Overview of a Domain-Driven Design Approach to Build Microservice-Based Applications

Roland H. Steinegger, Pascal Giessler, Benjamin Hippchen and Sebastian Abeck

Research Group Cooperation & Management (C&M)
Karlsruhe Institute of Technology (KIT)
Zirkel 2, 76131 Karlsruhe, Germany
Email: (steinegger | pascal.giessler | abeck)@kit.edu, benjamin.hippchen@student.kit.edu

***Abstract*—The current trend of building web applications using microservice architectures is based on the domain-driven design (DDD) concept, as described by Evans. Among practitioners, DDD is a widely accepted approach to building applications. Applying and extending the concepts and tasks of DDD is challenging because it lacks a software development process description and classification within existing software development process approaches. For these reasons, we provide a brief overview of a DDD-based software development process for building resource-oriented microservices that takes into consideration the requirements of the desired application. Following the widely accepted engineering approach suggested by Brügge et al., the emphasis is on the analysis, design, implementation and testing phases. Furthermore, we classify DDD and microservice-based application into regular software development activities and software architecture concepts. After the process is described, it is applied to a case study in order to demonstrate its potential applications and limitations.**

Keywords–Domain-driven design, API, resource-orientation, domain model, software development process, microservices, backend-for-frontend

I. INTRODUCTION

Over the past few years, microservice architectures have evolved into a popular method for building multiplatform applications. A well-known example is Netflix, who offers applications for a number of platforms, including mobile devices, smart TVs and gaming consoles [1]. Service-oriented architectures are the foundation of microservice architectures, as microservices have special properties [2]. A microservice is autonomous and provides a limited set of (business) functions. In service-oriented architectures, designing services and selecting boundaries is a key problem.

The traditional approach, as discussed by Erl [3], suggests a technical and functional separation of services. In contrast, according to Evans [4], domain-driven design (DDD) provides the key concepts required to compartmentalize microservices [1]. The DDD approach provides a means of representing the real world in the architecture, e.g., by using bounded contexts representing organizational units [5], and also identifies and focuses on the core domain; both of these characteristics lead to improved software architecture quality [6]. In microservice architectures, these bounded contexts are used to arrange and identify microservices [1]. Using DDD is a key success factor in building microservice-based applications [1].

When applying DDD to the development of microservice-based applications, several problems may arise, depending on the level of experience of the development team. Domain-driven design offers principles, patterns, activities and examples of how to build a domain model, which is its core artifact. However, it neither provides a detailed and systematic development process for applying these principles and patterns nor does it classify them into the field of software engineering. Classifying the activities, introduced by DDD, into the activities of a software development process could improve the applicability. Futher, the classification of the patterns and principles into software architecture concepts, such as architecture perspectives and architecture requirements, supports software architects in desiging microservice architectures.

In addition, there are no clear proceeding regarding how to derive the necessary web application programming interfaces (web APIs) that act as a service contract between microservices and the application. The importance of a service contract is described by Erl [3]. From the business perspective, the web APIs also have strategic value; therefore, they must be designed in manner that emphasizes quality [7].

Furthermore, applications and, in particular, user interfaces, are often not considered or only considered superficially during the process of designing service-oriented architectures [1] [3]. However, the application can play a major role when building the underlying microservices. Domain-driven design emphasizes that the application is necessary to determine the underlying domain logic of microservices; the user interface is important to consider when designing specific web APIs for the UI when using the backends for frontends (BFF) pattern [1]. When designing microservices within the software-as-a-service (SaaS) context, there is no graphical user interface; instead, there is a technical one. The target group shifts from end users to external companies or independent developers who can benefit from the capabilities of the service offered. For this reason, a web API has to be designed in such a manner that it can map as many possible use cases for a particular domain as possible. The resulting set of use cases represents the requirements that must be handled by the web API and the microservices.

We experienced these challenges when establishing a software development process based on DDD to build SmartCampus, a service-oriented web application. During the process we could not find literature that addressed these problems. Thus, we classify DDD activities within the field of software engineering, arrange the components of a microservice-based application according to the layers of DDD and describe the ac-

 ISBN: 978-1-61208-553-1

tivities necessary in building microservice-based applications. We apply these activities in an agile software development process used to build parts of the SmartCampus application and discuss both the results and limitations.

This article is structured as follows: In Section II, DDD and microservice architecture, including a general introduction to software architecture and development and other related concepts, are introduced. Section III classifies DDD and microservices and introduces the software development activities required in building microservice-based applications according to the requirements of DDD. In the next section, a case study demonstrates the application of these activities within a software development process, including artifacts. The limitations discovered while applying the activities are described in Section V. A conclusion regarding the activities and possible future areas of inquiry is presented in Section VI.

II. Foundation and Related Work

This section provides an overview of model-driven engineering (an approach that is similar to DDD), DDD itself, traditional software engineering activities (which are used to classify DDD activites), software architecture in general (as the foundation being the foundation for classifying microservice architecture) and microservice architecture.

