 Template

2018-12-20

**About arc42** arc42, the Template for documentation of software and system architecture. By Dr. Gernot Starke, Dr. Peter Hruschka and contributors. Template Revision: 7.0 EN (based on asciidoc), January 2017 © We acknowledge that this document uses material from the arc 42 architecture template, <http://www.arc42.de>. Created by Dr. Peter Hruschka & Dr. Gernot Starke.

# Introduction and Goals

The Nexus Card Game (NCG) is a collectible card game (CCG), also called a trading card game (TCG), among other names, is a kind of strategy card game that consists of specially designed sets of playing cards. Games are commonly played between two players, though multiplayer formats are also common. Players may also use dice, counters, card sleeves, or play mats to complement their gameplay.

## Requirements Overview

Two players can engage in a round-based gameplay using pre-defined, shuffled decks of cards, with different attributes that are consuming mana and cause damage to the opponent player’s health. The game is over as soon as one player runs out of health.

* Each player starts with 30 health and 0 mana and three cards in the hand stack
* Each player starts with a pre-defined, shuffled deck of 20 cards
* For each round the mana availability will increase by one to a maximum of 10

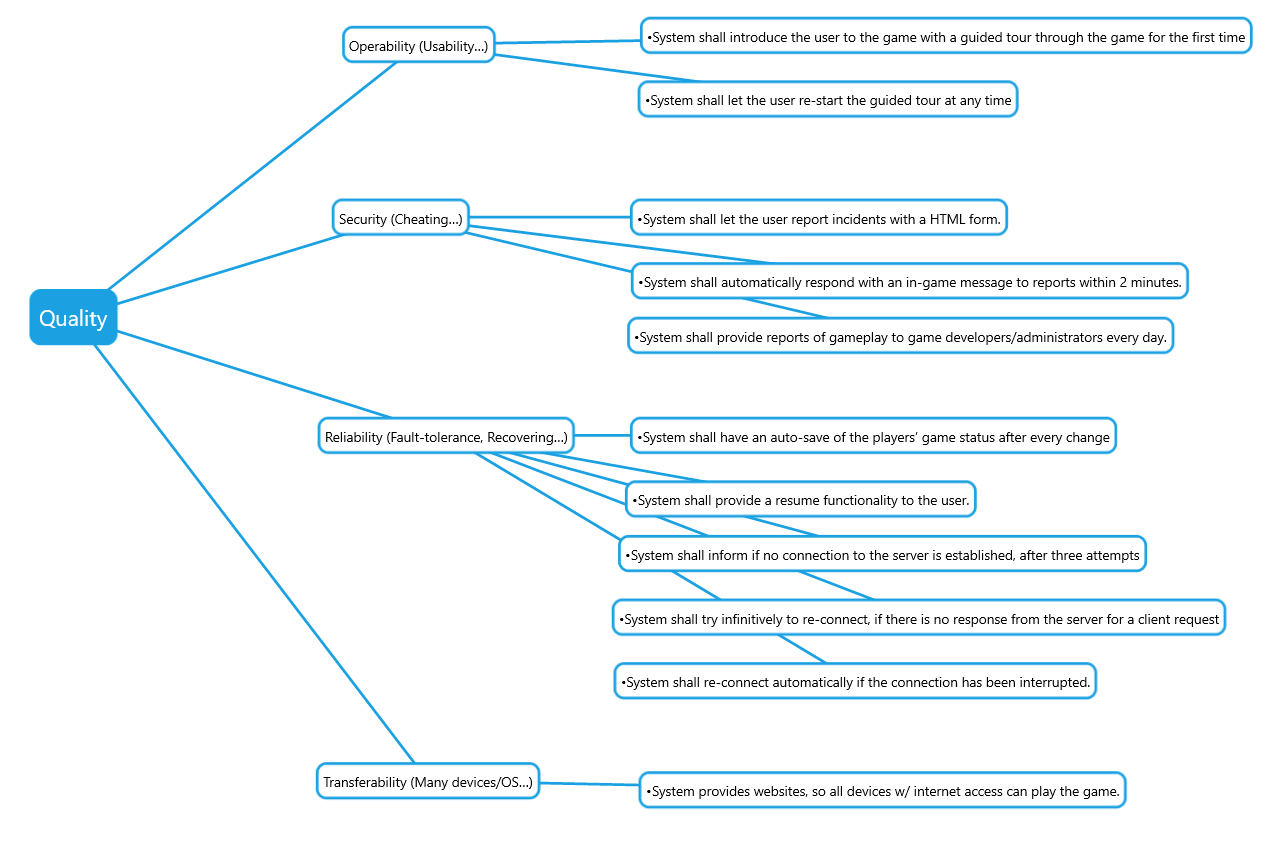
Mana is a kind of magical energy.

