Software Architecture

15-313: Foundations of Software Engineering

Travis Breaux (with slides by Jonathan Aldrich)
Software Architecture

- 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 [Clements et al. 2010]

- A software system’s architecture is the set of **principal design decisions** made about the system [TMD09]

- **Architectural questions**
  - What are the entities (modules, servers, threads, objects) in a system and connections between them?
    - Does this change over time? If so, how?
  - How are entity interactions constrained?
    - Communication: types, protocols, synchronization
  - System composition and evolution rules
    - Extension rules (e.g. for framework plug-ins)
    - Architectural patterns/styles (e.g. pipe-and-filter, repository, 3-tier)
Design vs. Architecture

**Design Questions**
- How do I add a menu item in Eclipse?
- What lock protects this data?
- How does Google rank pages?
- What encoder should I use for secure communication?
- What is the interface between objects?

**Architectural Questions**
- How do I extend Eclipse with a plugin?
- What threads exist and how do they coordinate?
- How does Google scale to billions of hits per day?
- Where should I put my firewalls?
- What is the interface between subsystems?
Objects [Dahl and Nygaard '67]
Patterns [Gamma, Helm, Johnson, and Vlissides ’ 95]
Patterns [Gamma, Helm, Johnson, and Vlissides ’95]

Diagram:
- Factory → View
- Observer → Model / Subject
- Model / Subject → Controller
- Controller → Command
Patterns [Gamma, Helm, Johnson, and Vlissides ’95]
Patterns [Gamma, Helm, Johnson, and Vlissides ’95]
Architecture [Perry and Wolf 1992] [Garlan and Shaw 1993]
Architecture [Perry and Wolf 1992] [Garlan and Shaw 1993]
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**

*Architecture is design, but not all design is architectural*
Levels of Abstraction

- **Requirements**
  - high-level “what”

- **Architecture**
  - high-level “how”, mid-level “what”

- **Detailed design** (e.g. design patterns)
  - mid-level “how”, low-level “what”

- **Code**
  - low-level “how”
Architecture as Communication

- **What are major system goals?**
  - Stakeholders ↔ Developers
  - Case study: After two days of requirements presentation, a client sees an architecture diagram, realizes he doesn’t understand something, and begins the first productive conversation about what the system is supposed to do

  – Software Architecture in Practice, pp 27-28

- **How are the system goals to be achieved? What guidelines must be followed for independently developed components to work together?**
  - Developers ↔ Developers
Benefits of Architecture [BCK03]

- 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
Business Case: Cell Phones [M. Bass]

- Market is driven by killer products
  - e.g. Razr, iPhone
- Most profit is made at initial release
  - Premium charged on initial sales
  - Drops rapidly when copycats arrive
- Business model
  - Be first to market with new features
- Software quality attributes
  - Ability to change rapidly and at low cost
- True story: effect of architecture
  - Leading cell phone manufacturer
    - not enough new products
    - starts to lose market share, decides to release faster
    - leads to trouble: e.g. tone so loud it damages hearing ➔ recalls
- Analysis
  - software structure did not enable rapid change
  - too costly to rewrite software from scratch
  - eventually left cell phone business entirely
More is Not Necessarily Better [M. Bass]

- **Domain**: mobile infotainment
  - Key quality attribute was Modifiability

- **Architecture**
  - Chose very general, heavyweight framework that promoted modifiability

- **Results**
  - Too much resource use
  - Unstable, overly complex system
  - Full generality of architecture was not needed

- **Lessons**
  - Every architectural decision has a cost
  - Ensure decisions are traceable to a business goal
    - Make quality attributes concrete with scenarios
    - Prioritize quality attribute scenarios
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
Architectural Drivers: Functional Requirements

• Typically specified as use cases

• Architectural role
  • Assign functionality to components
  • Ensure no functionality is forgotten
  • Ensure adequate connections
    • Communication necessary to perform tasks
Architectural Drivers: Business Constraints

- Imposed by the organization, the marketplace, or business issues

- Examples
  - Time to market
  - Cost
  - Available personnel and expertise
  - Required standards & certifications
Architectural Drivers: Technical Constraints

