ECSA2021 - Coding Book

Mohamed Soliman

May 2021

1 Coding book

Here we will write a list for general AK concepts, which provide a base for the coding book

AK concepts from Zimmermann et al.[Zim+09], Soliman et al. [SRZ15; SGR17], Jansen et al. [JB05], and Tang et al. [TV09]

1.1 Simple AK concepts

Simple AK concepts appear in issues as single terms (not statements or clauses, see below for examples). We do not annotate these simple AK concepts, because considered on their own and in isolation they do not present a meaningful and reusable useful AK concept. Instead, we use simple AK concepts as indicators for composite AK concepts that we annotate. Thus, it is important to recognize simple AK concepts in order to annotate the composite AK concepts.

- *Architectural components*: Components are basic building blocks in the architecture of software systems. Components can be any entity (e.g. a class, module, package, subsystem, external software or code, etc.). This AK concept is commonly used in other composite AK concepts (e.g. existing system and architectural configuration as discussed below). Components are usually referred to by using single words that indicate the name or function of a component and, based on our pilot study, appear in the following forms:

    1. *Logical components*: e.g. "optimizer", "engine", "resource manager".
    2. *Physical components*: e.g. "server" or "directory".
    3. *Class names*: e.g. class TajoMaster in the Tajo system[1]
    4. *Component types*: e.g. text that refers to "... these classes...", "... these tables..." or "... these interfaces and packages..."
    5. *Whole system*: e.g. Tajo, a big data relational and distributed data warehouse system for Apache Hadoop.

[1]https://tajo.apache.org

- *Quality attribute terms*, such as “performance”, “maintainability”, “availability”, etc.
- *Technology solutions* [SRZ15], such as “Json”, “Lucene” or “protobuffer”.
- *Architectural patterns and tactics terms*, such as “visitor”, “ping echo” or “caching”.

1.2 Composite AK concepts

As mentioned before, composite AK concepts are made (or composed) of simple AK concepts. Composite AK concepts can be represented as a clause or a phrase or a sentence or multiple sentences. We differentiate the following composite AK concepts:

1.2.1 Design issue

The problem, which the architectural design decision wants to solve. [JB05; Zim+09]

- *Motivation of design issue*: This concept captures “the causes of this problem” [JB05]. The reason for resolving a design issue is different and can come in different forms:
    1. *Support developers*: To encourage developers to extend the system (e.g. “*It may encourage contributors to implement various join enumeration ways, and users could choose the best join ordering algorithm for their purpose*” in TAJO-99) , or to facilitate testing the system. (e.g., “*Also, this visitor will be helpful for making unit tests*” in TAJO-121).
    2. *Support sales*: To attract more users, e.g., “Many users use Hive to analysis big data. If Hive user can use HiveQL, they can easily use Tajo.” (TAJO-101)
    3. *Adaptive change*: For example to cope with technology trends. For example “*Since unstructured data are generally processed in the Hadoop world, I think that Tajo also has a need to provide the processing of unstructured data as well as the relational data*” (TAJO-283)
- *User requirements*: User requirements describe the new user requirements to be introduced in a software system and can appear in the following forms:
    1. *User input*: This describes the desired user input for the software. For example, “*In some case, a user wants to determine explicitly some parts of the join orders*” (TAJO-24 attachment)

2. *System output*: This describes the output of the software, which will be provided to the user. For example, "*For user convenience, Tajo needs to show users function information, including signature, parameters, results, descriptions, and examples*" (TAJO-408)

- *Quality attributes (QA) requirements*: This concept describes quality attributes (e.g. performance or security), which are considered important to fulfill for a particular system. It comes in the following forms:

  1. *Explicitly*: Concept is described using well-known QA terms like "extensibility" or "performance" (see Section 1.1). The terms could come in separate bullet points or in a complete sentence. For example "*This improves the code readability and maintainability*" (TAJO-121).
  2. *System behavior quality adjective*: This describes the quality of certain system behaviors using adjectives, which refer to a certain quality attribute (rather than expressing quality attributes or names of quality attributes explicitly). For example "*Tajo workers...will provide low-latency responses for query tasks*" (TAJO-88) and "*For the sake of efficient join order enumeration,...*" (TAJO-229) could point to performance.
  3. *Component quality adjective*: It can also come as an adjective for a certain architectural component. The adjectives could be derived from QA terms, For example "*This change will make Tajo more scalable and remove SPOF of master*" (TAJO-91) could point to scalability of the master component. Moreover, the adjective might not be derived from QA terms but implicitly refer to certain QA, For example "*It would like to great to provide a generic and pluggable interface for join enumeration algorithms*" could point to maintainability of enumeration algorithms (TAJO-99)

- *Existing system*: This concept explains part of the architecture of an existing system and its issues [SRZ15]. This can come in two main forms:

  – *Existing system architecture description*: Describes (parts of) an architecture of an existing software system, this can include descriptions of existing components, their behavior, and their dependencies. This also includes dependencies on external systems. For example "*In the current implementation, the logical plan is serialized into a JSON object and sent to each worker*" (TAJO-269)

  – *Existing system issue*: Describes possible quality issues in a software system [SRZ15]. This can be divided into two sub-categories [Avg+16]:

    1. *Technical debt*: Quality issues, which impact the maintainability and evolvability of a system, such as architectural or requirements debt items [Avg+16]. For example "*The current query*

optimizer is too simple and is not extensible. In addition, it does not support scalar and table subqueries" (TAJO-24).

