Element Characteristics

Definition

Company / Organization: becke.ch

Scope: 0 (language=en;language=default;technology=odt;organization=becke.ch;)

Version: 1.0.0

Author: convention-n-definition--s0-v1@becke.ch

Document Version History

Version	Date	Author	Description
0.9	20.02.2014	Raoul Becke	Initial version
1.0	18.08.2017	Raoul Becke	First Published Version: Slightly reworked introduction, improved: “Element: Modification: State versus Structure & Behavior”, “Element: Instance versus Copy”, “Element: State: (Data) Migration”

Module / Artifact / Component / Work-Product Version History

Version	Date	Author	Requirements	Components Changed
0.9	20.02.2014	Raoul Becke	Establish an element definition that can be used as the basis for further definitions and conventions.	Definitions & Conventions
1.0	18.08.2017	Raoul Becke	Prepared document for publication in the Internet. As basis for [1], [2] and especially the naming convention in the MS Cloud Architecture Foundation.	This document

Table of Contents

1. Introduction..... 6
2. Definition..... 6
3. Element Containment..... 9
3.1. Sets & Set Theory..... 9
3.1.1. Description..... 9
3.1.2. Basic Operations..... 10
3.1.2.1. Unions..... 10
3.1.2.2. Intersections..... 10
3.1.2.3. Complements..... 10
3.1.2.4. Cartesian product..... 10
4. References and glossary..... 11
4.1. References..... 11
4.2. Glossary (terms, abbreviations, acronyms)..... 11

Illustration Index

Index of Tables

Table 1: References..... 11
Table 2: Glossary..... 11

1. Introduction

This document defines the characteristics of an element and is focused but not necessarily restricted to information technology (IT). This document serves as basis for further definitions and conventions – namely: “Scope and Version Convention” see [1] and “Naming Convention” see [2].

Whether element is the correct term or whether it rather should be called application, solution, artifact, module, component, work-product or ... depends on the context. For example in IT an UML element can be a: class, artifact, component, etc.

For clarification every paragraph below has been extended with an IT and a non-IT related sample where appropriate.

2. Definition

Element: Structure (Interface), Behavior: Every element (application, solution, artifact, module, component, work-product, etc.) is declared and defined through its structure and behavior. The structure consists of an external and an internal part. The external part is called the interface and is exposed externally for interaction whereas the internal part can only be accessed internally.
For example a door latching device (a simple one without a keyhole lock construction) consists of the following interface: a door knob (or door lever) and a latch bolt (or just simply latch) and the following behavior: turning the door knob moves the latch bolt. The structure furthermore defines how much degrees the knob can be turned in one or the other direction and how much the latch can move into or out of the device. The internal structure is not obvious and varies from vendor to vendor (USP: Unique Selling Point) but normally consists of some gear wheel attached to the door knob and operating on the latch bolt.
Or for example a sales-opportunity management software component exposes the object fields: “title”, “type”, “closure reason”, “stage”, etc. and implements the following behavior respective business rule: “*before an opportunity can be set to stage “closed” or “offer rejected” a closure reason must be selected”*. The structure furthermore defines that the “title” field is of type string with a maximal length of 50 characters and that the “type” is a drop-down field that can only take the values: “New Business” and “Existing Business – Increase”.

Element: State (Values): Through interaction and behavior the state respective values of an element change over time. The states respective values an element can have are limited by its structure and have a certain type. The type is analogue to the element as well declared and defined through its structure and behavior. The values it exposes define its interface and the operations defined on these values reflect its behavior. There exist 4 different classifications of types respective levels of measurement aka scales of measure: Nominal scale, Ordinal scale, Interval scale and Ratio scale with the following binding base-structure and -behavior – see [3]:

