Grounded Architectures: Using Grounded Theory for the Design of Software Architectures

Habtom Kahsay Gidey
Technische Universität München, Germany
habtom.gidey@tum.de

Diego Marmsoler
Technische Universität München, Germany
diego.marmsoler@tum.de

Jonas Eckhardt
Technische Universität München, Germany
eckharjo@in.tum.de

***Abstract—Context*: Designing new architectures is a challenging task. A common and also effective approach for this task is to apply architectural design experience.**

***Problem*: If, however, architectural design experience is not available, two major problems arise: (i) how can we identify architecturally significant requirements (ASRs) and (ii) how can we identify architectural design decisions (ADDs) which address those ASRs?**

***Approach*: To address these problems, we propose an approach to systematically elicit ASRs and identify ADDs based on grounded theory (GT). By using GT, our approach provides transparency and fosters objectivity: ASRs as well as ADDs are elicited by qualitative data analysis and each ADD is motivated by corresponding ASRs. While objectivity addresses the correctness of the identified ASRs and ADDs, transparency allows for assessing ADDs with respect to the corresponding ASRs.**

***Evaluation*: We evaluate our approach by a case study in which we apply it to develop an architecture in the context of the technology trend "appification".**

I. Introduction

The process of specifying and describing software architecture (SA) involves complex assumptions, argumentation, and taking important decisions [34]. For well-known domains, a common and effective way to design an architecture is by applying existing architectural design experience [33]. Designing a new web-shop, for example, is easily done by applying the large amount of existing knowledge in this domain. Difficulties arise, however, when it comes to the design of architectures in a problem domain for which no or only little experience is available yet.

When designing architectures in new domains, two major problems arise: First, we need to identify architecturally significant requirements (ASRs) and second, we need to identify architectural design decisions (ADDs) which address those requirements. Identifying ASRs involves the identification of key requirements determining the major architectural design decisions. This includes, inter alia, identifying the most important quality requirements impacting the architecture.

Identifying architectural design decisions (ADDs), on the other hand, involves the identification of architectural design options and choosing those which best address the identified requirements. In general, identifying requirements and design decisions is not a problem per se, but the problem arises when it comes to identifying requirements significant to the architecture as well as the ADDs which effectively address the identified ASRs.

To address these problems, we propose an approach to systematically elicit ASRs and identify ADDs based on grounded theory (GT) [4], [5], [11]: We consider the development of a new architecture as the development of a new *theory* which consists of ASRs and corresponding ADDs. Then we apply GT as a systematic way of building this theory.

Our approach ensures *transparency* and fosters *objectivity* in the design process. Objectivity is achieved by the inherent nature of the GT method, which systematically grounds understanding on the data source. Identified ASRs, as well as the corresponding ADDs, are founded on a firm base of qualitative data. Transparency, on the other hand, is achieved by the proposed structure of the theory. Each identified ADD has to be related to a corresponding ASR by the theory. Thus, each decision (ADD) is motivated by corresponding requirements (ASRs).

To evaluate our approach, we conducted a case study in the context of the technology trend "appification". In particular, we identified ASRs and corresponding ADDs relevant to appification by means of expert interviews. The relationship of ADDs and ASRs was further evaluated by reviewing relevant literature.

In summary we identified a total of 26 significant ASRs, grouped into 8 categories. For each ASR, we identified a corresponding ADD which we evaluated by reviewing existing literature. Based on the identified ADDs, we propose a Reference Architecture (RA) for the appification technology trend.

The remainder of the paper is organized as follows: In Sect. II, we provide our approach: a brief discussion of GT and how it can be applied for the design of new architectures. In Sect. III we apply the approach to design an architecture for the appification technology trend. Next, in Sect. IV we provide related work. Finally, we conclude our work with a brief discussion in Sect. V and a conclusion in in Sect. VI.

II. Approach

A. Grounded Theory

GT is a method used to conceptualize and abstract raw data into a theory [4], [11], [5]. It explores and continuously inspects data making sure that the derived theories remain grounded in reality. As shown in Figure 1, GT follows an inductive approach [11]. It conceptualizes theories based on abstraction outcomes as *concepts*, *categories* and *core categories*. GT applies three coding procedures; open, axial and selective coding. Coding is the process of assigning labels to decomposed parts of the data. Coding portrays cases of observations and facilitates the classification and categorization [28]. Figure 2 provides a running example from a text extract. It shows the process and outcomes, taking a snippet example of the analysis followed to define appification. Irrespective of the procedures, codes are classified as *substantive* or *theoretical* [4]. It starts with open coding, which results *substantive codes* and ends with axial and selective coding, which results *theoretical codes*.

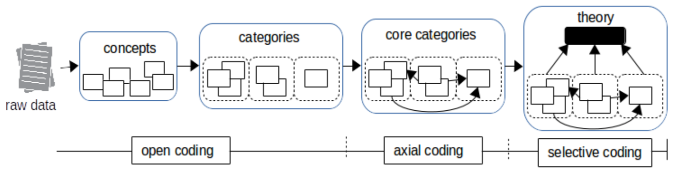

Figure 1. Levels of Abstraction in Grounded Theory. *(Adapted from [35])*

Open coding is used to investigate, decompose and categorize the raw data into *concepts* and *categories* [11]. To code, the researcher identifies essential words or phrases and assigns them with labels. This labels or initial codes, usually in groups, make *concepts*. Concepts help on capturing and collecting the main *objects, instances, incidents and actions* on the data. In Figure 2, a researcher, for instance, can label the phrase appearing in the text as "no longer monolithic large applications but smaller applications ..." into "non-monolithic". Next, the collections of concepts associated are grouped to form categories [11]. *Categories* are further abstractions of the codes with a larger group of similar and pertaining *concepts*. Figure 2, shows example categories, such as *app cooperation, and separate flat functionality*.

