Software Interfaces

Interface: how elements interact, communicate and coordinate

Elements: have interfaces controlling access to their internals

Actor: user, system or element interacting with another

Environment: set of actors interacting with an element

Resources: facilitate direct interactions (data streams, transfers…)

All elements have interfaces and they are two-way (provided and required), and elements can interact with multiple actors via the same interfaces

Concepts:

- Multiple interfaces: an interface can be split into multiple ones and supports different access levels (main, debugging, monitoring, admin)
- Resources: syntax (name, arguments and datatypes) and semantics (defines the result of invoking the resource)
    - Operations: transfer control and data to element for processing
    - Events: asynchronous, indicate message receipt or stream element arrival
    - Properties: metadata (access rights, units…)
- Interface evolution
    - Deprecation: removing interface
    - Versioning: keeping old and adding new interface
    - Extension: adding new resources to original interface

Design principles for interfaces:

- Least surprise: align behavior with actor expectations
- Small interfaces: minimiza data exchange
- Uniform access: hide implementation details
- Don’t repeat yourself: avoid redundant methods for the same task

Interface Scope: define resources available to actors

Access Control: can expose all or limit resources for security, performance…

Gateway Pattern: provide access to variety of resources with a more stable interface over time

Interaction Styles (how elements communicate and coordinate

- RPC (Remote Procedure Call): call a procedure located elsewhere on a network, translated into a message and sent to remote element

- REST (Representational State Transfer): uniform interface HTTO and URI, client-server, stateless, chacheable, tiered system architecture, code on demand

Representation and Structure of exchanged data: interfaces provide the opportunity to abstract the internal representation into a different one, and this is important for general purpose data inter-change formats

Extensible Markup Language (XML): meta-language, can define a customized language to describe data by defining an XML schema. Tags to annotate textual documents, braking info into chunks or fields and identifying datatypes of each field

JavaScript Object Notation (JSON): textual representation with its own schema language but less verbose than XML, structures data as nested name/value pairs and arrays. Much more efficient serialization and deserialization

Protocol Buffers: use data type close to programming languages (like JSON), making serialization and deserialization efficient, and it has a schema (like XML) that defines valid structure that can include required and optional elements. Binary format.

Error Handling: failed operations may throw an exception, return a status indicator; properties may be used to store data of latest successful operation; error events such as timeouts may be triggered for failed asynchronous interaction; error log may be read by connecting to an output data stream

Common sources: incorrect, invalid, illegal information sent to the interface, element in wrong state for handling event, hardware or software error, element not configured correctly

Documenting the interface: only document what actors need to interact with the interface. And need to take into account that different stakeholders need different information.

Evaluation an architecture

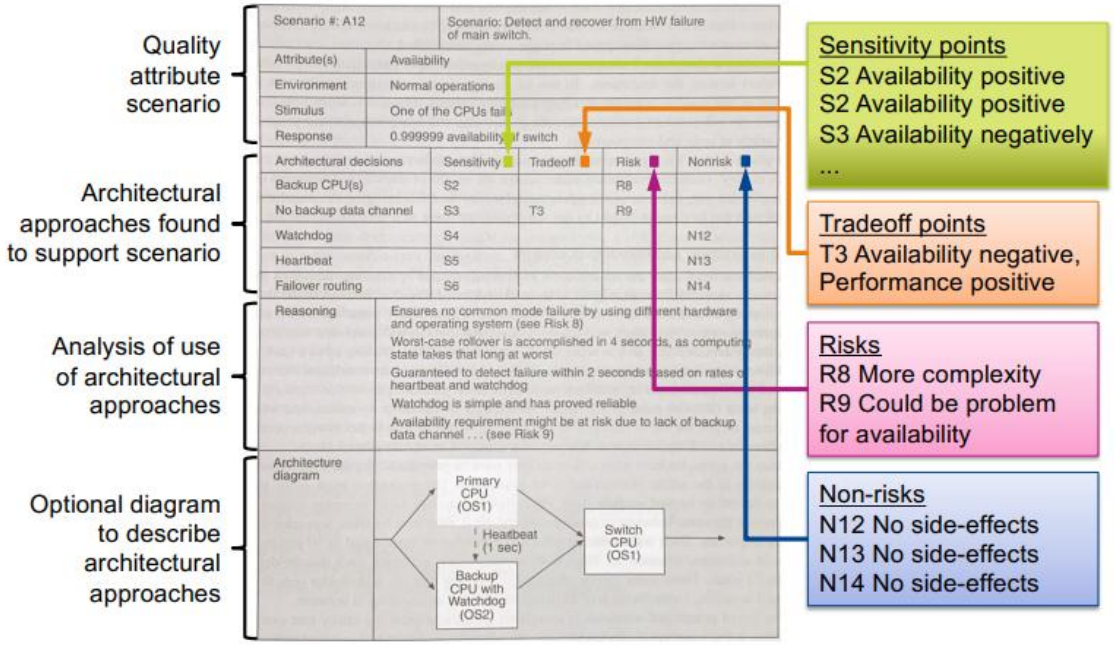
Who can perform it? The architect and/or peers within the design process, or outsiders once it has been designed

Contextual factors for evaluation: which stakeholders, who performs the evaluation, what artifacts are available, which business goals and who sees the results

ATAM (Architecture tradeoff analysis method): method to avaluate if an architecture meets quality requirements bu analyzing trade-offs

