15-413: Software Architecture

Bradley Schmerl

Some material sourced from David Garlan, Anthony Lattanze, George Fairbanks, Jonathan Aldrich, Travis Breaux


The **software architecture** of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both. [DSA10]

**Architecture** is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution. [IEEE 2000]

- **In loose language:**
  - It’s the “big picture” or “macroscopic” organization of the system
  - Not just *what* but *why* the system is the way it is
Rationale for Architecture

- Need for an additional level of abstraction
  - System-wide decomposition
  - Allow teams to work independently on subsystems
  - Specify interfaces before full detailed design is known

- System-wide design constraints
  - Avoid different conventions for each subsystem
    - and therefore avoid interface mismatch

- Promote system-wide quality attributes
  - Design pattern are often at too low a level of abstraction to affect system-level quality attributes
    - e.g. reliability requires end-to-end reasoning
Architecture is an Abstraction

Focus on principal design decisions
- **Structure** - components and connections
- **Behavior** - responsibilities of each component, high level algorithms
- **Interaction** - rules governing how components communicate
- **Quality attributes** - strategy for achieving
- **Implementation** - language, platform, libraries, etc.

Focus on reasoning about the system
- Any decision that impacts key stakeholder concerns or has global impact on the program

Hide unimportant details
- Decisions that are internal to a component
  - i.e. which other components cannot depend on
  - e.g. internal algorithms, data structures, local design patterns
- AND do not impact key stakeholder concerns
Benefits of Architecture

- Aids in communication with stakeholders
  - Shows them “how” at a level they can understand, raising questions about whether it meets their needs
- Defines constraints on implementation
  - Design decisions form “load-bearing walls” of application
- Dictates organizational structure
  - Teams work on different components
- Inhibits or enables quality attributes
  - Similar to design patterns
- Supports predicting cost, quality, and schedule
  - Typically by predicting information for each component
- Aids in software evolution
  - Reason about cost, design, and effect of changes
- Aids in prototyping
  - Can implement architectural skeleton early
Architectural Drivers

- Functional requirements
  - What the system is supposed to do

- Quality attributes
  - How the system does what it does

- Technical constraints
  - Design decisions you have to work around
    - e.g. Google has 4 approved programming languages
    - e.g. Must use platform X to interoperate with legacy code

- Business constraints
  - e.g. must deliver the product in time for Christmas

- Quality attributes typically drive the architecture
  - But allocation of functionality, technical and business constraints cannot be ignored
Quality Attributes

- The system shall be...
  - Modifiable
  - Efficient
  - Reliable
  - Scalable
  - Usable
  - Maintainable

  For what changes? At what cost?
  How much performance is enough?
  What percentage? In what scenarios?
  To what level? At what cost?
  To whom? For what? Is this architectural?
  For what and when? At what cost?

And which of these are most important? e.g. Reliability probably imposes efficiency costs.

- Great, now let’s design!
  - Or are we really ready for that?
Quality Attribute Scenarios [BCK03]

- **Stimulus**: A condition that affects a system
- **Source of stimulus**: The entity that generates the stimulus
- **Environment**: The condition under which the stimulus occurred
- **Artifact**: The artifact that was stimulated
- **Response**: The activity that must result from the stimulus
- **Response measure**: The measure by which the response is evaluated
Availability

Is the system able to provide services to users?
- Often measured as a probability

Issues
- faults $\rightarrow$ failures
  - can intervene to avoid this
- fault/failure detection
- failure notification
- failure recovery
  - how long to repair?
## Availability Scenarios

<table>
<thead>
<tr>
<th>Scenario Portion</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Internal and external sources</td>
</tr>
<tr>
<td>Stimulus</td>
<td>Crash, omission, timing, incorrect response</td>
</tr>
<tr>
<td>Artifact</td>
<td>System’s processors, communication channels, persistent storage, particular processes</td>
</tr>
<tr>
<td>Environment</td>
<td>Normal operation; degraded mode; other specific conditions</td>
</tr>
<tr>
<td>Response</td>
<td></td>
</tr>
<tr>
<td>Measure</td>
<td></td>
</tr>
</tbody>
</table>
## Availability Scenarios

