Designing Microservice-Based Applications by Using a Domain-Driven Design Approach

Article · December 2017

CITATIONS
14

READS
2,980

5 authors, including:

Benjamin Hippchen
Karlsruhe Institute of Technology
6 PUBLICATIONS **52** CITATIONS
SEE PROFILE

Pascal Giessler
Karlsruhe Institute of Technology
6 PUBLICATIONS **70** CITATIONS
SEE PROFILE

Roland Steinegger
Engineering Steinegger GmbH
10 PUBLICATIONS **76** CITATIONS
SEE PROFILE

Michael Schneider
Karlsruhe Institute of Technology
3 PUBLICATIONS **32** CITATIONS
SEE PROFILE

Designing Microservice-Based Applications by Using a Domain-Driven Design Approach

Benjamin Hippchen, Pascal Giessler, Roland Heinz Steinegger,
Michael Schneider and Sebastian Abeck

Research Group Cooperation & Management (C&M)
Karlsruhe Institute of Technology (KIT)
Zirkel 2, 76131 Karlsruhe, Germany
(benjamin.hippchen | pascal.giessler | steinegger | abeck)@kit.edu,
michael.schneider5@student.kit.edu

***Abstract*—The current trend of building web applications using microservice architectures is based on the domain-driven design concept. Among practitioners, domain-driven design is a widely accepted approach to building applications. Applying and extending the concepts and tasks of domain-driven design is challenging because it lacks a software development process description and classification within existing software development process approaches. For these reasons, this paper provides a brief overview of domain-driven design-based software development activities and their classification into a well-known software development process.**

Keywords–Domain-driven design; behavior-driven development; domain model; microservices; API;

I. INTRODUCTION

This article is an extended version of [1], which was published at SOFTENG 2017. Our former article presents an overview of activities for building microservice-based applications by using a domain-driven design (DDD) approach. In addition, we discuss the elicitation of requirements, align the hexagonal architecture for microservices into the field of software architecture and discuss testing and implementing the different layers of microservices. By considering application requirements, we tackle limitations described in our previous article. Microservice architectures have evolved into a popular method for building multiplatform applications over the past few years. A well-known example is Netflix, who offers applications for several platforms, including mobile devices, smart TVs and gaming consoles [2]. Service-oriented architectures are the foundation of microservice architectures, but microservices have unique properties [3]. A microservice is autonomous and provides a limited set of (business) functions. In service-oriented architectures, designing services and selecting boundaries are fundamental problems.

The traditional approach, as discussed by Erl [4], suggests a technical and functional separation of services. In contrast, according to Evans [5], DDD provides the key concepts required to compartmentalize microservices [2]. The DDD approach provides a means of representing the real world in the architecture, for instance, by using bounded contexts representing organizational units [6], and also identifies and focuses on the core domain; both of these characteristics lead to improved software architecture quality [7]. In microservice architectures, these bounded contexts are used to arrange and identify microservices [2]. Using DDD is a critical success factor in building microservice-based applications [2].

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. Further, the classification of the patterns and principles into software architecture concepts, such as architecture perspectives and its requirements, supports software architects in designing microservice architectures.

In addition, there are no clear guidelines regarding how to derive the basic 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 [4]. From the business perspective, the web APIs also have strategic value; therefore, the development team must design it in a manner that emphasizes quality [8].

Furthermore, applications and, in particular, user interfaces, are often not considered or only considered superficially during the process of designing service-oriented architectures [2][4]. 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 [2]. 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 offered service. 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 activities 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. Further, we combined the application of DDD with behavior-driven development (BDD). The gap of the requirement specification on the application level, which we discussed in our previous article, is tackled through the "living documentation" [9]. So, DDD provides the core of the application and BDD specifies the access to it. BDD also helps to test the application from a user's point of view and to test the domain model at the same time.

This article is structured as follows: In Section II, DDD, BDD and microservice architecture, including a general introduction to software architecture and development and other related concepts, are introduced. Section III classifies DDD, BDD 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), BDD for requirements elicitation, software architecture in general (as the foundation being the foundation for classifying microservice architecture) and microservice architecture.

A. Model-Driven Engineering and Model-Driven Architecture

Schmidt [10] 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) [11] 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.

Using a model-driven approach, models become the primary artifacts in the development of applications. Thus, a clear understanding of the model and modeling language is necessary for an effective use. Metamodels are used to define the possibilities about what the model language can express [12]. In case of the Unified Modeling Language (UML), OMG defined the Meta-Object Facility (MOF) as the basis for all metamodels [12]. Defining model languages with this framework, a four-layer metamodeling architecture is applied: 1) M0 designates the real world object, that will be model, 2) M1 denotes the model, which represents M0, 3) M2 defines a metamodel for limiting the modeling possibilities from M1, 4) M3 represents the meta-metamodel, which specifies the modeling language for metamodels. Since MDA is a model-driven appraoch, it sticks to the four-layer architecture from MOF.

B. Software Engineering Activities and Domain-Driven Design

Brügge et al. [13] 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 span a several domains. [13] offers an overview of the activities that take place during software development: requirements elicitation, analysis, systems design, object design, implementation, and testing. (These activities are discussed further in the article's introduction of the development activities.)

DDD is an approach that is used in application development where the domain model is the central artifact. Software architects and developers use the domain model as main source for software design and development. Furthermore, DDD focuses on the business logic of the customer's domain and neglects technical aspects of the application. Evans introduced this approach in the book "Domain-Driven Design: Tackling Complexity in the Hearth of Software" and identified the essential principles, activities, and patterns required when using DDD [5].

A domain model that conforms to Evans' DDD approach contains everything that is necessary to understand the domain [5]. This approach goes beyond the traditional understanding of a domain model, which is connected to a formalized model using the UML [14]. To distinguish between the two concepts, following Fairbanks [15], we use the term information model which corresponds to a CIM. It is a part of the domain model and consists out 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 case of DDD, the modeling language of the domain model is not specified by an metamodel—like the CIM and PIM in MDA. A metamodel would be contradictory to the "everything is allowed in the domain model" philosophy of Evans. The comparison of the domain model to a CIM and PIM only reflects the evolution of the domain model and states nothing about the modeling language. As a result, the systematic approach of MDA cannot be applied to DDD's domain models.

In Evans' approach to DDD, the central principle is to align the desired 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. Requirements Elicitation with Behavior-Driven Development

The requirement elicitation activity described by Brügge [13] could be carried out in various ways. With DDD, requirements are gathered and stated in the domain model [5]. Regarding the layered architecture of DDD (see Section II-B), just requirements of the domain layer are covered. So, there is a lack of specifications for the other layers.

With BDD the requirement elicitation is carried out by the developers and application users themselves [9]; there is a difference between users and domain experts. BDD builds on an informal and executable requirement specification for the intended application. Both, users and developers, are exploring the requirements of the application, sharpening their picture of the application and establishing a shared understanding. The idea of BDD is based on test-driven development (TDD) and its automated acceptance tests [9], which determine the correctness of the application [16]. Furthermore, BDD emphasizes the understanding of the user's domain. BDD uses the ubiquitous language known from DDD [9].