Axial coding is used to identify dependencies and relate the categories and subcategories to form the core categories. *Core Categories* are further abstractions of broader groups of related and similar categories. Developed core categories are compared iteratively with new data, resulting refined categories and new dependencies. The example, in Figure 2, shows an abstracted view of dependencies and relationships between the core categories.

Selective coding is the final stage where the core categories are unified and described with broader dimension to form a theory. *Theories* are general conceptualizations of carefully developed and related *core categories* with depth and attributed properties. An example in software engineering (SE) can be any theory fragment such as architectural patterns, styles, descriptions, and formalizations [31]. Furthermore, in SA, an example of a theory could also be an abstraction of specified SA as a holistic model, such as games application RA, robot operating systems architecture or other architectural specification in a domain. In Figure 2, a definition of a SA context appfication is pointed out as an example. To form the definition, see Section III, core categories are extensively described, and deeply investigated to widen the dimension of the conceptualized phenomena, appification.

In GT, data collection and data analysis run in parallel. The analysis starts with the first raw data collected, which in turn guides the next collection or sampling iteratively. Theoretical samples are re-sampled and continuously refined until saturation. The data comes from various sources. In SE, these could be documentation, guidelines, mailing list, system specifications, architectural or design artifacts, field observations, interviews, surveys or even source codes [35].

Interestingly, as illustrated in Figure 3, we observe similarities between GT and SA abstractions. Similarly, others have also observed potentials of GT to systems engineering [13]. The figure shows the conceptualizations of GT mapped with their equivalent examples related to SAs. GT terms are mapped to their close resemblance in SA. For instance, open coding can yield the abstract structural elements of a system architecture such as classes, components, and compositions. An example could be to decompose data source looking for indicators of *structural*, *relational* and *behavioral* elements of a software system. These can, for instance, be further refined into concrete architectural elements such as classes, actors, components, packages or services. During axial coding, for example, the researcher can also look in the data for indicators of connectors, causal or structural relations and this can be represented as *"is a", "has", "extends"* and so on. This connectors could also be, for instance, relations and coupling between classes, components or subsystems.

B. Using GT to specify Software Architectures

In the following, we describe the use of GT to design SAs. We propose a five-step systematic approach. Figure 4 illustrates the main steps, the inputs or data sources, applied GT procedures, and the outcomes. We discuss the purpose, the input, the process, and the output of each particular step. The first step starts by defining the context. The next two steps focus on the elicitation and identification of ASRs and ADDs. In the fourth step, we give emphasis to Glaserian GT ideas such as delayed literature reviews [15], [25]. As the final step, summarized ADDs are compiled into AK baseline to specify the SA. Although the underlying principles of GT maintain similarities, there exist several versions of GT procedures. While this study uses general GT concepts as introduced by Glaser and Strauss [4], the specific processes are set to follow the procedures and techniques by Strauss and Corbin [11]. However, the execution is followed in a flexible manner when a procedure is less applicable to our adapted approach.

a) Step 1: Defining the Context: The purpose in this step is to define the SA context and determine its scope. An example of a SA context could be a specific software system to be developed, a product family, or an RA for a new technology stack.

Input: The data source can be from interviews and/or other artifacts. It can be complemented by minor reviews of publications related to the context. The minor review helps to increase theoretical sensitivity and effective sampling for continued data collection.

Process: The first activity of the process is to sample and collect preferred data sources. The analysis should start with

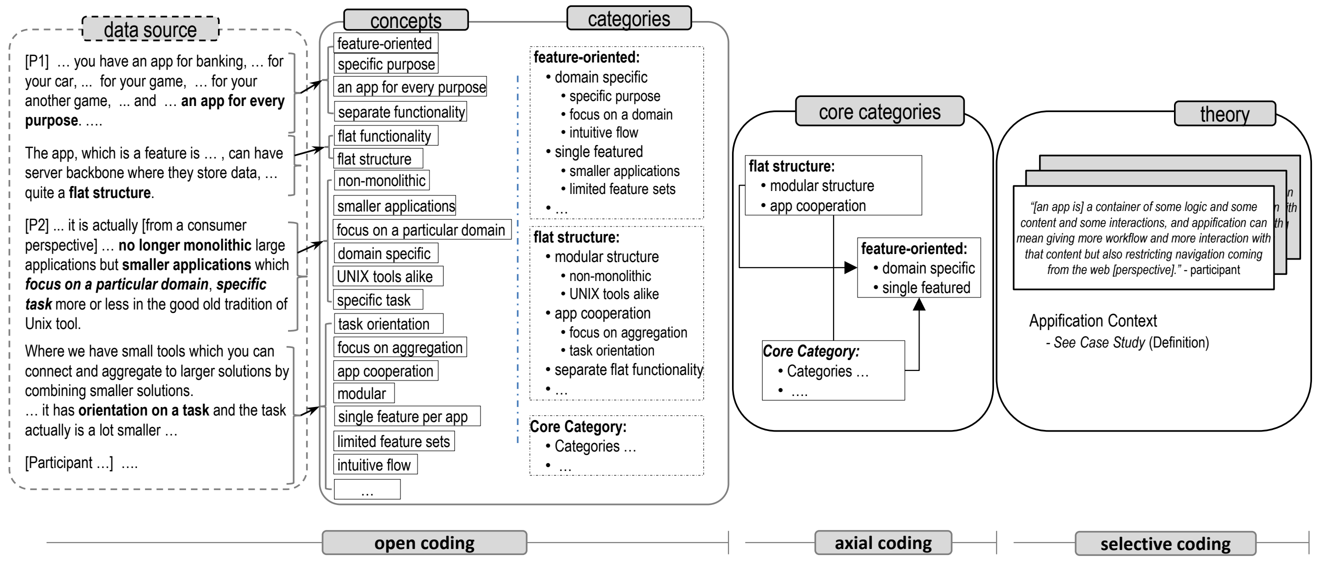
data source
[P1] … you have an app for banking, … for your car, … for your game, … for your another game, … and … an app for every purpose. ….
The app, which is a feature is … , can have server backbone where they store data, … quite a flat structure.
[P2] … it is actually [from a consumer perspective] … no longer monolithic large applications but smaller applications which focus on a particular domain, specific task more or less in the good old tradition of Unix tool.
Where we have small tools which you can connect and aggregate to larger solutions by combining smaller solutions.
… it has orientation on a task and the task actually is a lot smaller …
[Participant …] ….
concepts
feature-oriented
specific purpose
an app for every purpose
separate functionality
flat functionality
flat structure
non-monolithic
smaller applications
focus on a particular domain
domain specific
UNIX tools alike
specific task
task orientation
focus on aggregation
app cooperation
modular
single feature per app
limited feature sets
intuitive flow
…
categories
feature-oriented:
• domain specific
• specific purpose
• focus on a domain
• intuitive flow
• single featured
• smaller applications
• limited feature sets
• …
flat structure:
• modular structure
• non-monolithic
• UNIX tools alike
• app cooperation
• focus on aggregation
• task orientation
• separate flat functionality
• …
Core Category:
• Categories …
• …
core categories
flat structure:
• modular structure
• app cooperation
feature-oriented:
• domain specific
• single featured
Core Category:
• Categories …
• ….
theory
"[an app is] a container of some logic and some content and some interactions, and application can mean giving more workflow and more interaction with that content but also restricting navigation coming from the web [perspective]." - participant
Application Context
- See Case Study (Definition)
open coding
axial coding
selective coding

