# Utility trees and quality attributes

There are two types of requirements for software projects: functional and non-functional requirements. Functional requirements are the requirements for what the solution must do (which  are usually expressed as use cases or stories). The functional requirements are what the users (or systems) that interact with the system do with the system (fill in an order, update customer details, authorize a loan etc.).

Non-Functional requirements are attributes the system is expected to have or manifest. These usually include requirements in areas such as performance, security, availability etc. A better name for non-functional requirements is “Quality Attributes” . Below are some formal definitions from IEEE standad 1061 “Standard for a Software Quality Metrics Methodology”  for quality attributes and related terms:

* **Quality attribute** A characteristic of software, or a generic term applying to quality factors, quality subfactors, or metric values.
* **Quality factor** A management-oriented attribute of software that contributes to its quality.
* **Quality subfactor** A decomposition of a quality factor or quality subfactor to its technical components.
* **Metric value** A metric output or an element that is from the range of a metric.
* **Software quality metric** A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality.

Most of the requirements that drive the design of a software architecture comes from system’s quality attributes. The reason for this is that that the effect of quality attributes is usually system-wide (e.g. you wouldn’t want your system to have good performance only in the UI – you want the system to perform well no matter what) –  which is exactly what software architecture is concerned with. Note however, that few requirements might still come from functional requirements).  The question is how do we find out what those requirements are?

The answer to that is also in the software architecture definition. The source for quality attributes are the stakeholders. So what or who are these “stakeholders”? Well, a stakeholder is just about anyone who has a vested interest in the project. A typical system has a lot of stakeholders starting from the (obvious) customer, the end-users (those people in the customer organization/dept that will actually use the software) and going to the operations personnel (IT – those who will have to keep the solution running), the development team, testers, maintainers, management. In some systems the stakeholders can even be the shareholders or even the general public (Imagine for example, that you build a new dispatch system for a 911 center).

One of the architect’s roles is to analyze the quality attributes and define an architecture that will enable delivering all the functional requirements while supporting the quality attributes. As can be expected ,sometimes quality attributes  are in conflict with each other – the most obvious examples are performance vs. security or flexibility vs. simplicity and the architect’s role is to strike a balance between the different quality attributes (and the stakeholders) to make sure the overall quality of the system is maximized.

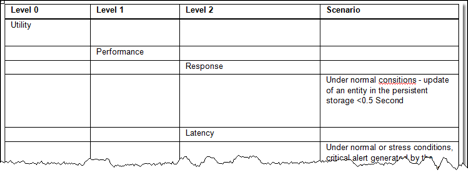
Contextual solutions (e.g. patterns) can be devised to solve specific quality attributes need. However saying that a system needs to have “good performance” or that it needs to be “testable” doesn’t really help us know what to do. In order for us to be able to discern which patterns apply to specific quality attribute ,  we need a better understanding of quality attributes besides the formal definition, something that is more concrete.

The way to get that concrete understanding of the effect of quality attributes is to use scenarios. Scenarios are short, “user story”-like proses that demonstrate how a quality attribute is manifested in the system using a functional situation

Quality attributes scenarios originated as a way to evaluate software architecture. The Software Engineering Institute developed several evaluation methodologies, like Architecture Tradeoff Analysis Method  (Clements, Kazman and Klein, 2002) that heavily build on scenarios to contrast and compare how the different quality attributes are met by candidate architectures. ATAM (and similar evaluation methods like LAAAM which is part of MSF 4.0) suggest building a “utility tree” which represent the overall usefulness of the system. The scenarios serve as the leafs of the utility tree and the architecture is evaluated by considering how the architecture makes the scenarios possible.

Using scenarios and the utility tree approach early in the design of the architecture can greatly enhance the quality of the architecture that is produced. When you examine  the scenarios you can also  prioritize them and better balance conflicting attributes.

The scenarios can be used as an input to make sure the quality attributes are actually met.  Furthermore  you can use the scenarios to help identify the strategies or patterns  applicable to make the scenarios possible (and thus ensure the quality attributes are met) within the  system. We usually group scenarios into a “utility tree” which is a representation of the total usefulness (“utility”) of a system . As you can see in the diagram below we have the   key quality attributes (Performance, Security etc.). Each of the quality attributes has sub categories (e.g. Performance is broken into latency, data loss etc.). Each sub category is demonstrated by a scenario that we expect the system to manifest.



The tree representation helps get the whole picture but the important bits here are the scenarios so let’s explore them some more. Scenarios are expressed as statements that have  3 parts: a **stimulus**, a **context** and a **response**. The  stimulus is the action taken (by the system / user/ other system / any other person); response is how the system is expected to behave when the stimulus occur, and the context specifies  the environment or conditions under which we expect the to get the response. For example in the following scenario: “When you perform a database operation , under normal condition, it should take less than 100 miliseconds.”

* “Under normal condition” is the **context**
* “When you perform a database operation” is the **stimulus**
* “it should take less than 100 millisecond” is the **response** expected from the system.

Here are a couple of  additional examples for quality attribute scenarios:

* Performance –>Latency -> under normal conditions a client consuming multiple services should have latency less than 5 seconds.
* Security->Authentications -> Under all conditions, any call to a service should be authenticated using X.509 certificate