Requirements Engineering and Design Thinking: From why and why ever to what and how

Manfred Broy
From programs to cyber-physical systems

• Programs:
  ◊ mappings states to states or data to data,
  ◊ supposed to terminate (exception OS),
  ◊ time and interaction (often) not an issue,
  ◊ concept of computation: Turing machines – algorithms

• Cyber-physical systems:
  ◊ connected to the physical world,
  ◊ need a coherent model of context, interface, interaction, time, architecture, state, probability, data and event flow, perhaps even space, geometry and movement
  ◊ concept of computation: interaction, generalized timed Mealy machines
  ◊ extensive requirements for dependability
The key role of requirements engineering (RE) in SSE

- RE is a key to software & systems engineering (SSE)
- Functionality
  - what is the needed functionality
  - do systems offer the needed functionality
  - are unneeded functions excluded
  - Functional quality
    - usability: is the functionality easy to access
    - safety and security
    - ...
- Nonfunctional quality
  - Reusability
  - Changeability
  - Portability
  - ...

REConf, Munich March 2018  Manfred Broy
Making requirements explicit

Do we distinguish between
• the system as required
• the system as implemented?

If yes, we need documentation!

When do we decide about requirements?

• Up front: Before we start implementation?
• Iterative and incremental: While we carry out implementation?
• After mortem: after implementation?
• Not at all: No distinction between implementation and requirements: It is the code that counts!
The challenge: machine learning

• Neuronal net learns from a training set
  ◊ The training set is chosen according to the required functionality

• Result is an algorithm – being a black box
  ◊ No spec
  ◊ No verification
Agile development: finding requirements on the way

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

Questions

- Where is the decision about the requirements?
- How are requirements documented?
The roles in RE: who decides what!

- **Product manager**
  - Decides about the goals and the key requirements

- **Requirements engineer**
  - Responsible for the methodology applied and
  - The quality of the requirements artifacts

- **Requirements manager**
  - Responsible for the requirements life cycle

- **Architect**
  - Responsible for reflecting the requirements in the overall system structure

- **The verifier (tester)**
  - Needs requirements to define test cases

- **Stakeholders requirement and sources**
  - Bring in their needs and expectations
Deficiencies in practice

- Wrong assumptions about functionality:
  - high discrepancies between expected use and effective use
- Role of a requirements engineer or a product manager missing
  - Product managers and architects responsible for requirements
- Requirements incomplete and description inadequate
  - missing structuring
- Requirements not reviewed and not validated
- Requirements finally not documented
  - Documentation not updated
  - In software evolution unclear what is required
- Verification starts too late
  - Only during test case engineering insufficient requirements identified
Crucial Aspects of Requirements Engineering

• Requirements – Quality in Use
  ◊ Functionality
  ◊ MMI
  ◊ External Quality

• Architecture
  ◊ Structuring
  ◊ Modularity
  ◊ Reusability

• Quality
  ◊ External
  ◊ Internal

• Evolution
  ◊ Time to market
The challenges

• To find out what is actually needed and what is feasible – innovation: who knows – design thinking
  ◊ Which functionality
  ◊ Usability
  ◊ New ideas

• Having a prototype – how do we know what is essential
  ◊ Identifying requirements – elicitation

• Achieving structured requirements
  ◊ Functional architecture – use cases – detailed specification
    • Real time
  ◊ Functional quality: safety, reliability, security, usability
    • Probabilities
  ◊ Quality beyond functionality

• Managing requirements: implementation, verification, change
Design thinking uses the designer’s sensibility and methods to match people’s needs with what is technologically feasible and what a viable business strategy can convert into customer value and market opportunity.

Design thinking – find innovative functionality and usability – user centric engineering

The life cycle:

The missing link: How to extract from the prototype the requirements
Why: the business case

Why is it interesting to develop a specific functionality: the business case - innovation

• Individual solution
  ◊ higher efficiency
  ◊ higher quality

• Standard product
  ◊ innovative functionality

• Embedded
  ◊ better product
Aspects (attributes) of requirements

• Characteristic
  ◊ Functional or quality

• Level of detail
  ◊ From abstract to concrete

• Ways to express it
  ◊ from informal to semiformal to formal

• Source
  ◊ Where it came from

• Significance
  ◊ Must or may

• Status
  ◊ Accepted, implemented, verified
System and its context
From the informal to the formal

• In the beginning, properties of the universe of discourse are formulated in natural language, in general
  “The airbag is activated within 200 msec whenever the crash sensor indicates a crash”

• The step to the formal means
  ◊ Derivation of a “data” model: Introducing a set of attributes forming an ontology
  ◊ Capturing properties by assertions in terms of these attributes