Figure 2. Coding Example in Grounded Theory.

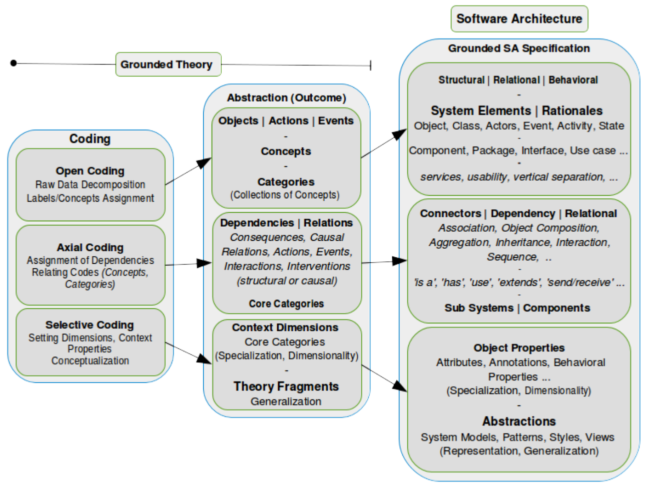
Grounded Theory
Software Architecture
Coding
Open Coding
Raw Data Decomposition
Labels/Concepts Assignment
Axial Coding
Assignment of Dependencies
Relating Codes (Concepts, Categories)
Selective Coding
Setting Dimensions, Context Properties
Conceptualization
Abstraction (Outcome)
Objects | Actions | Events
Concepts
Categories
(Collections of Concepts)
Dependencies | Relations
Consequences, Causal Relations, Actions, Events, Interactions, Interventions
(structural or causal)
Core Categories
Context Dimensions
Core Categories
(Specialization, Dimensionality)
Theory Fragments
Generalization
Grounded SA Specification
Structural | Relational | Behavioral
System Elements | Rationales
Object, Class, Actors, Event, Activity, State
Component, Package, Interface, Use case ...
services, usability, vertical separation, ...
Connectors | Dependency | Relational
Association, Object Composition, Aggregation, Inheritance, Interaction, Sequence, ..
'is a', 'has', 'use', 'extends', 'send/receive' ...
Sub Systems | Components
Object Properties
Attributes, Annotations, Behavioral Properties ...
(Specialization, Dimensionality)
Abstractions
System Models, Patterns, Styles, Views
(Representation, Generalization)

Figure 3. Mapping of GT Abstraction outcomes with SA Specification.

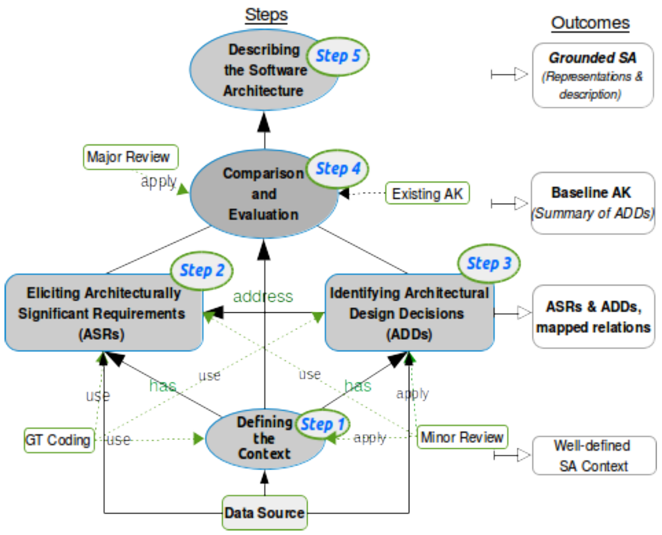
Steps
Outcomes
Describing the Software Architecture
Step 5
Grounded SA
(Representations & description)
Major Review
apply
Comparison and Evaluation
Step 4
Existing AK
Baseline AK
(Summary of ADDs)
Eliciting Architecturally Significant Requirements (ASRs)
Step 2
address
Identifying Architectural Design Decisions (ADDs)
Step 3
ASRs & ADDs, mapped relations
use
has
use
use
has
apply
GT Coding
use
Defining the Context
Step 1
apply
Minor Review
Well-defined SA Context
Data Source

Figure 4. A Systematic Approach to specifying Grounded Architectures

the first data collected. To this end, the three GT coding procedures are applied to build concepts, categories, and core categories related to this step. Assuming the data source is an interview, the data can be broken down based on the interview questions and analyzed separately. On the other hand, if concepts, pertaining the context of the system, exist mixed all over the data, several analysis iterations can be performed with separate abstraction goals. First, using open coding the concepts related to the definition of the context should be examined and coded. Next, concepts can be classified into related categories and core categories with axial and selective coding. Subsequently, the definition and scope of the architecture context should be formed based on the developed core categories. However, to maintain consistency and polish the context definition it can be supported and refined with a minor literature review.