A. Model-Driven Engineering

Douglas C. Schmidt [8] describes Model-Driven Engineering (MDE) as an approach that is used to effectively express domains in models. The Object Management Group (OMG) introduced their framework model-driven architecture (MDA) [9] to support the implementation of MDE. MDA identifies three steps necessary in moving from the abstract design to the implementation of an application. Three models are created by carrying out these steps: 1) computation independent model (CIM) provides domain concepts without taking technological aspects into consideration, 2) platform independent model (PIM) enriches the CIM with computational aspects; and 3) platform specific model (PSM) enriches the PIM with the aspects of implementation that are specific to a particular technological platform.

B. Software Engineering Activities and Domain-Driven Design

Brügge et al. [10] describe a widely accepted software engineering approach in the context of object-orientation. We use their concepts to classify the activities we identified to build microservice-based applications using DDD. This object-oriented approach works well when small teams build applications that range over few domains implemented. [10] offers an overview of the activities that take place during software development: requirements elicitation, analysis, systems design, object design, implementation, and testing. (These activites are discussed further in the article's introduction of the development activities.)

Domain-driven design is an approach that is used in application development where the domain model is the central artifact. Eric Evans introduced this approach in the book Domain-Driven Design and identified the essential principles, activities and patterns required when using DDD [4].

A domain model that conforms to Evans' DDD approach contains everything that necessary to understand the domain [4]. This approach goes beyond the traditional understanding of a domain model, which is connected to a formalized model using the unified modeling language (UML) [11]. To distinguish between the two concepts, following Fairbanks [12], we use the term information model which corresponds to a computation independent model (CIM). It is a part of the domain model and consists of concepts, relationships and constraints. In order to support downstream implementation, Evans adds implementation specific details to the model. The resulting domain model corresponds to a PIM. In Evans' approach to DDD, the central principle is to align the intended application with the domain model. The domain model shapes the ubiquitous language that is used among the team members and functions as a tool used to achieve this goal.

C. Microservice Architectures

Vogel et al. provide a comprehensive framework for the area of software architecture [13], which is used to classify microservices and DDD. Their architecture framework has six dimensions: 1) architectures and architecture disciplines, 2) architecture perspectives, 3) architecture requirements, 4) architecture means, 5) organizations and individuals and 6) architecture methods. The essential terms used in describing an architecture are: systems, which consist of software and hardware building blocks; a software building block can be a functional, technical or platform building block. Building blocks can also consist of other building blocks and may require them. The authors also introduce the concept of architecture views; their definition is influenced by the IEEE [14]. Architecture views are part of the documentation that describes the architecture. Architecture views are motivated by stakeholders' concerns. These concerns specify the viewpoint on the architecture and, thus, specify the views.

Newman provides a comprehensive overview of microservices and related topics from and industry perspective [1]. He defines a microservice as a "small, autonomous service" that does one thing well; and adds that the term "small" is difficult to define. In contrast to services in a service-oriented architecture according to Erl [3], the single purpose principle results in microservices having similar sizes within an architecture [2]. Two mapping studies regarding microservices and microservice architecture reveal that a gap in the literature regarding these topics exists [15] [16]. (Further relevant information is discussed during the section of this article that classifies microservice architectures.)

III. Process

This section classifies the activities involved in of DDD and concepts related to microservice architectures; furthermore, the software development activities involved in building microservice-based applications using DDD are introduced. The activities discussed can be applied to various software process models. However, DDD requires one to continuously question and adapt one's understanding of the domain. Thus, agile software development processes are most suitable.

A. Classification

We identify specifications, that are missing when just applying DDD to build a microservice-based application, by classifying DDD and microservice architecture using the software architecture concepts of Vogel et al. [13]. We divide the

 ISBN: 978-1-61208-553-1

classification process into two parts: first, we discuss the architecture perspective and second the architecture requirements.

Concerning the architecture perspective, software architecture can be divided into macro- and micro-architecture; it can further be divided into organization, system and building block level. The organization and system levels form the macro-architecture whereas the building block level can be assigned either to the macro or micro-architecture depending on what is required for the concrete architecture. [13]

Despite their names, microservice architecture and the domain model describe the macro-architecture. A microservice is a functional or technical software building block that require a platform to run on. Neither DDD nor microservices limit the underlying platform. When using DDD, microservices are structured according to the organizational units using bounded contexts from the domain model [1] [17]. The domain objects within a bounded context specify the core architecture of a microservice.

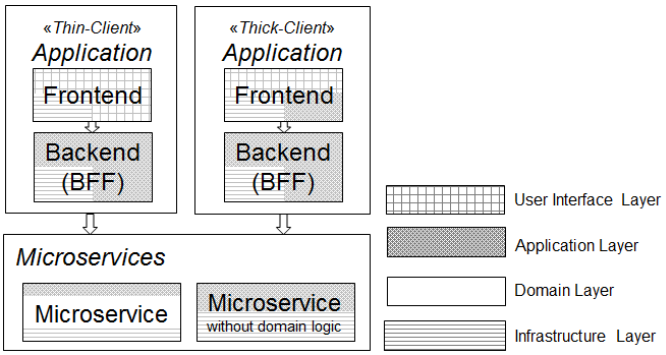

Figure 1. Software building blocks and their layers in a microservice-based application

Domain-driven design requires a layered architecture to separate the domain from other concerns [4]. Evans suggests a four layered architecture, consisting of the user interface, application, domain and infrastructure layers. Figure 1 shows the distribution of these layers among the software building blocks of microservice-based applications. On the highest abstraction level, microservice-based applications can be divided into applications and microservices. The application consists of a frontend, which is either thin or thick (meaning that it is with or without application logic), and its backend, which provides the application logic. The backend uses the microservices to access the domain layer or general infrastructure functionality. Each microservice has an application layer on top. The application layer translates requests into either the domain or infrastructure layers.Infrastructure logic *may* be part of each software building block. In our approach, we applied the layer distribution following Miller's approach [18].

