From Requirements to Software Architecture

OOP 2010 Tutorial

Dr. Michael Stal

© Dr. Michael Stal, 2010

Agenda

- Part I: Software Architecture and Architects
- Part II: Requirements
- Part III: Creating the Software Architecture
- Part IV: Documenting the Architecture
- Part V: Architecture Quality
- Part VI: Refactoring and Reengineering
- Part VII: Architecture Reviews
- References
Secret to success

There are no secrets to success. It is the result of preparation, hard work, and learning from failure.

[Colin Powell, Former US Secretary of State]
Essential Software Architecture

Agenda

- Motivation
- What is Software Architecture
- Role of a Software Architect
- Summary
From Requirements to Software Architecture

Software Architecture is the cure to all Software Engineering problems

Software Architecture makes all your dreams come true. It auto magically provides you with

- flexibility;
- maintainability;
- scalability;
- stability;
- performance;
- openness;
- <enter your requirements here>

Satisfaction guaranteed!

Unfortunately ...

This is plain wrong.

Why do only 30% of all software projects succeed?

Didn’t they have an architecture?

Is building quality software architectures a complicated art that only rocket scientists and gurus can master?

Or are we simply missing something?
A Typical Story

But what are the consequences of this observation?

... are all those technologies just crap?
... do we need better technologies?
... do we need better manuals for using our technologies?
... do our systems have an appropriate architecture?

Lessons Learned

To create an architecture is about thinking and deciding, and not about using tools and technologies

- In other words, you as an architect need to craft an appropriate software architecture!
- The best way to do so is strictly following a requirements-, test- and risk-driven approach.
And of course: Learning from Failure (see parallel tutorial by Frank)

Henry Petroski:

- At no time in history, any successful technology was able to scale beyond certain limits!
- Scaling into new “dimensions” always requires new technologies!
- Failure and understanding failure is a key factor for successful design!

Essential Software Architecture

Agenda

- Motivation
- What is Software Architecture
- Role of a Software Architect
- Summary
A Short History of Time

- Software architecture has always been about reducing complexity by separation of concerns and abstraction
- The origin of software architecture as a concept was first identified in the research work of Edsger Dijkstra in 1968 and David Parnas in the early 70s
- Increased popularity by “Pattern Movement” in the 90s
- One of the fundamental books: Mary Shaw and David Garlan (Carnegie Mellon); Software Architecture: Perspectives on an Emerging Discipline in 1996
- Mantra of Software Architecture is strategic design (think global! …) in contrast to tactical design (… but act local!)

Yes, but what is it?

- ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems:
  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.

  - … organization of a system in terms of components; e.g., a web shop contains (among other components) a payment subsystem and a product catalog
  - … relationships to each other and the environment; e.g., the payment system uses an external credit card payment subsystem
  - … principles governing its design and evolution; e.g., the strategy pattern is used to vary different payment options such as credit card payment, PayPal

  "Architecture is about the important things" [Martin Fowler]
  "Architecture is about everything costly to change" [Grady Booch]
Architecture versus Design (according to Len Bass)

- Design is a continuous activity of making decisions beginning with a collection of decisions that have broad system wide scope and moving to a collection of decisions that have very narrow scope.
- I would characterize a decision as architectural if it has one or more of the following properties:
  - it has system wide impact
  - it affects the achievement of a quality attribute important to the system

Strategic and Tactical Design

- Strategic design focuses on global system scope
- At the beginning consider only strategic requirements, i.e., requirements with systemic and strategic impact:
  - All functional requirements
  - All operational requirements
- Tactical Design encompasses all local design decisions with non-systemic impact
- Tactical requirements are requirements with local scope such as developmental requirements (e.g., modifiability)
Ad-hoc versus Systematic Architecture

- My personal addendum:
  - All software systems have an architecture, no matter whether they were built in a systematic way or just implemented in a language without any design efforts
  - I consider software architecture as both the organization of a system as well as the systematic process to create this organization
  - A software architecture is only available when there is a process and a structure

Essential Software Architecture

Agenda

- Motivation
- What is Software Architecture
- Role of a Software Architect
- Summary
There are many definitions for Software Architecture... but what a software architect is appears to be undefined:

- Some companies maintain a list of (leveled) questions a potential software architect must be able to answer.
- Other companies have the title, but no job profile.
- Yet other companies have a job profile, but that is insufficient and matches more the position of a key developer. (BTW: the RUP does the same mistake)

One possible job description for a software architect:

A software architect is responsible for the definition, realization, and evolution of a software system—during its entire lifetime:

- Understanding the business case for the system under development.
- Understanding and clarifying the requirements for the system.
- Designing a concrete software architecture / supervising its design.
- Communicating the architecture to its stakeholders.
- Selecting the technologies / tools for realizing the software architecture
- Planning, supervising, and mentoring the implementation of the software architecture to avoid architectural drift or to react on fundamental problems regarding its implementation.
- Assuring adequate quality of the overall software system particularly in architecture-related areas, for instance, via design and code reviews.
From Requirements to Software Architecture

<table>
<thead>
<tr>
<th>Software Architects face a lot of Responsibilities (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Business and strategy</strong></td>
</tr>
<tr>
<td>Validate technology roadmap</td>
</tr>
<tr>
<td>Evaluate selected technologies</td>
</tr>
<tr>
<td>Implement IPR strategy</td>
</tr>
<tr>
<td>Support make-or-buy decisions</td>
</tr>
<tr>
<td>Support supplier evaluation</td>
</tr>
<tr>
<td><strong>Requirements engineering</strong></td>
</tr>
<tr>
<td>Support definition of system architecture</td>
</tr>
<tr>
<td>Derive software requirements and functional specifications</td>
</tr>
<tr>
<td>Define external software interfaces</td>
</tr>
<tr>
<td>Validate software requirements</td>
</tr>
<tr>
<td>Prove software feasibility</td>
</tr>
<tr>
<td>Support system lifecycle planning</td>
</tr>
<tr>
<td>Identify and create patents</td>
</tr>
<tr>
<td><strong>Design</strong></td>
</tr>
<tr>
<td>Domain Modeling</td>
</tr>
<tr>
<td>Determine software architecture and design</td>
</tr>
<tr>
<td>Define architecture- and design- rationale</td>
</tr>
<tr>
<td>Establish architecture traceability</td>
</tr>
<tr>
<td>Identify and specify interfaces</td>
</tr>
<tr>
<td>Allocate requirements to increments</td>
</tr>
<tr>
<td>Support specification of system architecture</td>
</tr>
<tr>
<td>Establish integration sequence</td>
</tr>
<tr>
<td>Provide detailed planning for tracking of integration</td>
</tr>
<tr>
<td>Support setup of supplier agreements</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software Architects face a lot of Responsibilities (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Implementation</strong></td>
</tr>
<tr>
<td>Review detailed software design</td>
</tr>
<tr>
<td>Support implementation of critical algorithms / components</td>
</tr>
<tr>
<td>Setup software integration plan</td>
</tr>
<tr>
<td>Control software integration</td>
</tr>
<tr>
<td>Update increment requirements</td>
</tr>
<tr>
<td>Support code reviews, resolution of test defects, integration etc.</td>
</tr>
<tr>
<td>Validate deliveries of external suppliers</td>
</tr>
<tr>
<td>Provide technical coaching of the implementation team</td>
</tr>
<tr>
<td><strong>Test and quality</strong></td>
</tr>
<tr>
<td>Monitor system integration</td>
</tr>
<tr>
<td>Drive architecture and interface quality</td>
</tr>
<tr>
<td>Update software deliveries for integration according to defects resolved</td>
</tr>
<tr>
<td>Support integration test, resolution of test defects and system integration test</td>
</tr>
<tr>
<td><strong>Maintenance</strong></td>
</tr>
<tr>
<td>Manage architecture continuously</td>
</tr>
<tr>
<td>Monitor internal software quality and structural changes</td>
</tr>
</tbody>
</table>
Architects require Knowledge of the Problem Domain

- Software architects need sound knowledge of the problem domain
  - what are the main actors, entities in the domain
  - how do they relate and interact with each other
  - what are the typical user stories
- Otherwise architects won’t be able to support and assess the business case
  - to understand requirements related to the problem domain
  - to create an appropriate design starting with a domain-object model

Architects require Leadership skills and social skills

- Software Architects need leadership skills:
  - courage to decide even in the absence of complete details
  - getting buy-in from other stakeholders
  - coaching and mentoring developers
  - learning from failure
  - dealing with matrix organizations
- To meet these challenges, they must be able to lead and decide
  - require coaching and mentoring skills
Architects require Project management skills

- Software architecture serves as foundation for project management and thus for project success
- Although, software architects should not act as project managers, they need to participate in
  - project planning, creation of release/version plans and work packages,
  - estimation of required human resources and budgets.
- For this purpose, architects require fundamental knowledge and experience in project management

Architects require Knowledge of the Solution Domain and Development Environment

- Software architects needs sound knowledge of the solution domain, in theory and practice
  - what are the main solution technologies and tools available
  - how do they complement or exclude each other
  - when are how are they applicable
- Otherwise architects won’t be able
  - to meet non-functional requirements
  - to understand technology implications
  - to select technologies
  - to create an implementation effectively and efficiently
Architects require Co-operation with other roles

- Software architects are in the center of communication

Essential Software Architecture

Agenda

- Motivation
- What is Software Architecture
- Role of a Software Architect
- Summary
What we learned

- We can only obtain sustainable software systems with a sound software architecture
- Architecture is about the important things of a software system. It comprises
  - the important subsystems
  - their relations
  - the guiding principles used to create the architecture
- If software architecture is essential, we need to make sure that
  - professional software architects with the required skill level are in our organization
  - sufficient resources are dedicated to architecture design, maintenance and evolution
- However, design by community does not work. Lead architects and small architecture teams are essential.

A departing thought

To err is human, but to really foul things up requires a computer.