Output: The outcomes at this step are selected core categories characterizing the SA context and scope. Additionally, these theory fragments are additional inputs for steps two and three. At this level, the theory parts defining the SA context are ready for validation and refinement, which is performed in step four. These conceptual pieces are building blocks to construct further generalizations of the SA specification. As a result, at this stage, the context is precisely defined. The software system context is clear, and the scope is determined.

b) Step 2: Eliciting ASRs: The purpose of this step is to elicit the ASRs. ASRs are the software requirements that represent functionality, elements, and interactions of the system determining a systems architecture [9].

Input: Similarly, the data source is extracted from interviews, specifications, or any related document.

Process: Similar to the first step, all of the GT coding procedures are applied. During open coding, the intention of the conceptualization should be directed to look for instances and phenomena discussing the requirements significant to the architecture. The data source can be organized into sections concerned to ASRs to facilitate separate coding and analysis. Following that, categorized concepts should be classified into related categories or core categories with axial and selective coding. Similar to the first, this step can be supported with a minor literature review to increase sensitivity to the concepts coming from the data.

Output: The outcomes at this step are selected core categories of ASRs. These are important motivations and inputs to identify ADDs in step three. The elicited ASRs as conceptual pieces are ready for validation and refinement, which is executed in step four. As a result, at this stage, ASRs are identified, related and described.

c) Step 3: Identifying Architectural Design Decisions: The purpose of this step is to distinguish the ADDs addressing elicited ASRs. ADDs are the rules and constraints that shape architectural instances of a software [22]. Usually, during design, architects consider choices of distilled reusable architectural decision models or established sets of design decisions which fit a problem scenario they encounter [30].

Input: Similarly, the data source for identifying ADDs is extracted from interviews, design rationale records, architectural and design artifacts or any document discussing design decisions. As in the previous steps, a minor review can be performed to increase sensitivity to the concepts in the data.

Process: Similar to the first two steps, all of the GT coding procedures are applied to build the abstractions in the same way. However, during the open coding, unlike the previous steps the researcher or architect must look for rationales, decisions and actions proposed addressing the elicited ASRs. On the other hand, similar guidelines can be followed to organize, code the raw data and reach the desired abstractions of ADDs. Besides, at this stage, a clear mapping of the ASRs and their associated ADDs should be provided. An example of the mapping of ASRs and ADDs is provided from a case study in Section III in Table I.

Output: The outcomes at this step are selected core categories of ADDs, their relations, and mappings to the addressed ASRs. The identified ADDs are ready for validation and refinement. As a result, at this stage, ADDs addressing elicited ASRs are identified and described. Furthermore, mappings of ASRs and their associated ADDs are documented.

d) Step 4: Comparison and Evaluation: The purpose of this step is to validate and refine resulting outcomes from the previous steps, mainly the ASRs and ADDs, with a critical assessment and comparison of existing literature and established AK. This extensive review improves the validity and quality of the specification. Another purpose of the review is to compare and evaluate related architectural styles and patterns to apply in the new specification. This delayed literature review is similar to the Classic or Glaserian GT approach on the role of literature [15], [25].

Input: The data source for this step is conceptualized results from the previous three steps, such as the core categories of ASRs and ADDs.

Process: At this stage, a major literature review is performed. The literature review focuses on validating each outcome of the previous three steps. The validity of the context definition, the ASRs and ADDs can be investigated by evaluating each resulting conceptualization from the coding process in comparison to literature and existing distilled design experiences. Terms and phrases in core categories and the theoretical constructs can be improved and refined based on the comparisons. A thorough evaluation of each category and core categories is necessary to validate and improve the coding results.

Output: The outcomes at this step are polished ASRs and ADDs. AK is summarized from both the ASRs and ADDs to enable a baseline specification and description of the SA. The baseline AK is an initial specification that evolves through development time. As a result, at this stage, the applicability of architectural styles, patterns, technical standards and implementation technologies are identified. Furthermore, a summary of baseline AK is established.

e) Step 5: Constructing and Describing the SA: The purpose of this final step is to specify and describe the SA based on the established baseline AK. This process forms a holistic model of the whole SA. The baseline knowledge guides the description and representation of the architectural artifacts addressing different views. SAs can be constructed in various views and representations [16]. Additionally, SAs can have varying levels of abstraction. A particular SA could reflect the concrete structure of a system or conceptual one. Concrete architectures describe specific system structures with detailed implementation models and interactions. Conceptual architectures, a.k.a. Reference Architectures, on the other hand, are higher level abstractions of system architectures that also serve as a template for designing several detailed architectures [10].

Input: The data source for this step is the established baseline AK from the previous step.

Process: This stage is the final specification process based on the established AK baseline. A specification can begin by representing the key ASRs and a baseline design decisions summary or detailed concrete equivalence depending on the required level of abstraction. In Section III-B5, we discuss an example RA from a case study as illustrated in Figure 5. This step does not use the coding procedures from GT.

Output: The outcomes at this step are the specification representations, description of the SA, and a combined holistic model or a theory of the software context. As a result, at this stage, a model of the SA is specified and represented. Since the representations and specifications are grounded on the data collected, the resulting system architecture is a grounded SA with increased objectivity and transparency.