- *Nominal scale: The nominal type differentiates between items or subjects based only on their names or (meta-)categories and other qualitative classifications they belong to:*
  - *Logical / mathematical operations: “=”/”≠”*
  - *Example: Gender: Male, Female*
- *Ordinal scale: The ordinal type allows for rank order (1st, 2nd, 3rd, etc.) by which data can be sorted, but still does not allow for relative degree of difference between them:*
  - *Logical / mathematical operations: “=”/”≠” ; “<”/”>”*
  - *Example: School grade: From “very good” down to “unsatisfactory”*
- *Interval scale: The interval type allows for the degree of difference between items, but not the ratio between them:*
  - *Logical / mathematical operations: “=”/”≠” ; “<”/”>” ; “+”/”–”*
  - *Example: Examples include temperature with the Celsius scale, which has an arbitrarily-defined zero point (the freezing point of a particular substance under particular conditions), date when measured from an arbitrary epoch (such as AD) and direction measured in degrees from true or magnetic north. Ratios are not allowed since 20 °C cannot be said to be "twice as hot" as 10 °C, nor can multiplication/division be carried out between any two dates directly.*
- *Ratio scale: The ratio type takes its name from the fact that measurement is the estimation of the ratio between a magnitude of a continuous quantity and a unit magnitude of the same kind:*
  - *Logical / mathematical operations: “=”/”≠” ; “<”/”>” ; “+”/”–” ; “×”/”÷”*
  - *Example: Examples include mass, length, duration, plane angle, energy and electric charge. Ratios are allowed because having a non-arbitrary zero point makes it meaningful to say, for example, that one object*

has "twice the length" of another (= is "twice as long").

The state change has no bearing on the structure and behavior of the element. There are exceptions where state change has impact on structure and its related behavior: abrasion and self-modifying code. Through intensive usage most elements (not software) start to wear down which has impact on structure and behavior to the point where the element becomes unusable. According to [4]: *In computer science, self-modifying code is code that alters its own instructions while it is executing - usually to reduce the instruction path length and improve performance or simply to reduce otherwise repetitively similar code, thus simplifying maintenance*.
For example a door latching device respective its type (nominal scale) can have 2 different states: “open” and “close”. When turning the door knob in one direction the door changes into open state and when turning the knob in the other direction the door goes into closed state. Actually the underlying type (ration scale) of the knob respective latch is more fine granular and can have any value in terms of degrees respective centimeters between the two final states: “open” and “close”. The structure of the latching device defines how much degrees respective centimeters the knob respective bolt can be turned in one or the other direction respective moved in or out of the device until it reaches the state called open respective closed. Through intensive usage and state changes the structure of the door latching device wears down to the point where it becomes unusable respective broken.
Or for example the JRE (Java Runtime Environment) while executing the java class files will start to inline certain frequently called methods instead of calling them, because method invocation is more expensive i.e. takes more time than code in-lining.

Element: Modification: State versus Structure & Behavior: Modification of State is expected and happens through interaction on the exposed interfaces and related behavior. Modification of Structure and/or Behavior is not expected and can have a Major or Minor impact on related elements interacting with this element. A Major change implies that this element is no longer compatible with some (or all) related elements. A Minor change has no effect on the related elements and can be performed risk-less.
For example when increasing the diameter of the latch bolt of a door latching device then this is a Major change because the device will not fit anymore into the hole of the door case. On the other hand reducing the diameter of the latch bolt is a Minor change because it will still fit into the hole of the door case.
Or for example when removing fields on the sales-opportunity management software then this is a major change because related software that is using this interface cannot operate anymore on sales opportunities the way it used to. On the other hand adding optional (NOT mandatory) fields is a minor change because the related software can still operate on sales opportunities the way it did before and just ignore the newly added optional fields.