In a layered architecture, higher layers can communicate with lower layers. Figure 2 depicts the layered architecture's communication process applied to the above-mentioned software building blocks [18]. The frontend should not directly call the microservices; we emphasize this by using dashed arrows.

Concerning architecture requirements, the decision to build microservice-based applications is taken at the organizational level (see the classification of service-oriented applications in [13]). Along with a microservice architecture, the organization

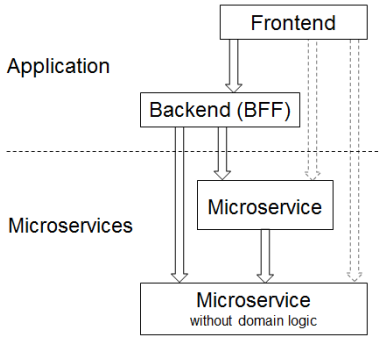

Figure 2. Communication between components

should choose a protocol that allows all of the microservices within the organization to communicate; e.g., using representational state transfer (REST) over hypertext transfer protocol (HTTP) with a set of guidelines or an event bus. The platform running the microservices (e.g., docker), the database technologies, the implementation of identity and access management etc. *might* also be organizational requirements; when building a microservice architecture the software architects have to decide, whether or not these concerns should be homogenous. We could not identify any requirements concerning the system or building block levels that are based on DDD or the microservice approach.

Some specification is still missing. The domain model specifies the functional view on the domain but does not consider technical aspects [4]. Thus, in addition to the domain model, there is a need for artifacts that describe the microservice architecture, including technical microservices and platform architecture. Furthermore, assuming that the domain model describes the architecture of the domain layer, the user interface, application, and infrastructure layer are not specified. Translating this into the context of the software building blocks, the frontend and backend may require specification. The decision to add further artifacts could be based on the risks involved in the application, as discussed by Fairbanks [12]. In our activities, we decided to add a user interface (UI)/user experience (UX) design, which specifies both the user interface and the user's interaction. Thus, this artifact specifies the frontend and backend.

B. Activity Overview

Next, we introduce the activities involved in building microservice-based applications. These activities facilitate the development of applications within similar domains. We align our activities with the traditional software development activities described by Brügge et al. [10]. Therefore, the activities end after testing, and we do not discuss deployment and/or maintenance. Figure 3 depicts the three activities and their interrelations: *requirements elicitation and analysis*, *design* and *implementation and testing*.

During the *requirements elicitation and analysis*, two subactivities take place: first, the information model, as part of the domain model, is created by "crunching knowledge" with domain experts; second, a prototype is designed and is discussed with both the user and customer. As both activities are closely related (when discussing prototypes, the knowledge of the domain gets deeper, and when discovering the information

 ISBN: 978-1-61208-553-1

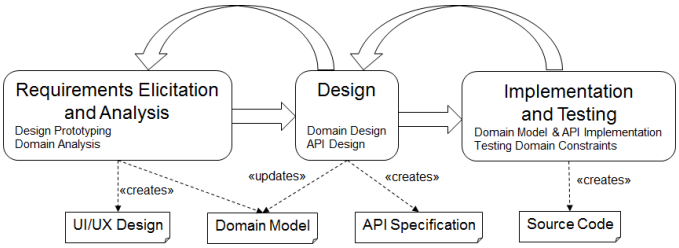

Figure 3. Overview of the activities used in building microservice-based applications

model, terms or workflows might change), we combined them into a single activity.

The *design* is comprised of the sub-activities involved in designing the domain and the APIs of the microservices. Based on the UI/UX design and further discussions with the user, the information model is refined, e.g., design decisions are made, and design patterns are applied. Domain design is comparable to the system design activity discussed by Brügge et al. [10]. The system is divided into subsystems that, according to Conway's Law [5], can be realized by individual teams using bounded contexts. Domain design results in a domain model that must be bound to the implementation artifacts. As the microservices offer access to the domain model and translate from the application layer to the domain layer, both the UI/UX design (representing the user interface layer and the application layer) as well as the domain model (representing the domain layer of DDD) is used to design the web APIs of the microservices. If using a BFF, its web API is designed, too. This activity can be assigned to the object design activity discussed by Brügge et al. [10].

After this preliminary work, the microservices are *implemented and tested*. The web APIs describe the microservices' entry points. These entry points and their application logic are implemented and tested, as the microservices' domain model. The constraints defined in the domain model, such as multiplicities or directed associations, are sources for domain tests.

Evans states that developing a "deep model" with which to facilitate software development requires "exploration and experimentation" [4]. Thus, software developers have to be open-minded to gain insights into the domain across the whole software development activities. This knowledge probably leads to changes in artifacts created during the previous activities. Therefore, iterations and jumping back to previous phases is possible in each phase. To be more clear, it is common to switch between phases and activities. Of course, experienced developers may do fewer mistakes and discover insights earlier, but hidden knowledge and misunderstandings are common. In the next sections, the phases are explained in more detail.