See spreadsheet…

## Quality Goals

For all of our stakeholders we have the same quality goals identified.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operability (Usability…)** | **Security (Cheating…)** | **Reliability (Fault-tolerance, Recovering…)** | **Transferability (Many devices/OS…)** |
| * System shall introduce the user to the game with a guided tour through the game for the first time. * System shall let the user re-start the guided tour at any time. | * System shall let the user report incidents with a HTML form. * System shall automatically respond with an in-game message to reports within 2 minutes. * System shall provide reports of gameplay to game developers/administrators every day. | * System shall have an auto-save of the players’ game status after every change * System shall provide a resume functionality to the user. * System shall inform if no connection to the server is established, after three attempts * System shall try infinitively to re-connect, if there is no response from the server for a client request * System shall re-connect automatically if the connection has been interrupted. | * System provides websites, so all devices w/ internet access can play the game. |



## Stakeholders

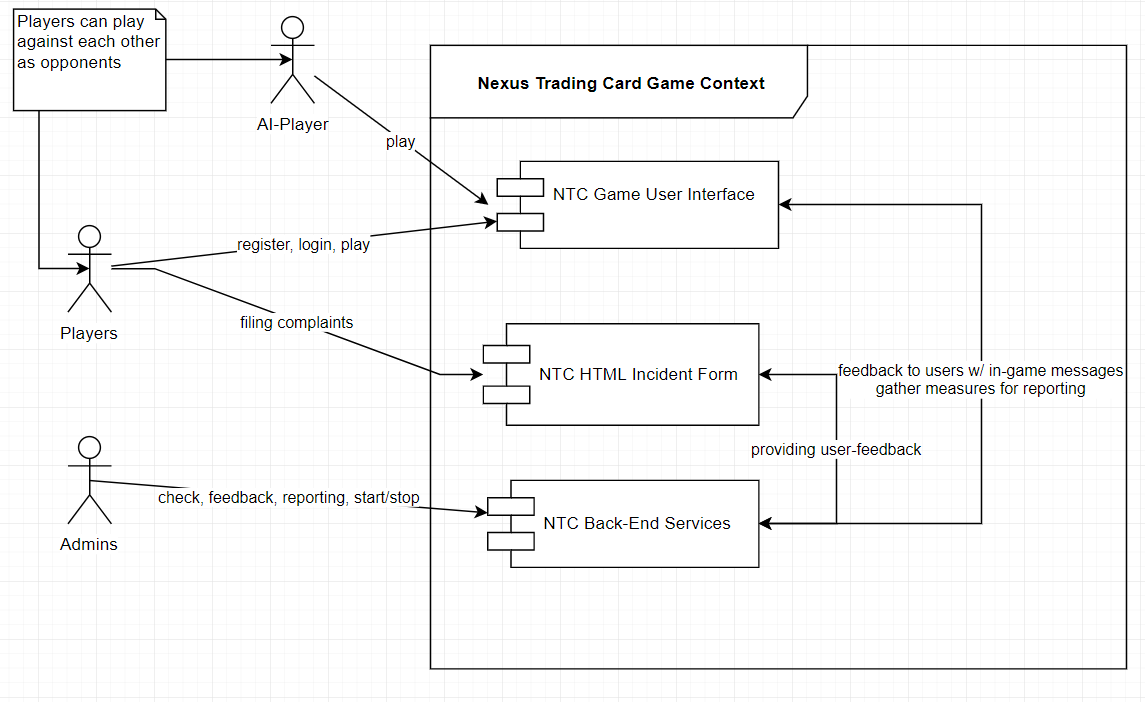
|  |  |  |
| --- | --- | --- |
| Role/Name | Contact | Expectation |
| Developer | Medi-Team | * Honing skills: Practicing Clean Code principles, OOD, integrational tests, C#, architectural documentation * Engage in team coding * Grow together as an international team * Deliverable: Detailed documentation of (third-party) interfaces to implement * Deliverable: Sample data, use cases to understand better how to implement and test |
| Architect |  | * Coding guidelines and design/architecture is respected in implementation * Receives feedback from developers * Changes to the architecture (maybe based on developers feedback) result in his ultimate decision |
| CDO | Edgar Kuner | * Guidelines of development and constraints, limits are respected * Deliverable: Business Summary, Architecture Constraints |
| End-User (Gamer) | Medi-Team and  others | * Give feedback to development * Be informed about changes to the game (Patch Notes) * Deliverable: System Requirements, Business Context and Patch Notes |
| Project Manager |  | * Staying within budget * Keep the deadlines |
| Product Manager |  | * Deliverable: Decision log to understand the product implementation |
| Technical Administrator |  | * Deliverable: Documentation about configuration and setup, Knowledge how to deploy |

