# Microservice Development Based on Tool-Supported Domain Modeling

Michael Schneider, Benjamin Hippchen, Pascal Giessler, Chris Irrgang, Sebastian Abeck

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

***Abstract*—Developing complex business-related software solutions with domain-driven microservices has become popular recently. Based on the concepts of domain-driven design, the business is expressed as a domain model. However, domain-driven design does not mention any modeling guidelines or tools for assisting the design process. In addition, modeling a complex domain can lead to a complex domain model that is difficult to read and implement. To tackle the complexity of the domain model, we introduce a concept for splitting the domain model into several diagrams, and we apply formalization based on the Unified Modeling Language. Furthermore, we illustrate how the created domain model is transferred step by step into code.**

***Keywords–Domain-Driven Design; Modeling; Tool; Microservices; UML profile; Model to Code.***

## I. INTRODUCTION

Modeling the domain of a business unit is part of many design procedures and decisions in software development. For the development of microservices-based systems, Domain-Driven Design (DDD) is a suitable approach [1]. The concept of the domain model was clarified by Eric Evans in his book *Domain-Driven Design: Tackling Complexity in the Heart of Software* [2], and it was further refined by Vernon [3]. After DDD, everything goes for modeling the domain. This includes the activities or business processes, the information involved, and any restrictions that may appear. Therefore, the creation of a domain model helps not only to better build the software architecture through a mannequin-driven design but also to increase the understanding of the business area in which an application operates.

When one is modeling with DDD, there are no restrictions as to how to express the domain. However, DDD emphasizes that the domain implementation should represent the domain model. Without a systematic modeling approach, it is possible that the development of the domain model results in models that are not suitable for the implementation. Furthermore, our experiences have shown that modeling the domain without any tool support can lead to domain models that are different. One example of the differences relates to the used designations, such as notations, names, and elements. This makes collaboration on the models within a team challenging and makes the automatic code generation from the domain model impossible. A further step can entail automatically generating the code from the model. However, a certain degree of formalization of the model is required in order to generate code automatically. Formalization can ensure that the model and the code are synchronized. For example, the automatic generation of Java code when one uses a UML-compliant domain model could be possible [4]. To assist the modeling process of a formalized model, as well as automatic code generation, a tool should be used. Furthermore, interfaces of microservices, which often are RESTful APIs, can be derived [5] when one follows API guidelines [6]. In addition, in terms of a microservice architecture, it is important to maintain the domain model in order to maintain the microservices. Without a tool, this maintenance can be difficult. Therefore, applying a tool-supported domain model creation process can help to solve this problem. Furthermore, we discuss how to generate the code from the created domain models.

The paper is structured as follows. Section II presents related work and articles. Section III illustrates why there is the requirement of additional modeling elements. In addition, a suggestion for structuring the domain models is shown. Section IV discusses the UML profile enhancements. Section V explains the necessary steps for using the UML profile with the tool; moreover, this section illustrates the usage of the tool. Section VI discusses the conversion of the model into the code. Finally, Section VII gives a summary of the paper and surmises what the prospects are for future research.

## II. RELATED WORK

Our results presented in this paper are related to or were inspired by the work of several other authors. First of all in this section, we introduce DDD as our main software development approach for modeling patterns in greater detail. Its concepts are the basis for our research. After this section, a first step for formalizing DDD's domain modeling is evaluated. Furthermore, we explain how our systematic approach is based on model-to-code approaches, such as the Model-Driven Architecture (MDA).

### A. Domain Modeling with Domain-Driven Design

DDD is a software development approach introduced by Evans [2] that emphasizes the design phase in modeling activities. Model activities aim at gathering information about a given customer's domain–the so-called domain knowledge. The domain knowledge is stated in a domain model, the central artifact of DDD. In line with the principles of Evans, only business logic is relevant to the domain model. Other information, such as technical aspects of applications is neglected. DDD provides several stereotypes of domain objects; a domain object represents a business object from the real world. Classifying the domain objects is important for both

 ISBN: 978-1-61208-701-6 

the domain model and the implementation of the application. The stereotype is stated in the domain model at the modeling element. Two of the most important stereotypes are "entities" and "value objects". Entities are real-world objects with an identity; this identity enables to find the specific instance of this real-world object. This identity never changes for these kinds of domain objects. Value objects describe also real-world objects, but an identity is not necessary in this case. A more detailed look at these stereotypes has been provided by Vernon [3]. Derived from this stereotype, the implementation is adapted accordingly.