[Farmer’s Almanac, 1978]
Part II: Requirements

Dr. Michael Stal

Be prepared

Before anything else, preparation is the key to success.

[Alexander Graham Bell, 1847-1922]
All about Requirements and Use Cases

Agenda

- Motivation and Introduction
- Handling Requirements
- Use Case Engineering
- Summary
The Problem of Requirements Engineering

Architect's Challenge

Forces + Creativity + Communication + Decisions = Requirement (such as scalability) + Constraints (such as prescribed components)
Requirements Engineering and Software Architecture

- The software architect is not directly involved in requirements engineering
- However, she/he must co-operate
  - with the requirements engineer to clarify and understand requirements as well as to uniquely prioritize them
  - with the product manager to clarify and understand user requirements
  - with the test manager to establish a test strategy and test plans
  - with the developer to ensure development takes the right direction
- Different roles may have different perspectives and forces

It is not only about technology: Other Influential Forces

- Architects do not only face challenging requirements (product forces) but also other factors that limit their freedom such as
  - **Business objectives**: What goals are important for the customer and for the development organization? Issues like time-to-market, development costs, required staffing, offshoring necessities, …
  - **Organizational and process constraints**: How is the development organization structured? What process must be applied? Issues like team distribution, political factors, competences, make-or-buy, budgets. *Organization often drives architecture!*
  - **Technology constraints**: predefined 3rd party integration, legacy code, operating system requirements, tools, middleware, available hardware, standards, existing architectures.
Architects explicitly address Risks

- All forces lead to risks
  - forces may contradict each other
  - forces may inherently exhibit risks. Example: if organization drives architecture and organization is brittle, then decision making might be challenging (which leads to eroded architectures)
- Architects must explicitly address risks
- Possible approach/template for risk analysis according to Hofmeister:
  - Description of risk (e.g., dependence on persistence layer)
  - Influential factors that lead to this risk (requirement to decouple business from persistence layer, not enough technology skills in team)
  - Solution approach (introduce data access layer)
  - Possible strategies (give subproject to external company, use open source solution, use platform-specific solution)
  - Related topics and strategies (decoupling business logic from other EIS layers)

All about Requirements and Use Cases

Agenda

- Motivation and Introduction
- Handling Requirements
  - Use Case Engineering
- Summary
Classifications

- Let me introduce my personal classification categories
  - **Functional** (the WHAT - related to problem domain)
  - **Non-Functional** (the HOW - related to solution domain/infrastructure domain)
    - **Operational** (Related to service operation)
      - Quality of Service (e.g., security, reliability, ...)
      - Infrastructure (distribution, concurrency, ...)
    - **Developmental** (Quality of Architecture) such as flexibility, modularization and subsystem/team distribution, maintainability

- **Note 1**: Requirements may define properties of the solution or the problem domain
- **Note 2**: Embedded requirements such as real-time constraints, memory restrictions, battery constraints, become high-priority operational requirements
- **Note 3**: Any non-functional requirement does only make sense in the context of the functional software architecture

**Desired Properties of Good Requirements**

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cohesive</td>
<td>The requirement addresses one and only one thing.</td>
</tr>
<tr>
<td>Complete</td>
<td>The requirement is fully stated in one place with no missing information.</td>
</tr>
<tr>
<td>Consistent</td>
<td>The requirement does not contradict any other requirement and is fully consistent with all authoritative external documentation.</td>
</tr>
<tr>
<td>Correct</td>
<td>The requirement meets all or part of a business need as authoritatively stated by stakeholders.</td>
</tr>
<tr>
<td>Current</td>
<td>The requirement has not been made obsolete by the passage of time.</td>
</tr>
<tr>
<td>Externally Observable</td>
<td>The requirement specifies a characteristic of the product that is externally observable or experienced by the user. &quot;Requirements&quot; that specify internal architecture, design, implementation, or testing decisions are properly constraints, and should be clearly articulated in the Constraints section of the Requirements document.</td>
</tr>
</tbody>
</table>
Desired Properties of Good Requirements (cont'd)

<table>
<thead>
<tr>
<th>Desired Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feasible</td>
<td>The requirement can be implemented within the constraints of the project.</td>
</tr>
<tr>
<td>Unambiguous</td>
<td>The requirement is concisely stated without recourse to technical jargon, acronyms (unless defined elsewhere in the Requirements document), or other esoteric verbiage. It expresses objective facts, not subjective opinions. It is subject to one and only one interpretation. Vague subjects, adjectives, prepositions, verbs and subjective phrases are avoided. Negative statements and compound statements are prohibited.</td>
</tr>
<tr>
<td>Mandatory</td>
<td>The requirement represents a stakeholder-defined characteristic the absence of which will result in a deficiency that cannot be ameliorated.</td>
</tr>
<tr>
<td>Verifiable</td>
<td>The implementation of the requirement can be determined through one of four possible methods: inspection, analysis, demonstration, or test.</td>
</tr>
</tbody>
</table>

Non-Functional Requirements (aka Quality Attributes): Utility Trees

- Utility Trees help evaluating the qualities:

```
Utility
<table>
<thead>
<tr>
<th>Performance</th>
<th>Modifiability</th>
<th>Availability</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>Throughput</td>
<td>HW failure</td>
<td>Integrity</td>
</tr>
<tr>
<td>Transaction</td>
<td>New products</td>
<td>COTS SW</td>
<td>Data confidentiality</td>
</tr>
<tr>
<td></td>
<td>Change</td>
<td>failures</td>
<td>Data integrity</td>
</tr>
</tbody>
</table>
```

- (L,M) Reduce storage latency on customer DB to ≤ 200 ms.
- (M,H) Deliver video in real time.
- (H,H) Add CORBA middleware in < 20 person-months.
- (H,L) Change web user interface in < 4 person-years.
- (H,H) Power outage at site X requires traffic redirected to site Y in < 3 seconds.
- (H,H) Network failure detected and reconnected in < 1.5 minutes.
- (H,M) Credit card transactions are secure 99.999% of the time.
- (H,L) Customer DB authorization works 99.999% of the time.
All about Scenarios

- According to ATAM scenarios describe all interactions with a system
  - How does the system react on stimuli
  - How are qualities implemented and executed
- Different variants
  - Use Cases/Scenarios: a user interacts with the system
  - Change Cases/Scenarios: developer or operator changes part of the system or its environment
  - Boundary or Load Cases/Scenarios: system is confronted with unanticipated situations
- This implies: we need to express qualities with scenarios

Scenarios by Example: Availability

<table>
<thead>
<tr>
<th>Source:</th>
<th>Artifact:</th>
<th>Environment:</th>
<th>Response:</th>
<th>Response Measure:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Internal</td>
<td>- Process</td>
<td>- Normal</td>
<td>- Record</td>
<td>- Repair</td>
</tr>
<tr>
<td>- External</td>
<td>- Storage</td>
<td>- Degraded</td>
<td>- Notify</td>
<td>- Time</td>
</tr>
<tr>
<td></td>
<td>- Processor</td>
<td>- Operation</td>
<td>- Disable</td>
<td>- Availability</td>
</tr>
<tr>
<td></td>
<td>- Communication</td>
<td></td>
<td>- Continue</td>
<td>- Available/</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Degraded</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Time Interval</td>
</tr>
</tbody>
</table>

- Stimulus:
  - Fault
  - Omission
  - Crash
  - Timing
  - Response

- Environment:
  - Normal
  - Degraded
  - Operation

- Response:
  - Record
  - Notify
  - Disable
  - Continue
  - (Normal/ Degraded)
  - Be Unavailable

- Response Measure:
  - Repair
  - Time
  - Availability
  - Available/ Degraded
  - Time Interval
Structure of Scenario

- Mandatory
  - **Stimulus**: event that is caused by a stakeholder on the system.
    - Examples: User presses button on UI, Operator changes configuration, DDoS attack
  - **Environment**: state of the system when the stimulus is initiated
    - Examples: waiting for user input, normal operation, waiting for connection requests
  - **Response**: how does the system react
    - Is the requested functionality actually provided?
    - How long does it take until the configuration change is available?

Structure of Scenario (cont’d)

- Extensions:
  - **Source of stimulus**: who or what was the originator
    - Examples: user, other external machine, external connection request
  - **Artifact**: which part of the system is impacted by the stimulus
    - Examples: the system as a whole, the application server
  - **Response Measure**: how can we measure the response
    - Examples: response times, system consistency
Introducing Solution Strategies and Tactics with Design Tactics Diagrams

- Mappings between quality attributes and tactics help to structure the process
- Example Availability:

<table>
<thead>
<tr>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault detection</td>
</tr>
<tr>
<td>Recovery – Preparation And Repair</td>
</tr>
<tr>
<td>Recovery – Reintroduction</td>
</tr>
<tr>
<td>Prevention</td>
</tr>
<tr>
<td>Ping/Echo</td>
</tr>
<tr>
<td>Heartbeat</td>
</tr>
<tr>
<td>Exception</td>
</tr>
<tr>
<td>Voting</td>
</tr>
<tr>
<td>Active Redundancy</td>
</tr>
<tr>
<td>Passive Redundancy</td>
</tr>
<tr>
<td>Spare</td>
</tr>
<tr>
<td>Fault Masked</td>
</tr>
<tr>
<td>Or</td>
</tr>
<tr>
<td>Repair</td>
</tr>
<tr>
<td>Made</td>
</tr>
</tbody>
</table>