C. Requirements Elicitation and Analysis

The first activity is about understanding the needs of the user. Two non-chronological ordered activities take place in this phase: exploration of the domain and designing a prototype. These activities highly influence each other, e.g., the terms from the domain model are used in the prototype while new insights might change them. We see a strong binding between the origination process of the domain model and design prototyping, due to the missing specifications that are not captured during domain modeling. Every domain concept displayed on the design prototype has to be modeled in the domain model and vice versa. Small iterations within the analysis are needed in order to validate that both artifacts are consistent.

1) Domain Analysis: Exploring the Domain with DDD: Without a complete understanding, building satisfying applications is getting hard. In our presented approach, we focus on Evans book "Domain-Driven Design: Tackling Complexity in the Heart of Software" (DDD) to understand the needs and, thus, the domain through modeling [4]. Creating a comprehensive domain model in this phase needs experienced domain modelers to gain knowledge. After this step, we have a domain model that is equal to an information model (see Section II-B). Unified Modeling Language (UML) class diagram syntax is used to describe concepts and their relationships, constraints, etc. [19] [12].

According to DDD, collaboration with customers is essential to explore and particularly model the domain. So the first and recurring step of DDD is Knowledge Crunching [4]. Simultaneously to discussions with customers, the development team carries out the modeling activity and creates the domain model step by step. By following the pattern *Hands-On Modelers*, every team member involved in the software development process should also be part of the domain modeling to increase creativity [4]. In addition, a *Ubiquitous Language* will be established, which is the cross-team language. The origination process of the domain model is highly influenced by exploration and experimentation [4]. It is far better if a not completely satisfying model is going to implementation, than to refine the domain model over and over again without risking the implementation [4]. Creating the domain model under influence of DDD, makes it an iterative activity and fitting to principles from agile development processes, such as short time to market.

Complex domains automatically lead to a complex domain model. This complexity makes it hard for readers to understand the domain model. Due to that fact, it is necessary to split the model into multiple diagrams [18], which enables the modeler to model different aspects of the domain. Dynamic behavior, such as workflows, are relevant concepts of the domain. We adapt the view approach from software architecture [13] and introduced a concept named *domain views* to model different behaviors. We have created various types of domain views, such as an interactional view. They are motivated by a stakeholder with an special concern, too. During knowledge crunching, this predefinition makes it easy to choose the right person to discuss with.

The result of exploring the domain is a domain model, which contains relevant concepts of the domain, also called the domain knowledge [4]. DDD emphasizes this as focusing on the core domain that is relevant for the downstream implementation of the application [4].

2) Design Prototyping: By knowledge crunching, we get a complete understanding of the considered domain. The application requirements are use case specific and indicators for domain logic that has to be modeled in the domain model accordingly. Each identified use case based on the discussion

with the stakeholders will be represented as part of a so-called design prototype. A prototype is an efficient way for trying out new design concepts and determine their efficiency [20]. The design prototype is a specialization and focuses on the UI and the UX of the application. Since the customer primarily interacts with the UI, it is also an ideal artifact for further discussions with customers along the domain model. Further benefits by using a prototype can be found in [20]. Similar to knowledge crunching, design prototyping is an iterative activity. Each iteration consists of a brain storming regarding design ideas with respect to given boundary conditions, realization of the previously chosen design ideas, presentation and review of the resulting design prototype. The feedback from the customer as part of the review will be collected and analyzed to derive the necessary design changes for the next iteration. The design prototyping is finished when the prototype represents all of the customer needs.

D. Design Phase

Two activities take place during the design phase: Domain and API Design. These activities require the domain model and the UI/UX design created during the previous phase. After the design phase, the domain model as well as the API specification are ready to be implemented.

1) Domain Design: From Computational to Platform Independent Model (PIM): An important idea of DDD is the binding of the domain to the implementation [4]. The domain model is the core artifact to achieve this goal in the domain layer. During the analysis phase a computational independent model, the information model as part of the domain model, is created. Now, first, this model is separated into bounded contexts and, second, these bounded contexts are extended and refined, e.g., by applying design patterns to fulfill application requirements. These activities are based on examples of Evans and Vaughn [4] [17].

The organizational structure is used to decompose the information model into bounded contexts. The task requires experience and several iterations due to its importance [17], [21]. The decomposition is tightly coupled to the division of the development teams, each working on a bounded context [1]. Thus, intermediate results are discussed with the domain experts and other team members. The result is a context map, showing the relations of the bounded contexts.

The next steps are mainly carried out by the development team that is responsible for each bounded context. The goal of the next activity is to refine and extend the domain model according to the requirements of the applications. The UI/UX design is the main source for the application requirements.

Probably, the domain objects in the information model are already marked with stereotypes indicating their type, i.e., aggregate root, value object, entity or domain event. Even some services might be identified during the analysis phase. Domain objects missing a stereotype should be treated first; a stereotype should be added. Next, the design patterns repository, factory and domain service are added according to the requirements. For example, if there is functionality needing to display a domain object in the UI, a repository is added, or if there is a complex aggregate root, a factory might be added [4]. During the whole design process, the domain experts and other sources of information are involved (continuous knowledge crunching). After applying the design patterns, the domain model is ready to get implemented.

