Architecting Systems

15-313:
Foundations of Software Engineering

Jonathan Aldrich
Engineering ↔ Business

- Engineering decisions have central impact on business
  - Cell phone: tension between modularity (for adding features) and performance (for meeting real-time deadlines)

- Must trace QA scenarios to business drivers
  - Understanding why a scenario exists explains its importance and whether it can be addressed effectively

- Central role of architecture
  - Bridge between requirements and implementation
Architectural Styles
Architectural Styles

An architectural style is a named collection of architectural design decisions that:

(1) are applicable in a given development context,
(2) constrain architectural design decisions that are specific to a particular system within that context, and
(3) elicit beneficial qualities in each resulting system.

- Taylor et al.

An architectural style typically describes

- A set of element types (e.g. database, pipe, server)
- Topological constraints (e.g. no cycles, layer structure, …)
- A set of semantic constraints (e.g. no sharing between filters)
- A set of interaction mechanisms (e.g. event bus)

- Bass et al.
Example: Pipe and Filter

- **Elements: Filters**
  - Incrementally process streams of data

- **Interactions: Pipes**
  - Asynchronous data streams
    - often buffered

- **Constraints**
  - No cycles in graph

- **Semantics**
  - Filters are independent
    - no shared state
Unix Pipes and Filters

**Filters:** Unix processes
- Built-in ports: “stdin” “stdout” “stderr”
- Filters usually transform “stdin” to “stdout”
- Examples: ls, more, grep, sort, …

**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
Pipe-and-Filter Style Tradeoffs

- Quality Attributes promoted
  - Simplicity / Conceptual Integrity
    - Behavior is a composition of the behavior of individual filters
    - Permit certain kinds of specialized analysis, e.g., throughput, deadlock analysis
  - Reusability of filters (e.g. Unix)
  - Modifiability
    - Filters can be independently modified
    - System can be easily reconfigured
  - Performance: concurrent execution of filters

- Quality Attributes inhibited
  - Performance: may be a lot of copying
    - Especially if passing structured data as a character stream
  - Usability: interactive applications are awkward
Example Styles

- **Pipe and Filter**
- **Client / Server**
- **Layered / Tiered**
- **Repository**
- **Implicit Invocation**

**Not styles (IMHO)**
- Call-return
- Object-oriented

**These are too low-level**
- No system-level structure constraints

**They are technologies for implementing other architectural styles**
- Note: others disagree with me!
Client-Server Style

- **Elements**
  - **Client**: requests a service from a server
  - **Server**: provides a service to a client

- **Interactions**
  - Distributed synchronous call-return connectors

- **Constraints**
  - Two-level structure
  - Typical: one server, many clients
  - Variations: multiple redundant servers, different servers for different services

- **Semantics**
  - Servers do not know identities of clients
  - Clients know identity of server, or else look up in another server
    - Do not know about other clients
  - Application specific connection protocols used
    - Web services: typically XML over HTTP – but lots of other options
Client-Server Tradeoffs

- Quality attributes promoted
  - Scale
    - easy to add more clients
    - easy to add more data

- Quality attributes inhibited
  - reliability?
  - performance?
  - security?
Layers / Tiers

- Clients and Servers are organized into levels, called *layers* or *tiers*
  - Layers typically describes code organization
  - Tiers typically describes deployment
- Each layer provides a set of services to the layers above it
- Each layer relies on services from the layers below it
- A layer 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 layer above
Hierarchical: Virtual Machine

Usually procedures

Composites of various elements

Useful Systems

Basic Utility

Core Level

Users
Examples of Layered/Tiered Architectures?
Layered Operating System

A TIMESHARING SYSTEM
- USER INTERFACE
- SWAPPING
- SPECIAL DEVICES
- SYNCHRONIZATION
- PROCESS MANAGEMENT
- ADDRESS SPACES
- HARDWARE

A BATCH SYSTEM
- JOB CONTROL LANGUAGE
- FILE SYSTEM
- DISK I/O
- PROCESS CREATION
- ADDRESS SPACE CREATION
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
  - Business Logic Tier
  - Data Tier
Variations on the 3-Tier Style

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

Front end, or front end processes...

User ➔ Logic ➔ Data

Middleware

Typically Browser ➔ Typically RDBMS

Back end, or back end processes...