Tactics: Modifiability

<table>
<thead>
<tr>
<th>Modifiability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stimulus: Change arrives</td>
</tr>
<tr>
<td>Localize Changes</td>
</tr>
<tr>
<td>Prevention of Ripple Effect</td>
</tr>
<tr>
<td>Delay Binding Time</td>
</tr>
<tr>
<td>Semantic coherence</td>
</tr>
<tr>
<td>Anticipate expected changes</td>
</tr>
<tr>
<td>Generalize module</td>
</tr>
<tr>
<td>Limit possible options</td>
</tr>
<tr>
<td>Abstract common services</td>
</tr>
<tr>
<td>Hide information</td>
</tr>
<tr>
<td>Maintain existing interface</td>
</tr>
<tr>
<td>Restrict communication paths</td>
</tr>
<tr>
<td>Use an intermediary</td>
</tr>
<tr>
<td>Response: Changes made, tested, and deployed within time and budget</td>
</tr>
<tr>
<td>Runtime registration Configuration files Polymorphism Component replacement Adherence to defined protocols</td>
</tr>
</tbody>
</table>
Tactics: Performance

Stimulus: Events arrive

Response: Result generated within time constraints

Performance

Resource Demand

Resource Management

Resource arbitration

Increase computation Efficiency

Reduce computational overhead

Manage event rate

Control freq. of sampling

Introduce concurrency

Maintain multiple copies

Increase available resources

Scheduling policy

Tactics

Stimulus: Attack

Response: System detects, resists, recovers from attacks

Security

Resisting Attacks

Detecting Attacks

Recovering From attack

Authentication

Authorization

Maintain data confidentiality

Maintain integrity

Limit exposure

Limit access

Restoration

Identification

Audit Trail

Tactics

Stimulus: Events arrive

Response: Result generated within time constraints

Performance

Resource Demand

Resource Management

Resource arbitration

Increase computation Efficiency

Reduce computational overhead

Manage event rate

Control freq. of sampling

Introduce concurrency

Maintain multiple copies

Increase available resources

Scheduling policy

Tactics

Stimulus: Attack

Response: System detects, resists, recovers from attacks

Security

Resisting Attacks

Detecting Attacks

Recovering From attack

Authentication

Authorization

Maintain data confidentiality

Maintain integrity

Limit exposure

Limit access

Restoration

Identification

Audit Trail

Tactics
Tactics: Testability

- **Stimulus:** Completion of an increment
- **Manage Input/Output**
- **Internal Monitoring**
- **Record/playback**
- **Separate interface from implementation**
- **Specialized access routines/interfaces**
- **Built-in Monitors**
- **Response:** Faults detected

Provide Checklists

- At least for the high priority qualities checklists should be provided by experts
- In a concrete project key developers (sub-teams) should be identified
  - who are in charge of introducing high priority/risk quality concerns to the overall architecture
  - who provide tactics and checklists for these concerns
- Note: the following slides reveal the principle approach but are insufficient
All about Requirements and Use Cases

Agenda

- Motivation and Introduction
- Handling Requirements
- Use Case Engineering
- Summary

Use Case Engineering

- According to Wikipedia: In software engineering and system engineering, a use case is a technique for capturing functional requirements of systems and systems-of-systems ...
- Use Case Engineering often underestimated its value or applied inappropriately
- It is not about drawing an UML Use Case Diagram with actors, ovals and lines
Example Use Case

- "Receiving a question from you in this tutorial"
  - GOAL OF PRIMARY ACTOR
- (User Goal, Participant)
  - PRIMARY ACTOR
- Main scenario
  1. You raise your hand
  2. I am asking you for your question
  3. You are asking
  4. I am thinking about the question and giving an answer
  - LEVEL OF GOAL [USER, SUMMARY, SUBFUNCTION]
  - 4 STEPS IN FULL SENTENCES
  - SHOWING WHO TAKES THE ACTION
- Extensions
  - 4a Too many details to answer directly: I am asking you to contact me in the break
  - ... CONDITION CAUSING DIFFERENT ACTION

General Format

- Name
- Actors
- Trigger
- Preconditions
- Post conditions
- Success Scenario
- Alternatives flows

Alistair Cockburn
"Writing Effective Use Cases"
**Actors**

- An **Actor** is an external object (human or machine) that
  - either triggers activities of the system (that's the use cases)
  - or is being involved in such activities
- Examples:
  - Customers browsing the product catalog of a web shop
  - Order processing application called by the Web shop system to process a customer's purchase order
- Actors can have inheritance relationships: a „premium customer“ is a sub type of „customer“

**Triggers**

- A trigger denotes an event that starts a use case
- Examples:
  - Customer presses button to add item to shopping cart
  - Security system issues an alarm
  - Phone rings
  - Timeout
- The trigger is always caused by an actor
Preconditions

- Conditions that need to be true so that the system can execute the use case
- Examples:
  - User account exists
  - Sufficient items available
  - Network connection available

Post Conditions

- A post condition represents the outcome of a use case
- Examples
  - Connection with remote machine established
  - Money withdrawn from customer account
  - Meal order accepted and processed by clerk
- Minimal guarantee
  - The minimal things a system can promise, holding even when the use case execution ended in failure
  - Examples: Money is not transferred unless authorization is granted by the user
- Success guarantee
  - What happens after a successful conclusion of the use-case.
  - Examples: The file is saved; money is transferred
Success Scenario (aka Happy Day Scenario)

- The main story-line of the use case
- This is expressed with the assumption that no errors or exceptions occur and that it directly leads to the desired outcome of the use case (post-condition)
- Composed of a sequence of action steps each of them telling what the actor is doing in each step:
  1. Customer enters login data
  2. Systems checks identity and provides customer access to order history
  3. Customer selects one of the orders
  4. System displays all information on order

Alternative Flows

- Alternative flows describe exceptional behavior deviating from or breaking the main flow
- Examples:
  - Errors & Failures ("Connection to credit card company failed")
  - Unusual cases ("credit card is recognized as invalid")
  - Short Cuts ("user can interrupt process by clicking escape-key")
What a use case is and what it isn’t

- According to Alistair Cockburn good use cases are
  - text based descriptions but not UML use case diagrams
  - not describing GUIs
  - not describing data formats
  - containing 3-9 steps in their main scenarios
  - easy to read
  - easy to understand from user level
  - at user’s goal level, not program feature level
  - recordings of decisions made instead of tutorials on the domain
- They can be written all upfront or just-in-time
- They can be written to full completion or in incremental steps

What Use Cases are (not) for

- Use cases are a means to document functional requirements in a goal and user-driven, easy-to-read style
- They serve as a base for design, but can only cover certain aspects
- Use cases are not about non-functional requirements
- They don’t specify all the details such as numbers, cardinality, formula, state, performance, ...
- They are helpful for prioritization and project scheduling
All about Requirements and Use Cases

Agenda

- Motivation and Introduction
- Handling Requirements
- Use Case Engineering
- Summary

What we learned

- Requirements come in different flavors, e.g., functional and non-functional requirements.
- They may describe or constrain the problem or the solution domain.
- Requirements must always be prioritized (n requirements get n priorities) to solve conflicts by precedence rules.
- Architects need to base all their decisions on requirements which is why requirements traceability is a first order goal.
- Architects co-operate with requirements engineers to understand and clarify requirements and with test managers to plan for testing.
- Use Case engineering is an appropriate approach to handle user requirements.
A departing thought

You cannot escape the responsibility of tomorrow by evading it today.

[Abraham Lincoln, 1809-1865]
Creating the Software Architecture

Agenda

- Motivation and scope
- Approach from 10000 feet
- Architect’s Toolbox
- Summary and conclusion
Creating the Software Architecture

Agenda

- Motivation and scope
- Approach from 10000 feet
- Architect's Toolbox
- Summary and conclusion

How to create a Software Architecture

- Software Architecture design maps the problem domain to the solution domain considering forces and risks
- Some observations and experiences
  - The best designs were not invented by committees but by (a small number of) individuals [F. Brooks]
  - To take the right directions architects need a broad set of skills (such as domain, technology, communication, experience)
  - Most projects experience a volatile background where change is the rule not the exception
Implications for Architecture Design

- Establish a small team of architects with one architecture owner
- Take care that the architects have the right level of skills and expertise
- Mind that architecting is about deciding & communicating and requires courage
- Each design decision must explicitly address a risk or requirement
- Embrace change and don’t use a BDUF approach (Big Design Up Front)

Top-Down versus Bottom-Up Design?

- To master inherent complexity architecture should mainly happen in a top-down direction using problem partitioning styles such as divide-et-impera
- But each project has to deal with constraints and only few projects are green field projects
- However, a complementary bottom-up approach is necessary to consider forces such as
  - Integration of legacy components, frameworks or services into the system
  - Software integration such as providing interfaces to other applications
  - System integration (e.g., into embedded and real-time environments)
  - Required usage of specific tools or technologies
  - Quality of service constraints
Combining Top-Down and Bottom-Up Design

- Combine both approaches:
  - Identify bottom-up requirements first
  - Follow top-down-approach but explicitly consider all bottom-up-constraints in architectural decisions
- This works as most bottom-up problems typically appear in tactical design, not in strategic design

Agile Architecture

- Precondition: all requirements, risks and other forces have been identified and clarified; an appropriate engineering process is in place
- A software architect needs to keep the balance between
  - the need to create a stable architectural base
  - the need to cope with change
- Can we balance these forces?
- Steps:
  - Create design principles and best practices the resulting architecture must follow. This helps achieving conceptual integrity
  - Establish a stable base line architecture comprising all core components and their relations
  - Step-wise refine, evaluate and refactor this architecture using an agile approach until the final „executable“ architecture is available
Major Steps

- Establish domain model for problem space
- Map requirements to architecture-relevant requirements
- Establish a stable base line architecture comprising all core components and their relations
- Step-wise refine, implement, evaluate and improve this architecture using an agile approach
- Create design principles and best practices the resulting architecture must follow. This helps achieving conceptual integrity

The Onion Model

- Architecture design follows an onion model (tears included)
  - Functional Core
  - Distribution & Concurrency
  - Infrastructure
  - Strategic Qualities
    - Prio high … low
  - Tactical Qualities
    - Prio high … low
The Pyramid Model

How deep do we need to go?