2) API Design: Deriving the Web API from PIM: Microservices expose their implemented business functionality via web APIs [1]. A web API can be seen as a specialization of an traditional API, which is why, we extend the definition by Gebhart et. al a bit further: “a contract prescribing how to interact with the underlying system [over the Web],” [22, p. 139]. From business perspective, a web API can be seen as a highly valuable business asset [7], [23] that can also serve as a solution for digital transformation [22].

A web API can be used for composing microservices to map a complex business workflow onto the area of microservices or offering business functionality for third-party developers [22]. To facilitate the reuse and discovery of existing functionality in form of microservices, the exposed web APIs have to be designed with care. According to Newman [1], Jacobsen [23] and Mulloy [24], web APIs should adhere to the following informal quality criteria: 1) Easy to understand, learn and use from a service consumer point of view, 2) Abstracted from a specific technology, 3) Consistent in look and feel and 4) Robust in terms of its evolution.

To overcome these challenges, we have to form a systematic approach on how to derive the web API from the underlying domain model. First, we have made the decision to build web APIs in a resource-oriented manner that can be positioned on the second level of the Richardson Maturity Model [25]. We do not pursue the hypermedia approach by Fielding [26] to reduce the complexity when building microservice-based applications. Second, we have identified resources and sub resources from the underlying PIM by looking at the relationship between the domain objects. Third, we have derived the required HTTP methods as well as their request and response representations from the interactional view as one of the mentioned domain views (see Section III-C1). Besides this, we have also developed a set of guidelines to support architects and developers by fulfilling the previously described informal criteria web APIs. These guidelines were derived from existing best practices by designing resource oriented web APIs [27]. The result of this design work is finally structured according to OpenAPI specification, which has the goal to “define a standard, language-agnostic interface to REST APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection.,” [28].

3) API Design: Deriving the Web API for BFF from Design Prototyp: A BFF is a common pattern to avoid so-called chatty APIs [1]. Chatty APIs often result in a huge amount of requests for the service consumer to get the needed information [24, p. 30f]. This is mainly due the fact that the needed domain information or logic is spread over multiple microservices and primarily designed for reusability rather than a specific use case in form of a concrete application. Besides, BFFs allow a development team to focus on the UI and UX specific requirements of an application by not restricting them on the exposed web APIs of the microservices. Additional and necessary application logic, such as data transformation, caching or orchestration can be implemented on the BFF level or application layer according to DDD [4]. That is why, the BFF can be seen as part of the UI [1].

 ISBN: 978-1-61208-553-1

In our approach, the UI and UX specific requirements are represented through a design prototype as a result of a conducted analysis phase (see Section III-C2). Similar to Section III-D2, we have decided to go with a resource-oriented style for the BFF web API and applied the same web API guidelines. Other solutions such as a method-oriented approach is also possible. For deriving the web API, we are looking at each view regarding the represented information as well as the interaction elements that cause data manipulations. This allows us to build resources, their representations as well as their needed operations. The resulting web API is highly coupled with the UI and now needs to be connected with the underlying domain represented by microservices. Since the domain model, as well as the design prototype are designed by using the *Ubiquitous Language*, the required microservices can be identified with minimal effort and orchestrated on the application layer to fulfill the requirements specified by the derived BFF web API.

E. Implementation and Testing

The domain model and web API specification enable the development team to implement the application. In this section, the implementation and testing of the microservices is introduced. We do not discuss the implementation of the UI/UX design, as we focus on DDD and building microservices. But, the implementation and testing of the BFF, being the connection between front end and microservices, is discussed.

First, we focus on implementing and testing the microservices. Each bounded context is implemented as an microservice using the specified API. A development project, e.g., a maven project including source code, is created and pushed into the version control repository. We recommend to offer the API specification as part of the microservice. It is added to the repository and delivered through its web interface. This way, changes to the API can be pushed to the repository together with their implementation. DDD highly recommends to use continuous integration [4], thus, the continuous integration pipeline is configured, too.

Venon [17] describes how to implement REST resources separating the application from the domain layer. The web API describes entry points to the microservice; it can be implemented straight forward. The logic at the entry points should be application specific in order to separate application specific parts from domain specific, e.g., the usage of REST. Thus, a microservice should have an application layer on top. Typically, this layer is implemented as an anti corruption layer; a design pattern to achieve a clean separation of application and domain terms [17]. Additionally, by preventing the use of domain objects as input parameters, the coupling of domain objects and web API is reduced. Thus, some minor changes in the domain model do not influence the implementation of the interface [17].

The domain layer is implemented according to the domain model. Thus, the domain objects in the bounded context are mapped to classes, when using an object-oriented programming language. Constraints, such as multiplicities, and domain logic is implemented in the domain object. If a domain object from another microservice is used, a reference to the object is saved, e.g., by using the identifier [17], [29]. Implemented domain objects are intelligent objects that ensure the constraints in the domain. The application layer should never have access to domain objects, that do not comply with the constraints. Development approaches, such as test-driven development [30] or even behavior-driven development [31] are a good choice in order to achieve this goal. These constraints might be distributed among the domain model, thus, constraints might be overseen. Separating tester and developer of functionality, pair-programming as well as reviews can help to overcome this problem.