DDD offers a substantial number of useful patterns that help to understand the domain and manifest it into a model. Nevertheless, one major problem of DDD is the missing modeling guideline, such as a specified modeling language. Actually, Evans emphasizes the use of any kind of representation for domain models as long as it supports the understanding of the customer's domain. When one examines the domain models in [2], they mostly remind one of UML class diagrams, but Evans has never stated that UML acts as modeling syntax. In Section IV, we build on a DDD-based UML profile to tackle the missing modeling guidelines.

### B. Formalization of Domain-Driven Design's Domain Model

While DDD does provide useful patterns to model the domain, the application for the representation within the model is challenging. The look and feel of domain models differ from development team to development team. Especially when one develops applications in a microservice architecture, it is necessary that development teams have a common understanding of how to model a given customer's domain. Thus, applying these patterns would be more efficient with the help of a formalized modeling language.

To tackle these problems, [7] has provided a first Unified Modeling Language (UML) profile for DDD. Based on the domain models used by Evans in [2], the authors have created an overview of which UML elements are used, and they have derived their domain-driven MSA modeling (DDMM); MSA stands for "microservice architecture." More or less, Evans has used UML modeling elements which has led to the decision that a UML profile would close the lacking modeling guidelines.

The UML profile provided by [7] presents an inspiring first step for closing the modeling language gap for DDD. Nevertheless, we can see further room for improving the UML profile. When one considers a complex domain, modeling it in a domain model can lead automatically to a complex model. Thus, we have introduced a concept called "relation view" that decreases the complexity of the domain model by splitting the model apart. Further, we have provided a concrete example for a modeling tool that is able to apply UML profiles.

### C. Model-to-Code Transformation

The classic approach for model-to-code transformation is directly associated with an Object Management Group's (OMG) Model-Driven Architecture (MDA) [8]. MDA is a software development approach that emphasizes the use of models. Different types of models have different purposes in the software development phases. Furthermore, depending on the model's type, the depth of details is more fine-grained or coarse-grained. The idea behind MDA is to focus on the modeling aspects, while software development for providing (domain) knowledge rich models. As a next step, the source code can be generated automatically based on the knowledge within these models. Previous research has claimed a great number of advantages for MDA-based software development [9], but the establishment in software development companies has proven that this approach has its own problems to apply. The main problem is that the automated generation of source code is not well realized. Often source code has to be adjusted to either work or fit to the problem modeled in the model. Due to this knowledge about automated model-to-code transformations, we elected a systematic (not automated) approach to transform a model into the source code. We provided a fix structure (for example, packages) for the microservice's source code.

## III. Structuring and Modeling of the Domain

Domain-driven design differs between several types of objects that we translate into a systematic modeling approach. A domain may contain several domain objects located in distinct bounded contexts. Modeling each domain object into only one diagram may lead to a complex and incomprehensible diagram. Therefore, we separate the domain model into different diagrams: for instance, the relation view for modeling the structural domain aspects.

### A. Systematic Domain Structure

When modeling the domain, several diagrams are created. Figure 1 shows a simplified version of a so-called "relation view", a tactical diagram concerning the to-do list domain. The to-do list domain is concerned with managing to-do lists as well as the to-dos themselves. The considered domain is simple and easy to understand, but the handling of the relation view can be shown well. Especially for modeling larger and complex domains we see a benefit for using the relation view diagram. Developers can work simultaneously on the different relation view diagrams. Tactical modeling focuses on a partial aspect of the domain within a bounded context, while strategical modeling concerns the higher-level structure of the domain model. A bounded context defines the scope of validity of the model and the code [10]. For each bounded context, we modeled the relation view as depicted in Figure 1.

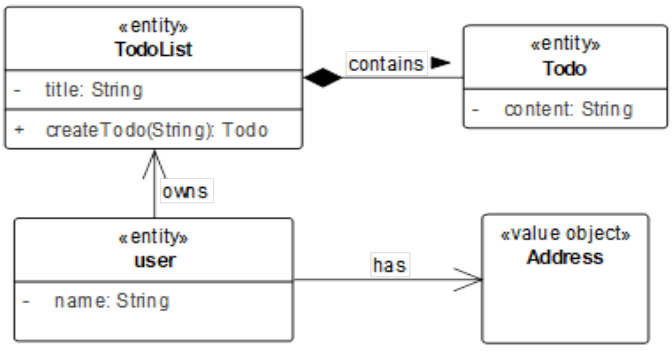


Figure 1. Extract of a relation view

The model is similar to a UML class diagram [2][7], but, in addition, the excerpt contains additional identifiers. These