The baseline architecture must be as complete as necessary to govern the subsequent software development, but it must also be as simple as possible, otherwise it cannot be communicated

- Three levels of detail to limit depth
- A focus on architecturally significant requirements and corresponding architecture views to limit breadth

How does it work in practice

- Suppose, you are going to design the functional architecture of a web shop
- Here is the list of identified use cases sorted by priorities, risks:
  - Search product catalog
  - Add item to shopping cart
  - Order shopping cart
  - ...
- In a waterfall approach you would try to come up with a complete design at once which, as we know, never works
- In an incremental, agile approach, however, you may:
  - start with the first use case (happy day scenario first and different alternative flows next),
  - then integrate the second use case, maybe after reviewing the current design and preparing it for further refinement,
  - and go on ... until you have covered all scenarios
- The architects then continue with
  - infrastructural aspects and
  - non-functional scenarios (e.g., using the scenario-based approach by Bass, Clements, Kazman)
Creating the Software Architecture

Agenda

- Motivation and scope
- Approach from 10000 feet
- Architect's Toolbox
- Summary and conclusion

Embrace Change

- Preconditions:
  - The most important forces have been sufficiently identified and clarified for the next iteration;
  - An appropriate software development process is in place.

- A software architect needs to keep the balance between
  - the need to create a stable architectural base
  - the need to cope with change

- How can we balance these forces?
An Architecture Process from 10000 feet

Preparation

Create Architecture Baseline

Evaluate & Refine Architecture

Refactor Architecture

Feedback Loop

no

Complete?
yes

Executable Architecture

Perform User Requirements Elicitation

Goals:
- Identify user-requirements via use case analysis

Activities:
- Collect main (also known as happy day) scenarios from user perspective
- Integrate alternative flows (rainy day scenarios)
- Clarify priorities of use cases

Results:
- Documented set of use cases with priorities (main and alternative flows)

Name
Actors
Trigger
Preconditions
Post conditions
Success Scenario
Alternatives flows
Domain Models

- Goals:
  - Get knowledge about problem domain and proven solutions
- Activities:
  - Draw a domain model of the domain you are going to address
  - Try to find whether a reference architecture or existing software systems already exist that can serve as a base
  - Introduce guidelines and principles all engineers must abide to (such as patterns to use, fault management guidelines, logging guidelines, ...)

Reference Architectures and Architecture Blueprints

- If your problem domain is clearly identified, existing solutions may serve as a base. You don’t need to start from scratch
- Think about the following problem domains:
  - Compilers: What are the domain objects in a compiler?
  - GUIs: What does a GUI Framework typically provide?
  - Operating systems: What are constituents of an operating system?
- A Reference Architecture provides a proven template solution for an architecture for a particular domain. It also provides a common vocabulary with which to discuss implementations, often with the aim to stress commonality
- If a reference architecture already exists, use it as a base
- For platforms and infrastructures (the solution domain) architectural blueprints give guidance how to best leverage the underlying platform.
- Use architectural blueprints such as Java EE Patterns or Enterprise .NET Patterns when designing your software system
Boundaries

- **Goals:**
  - From user perspective show interactions of actor with system
  - Define boundaries of system (find out what is part of the system and what isn’t)
- **Precondition:** All requirements and all other factors have been identified and clarified, all risks analysed
- **Activities:**
  - Apply a use case analysis to determine all necessary scenarios
  - Draw a context diagram to define the boundaries between the software system and its environment

Dynamics

- **Goals:** Understand which interactions are triggered by use-case scenarios
- **Activities:**
  - Introduce sequence or activity diagrams that illustrate the typical scenarios from a domain viewpoint (i.e., without considering infrastructures or qualities). Focus on happy day scenarios first
First Conceptual Draft

- Goals:
  - Get a first concept of a possible solution
- Activities:
  - Use the results you got so far and add infrastructural/strategic issues to design a rough design sketch of your system (conceptual level)

Base Line Architecture - Skeleton

- Goals:
  - Define a stable architecture base for later refinement
- Preconditions:
  - The list of functional and non-functional requirements is available, e.g., use cases and a utility tree
- Activities:
  - Use architectural patterns (styles) or reference architecture if available and include infrastructural considerations to embed functional design into infrastructure
Example: Layers

Solution Structure: **Decouple the abstraction levels of an application by splitting it into several loosely connected layers—one layer for each abstraction level.**

- **Services** realize the application’s functionality.
- **Layers** comprise all services of a particular abstraction level.

Base Line Architecture - Increments

- **Goals:**
  - Get a refined base line architecture
- **Activities:**
  - Strategic Design: Incrementally take operational scenarios and functional use cases (end to end) to refine architecture
  - Tactical Design: At the end also consider strategic modifiability (variability) requirements
Creating the Software Architecture

Agenda

- Motivation and scope
- Approach from 10000 feet
- Architect's Toolbox
- Summary and conclusion

An Architect's Toolbox

- Create an Architecture Vision
- Refine and Refactor Incrementally
- Time-box all Increments
- Develop Test-Driven
- Do Strategic Design before Tactical Design
- Enforce the Architecture Vision
Why Process Principles

Process Principles help to:

- specify and implement a software architecture constructively and timely.
- check and ensure the architecture’s quality.
- respond to changes of all kind.
- deal with problems arising during the architecture’s specification and implementation.
- support an architecture-driven software development.

Different concrete software development processes can implement these principles differently.

Where we are ...

Create An Architecture Vision describes the very first activity in designing a software system: specifying its baseline architecture.

This principle assumes an initial requirements set being available.
Architecture Vision I

Many architecture specifications look like this one ...

This diagram is not wrong!

However:

- How much information does it contain?
- Can developers tell you the core structures and interactions within the system?
- Do you trust those who specified this architecture that they know what really is to be built?

Architecture Vision II

Therefore:

Create an architecture vision: a base-line architecture that specifies the core functional and infrastructural subsystems and components of the system to be built, their main responsibilities, relationships, and collaborations, as well as all design principles that this base-line architecture follows.

In addition, tie back each design decision to the system-wide requirements it addresses.
Architecture Vision III

- The base-line architecture must be as complete as necessary to govern the subsequent system development.
- The base-line architecture must be as simple as possible, to be able to communicate it.
- The base-line architecture must consider core functional aspects, such as the system's main responsibilities, major non-technical factors, such as the need to integrate 3rd party or legacy software, and operational aspects, such as distribution and security concerns.
- The base-line architecture should be immune to later changes.

Architecture Vision IV

1. Start with the infrastructure in which application functionality should “live“:
   - Think Distributed and Concurrent
   - Think in Components and Services
   - Design for Operational Quality
   - Design for Developmental Quality
   - Use architectural patterns if useful
2. Identify the core functional subsystems and their collaborations:
   - Use architectural patterns if useful
3. Integrate the functional subsystems into the infrastructure.
4. Refine and Refactor Incrementally until the base-line is stable.
5. Refine and Refactor Incrementally to detail the base-line architecture.
Architectural Patterns

Architectural Patterns help specifying an Architecture Vision by:

- addressing important infrastructural concerns and providing placeholders for application functionality
- providing fundamental domain concepts.

Examples:

- Distribution: Broker, Half Object plus Protocol
- Concurrency: Half-Sync/Half-Async, Leader/Followers, Active Object
- Interaction: Model-View-Controller, Presentation-Abstraction-Control
- Database access: Database Access Layer
- Flexibility: Reflection, Microkernel
- Event Handling: Reactor, Proactor
- General Decomposition: Layers, Pipes and Filters, Blackboard

Architecture Documentation

Templates for documenting software architectures help to:

- organize the architecture’s description appropriately
- ensure that no important concern gets forgotten
- communicate the architecture

Examples:

- Architecture description template provided by the Rational Unified Process (RUP), which is based on the 4 + 1 view.
- Architecture description template suggested by Siemens CT or Siemens SCR. – Hofmeister, Soni, Nord
- A project diary helps documenting the „why“ of an architecture.
Where we are ...

Refine And Refactor Incrementally outlines the key process principle for constructing software systems. This principle fosters a process of piecemeal growth, a mixture of top-down and bottom-up activities, in which a software architecture grows and changes continuously until it is complete and consistent in all its parts.

Refine and Refactor Incrementally

After a while, many architectures tend to look like this one ...

- The original architecture vision is hardly visible.
- Design flaws are scaffold by many small and local "corrections."
- Missing parts are attached via backpacks.

However:

Such an architecture is doomed to fail before it goes into implementation or operation, because it suffers from:

- developmental qualities like flexibility and maintainability.
- operational qualities like performance and scalability.
Refine and Refactor Incrementally II

Therefore:
Create a software architecture step-wise via a number of well-defined, small increments. Each increment includes:

- top-down refinement activities to detail and complete the software architecture.
- bottom-up refactoring activities to garden and clean-up inconsistent or insufficient design decisions.

The process stops if the software architecture is complete and consistent in all its parts and details.

Refine and Refactor Incrementally III

1. Time-box all increments and plan their number and content.
2. Realize increments:
   - Develop Test-Driven.
   - Do Strategic Design before Tactical Design:
     - Think Distributed and Concurrent.
     - Think in Components and Services.
     - Design for Developmental Quality
   - Enforce the Architecture Vision.
   - Refactor all wrong, inappropriate, or insufficient design decisions.
   - Optimize for Quality of Service
Where we are …

Time-box all Increments emphasizes that time—in particular meeting deadlines—has a higher priority than the number of delivered features.

Time-box all Increments

Where we are …

Most software development projects have a defined project plan:

- A set of features to be delivered.
- A development schedule with defined milestones.
- Budget and cost estimations.
- …

However:

Early and continuous feedback is missing in most of these projects. Consequently:

- Only few software systems deliver on time or within budget. Milestones get stretched time and again.
- Many software systems suffer from the required operational and developmental qualities.