Beside of the application and domain layer, the infrastructure layer is part of the microservice. This layer contains functionality to access databases, log events, enforce authorization, cache results, discover services etc.; everything supporting the application and domain layer. Apparent is the support for domain repositories. If a microservice has a repository, the infrastructure layer must offer access to a database.

Last, we discuss the implementation of the UI's backend. The backend is an application of the BFF pattern. Therefore, a main goal is to offer a facade hiding the microservice architecture. The implementation can be kept simple. Its web API is implemented according to the specification. In our case, the specification is oriented on the microservice web API specification, thus, the request can be directly forwarded to the microservice. Depending on the specification, the frontend supports further functionality, e.g., authentication and access control may be implemented in the UI's backend.

IV. Case Study: Thesis Administration

In our case study, we were attempting a modernization of the thesis administration within the KIT department of informatics at Karlsruhe Institute of Technology. Our goal was to create an application based on microservices and to provide it to the university through the service-oriented platform SmartCampus [32] that we develop in our research group. For project execution, we chose Scrum as our software development process.

A. Crunching the Information Model

In relation to the presented approach, we started eliciting the domain knowledge with knowledge crunching. When we found the domain experts - members of the Main Examination Committee -, we started to discuss the domain. Quickly we noticed that the thesis is one of the main concepts of the domain. Thus, we explored the thesis by interviewing domain experts at first. Besides the concepts and relationships of the thesis, we also noticed constraints, which we included in the information model. Figure 4 shows a piece of our crunched information model. We put the *Thesis* in the middle of the model to reflect the central position within the core domain.

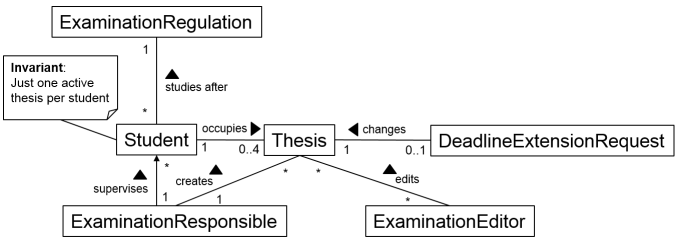

Figure 4. Piece of the information model showing concepts of the thesis domain object

Deeper discussions about the thesis told us somewhat about states that a thesis can occupy. At this point, we took into account the approach of domain views. We modeled a finite automaton to determine our understanding and discuss it with the domain expert, as shown in Figure 5. The diagram supports the understanding without using UML typical elements.

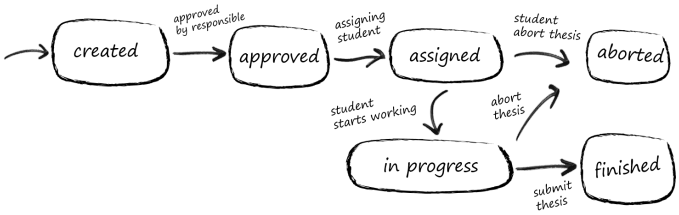

Figure 5. Finite automaton sketches the possible thesis states

After discussions with the domain experts, we had our desired information model and were able to transform it into a PIM.

B. Creating the Design Prototype

Besides crunching the information model, we started the design prototyping and sketched each identified use case. In Figure 6 we illustrate an information page of a specific thesis. This prototype was used to validate the elicited domain knowledge.

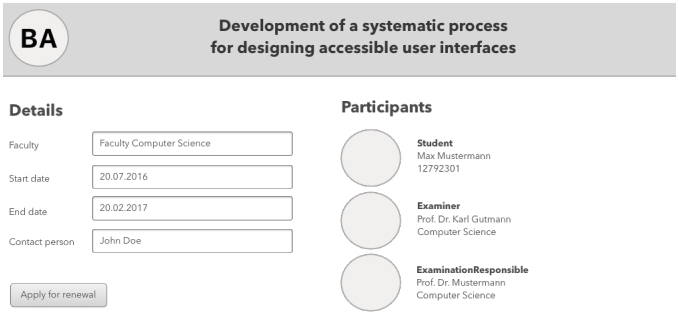

Figure 6. Design mockup (in early phase) for visualizing details about a thesis

C. Enriching the Information Model

After eliciting the domain knowledge and creating a design prototype, we were able to enrich our information model with implementation details. Mainly we focused on using the DDD patterns such as Bounded Context, Entities, Value Objects or Repositories [4]. At first, we structured the domain into bounded context according to Conway's Law [5] and, thus, divide the thesis administration domain into microservices. Then we could create the context map by the composition of the bounded contexts (see Figure 7).

Afterward, we started to identified entities, value objects and made a decision about persistence within our intended application through repositories. We oriented ourselves to the requirements of the application when applying the patterns. For example, we decided that the domain object "Student" in Figure 8 did not need a repository because it does not need to be globally accessible.