- Externally imposed design decisions

- Examples
  - Language
  - Platform (EJB, ASP.NET)
  - Standards
  - Interoperability with legacy systems
Documenting Constraints [Lattanze/M. Bass]

- Record
  - Rationale for constraint
  - Stakeholder originating constraint
  - Flexibility of constraint
  - Alternatives considered

- Typically no notion of priority, as there is no choice in the matter!
Quality Attributes
Amazon.com

- What quality attribute is relevant here and how does an architecture help?
  - It was unable to give you recommendations
  - Prices were 10% higher than at the competition
  - Your stored credit card information was stolen
  - You had to wait for 30 seconds after each click
Quality Attributes

• The system shall be…
  • Modifiable     For what changes? At what cost?
  • Efficient      How much performance is enough?
  • Reliable       What percentage? In what scenarios?
  • Scalable       To what level? At what cost?
  • Usable         To whom? For what? Is this architectural?
  • Maintainable   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 → 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
Availability Tactics: Fault Detection

- ping/echo
  - ping another component
  - expect an echo before a timeout

- heartbeat
  - expect periodic message

- exceptions
  - detect generated exception
Availability Tactics: Fault Recovery

- voting
  - multiple components produce answer
  - give client the answer with the most votes
  - most useful for hardware failures
    - buggy software will fail in the same way
    - occurs even if built by different teams!

- active replicas
  - all replicas respond to all messages

- passive replicas
  - passive replicas periodically updated with current state
  - requires limited replay

- spare
  - must boot, load checkpoint, and replay recent messages

- checkpoint/rollback
  - allows undoing operations after a failure
Availability Tactics: Fault Prevention

• remove from service
  • e.g. reboot a component that’s getting low on memory
    • surprisingly effective for OS drivers

• transactions
  • avoids failures/inconsistencies when part of an operation fails

• process monitor
  • detect fault in running process, then restart and reinitialize before errors propagate
Why not do it all?

- Cost
- Complexity
- Compromise of other quality attributes
  - Is a highly available system as fast? Is it as modifiable?
Modifiability

- What is the cost of change?

- Issues
  - What is changing?
    - functions, platforms, hardware, protocols…
    - quality attributes
  - Who changes it?
  - When is it changing?
## Modifiability Scenarios

<table>
<thead>
<tr>
<th>Scenario Portion</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>End-user, developer, system-administrator</td>
</tr>
<tr>
<td>Stimulus</td>
<td>Add/delete/modify functionality or quality attributes</td>
</tr>
<tr>
<td>Artifact</td>
<td>System user interface, platform, environment</td>
</tr>
<tr>
<td>Environment</td>
<td>At runtime, compile time, build time, design-time</td>
</tr>
<tr>
<td>Response</td>
<td>Locate places in architecture for modifying, modify, test modification, deploys modification</td>
</tr>
<tr>
<td>Measure</td>
<td>Cost in effort, money, time, extent affects other system functions or qualities</td>
</tr>
</tbody>
</table>
Modifiability Tactics

- Localize modifications
  - Modifications affect the requirements of as few modules as possible

- Prevent ripple effects
  - Limit effect of changing one module on other modules

- Defer binding time
  - Allow modifications late in process at low cost
Modifiability Tactics: Localization

- Maintain semantic coherence
  - group responsibilities that are likely to change together in the same module

- Hide information based on anticipated changes
  - organize architecture to minimize number of modules affected by specific changes

- Generalize the modules
  - the more general the module’s interface, the more likely changes in requirements won’t require changing the interface or the implementation
  - BUT – generality creates complexity and cost, so only use it if you need it

- Limit possible options
  - Restrict the set of possible changes so that you can plan better for the supported changes
Modifiability Tactics: Prevent Ripples

- **Key issue: Notion of interfaces**
  - types and signatures, semantics, sequences
  - identity, location, existence
  - quality of service, resource use