The philosophy behind BDD is a requirement elicitation from outside-in [9]. North characterizes "*code-by-example*" in the context of BDD [17]. The most visible behavior is elicited first; then it is implemented. During the implementation, new details are discovered, which will also be stated as requirements. In BDD, requirements are stated as features, which are further divided into scenarios [9]. The language Gherkin is usually used to describe these artifacts formally. Each scenario is written in common speech and is refined in steps. Each step starts with a predefined (Gherkin) keyword, which is necessary to execute the tests in later stages. Through the binding of BDD features to implementation and testing activities, the requirement specification must be kept up-to-date; it becomes a "living documentation" [9].

D. Microservice Architectures

Vogel et al. provide a comprehensive framework for the area of software architecture [18], 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 [19]. Architecture views are part of the documentation that describes the architecture. Each view is motivated by stakeholders' concerns. These concerns specify the viewpoints on the architecture and, thus, specify the views.

Newman provides a comprehensive overview of microservices and related topics from an industry perspective [2]. 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 [4], the single purpose principle results in microservices having similar sizes within an architecture [3]. Two mapping studies regarding microservices and microservice architecture reveal that a gap in the literature regarding these topics exists [20][21].

Regarding the structure of a microservice based application, Vernon introduces the hexagonal architecture in the context of DDD [22]. Initially, the corresponding architecture pattern was postulated by Cockburn in [23]. In the resulting architecture by applying this pattern, the concerns of a microservice are separated into several layers, see Figure 1. According to Cockburn, the hexagonal architecture consists out of the domain model, application services and adapters with ports. Each side of the hexagon stands for a particular port, although, in practice, there could be more than six distinct ports. Using these ports, the microservice can upstream or downstream information with clients. For consuming microservices, the client has to use one of the exposed ports and its corresponding adapter. The adapters work like an anti corruption layer. Highly relying on the dependency inversion principle (DIP) from Martin [24], the domain model as the core and hence the business logic is independent of the surrounding application services and adapters. The layer dependencies are applied from the outside to inside. (Further relevant information is discussed during the section of this article that classifies microservice architectures.)

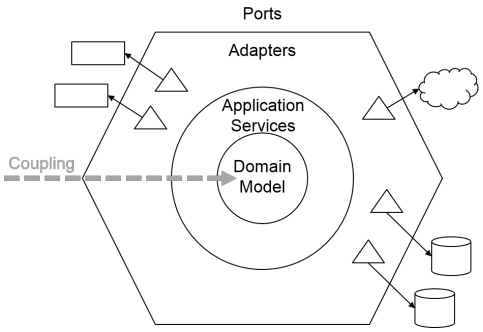

Figure 1. The hexagonal architecture pattern from Cockburn [23]

III. Process

This section classifies the activities involved in 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 that one constantly scrutinizes and adjusts the 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. [18]. We divide the 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. [18] Despite their names, microservice architecture and the domain model describe the macro-architecture. A microservice is a functional or technical software building block that requires 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 [2][22]. The domain objects within a bounded context specify the core architecture of a microservice.

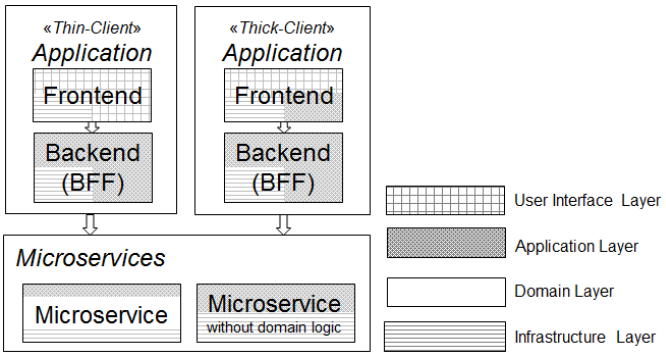

Figure 2. Software building blocks and their layers in a microservice-based application

Domain-driven design requires a layered architecture to separate the domain from other concerns [5]. Evans suggests a four-layered architecture, consisting of the user interface, application, domain, and infrastructure layers. Figure 2 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 thick or thin (meaning that it contains application logic or not), 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 [25].

In a layered architecture, higher layers can communicate with lower layers. Figure 3 depicts the layered architecture's communication paths applied to the above-mentioned software building blocks [25]. The frontend should not directly call the microservices; we emphasize this by using dashed arrows.

The layering from Figure 2 shows that the layering introduced by DDD is also applied to the microservice architecture. Going into the implementation of a microservice, the sequence of the layers changes. The underlying structure of the microservices is defined through the hexagonal architecture pattern by Cockburn [23]. Having a more accurate look at the building blocks of the microservices, we are using an onion architecture introduced by Palermo

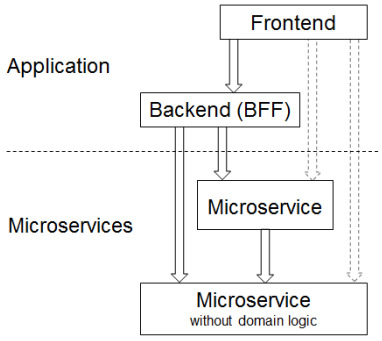

Figure 3. Communication between software building blocks

[26] with minimal adoptions as shown in Figure 4. The onion architecture builds on the hexagonal architecture. Vernon states that the hexagonal architecture and the onion architecture are the same [22]. On the outer layer of the onion architecture, we find the infrastructure as well as the exposed interface as fine-grained building blocks. The infrastructure part of the layer provides technical functionality for the operation of the service such as database access. The exposed user interfaces focuses on the provisioning of the underlying business domain that can be used by clients for interaction. Like in the hexagonal architecture, the adapter pattern is used on this layer. In addition, the onion architecture adds a new layer, the so-called domain services. Domain services represent behavior that cannot be mapped to domain objects [5][22]. For instance, this is the case if the behavior is spread over multiple domain objects to form a business workflow. This way, the domain services are tightly coupled with the domain objects; together, they build the whole domain. In [26], Palermo also puts the interface definition of the repository on this layer. But, in our opinion, this approach would mix the domain-specific layer with technical aspects. That is why we add the so-called glue layer that acts as a link layer between the application and domain. The from DDD known repositories or factories are put into this layer. The heart of the onion architecture is represented by the domain objects.

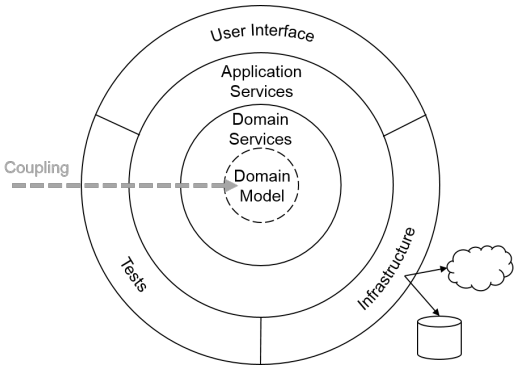

Figure 4. Our adapted onion architecture based on Palermo [26]

The layered architecture is applied to the whole application and divides it into horizontally-divided layers. Meanwhile, the onion architecture is applied to the microservice building

blocks from Figure 3 and divides them in a vertical manner. Going back to DDD, a microservice is defined through a single bounded context in the customer's domain [2]. Considering the microservice architecture as an onion makes it more suitable with the concept of bounded contexts. Every microservice has to work autonomously, which is intended with the layering of the onion architecture.