Element: Instance versus Copy: An instance and a copy of the same element have identical structure and behavior but potentially a different state. While a copy has the same state as the element it was copied from, the state of an instance starts in its initial state. Sometimes the initial state respective initial values are undefined which is also known as NULL value(s) in IT. Furthermore the difference between an instance and a copy is the origination process. Instances are produced en masse (on the conveyor), all following the same blue-print, whereas copying starts on the final element, analyzing its structure and behavior and copying it bit by bit (and potentially improving it while copying) and while copying reconstructing as well the underlying blue-print. In IT the difference between an instance and a copy is that instances of a program share the same code (structure & behavior) base and from there the code is loaded/instantiated into memory for execution, whereas copies of a program contain as well a copy of the underlying code. Based on this observation to modify the structure & behavior of an element, without altering the original/source, it needs to be copied not instantiated (see next section “Element: Branching”).
There can exist several instances respective copies of an element with identical structure and behavior but different states after some time of interaction.
For example if there are several users at the same time working with the sales-opportunity management software component then normally everyone requires his own instance/copy of the software and these different copies will soon have different states due to user interaction and behavior.

Element: Branching: Copy & Modification: When an element is copied and modified but still serves the same (original) purpose then this is called branching where the different copies of an element with the same point of origin and purpose coexist. Normally the target element maintains a link to the source element it was branched from.

Element: Historization: Element historization is achieved by making a copy of the current element and assigning it a time-stamp or version-number before changing its structure, behavior and/or state.

Element: State: (Data) Migration: Once the structure changes, the element should be historized and (special) transformation rules are required to map the old values into the new structure. The mapping of the old values into the new structure is called (data) migration. In this context we need to differentiate between major and minor changes (see above). If a major change happens then a migration cannot be performed without information loss and therefore the element needs to be historized first, this is not optional, otherwise the information content is lost and cannot be reconstructed.
For example if the address structure changes and the street-name and -number are stored in separate fields instead of one then the information content is still the same but if the street-number is not stored anymore at all then the information content is lost and the element has to be historized before migration because otherwise the

information cannot be reconstructed anymore.

Element: Life-cycle: Build- & Run-Time: The life-cycle of an element starts with its creation respective the creation of its structure and behavior where requirement gathering, design, implementation and packaging takes place. Next the element is deployed centrally and/or in different locations and then starts respective is instantiated in a specific initial state with specific initial values. After its creation respective instantiation the element goes through different modifications of state and sometimes in course of a redesign goes through a modification of structure and behavior and finally the life-cycle ends when the element respective its instances get deleted. Analogue to the creation the modification of structure and behavior is normally as well accompanied by a requirement, design, implementation and packaging phase; followed by a deployment respective upgrade and last but not least migration phase. The life-cycle of an element starting with the creation-, followed by modification and ending with the deletion of structure and behavior is known as “**build-time**” whereas the life-cycle of an instance of an element starting with its instantiation-, followed by modifications of state and ending with the deletion of the instance is known as “**run-time**”.

Element: Composition: Atomic Element: Elements are normally composed of other elements and the way they are composed determines their structure and behavior. Some elements are contained in respective internal part of the composition and accordingly deployed together with the composed element while other elements are located outside respective external to the composition, can be shared between different instances, copies and potentially different compositions and are not necessarily deployed together with the composition. Furthermore some elements are exclusively dedicated for this composition while others can be reused in other compositions. Elements that are exclusively dedicated for this composition have the same life-cycle as the composed element whereas elements reused in other compositions have an own life-cycle driven by the compositions where they are used. The composition ends once we reach the atomic element level. Through composition new elements are created.
For example water is composed of the atoms: hydrogen and oxygen in the combination H_2O.
Or for example a radio consists of internal parts that are exclusively dedicated like for example the chassis which differentiates it from other radios and internal parts that are reusable like for example: transistors, resistors, etc. And consists of external parts that are exclusively dedicated like for example a car mount to mount this specific radio within the car respective, reusable like for example the radio station that provides the audio for all different kind of radios.
Or for example a sales-opportunity management software component is composed of the opportunity object fields plus some picker components: product picker, internal participant picker and external participant picker to assign products, internal- and external-participants to an opportunity. While the opportunity object is exclusively dedicated for and internal part of the sales-opportunity management software, the picker components belong to other software components and are reused as well in different other compositions for example task management software. Some picker components are provided as libraries and accordingly become an internal part of the sales opportunity while others are provided externally as a central service for use in different solutions.