• This step into formalization has two aspects
  ◊ Abstraction: the attributes can only address a limited set of properties
  ◊ Precision: informal properties are made precise
    This includes
      • Decisions: there are usually several ways to make an informal property precise
Example: Assertions

- For a simple universe of discourse **Car** representing cars, consider attributes such as
  - length: Car → IN
  - number_of_seats: Car → IN
  - speed: Car → IN
  - situation: Car → {city, country, high_way}
- Based on the attributes, given \( d \in \text{Car} \), we write logical expressions such as
  \[
  \text{speed}(d) \geq 50 \land \text{situation}(d) = \text{city}
  \]
- This notation can be simplified for a fixed car \( d \):
  \[
  \text{speed} \geq 50 \land \text{situation} = \text{city}
  \]
- Such a logical expression referring to the attributes of the elements of the considered universe is called **assertion**.
Sets of typed channels

\[ I = \{x_1 : T_1, x_2 : T_2, \ldots \} \]

\[ O = \{y_1 : T'_1, y_2 : T'_2, \ldots \} \]

syntactic interface

\( (I \triangleright O) \)

data stream of type \( T \)

\[
\text{STREAM}[T] = \{\text{IN}\backslash\{O\} \rightarrow T^*\}
\]

valuation of channel set \( C \)

\[
[C] = \{C \rightarrow \text{STREAM}[T]\}
\]

interface behaviour for syn. interface \((I \triangleright O)\)

\[
[I \triangleright O] = \{[I] \rightarrow \emptyset([O])\}
\]
Interface Assertion

• Given a syntactic interface \((I \triangleright O)\) with
  ◦ a set \(I\) of typed input channels and
  ◦ a set \(O\) of typed output channels,

  The channels form attributes in assertions.

• an interface assertion is a logical formula with the channel identifiers in \(I\) and \(O\) as free logical variables denoting streams of the respective types.
Example: Component interface specification

A transmission component TMC

Spec name

Input channel

TMC

<table>
<thead>
<tr>
<th>in</th>
<th>x: T</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>y: T</td>
</tr>
<tr>
<td>x ~ y</td>
<td></td>
</tr>
</tbody>
</table>

Output channel

Interface assertion

x ~ y (m T: m#x = m#y)
How to structure system functionality?

• Typically systems offer a rich functionality structured into functional features

• A functional feature can be represented by some interface behavior $[I \rightarrow O]$

• Interface behavior of functional features can be composed the same way as sub-systems are composed
What is a feature ...

- Is a **feature** just a name ... ?
  - If yes – for what?
  - What is the relation of a **feature tree** to system models?

- What are **relation** between features?
  - Feature interactions?
  - Requires?
  - Excludes?

- Is there a way to **model features**?
  - How can we find and identify features of a system?
  - What is the semantic interpretation of a feature tree?

- Is there a way to interpret relations between features such as **feature interactions**?
We concentrate on functional (behavioral) features!

◊ These are at the level of system level interface behavior!

• A (functional) feature is a sub-function of a multi-functional system

◊ that serves a certain purpose
Modeling functional (behavioral) features

• We give an interpretation of the notion of a (functional) feature in terms of the system interface model $F \in [I \rightharpoonup O]$

• The functionality of a system is modeled by its interface behavior

• A (functional) feature is modeled by the
  ◇ projection applied to $F$ to the sub-interface $(I' \rightharpoonup O')$ resulting in a sub-interface behavior $F' \in [I' \rightharpoonup O']$
  ◇ absence of feature interactions is modeled by faithful projections
  ◇ feature interactions are modeled by modes
Modularity: Rules of compositions for interface specs

F1
\[ \text{in } x_1, z_{21}: T \]
\[ \text{out } y_1, z_{21}: T \]
\[ S_1 \]

F2
\[ \text{in } x_2, z_{12}: T \]
\[ \text{out } y_2, z_{21}: T \]
\[ S_2 \]

F1 $\&$ F2
\[ \text{in } x_1, x_2: T \]
\[ \text{out } y_1, y_2: T \]
\[ z_{12}, z_{21}: S_1 \cup S_2 \]
Feature Specification – Constructive Approach
Combining Functions without Interference

Given two functions $F_1$ and $F_2$ in isolation

We want to combine them into a function $F_1 \oplus F_2$
Combining Functions without Interference

Their isolated combination

\[ F_1 \oplus F_2 \]
If services $F_1$ and $F_2$ have feature interaction we get:

We explain the functional combination $F_1 \otimes F_2$ as a refinement step.
The steps of function combination