Concerning architecture requirements, the decision to build microservice-based applications is taken at the organizational level (see the classification of service-oriented applications in [18]). Along with a microservice architecture, the organization 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 [5]. 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 [15]. In case of the application layer, we started using BDD as an approach for eliciting requirements as features from the application users. The features could also be used for testing the domain layer. Further, 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 are introducing the activities involved in building microservice-based applications. These activities facilitate the development of applications within similar domains, e.g., an application that offers information on points of interest, an application navigating from and to points of interest and an application that enables the management of points of interest. We align our activities with the traditional software development activities described by Brügge et al. [13]. Therefore, the activities end after testing, and we do not discuss deployment and/or maintenance. Artifacts associated to deployment and maintenance, such as a deployment diagram or a platform description, are not discussed. Figure 5 depicts the three activities and their interrelations: *requirements elicitation and analysis*, *design* and *implementation and testing*.

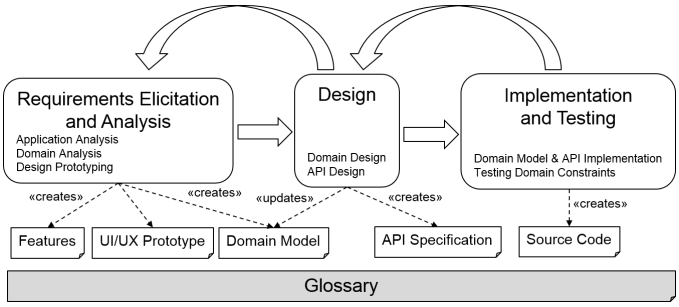

Figure 5. Overview of the activities used in building microservice-based applications

During the *requirements elicitation and analysis*, three sub-activities take place: first, the application requirements are stated in features with and from users by following the BDD approach; second, the information model, as part of the domain model, is created by "crunching knowledge" with domain experts; third, a prototype is designed and is discussed with both the user and customer. As all activities are closely related (when discussing prototypes, the knowledge of the domain gets deeper, when discovering the information model, terms or workflows might change, and while eliciting requirements, the goal of the application is sharpened), 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 domain experts, the information model is refined, e.g., design decisions are made, and design patterns from DDD are applied. Domain design is comparable to the system design activity discussed by Brügge et al. [13]. The system is divided into subsystems that, according to Conway's Law [6], 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) are 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. [13].

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 features from the application analysis are used to test the application. In particular, they are used to test the constraints defined in the domain model, such as multiplicities or directed associations.

The ubiquitous language is central to the use of DDD, therefore, we introduced a glossary to capture the domain terms. Each term of the ubiquitous language is listed and described by a few sentences. We see the glossary as a cross-sectional artifact. It is created and updated in each activity of Figure 5. Maintaining the glossary takes effort, but the benefits outweigh this effort.

Evans states that developing a "*deep model*", with which

to facilitate software development requires "*exploration and experimentation*" [5]. Thus, in order to gain insights into the domain across the whole software development activities, software developers must be open-minded. The knowledge gained will probably lead to changes being made to the artifacts created in the previous phases. Therefore, a process of iteration and returning to previous phases is possible in each phase; in other words, it is very common for developers to switch between phases and activities. Of course, experienced developers may make fewer mistakes and discover insights earlier, but hidden knowledge and misunderstandings are common. In the following sections, the phases are explained in more detail.

C. Requirements Elicitation and Analysis

The first activity deals with the understanding of the application and the needs of the users. Three non-chronological ordered activities take place in this phase, namely the analysis of the application, exploration of the domain and the design of a prototype. These activities influence each other to a significant degree; terms from the application features and the domain model are used in the prototype, while new insights may result in changes being made to them. We see a strong connection between the process of eliciting the application requirements, developing the domain model and design prototyping that arise due to the specifications that are not captured during the domain modeling process. Every domain concept displayed in the design prototype must be modeled in the domain model and vice versa. In addition, the domain concepts must be used by the application features. Small iterations during the analysis phase are required to ensure that all artifacts are consistent.

1) Application Analysis: Specifying the Application with BDD: The functionality of the application depends on the needs of the user. Therefore, we see a comprehensive requirement elicitation with the users. By applying BDD to elicit requirements, the users are fully involved [9]. Requirements are stated in features (see Section II-C) and scenarios. Each feature is created by developers, users or both. A developer explores the intended application while discussing and creating features with users. BDD also emphasizes that users are able to create features on their own. Both, developer and user, features have to be considered equal for the implementation activity. This activity is resulting into a comprehensive requirement specification, which is executable for automated application testing. Regarding the layered architecture from DDD, the application layer from our intended application is specified.

By using BDD, our approach starts with the creation of features by the developers and users. Both are discussing and simultaneously creating the features. We noticed an improvement in quality and accuracy, if developers and users are collaborating during feature creation. BDD emphasizes an "*outside-in*" approach for the elicitation of application requirements [9]. The most visible behavior is stated as a feature and implemented subsequently. During implementation, developers will discover more details, which have to be discussed with the users. Either an existing feature is adjusted or a new feature is created. Afterward, the features are implemented step-by-step. North calls this "*code-by-example*" in a presentation at London's QCon 2009 [17].

During discussions and feature-writing, the synergy with DDD gets visible. The ubiquitous language is established and sharpened while talking about the features. Developers get insights about the relevant domain concepts. In addition, BDD emphasizes a "*living documentation*", which is achieved by the strong binding of features, implementation and testing [9][27]. This living documentation requires a continuous adoption of features; they need to be up-to-date in every phase of the project. The living documentation leads to a meaningful requirement specification.

The application analysis using BDD is resulting into a requirement specification, which consists of informal features and scenarios. The informal character makes it possible for users and developers to validate the correctness. The "for everybody" readable features provide an up-to-date source of knowledge, which enables that the functionality is implemented properly and works as intended. Developers can use the features as guidance during the implementation and as a main artifact for writing tests.

2) Domain Analysis: Exploring the Domain with DDD: Without a complete understanding of the domain, building applications that satisfy the requirements is difficult. In our approach, we focus on Evans' book "Domain-Driven Design: Tackling Complexity in the Heart of Software" in order to understand the needs and, thus, the domain through modeling [5]. Creating a comprehensive domain model in this phase even requires experienced domain modelers to acquire new knowledge. After this activity, we are left with a domain model that is the equivalent of an information model (see Section II-B). UML class diagram syntax is used to describe concepts and their relationships, constraints, etc. [15][28]. Also, this information model corresponds to a CIM. CIM is well known from the MDA [29][30].

According to DDD, collaboration with customers is essential to the exploration process and, in particular, the modeling of the domain. Thus, the permanently recurring activity, in DDD is knowledge crunching [5]. The development team simultaneously holds discussions with customers while carrying out the modeling process and creating the domain model. Conforming to the pattern *Hands-On Modelers*, every team member involved in the software development process should also participate in the domain-modeling process in order to promote creativity [5]. In addition, the "*ubiquitous language*" is established, which is the cross-team language. As Vernon [22] suggests, we create a glossary to record the terms of the domain that are part of ubiquitios language. The process, by which the domain model is developed, is influenced to a great degree by exploration and experimentation [5]. It is far better to implement a domain model that is not completly satisfactory than to repeatedly refine the domain model without actually implementing it [5]. Using the DDD approach requires an iterative process that takes into account principles taken from the agile development processes, such as short time to market. Especially the domain model needs to be adapted iteratively.

