legacy components
Connectors as Facilitators

• Improve interaction of components that were intended to interoperate
  – Usually optimizes or streamlines interactions
• Ensure proper performance profiles
  – Load balancing or scheduling
• Synchronization mechanisms
  – Monitors (enforce mutex access to resources)

Connector Types

• Connectors semantics can vary:
  – Procedure call
  – Event
  – Data access
  – Stream
  – Linkage
  – Distributor
  – Arbitrator
  – Adaptor
Procedure Call Connectors

• Traditional function/method call
  – Communication: Transfer data
  – Coordination: Transfer/return control
Event Connectors

- An event is a kind of software signal
- Used for communication & coordination
  - Usually behaves asynchronously
- Virtual connections enabled/disabled at runtime
  - Logically an event:action pair
Data Access Connectors

- Specialized data store accessors
  - Communication & conversion
- Examples: JDBC, WebDAV
Linkage Connectors

- Links components for execution
- Examples include:
  - Make
  - Runtime class loading
  - Binary patching tools
Stream Connectors

- Supports interprocess data transfer
- I/O abstraction plays communication role
- Can be combined with other types, such as data access
Arbitrator Connectors

- Manage interactions between multiple components
- Coordination & facilitation
  - Guarantee data atomicity
  - Preserve service level agreements
  - Fault isolation
Adaptor Connectors

- Allows mismatched components to interact
- Can adapt communication policies, protocols, etc.
- Some examples
  - RPC calls in heterogeneous language environments
  - CORBA wrappers for legacy system components
Distributor Connectors

- Route communications between components
  - Don’t usually exist independent of other connectors
- Ex:
  - DNS
  - Routing protocols
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
Architecture in Action: WWW

• This is the WWW

Architecture in Action: WWW

• So is 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

• Notes/Implications
  – 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

• Notes/Implications
  – Client/server model
  – Representations, not resources, transferred between components
REST Principles (cont’d)

• [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

• Notes/Implications
  – Resource requests are stateless

REST Principles (cont’d)

• [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

• Notes/Implications
  – Strong emphasis on uniform component interfaces
REST Principles (cont’d)

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

• Notes/Implications
  – *Most operations have no effect on resource*
  – *Allows caching, which is important for scalability*

45

REST Principles (cont’d)

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

• Notes/Implications
  – *Components are strictly layered for plug-in compatibility*
WWW Data Elements

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

Connectors

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

- User agent
  - e.g., browser, web crawler
- 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.
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.
DSSA: Philips Television Product Line

Product-Line Architecture

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 Architecture

- 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.
**iPhone Example**

- iPhone apps generally organized around MVC
- Example: TheElements

```objective-c
@interface AtomicElement : NSObject

@property (nonatomic) NSNumber *atomicNumber;
@property (nonatomic) NSString *name;
@property (nonatomic) NSString *symbol;
@property (nonatomic) NSString *state;
@property (nonatomic) NSNumber *group;
@property (nonatomic) NSNumber *period;
@property (nonatomic) NSNumber *vertPos;
@property (nonatomic) NSNumber *horizPos;
@property (BOOL) radioactive;
@property (nonatomic) NSString *atomicWeight;
@property (nonatomic) NSString *discoveryYear;

@end

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)newIndexPath {
    // deselect the new row using animation
    [tableView deselectRowAtIndexPath:newIndexPath animated:YES];
    // get the element that is represented by the selected row.
    AtomicElement *element = [dataSource atomicElementForIndexPath:newIndexPath];
    // create an AtomicElementViewController.
    AtomicElementViewController *elementController = [[[AtomicElementViewController alloc] init] autorelease];
    // set the element for the controller
    elementController.element = element;
    // push the element view controller onto the navigation stack to display it
    [self.navigationController pushViewController:elementController animated:YES];
    [elementController release];
}
```

**Sense-Compute-Control**

**Objective: Structuring embedded control applications**
Lunar Lander

• 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
Architectural Styles

• Architectural style: a named collection of architectural design decisions that
  – are applicable in a given development context
  – constrain concrete architectural design decisions
  – elicit beneficial qualities in the resulting system

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
  – Interpreter
  – Mobile code
• Implicit invocation
  – Event-based
  – Publish-subscribe
• Peer-to-peer
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

Object-Oriented LL

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 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
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: Trans. Proc. in financial systems.

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)

- Variations
  -Pipelines — linear sequences of filters
  -Bounded pipes — limited amount of data on a pipe
  -Typed pipes — data strongly typed
- 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 \texttt{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
Rule Based LL

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
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