Systematic Application of Domain-Driven Design for a Business-Driven Microservice Architecture

Benjamin Hippchen, Michael Schneider, Pascal Giessler
Sebastian Abeck

Cooperation & Management (C&M), Institute for Telematics
Karlsruhe Institute of Technology
Karlsruhe, Germany
{benjamin.hippchen, michael.schneider, pascal.giessler, abeck}@kit.edu

***Abstract*—Today's cloud providers open up new opportunities for software development. Unfortunately, however, not all existing applications are ready for operation in the cloud. One reason for this is usually the chosen architecture, which offers little flexibility like the monolithic architecture. In order to take up the possibilities in the cloud, a flexible architecture such as the microservice architecture is required. Developing with such an architecture is challenging and requires experienced team members. Especially the design of microservice-based applications is the challenge. Utilizing domain-driven design can be beneficial in breaking business functionality down into microservices. But also the use of domain-driven design requires a lot of experience, due to the lack of systematic. For this reason, we have created a systematic for the application of domain-driven design in the context of microservice development. The systematic accompanies the development team through the development process and supports them in the design and modelling of microservices. To cover the choreography of microservices, a new diagram called context choreography has been added. We also present UML profiles to support the modeling activities. Our systematic has shown over a longer period of time that it has a verifiable positive effect on microservice development.**

Keywords–Microservice; Microservice Architecture; Domain-Driven Design; Context Map; Bounded Context.

I. INTRODUCTION

This article is an extended version of [1], which was published at SOFTENG 2019. The following aspects were added to the original work: (1) Consideration of event-driven communication within the context choreography; (2) Supporting the application of DevOps paradigms with the context map and context choreography. (3) Modeling of the context map; (4) UML profile for context map and context choreography. The digital transformation is in progress and organizations must participate; otherwise, they will be left behind. Existing business models have to be rethought and new ones created. Tightly coupled to the business model is the organization's application landscape. Thus, this landscape has also to be reimagined. Meanwhile, microservice architectures have established themselves as an important architectural style and can be considered enablers of the digital transformation [2]. Therefore, one major step towards a digital organization is the migration of legacy applications with monolithic software architecture into a microservice architecture. Afterward, the architecture must be maintained to provide long-lived software systems. However, neither the migration, design and development of a microservice architecture nor its maintenance are easy to achieve.

The structure of the new microservice-based application seems straightforward for the development team. Some microservices communicate with each other and deliver business-related functionalities over web application interfaces (web APIs). Another approach to communication is the event-based exchange of information. One also speaks of messaging [3]. The use of events is becoming more and more popular, as it provides a loose coupling in the context of microservices. However, at this point, the corresponding development team must ask itself decisive questions: How many microservices do we need? In which microservice do we put which functionality? Do we interact with third party applications? Domain-driven design (DDD) by Evans [4] provides important concepts which help answering these questions. As a software engineering approach, DDD focuses on the customer's domain and wants to reflect this structure into the intended application. The business and its business objects are the focus of each developing activity. Technical details, like the deployment environment or technology decisions, are omitted and do not appear in design artifacts. This is also a weak point of DDD, because from a certain point technical decisions are of great importance. An example of this is the operation of microservices, which is of a purely technical nature. DDD emphasizes the use of a domain model as a main development artifact: all relevant information about the domain, or business, is stored in it.

For microservice architectures, DDD helps structuring the application along business boundaries. Likely, these boundaries match the customer's domain boundaries. In his book *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Evans introduces the "context map" diagram. The main purpose of the context map is to explore the customer's domain and state it as visual elements. The context map focuses for example on the macro structure of the domain, sub domains, departments instead of micro elements like business objects. A further essential DDD element and pattern is the "bounded context," which represents a container for domain information. This container is filled with the mentioned domain's micro structure, creating a domain model. The name bounded context is derived from its explicit boundary. Through this boundary, the container's content is only valid inside of the bounded context. From the strategic point of view, a bounded context is a candidate for a microservice. It is important to note, that the bounded context should be mostly independent of other parts of the domain. This supports the idea of microservices. Thus, the context map could display the organisation's microservice

architecture.

Like most DDD concepts, creating a context map is challenging and the tasks are not straightforward. The vague definitions and lack of process description impose problems. In addition, DDD does not provide a modeling language like Unified Modeling Language (UML). The following example illustrates several problems. A development team wants to establish a microservice architecture at Karlsruhe Institute of Technology (KIT) for the administration of students. Typically, for this purpose, universities introduce Student Information Systems (SIS) to support the business process execution for their employees. There are several problems with those SIS: (1) in the hands of software companies, (2) little to no understanding of the university's domain, (3) primarily monolithic architecture, and, (4) little to no insights for third parties. Since the development team has no affect on the SIS and its architecture, the goal is to enhance the SIS with social media aspects to support interaction between students. A microservice architecture is planned for the new functionality. In order to use DDD, the team must gather information about the domain and create a domain model and context map. The first uncertainty is the creation order of the artifacts. Both artifacts rely on information from each other. While creating the domain model, the development team has to know where to look for specific domain information, which is stated in the context map. When creating the context map, several bounded contexts are needed, which contain a domain model. In addition, the content of a context map is not precisely defined. The literature states that the context map contains bounded contexts and relationships but does not state how to elicit them or what they represent in the real world. This lack of real-world representation is especially a problem for development teams, who have to interact with an existing application. On the one hand, it is necessary to provide the third party application in the context map, since the context map can capture the information transferred between the third-party and the university. On the other hand, it is unclear how to represent the third-party application in the context map. A bounded context needs a domain model, and there is no domain model in this case. These are only two problems with the application of the context map, but they illustrate how import it is to enhance usability. In the following sections, we discuss these and other problems in more detail.

In this article, we provide the following contributions to enhance the application of the context map and support the design and maintenance of a microservice-based application:

- **Context Map Foundations**: One major problem of DDD is the lack of integration and placement in existing software development processes. It is unclear in which phases the context map must be created and in which phases it supports the development. Therefore, in Section III, we provide the first integration and placement of this map. In addition, we discuss the foundations of the context map and define the elements in this section.
- **Context Choreography**: While applying DDD for the development of microservice-based applications, we realized the existing artifacts did not capture all relevant information. Thus, in Section II and Section III, we introduce a new type of diagram, the "context choreography". This diagram's purpose is to display the choreography between multiple microservices for the application. To suit the ever more popular event-driven communication, the context choreography can represent the exchanged events between the microservices needed for the application.
- **Artifact Creating Order**: As mentioned, it is unclear in which order the DDD artifacts must be created. Therefore, in Section III, we also provide a detailed order with an emphasis on the context map. The application of the order is presented in our case study in Section V.
- **UML Profiles** - The look and feel of a context map depends on the modeler. DDD does not provide any modeling language for the style of a context map. Thus, each context map must be read differently, depending on the conceptional understanding of the modeler. We see a problem in the missing modeling systematic of a context map. In Section IV, we provide UML profiles for the context map and a the context choreography. Especially for the collaboration between multiple development teams, it is necessary to have a clear understanding and a limited amount of diagram elements.

II. Placement and Integration of the Context Map

One main problem of DDD is its lack of placement in the field of software development. Neither its models nor its patterns, including the context map, are placed in common software development processes. For our placement, we focus on the context of microservices. Since the context map has some weaknesses in development, a new diagram is introduced to close the gap.

A. Placement

As mentioned in Section I, the use of the context map is not straightforward. The development team must analyze the domain, create a domain model, and develop a context map. On the one hand, the context map has a great benefit for microservice architectures. On the other hand, applying the context map correctly is difficult.