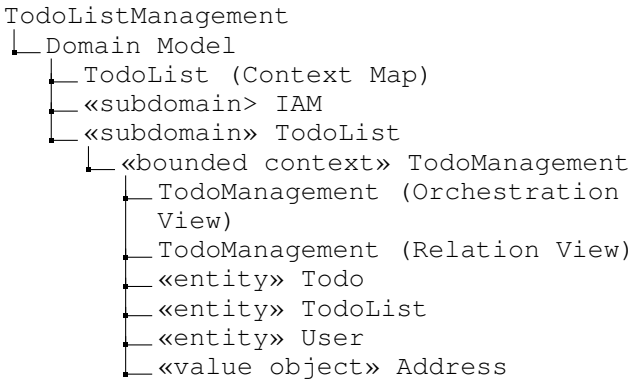


Figure 2. Domain structure

identifiers represent several elements that are required for modeling DDD, such as entities, value objects, and different kinds of relationships. Additionally, attributes and methods express the domain logic. For example, the method *createTodo(String):Todo* is responsible for creating a "to-do" that belongs to the to-do list.

The DDD elements need to be formalized in order to support tool-assisted modeling. In addition, the diagrams need to be stored in a structured way. Structuring the domain leads to several advantages, such as an easier communication across the team - thus, each team member knows exactly where the necessary diagrams are located. Therefore, we have provided a suggestion for structuring the different diagrams of the domain in order to increase retrievability and the value of the diagrams for the teams.

Each domain is structured in its own repository, comparable to folders and paths. Figure 2 illustrates the structure of the domain "TodoListManagement" of the *TodoListManagement* application. The repository contains strategical and tactical modeling diagrams. In this work, we only focus on the tactical diagrams, and we only briefly mention how the strategical modeling diagrams are placed in this structure. Each folder contains a domain model. The domain model is structured as follows. On the top level, the context map of the domain is shown. The context map is a concept of DDD [2] that contains bounded contexts related to a domain. In a microservice architecture, each bounded context is a candidate for a microservice that could be reused by other applications [11]. Using the context map diagram, the tool-support allows the easy access and navigation of the related diagrams simply by allowing one to click on the modeled bounded contexts.

Following the context map, all subdomains are located on the top level path. Figure 2 depicts two subdomains: identity and access management (IAM) as well as the TodoList, whereby the subdomain "TodoList" is unfolded. Each subdomain contains their related bounded contexts. In the example of the *TodoListManagement* the bounded context is called *TodoManagement*. Each bounded context contains diagrams concerning this bounded context (see Figure 2). The first diagram is a strategical diagram — the context orchestration that concerns the orchestration. In addition to the context orchestration, the tactical diagrams follow. Each bounded context consists of at least one relation view. The relation view is the tactical diagram that contains the domain elements and their relationships. In addition to the structural elements, the behavior of the domain behavior is modeled as well in different diagrams. For the dynamic components, ordinary sequence and activity diagrams can be used, which are, therefore, not considered further in the following analysis. The diagram elements corresponding to the relation view, such as entities, value objects, or relations, are located directly below the diagrams. These elements can be reused for all diagrams concerning the related bounded context.

## IV. UML Profile

In this section, the formalization of the model is discussed. Possible options are UML profiles or metamodels, but each has their own advantages and disadvantages. The creation of a UML profile is preferable to an extension of the metamodel due to the low added value. Therefore, we used a UML profile for our modeling purposes (see [7]) and added the relation view.

### A. UML Profile of the Relation View

The *relation view* describes the inner structure of a bounded context, essentially corresponding to a class diagram and representing the tactical part of DDD. A first approach for dividing the model into several views has already been mentioned by other scholars [1]. Many domains are complex and contain many domain objects. The relation view reduces the complexity for modeling the domain and should be used for complex domains. Only the domain objects corresponding to the current bounded context are considered in the relation view. Therefore, the relation view describes a manageable section of the domain. Each bounded context has at least one relation view, which can consist of entities, value objects, domain services, and their relationships. Therefore, the relation view defines the domain terms and correlates them into a relationship. The elements and relations used in the relation view are also already largely defined in UML. Established DDD concepts, such as entities and value objects, are supported with the UML profile. Due to the large intersection, only minor adjustments compared to UML are necessary since mainly new stereotypes have to be introduced and the behavior of existing elements, such as packages, components, and classes can be maintained. Thus, the power provided by heavyweight modeling using metamodels would hardly be used. The most important UML additions for the relation view are based on [7] and discussed in the following.