Complex domains automatically lead to a complex domain model; this complexity may make it difficult for readers to understand the domain model. As a result, it is necessary to split the model into multiple diagrams [25], which enables the modeler to model different aspects of the domain. Dynamic behaviors, such as workflows, are concepts that are relevant to the domain. We adopted the architectural view concept (see Section III-A) from software architecture [18] by introducing "*domain views*", which guides the modeler to model different

aspects of the domain. The concept is designed to split the domain model into multiple diagrams. Furthermore, it simplifies and structures the modeling activity. We see the need for a more guided modeling activity to support less experienced modelers.

Figure 6 displays the building blocks of the domain view concept and their relations. DDD is always based on an application and each artifact is aligned with the desired functionalities. Thus, the domain model belongs to the application and is only valid for the specific application. The domain model consists of mulitple domain views, which contains the domain objects. A domain view is assigned to a specific domain view type. This domain view type is used to determine the domain objects and identify the possible representation with UML diagrams. A type of a domain view is specified through one or more stakeholders. Each stakeholder describes the system from their respective viewpoints; stakeholders have one or more interests. Evans states, that for representing domain knowledge every possible diagram is allowed for modeling the domain model as long as it is supporting the understanding of the domain concept [5]. In our approach, we limited the possible representations with a set of UML diagrams. Each domain view type has its own set.

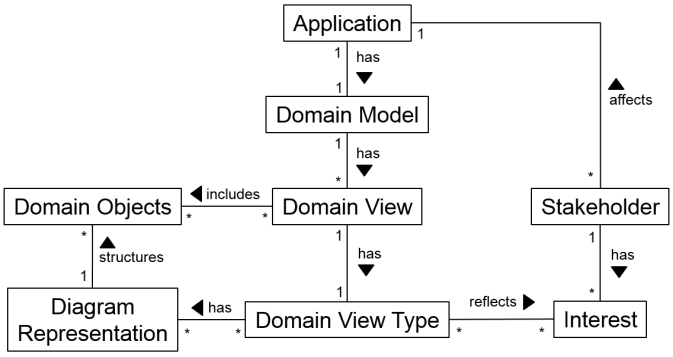

Figure 6. Building blocks of the domain view concept

We created two domain view types. First, the "*relation view*" models domain objects and their real world relationships with each other. This domain view type addresses the static behavior of the domain. Second, the "*process view*" represents the dynamic behavior of the domain. Domain objects and interactions are used to model processes of the domain.

The relation between the interest of a stakeholder and domain view type (see Figure 6) makes it possible to determine the right persons for the discussion of the domain view contents. Thus, during knowledge crunching, we were able to identify modelers and domain experts. Stakeholders have different interests and affect the structure of the domain view content. In addition to the domain expert, we defined developers, software architects, security architects and API designers as stakeholders. The API designer as a stakeholder emphasizes the development of microservice-based applications.

The result of exploring the domain is a domain model, which contains concepts that are relevant to the application (also called the "*domain knowledge*") [5]. Domain-driven design emphasizes the focus on the core domain. Implementing this domain has the highest priority [5]; the best developers are assigned.

3) Design Prototyping: By means of knowledge crunching, we obtain a complete understanding of the domain considered. The requirements of an application are use case-specific and function as indicators of the domain logic that must be modeled in the domain model. Based on BDD and the discussion with the stakeholders, each identified feature will be represented in the so-called design prototype. A prototype is an efficient means of testing new design concepts and determining their efficiency [31]. The design prototype is specialized, as it focuses on the application's UI and the UX. Since the customer primarily interacts with the UI, it is also an ideal artifact for further discussions with customers regarding the domain model. Further benefits of using a prototype can be found in [31]. Similarly to knowledge crunching, design prototyping is an iterative activity. Each iteration involves a brain-storming session that focuses on design ideas, taking into account the given boundary conditions, realization of the previously chosen design ideas and presentation and review of the resulting design prototype. As part of the review process, feedback from the customer will be collected and analyzed in order to identify the design changes necessary for the next iteration. The design prototyping process is completed when the prototype satisfies all of the customer's 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, both the domain model and the API specifications will be ready to implement.

1) Domain Design: From Computational to PIM: An important DDD concept is the binding of the domain to the process of implementation [5]. The domain model is the core artifact required in achieving this domain-layer goal. During the analysis phase, a CIM is created as a part of the domain model. First, this model is divided into bounded contexts, and, second, these bounded contexts are extended and refined, e.g., by applying design patterns to satisfy the application's requirements. These activities were based on the examples provided by Evans and Vernon [5][22].

The organizational structure is used to divide the information model into bounded contexts. Due to its importance, this task requires experience and several iterations [22][32]. The process of division is closely linked to the division of the development teams, as each works on a bounded context [2]. Thus, intermediate results should be discussed with the domain experts and other team members. The result is a context map that depicts the relationship between the bounded contexts.

The next steps are mainly carried out by the development teams that are responsible for each bounded context. The goal of the next activity is to refine and extend the domain model according to the application's requirements. Both BDD features and UI/UX design are the main source considered when it comes to determining the application's requirements.

In all likelihood, the domain objects in the information model will already be marked with stereotypes that indicate their type, e.g., aggregate root, value object, entity or domain event. Some services might be identified during the analysis

phase. Domain objects that lack a stereotype should be addressed first, meaning that a stereotype should be added. Next, the design patterns repository, factory and domain service are added according to the application's requirements. For example, if there is a need to display a domain object in the UI, a repository may be added, or, if there is a complex aggregate root, a factory could be added [5]. During the entire design process, the domain experts and other sources of information should be involved (this is referred to continuous knowledge crunching). After applying the design patterns, the domain model is ready to be implemented.

2) API Design: Deriving the Web API from the PIM: Microservices provide their implemented business functions via web APIs [2]. A web API can be seen as a specialization of a traditional API, which is why we extend the definition offered by Gebhart et al. slightly further: an API is "a contract prescribing how to interact with the underlying system [over the Web]," [33, p. 139]. From a business perspective, a web API can be seen as a highly valuable asset [8][34] that can also serve as a solution for digital transformation [33].

A web API can be used to coordinate microservices in the mapping of a complex business workflow onto the area of microservices or to offer business functionality to third-party developers [33]. To facilitate the reuse and discovery of existing microservice functions, the exposed web APIs should be designed with care. According to Newman [2], Jacobsen [34] and Mulloy [35], web APIs should adhere to the following informal quality criteria: 1) they should be easy to understand, learn and use from a service consumer's point of view, 2) they should be abstracted from a particular technology, 3) they should be consistent in look and feel and 4) they should be robust in terms of evolution.

To address these challenges, it is necessary to develop a systematic approach for deriving the web API from its underlying domain model. First, we decided to build web APIs in a resource-oriented manner that can be positioned on the second level of the Richardson maturity model [36]; we do not pursue the hypermedia approach suggested by Fielding [37] to reduce complexity when building microservice-based applications. Second, we have identified resources and sub-resources from the underlying PIM by examining the relationships between the domain objects. For the identification, we have created heuristics based on a UML class diagram that allows us to derive resource candidates systematically. The derived resources are then categorized in different types of resources according to Tilkov [38] and Rathod et al. [39]. Based on the resulting categorized set of resources, we have added URIs so that each resource can be addressed over the web without ambiguity. For the URI structure, we have followed several conventions to improve the discoverability and ensure the consistency in the context of our microservice landscape. The used conventions are listed as linguistic patterns and established best practices in this area [40]. Afterward, we have investigated the possible interactions of each of the identified resource types regarding their life cycle and mapped them subsequently to the modeled behavior of the domain. In addition, the interactions that will be exposed by the web API has to be also aligned with the BDD features, domain model, and UI/UX design; that is why the domain views (see Section III-C2) play a significant role. Nonetheless, the interactions should not be limited to the different use cases to ensure a high reusability. This fact is of particular importance when choosing an appropriate API strategy. Up to this point, we have made no technology decisions; the resulting model can be seen as a PIM. With the selection of an appropriate application protocol such as HTTP or Constrained Application Protocol (CoAP), we transform the PIM to a PSM.