Each DDD practice should be performed with the focus on an intended application [4]. This ensures the "perfect fit" of the gathered information, called "domain knowledge," for the application. Domain knowledge is captured in domain models. At this point, the pattern "bounded context" becomes important. An application consists of multiple bounded contexts, which all have their own domain models. With respect to the complexity of the domain knowledge, it makes sense to split the domain knowledge into multiple domain models. The validity of each domain model is limited by the bounded context. Furthermore, each bounded context has its own "ubiquitous language," which is based on the domain knowledge and acts as a contract for communication between project members and stakeholders. For the development of microservice-based applications especially, the multiple bounded contexts support the idea of a microservice architecture. Through connections between the bounded contexts, the domain knowledge is joined together in the application. The arising relationships are application-specific and differ from application to application. There are several types of relationships [5]. Modeling the

bounded contexts and their relationships is the purpose of a context map.

Considering a microservice architecture, the purpose of a context map is not only to elicit domain knowledge. Organizations that introduce microsevices need to manage their application landscape to maintain the microservice architecture. Without the knowledge about which microservices are available and who is in charge of them, the microservice architecture loses its advantages. Existing microservices are simply not used, even the domain knowledge they provide is required, due to the fact that other development teams could not find it, oversee it or forgot about it. The required domain knowledge is redeveloped in new microservices and the existing microservices become legacy. Sustaining the advantages of a microservice architecture is therefore important for organizations and the context map is one tool which helps to achieve this. In addition, the aspect that DDD focus its development artifacts on the customer's domain, supports the maintenance of the microservice architecture. Aligning the context map to the customer's domain leads to a natural-looking architecture [6]. Conway's Law [7] also supports the idea behind this. The organizational structure is adapted to the microservice architecture and vice versa. Looking at concepts like Martin Fowler's "HumaneRegistry" [8] or API management products like "apigee" [9], the idea and approach of the context map is required and furthermore it supports such concepts and products.

Using the context map as a tool for maintaining the microservice architecture is contradictory to one DDD aspect: focus always on an application. The mentioned maintenance does not require any kind of application-specific information. A microservice is mainly an application-independent software building block [10] and needs to be treated as such while maintaining the microservice architecture. Even if the development of a microservice is motivated through the development of an application. Thus, we see the context map as an application-independent diagram.

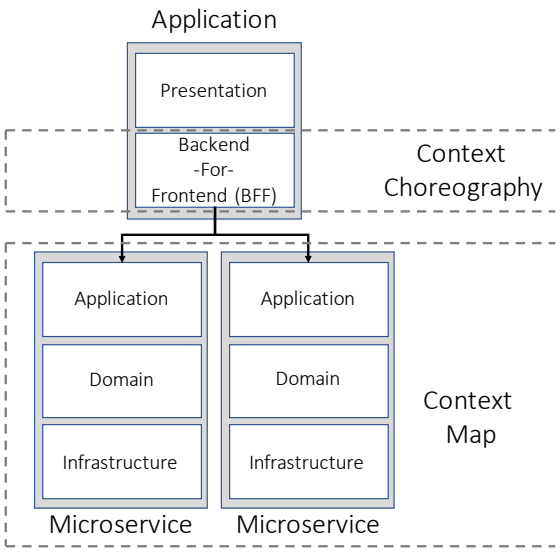

Figure 1. Placement of context mapping artifacts regarding the software building blocks from [10]

According to [10], for microservice-based applications, microservices are choreographed in applications through a backend-for-frontend (BFF) pattern. This is where application-specific information comes into play. Fig. 1 depicts the software architecture, including the application's BFF. To capture the choreography in the BFF, a new type of diagram is needed. The "context choreography" provides a view of the bounded context necessary for the application. Furthermore, the context choreography indicates which domain knowledge the bounded context transfers. We have chosen the term choreography because choreography is the real goal of the way Microservices communicate. A distinction is made between choreography and orchestration [11]. In orchestration, the interactions between the microservices must be triggered either by a central instance or by the microservices themselves. This results in an unwanted coupling between the building blocks. In contrast, the choreography relies on a loose coupling by reacting to events that take place in a message broker (such as RabbitMQ [12] or Kafka [13]). The microservices only need to know the events to which they have to react. It does not matter who created the event.

The context map can also be placed into software development activities. In [10], the first steps to place DDD into the software development activities from Brügge et al. [14] were taken. However, the context map itself was omitted. We built on these results for our placement of the context map. Domain-driven design introduces two types of "design activities" [4]. The first is the "strategic design," with tasks in modeling and structuring the domain's macroarchitecture (e.g., departments are used to define boundaries). This macroarchitecture is captured in the context map. Secondly, the "tactical design" further refines the macroarchitecture and enriches the bounded contexts with domain knowledge. This activity represents the microarchitecture of the domain and therefore of the microservice. Both activities rely on creating diagrams. Considering the software development activities from Brügge et al., Evans' designation as strategic and tactical "design" is misleading. Those focus more on the analysis than on the design phase. Many DDD practices and principles, such as "knowledge crunching," aim to analyze the domain. The development team explores the customer's domain and should simultaneously create the context map and domain model. Thus, the strategic and tactical designs are completed out, which is why the context map must be integrated at this point.

As mentioned, the content of the context map depends on its purpose. This is even the case for the relationships between the bounded contexts. Developing a monolithic application requires a different viewpoint on these relationships than a microservice-based application. A microservice architecture has many different microservices, which are managed by different development teams. By choreographing microservices in applications, development teams are automatically interdependent. This dependency is illustrated in the relationships in the context map. They could also be seen as communication paths between those development teams.

Our placement indicates that the context map has several possibilities to support the development of microservice architectures and microservice-based applications. We distinguish between a microservice architecture and the development of a microservice-based application. With regard to the microservice architecture style, the context map provides an overview

of all in the application landscape existing microservices and further the dependencies of the responsible development teams—which are also necessary information for maintaining the microservice architecture. Due to the application-independence of these information, the context map is an application-independent diagram. Additionally, we saw a lack of the context map while specifying microservice-based applications. Information transferred between microservices was missing a specification, which is necessary for choreographing the BFF of the application. Thus, we introduced the context choreography, which displays the application-specific dependencies between the microservices and their transferred domain knowledge. With this placement, we make a first step in advancing the use of the context map.

III. Foundations and Artifact Order

In addition to the placement, we see a high need for clear definitions and guidance in creating the context choreography and context map. Therefore, this section provides definitions for terms regarding both artifacts. Afterward, we explain how the artifacts could be created.

A. Foundations

We found that, in addition to the development process, terminology around the context map is not clearly defined. This also leads to difficult application. Therefore, we want to provide some basics.

1) Bounded Context: The bounded context is the main element for the context map and is an explicit boundary for limiting the validity of domain knowledge [4]. Thus, within this context, there is a domain model and its ubiquitous language. A bounded context does not represent an application. This is based on the layered architecture of DDD, which consists of four layers: (1) presentation, (2) application, (3) domain, and (4) infrastructure. Domain-driven design and its artifacts focus only on the domain layer and omit the others. Therefore, without any application logic in a domain model, a bounded context cannot represent an application at all. This definition is fundamental, when creating a context map. An intended application is usually integrated into an existing application landscape.

When developing a microservice-based application, a bounded context initially only represents a candidate for a microservice [6]. Thus, a bounded context is either large enough that two or more microservices are necessary or small enough that they are included in one microservice. The best practice, however, should be the one-to-one relationship. This relationship eases the maintenance of the architecture through a clearer mapping between bounded contexts, microservices, and the responsible development teams. Reconsidering the size of the bounded context helps achieve this mapping. Therefore, we have collected several indicators, or more precisely possible influence factors, for the size of bounded contexts from our experiences in research and practice. This list should not be considered complete or verified with an empirical study but should rather be seen as an aid. A bounded context (1) has a high cohesion and low coupling, (2) can be managed by one development team, (3) has ideally a high autonomy to reduce the communication/coordination effort between development teams, (4) has a unique language that is not (necessarily) shared, and (5) represents a meaningful excerpt of the domain.