- **Tactics**
  - Hide information (like the above) within a module
  - Maintain interfaces
    - Extend in backwards compatible way, add an adapter
  - Restrict communication paths
    - Fewer connections along which ripples can propagate
  - Add an intermediary
    - Convert data types, replace interfaces with a proxy
    - Hide identity using broker, location using name server
    - Resource manager hides resource behavior
Modifiability Tactics: Deferring Binding

• Event registration
  • Plug-and-play connection between components
  • Cost in understandability

• Polymorphism
  • Late binding of method calls
  • Cost in performance

• Configuration files
  • Bind at deployment
  • Cost in complexity
Performance

• How long does it take the system to respond to an event?
  • Generalizes to throughput, etc.

• Issues
  • Sources of events
    • end users, interrupts, timers, messages
  • Arrival patterns
    • periodic, sporadic, stochastic
  • Response criteria
## Performance Scenarios

<table>
<thead>
<tr>
<th>Scenario Portion</th>
<th>Example Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>A number of sources, both external and internal</td>
</tr>
<tr>
<td>Stimulus</td>
<td>Particular events, which may be periodic, sporadic, stochastic, etc.</td>
</tr>
<tr>
<td>Artifact</td>
<td>System or one of its components</td>
</tr>
<tr>
<td>Environment</td>
<td>Normal mode; overload mode, or other particular conditions</td>
</tr>
<tr>
<td>Response</td>
<td>Process stimuli; change level of service</td>
</tr>
<tr>
<td>Measure</td>
<td>Latency, deadline, throughput, jitter, miss rate, data loss</td>
</tr>
</tbody>
</table>
Performance Tactics

• Fundamental issues
  • Resource use
    • computation, memory, bandwidth, system-specific resources
  • Blocking for resources
    • contention, availability, dependencies

• Strategies
  • Reduce resource demand
    • Increase efficiency, lower performance expectations
  • Increase resources
    • Throw money at the problem
  • Coordinate resources
    • Share more efficiently
Performance Tactics: Reduce Demand

- Increase efficiency
  - Better algorithms
  - Trade space for time
- Reduce overhead
  - Avoid costly operations, e.g. indirection
    - Typically conflicts with other quality attributes!
- Reduce event rate
  - Sample environmental data less frequently
    - May reduce precision
- Discard events
  - Sample from input stream
    - Cost: some requests are lost
- Bound execution time
  - e.g. number of iterations in problem
    - Cost: solution may be less precise
Performance Tactics: Increase Resources

• Introduce concurrency
  • Adds complexity, but gets things done faster if there is inherent parallelism

• Duplicate data or computation
  • Avoid contention for shared resources
  • Cache data from a remote server

• Buy more resources
  • Faster CPU, more CPUs, more memory, faster network
  • Only barrier is $$$
    • So, does the value of the additional performance exceed its cost?
Performance Tactics: Coordinate Resources

• Typically scheduling strategies
  • FIFO
    • fair when all requests are equivalent
  • fixed priorities
    • most important
    • shortest deadline
  • dynamic priorities
    • earliest deadline first
  • static scheduling
    • allocate everything up front
    • good for real-time / embedded systems
Security

• Ability to resist attack while remaining functional

• Issues
  • Confidentiality – can’t see private data
  • Integrity – can’t modify data without permission
  • Nonrepudiation – can’t deny malicious action
  • Availability – no denial of service
## Security Scenarios

<table>
<thead>
<tr>
<th>Scenario Portion</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>User/system that is identified correctly/incorrectly or unknown, who is internal/external and authorized or not, with full/limited access</td>
</tr>
<tr>
<td>Stimulus</td>
<td>Attempt to display/modify data; access services; reduce availability</td>
</tr>
<tr>
<td>Artifact</td>
<td>System services, data</td>
</tr>
<tr>
<td>Environment</td>
<td>Online/offline, connected/disconnected, firewalled/open</td>
</tr>
<tr>
<td>Response</td>
<td>Grant/block access; audit requests; encrypt data; detect attack; enter degraded mode</td>
</tr>
<tr>
<td>Measure</td>
<td>Cost/benefit of attack to attacker, cost of attack to defender and users; probability of identifying attacker; availability during DOS attack; …</td>
</tr>
</tbody>
</table>
Security Tactics: Resisting Attacks