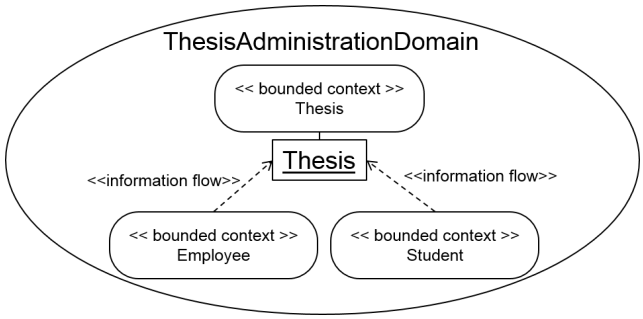

Figure 7. Context map composing bounded contexts of the thesis administration domain

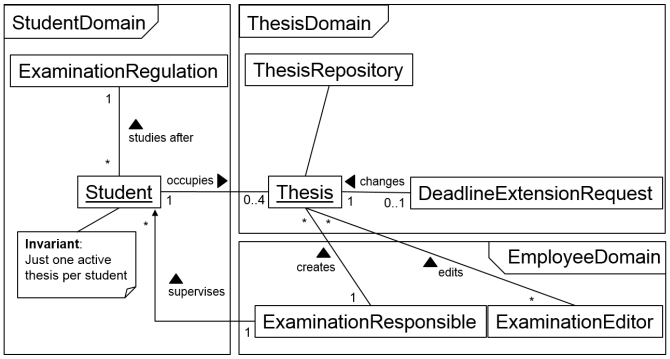

Figure 8. Thesis specific piece of the domain model including DDD patterns

D. Design and Implementation of the API Specification

The API specification was designed according to the domain model and UI/UX design. Figure 9 shows how to access a single thesis resource and its attributes. The attributes are mainly influenced by the information modeled in the design prototype.

Figure 9. OpenAPI specification of getting single thesis displayed with SwaggerUI

During the implementation phase, the domain objects in the bounded context were mapped to the source code. We used Java and the Spring Framework, which supported to focus on the domain layer. Spring is implemented having the concepts of DDD in mind. We separated the application and domain layers into different packages. We did not need an infrastructure layer, because Spring Data directly supports repositories through spe-

 ISBN: 978-1-61208-553-1

cialization. The database can be configured using configuration files. The entry point to the application is a Spring Controller. Several annotations helped to map HTTP requests to methods. Even more annotations enable the use of dependency injection, so that we could depend on repository interfaces while spring injected their implementation. The development team added sequence diagrams to model the interaction of the controllers. This is also due to a lack of experience.

E. Synergy between Approach and Scrum

It turned out that our presented approach complements itself well with Scrum. During each activity we did, we always had the Scrum artefacts in mind and tried to create them directly. Also the iterative approach from Scrum fit well to our executed activities. This corresponds to the principle of exploration and experimentation presented by Eric Evans in DDD [4].

Through the combination of information model and design prototypes, we could easily fill the Product Backlog. Also we were able to extract the user stories and their task within to create the Sprint Backlog from the PIM and API Specification. After each Sprint, we could adjust the PIM, transfer the changes into the Product Backlog and start a new Sprint.

V. Limitations

The activities we introduced provide an overview of the activities that take place when applying DDD in building microservice-based applications. These activities represent a first step towards a complete process that includes all of the required artifacts. Our research indicated that several topics require further investigation and more detailed descriptions; for example, it is quite difficult to systematize the design of the domain model according to DDD. Best practices could be identified and added to the process description to support the performance of this activity.

During the case study, we received useful feedback from the software development team. In Section III-A on classification, we discussed concerns regarding the specification that are not covered by the artifacts. We used the UI/UX design in addition to DDD and the microservice approach to provide the missing specification in the user interface and application layers; however, the development team still had problems implementing the functionality in the application layer. To address these problems, they added additional sequence diagrams that specified the usage of the domain layer within a microservice. It is likely, that there are more specification artifacts that must be identified, as, using the Spring framework, which supports developers in several ways, much of the application and infrastructure layer source code is supplied, which makes specification unnecessary.

While discussing the implementation process and testing activities, we noted that the implementation of a domain model created according to DDD is (slightly) bound to object-oriented programming languages. This is due to the fact that the concepts and diagrams introduced in [4] have object-oriented programming in mind. The use of a functional programming language might require a different set of patterns and diagrams; as such, the process identified in this article is also somewhat bound to implementation using an object-oriented language.

VI. Conclusion and Future Work

DDD offers key concepts and activities to build applications based on a microservice architecture, whereby the activities are missing links to existing software engineering knowledge. We classified both into software architecture concepts and software development activities. Further, we introduced an overview of software development activities and artifacts for building microservice-based applications, which extend DDD. In a case study, we showed the application of the activities in an agile software development process to build a thesis management applications as part of the SmartCampus and gave examples of the resulting artifacts. The overview of activities and their classification is a first step towards a complete process for developing such web applications and, thus, we described its limitations and missing artifacts.

DDD is about focusing on the domain including its concepts, their relationships and business logic. Microservice architecture is about arranging and dividing distributed software building blocks. We showed missing requirement specifications and missing artifacts with our classification and the case study. We will further refine the activities towards a software development process to identify a sufficient set of artifacts.

A major advantage of DDD and microservices is the reuse of existing functionality. Identity and access management is a domain (almost) each application needs, thus, we will investigate in building a knowledge repository and enriching the activities and artifacts so that models and functionality in this domain can be reused among applications. In addition to this research topic, we will continue to focus on how we can systematically derive web APIs for microservices with quality aspects in mind such as evolvability. The web API also plays a significant role in discovering and reusing microservices in the context of a microservice landscape.