Element: Linking: Static (ID) & Dynamic (Algorithm): Linking of target- to source-elements in a composition or target-elements to users requires that their adjacent interfaces are compatible. Linking can ether be done static, dynamic or mixed. Static linking retrieves the target element based on its unique identifier (ID) whereas dynamic linking looks up the target element applying dedicated search algorithms on ((meta) information related to) the behavior and structure of the element respective on its values. Applying search algorithms on ((meta) information related to) the structure of the element only makes sense for the non adjacent interfaces respective structure. A unique ID is assigned to the element during build-time when the element (structure and behavior) is created and in addition a further unique ID is assigned during run-time to each new instance. The search algorithm is applied during run time and first searches the elements that implement the requested (adjacent) interface(s), then it scans through the ((meta) information related to) behavior and structure of these elements for the best match and last but not least it searches through the different instances of the best matching element for the one whose values best match the search query.
For example a Philipps screw exposes a cross slot interface. To screw or unscrew this item we could ether statically always use the same screwdriver or we could dynamically search for alternatives that implement a cross slot interface and use for example a drill with a Philips screw mount. Next whether we can use a drill which would make our life much easier (depending on the number of screws) or not depends on whether we find a plug socket nearby where we can plug in the plug-interface. And last but not least the values the drill has should match i.e. we only take a drill (instance) whose drill head is in the optimal position that fits with our position towards the screw (but in this case we normally just turn ourselves the drill head so that it fits the positioning).
Or for example using Google we can ether dynamically search for a web page entering the search criteria we are interested in or we can statically bookmark the relevant web page. When searching then we first search all domains (DNS) respective web servers that implement respective expose the HTTP(S) protocol, in there we search all documents that implement the HTML protocol and last but not least we retrieve all pages (“HTML document instances”) that match our search criteria.

Element: ID: Modification, Copying & Branching: A unique ID is assigned to the element respective to each of its instances during creation respective instantiation. Modifications of structure and behavior respective state have no impact on the unique ID, which causes issues in case of major changes because static (ID) linking will fail on

structure respective behavior level. A problem arises when the element is copied, in this case a new unique ID needs to be assigned to the copy respective the copied ID needs to be extended with a copy identifier to make it unique again. In case the copied element still serves the same purpose the copied ID is extended with a copy identifier to show the affiliation to the same branch otherwise a new ID is assigned.

Element: Containment: Container elements have an own structure and behavior and are not influenced by the elements they contain. Compared to composition a container respective carrier element can exist without the elements it contains respective carries and these elements are mostly not or only loosely coupled.
For example in salt-water the water contains respective carries the salt and they can be separated quite easily through evaporation and exist on their own, in this case already the combined name “salt-water” suggests that this is a containment and not a composition. On the other hand when looking at water (element composition) then it has an own name, only exists the way it is composed of hydrogen and oxygen in the combination H_2O and it is difficult to decompose.
Or for example a ZIP is, besides its size, not influenced by the elements it contains and has its own structure and behavior.

Often container elements are used to collect and categorize elements with similar structure and behavior. In this case they can (“calculate” and) “expose” the common structure and behavior of the contained elements. The container themselves tend to overlap due to the fact that an element can belong to different structure and behavior categories.
For example a box of nails can contain further boxes containing the different kind of nails: wire-nails, spikes, pins, etc.. Furthermore most nails are made of metal but some are made of wood which puts most of them into the category metalware and some of them into the category woodware, which makes the nail container overlap the woodware and metalware container while the metalware and woodware container stay disjoint. Further nail categorizations are: fastener and sharp pointed object. Not all fasteners are sharp pointed objects and vice-versa which puts the nail containers into the intersection area of the fastener and sharp pointed object container.
Or for example the sales-opportunity management software component belongs to the sub-domain XSEL (Cross Selling) which itself belongs to the domain CRM (Customer Relationship Management).