The result of this design approach was finally structured according to the specifications of OpenAPI, which has the goal of "defining 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" [41]. In addition to this, we have extracted our guidelines for designing resource-oriented APIs in the form of a checklist in [40].

3) API Design: Deriving the Web API for BFF from the Design Prototype: A BFF is a pattern that is commonly used to avoid so-called chatty APIs [2]. Chatty APIs often result in the service consumer having to make a large number of requests in order to obtain the required information [35, p. 30f]. This is mainly due the fact that the domain information or logic required is spread over multiple microservices and primarily designed for reusability, rather than a specific use case in the form of a concrete application. In addition, BFFs allow a development team to focus on the UI and UX-specific requirements of an application by not restricting themselves to the microservices' exposed web APIs. When using DDD, additional application logic can be required, such as data transformation, caching or orchestration, which may be implemented at the BFF level or at the application layer [5]. For this reason, the BFF can be seen as part of the UI [2].

In our approach, the UI and UX specific requirements are represented using a design prototype that resulted from the analysis phase (see Section III-C3). Similarly to Section III-D2, we decided to adopt a resource-oriented style for the BFF web API and applied the same web API guidelines. The use of other solutions, such as a method-oriented approach, is also possible. For deriving the web API, we considered each view regarding the represented information as well as the interaction elements used for data manipulation. This approach allowed us to build resources, their representations and the necessary operations. The resulting web API is highly linked to the UI and must be connected to the underlying domain represented by microservices. Since both the domain model and the design prototype were designed using *ubiquitous language*, the required microservices can be identified with a minimum of effort and orchestrated on the application layer in order to fulfill the requirements specified by the BFF web API derived previously.

E. Implementation and Testing

The features, domain model and specification of the web API enabled the development team to implement the application. This section discusses the implementation and testing of the microservices. We do not discuss the implementation of the UI/UX design, as the focus of this study is on DDD and building microservices. However, the implementation and testing of the BFF, as the connection between the front-end and the microservices, are discussed. Furthermore, we discuss the use of BDD features for testing the domain model.

1) Implementation: Developing the Microservice: First, we focus on implementing and testing the microservices. Using the specified API, each bounded context is implemented as a microservice. A development project, e.g., a Maven or Gradle project that includes source code, necessary dependencies, and build instructions, is created and pushed into the version control repository. We recommend offering the API specification as part of the microservice; it is added to the repository and delivered through its web interface. Using this approach, changes to the API can be pushed to the repository together with their implementation. In addition, the API specification can also be delivered at runtime when offering a dedicated endpoint. This could, for example, be useful when having a dedicated API management system or a central service registry.

Domain-driven design highly recommends the use of continuous integration [5]; thus, the continuous integration pipeline was also configured. In addition to that, we have also added continous inspection that checks our latest build artifact against our test cases, coding guidelines or common issues from the Open Web Application Security Project (OWASP). Vernon [22] describes how to implement REST resources separating the application from the domain layer. The web API describes entry points to the microservice; they can be implemented in a straightforward manner. In order to separate application-specific parts from those that are domain specific, e.g., the usage of REST, logic at the entry points should be application specific. Thus, a microservice should have an application layer on top. Typically, this layer is implemented as an anti-corruption layer; a design pattern used to achieve a clean separation of application and domain-terms [22]. Additionally, by preventing the use of domain objects as input parameters,—the Spring framework offers this functionality by using the Jackson framework [42]—, the coupling of domain objects and the web API is reduced. Thus, some minor changes to the domain model will not influence the implementation of the interface [22]. Again, the whole architecture of the microservice is illustrated in Figure 4.

The domain layer is implemented according to the domain model. Thus, when using an object-oriented programming language, the domain objects in the bounded context are mapped to classes. Constraints, such as multiplicities and domain logic, are 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 its identifier [22][43]. Implemented domain objects are intelligent objects that ensure the constraints within the domain. To make sure that the constraints are correctly implemented, development approaches such as test-driven development [16] or even behavior-driven development [44] are good choices in order for achieving this. But, constraints can be distributed among the domain model; thus, constraints might be overseen. Separating the tester and the developer of the functionality, pair-programming and reviews can help to overcome this problem.

Besides the application and domain layers, the infrastructure layer is also part of the microservice. This layer contains the functionality used to access databases, log events, enforce authorization, cache results, discover services, etc.—in other words, everything that supports the application and domain layers. 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 that conceals the microservice architecture; implementation can be kept simple. Its web API is implemented according to the API specification. In our case, the specification is oriented to the microservice web API's specification; thus, the request can be directly forwarded to the microservice.

Depending on the specification, the frontend may be required to support further functionality; e.g., authentication and access control may be implemented in the UI's backend.

2) Testing: Using BDD for Verifying the Correctness of the Domain Model: In our approach, we use BDD for eliciting the requirement specification of the desired application. Thus, features are the main source of knowledge for developing and testing. These features are executable and intended to test the functionality of the application. Applying the code-by-example approach leads to an test-driven development activity. First, the most visible behavior of the application is stated as features. Afterwards, the development team implements the functionality until the feature passes the tests. During implementation, the development team will further explore the application requirements, add more features and implement them subsequently. Repeating this procedure over and over again leads to the intended application. The implementation is driven from the outside to the inside.

In consideration of the onion architecture of microservices, BDD starts with the specification of the outer layer and moves into the domain model during implementation. The domain model is getting the second source of knowledge. Through this refinement of the features the ubiquitous language and the domain knowledge appear; this makes the BDD testing effective [9]. Due to that, we see a strong synergy between BDD and DDD. North states that both approaches supports each other [17]. BDD enables modelers and domain experts to be more effective while knowledge crunching. The discussions are structured along the specified features. DDD helps the development team and the users to structure the discussion about the application requirements. Without a shared understanding of the domain concepts, the eliciting is impaired.

Domain concepts within the DDD domain model are tested through scenarios from the BDD features. The link between the domain model and the scenarios is the ubiquitous language. Furthermore, the domain model specifies the functionality of the domain objects. Thus, the usage of domain objects is indirectly limited; that helps developers and users to write correct features. Also dynamic behavior of the domain model, e.g., sequence diagrams in the process views (see Section III-C2), are tested. Through this limitation, we encountered the problem in the case study, that users desires functionality, which does not fit to the domain concepts.

IV. Case Study: Thesis Administration

In our case study, we attempted to modernize the thesis administration process within the Department of Informatics at the Karlsruhe Institute of Technology. Our goal was to create an application based on microservices and to offer it to the university through the service-oriented platform SmartCampus [45] that we developed in our research group. To execute the project, we chose Scrum as our software development process.

A. Eliciting the User Requirements

Following our approach, we began by eliciting the user-requirements with BDD. First, we identified users of the intended application. We focused on students and members of the examination committee, which are responsible for the coordination of exams; we chose a few contact persons from each group. After that, we started several discussions to elicit the most visible behavior, which we subsequently stated as BDD features. For each discovered feature, we developed scenarios as shown in Figure 7. These were easy to understand for our application users by following the "for everyone" readable approach. Therefore, we were able to further discuss our features and sharpen our understanding of the intended application.