III. Case Study: A Reference Architecture for Appification

The software industry has seen notable changes in business information systems (BIS) due to trending technologies, such as cloud computing and service orientation. Similarly, the recent technology trend appification has remarkably changed the way applications are developed and delivered. Appification is a term coined with new application development trends in the areas of mobile apps and other smart devices. It is a recent architectural design approach which transformed earlier trends in information systems design towards small, separate and single feature applications or apps. The provision of selective service of a BIS, such as a banking system's specific singled notification feature or service, as a mobile app could be an example of appification. Unlike others, such as web-based applications, an app is designed to provide only a single service of the appified information system. Usually, multiple apps may be required to use other features of the same information system. These characteristics are distinct to the appification technology trend. However, there is no precise definition and architectural description of the phenomena. The architectural requirements for appifying BISs are unclear. Additionally, generic RAs for new technologies are not readily available. To this end, we conducted a study applying the approach presented in Section II.

A. Case Study Design

The goal of the case study is to demonstrate the feasibility of our approach by providing an overall architectural specification of the case phenomenon appification. Therefore, following the approach, we keep the focus on defining the particular SA context, eliciting its ASRs, and identifying the ADDs addressing of the specified ASRs. Finally, we aim to specify and propose a RA for appification.

1) Data Collection and Analysis Procedures:

a) Semi-structured Interviews: We carried out semi-structured interviews. To guide the process, we prepared interview guidelines, based on the defined RQs. After initial collection and analysis, subsequent interviews were directed to focus on emerging concepts. The average interview was 60 minutes long and recorded with the participants' consent. Each interview was transcribed verbatim. The transcribed documents were used for analysis and facilitated memoing during repeated readings.

b) Study Subjects: We conducted interviews with five domain experts. We chose the participants based on their involvement in related practice. They have the technical background working as engineers, architects, and consultants designing and software systems with changing architectural trends. Moreover, they have done significant activities in BIS development and possess a solid understanding and experience in the area of SAs. A detailed profile of each participant, interview details, and further relevant results are reported in the full case study report [14].

c) Analysis and Literature Review: For analysis, we followed the adapted approach with all the GT guidelines. To automate the analysis process, we employed tool support with MAXQDA [27]. Moreover, we performed an in-depth and targeted systematic literature review with the aim of validating and assessing coding results. A combination of automated search in databases and snowballing [37] is used to get better results. The review helped to investigate, enrich and validate instances of findings from the data.

B. Study Execution

In the following, we describe the execution as proposed in Sect. II-B.

1) Step 1: Defining the Appfication Technology Trend: In this initial step, we set and establish the context appification as an architectural trend. From the core categories, we conceptualized and described key characteristics of appification. Some of the characteristics are focus on *specific functionality*, *workflow-orientation*, *multi-channel capability*, and *high usability*. Based on these essential of characteristics, we define the appification technology trend as:

Appification is the development and provision of smaller applications in a single feature set per application to work on devices with varying sets of platforms and form factors.

The applications are called *apps*. The term *app* is a short form of the word application, which is meant to be application software. The term has appeared before the trend appification [21]. One of the interview participants described appification and apps as: *"[an app is] ... a container of some logic and some content and some interactions, ... and appification can mean giving more workflow and more interaction with that content but also restricting navigation coming from the web [perspective]."* The term, *devices* indicates any smart device or hardware that can run applications, which could be a smartphone, smart watch, smart TV, a smart device in a car, or a device in a kitchen appliance, or even a traditional personal computer. A *platform* is the operating system (OS) on the device, such as Android, Ubuntu Touch or any other OS. A *form factor*, here, indicates device's hardware design aspects, such as dimensions, shapes, and related specifications. Typical form factors include tablets and diverse set of smartphone form factors.

2) Step 2: Eliciting ASRs for Appification: Following the second step of our approach, we elicited 26 ASRs grouped into 8 core categories of ASRs. The first column in Table I contains the resulting list of ASRs elicited. Each ASR is given an ID starting with two characters as AR (Architectural Requirement) and followed with a number.

3) Step 3: Identifying ADDs for Appfication: At this step, we identified the main ADDs that address the ASRs by appification. The second column in Table I contains some of the resulting ADDs identified. Similarly, each ADD is given an identification starting with two characters as 'DD' and followed with a number. Furthermore, the table shows the mapping of the ASRs and the ADDs addressing them.

4) Step 4: Comparison and Evaluation: At this stage, we refined and validated the results, mainly the ASRs and ADDs, by modifying coded categories and core categories. Additionally, we selected four architectural styles and patterns to compare and find possible applications to appification architecture. We selected the styles and patterns based on their value to networked applications and excluded others. Table I provides the requirements and the results of the comparison. The Harvey Balls represent the levels of fulfilled properties, partial and full shades, showing the degree of support.

Based on the comparison, we observed SOA and the microservices architecture meeting more of the required properties than others. It is important to note that appification architecture can benefit by partially applying this architectural