Plan

2003

Reality

2003

2004

2005
Time-box all Increments II

Therefore:
Give time a higher priority than functionality!
- Instead of associating milestones with the delivery of certain sets of features or functionality, partition the entire “run-time” of the project into a number of time-boxed increments to check progress early and continuously.
- Whenever you observe quality, schedule, or budget slips at the end of an increment, take appropriate actions immediately, such as refactoring, feature restrictions, or re-budgeting.

Time-box all Increments III

1. Prioritize your requirements / features to be delivered. If there are \( N \) requirements / features, there are \( N \) priorities!
2. Estimate the time needed to meet / deliver each requirement / feature.
3. Select time-box length. 1 – 2 weeks are appropriate for small projects, 4 – 6 weeks for large projects. 8 weeks is too long to provide early feedback, even for the largest project.
4. Determine the number of increments needed for the project \( \text{Development time} / \text{time-box length} \).
5. Assign the requirements / features to increments according to their priority and delivery time.
Where we are …

Develop Test-Driven helps to assure the operational and developmental quality of the software system under development.

Following this principle nothing is specified or implemented without having a corresponding set of test cases available to evaluate the quality of what was done.

Develop Test-Driven I

Systems time and again suffer from appropriate quality ...

- Some features are missing, others are not needed at all, and yet others are simply wrong.
- The operational quality, such as performance, stability, etc., suffers even though testing plays an important role within every development process.

However:
Defective systems will never be accepted by a customer, and ultimately not sold. Even in case of quality improvements, delivery delays because of quality deficiencies won't impress your customers.
Develop Test-Driven II

Therefore:
Never design or implemented a piece of software without having an appropriate set of test cases available to evaluate the quality of what was done.

- Run all tests frequently, whenever a new piece of code got added or changed to get immediate feedback on what works and what does not work in your system – and how it works.
- If a test fails, write more tests to isolate the problem, if all tests pass, refactor and continue with the development by starting the TDD-cycle again.

Develop Test-Driven III

1. **Specify and write a test (first):** For every new piece of design or code, specify what functionality it should provide, what interface it should offer, and how it should behave in detail, for instance regarding its operational quality. Then specify and write the corresponding set of test that check these properties.

2. **Make the tests run:** Passing the tests rules and governs all subsequent design and implementation activities. Provide a solution for the subject under work that fulfills all associated tests—regardless of the quality of that solution. Quick but dirty is allowed—if the test bar is green.

3. **Make your design and code right:** Once all tests run successfully, refactor your sinful solution such that it obeys your project’s rules for a clean design or code.
Where we are …

Do Strategic Design Before Tactical Design helps to focus your activities on the “right” concerns.

In particular, this principle emphasizes that it is important to define an application’s core abstractions and the workflow among these abstractions first, before addressing details of their implementation, for example how to vary algorithms that these core abstractions use.

Do Strategic Before Tactical Design I

Many detailed designs look like this one ...

- Components have no clearly defined responsibilities.
- Everything is extremely ‘flexible’ and ‘configurable.’
- The specification of central work /control-flows is deferred to a later point in time, ‘to not constrain the system too early’ or ‘when requirements become clear.’

However:

Such designs rarely work! Instead they suffer from diseases, such as the ‘strategy syndrome’!
Do Strategic Before Tactical Design II

Alternatively, many detailed designs look like this one...

- Most components have no clearly defined responsibilities.
- There is a single ‘God’-component that knows and does almost everything.

However:

Such designs rarely work! Instead they suffer from bad performance, and they are hard to maintain and extend!

Do Strategic Before Tactical Design III

Therefore:

At every level of abstraction, consider strategic design aspects first, such as the system’s infrastructure or its central domain entities and workflows, so that you know how the system works!

Then address the mandatory (!) tactical design aspects to shield the strategic design from uncontrolled changes, such as the extensibility of a given workflow.

If there are multiple strategic or tactical aspects to consider at a given level of abstraction, address them in the order of their importance for this particular abstraction level.
Do Strategic Before Tactical Design IV

- At the highest-level of abstraction, there are only strategic aspects: system-wide infrastructural and operational quality decisions.
- When decomposing subsystems, functional aspects are of highest strategic importance followed by subsystem-internal infrastructural aspects like component distribution and concurrency strategies and operational quality aspects.
- Tactical design issues mainly deal with addressing developmental requirements, such as changeability, extensibility, and adaptability.

Use a commonality/variability analysis to identify strategic and mandatory tactical aspects.

Design Patterns

Design Patterns and most Analysis Patterns Help Specifying a strategic design by:

- Providing general purpose approaches to subsystem/component decomposition and interaction.
- Providing domain-specific concepts and workflows

Examples:
- All Architectural Patterns.
- GoF Structural and Behavioral patterns: Composite, Flyweight, Chain of Responsibility, Interpreter, Iterator, Mediator, Observer, State.
- Analysis patterns: Party, Accountability, Observation, Measurement, Fresh Work before Stale, (Inverse) Leaky Bucket Counter, …
Where we are …

Enforce The Architecture Vision ensures that a software architecture is specified and implemented such that it is complete and consistent in all its parts and details—in accordance to the architecture vision and its fundamental design principles.

Enforce the Architecture Vision I

A Warehouse Management Framework

Knowing about the important technical issues in building software architectures …

… is an important prerequisite to build software systems effectively and successfully.

However:

Technical skills are only one side of the coin. All this knowledge does not help us if we apply it uncontrolled, specifically if we build large software systems. Throwing an architecture specification over the fence to developers and hoping that a working system is thrown back to you has never worked in practice.
Enforce the Architecture Vision II

Therefore:
Enforce the architecture vision by active communication and participation:

- Let a software architect supervise and coordinate activities of design and development groups.
- Besides being lead designer, his/her responsibilities are to detect problems early and to ensure that development groups apply design principles and solutions to recurring problems consistently.
- Create architectures by mentoring and walking around!
- Make sure that you always implement!!! (but never on a critical path)

Creating the Software Architecture

Agenda

- Motivation and scope
- Approach from 10000 feet
- Architect's Toolbox
- Summary and conclusion
The framework in retrospect

The framework of mindset, activities, practices, methods, and technologies for defining and realizing platform, product line, or solution architectures fosters the delivery of **software that provides value to customers** via:

- A strictly feedback- and quality-oriented definition and realization of software architectures
- An executable baseline early on
- A consequent focus on key architecture drivers.
- The definition and realization of sustainable and usable architectures
- Communication and close interaction with other roles in software engineering

A departing thought

© Scott Adams, Inc./Dist. by UFS, Inc.
Part IV: Documenting the Architecture

Dr. Michael Stal

Secret of Design

Design is not just what it looks like and feels like. Design is how it works.

[Steven Jobs, Chairman of Apple]
About Design and Documentation

Agenda

- Motivation
- Views
- Software Architecture Documentation
- Summary
Why do we need documentation?

Architecture is for communication – not for backup!

- Communication with different stakeholders and different roles in the project
- Educating new team members
- (Re)understanding the architecture e.g. for maintenance reasons, integration, change of underlying platform
- Base for architecture assessment

War story: In a project documentation was either missing or difficult to read. Although the documentation did not lack any information, developers failed to read it. As a result, at the end documented and realized architecture differed so much that the system had to be reengineered.

Different stakeholders and their needs

<table>
<thead>
<tr>
<th>Stakeholder</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements Engineer, Customer</td>
<td>- Negotiate requirements and scope</td>
</tr>
<tr>
<td></td>
<td>- Make tradeoffs between competing requirements</td>
</tr>
<tr>
<td>Developers</td>
<td>- Provide constraints to developer activities</td>
</tr>
<tr>
<td>Testers</td>
<td>- Specify correct black-box behavior</td>
</tr>
<tr>
<td>Architects of other systems</td>
<td>- Define interfaces and protocols of inbound and outbound interfaces</td>
</tr>
<tr>
<td>Program Managers</td>
<td>- Forming development teams</td>
</tr>
<tr>
<td>PLM</td>
<td>- Determine whether a new member of the product family is in or out of scope</td>
</tr>
<tr>
<td>QA team</td>
<td>- Basis for design and code reviews</td>
</tr>
</tbody>
</table>
What should be documented? What would I like to know about the architecture?

- Requirements driving the architecture
  - Main functional requirements
  - Non functional requirements describing the quality attributes
- Context – the boundaries of the system
- The architecture itself – structure and behavior in different views of the system, interfaces, protocols
- How the architecture addresses quality attributes, non-functional requirements and crosscutting concerns
- The design rationale behind the architecture

About Design and Documentation

Agenda

- Motivation
- Views
  - Software Architecture Documentation
  - Summary
Views

- Let me first focus on another engineering discipline to give you the idea
- Let's suppose we are going to build a house
- We will introduce different views each of which shows a different aspect
- A view is a model that focuses on some core abstractions seen from a specific angle (i.e., targeting specific stakeholders)
- For instance, floor plans and related diagrams helps to illustrate what the functionality provided will be without taking non-functional aspects into account

Views (cont'd)

- A 3D plan gives an impression of the overall system
- ... maybe even embedded into its environment
- 2D plans show details (interfaces) but cannot depict 3D surfaces (implementations)
- 3D plans and 2D plans focus more or less on a black box perspective
- Helpful for architects and customers
Views (cont’d)

- Plans concerning electricity, water and gas supply, or telecommunications add infrastructural aspects

- Structural engineering introduces non-functional aspects

View Sets

- We could provide a lot of additional views such as
  - static views to illustrate the structures of a system as well as their relationships; dynamic views to show interactions between entities
  - component views if a system consists of different parts (think of prefabricated houses)
  - ...