Given the isolated function $F_1$

We construct a refinement $F'_1$

And combine $F'_1$ with a refinement $F'_2$ of $F_2$
Feature Specification – Analytic Approach
From overall syntactic system interfaces ...
to ...
sub-interfaces
Projection of assertions describing features

Given:

$$(I' \triangleright O') \textbf{ subtype } (I \triangleright O)$$

and interface assertion $S$ for syntactic interface $(I\sqsubseteq O)$; we define its \textit{projection} onto the \textit{feature} with the syntactic interface $(I' \triangleright O')$ by

$$I\backslash I', O\backslash O': S$$

The projection is called \textit{faithful}, if

$$( I\backslash I', O\backslash O': S ) \iff ( O\backslash O': S )$$

Then the feature with syntactic interface $(I' \triangleright O')$ is \textit{free of feature interactions}.
Example: Component interface specification – Airbag Controller

An air bag controller

```
AB_Cont
  in  x: T
  out y: T

x >200> y
```

\[ x >200> y \equiv (\text{Time: } t \in \text{Time:} \crash\_sig \in x(t) \iff \act\_airbag \in y(t+200)) \]
Example: Component interface specification – Airbag Controller

An air bag controller

\[
\begin{align*}
\text{AB}_\text{Cont} \\
\text{in} & \quad \text{x: T, m: \{on, off\}} \\
\text{out} & \quad \text{y: T} \\
\text{x} & \quad \text{>200> y}
\end{align*}
\]

\[
x >200 > y \equiv ( \quad t \in \text{Time}: \\
(\text{ON}(m, t+199) \land \text{crash\_sig} \in x(t)) \iff \text{act\_airbag} \in y(t+200)
\]

\[
\text{ON}(m, t) = \begin{cases} 
\text{false} & \text{if } t = 0 \\
\text{true} & \text{if } \text{on} \in m(t) \\
\text{false} & \text{if } \text{off} \in m(t) \\
\text{ON}(m, t-1) & \text{else}
\end{cases}
\]

Manfred Broy 37
Given composable features $f \in F$ with specified by interface assertions $B_f$ the assertion of the functional specification reads

$$\land \{B_f : f \in F\}$$

and the interface assertion of the composed is given by hiding the mode channels in $M$

$$\exists M : \land \{B_f : f \in F\}$$
An interpreted feature tree

\[
\begin{align*}
F_1, \ldots, F_n \\
F_1 \quad & F_2 \quad \ldots \quad F_{k}, \quad F_{k+1} \quad \ldots \quad F_{n-1} \quad F_n
\end{align*}
\]
Feature interaction in the architecture view

Table 4.2: Extent of dependencies in the vehicle function graph

<table>
<thead>
<tr>
<th>Vehicle functions…</th>
<th>Number</th>
<th>Ratio</th>
<th>Number</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>with incoming dependencies</td>
<td>36</td>
<td>65.5%</td>
<td>81</td>
<td>86.2%</td>
</tr>
<tr>
<td>with outgoing dependencies</td>
<td>29</td>
<td>52.7%</td>
<td>72</td>
<td>76.6%</td>
</tr>
<tr>
<td>with incoming and outgoing dependencies</td>
<td>27</td>
<td>49.1%</td>
<td>68</td>
<td>72.3%</td>
</tr>
<tr>
<td>without dependencies</td>
<td>17</td>
<td>31.0%</td>
<td>9</td>
<td>9.6%</td>
</tr>
</tbody>
</table>

Taken from:
System Properties at Different Levels of Abstractions: Relating Views
Example: Relating Levels of Abstraction

<table>
<thead>
<tr>
<th>Logical_level</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>crash $\Rightarrow$ air_bag</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Technical_level</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>crash_sensor $\Rightarrow$ activate_air_bag</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Example: Relating Levels of Abstraction

Logical_level

...  

\text{crash} \Rightarrow \text{air\_bag}  

...  

Technical_level

...  

\text{crash\_sensor} \Rightarrow \text{activate\_air\_bag}  

...  

Translator

...  

\text{crash} \Leftrightarrow \text{crash\_sensor}  

\text{air\_bag} \Leftrightarrow \text{activate\_air\_bag}  

...
Why translators are useful?