# Architecture Constraints

|  |  |  |
| --- | --- | --- |
| **Organizational** | **Technical** | **Conventions** |
| * Server runs on DS-VM-MTEST01 * Release Notes are communicated via email | * C# * Oracle Database | * C# Camel-Case |

# System Scope and Context

## Business Context

****

## Technical Context

# Solution Strategy

|  |  |  |
| --- | --- | --- |
| **Goal/Requirement** | **Architecture Approach** | **Details** |
| Transferability: Many devices or OS shall be supported | Create a server-client architecture using web-technology | See later chapter |
| No C++:   * secure development can be tricky * development/implementation will be time consuming | No PHP:   * possibilities for database abstraction are limited * advanced development without frameworks is difficult and time consuming | No Perl, Catalyst:   * possibilities for database abstraction are very limited * more suitable for scripting |
| No Python:   * more suitable for scripting | JAVA, Tomcat, JBoss, Spring.IO, Hibernate   * development for web needs to be “heavily” configured * license issue could be a case in future * different tools needed (IntelliJ) | 8 ppl:  ASP.NET Core (C#), Entity Framework Core   * Core technology is very stable * Platform independent * support for a long time can be expected * community-backed and well documented * IIS needed to host web-components |
| PhaserIO (WebGL)   * Using JavaScript technology and HTML5 * Easy to learn * Big community, good documentation * Supports game physics | TypeScript, REACT   * Framework to build UI in JavaScript (TypeScript) * Maybe debugging more difficult due to virtual DOM * Security concerns due to provision by Facebook development * Big community, good documentation * Well-known and accepted in developer communities * Main focus is not on game development, but application development | 7 ppl:  Angular, Node.JS   * a lot of resources available * advanced UI can be build fast * Install of software on server needed * Creates a server that might conflict with other web-server * Supports real-time applications quite good * Nicely integrated into MEAN (MongoDB, Express.js, Angular, Node.JS) * Coding completely in JavaScript |
| Webassembly   * Browser extension of JavaScript, no special apps needed * No typical apps, only “websites” * Assemblies are compiled w/ C/C++ or RUST or other languages * Good use for heavy computation on client * Prevents a lot client-server traffic, as logic is moved from server to client * Similar to ActiveX * Client will load assembly from server as needed | HTML5   * Limited to client * Animations are limited to CSS or JavaScript * Embedding video, audio is easy * Without JavaScript HTML5 is too static for mobile game development | Oracle DB   * Commercial use could be expensive * Supporting infrastructure has to provide a lot of resources * One of the most stable DB systems available * Reliable, stable, transaction safe, fail-save * Well known, huge community and extensive documentation |
| MySQL   * No license costs * High-end configuration needs expertise * Out-of-the-box basic setup would support immediate development * Server-side implementation (stored procedures) is not as advanced as Oracle | MS SQL   * Feature availability “close” to Oracle * Might have license costs, but less expensive compared to Oracle * “Memory-hungry” behavior would reflect on infrastructure demands * Reliable, stable, transaction safe, fail-save, runtime recovery * Maintenance intensive | SQLite   * No database server needed * Could be used for a server-side cache * Could be used to abstract from direct file access, but limited to server-side * Main intension is on purely internal or temporary use * Lightweight database system * No license costs |
| NoSQL, MongoDB, Couch DB   * Advantage to handle data that is or cannot be structured * Easy maintenance * Easy to use because of no structure needed * Couch DB is best suited for mobile development * Couch DB is clustered across multiple devices/sites * MongoDB is a clustered system and such very scalable | Memcached IMDB   * Fast access due to RAM storage * Key-Value storage with generated keys * Using generated keys in every server the same way, data can be shared across servers * Client has knowledge of and can utilize many servers * Free and open-source * Many languages can be used * Suited for caching static data * Synch and distribution of data needs to be covered as service around this, because there is no link between memcached to other RDBMS | 7ppl:  Blazor, Razor   * How to code is in C# and similar to ASP.net * Code can be created as C# library and reused either for web and/or desktop (WPF) * Compiled artifacts can be deployed as Webassembly * Typesafe * Generating “client-code” similar to Angular * Intellectual Property is exposed, if used as webassembly |
| Aerospike IMDB   * Only running on LINUX/UNIX * Optimized for SSD/Flash drives * If RAM is consumed, the oldest data is removed from RAM | Redis IMDB   * Only running on POSIX * Manages in three different layers: Cluster-layer, Persistence layer and distribution layer * Risk of 2 seconds data fallout/loss * Securing random access | PostgreSQL   * <https://www.postgresql.org/> * Jack |