- Each view models a specific aspect of the system and is useful for a specific set of stakeholders
- Each view also has some limitations as it only can model a part of reality
- It is not important which set of views we leverage, but that the set of views is consistent, complete and expressive for the stakeholders targeted
- The set of views also gives guidance how to provide the documentation
- Not all systems need all views! For instance, a deployment diagram for „hello world“ doesn’t make sense
Some Observations

- First, we need to model the functional domain with its core components in a black-box approach (using 2D/3D/floor plans).
- To plan infrastructural aspects such as water or power supply, we need to provide additional plans. Note that a house plan must also consider its environment (where is the street, where are power and water supplies).
- Non-functional aspects such as construction analysis, material qualities, flexibility need to be considered in additional views depending on their priorities.
- There are some dependencies amongst views such as construction analysis depending on materials used.
- There are also some constraints imposed by regulations or other factors such as “position of power supply lines related to water pipes must be at least n meters” or “house design must follow some legal restrictions”.
- There could be also some general best practices such as number, size of positioning of windows, doors or even rooms, e.g. each floor should have at least one bathroom.

Example 4+1 View by Rational Unified Process

![Diagram of 4+1 View by Rational Unified Process]

- Logical View
- Implementation View
- Use Case View
- Process View
- Deployment View
- Conceptual
- Physical

- End-user Functionality
- Programmers Software management
- System integrators Performance Scalability Throughput
- System engineering System topology Delivery, installation Communication
Additional Views

- There are still other useful views not covered by 4+1
- **Context Views** help to describe the boundaries of the software system. They should serve as a starting point
- **Domain Views** provide a common vocabulary between experts in the problem domain and those in the solution domain

Example: Additional Context View

- We will augment the 4+1 view by context views depicting the boundaries of the system and showing how it is embedded into its functional environment
About Design and Documentation

Agenda

- Motivation
- Views
- Software Architecture Documentation
- Summary

Software Architecture Documents

- Software Architecture Documents help explain the software architecture to the various stakeholders
- Different level of abstractions to meet requirements of stakeholders
- High documentation quality so that the architecture can be evolved and evaluated on a sound basis
- Documents explain not only the „what“ but also the „why“ and the „how“. This implies
  - graphical representations require additional explanations
  - source code alone is not sufficient (in most cases)
- Documentation is an agile process. When the architecture evolves, its documentation can not remain unchanged
  - Architecture documents must meet reality
Anwers Architecture Documents should provide

- Which functional and non-functional requirements does the system meet and how (traceability)
- Which guiding principles were used
- How is the software system structured into constituents
- How do these entities interact with each other
- How is the system embedded into its environment
- What are the boundaries of the system
- What is the design rationale behind the architecture and what are its tradeoffs
- What are the commonalities and variabilities
- Which terminology has been used and how are the different artifacts defined

Documenting a Software Architecture

- First Step: Define the target audience of the documentation and what requirements they have. Documents should not be write-only entities!
- Define a hierarchy of documents and assign responsibilities
  - The topmost document should introduce the general architecture
  - Add documents for general principles and guidelines, for example:
    - patterns to use
    - tools to use
    - configurations to use
  - Special documents for cross-cutting concerns such as exception handling, logging, as well as other important aspects identified by the architects
  - Then introduce documents for subsystems ... and maybe for other building blocks such as components, services, libraries, ....
- Enforce an architecture diary where you keep
  - all minutes
  - meta information such as brainstorming results, alternatives discussed and rationale behind decisions
  - this helps prevent recurring endless and painful discussions
Structure of an Architecture Document

- Document should at least contain:
  - Meta Information such as authors, type of document, version, data, target audience
  - Context Information: What is the purpose of this document? Who should read this? What other documents are related?
  - Management Summary: At most one page overview of content
  - Introduction and Motivation
    - What exactly is the scope of the document (i.e., the problem being solved)?
    - Which forces / requirements had to be taken into account?
    - Which use cases (or parts thereof) does the solution address?
- 4+1+1 Views
  - Use Case View
  - Context View
  - Logical View
  - Process View
  - Development View
  - Deployment View
- Open Issues
- Summary
- References
- Glossary

Documentation Templates

- Various templates available such as the one of Rational Unified Process or arc42
- It is more important to use such a template than what template exactly you are going to use
- But use it consequently throughout the project
- Example: RUP
Add Guidelines for Cross-Cutting Concerns

**Why** should architects come up with **central guidelines** such as patterns to use, exceptions to throw, fault management, logging, resource allocation & eviction?

A typical project example:
- If resource A wasn't available, component A returned an error value, component B threw an exception, Component C caught the exception in an empty catch block
- This unsystematic handling of cross-cutting concerns
  - Increased entropy while decreasing simplicity
  - Dramatically decreased conceptual integrity
  - Made integration very difficult

**Conclusion:** Always establish guidelines all architects and developers must abide to. If possible, enforce policies by tools, otherwise by design reviews

---

10 Documentation Guidelines

1. Mind the requirements of the target audience
2. Introduce a document prototype to guide writers
3. Specify the structure of the documents including chapters, models and diagrams to be used, length restrictions, terminology used, and so forth
4. Apply the principles of least surprise and transparency
5. Always document top-down, from low information density/complexity to high information density/complexity
6. Only document what it is essential in the context of the document (KISS Principle)
7. Refer to other documents where appropriate. However, some replication for the sake of readability is acceptable
8. Let documents be part of configuration management and version control
9. Review documents in terms of their quality
10. You may leave parts of the document intentionally blank if they are not appropriate (e.g., deployment diagram for single address space applications)
About Design and Documentation

Agenda

- Motivation
- Views
- Software Architecture Documentation
- Summary

Summary

- Views help to understand and create a software architecture from different perspectives
  - various view models available with the UML
  - 4+1 views being one of them
- The documentation should
  - describe the what and the why
  - address all relevant stakeholders
  - offer the same quality as software artifacts
  - regular reviews
- Use templates to set a common ground
- And always remember: Someone is going to read the documents
Departing Thought

The best way to become acquainted with a subject is to write a book about it.

[Benjamin Disraeli, 1804-1881]
Designing with economy and elegance

Structural engineering is the science and art of designing and making, with economy and elegance, buildings, bridges, frameworks, and other similar structures so that they can safely resist the forces to which they may be subjected.

[The Institution of Structural Engineers]

Why Quality Properties

Quality Properties help to:

- assess the general quality of an architecture, regardless of its first-order operational and developmental qualities.
- check if a software architecture is balanced and well organized.
- develop elegant and aesthetic designs.

Quality properties denote a final sanity check for a software architecture.

Warning: A quality architecture exposes most, if not all, of the quality properties we present here, but the counter conclusion is plain wrong: a software architecture that exposes the properties is not necessarily good!
**Pattern Density**

Role-based pattern integration is the key to pattern density:

- Patterns are not modular building-blocks that are plugged together. Though such a design can yield a high pattern density, it is not a good design.
- Patterns are best integrated on basis of the roles they introduce. A single entity in a software architecture can thus embody multiple roles from multiple patterns.
- In a good pattern-based design it is difficult to identify the individual patterns, due to their tight integration.

The `AbstractStorage` class has less than 100 lines of code, but plays a role in six patterns:
- Composite
- Chain of Responsibility
- Object Manager
- Visitor
- Iterator
- Strategy

**Symmetry**

Quality architectures are inherently symmetric!

- Symmetry appears in the architecture vision, subsystem design, component design, class design, method design, and implementation.
- Symmetry can be both structural and behavioral. Structural symmetry is often an effect of behavioral symmetry.
- Symmetry is not a matter of arranging diagrams (though it indicates it), but largely of complementing responsibilities, and operations.
- A given symmetry may not be perfect.
Expressiveness I

Quality software architectures are expressive—from reading their design one can easily understand their structural and behavioral essence.

- Components and their relationships have proper names that indicate their responsibility or cooperation, respectively.
- Components have clearly defined responsibilities. Each component does just one thing, but it does this one thing well.
- Component interfaces reflect the roles the component plays for clients.
- Component interfaces define precise contracts.
- Components have only explicit relationships to other components.

Expressiveness II

Inexpressive software architectures are hard to communicate and understand—and thus hard to realize and maintain.

- Unclear responsibilities likely result in an improper separation of concerns: clients must cooperate with multiple components where they better communicate with just one. This increases structural complexity.
- Wholesale interfaces introduce implicit dependencies between components: even if an unused method changes its signature, the client gets affected.
- Clients that use component interfaces which do not define precise contracts either must fix the contractual deficiencies in their code—usually on basis of guessing.
- Implicit or not well-defined relationships between components result in structural complexity and communication overhead.
Expressiveness III

A simple metric to check expressiveness: the telephone test!

- If you can explain a design to a colleague in a short telephone call and the colleague is able to redraw the design just from your explanations and without asking (many) questions, then this design is expressive.

Simplicity

KISS – Keep It Stupid Simple: a fundamental rule to develop quality software architectures!

- Quality software architectures expose surprisingly simple designs. The simplest design that possibly could work is the one to target for. Such a design fulfills the requirements but avoids all unnecessary design overhead.
- Less is more, as there are less design decisions and design elements to understand as well as less points of failure.
- Consequently, „a design is not finished when you cannot add anything more to it, but when you cannot remove anything more from it without violating its core properties“. (Blaise Pascal, 1623 – 1662).
Simplicity II

A simple metric to check simplicity: the telephone test!

- If you can explain a design to a colleague in a short telephone call and the colleague is able to redraw the design just from your explanations and without asking (many) questions, then this design is simple.
- Note that simple is not simplistic—oversimplifying a design likely leads to architectures that do not fulfill their requirements.

Emergent Behavior

Self-organized behavior is the key to high performance and high scalability

- Central control and decision components allow explicit control over a system’s behavior, but are single points of failure as well as performance and scalability bottlenecks.
- High-performance and scalable systems delegate the responsibility for control flow decisions to the system itself wherever feasible. Examples include routing in P2P systems, telecom switching, and logistics systems.
What we learned

- Complexity is an important topic to address in software architecture
- It is about mastering inherent complexity and eliminating accidental complexity
- Architecture Entropy plays an important role in all practices and principles
- Abstraction, Modularization, Encapsulation, Information Hiding help mastering complexity
- Principles such as Open/Close, Liskov Substitution give guidance how to design a good architecture
- Quality Indicators such as simplicity form a value system that helps avoiding potential problems in a software architecture
- All those principles, practices and attributes should be in the toolbox of a professional software architect

A departing thought

A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away

[Antoine de Saint-Exupéry]
Part VI: Architecture Refactoring

Dr. Michael Stal

Panta rhei

There is nothing permanent except change.