• Translators relate requirements in terms of assertions to technical/physical assertions
• They force us to make explicit assumptions behind physical/technical designs
  ◊ As part of specifications
  ◊ To validate them – to discover invalid assumptions

• Thrust reversal can only be activated, if airplane is on the ground
• Sensor AoG yields true if and only if airplane is on the ground
• \( \neg \text{AoG} \Rightarrow \text{Thrust reversal cannot be activated} \)
Example: derived Link between two Ontologies

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁: ...</td>
<td>S₁: ...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Rₖ: Thrust reversal can only be activated, if airplane is on the ground</td>
<td>Sᵢ: Sensor AoG yields true if and only if airplane is on the ground</td>
</tr>
<tr>
<td>...</td>
<td>Sᵢ₊₁: ¬AoG ⇒ Thrust reversal cannot be activated</td>
</tr>
</tbody>
</table>

...
Traceability in Software and System Development
Illustration: Forward Tracing

Requirements → Specification → Architecture → Implementation

- Manfred Broy
- REConf, Munich March 2018
Meaning of Links and Traces

A link relates two syntactic named content chunks – formalized as assertions

• A link has a meaning that usually is related to the meaning of the assertions it relates.

• A link states a proposition about the relationship between its source and its target.

• A link can be valid or invalid.
  ◦ It is called valid, if the proposition associated with the link is true.
  ◦ Otherwise it is called invalid.
Three Artifacts
Three Levels of System Specification

- **Requirements** - system level
  - List of requirements - functional system property
  - Example: “The activation of safety relevant functions by the pilot is always double checked for plausibility by the system.”

- **Functional specification** - system level
  - Decomposition of system functionality in hierarchy of (sub-)functions
  - Specification of (sub-)functions
  - Specification of dependencies (feature interactions) between (sub-) functions based on a mode concept
  - Example: “Thrust reversal may only be activated, if the plane is on the ground.”

- **Architecture specification** - component level
  - Decomposition of systems in sub-systems (component)
  - Relationship to data flow diagram
  - Interface specification of component
  - Example: “The weight sensor indicates that the plane is on the ground.”
Illustrating Examples: Content Chunks

• System level requirements (functional requirements)
  "the car must not increase its speed without user’s control”

• System level functional specification
  "the function acc (adaptive cruise control) accelerates the car up to the speed selected by the user, provided no obstacles are recognized in front”

• Architecture specification
  "the radar signal based sensor measures the distance to the car in front and sends it to the acc monitor every 100 ms”

To formalize these statements by assertions we need appropriate ontologies on all three levels

In development we want to relate them by refinement or tracing
From content chunks to assertions

- To go from content chunks such as
  "the car must not increase its speed without user’s control"
  "the function acc (adaptive cruise control) accelerates the car up to the speed selected by the user, provided no obstacles are recognized in front"

needs modeling and formalization.

This involves the following steps

- Formalizing the elements of the universe – elicitation of the problem domain
  - Selecting the attributes
  - Defining basic propositions (called the problem domain theory)
    \( \forall (\text{speed} \leq 500) \)

- Expressing the informal statement by an assertion
Three levels of system description in logic

• system level requirements

\[ A = \bigwedge \{ A_r : r \in R \} \]

• functional specification at system level - functionality

\[ B = \bigwedge \{ B_f : f \in F \} \]

• architecture specification

\[ C = \bigwedge \{ C_k : k \in K \} \]

• Correctness

◊ functional specification correct w.r.t to requirements:

\[ B \Rightarrow A \]

◊ architecture correct w.r.t to functional spec (let \( M \) be the set of mode channels):

\[ C \Rightarrow \exists M : B \]
Relational view: Inter-artifact links and traces

<table>
<thead>
<tr>
<th>Function</th>
<th>Safety</th>
<th>Priority</th>
<th>Component</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>...</td>
<td>Yes</td>
<td>high</td>
<td></td>
</tr>
<tr>
<td>A_2</td>
<td>...</td>
<td>No</td>
<td>medium</td>
<td></td>
</tr>
<tr>
<td>A_n</td>
<td>...</td>
<td>no</td>
<td>low</td>
<td></td>
</tr>
</tbody>
</table>
Illustration: correctness and refinement

every assertion in the specification has to be guaranteed by the assertions of the architecture

Can we find and identify them?
### Relationship: req spec to function spec - tracing

<table>
<thead>
<tr>
<th>sub-function reqs</th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
<th>A₄</th>
<th>A₅</th>
<th>A₆</th>
<th>A₇</th>
<th>A₈</th>
<th>A₉</th>
<th>.</th>
<th>.</th>
<th>.</th>
<th>.</th>
<th>Aₖ</th>
</tr>
</thead>
<tbody>
<tr>
<td>B₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bₙ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Red**: Bᵢ is strong guarantor of Aⱼ  
**Yellow**: Bᵢ is weak guarantor of Aⱼ  
**Green**: Bᵢ is not a weak guarantor of Aⱼ
**Relationship: architecture to requirements**

