Foundations of Software Engineering

Lecture 9: Architecture Documentation, Patterns, and Tactics

Christian Kaestner
Learning Goals

• Use notation and views to describe the architecture suitable to the purpose
• Document architectures clearly, without ambiguity
• Understand the benefits and challenges of traceability.
• Understand key parts of architectural process
• Use architectural styles and tactics for design decisions
• Make justified architectural decisions for new systems and within existing systems
Architecture vs Object-Level Design
214 Review: Design

- Design process (analysis, design, implementation)
- Design goals (cohesion, coupling, information hiding, design for reuse, ...)
- Design patterns (what they are, for what they are useful, how they are described)
- Frameworks and libraries (reuse strategies)
Levels of Abstraction

- Requirements
  - high-level “what” needs to be done

- Architecture (High-level design)
  - high-level “how”, mid-level “what”

- OO-Design (Low-level design, e.g. design patterns)
  - mid-level “how”, low-level “what”

- Code
  - low-level “how”
Design vs. Architecture

Design Questions

• How do I add a menu item in Eclipse?
• How can I make it easy to add menu items 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?
Architecture Documentation & Views
Architecture Disentangled

Architecture as structures and relations
(the actual system)

Architecture as documentation
(representations of the system)

Architecture as (design) process
(activities around the other two)
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

• Support traceability.
Common Views in Documenting Software Architecture

• Static View
  – Modules (subsystems, structures) and their relations (dependencies, ...)

• Dynamic View
  – Components (processes, runnable entities) and connectors (messages, data flow, ...)

• Physical View (Deployment)
  – Hardware structures and their connections
Views and Purposes

• Every view should align with a purpose
• Different views are suitable for different reasoning aspects (different quality goals), e.g.,
  – Performance
  – Extensibility
  – Security
  – Scalability
  – …
Figure 1. The NIST/ECMA reference model.
Figure 2. Display PostScript interpreter components.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Functions/Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client Layer</td>
<td>Access domain management, Buffering and record-level I/O, Transaction coordination</td>
</tr>
<tr>
<td>Agent Layer</td>
<td>Implementation of standard server interface, Logger, agent, and instance tasks</td>
</tr>
<tr>
<td>Helix Directories</td>
<td>Path name to FID mapping, Single-file (database) update by one task, Procedural Interface for queries</td>
</tr>
<tr>
<td>Object (FID directory)</td>
<td>Identification and capability access (via FIDs), FID to tree-root mapping; table of (FID,root,ref_count), Existence and deletion (reference counts), Concurrency control (file interlocking)</td>
</tr>
<tr>
<td>Secure Tree</td>
<td>Basic crash-resistant file structure, Conditional commit, Provision of secure array of blocks</td>
</tr>
<tr>
<td>System</td>
<td>Commit and restart authority, Disk space allocation, Commit domains</td>
</tr>
<tr>
<td>Cache</td>
<td>Caching and performance optimization, Commit support (flush), Frame allocation (to domains), Optional disk shadowing</td>
</tr>
<tr>
<td>Canonical Disk</td>
<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

Legend:
- □ module or program
- ○ schema or tables
- ▶ ▶ ▶ calls B
- ▶ ▶ ▶ data path

Bash Component Architecture

Example source: http://www.aosabook.org
The RPython Translator, Translation steps:

1. Flow object space
2. Annotator
3. RTyper
4. Backend Optimizations
5. Garbage collector and exception transformation
6. C source generation
7. Python bytecode interpreter
Moodle: Typical university systems architecture – Key subsystems
Selecting a Notation

• Suitable for purpose
• Often visual for compact representation
• Usually boxes and arrows
• UML possible (semi-formal), but possibly constraining
  – Note the different abstraction level – Subsystems or processes, not classes or objects
• Formal notations available
• Decompose diagrams hierarchically and in views
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?
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
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
  - B gets its electricity from A
  - ...
Recommendations for Recitation and Homework

• Use UML or UML-like notations:
  – Class diagrams for static and physical views
  – Communication diagrams for dynamic view
  – Use correct abstraction level (usually not classes/objects)

• Extend notation as needed
  – Provide a legend explaining the extensions or deviations from standard UML notation
Case Study: The Google File System
Assumptions

- The system is built from many inexpensive commodity components that often fail.
- The system stores a modest number of large files.
- The workloads primarily consist of two kinds of reads: large streaming reads and small random reads.
- The workloads also have many large, sequential writes that append data to files.
- The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file.
- High sustained bandwidth is more important than low latency.