Table I
MAPPINGS OF ADDS TO ASRS AND COMPARISONS

ID	ASRs	Associated ADDs	Client-Server	Three-Layered	SOA	Micro-services
AR01	***Separated Parts***					
	Frontend and Backend	Horizontal Split, Concerns (DD01)	◕	◑	◔	◔
	Replaceable Components	Loose Coupling (DD02)	◔	◑	●	●
	Highly Changing Parts	Separate Changing Parts (DD05)	○	◔	◔	◕
	Functionality Domains	Vertical Split Functionalities (DD01)	○	◔	◕	●
AR02	***Multiple Views***					
	Support Multiple UI	Vertical Split Functionalities (DD01)	○	◔	◕	◕
	Flexible Data Representation	Well-defined Interfaces (DD02)	◑	◑	●	●
AR03	***Personalized Experience***					
	Adaptive Interfaces	Adaptive Intuitive UI (DD06)	◑	◔	◑	◕
	Intuitive Usability	Adaptive Intuitive UI (DD06)	◑	◑	◑	◕
	Flexible Interaction	Focus on Usability (DD06)	◕	◕	●	●
	Context Level Prediction	Context Aware Workflow (DD06)	◑	◑	◑	◕
AR04	***Multi-channel Support***					
	Deployable Across Platforms	Deployable Across Platforms (DD04)	○	◕	◕	●
	Compatible to Multi-devices	Adaptable to Multi-devices (DD04)	○	◕	◕	●
	Responsive to Form factors	Responsive Design(DD04)	○	◑	○	●
AR05	***Functional Capability***					
	Specific/Single Purpose	Smaller Exchangeable Modules (DD03)	◑	◔	◕	◕
	Seamless Work Process	Focus on Usability (DD06)	◕	◕	●	◕
	User Familiar Workflows	Context Aware Workflow (DD06)	◑	◑	◑	◕
AR06	***App Level Cooperation***					
	Interfacing Devices/Apps	Well-defined Interfaces (DD02)	○	◔	◕	◕
	Action and Data Level	Well-defined Interfaces (DD02)	○	○	◕	◕
	Self-organized	Well-defined Interfaces (DD02)	○	○	◔	◕
	Context Based Virtualization	Aggregate for Larger Solutions (DD03)	○	○	●	●
AR07	***Dependable Flexibility***					
	Reliable Functionality	Design for Scalability (DD07)	◔	◕	●	●
	Supports Offline Capability	Provide Offline Capability (DD07)	◕	○	○	◕
	Security Concerns	App Federation (DD07)	◔	●	●	●
AR08	***Smart Apps Management***					
	Apps Marketplace	Automate Release Management (DD08)	◕	◔	◕	◕
	Automatic Live Updates	Automate Release Management (DD08)	◕	●	●	◕
	Flexible Licensing	In-App Activation (DD08)	◕	●	●	◕

styles. However, the implementation of both architectural approaches is different to appification. In both cases, services are parts of a single application implementation, and the separation is vertical, not horizontal. The services are located in one single tier in the backend with the whole BIS. On the other hand, appification separates the system parts horizontally as backend and frontend. Moreover, the frontend is further divided into smaller apps vertically. This separation lets apps function in users' device independently. It also enables apps to adapt to the user's particular context, which is the device and the functionality domain. As a result, we strongly recommend both horizontal and vertical separation in designing appification architectures.

5) Step 5: Constructing and Describing the Appification RA: In the final step, we proposed an appification RA for BIS. The constructed architecture is a result of summarized *AK base* from the ADDs. This knowledge base enables to specify and describe RA as a generic *baseline* or starting point.

In summary, first, we observed a design focus on horizontal separation - separate frontend and backend as two tiers. The rationale behind this comes from their nature and associated concerns. The frontend is required to be autonomous, and for that, the backend needs loosely coupled processes. Furthermore, appification separates out views because there is a strong need for multiple views. Second, the next key focus is the vertical separation of the frontend into smaller applications (apps). These apps must provide only one particular functionality of the BIS. Apps must be able to get deployed separately in client devices. Third, another principal ADD concern is the need for well-defined interfaces or APIs. This interface specification enables highly decoupled apps in the frontend to communicate with their server end and to each other. Commonly, HTTP-based protocols with REST APIs are used along with a lightweight messaging bus for communication. In Figure 5 we illustrate the primary elements

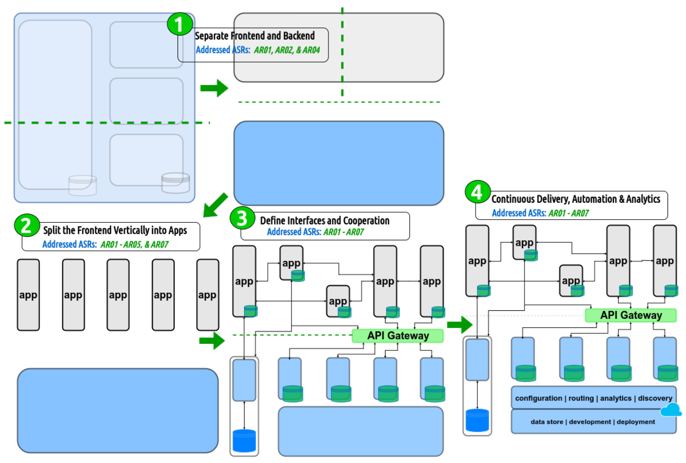
1
Separate Frontend and Backend
Addressed ASRs: AR01, AR02, & AR04
2
Split the Frontend Vertically into Apps
Addressed ASRs: AR01 - AR05, & AR07
3
Define Interfaces and Cooperation
Addressed ASRs: AR01 - AR07
4
Continuous Delivery, Automation & Analytics
Addressed ASRs: AR01 - AR07
app
API Gateway
configuration | routing | analytics | discovery
data store | development | deployment

Figure 5. Summarized baseline AK for the Appification RA (Representation).

of the RA. For representations, we used simple diagrams of architectural elements, such as frontend, backend, and the connecting interfaces. The fourth final baseline point is the use of a diverse set of tools and platforms that exist in a cloud environment to enable continuous delivery, analytics, and automation of development activities. In general, these four summarized baseline points address essential levels of the specification of appification RA in a BIS.

C. Threats to Validity

Results discovered with GT can be validated by surveys, further observations or literature. We applied a major literature review to validate and triangulate the results. Besides, our coding results are further evaluated with *analyst triangulation* at initial and final phases of the study. However, GT as a socio-technical method brings challenges of subjectivity to the study raising threats to validity. Although GT has adequate mechanisms to base the conceptualizations objectively on the data source, it does this by generating symbols in the form of codes. This labeled symbols could be subjective to the experience and perspectives of the examiner of the data. This threat goes beyond GT to the underlying philosophy behind it, which is symbolic interactionism [3]. Consequently, the meaning of symbols generated as concepts, and categories might be influenced by the individual interpretations of the symbols' representation. As a result, there might not be one solution for abstracting the categories and the core categories. This threat does not necessarily make the outcome, the understanding or theory, less objective but does not guarantee its objectivity as well. Understanding these key challenges, we refined and polished coding results, particularly the categories and core categories, by doing a major literature review to build reproducible theoretical fragments for the definition of appification, its ASRs, ADDs and the summarized AK base. Furthermore, the overall procedures and guidelines of GT, which firmly root the abstractions on the data, has contributed to our adapted approach the ability to foster objectivity on the design of grounded architectures.