<table>
<thead>
<tr>
<th>Scenario Portion</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Internal and external sources</td>
</tr>
<tr>
<td>Stimulus</td>
<td>Crash, omission, timing, incorrect response</td>
</tr>
<tr>
<td>Artifact</td>
<td>System’s processors, communication channels, persistent storage, particular processes</td>
</tr>
<tr>
<td>Environment</td>
<td>Normal operation; degraded mode; other specific conditions</td>
</tr>
<tr>
<td>Response</td>
<td>Log the failure, notify users/operators, disable source of failure, be unavailable, continue (normal or degraded mode), etc.</td>
</tr>
<tr>
<td>Measure</td>
<td></td>
</tr>
</tbody>
</table>
## Availability Scenarios

<table>
<thead>
<tr>
<th>Scenario Portion</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Internal and external sources</td>
</tr>
<tr>
<td>Stimulus</td>
<td>Crash, omission, timing, incorrect response</td>
</tr>
<tr>
<td>Artifact</td>
<td>System’s processors, communication channels, persistent storage, particular processes</td>
</tr>
<tr>
<td>Environment</td>
<td>Normal operation; degraded mode; other specific conditions</td>
</tr>
<tr>
<td>Response</td>
<td>Log the failure, notify users/operators, disable source of failure, be unavailable, continue (normal or degraded mode), etc.</td>
</tr>
<tr>
<td>Measure</td>
<td>Time interval available, availability %, repair time, unavailability time interval</td>
</tr>
</tbody>
</table>
### Example Availability Scenario

<table>
<thead>
<tr>
<th>Scenario Portion</th>
<th>Scenario-specific Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Internal coding defect</td>
</tr>
<tr>
<td>Stimulus</td>
<td>Crash</td>
</tr>
<tr>
<td>Artifact</td>
<td>Web server process</td>
</tr>
<tr>
<td>Environment</td>
<td>Normal operation</td>
</tr>
<tr>
<td>Response</td>
<td>Restart</td>
</tr>
<tr>
<td>Measure</td>
<td>Unavailable for less than 60 seconds</td>
</tr>
</tbody>
</table>
How to Increase Availability?

- **Architectural Tactic**
  - A “design pattern” for architecture
  - strategy for promoting quality attribute
  - independent of implementation technology
  - independent of exact architectural structure

- **Three kinds of Availability Tactic**
  - Fault detection
  - Fault recovery
  - Fault prevention
## Architectural Tactics (1/2)

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>General Tactic</th>
<th>Specialized Tactic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>Fault Detection</td>
<td>Ping/echo, Heartbeat, Exception</td>
</tr>
<tr>
<td></td>
<td>Recover-Preparation and Repair</td>
<td>Voting, Active Redundancy, Passive Redundancy, Spare</td>
</tr>
<tr>
<td></td>
<td>Recovery-Reintroduction</td>
<td>Shadow, State Resynchronization, Rollback</td>
</tr>
<tr>
<td>Modifiability</td>
<td>Localize Changes</td>
<td>Semantic Coherence, Anticipate Expected Changes, Generalize Module, Limit Possible Options, Abstract Common Services</td>
</tr>
<tr>
<td></td>
<td>Prevention of Ripple Effect</td>
<td>Hide Information, Maintain Existing Interface, Restrict Communication Paths, Use an Intermediary</td>
</tr>
<tr>
<td></td>
<td>Defer Binding Time</td>
<td>Runtime Registration, Configuration Files, Polymorphism, Component Replacement, Adherence to Defined Protocols</td>
</tr>
<tr>
<td>Usability</td>
<td>Separate User Interface</td>
<td>&lt;none&gt;</td>
</tr>
<tr>
<td></td>
<td>Support User Initiative</td>
<td>Cancel, Undo, Aggregate</td>
</tr>
<tr>
<td></td>
<td>Support System Initiative</td>
<td>User Model, System Model, Task Model</td>
</tr>
</tbody>
</table>

Table derived from BCK03 by Fairbanks
## Architectural Tactics (2/2)