[Heraclitus, 535–475 BC]
Refactoring and reengineering

Learning objectives

- Understand about design erosion and how to avoid it
- Learn about the principles of refactoring
- Know about activities and best practices necessary for refactoring
- Understand when to use reengineering instead of refactoring

Refactoring and reengineering

Agenda

- Motivation and foundation
- Refactoring
- Reengineering
- Summary
Refactoring and reengineering

Agenda

- Motivation and foundation
- Refactoring
- Reengineering
- Summary

An Architecture Process from 10000 feet Revisited
Design erosion is the root of all evil

- In the lifecycle of a software system changes are the rule and not the exception
  - New requirements or increments imply modifications or extensions
  - Engineers must adopt their solutions to new technologies
  - Changes in business force changes in IT
  - Bug fixes require patches or local corrections
- Unsystematic approaches ("workarounds") cure the symptom but not the problem
- After applying several workarounds, software systems often suffer from design erosion
- Such systems are doomed to fail as workarounds have a negative impact on operational and developmental properties

Step-by-step architecture design

- Agile development is a key success factor
- The software system is created and evolved in well-defined, small steps (increments), including
  - Top-down refinement activities, but also
  - Bottom-up gardening and clean-up activities (refactoring)
- Again, we need to apply strategic changes before tactical ones and consider priorities
- Such transformations of a system should happen in a systematic way
Refactoring and reengineering

Agenda

- Motivation and foundation
- Refactoring
- Reengineering
- Summary

Refactoring

- "Code refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [Martin Fowler]

- Put more generally: Refactoring is the process of changing a software system or process in such a way that it does not alter the external behavior, yet improves its internal structure

Note: external interfaces remain unchanged!
Types of refactoring

- Refactoring denotes the semantic-invariant transformation of a software artifact
- Most people have read Martin Fowler's excellent book on code refactoring
- Some engineers do also know of Joshua Kerievsky's book on refactoring to patterns
- However, refactoring is not constrained to code. Refactoring may also address architectural design, databases, documents, among other artifacts
- Current literature is mostly focused on code refactoring

Checking correctness

- To check the correctness of refactorings, various options are available
  - **Formal approach**: Prove semantics and correctness of program transformation
  - **Implementation approach**: Leverage unit and regression tests to verify that the resulting implementation still meets the specification
  - **Architecture analysis**: Check the resulting software architecture for its equivalence with the initial architecture (consider requirements) – see also CQM methods and tools
- **Use at least one method** to ensure quality
Refactoring to patterns

- Refactoring to patterns was introduced by Joshua Kerievsky
- General idea: Patterns might give guidance how to refactor on the architectural level
- Replace your proprietary solution with a pattern that solves the same problem
- Introduce symmetry by making sure the same problem is always solved using the same pattern / solution
- This represents a precursor of software architecture refactoring

Kerievsky's book focuses on design patterns
- However, we could also apply the same principle to architecture patterns
- In the latter case, we'll obtain architecture refactorings

Example: Replace hard-coded notifications with Observer [Joshua Kerievsky]

Do it yourself

Observer Pattern

Subject contains hardwired list of interested components
Architecture refactoring

What is architecture refactoring?

- Architecture refactoring is about the semantic-preserving transformation of a software design
- It changes structure but not behavior
- It applies to architecture-relevant design artifacts such as UML diagrams, models, DSL expressions, aspects
- Its goal is to improve quality. You got a "smell"? Use an architecture refactoring pattern to solve it!

Architecture smells

- Duplicate design artifacts
- Unclear roles of entities
- Inexpressive or complex architecture
- Everything centralized
- Home-grown solutions instead of best practices
- Over-generic design
- Asymmetric structure or behavior
- Dependency cycles
- Design violations (such as relaxed instead of strict layering)
- Inadequate partitioning of functionality
- Unnecessary dependencies
- Missing orthogonality

Application of architecture refactorings

- The usage of a concrete application refactoring should never happen in an ad-hoc, unsystematic way. Instead, the architect's role is to
- Check the applicability of refactorings
  - Would the refactoring affect parts that need to remain unchanged, such as integration of third-party APIs?
  - Could the refactoring impact requirements in a negative way?
- Define the order of refactoring
  - Strategic before tactical aspects
  - Requirement priorities drive order!
- Apply the refactorings
- Ensure the quality of the software architecture after the refactoring (in conjunction with the test manager)
Where to obtain architecture refactorings?

Send me a note to obtain a catalog

1. Rename Entities
2. Remove Duplicates
3. Introduce Abstraction Hierarchies
4. Remove Unnecessary Abstractions
5. Substitute Mediation with Adaptation
6. Break Dependency Cycles
7. Inject Dependencies
8. Insert Transparency Layer
9. Reduce Dependencies with Facades
10. Merge Subsystems
11. Split Subsystems
12. Enforce Strict Layering
13. Move Entities
14. Add Strategies
15. Enforce Symmetry
16. Extract Interface
17. Enforce Contract
18. Provide Extension Interfaces
19. Substitute Inheritance with Delegation
20. Provide Interoperability Layers
21. Aspectify
22. Integrate DSLs
23. Add Uniform Support to Runtime Aspects
24. Add Configuration Subsystem
25. Introduce the Open/Close Principle
26. Optimize with Caching
27. Replace Singleton
28. Separate Synchronous and Asynchronous Processing
29. Replace Remote Methods with Messages
30. Add Object Manager
31. Change Unidirectional Association to Bidirectional

Example: Remove unnecessary abstractions (1)

A true story: In this example architects introduced Transport Way as an additional abstraction. But can't we consider transport ways as just as another kind of storage? As a consequence the unnecessary abstraction was removed, leading to a simpler and cleaner design.
Example: Remove unnecessary abstractions (2)

- Context
  - Eliminating unnecessary design abstractions
- Problem
  - Minimalism is an important goal of software architecture, because minimalism increases simplicity and expressiveness
  - If the software architecture comprises abstractions that could also be considered abstractions derived from other abstractions, then it is better to remove these abstractions
- General solution idea
  - Determine whether abstractions / design artifacts exist that could also be derived from other abstractions
  - If this is the case, remove superfluous abstractions and derive dependent from other existing abstractions
- Caveat
  - Don't generalize too much (such as introducing one single hierarchy level: "All classes are directly derived from Object")

Example - Break dependency cycles (1)

In this example, the monitor invokes the state getter / setter methods but also provides GetDate() to the sensor, leading to a simple dependency cycle. Providing this method to monitors was a bad design decision, anyway. Introducing a separate date object solves the problem.
Example - Break dependency cycles (2)

- **Context**
  - Dependencies between subsystems

- **Problem**
  - Your system reveals at least one dependency cycle between subsystems
  - Subsystem A may either depend directly or indirectly on subsystem B (e.g., A depends on C which depends on B) which is why we always need to consider the transitive hull
  - Dependency cycles make systems less maintainable, changeable, reusable, testable, understandable
  - Thus, dependency hierarchies should form DAGs (directed acyclic graphs)

- **General solution idea**
  - Get rid of the dependency cycle by removing one of the dependencies

---

Split Subsystems (1)

Example: When analyzing interdependencies between entities in a middleware subsystem, two (or more) sets of components could be determined. Within each of these sets there was high cohesion; between these sets only low cohesion. Thus, the subsystem was split into two parts.

Special variant: Split layer in a layered system
### Split Subsystems (2)

- **Context**
  - Cohesion within a subsystem

- **Problem**
  - Within a subsystem the interdependencies (cohesion) should be high
  - Between two subsystems in a software architecture, the degree of coupling should be rather loose
  - If the cohesion between some parts is loose, then some design decisions seem to be questionable
  - It is recommendable to change this to obtain better modularization and understandability
  - Another potential problem are subsystems/components with too many responsibilities

- **General solution idea**
  - Loose cohesion within a subsystem implies that the functionality can be split into multiple subsystems
  - Thus, determine areas with high cohesion in a subsystem. All those areas with low cohesion are candidates for becoming subsystems of their own

### Obstacles to refactoring (1)

- **Organization / management**
  - Considering improvement by refactoring as less important than providing new features
  - Ignorance of refactoring necessity typically causes design erosion
  - “Organization drives architecture” problem

- **Process support**
  - No steps / activities defined in process for architecture refactoring: Refactoring should be addressed explicitly in the process; responsibilities must be assigned to different roles
  - Refactorings are not checked for correctness, test manager not involved: Architects should work hand in hand with test manager and leverage means such as tests, architecture reviews
Obstacles to refactoring (2)

- **Technologies and tools**
  - Unavailability of tools: Refactoring must be done manually, which can be tedious and error-prone
  - Unavailability of refactoring catalog: It is important to collect refactorings and document them

- **Applicability**
  - Refactoring used instead of reengineering, and vice versa
  - Wrong order of refactorings: Should be determined by requirements priority and relevance

Refactoring and reengineering

**Agenda**

- Motivation and foundation
- Refactoring
  - Reengineering
- Summary
Reengineering – what it is

- Reengineering and refactoring might look quite similar at the beginning
- However, reengineering deals with the examination and alteration of a system to reconstitute it in a new form, and the subsequent implementation of the new form

Reengineering – how it differs from refactoring

- **Scope**: Reengineering always affects the entire system; refactoring has typically (many) local effects
- **Process**: Reengineering follows a disassembly / reassembly approach; refactoring is a behavior-preserving, structure transforming process
- **Result**: Reengineering can create a whole new system – with different structure, behavior, and functionality; refactoring improves the structure of an existing system – leaving its behavior and functionality unchanged
Reengineering – when and how to use it