- Motivation from unsuitable architecture's disaster, project reuse, force stakeholders to meet, prioritize conflicting goals, find problems early, assurance...
- When? Early, before full specification || Late, ..
- Who are involved: evaluation team, project decision makers, architectures stakeholders

- Result of ATAM: evaluation report answers
    - Decision mapping
    - Utility tree
    - Identified sensitivity and tradeoff points
    - Concise architecture presentation
    - Defined business goals
    - Risks and non-risks
    - Risk themes
- Steps
    - 1 - Present the ATAM (evaluation team): steps, techniques, output
    - 2- Present the Business Goals (project decision makers): important functionality,, cosntraints, goals, stakeholders, ASRs…
    - 3- Present the Architecture (architect): views, constraints, interaction, architectural approaches…
    - 4- Identify Architectural Approaches (evaluation team): capture tactics and patterns, architectural approaches and architectural patterns
    - 5- Generate a Quality Attribute Utility Tree (evaluation team, project decision makers): utility root, quality attributes, subdomains, quality attribute scenarios… (use case scenarios, growth scenarios, exploratory scenarios)
    - 6 - Analyze Architectural Approaches (evaluation team): analyze top-ranked scenarios individually - architectural approaches, document key architectural decisions, identify risks, ron-risks, sensitivity points and tradeoffs… Mapping architectural decisions to quality requirements, analysis questions for each approach and architect’s reponses to these questions
        - Risk: architectural decisions that introduce project risks
        - Non-risk: safe decisions that don’t introduce risks
        - Sensitivity points: decisions that impact a quality attribute (positive or negative)
        - Tradeoff points: decisions that impact quality attributes both positively and negatively



- 7- Brainstorm and Prioritize Scenarios: invite stakeholders to propose additional scenarios to cover any gaps
- 8- Analyze the Architectural approaches: reanalyze architectural approaches using the new scenarios
- 9- Present/Document Results

Virtualization

Isolates applcations while sharing resources, can build applications as if they were only users

Shared Resources:

- Shared CPU (achieved through a thread-scheduling mechanism, all applications must go through scheduler to access the CPU)
- Shared memory (uses cirtual emory (disl) when RAM is insufficient)
- Shared Disk storage (disk controller ensures data strams to and from each thread sequentially, the OS may tag threads ad disk content with user ID, group...)
- Shared Network (isolation achieved through identification of messages)

Virtualization creates the illusion of multiple independent systems on a single machine. A **virtual machine (VM)** runs as if it were its own computer. A **hypervisor** manages VMs, with two types:

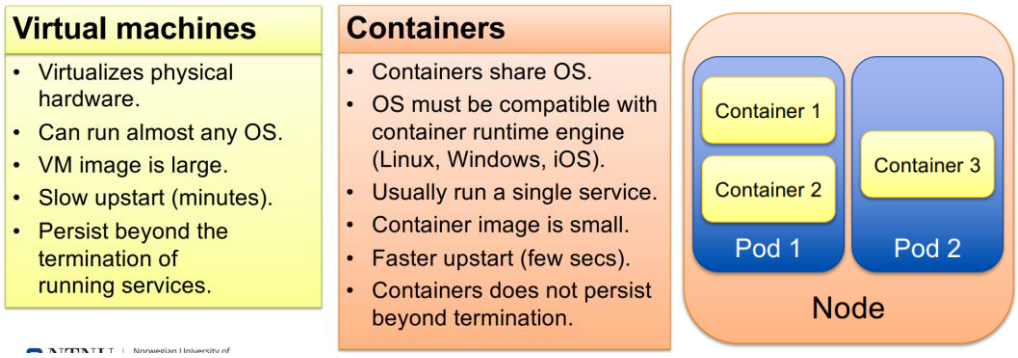
- **Type 1 (bare-metal)**: runs directly on hardware, more efficient.

- **Type 2 (hosted)**: runs on top of an OS, more overhead.

Virtualization allows better resource sharing and separation of concerns but can introduce performance penalties.

Containers are a lighter alternative — they share the host OS, start quickly, and consume less memory. However, containers must be stateless and are less isolated than VMs.

Kubernetes manages containers by grouping them into pods, enabling efficient scaling and orchestration.



Mobile Systems

Mobile systems must handle unique challenges:

- **Energy**: battery constraints require tactics like throttling usage, monitoring battery, and tolerating shutdowns.
- **Connectivity**: wireless protocols (NFC, Bluetooth, WiFi, LTE) vary in power, range, and performance. Systems must handle loss of signal, switch protocols, and defend against attacks.
- **Sensors and actuators**: used to detect environment and respond. Sensor fusion improves accuracy. Security and degraded modes are important.
- **Resources**: mobile systems are constrained by CPU, memory, and heat. Critical functions are mapped to appropriate processors (ECUs).
- **Life cycle**: updates must be safe and efficient. Testing covers components, devices, and full systems. Logging and updatability are essential, especially in long-lived systems like vehicles.

The Cloud and Distributed Computing

Cloud computing offers remote computing resources on-demand — apps, services, storage, and computation. It's based on:

- **Elasticity**: scale up/down dynamically

- **Multi-tenancy**: share infrastructure securely between users
- **Pay-per-use**: usage is measured and billed
- **Deployment models**: public, private, hybrid
- **Service models**: SaaS, PaaS, IaaS

Cloud systems face issues like **timeouts**, **long-tail latency**, and **failures**. Load balancing and autoscaling are used to handle traffic and ensure availability. Managing **state** is a challenge — can be internal, external (DB), or on client.