Element: Composition & Containment: Normally the elements used in a composition are as well categorized and their categories might be different from or redundant to the one of the composition. During build time while these elements are still loose they are put into a “composition” container which contains a container for internal and several containers for the different exclusively dedicated external parts. These containers might then contain further nested category containers to host the different elements used in the composition. Once the build is finished and packaged the internal respective external container(s) become internal respective external composition(s) and are then deployed within the “composition” container or separately.
For example when building any kind of equipment the internal part container and its sub-category containers hold the equipment pieces that have to be assembled and the exclusively dedicated external container normally holds among other things some manuals in different languages. Instead of calling the containers respective the elements they become “internal” respective “external”; rather give them clear category names like “equipment” and “documentation” or even better composed names consisting of composition name plus category name. These containers respective elements can then be deployed respective shipped together but stored at home separately.

3. Element Containment

Element containment is described best using the set theory.

3.1. Sets & Set Theory

Definition see [5]: “*A set is a gathering together into a whole of definite, distinct objects of our perception [Anschauung] or of our thought – which are called elements of the set.*”.

3.1.1. Description

(See [5]) “*There are two ways of describing, or specifying the members of, a set. One way is by **intensional definition, using a rule or semantic description**:*

> *A is the set whose members are the first four positive integers.*
>
> *B is the set of colors of the French flag.*

*The second way is by **extension** – that is, **listing each member of the set**. An **extensional definition** is denoted by enclosing the list of members in curly brackets:*

> *C = {4, 2, 1, 3}*
>
> *D = {blue, white, red}.*

Every element of a set must be unique; no two members may be identical.“

Typical definitions in computer science are:

- Intensional:
  - **EBNF**
  - **Regular Expressions**
  - **SQL**
- Extensional:
  - **Bit Battery**
  - **Set<Object> in Programming**

3.1.2. Basic Operations

See [5]: *There are several fundamental operations for constructing new sets from given sets.*

3.1.2.1. Unions

*Two sets can be "added" together. The **union** of A and B, denoted by **A ∪ B**, is the set of all things that are members of either A or B.*

3.1.2.2. Intersections

*A new set can also be constructed by determining which members two sets have "in common". The **intersection** of A and B, denoted by **A ∩ B**, is the set of all things that are members of both A and B. If A ∩ B = ∅, then A and B are said to be **disjoint**.*

3.1.2.3. Complements

*Two sets can also be "subtracted". The **relative complement** of B in A (also called the set-theoretic difference of A and B), denoted by **A \ B (or A − B)**, is the set of all elements that are members of A but not members of B. Note that it is valid to "subtract" members of a set that are not in the set, such as removing the element green from the set {1, 2, 3}; doing so has no effect.*

3.1.2.4. Cartesian product

*A new set can be constructed by associating every element of one set with every element of another set. The **Cartesian product** of two sets A and B, denoted by **A × B** is the set of all ordered pairs (a, b) such that a is a member of A and b is a member of B.*

4. References and glossary

4.1. References

Reference	Location	Remarks
[1]	http://becke.ch/data/becke-ch--scope-and-version-convention--s0-v1/document/	Scope and Version convention
[2]	http://becke.ch/data/becke-ch--naming-convention--s0-v1/document/	Naming convention
[3]	http://en.wikipedia.org/wiki/Level_of_measurement Or cached: http://becke.ch/data/becke-ch--element-characteristics-definition--s0-v1/document/cache/Level%20of%20measurement%20-%20Wikipedia.html	Levels of measurement or scales of measure
[4]	http://en.wikipedia.org/wiki/Self-modifying_code Or cached: http://becke.ch/data/becke-ch--element-characteristics-definition--s0-v1/document/cache/Set%20(mathematics)%20-%20Wikipedia.html	Self-modifying code
[5]	http://en.wikipedia.org/wiki/Set_%28mathematics%29 Or cached: http://becke.ch/data/becke-ch--element-characteristics-definition--s0-v1/document/cache/Set%20(mathematics)%20-%20Wikipedia.html	Set definition (mathematics)

Table 1: References

4.2. Glossary (terms, abbreviations, acronyms)

Terms / Abbreviations / Acronyms	Description

Table 2: Glossary