## Implementation

|  |  |  |  |
| --- | --- | --- | --- |
| Iterations | UI/UX | Game API | BackEnd Services |
| MEAN | Angular | Express.JS, Node.JS | Mongo |
| Core | Razor | ASP.NET Core | Entity Framework Core, SQLite |
| Blazor | Blazor | Blazor / ASP.NET | Entity Framework Core, PostgreSQL |
| IMDB | n.a. | To be choosen | |

# Building Block View



**Level 1** is the white box description of the overall system together with black box descriptions of all contained building blocks.

**Level 2** zooms into some building blocks of level 1. Thus it contains the white box description of selected building blocks of level 1, together with black box descriptions of their internal building blocks.

**Level 3** zooms into selected building blocks of level 2, and so on.

## Whitebox Overall System

Here you describe the decomposition of the overall system using the following white box template. It contains

* an overview diagram
* a motivation for the decomposition
* black box descriptions of the contained building blocks. For these we offer you alternatives:
  + use *one* table for a short and pragmatic overview of all contained building blocks and their interfaces
  + use a list of black box descriptions of the building blocks according to the black box template (see below). Depending on your choice of tool this list could be sub-chapters (in text files), sub-pages (in a Wiki) or nested elements (in a modeling tool).
* (optional:) important interfaces, that are not explained in the black box templates of a building block, but are very important for understanding the white box. Since there are so many ways to specify interfaces why do not provide a specific template for them. In the worst case you have to specify and describe syntax, semantics, protocols, error handling, restrictions, versions, qualities, necessary compatibilities and many things more. In the best case you will get away with examples or simple signatures.

***<Overview Diagram>***

Motivation

*<text explanation>*

Contained Building Blocks

*<Description of contained building block (black boxes)>*

Important Interfaces

*<Description of important interfaces>*

Insert your explanations of black boxes from level 1:

If you use tabular form you will only describe your black boxes with name and responsibility according to the following schema:

|  |  |
| --- | --- |
| **Name** | **Responsibility** |
| *<black box 1>* | *<Text>* |
| *<black box 2>* | *<Text>* |

If you use a list of black box descriptions then you fill in a separate black box template for every important building block . Its headline is the name of the black box.