2) Context choreography: As mentioned (see Section II), the specification of a microservice-based application was lacking some information. Thus, we introduced the context choreography as a new diagram.

For microservice-based application development, it is important to state the other needed microservices—and thus also the bounded context. Furthermore, the exchanged data between those microservices are important information. As a microservice-based application is developed, existing microservices could still be used, while new ones are developed. In both cases, the context choreography is supportive. Regarding the application itself, the context choreography maps all microservices dependent on web interfaces and the consuming events, which are necessary to achieve the application's functionality. In further steps, the events to be published by the microservices could be integrated. A connection of the two microservices via an event does not mean that both communicate directly with each other. This is the case when shared entities are exchanged via web interfaces. According to the software architecture provided by [10], the application's BFF is specified. Independent from the application, the context choreography states the microservice web interfaces. Both the consumed and the provided interfaces of the microservice are provided. Thus, while developing the application, the first hints for designing the API can be derived. With regard to the subsequent maintenance of the microservice, development teams are able to identify the microservices that rely on them and vice versa.

3) Context Map: The DDD's original purpose for the context map differs from the one provided in this paper. In the context of microservice architecture, the context map is a useful diagram for maintaining the architecture and supporting application development.

One major advantage is the comprehensive overview of existing microservices. According to the best practice from Section III-A1, each bounded context in the context map represents a microservice. Further, in software architecture, social and organization aspects have to be considered [15]. Therefore, dependencies between microservices, and thus development teams, are stated. When development teams want to evolve their microservices, it is important to ask who depends on these microservices. At this point, the dependencies on other teams must be considered because any change could affect the stability of the other microservices. Thus, changes have to be communicated.

Also, for the development of a microservice-based application, the context map is advantageous. Regarding the context choreography, existing microservices are used to compose functionality for the intended application. Using existing microservices is only possible if they are traceable in the microservice architecture. This is where the context map comes into play. After developing a microservice, it is placed as a bounded context into the model. While the application is in development, the development team can use the context map as a tool to locate the needed microservices.

4) Domain Experts and other Target Groups: The interaction between domain experts and developers is one principle of DDD [4]. Each artifact is created for and with domain experts. Thus, the artifacts should be understandable without a software development background.

The context map according to DDD's definition is also relevant for domain experts [16]. However, according to our definition, we do not see any advantages for domain experts since the context map provides an overview of bounded contexts and communication paths between development teams. Furthermore, the context choreography is irrelevant. Only the subdomains, which represent the organization's structure, contain helpful information.

5) Supporting DevOps Paradigms with the Context Map and Context Choreography: The paradigm of Development & Operations (DevOps) is becoming more and more relevant for software development [17]. Especially in the context of the development of microservices, essential concepts have opened up. The paradigms in the context of microservices are relevant for coping with the challenges of the microservice architecture, such as the shorter time-to-market. The shorter development cycles can thus be absorbed by continuous integration and continuous delivery (CI/CD) [18]. A great synergy between microservice architecture and DevOps arises during the handling of the development team [19]. The development of microservices requires a rethink within the development team. The classic approach of large development teams is to split up into smaller self-sufficient development teams [6]. This idea is also supported by DevOps, which empowers a development team for the entire lifecycle of a microservice.

Important for the use of DevOps is the consideration of the paradigms during the architectural design [20]. For example, it is highly relevant that a microservice can be further developed independently of other microservices. To achieve this, the content of the microservices must be highly cohesive in order to be loosely coupled to other microservices. This is precisely where the designed system and the artifacts that are created come in handy. By using the context map, the domain can be separated into coherent microservices, which exist independently and can be further developed independently of the other microservices. Furthermore, the number of microservices and thus the number of necessary development teams can be read when designing the context map.

The context choreography can support the operations team of the microservices with the operation. From the artifact the necessary endpoints can be identified, which have to be released for the communication between the microservices. Such approval is particularly relevant at the infrastructure level.

B. Process for Establishing a Context Map

To develop a microservice-based application, it is necessary to establish the bounded contexts needed for the application. The developed application may reuse existing microservices, which should be integrated into the application landscape. To obtain an overview of the microservice landscape, the context map is useful. In this section, we focus on the establishment of the bounded contexts, the context choreography, and the context map. For developing an application, we build on a development process based on behavior-driven development (BDD) [21] and DDD [4] introduced in [10]. We omit the steps in [10] and focus on the creation of context choreography and a context map. Therefore, this section describes how the context map is established and enhanced within the development process.

1) Forming the Initial Domain Model: Forming the initial domain model occurs in the analysis and design phases. Before developing an application, the requirements are specified with BDD in the form of features. As Fig. 2 illustrates, a tactical diagram is derived from the features (e.g., the domain objects and their relationships). If a domain model already exists (e.g., from an existing microservice), this should be taken into account. The resulting diagram represents the initial domain model, which contains the application's business logic. Thus, the domain model provides the semantic foundation for all the specified features. The resulting diagram is comparable to a UML class diagram and displays the structural aspects of the domain objects. If the domain structure is still vague when the number of features is satisfied, more features are considered until the domain model appears to be meaningful. Afterward, as presented in Fig. 2, this initial domain model is examined and structured into several bounded contexts.

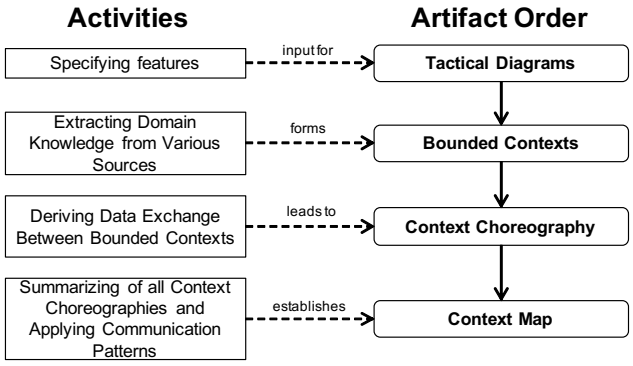

Figure 2. Creating order for artifacts their and impacting activities

2) Forming the Bounded Contexts: The model is strategical analyzed and separated based on the business and its functionality. This step depends on the domain knowledge and the structure of the business. Therefore, knowledge crunching from DDD [4] is applied to gather that knowledge. Often, a business's domain knowledge is scattered through the whole business. Therefore, analyzing the business is important to understand the business processes and the interaction of different departments. By default, each department knows its tasks the best. To extract the domain knowledge, various sources should be considered. These sources include domain experts who are part of a department, as well as documents and organizational aspects. This domain knowledge provides hints for structuring the domain and has to be considered while forming the bounded contexts. Considering the application analysis and the business analysis from [10] leads to the bounded contexts, as illustrated in Figure 2. If a context map has been established, then the context map is searched for the required domain knowledge of the application. If a bounded context representing the domain knowledge already exists, then this bounded context is taken into account. A new bounded context is established if the context map does not contain the required domain knowledge. For example, we integrated a profile context into an existing context map of the campus management domain.