Acknowledgment

We are very thankful to Pascal Burkhardt for his contributions, both through discussions and the input he provided regarding his projects, as well as to Philip Hoyer for providing his opinions during our discussions. Furthermore, we would like to thank the following members of the development team and domain experts for participating in the case study: Florian BReuer, Lukas Bach, Anne Sielemann, Johanna Thiemich, Rainer Schlund, Niko Benkler, Adis Heric, Pablo Castro, Mark Pollmann, Iona Gheta, Johannes Theuerkorn and David Schneiter.

References

[1] S. Newman, Building Microservices, 1st ed. O'Reilly Media, Inc., 2015.

[2] M. Richards, Microservices vs. service-oriented architecture. O'Reilly Media, Inc., 2015.

[3] T. Erl, SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl). Upper Saddle River, NJ, USA: Prentice Hall PTR, 2007.

[4] E. Evans, Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional, 2003.

[5] M. E. Conway, "How do committees invent," Datamation, vol. 14, no. 4, 1968, pp. 28–31.

 ISBN: 978-1-61208-553-1

[6] E. Landre, H. Wesenberg, and H. Rønneberg, "Architectural improvement by use of strategic level domain-driven design," in Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, ser. OOPSLA '06. ACM, 2006, pp. 809–814, URL: http://doi.acm.org/10.1145/1176617.1176728 [retrieved: 2017-03-03].

[7] B. Iyer and M. Subramaniam, "The Strategic Value of APIs," January 2015, URL: https://hbr.org/2015/01/the-strategic-value-of-apis [retrieved: 2017-03-03].

[8] D. C. Schmidt, "Model-driven engineering," COMPUTER-IEEE COMPUTER SOCIETY-, vol. 39, no. 2, 2006, p. 25.

[9] A. G. Kleppe, J. Warmer, W. Bast, and M. Explained, "The model driven architecture: practice and promise," 2003.

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

[11] I. Jacobson, G. Booch, and J. Rumbaugh, The unified software development process. Addison-wesley Reading, 1999, vol. 1.

[12] G. Fairbanks, Just enough software architecture: a risk-driven approach. Marshall & Brainerd, 2010.

[13] O. Vogel, I. Arnold, A. Chughtai, and T. Kehrer, Software Architecture: A Comprehensive Framework and Guide for Practitioners. Springer Berlin Heidelberg, 2011, URL: http://dx.doi.org/10.1007/978-3-642-19736-9 [retrieved: 2017-03-03].

[14] I. A. W. Group et al., "Ieee recommended practice for architectural description," IEEE Std, vol. 1471, 1998.

[15] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," in Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on. IEEE, 2016, pp. 44–51.

[16] C. Pahl and P. Jamshidi, "Microservices: A systematic mapping study," in Proceedings of the 6th International Conference on Cloud Computing and Services Science, 2016, pp. 137–146.

[17] V. Vernon, Implementing domain-driven design. Addison-Wesley, 2013.

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

[19] Y. T. Lee, "Information modeling: From design to implementation," in Proceedings of the second world manufacturing congress. Citeseer, 1999, pp. 315–321.

[20] J. Arnowitz, M. Arent, and N. Berger, Effective Prototyping for Software Makers. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[21] E. Evans, "Tackling complexity in the heart of software," January 2016, domain-Driven Design Europe 2016, URL: https://hbr.org/2015/01/the-strategic-value-of-apis [retrieved: 2017-03-03].

[22] M. Gebhart, P. Giessler, and S. Abeck, "Challenges of the digital transformation in software engineering," ICSEA 2016 : The Eleventh International Conference on Software Engineering Advances, 2016, pp. 136–141.

[23] D. Jacobson, G. Brail, and D. Woods, APIs: A Strategy Guide. O'Reilly Media, Inc., 2011.

[24] B. Mulloy, "Web API Design - Crafting Interfaces that Developers Love," March 2012, URL: http://pages.apigee.com/rs/apigee/images/api-design-ebook-2012-03.pdf [retrieved: 2017-03-03].

[25] J. Webber, S. Parastatidis, and I. Robinson, REST in Practice: Hypermedia and Systems Architecture, 1st ed. O'Reilly Media, Inc., 2010.

[26] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[27] P. Giessler, M. Gebhart, D. Sarancin, R. Steinegger, and S. Abeck, "Best Practices for the Design of RESTful web Services," International Conferences of Software Advances (ICSEA), 2015, URL: http://www.thinkmind.org/download.php?articleid=icsea_2015_15_10_10016 [retrieved: 2017-03-03].

[28] OpenAPI, "The OpenAPI Specification (fka The Swagger Specification)," 2017, URL: https://github.com/OAI/OpenAPI-Specification [retrieved: 2017-03-03].

[29] O. Gierke, "DDD & REST - Domain Driven APIs for the Web," November 2016, SpringOne Platform, URL: https://www.infoq.com/presentations/ddd-rest [retrieved: 2017-03-03].

[30] K. Beck, Test-driven development: by example. Addison-Wesley Professional, 2003.

[31] D. North, "Behavior modification: The evolution of behavior-driven development," Better Software, vol. 8, no. 3, 2006.

[32] R. Steinegger, J. Schäfer, M. Vogler, and S. Abeck, "Attack surface reduction for web services based on authorization patterns," The Eighth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2014), 2014, pp. 194–201.

 ISBN: 978-1-61208-553-1