- Use reengineering when
  - The system's documentation is missing or obsolete
  - The team has only limited understanding of the system, its architecture, and implementation
  - A bug fix in one place causes bugs in other places
  - New system-level requirements and functions cannot be addressed or integrated appropriately

- Process
  - **Phase I:** Reverse engineering
    - Analysis / recovery: determine existing architecture (consider using CQM)
    - SWOT analysis
    - Decisions: what to keep, what to change or throw away
  - **Phase II:** Forward engineering

Reengineering and testing

- Reengineering activities always require regression testing. For that a close cooperation between the software architect and the test manager is very important
- Test evaluation
  - The software architect must support the test manager in assessing the results of tests
- Selecting regression test cases – testing strategy
  - The software architect must support the test manager over changes in the testing strategy and in designing the required changed / new test cases

- Test automation strategy
  - The software architect must support the test manager with potentially necessary adaptations in test automation
- Design for testability
  - The software architect must determine appropriate strategies for design for testability for the new system artifacts, as well as for the artifacts to be reused from the existing system
Tool support for reengineering and refactoring

- "A fool with a tool is still a fool." [Grady Booch]. Unfortunately: "an expert without a tool is like a mule." [Personal experience]
- It is essential to leverage tools for refactoring and reengineering activities, e.g., tools for
  - Reverse engineering (UML tools provide this functionality)
  - Software architecture analysis, such as software tomography (or other CQM tools)
- Refactoring:
  - Development IDEs such as Visual Studio or Eclipse have code refactoring support
  - Architecture refactorings are not directly supported. We can use UML tools for this purpose
  - Nice to have: What-if-analysis tools – "what happens if I apply this refactoring?"

Consequence: It is never too early to analyze the applicability and availability of an appropriate tool chain and to establish appropriate process activities

Refactoring and reengineering

Agenda

- Motivation and foundation
- Refactoring
- Reengineering
- Summary
What we learned

- Refactoring changes artifacts without changing external behavior. It helps with local quality improvement and necessary changes.
- Reengineering is a complete redesign of a complete architecture and typically changes external behavior.
- Both methods are essential. Use the right one for the right purpose.
- Refactorings can be considered as transformation patterns. Refactorings are discovered, not invented. Best practices and pitfalls are potential sources.
- Different flavors of refactorings are available. Architecture refactorings might imply code refactorings.
- If refactoring is applied, make sure your environment is appropriate.
- Development process should be agile. Testing and architecture review are important.
- Requirements must drive the refactoring and reengineering activities.

A departing thought

Each problem that I solved became a rule which served afterwards to solve other problems.

[René Descartes, 1596–1650, in "Discours de la Methode"]
Design and complexity

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.

[C.A.R. Hoare]
Software Architecture Review and Evaluation

Agenda

- Architecture Reviews
- Summary
The architect's role in architecture quality assurance

- The software architect is responsible for designing the architecture, but also for assessing and maintaining it.
- He needs techniques to answer the questions:

  - Is the architecture correct?

  - Is the architecture implemented correctly?
### Initial architecture quality assurance

- Does the architecture fulfill all functional requirements?
  - Assignment of responsibilities to system components

- Does the architecture meet the quality expectations?
  - Clarification of quality goals
  - Agreement on priorities among qualities
  - Comparison of architectural options
  - Verification of tradeoffs
  - Early identification of technical risks

### Regular Architecture Reviews

- Lightweight Architecture Reviews should become regular part of the development process as they offer several benefits:
  - They might reveal dependencies between qualities and artifacts
  - They help determine whether non-functional requirements are met or can be met
  - They help motivate architecture decisions
  - They help increase quality of design and documentation
  - They serve as a means to keep all stakeholders informed and synchronized
Architecture review toolbox

- Quantitative assessments
  - Code quality assessment
  - Simulations
  - Prototypes

- Qualitative assessments
  - Experience-based approaches
  - Scenario-based approaches

Types of qualitative reviews

- Scenario-based
  - Scenario profile
  - Impact analysis
  - Assessment results
  - Quality attribute prediction
  - Predicted QA value

- Experience-based
  - Experience
  - Experience based reasoning
  - Experience based architecture
  - Possible issues
  - Strengths/opportunities
  - Weaknesses/threats
  - Mitigation evaluation
  - Measures

Bosch, 2000
### Qualitative review techniques

<table>
<thead>
<tr>
<th>Type</th>
<th>Active Design Review</th>
<th>Industry Practice</th>
<th>Software Architecture Analysis Method</th>
<th>Architecture Tradeoff Analysis Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experience-based</td>
<td>Experience-based</td>
<td>Experience-based</td>
<td>Scenario-based</td>
<td>Scenario-based</td>
</tr>
<tr>
<td>Scenario-based</td>
<td>Experience-based</td>
<td>Experience-based</td>
<td>Scenario-based</td>
<td>Scenario-based</td>
</tr>
<tr>
<td>Intention</td>
<td>Improve design, find errors</td>
<td>SWOT analysis, identify measures</td>
<td>Clarify and prioritize requirements, evaluate suitability of architecture for change scenarios</td>
<td>Clarify and prioritize requirements, find risks, sensitivity points, tradeoffs</td>
</tr>
</tbody>
</table>

#### Industry Practice

- Experience-based Reviews as Show Case
Experience-based review method

- **Purpose:** Confirm strength, find challenges and identify measures
- Reviewers are experienced architects
- System description by project externals
  - Elaboration of the key requirements
  - Elaboration of the key design elements
- Analysis and documentation of strengths, weaknesses, opportunities, and threats

- **Effort:**
  - Regular review: reviewer team 20–60 days, project team 8–16 days
  - Flash review: review team 2–3 days, project team 2–3 hours

- **Results:**
  - Detailed report including architecture description, SWOT analysis, measures

- **Benefits:**
  - Rating of a software architecture regarding compliance to its requirements,
  - dedicated measures; minimal effort for project team

Procedure: Overview

A proper architecture review comprises four steps:

- **Scoping:** what is the review all about?
- **Collection:** collect and retrieve information about the architecture with an emphasis on the review’s focus.
- **Evaluation:** how well meets the architecture the issues of interest and, if it does not, how can it be improved so that it gets back on track?
- **Feedback:** report the evaluation results back to the customer and the development team.
Procedure: Scoping

Every architecture review needs a focus!
Otherwise it is impossible to provide a valuable result back to the project team. The initial step of an architecture review is therefore dedicated to identifying the:

- **topics** of the review: what is the overall goal and what are the 3 to 5 key areas that contribute to this goal?
- **requirements** to evaluate against: what are the concrete measures regarding the goal and the key areas that the architecture under review should fulfill?
- **sources** from which the required information could be retrieved: documents, source code, a demo, test reports, and interviews (with architects, developers, testers, project managers, and product managers).

Procedure: Collection

Retrieving the relevant information about the architecture requires to „access“ multiple sources!

- **Documents** describe the „desired“ architecture, but not necessarily the implemented architecture.
- **Code, demos, and test reports** help to uncover the real architecture, its strengths and weaknesses, but do not tell whether particular deficiencies already get tackled and by what measures.
- **Interviews** with all stakeholders of the architecture will tell you how the architecture under review is received, assessed, and what the next development steps are.

Collecting information is neutral: no assessments of the retrieved information must be made.
Procedure: Interviews

Interviews are the most useful, but also the most sensitive information sources in an architecture review.

- Interviews uncover things that are neither documented nor implemented:
  how does the architecture look-and-feel, how well is it accepted, what are its good, bad, and ugly parts, what are the next development steps, what do people think about the quality of the architecture and how they think can it be improved.

- It is important that interviews are individual with each interviewee, that interviewees not communicate via official documents (you want to retrieve what they really think and know), that the information is only used anonymously within the review report, and that no interview runs longer than 1 hour. Otherwise it is hard to get an interviewee's time or honest answers to your questions.

Procedure: Evaluation

Assessing the information gathered during the collection step and drawing conclusions from it is the review's core activity.

The result of the evaluation step is a review report with the following structure:

- **Goals**: a description of the review's goal and the 3 to 5 key areas that got addressed, including the requirements for these key areas.
- **Procedure**: how got the information retrieved, condensed, assessed, documented, and communicated.
- **Description and Assessment**: A description of the software architecture from the perspective of each relevant key area, and the assessment of its quality with respect to the requirements for these areas.
- **Recommendations**: Measures for improvement, if certain parts of the reviewed architecture show deficiencies.

Be politically correct, but honest—decisions on how to proceed with the architecture or even the entire project will be made on the review results.
Procedure: Feedback

Workshops communicate the review results to the customer!

- Focus on key issues, do not run through the entire review report.
- Begin with the review goals and examined key areas to set the right scope.
- Not only mention the major weaknesses of the reviewed architecture, but also its key strengths.
- Spend most time on the suggestions for improvements, this is the information that is most important for the customer.
- Inform the architects / project team of the reviewed system before the results get presented to the customer—this avoids unpleasant surprises.

Software Architecture Review and Evaluation

Agenda

- Architecture Reviews
  - Summary
What we learned

- We need to keep track of software architecture quality in order to prevent and to identify problems early and systematically.
- Regular, lightweight reviews should be part of any development project.
- More thorough reviews are required if an existing architecture is subject of evaluation or if significant issues need to be evaluated.
- Scenario-based evaluations help to stay focused, especially when addressing non-functional requirements.

A departing thought

Engineers like to solve problems. If there are no problems handily available, they will create their own problems.

[Scott Adams]
Creating the Software Architecture

References (1)


References (2)

Creating the Software Architecture

References (3)

- A. Cockburn: Writing Effective Use Cases, Addison-Wesley, 2000
- K. Beck: eXtreme Programming Explained: Embrace the Change, Addison-Wesley, 1999

Architecture Evaluation

References (4)