3) Toward the Context Choreography: Forming the bounded contexts is only the first step towards a working

TABLE I. Overview of communication patterns and their impact

Comm. pattern	Description	Effort
Partnership	Cooperation between bounded contexts to avoid failure	Very high
Shared Kernel	Explicit shared functionality between different development teams	Medium to high
Customer/ Supplier	Supplier provides required functionality for the customer. The customer has influence on the supplier's design decisions	High to very high
Conformist	Similar to customer/supplier but with no influence on design decisions.	Low to medium
Separate Ways	No cooperation between development teams	Low
Anticorruption Layer	Additional layer that transforms one context into another	Low
Open Host Service	Uniform interface for accessing the bounded context	Low
Published Language	Information exchange is achieved using the ubiquitous language of the bounded context	Low

application. Each previously established bounded context is considered a microservice and requires or offers a unique interface for communication that can be based on REST or other paradigms like messaging. The microservices are choreographed with the BFF. Either the BFF directly accesses the web interfaces or creates events to which the microservices react. To allow choreography, the data exchange between the bounded contexts is considered next. The required data exchange is modeled in the context choreography. For each bounded context, a context choreography diagram is modeled. Domain objects that need to be shared or consumed from other bounded contexts are modeled as shared entities or events. The considered bounded context can either share the domain object itself or consume the domain object. Within an event there is information about the domain objects (see domain objects [5]).

4) Toward the Context Map: In microservice architectures, each established bounded context represents a microservice and is implemented by autonomous development teams. Thus, for relating bounded contexts, teams may need to communicate with each other. Therefore, the communication effort between the teams should be considered. The communication effort indicates how much communication between the teams is required. Clear communication paths are necessary, because a team needs to know which other team is responsible for relating bounded contexts. Therefore, dependencies and communication channels between teams are defined. Depending on the teams and the possible communication effort, a communication pattern is chosen based on [4] [5] (see Table I). The last three communication patterns listed in Table I are special patterns designed to reduce the communication between different teams, as well as the impact on interface changes. Other benefits and drawbacks of particular patterns exist but they are out of the scope for the current discussion. The context map illustrates the determined communication path between the bounded contexts. For example, when the communication between teams is not possible, such as when foreign services are adopted, DDD patterns need to be applied. For foreign services, the ACL pattern should be applied. In the last step, as depicted in Figure 2, the relationships (including the pattern) and the bounded contexts are added to the context map diagram.

Adding those bounded contexts and communication relationships is an essential part of the context map. This concludes the first cycle of the analysis and design phases. The information about team dependencies helps in managing the development team and especially in restructuring in the sense of microservice architecture and DevOps.

5) Adjustments of the models: After the design phase, the implementation phase follows. In this phase, the models are implemented and tested. Afterward, specific parts of the application are developed. Following the iterative process, new features are implemented into the next cycle. These features need to be analyzed and may change the domain model. In addition, this may lead to the establishment of new bounded contexts. Thus, the models, including the tactical diagrams, the context choreography, and the context map, are refined according to the features and the knowledge crunching process in the previous steps.

IV. Modeling of Context Map and Context Choreography

In Chapter III we defined the foundations of a context map and also introduced the context choreography as a new type of diagram. Both types of diagrams have to be modeled. Considering the fact, that DDD does not provide any modeling guideline, the appearance—or the syntax—of a context map is broadly diversified. But also the semantics varies from model to model. As a consequence, comprehensibility and interoperability of development teams is reduced. Therefore, this chapter discusses the feasibility of creating a UML profile for the context map and context choreography. It will be integrated in the modeling application Enterprise Architect (EA), although any other software modeling tool like ArgoUML could be used alike.

A. UML Diagram Type and Elements

The purposes of the context map and context choreography are different, but for both it is necessary to express relationships between bounded contexts. On the one hand, the context map displays the way how teams interact with each other. On the other hand, the context choreography shows, which information is shared between bounded contexts. Both models are essentially component diagrams and share the concept of bounded contexts and relationships, but other diagram elements differ.

1) Context Map Elements: The context map provides an overview of all bounded contexts and their relationships. It is important to use an expressive and at the same time limited syntax. This should keep the complexity manageable and still cover all important business and domain information. In order to meet these requirements, the model should contain at least three types of visual elements. The essential diagram elements are components, stereotypes and associations which are shown in 5. Unlike components and associations, stereotypes can not exist on their own. They are an extension for adding specific information to an element. Additionally the use of more modeling elements, as packages for grouping components into subdomains and notes for general annotations, are advised. Components are used as containers for artifacts, which are in case of the context map a domain model. To distinguish their purpose they are tagged with stereotypes either as a "bounded context", "foreign bounded context", an "anti corruption layer (ACL)" or as a "shared kernel" [22]. The purpose of bounded contexts are already explained in Section III-A1.

a) Foreign Bonded Context: In comparison to a bounded context the foreign bounded context should be used for any bounded context whose domain knowledge is not fully accessible; just like a blackbox. Therefore, this the foreign bounded context might share an domain object which does not suit the needs of a bounded context. Neither responsibilities of development teams nor the content can be affected for a foreign bounded context. Without further treatment the domain knowledge of a bounded context must be adapted to this domain object which can affect its design in a bad way. Therefore the ACL supports a bounded context by converting domain objects from such a foreign bounded context into the required form. It is important, that no domain knowledge is packed into an ACL, which should be in the bounded context itself.

b) Anti Corruption Layer: The anti corruption layer (ACL) is used by the downstream as an isolating layer to communicate with the upstream. Hereby, the ACL converts the domain objects according to the domain models of those two bounded contexts. This means few or no changes to the domain model of the downstream have to be made. As mentioned earlier, the ACL does not provide any domain knowledge itself. It could be seen as an additional layer.

c) Shared Kernel: The shared kernel is the last essential element for the context map. It is used to share a subset of domain objects among multiple bounded contexts [5]. A reference from a bounded context to a shared kernel means that the domain knowledge of the shared kernel is included in the bounded context. As soon as two bounded contexts refer to a shared kernel, further developments of it has to be coordinated between the regarding development teams. This also leads to a shared responsibility.

d) Relationships: Associations are used to connect bounded contexts regarding their organizational structure and dependencies to other development teams. There are two types of associations. Firstly directed ones to represent an upstream/downstream relationship where the arrow points in the direction of the downstream. Secondly there are undirected ones to express that they are equal. To make the associations more expressive and thus more valuable they are tagged with additional stereotypes. Directed associations can be tagged as "customer/supplier" and "confirmist", while undirected ones can be tagged as "partnership" [5].

e) Omitted Associations: Literature introduces more patterns than these described above, but in our case, we did not see any advantages for them in the context of designing a microservice architecture. One pattern is called "separate way", which is used when two bounded contexts should stay independent of each other despite one could integrate the other to fulfill its functionality. It should be applied when there is an expensive integration of the regarding bounded contexts. But having microservices in mind, each microservice is independent by definition [6].

Another pattern is the "published language". Here, the upstream shares a well-documented language which describes how the communication is done and how the domain objects are structured. Because we assume a microservice architecture, every bounded context must have an API specification, like OpenAPI. Hence it is not necessary to model it.

Finally an "open-host service" defines a protocol to give access to multiple subsystems as a set of services. This can be done globally by an API management tool which makes it unnecessary to model it at every bounded context.

2) Modeling example for a Context Map: A simple context map containing all previously defined elements can be seen in Figure 3. It should give a good impression of what a more complex context map would look like. The bounded contexts, subdomains and relationships mainly serve exemplary purposes, which is why their meaningfulness is of secondary importance. Note that the anti-corruption layer must always be part of a bounded context, which is depicted with the UML nesting connector. Furthermore, the conformist relationship between *StudentDirectory* and *StudentDirectoryACL* can be understood as a direct relationship between *StudentDirectory* and *Examination* where *StudentDirectoryACL* is just an abstraction layer. The remaining elements and connections should be rather self-explanatory after the explanations above.