### <Name black box 1>

Here you describe <black box 1> according the the following black box template:

* Purpose/Responsibility
* Interface(s), when they are not extracted as separate paragraphs. This interfaces may include qualities and performance characteristics.
* (Optional) Quality-/Performance characteristics of the black box, e.g.availability, run time behavior, ….
* (Optional) directory/file location
* (Optional) Fulfilled requirements (if you need traceability to requirements).
* (Optional) Open issues/problems/risks

*<Purpose/Responsibility>*

*<Interface(s)>*

*<(Optional) Quality/Performance Characteristics>*

*<(Optional) Directory/File Location>*

*<(Optional) Fulfilled Requirements>*

*<(optional) Open Issues/Problems/Risks>*

### <Name black box 2>

*<black box template>*

### <Name black box n>

*<black box template>*

### <Name interface 1>

…

### <Name interface m>

## Level 2

Here you can specify the inner structure of (some) building blocks from level 1 as white boxes.

You have to decide which building blocks of your system are important enough to justify such a detailed description. Please prefer relevance over completeness. Specify important, surprising, risky, complex or volatile building blocks. Leave out normal, simple, boring or standardized parts of your system

### White Box *<building block 1>*

…describes the internal structure of *building block 1*.

*<white box template>*

### White Box *<building block 2>*

*<white box template>*

…

### White Box *<building block m>*

*<white box template>*

## Level 3

Here you can specify the inner structure of (some) building blocks from level 2 as white boxes.

When you need more detailed levels of your architecture please copy this part of arc42 for additional levels.

### White Box <\_building block x.1\_>

Specifies the internal structure of *building block x.1*.

*<white box template>*

### White Box <\_building block x.2\_>

*<white box template>*

### White Box <\_building block y.1\_>

*<white box template>*

# Runtime View

**Contents.**

The runtime view describes concrete behavior and interactions of the system’s building blocks in form of scenarios from the following areas:

* important use cases or features: how do building blocks execute them?
* interactions at critical external interfaces: how do building blocks cooperate with users and neighboring systems?
* operation and administration: launch, start-up, stop
* error and exception scenarios

Remark: The main criterion for the choice of possible scenarios (sequences, workflows) is their **architectural relevance**. It is **not** important to describe a large number of scenarios. You should rather document a representative selection.

**Motivation.**

You should understand how (instances of) building blocks of your system perform their job and communicate at runtime. You will mainly capture scenarios in your documentation to communicate your architecture to stakeholders that are less willing or able to read and understand the static models (building block view, deployment view).

**Form.**

There are many notations for describing scenarios, e.g.

* numbered list of steps (in natural language)
* activity diagrams or flow charts
* sequence diagrams
* BPMN or EPCs (event process chains)
* state machines
* …

## <Runtime Scenario 1>

* *<insert runtime diagram or textual description of the scenario>*
* *<insert description of the notable aspects of the interactions between the building block instances depicted in this diagram.>*

## <Runtime Scenario 2>

## …

## <Runtime Scenario n>

# Deployment View

**Content.**

The deployment view describes:

1. the technical infrastructure used to execute your system, with infrastructure elements like geographical locations, environments, computers, processors, channels and net topologies as well as other infrastructure elements and
2. the mapping of (software) building blocks to that infrastructure elements.

Often systems are executed in different environments, e.g. development environment, test environment, production environment. In such cases you should document all relevant environments.

Especially document the deployment view when your software is executed as distributed system with more then one computer, processor, server or container or when you design and construct your own hardware processors and chips.

From a software perspective it is sufficient to capture those elements of the infrastructure that are needed to show the deployment of your building blocks. Hardware architects can go beyond that and describe the infrastructure to any level of detail they need to capture.

**Motivation.**

Software does not run without hardware. This underlying infrastructure can and will influence your system and/or some cross-cutting concepts. Therefore, you need to know the infrastructure.