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>General Tactic</th>
<th>Specialized Tactic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>Resource Demand</td>
<td>Increase Computation Efficiency, Reduce Computational Overhead, Manage Event Rate, Control Frequency of Sampling</td>
</tr>
<tr>
<td></td>
<td>Resource Management</td>
<td>Introduce Concurrency, Maintain Multiple Copies, Increase Available Resources</td>
</tr>
<tr>
<td></td>
<td>Resource Arbitration</td>
<td>Scheduling Policy</td>
</tr>
<tr>
<td>Security</td>
<td>Resisting Attacks</td>
<td>Authenticate Users, Authorize Users, Maintain Data Confidentiality, Maintain Integrity, Limit Exposure, Limit Access</td>
</tr>
<tr>
<td></td>
<td>Detecting Attacks</td>
<td>Intrusion Detection</td>
</tr>
<tr>
<td></td>
<td>Recovering from an Attack</td>
<td>Restoration (see Availability), Identification &amp; Audit Trail</td>
</tr>
<tr>
<td>Testability</td>
<td>Manage Input / Output</td>
<td>Record / Playback, Separate Interface from Implementation, Specialized Access Routines / Interfaces</td>
</tr>
<tr>
<td></td>
<td>Internal Monitoring</td>
<td>Built-in Monitors</td>
</tr>
</tbody>
</table>

Table derived from BCK03 by Fairbanks
Software Architecture is a representation of the *structure* of the software

- Structures focus on implementation issues
- But, software has many structures, e.g.,
  - Processes/threads
  - Code (classes, modules, methods)
  - Files, data, configuration files

Each of these structures is usually called a *view* of the system

- Software architecture comprises a collection of these views
Architecture Views – An Analogy

A human body is comprised of multiple structures.

One body has many structures, and those structures have many views. So it is with software…
There are many possible views of software.

An architect considers the system in three ways:
- How is the software’s code organized?
- How are the software’s runtime elements organized, and how do they interact?
- How does the software relate to its environment?

This suggests three kinds of views:
- How is the software’s code organized?
- How are the software’s runtime elements organized, and how do they interact?
- How does the software relate to its physical environment?
Elements: Modules. A module is a code unit that implements a set of responsibilities.

Relations: Relations among modules include
- A is part of B: Defines a part-whole relation
- A depends on B: Defines a dependency relation
- A is a B: Defines specialization and generalization (or refinement) relations

Uses:
- Construction: Modules are assigned to teams for implementation.
- Analysis: Traceability and impact analysis; portability and maintainability; project management, budgeting, planning, and tracking.
Module View Example 2

Figure from Fai10
Also known as C&C Viewtype

**Elements:**
- **Components**: principal units of run-time computation and data stores
- **Connectors**: interaction mechanisms

**Relations**: Attachment of components’ *ports* to connectors’ *roles* (interfaces with protocols)

**Uses:**
- **Construction**: Define how system will appear at runtime, and therefore the kinds of behaviors.
- **Analysis**: Dynamic properties of a system
  - E.g., availability, performance, aspects of security, reliability
C&C Structural Concepts

system

component

port

role

connector
C&C View Example

The User, Television, and Stereo are not software components but are included on this diagram for improved comprehension.

User

Component instance
Database component instance
Separate process
Port instance
Messaging connector instance
Shared memory connector instance
Pipe connector instance
Database connector instance
Internet connector instance
File system connector instance

Figure from Fai10
Guidelines: Avoid Ambiguity

- Always include a legend
- Define precisely what the boxes mean
- Define precisely what the lines mean
- Don’t mix view types 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
Allocation (Deployment) Viewtype

- **Elements:**
  - Software elements (as defined in module or C&C styles)
  - Environment elements
    - Machines, processors, sensors, networks, ...

- **Relations:** *allocated-to*

- **Uses:**
  - *Construction:* Where system elements are located on file systems; allocates software elements to organizational work units
  - *Analysis:* Processing and communication node allocation to aide in calculating dynamic properties like performance, availability.
Deployment Diagram

- **User PC**
  - **OS = Linux**
  - **Web Browser**

- **Yinzer Company**
  - **Primary: Data Center**
    - **Yinzer System**
  - **Backup: Data Center**
    - **Yinzer System**

- **Router**

- **internet**
- **intranet**

Environmental Element (Node)
Component Instance
Communication Channel

Figure from Fai10
Software Architecture deals with the high-level design decisions
  - Focus on quality attribute, business, and technical constraints

Quality Attribute Scenarios help to focus on important quality concerns
  - Architectural tactics provide library of possible solutions

Three Architectural Viewtypes used to document software architecture