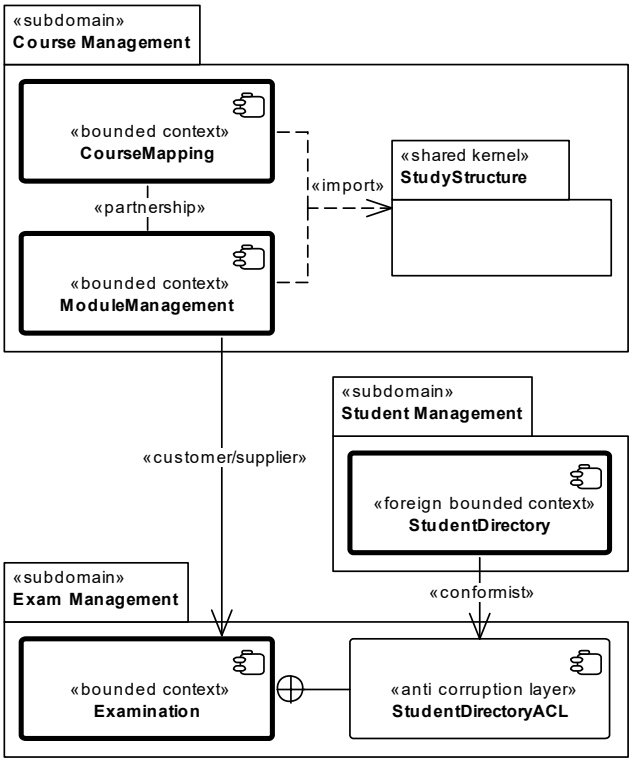

Figure 3. Example Context Map of the CampusManagement domain

3) Context Choreography: Modeling the context choreography differs from the context map and is way easier to represent. Its purpose is to display communications between bounded contexts and further, the domain object, which is transferred by this communication. Therefore, we need bounded contexts, two types of associations and a shared entity as a new element. The bounded context works the same as for the context map.

a) Shared Entity: The shared entity or "*shared thing*" as Newman refers to it [6] is a domain object that is used for exchanging information between two bounded contexts. Thus, it is provided by one bounded context and consumed by another. It contains attributes and their types. Due to reusability and interchangeability, attribute types should be limited to the use of basic types.

b) Event: An event represents an event that occurs within a process flow. Usually, an event is generated when a domain object (especially the entities) has been manipulated. For example, if a new profile of a student was created, an event "StudentProfileCreated" could be thrown. The event contains all the information required for further processing of the process flow.

c) Association: The associations for the context choreography are simple. One connects the bounded context, which publishes the domain object, with the shared entity. It is tagged with "shares". The second one is to connect the shared entity with the consuming bounded context. It is tagged with "uses". The associations "produces" and "comsumes" are required for the events. The relationship between the event-generated bounded context and the event is labeled "produces". The consuming bounded context has a "consumes" labeled relationship.

B. UML Profiles for Context Map and Context Choreography

In case of both models, the syntax while modeling them has to be limited. Possible are UML profiles or metamodels, but each has their own advantages and disadvantages. Our approach to create a UML profile based on the UML component diagram. Further, we integrated it into a modeling tool named Enterprise Architect.

1) Metamodel vs. UML profile: Basically there are two options to lever UML for custom use cases. A lightweight extension describes the process of creating a UML profile with corresponding stereotypes, whereas an adaptation of the UML metamodel, by adding or changing existing elements, is called a heavyweight extension [23]. Usually, a lightweight extension is the better option because the majority of use cases can be covered, while keeping the complexity low. Additionally, the created UML profile is portable and can be used across most enterprise modeling applications like Enterprise Architect via the standardized XML Metadata Interchange (XMI) format. XMI, like UML, is defined by the Object Management Group (OMG). This open standard is supported by many software modeling tools and its main purpose is exchanging models, expressible in Meta-Object Facility, including their metadata. Finally, when extending UML via profiles, the modeling elements do not have to be relearned, because the base elements are the same and generally well understood.

The advantage of a heavyweight modification would be the even greater flexibility in mapping the modeling problem. It can be adapted to the most complex modeling requirements, which could not be mapped with a UML profile in a clean way. The downside is, that a custom modeling language, based on UML, has to be created, that every project member has to learn and understand, which usually is a time-consuming task to do. In addition, your modeling software has to support the change of the underlying metamodel, which is far less common than UML profile support.

In conclusion, the creation of a UML is preferable to an extension of the metamodel due to the low added value. Context maps, context relation views and domain models are rather intuitive and straightforward adaptations of existing diagrams, which are easily modeled by extending standard UML diagrams with stereotypes.

2) Enterprise Architect and Profiles: Enterprise Architect (EA) is a software modeling tool that is based on OMG's UML [24]. The tool allows the integration of user-defined extensions which includes the required UML profiles. In addition, EA already provides useful profiles for modeling languages, such as Business Process Modeling Notation (BPMN). Custom UML profiles can be created through EAs so-called Model Driven Generation (MDG) technologies. The UML profiles provided in this article are extended versions of EA's standard UML profiles. To create a UML profile with optimal user experience, additional diagram and toolbox profiles are necessary, which are specific to the EA. These custom diagrams extend standard UML diagram metaclasses and define the appearance of the diagram elements. A toolbox shows these defined elements to the user. Figure 4 displays an overview of the toolbox for the context map. The user sees only the modeling elements defined by the custom UML profile, which simplifies the modeling process and reduces modeling inconsistencies. We use EA to model the context map and the context choreography.

Figure 4. Custom toolbox for the context map

3) UML Profile for the Context Map: The core elements of our Context Map can be seen in Figures 5 and 6 and are the same that were defined earlier. Note that the metaclasses are only references to the elements in UML and are therefore not created by us, but only used by the stereotypes as a basis. The *direction* attribute of the association metaclasses specifies the default direction of a newly created association, whereas the *compositionKind* attribute specifies whether the association is an aggregation or a composition. Additionally, the standard UML package import connector was added to the toolbox for denoting shared kernel usage and is therefore not shown in Figure 6. It is a directional dependency relationship with the stereotype « import », starting from the using bounded context to the shared kernel.

4) UML Profile for the Context Choreography: Figure 7 shows the profile for the context choreography diagram. The metaclasses and the *direction* attribute behave in the same way as in the context map. The *isActive* attribute on the class specifies whether the class can operate as an independent thread of behavior. Besides the depicted elements in the profile, the associated toolbox does not contain any further elements, since these should be created in the context map and imported afterward.

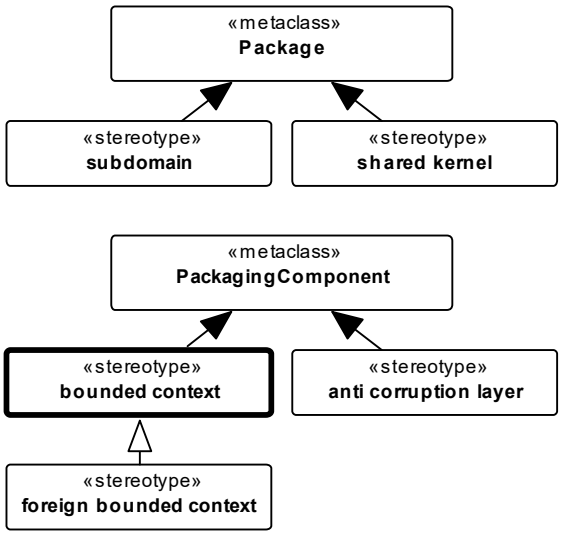

Figure 5. Core elements of the context map profile

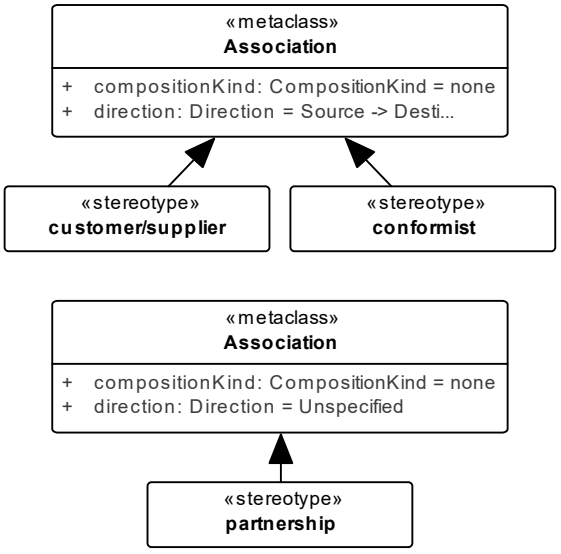

Figure 6. Excerpt of the available context map profile associations

V. Case Study: Campus Management

In our case study, we illustrate our approach of establishing a new bounded context and integrating it into an existing context map. The case study orientates itself on the process as described in Section III. Over three semesters, we expanded the campus management system of KIT with microservice-based applications. The experiment carried out can be understood as type I validation [25]. With a type 1 validation, the testing of the design process can be seen in a realistic project context. It is legitimate to conduct the experiment with students as test persons [26]. The case study represents our recent project in this field and adds a social media component to the campus management system.