  2. Run-time quality issues: These are issues, which are visible to users, such as performance issues and bugs, for example "*the transmission of JSON object incurs the high overhead due to its large size*" (TAJO-269)

- Contextual constraints: such as dependencies on external systems, which cannot be removed and limit the design decisions. (We have less instances from this). For example "*In order to keep Hive compatibility, we need to add Hive partition type that does not exists in existing DBMS systems*" (TAJO-283)

1.2.2 Architectural solutions

Architectural solutions are potential solution alternatives for the design issue [JB05; Zim+09]. AK regarding architectural solutions could come from two sources:

1. *Explicitly proposed architectural solutions*: These are solutions proposed directly by software engineers within an issue. For example "*in my humble opinion we could write several classes implementing the rewrite engine interface, e.g. one for the outer join transformation, another for selection pushdown*" (TAJO-96)

2. Other system architectural solutions: These are solutions, which are already implemented in other software systems; software engineers re-use the AK from other systems to design a new system, for example "*So facebook guys create table partitioning, and later partition pruning to avoid scanning the whole table*" (TAJO-283)

Architectural solutions come in different types, which are explained in the following. An architectural solution could be either explicitly proposed or inspired from other systems:

- *Architectural component behavior and structure*: This describes the behavior of an architecture component. It gives an overview about the type of implemented logic and complexity [SRZ15]. This comes in the following forms:

  1. *Approach*: This describes briefly the main approach (e.g. algorithm or logical concept) on which the behavior of the component is based. For example "*the optimizer will be based on some cost model*" (TAJO-24 attachment).

  2. *Sub-components*: This describes the sub-components (e.g. interfaces or data structures), which implement the component's behavior, however without specifying the dependencies between the components, e.g. "*This optimizer will provide the interfaces for join enumeration algorithms and rewrite rules*" (TAJO-24 attachment).

3. *Component technologies*: This describes the technologies used within this component. For example "*Change TaskRequestProto to use protobuf-based serialization instead JSON-based serialization*" (TAJO-269).
4. *Component inputs and outputs*: This describes the input expected by a component and its output. For example "*It will be an interface or an abstract class to take a list of tables, a join tree, or a mixed of them. It will result in the best-ordered join tree*" (TAJO-24 attachment)

The difference between user requirements and component behavior that user requirements are focused on the user (input and output) while component behavior focuses on a certain component with no interaction with or visibility to the user.

Quality attributes should not be confused with component behavior. A component behavior describes the functionality or sub-components of a component. However, a quality attribute provides a quality attribute for a certain component without mentioning the behavior of the component itself.

Some uncertainties: if there is a description of the behavior but the component term is not their. Would this be component behavior? for example query optimization will do .... this is different than query optimizer, which refer to a specific component.

If there is no concrete component term but there is a term to refer to the whole system. For example, this patch will contain...Tajo should ....We will support ...

- *Architectural design configuration*: This concept describes the relationships and dependencies of components.

  This could involve adding, removing or refactoring components and their dependencies. A textual segment capturing this concept must contain more than an architectural component and one or more dependencies between components. It comes in the following forms:

  1. *Static dependencies*: They describe dependencies between components independent of their sequence, for example "*TajoMaster launches QueryMaster on NodeManager*" (TAJO-91).
  2. *Dynamic dependencies*: They describe the sequence of interactions between components, for example "*YARN mode 1. TajoClient request query to TajoMaster. 2. YarnRMClient request QueryMaster(YARN Application Master) 3. YARN launches QueryMaster...*" (TAJO-88 attachment)

  The static and dynamic dependencies between components come in two forms:

1. *Using connector verbs*, such as “access”, “share”, “obtain”, and “use”. For example, “*This interface should be able to access Tajo catalog in order to obtain statistics information of joined tables*” (TAJO-24 attachment)
2. *Explicit dependency type* Using explicit software design relationships defined in modeling or programming languages, such as “extends”, “implements”, etc. For example, “*in my humble opinion we could write several classes implementing the rewrite engine interface, e.g. one for the outer join transformation, another for selection pushdown*” (TAJO-96)

The difference between component behavior and architectural configuration is that component behavior focuses on a single component, with an explanation of what the component does. A sentence referring to component behavior should not mention multiple components which communicate with each other. The component behavior could mention sub-components, but without explaining the relationships between theses sub-components.