Questions

1. What are the most important quality attributes in the design?
2. How are those quality attributes realized in the design?
Qualities:
- Scalability
- Reliability
- Performance
- Cost

Exercise

For the Google File System, create a physical architecture view that addresses a relevant quality attribute.
Traceability
Traceability - Definition

"The ability to interrelate any uniquely identifiable software engineering artifact to any other, maintain required links over time, and use the resulting network to answer questions of both the software product and it's development process" – CoEST
Traceability in Requirements?
Quality Goal: High Availability

Req.: Should run on redundant servers

Architecture: Fault recovery with voting

Fault prevention: Regular restarts

OO Design/Impl.: Voting mechanism, socket communication

Test: Killing random servers in test setup
Traceability Compliance

• Traceability required in some domains (avionics)
  – Why does X piece of code exist?
• "Enable verification of the absence of undocumented source code and verification of the complete implementation of the low-level requirements"
• Link to specifications and test procedures
Traceability and Architecture

• Architecture links quality attributes to the high-level and low-level system design
• Ensures quality attributes often not even visible in code
• Cost, effort, discipline needed to create and maintain.
  – Often incomplete, incorrect, outdated
• Developers hate it, and often do not understand the need.
  – "Unnecessary evil"
So far in course

Requirements

Implementation

Architecture

<table>
<thead>
<tr>
<th>Feature</th>
<th>Definition</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard-based and</td>
<td>Messaging protocol must be based on industry standards to enable interoperability</td>
<td>Shopping</td>
</tr>
<tr>
<td>Interoperable messaging</td>
<td></td>
<td></td>
</tr>
<tr>
<td>protocol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send Only</td>
<td>Also called Push MEP is simple one-way messaging where a message is sent with no expectation response.</td>
<td></td>
</tr>
<tr>
<td>Receive only</td>
<td>Also called Pull MEP is a message pattern where a non-addressable sender supports the ability to explicitly obtain messages from another application. This can be used for exchanges.</td>
<td></td>
</tr>
<tr>
<td>Request/Response</td>
<td>Message pattern consists of one or more request/response pairs. The correlation between request and a response is well defined. In this response, maybe deferred and the requesting application may or may not block application processing until a response is received.</td>
<td></td>
</tr>
<tr>
<td>exchange</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diagnostics</td>
<td>Authentication, diagnostic, logging, &amp; routing information should be included in the message and not the payload.</td>
<td></td>
</tr>
<tr>
<td>Reliability</td>
<td>Protocol capability to support assured and single delivery to the receiving application with no loss.</td>
<td></td>
</tr>
</tbody>
</table>
Levels of abstraction

- Requirements
  - high-level “what” needs to be done

Architecture (High-level design)
  - high-level “how”, mid-level “what”

OO-Design (Low-level design, e.g. design patterns)
  - mid-level “how”, low-level “what”

- Code
  - low-level “how”
What is architecture?

Architecture as structures and relations (the actual system)

Architecture as documentation (representations of the system)

Architecture as process (activities around the other two)
Architectural Styles and Tactics
Architectural style (pattern)

• Broad principle of system organization
• Describes computational model
  – E.g., pipe and filter, call-return, publish-subscribe, layered, services
• Related to one of common view types
  – Static, dynamic, physical
Example Architectural Patterns

• System organization
  – Repository model
  – Client-server model
  – Layered model

• Modular decomposition
  – Object oriented
  – Function-oriented pipelining

• Control styles
  – Centralized control
  – Event-driven systems
Architectural style (pattern)

• Broad principle of system organization
• See reading
Architectural style (pattern)

Source: codeproject.org
Client-server pattern

• Separation of clients and servers
  – Servers provide services; known and “stable”
  – Clients request services; come and go
• Varieties: synchronous/asynchronous
• Impact on security, performance, scalability
• Examples: TCP, HTTP, X11
Client

Server

Database

Where to validate user input?

Example: Yelp App
Client-server style

Source: wikimedia commons
Layered system

Source: eclipse.org
Tiered architecture

Client browser

Web server

Webapp (e.g. ruby program)

Database

Client computer

Internet

Web server

LAN

App server

LAN

DB server

https

xml

sql

stream

Request and response

Deployed on

Wired connection
Figure 1: GFS Architecture

Tactics

• Architectural techniques to achieve qualities
  – More tied to specific context and quality
• Smaller scope than architectural patterns
  – Problem solved by patterns: “How do I structure my (sub)system?”
  – Problem solved by tactics: “How do I get better at quality X?”
• Collection of common strategies and known solutions
  – Resemble OO design patterns
Example Tactic Description: Record/playback

• Record/playback refers to both capturing information crossing an interface and using it as input into the test harness. The information crossing an interface during normal operation is saved in some repository and represents output from one component and input to another. Recording this information allows test input for one of the components to be generated and test output for later comparison to be saved.
Example Tactic Description: Built-in monitors

- The component can maintain state, performance load, capacity, security, or other information accessible through an interface. This interface can be a permanent interface of the component or it can be introduced temporarily via an instrumentation technique such as aspect-oriented programming or preprocessor macros. A common technique is to record events when monitoring states have been activated. Monitoring states can actually increase the testing effort since tests may have to be repeated with the monitoring turned off. Increased visibility into the activities of the component usually more than outweigh the cost of the additional testing.
Modifiability

Localize Changes
- Semantic Coherence
- Anticipate Expected Changes
- Generalize Module
- Limit Possible Options
- Abstract Common Services

Prevention of Ripple Effect
- Hide Information
- Maintain Existing Interface
- Restrict Communication Paths
- Use an Intermediary