A. Project Scope

Our vision is to simply and efficiently support the exchange of information and facilitate cooperation between students.

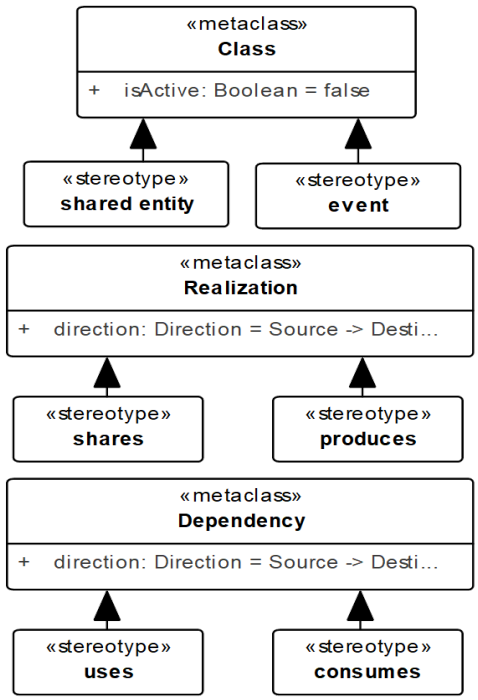

Figure 7. Context choreography profile

For this purpose, we wanted to introduce a profile service in the campus management system. This profile service should allow students to create custom profiles to display information about their studies, like currently visited lectures and future exams. The purpose of this service is to assist students with their studies and their search for study partners. For example, students can find study partners with the help of the profile service when other students share the lectures they attend.

B. Requirement Elicitation

In our process, we began by eliciting the requirements with BDD and formulating them as features. Fig. 8 presents one of the main features that enables students to edit their profile.

1. **Feature**: Providing student profiles
2. As a student
3. I can provide relevant information about myself
4. So that others can see my interests and study information

5. **Scenario**: Publish profile
6. **Given** I was never logged in to the ProfileService
7. **When** I log in to the ProfileService for the first time
8. **Then** my study account is linked to the ProfileService
9. **And** I choose which profile information I want to publish

Figure 8. Example of a BDD feature for publishing an user profile

C. Initial Domain Model

Analyzing the features leads to the initial domain model by deriving domain objects and their relationships. In our previously defined feature (see Section V-B), we identified, the terms “Profile,” “Examination,” and “Student” and added them

to the initial domain model. By repeating this procedure with all features, the domain model is enriched with the business logic. The result of the initial tactical diagram is presented in Fig. 9.

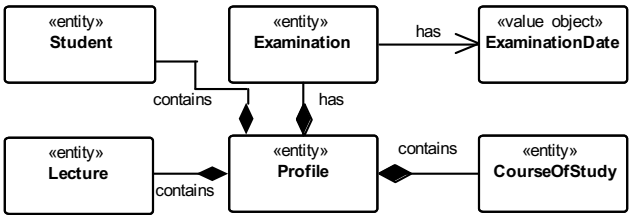

Figure 9. Initial domain model derived from BDD features and other sources

D. Bounded Contexts and Context Choreography

While we analyzed the domain, we also considered the existing context map of the campus management domain. We noticed that the bounded contexts "StudentManagement," "ExaminationManagement," "ModuleManagement," and "CourseMapping" already offered the required functionality. Only "ProfileManagement" had to be established as a new bounded context. Therefore, we considered the data exchange between the bounded contexts and created the context choreography on that basis. The result is illustrated in Fig. 10. The existing bounded contexts provide the required data as shared entities. Furthermore, the relevant events for the process flow were considered. Unfortunately, not every third-party application is designed for communication with events. Therefore we still had to use the web interfaces. The new bounded context

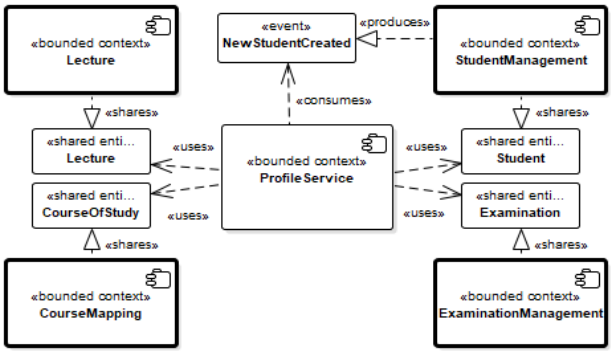

Figure 10. New bounded context "ProfileManagement" in a context choreography

"Profiles" adapts the shared entities and delivers the data required from the profile service. It also consumes the event "NewStudentCreated" of the "StudentManagement" Bounded Context. As soon as this event is consumed, a new profile based on the student can be created. Last, the microservices are choreographed in the BFF of the intended application, to achieve the required application logic.

E. Integrating in Context Map

After we had established the bounded contexts and the context choreography, we needed to add the profile management context to the context map. Therefore, we determined the dependencies and communication channels between bounded contexts based on the context choreography. We found our development team did not influence any other bounded context. Thus, we applied the conformist as communication pattern. As a result, the context map depicted in Figure 11 was enhanced with the "Profiles" context. Afterward, we started the first implementation cycle.

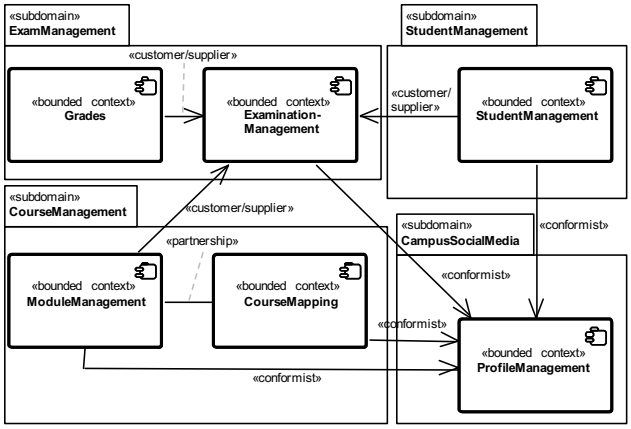

Figure 11. "ProfileManagement" context integrated into context map

F. Context Map as Template for a Deployment Map

The resulting context map provides an overview of the microservices that need to be deployed. Assuming that each bounded context represents one microservice, we can enhance the context map with technical information that is needed for the deployment in a secure manner. For instance, we can define which ports listen for incoming or are allowed for outgoing requests. By following such an approach, each microservice is initially considered in isolation. We enforce this by defining default policies on the execution environment that need to be taken into account during deployment. The enhancement with technical information is transferred into a new diagram called a deployment map. For the modeling, we use a UML deployment diagram. In addition to a general overview of the deployment, it can also be used for an upfront security audit. We are planning to present the derivation rules in an upcoming publication.