Fat client ➔ Thin Client

Fat Server ➔ Thin Server

User ➔ Logic ➔ Data
The \textit{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
Runtime View vs. Allocation View

• Many ways to map runtime components and connectors to hardware and network topologies
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
Tiered Style Tradeoffs
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
Tiered Style: Tunneling/Wormholes

Tier n
...
Tier j
...
Tier i
...
Tier 1
Repository Style

- **Elements**
  - Repository: stores data
  - Client: creates, reads and updates data

- **Interactions**
  - Procedure call; direct memory access; database access
  - Updates via polling or change notification

- **Constraints**
  - Star topology, with repository in center

- **Semantics**
  - Clients are independent of each other
Repository: Example Uses

- Artificial intelligence
  - Repository is "blackboard" representing knowledge

- IDEs
  - Repository is the code and metadata
Repository Style Tradeoffs

- Quality Attributes promoted
  - Data-driven problem solutions
  - Modifiability: can easily change or add clients
  - Performance good if data is shared

- Quality Attributes inhibited
  - Modifiability of data structures in repository
    - Everyone depends on it!
  - Complexity of coordinating clients
    - Concurrency, dependencies
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

Explicit Invocation

```
Objects

op1

op2

op3
```

Implicit Invocation

```
ev1

op1

op2

op3
```

Objects
Implicit Invocation System

![Diagram of Implicit Invocation System]

<table>
<thead>
<tr>
<th>Components</th>
<th>Connectors</th>
<th>Ports</th>
<th>Roles</th>
</tr>
</thead>
<tbody>
<tr>
<td>SharedData</td>
<td>EventBus</td>
<td>p_score</td>
<td>p_publisher</td>
</tr>
<tr>
<td></td>
<td>DataAccessConn</td>
<td>p_uts</td>
<td>r_subscriber</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p_erreur</td>
<td>r_provider</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p_provider</td>
<td>r_user</td>
</tr>
</tbody>
</table>
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
Implicit Invocation Tradeoffs
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
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
What Style to Choose, and Why?

- Refrigerator?
- Linux?
- IDE?
- eBay?
- Oscilloscope?
- My home page?
Choosing a Style

- Partly Quality Attributes
- Partly Domain
  - Streams of data → Dataflow
  - Persistent data → Repository
  - Web → Client/Server
  - OS → Layered
  - Enterprise Web → 3-Tier
  - Embedded → Communicating processes
Role of Connectors

- Higher-level abstraction
  - (a-)synchronous, buffering, operations, data type, protocol, reliability, encryption, ...
- Examples
  - Pipes, event bus, cache, group membership, byzantine fault tolerance

- Significant intellectual leverage over call-return primitives
  - Similar to design patterns in nature
Architecture is not just Structure

- **Examples of Design Rules**
  - All data will be validated before use [Security]
  - Only the GUI thread will access GUI objects [Threading]
  - All business-relevant data will be stored in a database [Reliability]
  - All operations must yield after 5 ms [Timing]
  - All memory must be statically allocated [Memory use]
  - All communication will use XML [Modifiability]
Designing Architectures
Iterative Refinement [Bass et al.]

- Choose an element to refine
- Determine major architectural drivers
  - Which are most important for this element?
- Choose relevant styles, patterns, and tactics
  - That enhance the relevant attributes
- Decompose element into parts
  - Allocate functionality between them
- Define interfaces for children
- Assign use cases, quality scenarios for children
- Repeat for subcomponents
  - until leave architecture level of abstraction
Hueristics  [Taylor et al.]

- Search for analogies in other domains
  - sewer systems → dataflow
  - biological reactions → repository
- Brainstorm
  - Generate ideas, later evaluate and refine
- Literature search
  - Find architectures for related problems
- Divide and conquer
  - Split problem into parts, solve each part, connect the solutions
    - Careful: functional decomposition may not lead to the quality attributes you want!

- Architecture solutions come from intuition and experience
  - But principles and quality attributes are used to evaluate, refine, and choose among these solutions
Documenting Software Architectures
Architectural Views

- Many possible “views” of architecture
  - Implementation structures
    - Modules, packages
    - Modifiability, Independent construction, …
  - Run-time structures
    - Components, connectors
    - Interactions, dynamism, reliability, …
  - Deployment structures
    - Hardware, processes, networks
    - Security, fault tolerance, …
Why Document Architecture?