<table>
<thead>
<tr>
<th>sub-system reqs</th>
<th>A_1</th>
<th>A_2</th>
<th>A_3</th>
<th>A_4</th>
<th>A_5</th>
<th>A_6</th>
<th>A_7</th>
<th>A_8</th>
<th>A_9</th>
<th>...</th>
<th>...</th>
<th>A_k</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Red:** C_i is strong guarantor of A_j  
**Yellow:** C_i is weak guarantor of A_j  
**Green:** C_i is not a weak guarantor of A_j
Interfaces with assumptions

Often, in an interface specification for the syntactic interface \((I \rightarrow O)\) we include

- an **assumption** \(asu(y, x)\) which is a specification of the *inverse* interface \((O \rightarrow I)\) and defines properties of the context

- a **commitment** \(cmt(y, x)\) which is a specification of the behavior the syntactic interface \((I \rightarrow O)\) as long as the assumption is fulfilled.

This leads to the specification

\[
asu(y, x) \Rightarrow cmt(y, x)
\]
Example: System interface specification

A transmission component TMCWA

TMCWA

\[
\begin{array}{l}
\text{in} \quad x: T \\
\text{out} \quad y: T \\
\text{assume} \quad t \in \mathbb{N}: \#x \downarrow t \leq 1 + \#y \downarrow t \\
\text{commit} \quad m \in T: m\#x = m\#y
\end{array}
\]
Example: System interface specification

A transmission component TMCWA

<table>
<thead>
<tr>
<th>TMCWA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>in</strong> x: T</td>
</tr>
<tr>
<td><strong>out</strong> y: T</td>
</tr>
<tr>
<td>asu(x, y) ⇒ x ~ y</td>
</tr>
</tbody>
</table>

\[
x \sim y \equiv (m \in T: m\#x = m\#y) \\
asu(x, y) \equiv (t \in \mathbb{IN}: \#x\downarrow t \leq 1 + \#y\downarrow t)
\]

We speak of a **contract**
Quality model: functional requirements – conventional view

Main Quality Attributes

Quality in End-Use
- Usability
  - Operability
  - Learnability
- Functional Suitability
  - Functional correctness
  - Functional appropriateness
  - Functional completeness
  - Time behavior
- Reliability
- Security
  - Confidentiality
  - Integrity
  - Economic damage risk
  - Health and safety risk
  - Environmental harm risk
- Safety
- Maintainability
  - Analyzability
  - Modifiability
  - Verifiability
- Releasability
  - Configurability
  - Installability
- Reusability
- Executability
  - CPU Consumption
  - Memory Consumption
  - Co-existence
- Supportability
- Appraisability

Auxiliary Quality Attributes
- Adaptability
- Portability
- Performance
Quality model: functional requirements – novel view

Main Quality Attributes

- Quality
  - Quality in End-Use
    - Usability
      - Operability
      - Learnability
    - Functional Suitability
      - Functional correctness
      - Functional appropriateness
      - Functional completeness
      - Time behavior
    - Reliability
    - Security
      - Confidentiality
      - Integrity
      - Economic damage risk
      - Health and safety risk
      - Environmental harm risk
    - Safety
  - Quality in Development and Evolution
    - Maintainability
      - Analyzability
      - Modifiability
      - Verifiability
    - Releasability
    - Reusability
  - Quality in Operation
    - Executability
    - Supportability
  - Quality in Business
    - Appraisability

Auxiliary Quality Attributes

- Error protection
- Accessibility
- Response time
- Throughput
- Testability
- Reviewability
- CPU Consumption
- Memory Consumption
- Co-existence
- Adaptability
- Portability
- Performance
Classification

- **Functional requirements:** logical and probabilistic interface behavior (including faults):
  - functional features
  - safety
  - reliability
  - ...

- **Architectural requirements:** logical and probabilistic sub-system interface behavior (including faults)
  Quality requirements such as:
  - Performance
  - Security

- **Requirements related to system context**
  - Usability
  - Business - Return on investment
Conclusion

Probability

- Probabilistic system models and specifications are refinements of nondeterministic system models and specifications
- A rich set of so-called “non-functional” properties is captured by probabilistic interface specifications and thus become functional

Time

- Time-aware system models and specifications are refinements of non-time-aware system models and specifications
- For time critical systems so-called “non-functional” timing properties can be captured by time-aware interface specifications and thus become functional

Performance

- There are two concepts of performance:
  - response time (in the case of non-time critical functionality)
  - efficient utilization of resources
- Response time is a functional property captured by time-aware probabilistic interface specifications