IV. Related Work

Here, we present related work on the use of GT in specifying architectures and SE in general.

A. Approaches to Specify SA

Several architectural frameworks and architecture specification techniques, such as scenario-based [12], [24], are used in analogous abstractions as mechanisms and resources to describe or specify SAs [1]. Furthermore, we find a clear indication of GT's value employed on re-engineering systems from earlier proposals by Galal et al. [13]. However, frameworks such as TOGAF and the Zachman Framework are exercised in the scope of business enterprises addressing mostly additional strategic business concerns of enterprises [38], [6]. They are not well-suited for contexts which are not well-defined, such as new SA trends. Moreover, they put emphasis on the description of SAs as components, connectors, and configurations putting inadequate attention on the transparency and objectivity of the decision-making and development of the architecture [34].

Subsequently, in addition to the use of qualitative knowledge in the decision making, the distinction of our approach lies in the focus of establishing AK base from ASRs and ADDs [34]. To the best of our knowledge, no systematic approach exist that specify SAs from ASRs and ADDs making the specification process transparent and objective.

B. Grounded Theory in Software Engineering

The use of GT is widespread in the SE community, and a fair amount of literature exist that applies GT to varying contexts [20]. A study investigating the use of GT by Stol et al. [25] shows the increasing significance of GT in SE and provides guidelines for its proper application.

C. Grounded Theory and SA

a) Discovering Patterns and Scenarios: In a study by Carsten et al. [18], they use GT for pattern mining with the goal of investigating architectural patterns to integrate business processes in service-oriented architecture. The study employs particularly Glaserian GT [15] using a pattern language extracted mainly from interviews. A related study by Galal [12] deploys GT to drive system-wide scenarios from a contextual analysis of BISs. This technique is pointed out as Grounded Systems Engineering Methodology (GSEM) [12]. Similarly, Bery et al. use GT as a means for discovering requirements and architectural artifacts in the process of architectural recovery [7].

b) Analysis and Specification: Rosasco and Dehlinger [26] also use GT as a part of a lightweight enterprise architecture (EA) elicitation aiming to lower the level of expertise required by EA frameworks. Furthermore, the study by Singh and Bartolo also propose an approach for transforming GT to user requirements and tries to show how the approach could bring rigor to software systems design [32]. In a different context, a study by Chakraborty and Dehlinger [8] uses GT to extract functional and non-functional requirements from EA documentations in a structured approach. The process also enables backward traceability of architecture to requirements. Another work, by Adams and Courtney, reports the integration of GT as a part of a multi-methodological framework which is introduced as DAGS - Design Science, Action Research, Grounded Theory, and Systems Development [2]. Increasingly, other studies also employ GT to some level of SA specification or improving better understanding of the design and specification process, such as identifying requirements or types and metaphors of SA specification [29], [17], [23], [36]. Both groups of studies focus on analyzing, specifying, or mining of a single aspect of a SA, such as eliciting requirements, discovering patterns or describing specific scenarios. Our approach, on the other hand, provides a systematic way to guide the whole process from the initial level of defining a software context into the final process of SA specification. The approach considers each step's value to foster objectivity and improve transparency in the decision making. Furthermore, taking SA as a set of design decisions [22] we emphasize on identifying the ADDs addressing ASRs.

V. Discussion

From the presented results of the case study, we conclude that applying GT to specify SAs is feasible and that the resulting architecture is based on an *objective* and *transparent* decision process.

a) Transparent Architectural Design Decisions: Furthermore, we argue that the resulting AK is transparent due to the structure of the developed theory. In particular, each ADDs is justified by one or more ASRs, i.e., each architectural decision can be traced back to a corresponding requirement. By this, ADDs can be justified based on requirements rather than personal preferences of the architect or conventional wisdom [19].

b) Objective elicitation of Architectural Design Experience: Overall abstraction is grounded in the data. Consequently, we argue that the results are objective as we applied the GT approach to elicit ASRs and ADDs. Objectivity is addressed in several ways: First, theories in GT are based on a firm set of qualitative data. In particular, in our case, we derived a set of 8 ASRs and 8 ADDs that characterize architectures in the context of the technology trend appification based on expert interviews. Thus, elicited ASRs as well as ADDs are based on the consolidated experience of practitioners working in the respective context rather than personal preferences of corresponding architects [19].

Furthermore, theories in GT are systematically derived from collected data in a reproducible way. Thus, the validity of architectural decisions can be independently validated by simply replicating the coding on available data. Indeed, we argue that an independent coding of the data collected by the interviews would yield a similar architecture as proposed in Sect. III.

c) Limitations of our approach: As any empirical approach, our approach is subject to threats to validity which should be mentioned in the following: (i) The selection of data sources provides a major threat to the validity of the developed architecture. Data sources may be selected based on the architect's preferences. (ii) Another serious threat to validity concerns the coding process itself. Coding may again be subject to personal preferences and introduce subjectivity in the approach. If applying our approach to the design of new architecture, particular attention has to be given to cope with those threats.

d) First step towards a well-founded RA: We do not claim that the resulting architecture proposed in Sect. III is the best possible architecture, however, by applying our approach, we provide the first step towards a well-founded reference architecture in the context of the technology trend appification. Thus, we propose that our approach should be independently replicated to eventually create such a well-founded RA.

VI. Conclusion

In this paper, we provided an approach to architectural design based on GT. In particular, we consider the development of an architecture as the development of a theory consisting of ASRs and corresponding ADDs. GT can then be used to build this theory. We evaluated the approach by developing an architecture in the context of the technology trend appification. To this end, we conducted expert interviews to elicit ASR and ADDs and literature reviews to back the relationship of ASRs and ADDs.