- Blueprint for the system
  - Artifact for early analysis
  - Primary carrier of quality attributes
  - Key to post-deployment maintenance and enhancement
- Documentation speaks for the architect, today and 20 years from today
  - As long as the system is built, maintained, and evolved according to its documented architecture
What is Wrong Today?

- In practice today’s documentation consists of
  - Ambiguous box-and-line diagrams
  - Inconsistent use of notations
  - Confusing combinations of viewtypes
- Many things are left unspecified:
  - What kind of elements?
  - What kind of relations?
  - What do the boxes and arrows mean?
  - What is the significance of the layout?
What could the arrow mean?
What could the arrow mean?

Many possibilities:

- A passes control to B
- A passes data to B
- A gets a value from B
- A streams data to B
- A sends a message to B
- A creates B

...
Representing C&C Views

- system
- component
- connector
- port
- role
Guidelines: Avoiding Ambiguity

- **Always include a legend**
- Define precisely what the boxes mean
- Define precisely what the lines mean
- Don’t mix viewtypes unintentionally
  - Recall: Module (classes), C&C (components)
- Supplement graphics with explanation
  - Very important: rationale (architectural intent)
- Do not try to do too much in one diagram
  - Each view of architecture should fit on a page
  - Use hierarchy
Technique: Hierarchy

- Use hierarchy to define elements in more detail in separate views.
- Helps keep an architectural description manageable.
Top-level C&C View

Legend
- Web Component
- LDAP Directory
- RDBMS
- Direct Adapter
- Indirect Adapter
- Controller
- Viewer
- Interface
- SOAP Connector & roles
- LDAP Connector & roles
- DB Connector & roles
- RMI Connector & roles
- Event Connector & roles
- System Boundary
Showing Details of Component
Analyzing Architectures
How to Analyze an Architecture?

- **Completeness**
  - Does the architecture capture all functional requirements, quality attributes, and constraints?
  - Are all relevant elements present and specified?
    - e.g. no components or connections in implementation should be missing

- **Consistency**
  - Do types, behavior, and protocol match for connected elements?
  - If there are multiple views, are they consistent?
  - Is a component consistent with its refinement?

- **Compatibility**
  - Is the architecture compatible with constraints from the chosen style or architecture standard?

- **Correctness**
  - Does the architecture correctly implement the specification?
  - Related: does the code correctly implement the architecture?
Tool-Supported Architectural Analysis

- Typechecking interfaces — various tools
- Protocol checking — Wright ADL
- Style conformance — Aesop, Acme ADLs
- Implementation structure
  - module view: Reflexion Models, others
  - C&C view: ArchJava
- Implementation behavior
  - Rapide ADL (dynamic analysis)
- Schedulability analysis
  - MetaH, Unicon, Acme, and other ADLs
Architecture Tradeoff Analysis Method (ATAM)

- Developed by the CMU Software Engineering Institute (SEI)

- Goals
  - Identify and codify quality attributes
  - Develop architecture-relevant scenarios
  - Identify important architectural decisions, risks, and tradeoffs

- Requirements
  - The system must have an architecture that can be described well enough to evaluate it
  - Commitments to participate by the architect and management
ATAM Phase I Steps

- Present the ATAM
- Present business drivers
- Present the architecture
- Identify architectural approaches
  - Distill quality attributes from the business drivers
  - Identify other architectural drivers
  - Identify tactics and styles in architecture that are important to achieving quality attributes
- Generate quality attribute utility tree
  - Top level: abstract goals like performance, modifiability, …
  - Illustrate each goal with a set of particular business scenarios
  - Repeatedly refine until 6-part scenarios are at leaves of tree

- All presentations & meetings < 1.5 hours!
ATAM Phase I Steps (continued)

- Analyze architectural approaches
  - Consider each quality attribute scenario
  - Walk through scenario, asking architect to explain how architectural elements work together to ensure response is achieved

- Identify
  - Risks: potentially problematic architectural decisions
  - Non-risks: good architectural decisions that may not have been stated explicitly
  - Sensitivity point: property of one or more components that is critical to achieving quality attribute
  - Tradeoff: a property that is a sensitivity point for more than one attribute
ATAM Phase II and Phase III

• Phase II
  • Includes more non-technical stakeholders
  • Generate more scenarios and vote on priorities
  • Reanalyze architectural approaches with more diverse audience
  • Present results to all stakeholders

• Phase III
  • Develop a report for the customer
  • Assess quality of ATAM evaluation