Defer Binding Time
- Runtime Registration
- Configuration Files
- Polymorphism
- Component Replacement
- Adherence to Defined Protocols

Changes Arrive

Changes Made, Tested, and Deployed Within Time and Budget
Performance

- Resource Demand
  - Increase Computation Efficiency
  - Reduce Computational Overhead
  - Manage Event Rate
  - Control Frequency of Sampling

- Resource Management
  - Introduce Concurrency
  - Maintain Multiple Copies
  - Increase Available Resources

- Resource Arbitration
  - Scheduling Policy

Events Arrive

Response Generated Within Time Constraints
Security

Resisting Attacks
- Authenticate Users
- Authorize Users
- Maintain Data Confidentiality
- Maintain Integrity
- Limit Exposure
- Limit Access

Detecting Attacks
- Intrusion Detection

Recovering from an Attack
- Restoration
- Identification
- See Availability
- Audit Trail

System Detects, Resists, or Recovers from Attacks
Completion of an Increment

Testability

Manage Input/Output
- Record/Playback
- Separate Interface from Implementation
- Specialized Access Routines/Interfaces

Internal Monitoring
- Built-in Monitors

Faults Detected
Many tactics described in Chapter 5

Brief high-level descriptions (about 1 paragraph per tactic)

Second and more detailed third edition available as ebook through CMU library.
Architecture Design Process
What is architecture?

Architecture as structures and relations
(the actual system)

Architecture as documentation
(representations of the system)

Architecture as process
(activities around the other two)
Architecture design process

• Choose part or whole system to focus on
• Understand relevant requirements
• Choose a notation
  – Type of view, vocabulary of elements
• Create a design
  – Patterns, tactics
• Evaluate
• Go vs no-go
  – Issues feed back into process
Architecture design process

Choose scope → Understand relevant requirements → Choose a notation → Create/refine a design → Evaluate

“Go”

Source: ACDM, ADD
Architectural decisions

• Heart of architecture – deciding which path to go
• Involve tradeoff analysis
• Representing the alternatives clearly – half of work
Architectural decisions

• Software architecture is *design*
  
  “Engineering design is [...] a decision-making process (often iterative), in which the basic sciences, mathematics, and engineering sciences are applied to convert resources optimally to meet a stated objective” – ABET

• A decision is a step in the process
  
  – Record rationale! (not just diagrams)
  – Tradeoffs
  – Backtracking
Architecture design process

Choose scope → Understand relevant requirements → Choose a notation → Create/refine a design → Evaluate

Source: ACDM, ADD
Architectural decisions

• Software architecture is design

  “Engineering design is [...] a decision-making process (often iterative), in which the basic sciences, mathematics, and engineering sciences are applied to convert resources optimally to meet a stated objective” – ABET

• A decision is a step in the process
  – Record rationale! (not just diagrams)
  – Tradeoffs
  – Backtracking
Architecture evaluation

• Goal: does the architecture satisfy requirements?

• ATAM – Architecture Tradeoff Analysis Method
  – Present requirements
  – Present architecture
  – Analyze architecture
  – Present results – risks and non-risks
Performance
  - Data Latency
    - (M,L) Minimize storage latency on customer DB to 200 ms.
    - (H,M) Deliver video in real time
  - Transaction Throughput

Modifiability
  - New product categories
    - (L,H) Add CORBA middleware in < 20 person-months
  - Change COTS
    - (H,L) Change web user interface in < 4 person weeks

Utility
  - Availability
    - H/W failure
      - (M,M) Restart after disk failure in < 5 mins
    - COTS S/W failures
      - (H,M) Network failure is detected and recovered in < 1.5 mins

Security
  - Data confidentiality
  - Data integrity
    - (L,H) Credit card transactions are secure 99.999% of time
    - (L,H) Customer database authorization works 99.999% of time
Athena – code review system

Source: Jansen and Bosch 2005
Source: Jansen and Bosch 2005
Architecture design process

Choose scope → Understand relevant requirements → Choose a notation → Create/refine a design → Evaluate

Source: ACDM, ADD
Challenges of architecting

• Describe the system that is not built yet
• Domain knowledge is essential
• Huge space of options
• Heavily reliant on judgment
How much architecture?

• Design and document when needed, based on risk

• When:
  – Beginning
  – Whenever circumstances change

• Agile
How much architecture?

• YAGNI
• Risk
• When to start:
  – Before implementation
  – Circumstances change
• When to stop:
  – Well-defined, requirements addressed, passes evaluation
Source: Boehm and Turner 2003
Student application system

Source: Boehm and Turner 2003
Manned space mission software

Source: Boehm and Turner 2003
Challenges of architecting

• Describe the system that is not built yet
• Domain knowledge is essential
• Huge space of options
• Heavily reliant on judgment
Summary

Architecture as structures and relations
• Patterns
• Tactics

Architecture as process
• Decisions
• Evaluation
• Reconstruction
• Agile

Architecture as documentation
• Views
• Rationale
References

Further Readings