Furthermore, we got some first impressions about the domain through the revealed domain logic in our scenarios; the guidance for exploring the domain with DDD.

Scenario: Assigning a student to a thesis offer
Given an approved thesis offer
When a student is assigned to a thesis
And the student accepts the assignment to the thesis
Then the student is assigned to the thesis
And the thesis offer is no longer available for any further assignments

Figure 7. Extracted scenario of a feature regarding the assignment of a student to a thesis

In addition, the specified scenarios allowed us to implement tests before starting with the implementation of the application. If a test fails after the implementation of the feature, or after changing parts of the code, we know that the application-requirements are not fulfilled.

B. Crunching the Information Model

Following the approach further, we began eliciting domain knowledge through knowledge crunching. After we identified the domain experts (members of the Main Examination Committee), we began to discuss the domain. We soon noticed that the thesis is one of the main concepts in this domain. Thus, we first explored the concept of the thesis by interviewing domain experts. Beyond the concepts and relationships we also noticed constraints regarding the thesis. We included these constraints in the information model. Figure 8 shows a section of our crunched information model. We placed the *Thesis* in the center of the model to reflect its central position within the core domain.

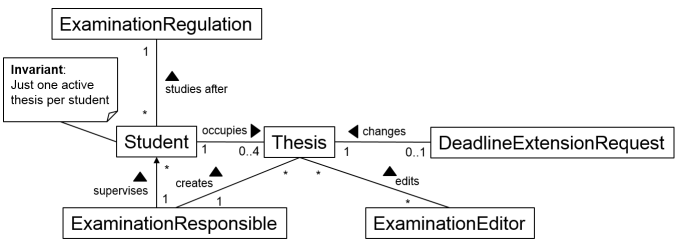

Figure 8. Section of the information model, showing concepts in the thesis domain

Further discussions regarding the concept of the thesis provided us with information about states that a thesis can demonstrate. At this point, we adopted the domain view approach. We modeled a finite automaton in order to determine our understanding of this concept and discuss it with the domain experts, as shown in Figure 9. The diagram supports the process of understanding, without the use of typical UML elements.

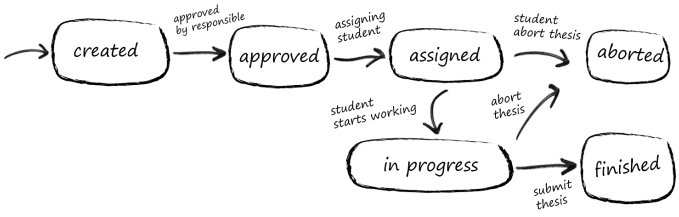

Figure 9. Finite automaton sketches the possible thesis states

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

C. Creating the Design Prototype

Beyond crunching the information model, we started the design prototyping process and sketched each identified use case. Figure 10 shows the information page of a specific thesis. This prototype was used to validate the elicited domain knowledge.

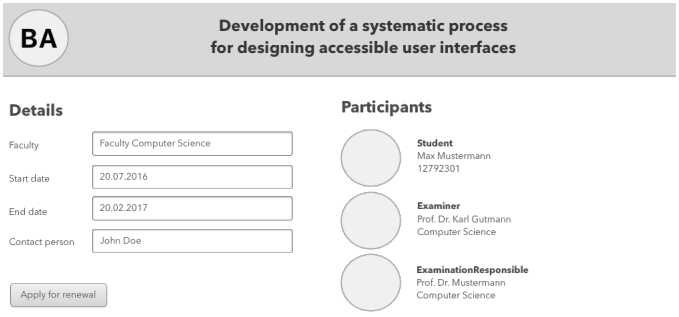

Figure 10. Early phase of a mockup for visualizing thesis details

D. 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. We mainly focused on using DDD concepts such as bounded context, entities, value objects or repositories [5]. At first, accordingly to Conway's Law [6], we structured the domain into bounded contexts, thus dividing the thesis administration domain into microservices. Having done this, we were able to create a context map that represented the composition of the bounded contexts (see Figure 11).

Thereafter, we began to identify entities and value objects and made a decision regarding persistence within our intended

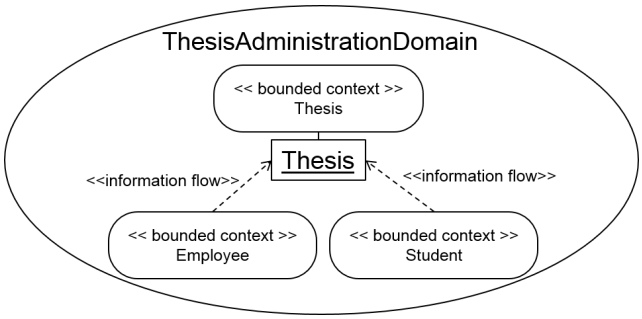

Figure 11. Context map representing the bounded contexts of the thesis administration domain

application through repositories. When applying the patterns, we took into account the application's requirements. For example, we decided that the domain object "Student" in Figure 12 did not require a repository, as it does not need to be globally accessible.

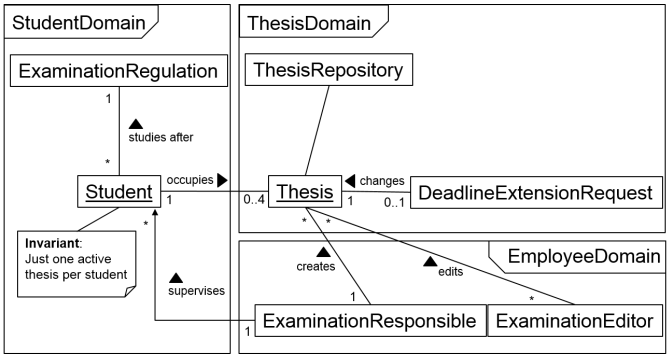

Figure 12. Thesis-specific section of the domain model, including DDD concepts

E. Design and Implementation of the API Specification

The API specification was designed with reference to the domain model and the design of the UI/UX. Figure 13 depicts how a single thesis resource and its attributes can be accessed. The attributes are mainly influenced by the information modeled in the design prototype.