Maybe the highest level deployment diagram is already contained in section 3.2. as technical context with your own infrastructure as ONE black box. In this section you will zoom into this black box using additional deployment diagrams:

* UML offers deployment diagrams to express that view. Use it, probably with nested diagrams, when your infrastructure is more complex.
* When your (hardware) stakeholders prefer other kinds of diagrams rather than the deployment diagram, let them use any kind that is able to show nodes and channels of the infrastructure.

## Infrastructure Level 1

Describe (usually in a combination of diagrams, tables, and text):

* the distribution of your system to multiple locations, environments, computers, processors, .. as well as the physical connections between them
* important justification or motivation for this deployment structure
* Quality and/or performance features of the infrastructure
* the mapping of software artifacts to elements of the infrastructure

For multiple environments or alternative deployments please copy that section of arc42 for all relevant environments.

***<Overview Diagram>***

Motivation

*<explanation in text form>*

Quality and/or Performance Features

*<explanation in text form>*

Mapping of Building Blocks to Infrastructure

*<description of the mapping>*

## Infrastructure Level 2

Here you can include the internal structure of (some) infrastructure elements from level 1.

Please copy the structure from level 1 for each selected element.

### *<Infrastructure Element 1>*

*<diagram + explanation>*

### *<Infrastructure Element 2>*

*<diagram + explanation>*

…

### *<Infrastructure Element n>*

*<diagram + explanation>*

# Cross-cutting Concepts

**Content.**

This section describes overall, principal regulations and solution ideas that are relevant in multiple parts (= cross-cutting) of your system. Such concepts are often related to multiple building blocks. They can include many different topics, such as

* domain models
* architecture patterns or design patterns
* rules for using specific technology
* principal, often technical decisions of overall decisions
* implementation rules

**Motivation.**

Concepts form the basis for *conceptual integrity* (consistency, homogeneity) of the architecture. Thus, they are an important contribution to achieve inner qualities of your system.

Some of these concepts cannot be assigned to individual building blocks (e.g. security or safety). This is the place in the template that we provided for a cohesive specification of such concepts.

**Form.**

The form can be varied:

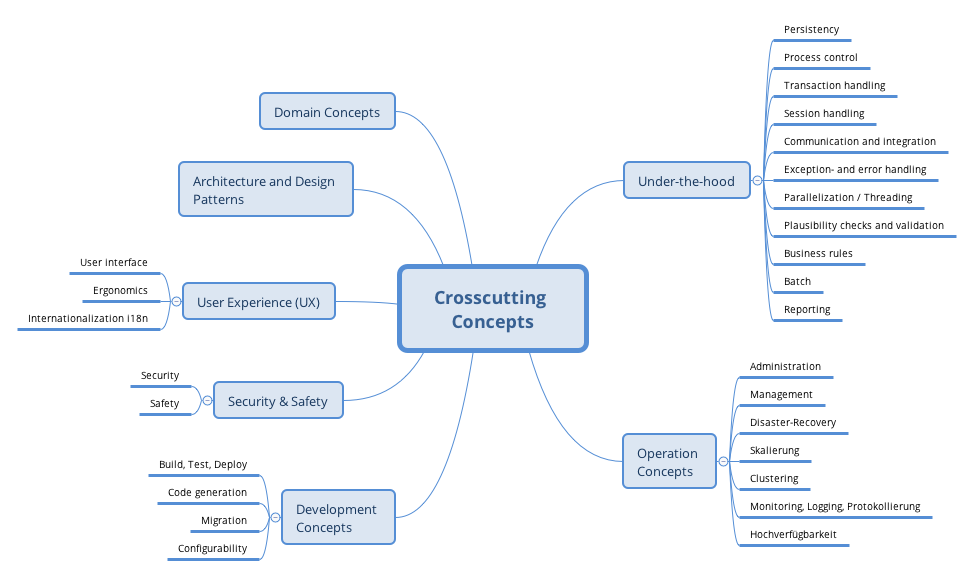
* concept papers with any kind of structure
* cross-cutting model excerpts or scenarios using notations of the architecture views
* sample implementations, especially for technical concepts
* reference to typical usage of standard frameworks (e.g. using Hibernate for object/relational mapping)

**Structure.**

A potential (but not mandatory) structure for this section could be:

* Domain concepts
* User Experience concepts (UX)
* Safety and security concepts
* Architecture and design patterns
* "Under-the-hood"
* development concepts
* operational concepts

Note: it might be difficult to assign individual concepts to one specific topic on this list.