For testing purposes, we have used a hosted Kubernetes [27] cluster on a cloud provider. Kubernetes is an open source system for provisioning and management of container-based applications and aroused from the collected experience behind Omega and Borg. First of all, we have defined policies to deny all ingress and egress traffic to all running Pods by default. A Pod groups one or more containers and can be seen as the central brick of Kubernetes when deploying applications. Each bounded context will be represented by a microservice running in a container (Docker or rkt). Depending on their relation to each other (see TABLE I), we put them in corresponding Pods. Next, we have used the ports for incoming and outgoing traffic to derive the network policies. Finally, we have defined the services that wrap the Pods and offer a central access point for interaction. The application shows us that the underlying context map can be used as a basic scaffolding for deriving the deployment map but need to be

enhanced with technical information as well as information from the development teams that develop the microservices. For instance, a persistent storage is missing in a context map due to its domain orientation but is needed for the deployment map.

G. Organizing the artifacts in the Enterprise Architect

An essential part in software development is to retrieve design artifacts. To achieve this, a centralized repository is helpful. The EA innately provides such a centralized repository, so we were able to make use of it and easily retrieve our design artifacts. We placed each design artifact into a folder structure accordingly to our experiences from other projects [28]. The first artifact in the folder, as shown by figure 12 is

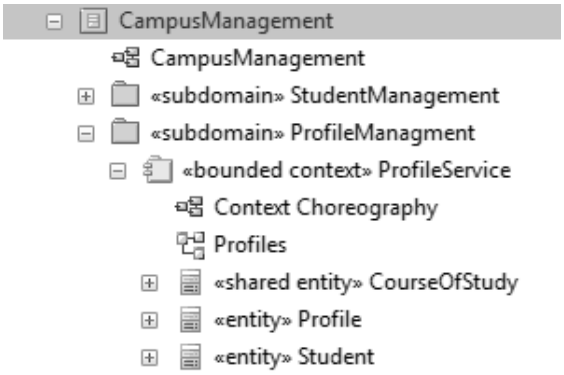

Figure 12. Repository structure of the campus management domain

the context map which has the same name as the repository. Folders of the different subdomains follow this artifact. Each subdomain folder can be expanded and contains the bounded contexts that belong to this subdomain. A bounded context itself contains the context choreography, the relation view and the domain objects which belong to the relation view.

VI. Related Work

During our research, we searched for works comparable to the context map and its purpose. We encountered several inspiring works regarding different aspects of the context map. Especially, the focus on the microservice architecture is an important part of this paper.

A. Microservice Architecture

A microservice should concentrate on the fulfilment of one task and should be small, so a team of five to seven developers can be responsible for the microservice's implementation [6]. A microservice itself is not an application, but rather a software building block [10]. In microservice architectures, applications are realized through choreography of these building blocks. A central aspect of microservice architecture is the autonomy of the single microservices [29]. Each microservice is developed and released independently to achieve continuous integration.

B. Approaches for Designing a Microservice Architecture

The objective of microservice architectures is to subdivide large components into smaller ones to reduce complexity and create more clarity in the single elements of the system [29]. In this article, we described our approach of designing microservice architectures with a context map from DDD. However, there are further strategies to identify microservices, which we considered in this article.

One possible approach is event storming, as introduced by Alberto Brandolini in the context of DDD [30]. Event storming is a workshop-based group technique to quickly determine the domain of a software program. The group starts with the workshop by "storming out" all domain events. A domain event covers every topic of interest to a domain expert. Afterward, the group adds the commands that cause these events. Then, the group detects aggregates, which accept commands and accomplish events, and begins to cluster them together into bounded contexts. Finally, the relationships between bounded contexts are considered to create a context map. Like our approach, this strategy is based on DDD and results in a context map displaying the bounded contexts. Instead of a workshop for exploring the domain and defining domain events, we develop our bounded contexts through an iterative analysis and design phase. Furthermore, we enhanced the context map with maintenance aspect for microservice discovery and dependencies between development teams. The purpose of the resulting context map from [30] is comparable to the context choreography. Both focusing on the interactions between bounded context and identify the transferred domain knowledge.

Another method for approaching a microservice architecture is described in [31]. First, required system operations and state variables are identified based on use case specifications of software requirements. System operations define public operations, which comprise the system's API, and state variables contain information that system operations read or write. The relationships between these systems operations and state variables are detected and then visualized as a graph. The visualization enables developers to build clusters of dense relationships, which are weakly connected to other clusters. Each cluster is considered a candidate for a microservice. This bears a resemblance to our approach because we also begin by focusing on the software requirements and take visualization for better understanding the domain.

A further widely used illustration of partitioning monolithic applications is a scaling cube, which uses a three-dimensional approach as described in [32]. Here, Y-axis scaling is important because it splits a monolithic application into a set of services. Each service implements a set of related functionalities. There are different ways to decompose the application which differ from our domain-driven approach. One approach is to use verb-based decomposition and define services that implement a single use case. The other possibility is to partition the application by nouns and establish services liable for all operations related to a specific entity. An application might use a combination of verb-based and noun-based decomposition. X-axis and Z-axis regards the operation of the microservices. The X-axis describes the horizontal scaling which means cloning and load balancing the same microservice into multiple instances. Meanwhile, the Z-axis denotes the degree of data separation. Both axis are important for microservice architectures and currently omitted in our context map approach.

C. Software Development Approaches

The development process we apply is based on BDD and DDD. As a method of agile software development, BDD

should specify a software system by referencing its behavior. The basic artifact of BDD is the feature, which describes a functionality of the application. The use of natural language and predefined keywords allows the developer to define features directly with the customer [33]. During our analysis phase, we used BDD for requirement elicitation.
In our design phase, we applied DDD based on the features we defined with BDD. DDD's main focus is the domain and the domain's functionality, rather than technical aspects [16]. The central design artifact is the domain model, which represents the target domain. In his book *Domain Driven Design - Tackling Complexity in the Heart of Software*, Eric Evans describes patterns, principles, and activities that support the process of software development [4]. Although DDD is not tied to a specific software development process, it is orientated toward agile software development. In particular, DDD requires iterative software development and a close collaboration between developers and domain experts.

D. Application of the Context Map

The goal of a context map, which Evans describes as a main activity of DDD, is to structure the target domain [4]. For this purpose, the domain is classified into subdomains, and in those subdomains, the boarders and interfaces of possible bounded contexts are defined. A bounded context is a candidate for a microservice, and one team is responsible for its development and operations [6]. Moreover, the relationships between bounded contexts are defined in a context map. Both the technical relationships and the organizational dependencies between different teams are considered.
A further aspect of the context map involves the maintenance of the miscroservice architecture. Without managing the application (or service) landscape, existing microservices are not used, even if they provide needed domain knowledge. The usage of a context map helps concepts like humane registry or API management products which tries to achieve maintenance goals. Martin Fowler introduced humane registry as a place, where both developers and users can record information about a particular service in a wiki [8]. In addition, some information can be collected automatically, e.g., by evaluating data from source code control and issue tracking systems. API management products like "apigee" [9] reach maintenance by pre-defining API guidelines such as key validation, quota management, transformation, authorization, and access control.

E. Unified Modeling Language Profiles in the Context of Domain-Driven Design

How to model the artifacts of the DDD is a known problem. Depending on the experiences and the understanding of the modeler, the appearances of the domain model differs. This poses a problem for the inter-team communication, due to the fact, that each team possibly interprets modeled concepts different.

Based on the inter-communication problem, [34] provides a survey on DDD modeling elements together with an initial UML profile for the domain model. Their survey is based on the examples shown in Evans DDD book [4]. It shows that the most examples are based on informal UML diagrams, which prevents automatically validation, transformation and code generation. The UML profile is applied in the context of microservice architectures with the goal of deriving code for microservices.