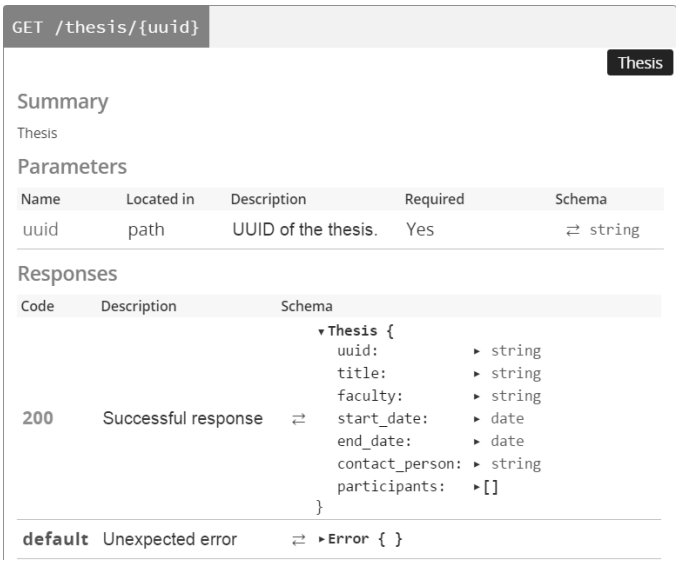

Figure 13. OpenAPI specification of displaying a single thesis using SwaggerUI

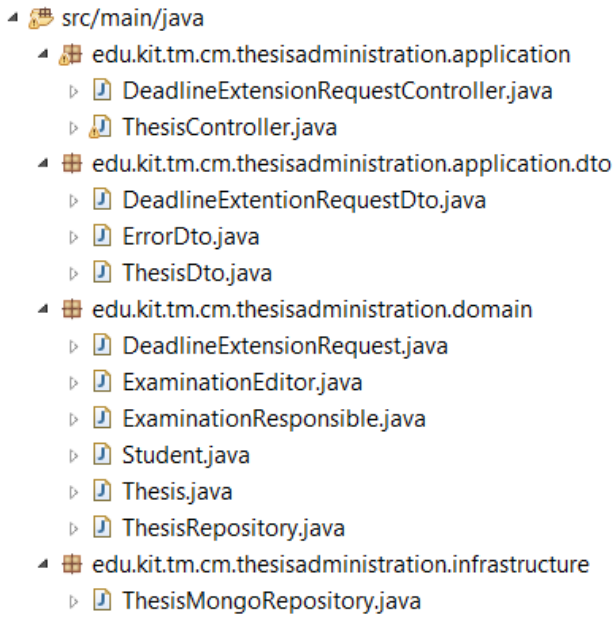

Figure 14. Package structure and classes of the Java implementation

F. Implementing the Microservice

During the implementation phase, the domain objects in the bounded context were mapped to the source code. As mentioned in Section III-A, a onion architecture is used to separate domain, application and infrastructure logic within a microservice. We used Java and the Spring framework to implement the microservices. In another project, we have also used the classic Java Enterprise Edition (JEE) approach so our implementation phase can be considered independent from the chosen implementation technology.

Java packages were used to separate the domain, application and infrastructure layers as shown in Figure 14. The Spring framework supported the developers to focus on the domain layer, because it has the concepts of DDD in mind. We used several annotations named according to DDD concepts. Also, we used the *data transfer object (DTO)* pattern from Fowler in the application layer [46]. The DTOs define the input and output structure of requests and enable to use the serialization functionality of Spring. The DTO has to be aligned with the representations defined in the OpenAPI specification in Section III-D2.

The source code of the thesis controller in Figure 16 shows the usage of Spring annotations and the DTO pattern. The entry point to the application is defined by the *RestController* annotation at Line 1). Each request to the path

```
1  @RestController("/theses")
2  public class ThesisController {
3   private ThesisRepository thesisRepository;
4   @Autowired
5    public ThesisController(ThesisRepository
         thesisRepository) {
6     this.thesisRepository = thesisRepository;
7    }
8    @RequestMapping(method=PUT)
9    public ThesisDto create(ThesisDto thesisDto) {
10    try {
11         Thesis thesis = new Thesis(thesisDto.
               getTitle(), /* ... */);
12         this.thesisRepository.save(thesis);
13    } catch (Exception ex) {
14         handleException(ex);
15    }
16         return new ThesisDto(thesis);
17    }
18    // ...
19 }
```

Figure 15. Part of the ThesisController

"$/theses$" relative to the application's path is handled by the *ThesisController*. By using dependency injection offered by Spring, interfaces and their implementation could be seperated. E.g., the implementation of the *ThesisRepository* is not part of the application layer. Instead, the infrastructure layer contains the implementing class *ThesisMongoRepository*, which uses, in this case, the NoSQL database Mongo DB as its persistence technology. Spring injects this implementation in the constructor of the *ThesisController*, because of the *Autowired* annotation (see Lines 4) to 7)). Starting at Line 8), a method for receiving and processing the thesis creation requests is implemented. The *RequestMapping* annotation with the parameter *PUT* makes Spring forward HTTP PUT requests to this method. Spring serializes the request body into a *ThesisDto* object. A thesis domain object is created by using the information encapsulated by this DTO. Afterward, the thesis domain object is persisted using the *ThesisRepository*. The *handleException* method handles exceptions and makes Spring respond with an *ErrorDto* object according to our API style guidelines (see [40]).

```
1  public Thesis(String title, Date startDate, Date
       endDate, UUID examinationEditorId, UUID
      exam /* ... */) throws ValidationException {
2   super();
3   setTitle(title);
4   setStartDate(startDate);
5   // ...
6  }
7  public void setTitle(String title) throws
       ValidationException {
8   if ((title != null) && (title.length() >
        MIN_TITLE_LENGTH)) {
9     this.title = title;
10  } else {
11   throw new ValidationException(/* ... */);
12  }
```

Figure 16. Part of the Thesis domain object

Domain objects shall always be valid and, thus, never contain information that is not consistent with the domain model. Therefore, the creation and manipulation of domain objects has to be handled with care. In our simple implementation, we decided to handle validation on our own and did not use a framework. The creation of the thesis domain object is depicted as a source code example in Figure 16. The constructor expects all attributes that are needed for a valid thesis. As an alternative, the factory pattern [47] could be applied to reduce the complexity of the constructor. The parameters are forwarded to the setters of the attribute starting from Line 3. The validation is implemented in the setters according to the domain model. As an alternative, one can use a dedicated method for verifying the invariants of a domain object before creating or updating it. In our source code example, the title must not be null and must have more than MIN_TITLE_LENGTH characters. If the validation fails, the setters, as well as the constructor, throw a *ValidationException* (Line 11)). The controller on the application layer can handle this exception and translate it for the requesting client. In our case, just the first failing validation is communicated to the application layer. The factory pattern could improve the implementation. A *ThesisFactory* might offer a method, which communciates a summary of validation problems.

G. Synergy between our Approach and Scrum

It turned out that our approach worked well with Scrum. We considered Scrum artifacts during each activity and attempted to directly create them. In addition, Scrum's iterative approach proved a good fit for our activities. This corresponds with the DDD principle of exploration and experimentation discussed by Evans [5].

In addition to our original article [1], we added features to describe user requirements. These improved the activity of writing items for the Product Backlog. BDD features and Scrum user stories are quite similar, because both describe the user's view on the application. Thus, through the combination of features and the design prototype, we could easily fill the Product Backlog. Furthermore, the activities of our approach can be used to refine the Items for the Spring Backlog. The interconnection of feautres, models and source code simplified the changes in the Backlog after each iteration.

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 for the user interface and application layer; however, the development team still had

problems implementing the application layer. At this point, we discovered that the application layer is still not fully specified. To further solve this problem, we added BDD to our process. With BDD the development team was able to specify the overall application and capture the functionality. It is likely, that there are more specification artifacts that must be identified, but the introduction of BDD helped us applying DDD for building microservice-based applications. Using the Spring framework, which supports developers in several ways, much of the infrastructure layer source code is supplied. Thus, these specifications are 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 [5] 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.

According to our domain view concept, the domain model consists out of multiple diagrams, which underlie one or more predefined representation styles. Mostly we used to the UML for the representation of domain model content. This use of UML suggests that we also have a underlying modeling language—or metamodel—for our domain model, but that would contradict Evans' premise "everything is allowed in the domain model". Introducing a modeling language for domain modeling would allow a systematic approach for deriving a web-API from the domain model.