- Architectural tactics: Tactics should refer to using certain solutions to improve a specific quality attribute. For example, “*we will support the standby mode for low latency*” (TAJO-88)

1.2.3 Design decision rationale

This AK concept captures the reason for selecting a particular architectural solution from a list of alternatives. [TV09]

- Architectural solution benefits and drawbacks: This concept describes the strengths and weaknesses for certain alternatives. This can be based on generic judgment or based on testing and benchmarks [SRZ15]. It comes in several forms:

1. *Explicitly*: Solution benefits and drawbacks can be explicitly described in a statement by terms like “advantages”, ”disadvantages”, ”limitations’, etc. For example ”*The most advantage of supporting the column partition is the compatibility with Hive*” (TAJO-283)
2. *Using adjectives*: The adjective could refer to to the whole solution or a specific component or feature of the solution. Adjectives could be generic like ”good”, ”nice”, ”simple”, ”only”, ”ugly”, etc. For example, “*This is quite ugly and mess*” (TAJO-283). It could also be more related to special quality attribute, e.g. “optimize”, ”accelerate”, ”boost”, ”fast”. For example ”*That approach will boost Tajo’s performance*” (TAJO-472).
3. *Implicitly using quality measurement*: Expressing special quality measurements, for example performance measurements. For example, “*We can do group by aggregations on billions of rows with only a few milliseconds*” (TAJO-283).

Solution drawbacks can be expressed especially using certain special forms:

1. *Negation*: Solution drawbacks can be presented by negation or negative adjectives regarding a quality attribute. For example, we cannot do something, or it does not support a certain feature, or negation of adjectives e.g. "not the best". For example, "*Yarn is not the best choice for low latency query executions*" (TAJO-88)
2. *Problems or issues*: Speaking about problems or issues which result as a consequence from using a particular solution. For example, "*the column partition can incur a problem when there are a large number of partitions*" (TAJO-283). The problem can be described in different ways, such as a solution that needs more memory or creates too much data. For example, "*It looks that the deserialization process of Protocol Bufers requires CPU processing as much as that of Json*" (TAJO-269)

- *Assumptions*: This concept captures facts which are assumed without proof when deciding on an architectural solution [Yan+17]. Assumptions are presented in two forms:

  1. *Explicitly*: Using explicit references to assumptions, e.g., the word "assumption", "assume" or synonyms. For example, "*Now, we assume that one hdfs directory is one partition*" (TAJO-283)
  2. *Using uncertainty-related terms*: Examples of uncertainty-related terms include "I think", "it might'". For example, "*The reason why I think tajo's partition is like hive's bucket is that both are designed to distribute their row according to one column's value of this row*" (TAJO-283).

  Assumptions can co-occur with other AK concepts. In other words, software engineers could assume AK about architectural solutions or rationale or the design issue.

- *Trade-offs*: Trade-offs describe balanced analysis of what is an appropriate option after prioritizing and weighing different design options [TV09]. We have less instances for this.

- *Risks*: Risks capture considerations about the uncertainties or certainties of a design option [TV09]. We have less instances for this.

References

[JB05] Anton Jansen and Jan Bosch. "Software Architecture as a Set of Architectural Design Decisions". In: *WICSA*. 2005, pp. 109–120. ISBN: 0-7695-2548-2. DOI: http://dx.doi.org/10.1109/WICSA.2005.61.

[TV09] Antony Tang and Hans Van Vliet. “Software architecture design reasoning”. In: *Software Architecture Knowledge Management: Theory and Practice*. Springer Berlin Heidelberg, 2009, pp. 155–174. ISBN: 9783642023736. DOI: `10.1007/978-3-642-02374-3{\_}9`. URL: `https://link.springer.com/chapter/10.1007/978-3-642-02374-3_9`.

[Zim+09] Olaf Zimmermann et al. “Managing architectural decision models with dependency relations, integrity constraints, and production rules”. In: *Journal of Systems and Software* 82.8 (2009), pp. 1249–1267. DOI: `https://doi.org/10.1016/j.jss.2009.01.039`.

[SRZ15] M Soliman, M Riebisch, and U Zdun. “Enriching Architecture Knowledge with Technology Design Decisions”. In: *WICSA*. May 2015, pp. 135–144. DOI: `10.1109/WICSA.2015.14`.

[Avg+16] Paris Avgeriou et al. *Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)*. Tech. rep. 4. 2016, pp. 110–138. DOI: `10.4230/DagRep.6.4.110`. URL: `http://drops.dagstuhl.de/opus/volltexte/2016/6693%20http://www.dagstuhl.de/16162`.

[SGR17] M Soliman, M Galster, and M Riebisch. “Developing an Ontology for Architecture Knowledge from Developer Communities”. In: *IEEE/IFIP ICSA 2017*. Apr. 2017, pp. 89–92. DOI: `https://doi.org/10.1109/ICSA.2017.31`.

[Yan+17] Chen Yang et al. “An industrial case study on an architectural assumption documentation framework”. In: *Journal of Systems and Software* 134 (Dec. 2017), pp. 190–210. ISSN: 01641212. DOI: `10.1016/j.jss.2017.09.007`.