- **Authentication**
  - User identification – often username/password
- **Authorization**
  - Which users can perform which operations?
- **Encryption**
  - Protect confidential data
- **Checksums / signatures**
  - Ensure integrity of data
- **Principle of least privilege**
  - Reduce damage attacker can do
- **Limit access**
  - Firewalls, etc.
More Quality Attributes

- Security
- Testability
- Usability
- Interoperability
- Scalability (Performance? Modifiability?)
- Portability (Modifiability?)
Architecture “Hoisting”

• Building a quality attribute into the architecture
  • Example: Map-reduce
    • Scalability, reliability built in to model
  • Example: Rackspace email log analysis
    • Version 1: ssh to machines and scan log files
      • +modifiability, +latency, -usability, -scaleability
    • Version 2: machines write to central database
      • +modifiability, +latency, +usability, ? scaleability
    • Version 3: machines write to distributed database
      • +modifiability, -latency, +usability, +scaleability
Architectural Views
Examples of Architecture Descriptions

Figure 1. The NIST/ECMA reference model.
Figure 2. Display PostScript interpreter components.

<table>
<thead>
<tr>
<th><strong>Client Layer</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Access domain management</td>
</tr>
<tr>
<td>Buffering and record-level I/O</td>
</tr>
<tr>
<td>Transaction coordination</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Agent Layer</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation of standard server interface</td>
</tr>
<tr>
<td>Logger, agent, and instance tasks</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Helix Directories</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Path name to FID mapping</td>
</tr>
<tr>
<td>Single-file (database) update by one task</td>
</tr>
<tr>
<td>Procedural interface for queries</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Object (FID directory)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Identification and capability access (via FIDs)</td>
</tr>
<tr>
<td>FID to tree-root mapping; table of (FID,root,ref_count)</td>
</tr>
<tr>
<td>Existence and deletion (reference counts)</td>
</tr>
<tr>
<td>Concurrency control (file interlocking)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Secure Tree</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic crash-resistant file structure</td>
</tr>
<tr>
<td>Conditional commit</td>
</tr>
<tr>
<td>Provision of secure array of blocks</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>System</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Commit and restart authority</td>
</tr>
<tr>
<td>Disk space allocation</td>
</tr>
<tr>
<td>Commit domains</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Cache</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Caching and performance optimization</td>
</tr>
<tr>
<td>Commit support (flush)</td>
</tr>
<tr>
<td>Frame allocation (to domains)</td>
</tr>
<tr>
<td>Optional disk shadowing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Canonical Disk</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical disk access</td>
</tr>
</tbody>
</table>

*Also called client Helix.

**Figure 2. Abstraction layering.**

Figure 3.1 The Configuration of the GENESIS Prototype

Genesis: A Reconfiguration Database Management System, D. S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, T.E. Wise, Department of Computer Sciences, University of Texas at Austin,
FIGURE 7. Flight Computer Operating System (The FCOS dispatcher coordinates and controls all work performed by the on-board computers.)

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
  - A occurs before B
  - …
Architectural Views

• Many possible “views” of architecture
  • Code structures
    • Modules, packages
    • Modifiability, Independent construction, …
  • Run-time structures
    • Components, connectors
    • Interactions, dynamism, reliability, …
  • Deployment structures
    • Hardware, processes, networks
    • Security, fault tolerance, …
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)
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 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
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 Bus Connector & roles
- System Boundary
Showing Details of Component
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]
Risk-Driven Architecture [Fairbanks]

Q: How much architecture should you do?
A: Enough to address your most critical risks
  • Distributed development -> clear documentation
  • Security -> data flow diagrams and attack analysis
  • Scalability -> replication strategy and queuing analysis
  • Modifiability -> design patterns and change impact analysis

What are the main risks in a game framework?
  • And what should architecture work focus on to address these?