The approach is characterized to ensure transparency and foster objectivity in the design process. Objectivity is reached by the objective nature of the GT methodology. Decisions are grounded in data rather than subjective opinions. Transparency, on the other hand, is reached by the proposed structure of a theory. Since ADDs are motivated by identified ASRs, each decision can be traced back to the corresponding ADD.

As shown by the case study, the approach is well-suited for the design of architectures in new application domains. Due to the lack of experience in such domains, significant ASRs as well as corresponding ADDs have to be identified. The approach described in this study provides a systematic way to do so in an objective and transparent way.

Acknowledgment

We would like to thank all the interview participants as well as Ilias Gerostathopoulos, Veronika Bauer, and Anna-Lena Lamprecht for their suggestions and helpful comments on earlier versions of this work.

References

[1] ISO/IEC/IEEE 42010. Iso/iec/ieee systems and software engineering: Architecture description. 2011.

[2] Lascelles A Adams and James F Courtney. Achieving relevance in is research via the dags framework. In *System Sciences, 2004. Proceedings*. IEEE, 2004.

[3] Khaldoun M Aldiabat and Carole-Lynne Le Navenec. Philosophical roots of classical grounded theory: Its foundations in symbolic interactionism. *The Qualitative Report*, 2011.

[4] L. Strauss Anselm and Glaser Barney. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. LWW, 1967.

[5] Glaser Barney. The discovery of grounded theory: Strategies for qualitative research, 1978.

[6] Hans van den Bent, Tom van Sante, Dennis Kerssens, Janine Kemmeren, et al. Togaf, the open group architecture framework, 2008.

[7] Daniel M. Berry, Michael W. Godfrey, Ric Holt, Cory J. Kapser, and Isabel Ramos. Requirements specifications and recovered architectures as grounded theories, 2013.

[8] Suranjan Chakraborty and Josh Dehlinger. Applying the grounded theory method to derive enterprise system requirements. IEEE, 2009.

[9] Lianping Chen, Muhammad Ali Babar, and Bashar Nuseibeh. Characterizing architecturally significant requirements. 2013.

[10] Robert Cloutier, Gerrit Muller, Dinesh Verma, Roshanak Nilchiani, Eirik Hole, and Mary Bone. The concept of reference architectures. 2010.

[11] Juliet M. Corbin and Anselm Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 1990.

[12] Galal H Galal. Scenario-based systems architecting. In *Requirements Engineering, 2001*. IEEE, 2001.

[13] Galal H Galal and Janet T McDonnell. Knowledge-based systems in context: a methodological approach to the qualitative issues. *AI & SOCIETY*, 1997.

[14] Habtom Kahsay Gidey. Conceptualisation and development of a future-proof architecture in the context of business information systems. Master's thesis, Technische Universität München, Fakultät für Informatik, 2015.

[15] B.G. Glaser. *Basics of Grounded Theory Analysis: Emergence Vs. Forcing*. Sociology Press, 1992.

[16] Milena Guessi, Lucas Bueno Ruas de Oliveira, and Elisa Yumi Nakagawa. Representation of reference architectures: A systematic review. In *SEKE*, 2011.

[17] Werner Heijstek and Michel RV Chaudron. The impact of model driven development on the software architecture process. In *SEAA*. IEEE, 2010.

[18] Carsten Hentrich, Uwe Zdun, Vlatka Hlupic, and Fefie Dotsika. An approach for pattern mining through grounded theory techniques and its applications to process-driven soa patterns. In *PLoP*. ACM, 2015.

[19] Tom-Michael Hesse and Barbara Paech. *Documenting relations between requirements and design decisions: A case study on design session transcripts*. 2016.

[20] Rashina Hoda, James Noble, and Stuart Marshall. Grounded theory for geeks. ACM, 2011.

[21] Thom Holwerda. The history of 'app' and the demise of the programmer. http://www.osnews.com/story/24882/. [Online; accessed 18-July-2015].

[22] Anton Jansen and Jan Bosch. Software architecture as a set of architectural design decisions. In *WICSA*, 2005.

[23] Andreas Kaufmann and Dirk Riehle. Improving traceability of requirements through qualitative data analysis. In *Software Engineering & Management*, 2015.

[24] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. Scenario-based analysis of software architecture. *Software, IEEE*, 1996.

[25] Paul Ralph Klaas-Jan Stol and Brian Fitzgerald. Grounded theory in software engineering research: A critical review and guidelines, 2015.

[26] Nicholas Rosasco and Josh Dehlinger. Application of a lightweight enterprise architecture elicitation technique using a case study approach. IEEE, 2015.

[27] Elif Kuş Saillard. Systematic versus interpretive analysis with two caqdas packages: Nvivo and maxqda. *Forum: Qualitative Social Research*, 2011.

[28] Johnny Saldaña. *The coding manual for qualitative researchers*. Sage, 2015.

[29] Kari Smolander, Matti Rossi, and Sandeep Purao. Going beyond the blueprint: Unravelling the compex reality of software architectures. *ECIS*, 2005.

[30] Stefan Sobernig and Uwe Zdun. Distilling architectural design decisions and their relationships using frequent item-sets. 2016.

[31] Klaas-Jan Stol and Brian Fitzgerald. Theory-oriented software engineering. *Science of Computer Programming*, 2015.

[32] Singh Supriya and Bartolo Kylie. Grounded theory and user requirements: a challenge for qualitative research. 2005.

[33] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[34] Jan Salvador van der Ven, Anton GJ Jansen, Jos AG Nijhuis, and Jan Bosch. Design decisions: The bridge between rationale and architecture. In *Rationale management in software engineering*. Springer, 2006.

[35] Stefan Wagner and Daniel Méndez Fernández. Analyzing text in software projects. In *The Art and Science of Analyzing Software Data*. 2015.

[36] Michael Waterman, James Noble, and George Allan. How much up-front? a grounded theory of agile architecture. In *ICSE*. IEEE, 2015.

[37] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. EASE '14. ACM, 2014.

[38] John Zachman et al. A framework for information systems architecture. 1987.