We see [34] as an important work for the application of DDD. Their results inspired us to also create an UML profile for the context map and context choreography.

VII. Limitations and Conclusion

The concepts we provide in this article have some limitations. These are addressed in the next section. Afterward, we provide a short conclusion discussing our results.

A. Limitations

Domain-driven design has no special application or architectural style in mind. The concepts should be applied to DDD's layered architecture but could be applied to different architectural styles. For a better fit while developing microservice-based applications, we always had the microservice architecture in mind. Therefore, our provided concepts are only valid when developing a microservice-based application.

The concepts provided by this article are built from our experiences which we gathered in various projects. Most of our projects were in the academic branch, but we also worked with industrial partners. For the context map, we developed and proved our concepts over a longer time. Especially the provided UML profiles reflects our needs from the mentioned projects. It is likely, that the shown profiles are incomplete and need to be adjusted for further projects. The case study describes our last project. Project members and partners gave us useful feedback about the concepts when they applied them. In addition, the feedback also included points we had not yet addressed, like a modeling language for context mapping. Nevertheless, evidence of our concepts in large microservice architectures, such as 50 or more microservices, still lacks. Our goal is to obtain prove for large microservice architectures in such projects.

Another limitation of our concept is that we only applied them in "clean" microservice architectures. However, in the real world, there are also legacy applications in the application landscapes of organizations. Typically, a legacy application is not a microservice-based one; often, it is a monolithic architecture. In future work, we must determine how to integrate legacy applications into the context choreography and context map.

B. Conclusion

During our research, we found many different studies that consider model-driven approaches for developing microservices. Using these approaches for microservices is common. In domain-driven design, especially, the approaches that focus on the development itself but omit the design and maintenance phases. Therefore, we wanted to provide details on the design and maintenance of a microservice architecture using DDD's context map.

The context map has great potential to aid in developing and maintaining applications and is more useful when considering a microservice architecture. However, the context map shares a problem with most DDD concepts: its lacking placement in software engineering, foundations and concrete guidelines. Therefore, we first provided placement for the context map. Next, we clarified its foundations with a focus

on the bounded context, the main concept of the context map. It needs to be emphasized that the context map is not only showing microservices and therefore the microservice architecture. Communication paths and dependencies between development teams are also considered in the context map. After the foundations were clear, we could develop a systematic approach for creating the context map. This approach began in the analysis phase with an initial domain model, separating the domain knowledge into bounded contexts, stating relationships between them, and putting the bounded contexts into a context map. The separation of bounded contexts and their relationships are stated in our new diagram, the "context choreography." This diagram's purpose is to illustrate necessary bounded contexts for microservice-based applications. Furthermore, we created UML profiles for the context map and context choreography to unify their appearances. With those profiles, we especially tackle the various interpretations of the context map and support the inter-team communication.

This article's contributions are the first step in making use of the context map, and the newly defined context choreography more efficient.

ACKNOWLEDGMENT

We want to give special thanks to Iris Landerer, Chris Irrgang and Tobias Hülsken for always providing their opinions and useful feedback on our concepts. Furthermore, we would like to thank the following development team, which provided the results in our case study: Alessa Radkohl, Nico Peter, and Stefan Throner.

REFERENCES

[1] B. Hippchen, M. Schneider, I. Landerer, P. Giessler, and S. Abeck, "Methodology for splitting business capabilities into a microservice architecture: Design and maintenance using a domain-driven approach," *The Fifth International Conference on Advances and Trends in Software Engineering*, 2019.

[2] M. Gebhart, P. Giessler, and S. Abeck, "Challenges of the Digital Transformation in Software Engineering," *ICSEA 2016*, p. 149, 2016.

[3] C. Richardson, "Pattern: Messaging," https://microservices.io/patterns/communication-style/messaging.html [retrieved: 11, 2019].

[4] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.

[5] V. Vernon, Ed., *Implementing Domain-Driven Design*. Addison-Wesley, 2013.

[6] S. Newman, *Building Microservices: Designing Fine-grained Systems*. " O'Reilly Media, Inc.", 2015.

[7] M. E. Conway, "How do Committees Invent," *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[8] M. Fowler, "HumaneRegistry," URL: https://martinfowler.com/bliki/HumaneRegistry.html [retrieved: 08, 2019].

[9] Google, "apigee, API management," https://apigee.com/api-management/ [retrieved: 08, 2019].

[10] B. Hippchen, P. Giessler, R. Steinegger, M. Schneider, and S. Abeck, "Designing Microservice-Based Applications by Using a Domain-Driven Design Approach," in *International Journal on Advances in Software, Vol. 10, No. 3&4, pp. 432–445*, 2017.

[11] M. Bruce and P. A. Pereira, *Microservices in Action*. Manning Publications, 2019.

[12] Pivotal, "Rabbitmq," https://www.rabbitmq.com [retrieved: 11, 2019].

[13] Apache, "Kafka," https://kafka.apache.org [retrieved: 11, 2019].

[14] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering Using UML, Patterns and Java-(Required)*. Prentice Hall, 2004.

[15] O. Vogel, I. Arnold, A. Chughtai, and T. Kehrer, *Software Architecture: A Comprehensive Framework and Guide for Practitioners*. Springer Science & Business Media, 2011.

[16] S. Millett, *Patterns, Principles and Practices of Domain-Driven Design*. John Wiley & Sons, 2015.

[17] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[18] L. Chen, "Continuous delivery: Overcoming adoption challenges," *Journal of Systems and Software*, vol. 128, pp. 72–86, 2017.

[19] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[20] L. Chen, "Microservices: Architecting for continuous delivery and devops," in *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018, pp. 39–397.

[21] J. F. Smart, *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Manning, 2015.

[22] E. Evans, *Domain-Driven Design Reference: Definitions and Pattern Summaries*. Dog Ear Publishing, 2014.

[23] J. Osis and U. Donins, *Topological UML Modeling: An Improved Approach for Domain Modeling and Software Development*, ser. Computer Science Reviews and Trends. Elsevier Science, 2017.

[24] O. OMG, "Unified Modeling Language (OMG UML)," *Superstructure*, 2007.

[25] Z. Durdik, *Architectural Design Decision Documentation Through Reuse of Design Patterns*. KIT Scientific Publishing, 2016, vol. 14.

[26] W. F. Tichy, "Hints for reviewing empirical work in software engineering," *Empirical Software Engineering*, vol. 5, no. 4, pp. 309–312, 2000.

[27] K. Hightower, B. Burns, and J. Beda, *Kubernetes: Up and Running: Dive Into the Future of Infrastructure*. O'Reilly Media, 2017.

[28] M. Schneider, B. Hippchen, P. Giessler, C. Irrgang, and S. Abeck, "Microservice development based on tool-supported domain modeling," *The Fifth International Conference on Advances and Trends in Software Engineering*, 2019.

[29] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. " O'Reilly Media, Inc.", 2016.

[30] A. Brandolini, "Introducing Event Storming," *blog, Ziobrando's Lair*, vol. 18, 2013, URL: http://ziobrando.blogspot.com/2013/11/introducing-event-storming.html [retrieved: 0, 2019].

[31] S. Tyszberowicz, R. Heinrich, B. Liu, and Z. Liu, "Identifying Microservices Using Functional Decomposition," pp. 50–65, 2018.

[32] N. Dmitry and S.-S. Manfred, "On Micro-Services Architecture," *International Journal of Open Information Technologies*, vol. 2, no. 9, 2014.

[33] M. Wynne, A. Hellesoy, and S. Tooke, *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2017.

[34] F. Rademacher, S. Sachweh, and A. Zündorf, "Towards a uml profile for domain-driven design of microservice architectures," in *International Conference on Software Engineering and Formal Methods*. Springer, 2017, pp. 230–245.