## *<Concept 1>*

*<explanation>*

## *<Concept 2>*

*<explanation>*

…

## *<Concept n>*

*<explanation>*

# Design Decisions

**Contents.**

Important, expensive, large scale or risky architecture decisions including rationals. With "decisions" we mean selecting one alternative based on given criteria.

Please use your judgement to decide whether an architectural decision should be documented here in this central section or whether you better document it locally (e.g. within the white box template of one building block).

Avoid redundancy. Refer to section 4, where you already captured the most important decisions of your architecture.

**Motivation.**

Stakeholders of your system should be able to comprehend and retrace your decisions.

**Form.**

Various options:

* List or table, ordered by importance and consequences or:
* more detailed in form of separate sections per decision
* ADR (architecture decision record) for every important decision

# Quality Requirements

**Content.**

This section contains all quality requirements as quality tree with scenarios. The most important ones have already been described in section 1.2. (quality goals)

Here you can also capture quality requirements with lesser priority, which will not create high risks when they are not fully achieved.

**Motivation.**

Since quality requirements will have a lot of influence on architectural decisions you should know for every stakeholder what is really important to them, concrete and measurable.

## Quality Tree

**Content.**

The quality tree (as defined in ATAM – Architecture Tradeoff Analysis Method) with quality/evaluation scenarios as leafs.

**Motivation.**

The tree structure with priorities provides an overview for a sometimes large number of quality requirements.

**Form.**

The quality tree is a high-level overview of the quality goals and requirements:

* tree-like refinement of the term "quality". Use "quality" or "usefulness" as a root
* a mind map with quality categories as main branches

In any case the tree should include links to the scenarios of the following section.

## Quality Scenarios

**Contents.**

Concretization of (sometimes vague or implicit) quality requirements using (quality) scenarios.

These scenarios describe what should happen when a stimulus arrives at the system.

For architects, two kinds of scenarios are important:

* Usage scenarios (also called application scenarios or use case scenarios) describe the system’s runtime reaction to a certain stimulus. This also includes scenarios that describe the system’s efficiency or performance. Example: The system reacts to a user’s request within one second.
* Change scenarios describe a modification of the system or of its immediate environment. Example: Additional functionality is implemented or requirements for a quality attribute change.

**Motivation.**

Scenarios make quality requirements concrete and allow to more easily measure or decide whether they are fulfilled.

Especially when you want to assess your architecture using methods like ATAM you need to describe your quality goals (from section 1.2) more precisely down to a level of scenarios that can be discussed and evaluated.

**Form.**

Tabular or free form text.

# Risks and Technical Debts

**Contents.**

A list of identified technical risks or technical debts, ordered by priority

**Motivation.**

“Risk management is project management for grown-ups” (Tim Lister, Atlantic Systems Guild.)

This should be your motto for systematic detection and evaluation of risks and technical debts in the architecture, which will be needed by management stakeholders (e.g. project managers, product owners) as part of the overall risk analysis and measurement planning.

**Form.**

List of risks and/or technical debts, probably including suggested measures to minimize, mitigate or avoid risks or reduce technical debts.

# Glossary

**Contents.**

The most important domain and technical terms that your stakeholders use when discussing the system.

You can also see the glossary as source for translations if you work in multi-language teams.

**Motivation.**

You should clearly define your terms, so that all stakeholders

* have an identical understanding of these terms
* do not use synonyms and homonyms

**Form.**

A table with columns <Term> and <Definition>.

Potentially more columns in case you need translations.

|  |  |
| --- | --- |
| Term | Definition |
| <Term-1> | <definition-1> |
| <Term-2> | <definition-2> |