VI. Conclusion and Future Work

Domain-driven design offers key concepts and steps for building applications that are based on a microservice architecture. However, the concepts lack links to existing software engineering knowledge. We classified both with reference to software architecture concepts and software development activities. In addition, we provided an overview of software development activities and artifacts used in building microservice-based applications, expanding on existing DDD literature. Using a case study, we demonstrated the application of these activities in an agile software development process, by building a thesis management application as part of the SmartCampus platform; we also provided examples of the artifacts that resulted. The overview of activities and their classification represents a first step towards a complete process for developing such web applications; we also described its limitations and discussed the missing artifacts. However, the application of DDD is still challenging and requires further investigations. The concept of domain views seems as a step forward but the concept is not yet mature; it lacks a well definition of its application and benefits.

Domain-driven design is about focusing on the domain, including its concepts, their relationships and business logic. Microservice architectures are about arranging and dividing distributed software building blocks. We identified a missing requirement specification and absent artifacts during the process of classification and the case study. We will further refine the activities towards a software development process to identify a sufficient set of artifacts.

The focus of DDD is the domain and its specification within a domain model. The other layers of the layered architecture are not provided through the domain model. For providing the application layer, we introduced BDD to our approach. The behavior of the application was stated in the form of plain text features. Furthermore, these features could be used to automatically test the application. In consideration of the architecture of our microservices, BDD tests each layer of the onion architecture. In case of our approach, we only described the testing activity of the domain model. Our research in this topic is not completed yet, so further research activities will concern the application of DDD in combination with BDD.

A major advantage offered by the use of DDD and microservices is the ability to reuse existing functions. Identity and access management are domains (almost) every application requires; thus, we will investigate building a knowledge repository and enriching the activities and artifacts so that the models and functionality used in this domain can be reused by other applications. In addition to this research topic, we will continue to focus on how we can systematically derive web APIs for microservices while bearing in mind quality requirements such as potential for future evolution. The web API also plays a significant role in discovering and reusing microservices in the context of a microservice landscape.

Applying the concept of the domain views on the modeling activities of DDD supports modelers and domain experts. Based on the definition of domain view types, modelers are more guided while modeling the domain. Nevertheless, cutting the domain model into multiple diagrams with domain views still requires much experience. The identification of the cutting edges of the domain model depends on the purpose of the application and the design decisions of the development team. In further investigations, we are going to extend the domain view concept to provide better support for modelers.

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 Beuer, 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] R. Steinegger, P. Giessler, B. Hippchen, and S. Abeck, "Overview of a Domain-Driven Design Approach to Build Microservice-Based Applications," in SOFTENG: The Third International Conference on Advances and Trends in Software Engineering, April 2017.

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

[3] M. Richards, Microservices vs. Service-Oriented Architecture. O'Reilly Media, Inc., 2015.

[4] 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.

[5] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003.

[6] M. E. Conway, “How do Committees Invent,” Datamation, vol. 14, no. 4, 1968, pp. 28–31.

[7] 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: 2017.11.30].

[8] B. Iyer and M. Subramaniam, “The Strategic Value of APIs,” January 2015, URL: https://hbr.org/2015/01/the-strategic-value-of-apis [retrieved: 2017.11.30].

[9] M. Wynne and A. Hellesoy, The Cucumber Book: Behaviour-Driven Development for Testers and Developers. Pragmatic Bookshelf, 2012.

[10] D. C. Schmidt, “Model-Driven Engineering,” Computer-IEEE Computer Society, vol. 39, no. 2, 2006, p. 25.

[11] A. G. Kleppe, J. Warmer, W. Bast, and M. Explained, “The Model Driven Architecture: Practice and Promise,” 2003.

[12] E. Seidewitz, “What Models Mean,” IEEE Software, vol. 20, no. 5, 2003, pp. 26–32.

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

[14] I. Jacobson, G. Booch, and J. Rumbaugh, The Unified Software Development Process. Addison-wesley Reading, 1999, vol. 1.

[15] G. Fairbanks, Just Enough Software Architecture: A Risk-Driven Approach. Marshall & Brainerd, 2010.

[16] K. Beck, Test-Driven Development: By Example. Addison-Wesley Professional, 2003.

[17] D. North. BDD & DDD. QCon London 2009. URL: https://www.infoq.com/presentations/bdd-and-ddd [retrieved: 2017.11.30]. (2009)

[18] 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: 2017.11.30].

[19] I. A. W. Group et al., “IEEE Recommended Practice for Architectural Description,” IEEE Std, vol. 1471, 1998.

[20] 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.

[21] 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.

[22] V. Vernon, Implementing Domain-Driven Design. Addison-Wesley, 2013.

[23] A. Cockburn, “The Pattern: Ports and Adapters,” 2005, URL: http://alistair.cockburn.us/Hexagonal+architecture [retrieved: 2017.11.30].

[24] R. C. Martin, Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, 2002.

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

[26] J. Palermo, “The Onion Architecture,” URL: http://jeffreypalermo.com/blog/the-onion-architecture-part-1/ [retrieved: 2017.11.30].

[27] G. Adzic, Specification by Example: How Successful Teams Deliver the Right Software, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2011.

[28] Y. T. Lee, “Information Modeling: From Design to Implementation,” in Proceedings of the Second World Manufacturing Congress. Citeseer, 1999, pp. 315–321.

[29] J. Osis, E. Asnina, and A. Grave, “Formal Computation Independent Model of the Problem Domain Within the MDA.” in ISIM. Citeseer, 2007.

[30] T. Stahl, M. Voelter, and K. Czarnecki, Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons, 2006.

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

[32] E. Evans, “Tackling Complexity in the Heart of Software,” January 2016, Domain-Driven Design Europe 2016, URL: https://dddeurope.com/2016/eric-evans.html [retrieved: 2017.11.30].

[33] 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.

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

[35] 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: 2017.11.30].

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

[37] R. T. Fielding, “Architectural Styles and the Design of Network-Based Software Architectures,” Ph.D. dissertation, University of California, Irvine, 2000.

[38] S. Tilkov, M. Eigenbrodt, S. Schreier, and O. Wolf, REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web. dpunkt, 2015, URL: https://books.google.de/books?id=pJF-ngEACAAJ [retrieved: 2017.11.30].

[39] D. M. Rathod, S. M. Parikh, and B. V. Buddhadev, “Structural and Behavioral Modeling of RESTful Web Service Interface Using UML,” in 2013 International Conference on Intelligent Systems and Signal Processing (ISSP), March 2013, pp. 28–33.

[40] 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: 2017.11.30].

[41] OpenAPI, “The OpenAPI Specification (fka The Swagger Specification),” 2017, URL: https://github.com/OAI/OpenAPI-Specification [retrieved: 2017.11.30].

[42] FasterXML, LLC , “Jackson JSON Processor Wiki,” 2017, URL: http://wiki.fasterxml.com/JacksonHome [retrieved: 2017.11.30].

[43] O. Gierke, “DDD & REST - Domain Driven APIs for the Web,” November 2016, SpringOne Platform, URL: https://www.infoq.com/presentations/ddd-rest [retrieved: 2017.11.30].

[44] D. North, “Behavior Modification: The Evolution of Behavior-Driven Development,” Better Software, vol. 8, no. 3, 2006.

[45] 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.

[46] M. Fowler, Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., 2002.

[47] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, “Design Patterns: Elements of Reusable Object-Oriented Software,” Reading: Addison-Wesley, vol. 49, no. 120, 1995, p. 11.