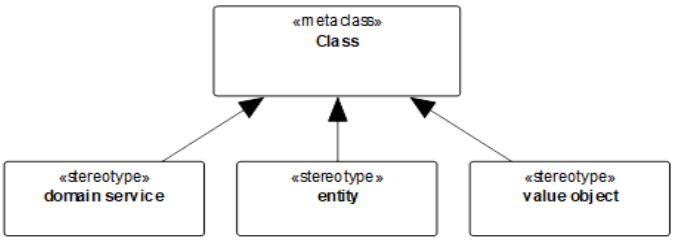


Figure 3. Profile of the relation view

*1) Entity:* An entity is one of the most important elements of the relation view. The entities represent the corresponding objects from the domain, are defined as UML classes, and include attributes and methods. They encapsulate all functionality associated with or emanating from this entity. Entities can be provided via the API. As Figure 3 illustrates, we added the stereotype "entity" to the UML class element. For example, the domain object "todo" (see Figure 1) is an entity; therefore, the modeling element for an entity (declared with the stereotype "entity") was added to the diagram.

*2) Value Object:* A value object behaves similarly to an entity but does not have an identity. Another difference between entities and value objects is that a value object is not immutable, and, therefore, a new object must be created when changes are made. This means that only the attributes are considered in object comparisons, which makes a comparison between two different objects with the same attribute values true. For example, an address consisting of first name, surname, street, and the city used by several people (in this example, "users") - could become a value object. In a different domain, the address could be an entity as well. Value objects are identified with the stereotype "value object."

*3) Domain Service:* Domain services are used when the responsibility of a process is incumbent upon several entities or value objects. A domain service does not maintain any state in order to guarantee consistent and predictable behavior. The stereotype "domain service" identifies a domain service.

*4) Relationships between Domain Objects:* The relation view does not contain any additional self-defined stereotypes for relationships. Instead, the most important relationships from the UML class diagrams are used. This includes the generalization, the composition, the aggregation, and the binary association. Relationships between the domain elements are usually defined by means of a verb and the reading direction. In addition, multiplicities and directions are assigned in the same manner as in a UML class diagram.

## V. Used Tool and Example

This section presents Enterprise Architect (EA) [12], which is the tool we are have used for modeling the domain. Furthermore, this section explains how the UML profile is applied in the tool.

### A. Enterprise Architect

EA is a software modeling tool that is based on OMG UML [13]. By default, Enterprise Architect provides support for user-defined extensions, including the use of UML profiles. Enterprise Architect already provides some useful profiles for popular modeling languages, such as Business Process Model and Notation (BPMN), Systems Modeling Language (SysML), or ArchiMate.

### B. Enterprise Architect Profiles (MDG)

Besides the UML profile discussed in Section IV, further profiles are required. In order to create a UML profile for DDD with optimal user experience, additional diagram and toolbox profiles are required next to the previous (see Section IV) UML profiles. These diagram and toolbox profiles are specified in EA. The diagram profiles allow the easy creation of custom diagram types that are suitable for the DDD modeling problem. Figure 4 depicts an excerpt of the definition of the custom

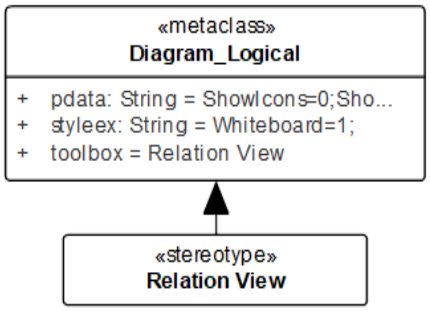


Figure 4. Excerpt of the EA DDD profile

diagram profile for the relation view. The custom diagrams illustrated in Figure 4 extend the standard UML diagram metaclasses and predefine the appearance and feature visibility of the diagram elements. These diagrams can be accessed via toolboxes. Therefore, in addition to the profiles, a toolbox profile is specified. The toolbox profile links an arbitrary diagram type to a custom-built toolbox. Opening a diagram type automatically displays the corresponding toolbox. This toolbox contains the configured elements and connectors. For example, the toolbox of the relation view contains elements, such as "Entity", "Value Object", and "Domain Service". Figure 5 illustrates our defined toolbox for the relation view. Thus,

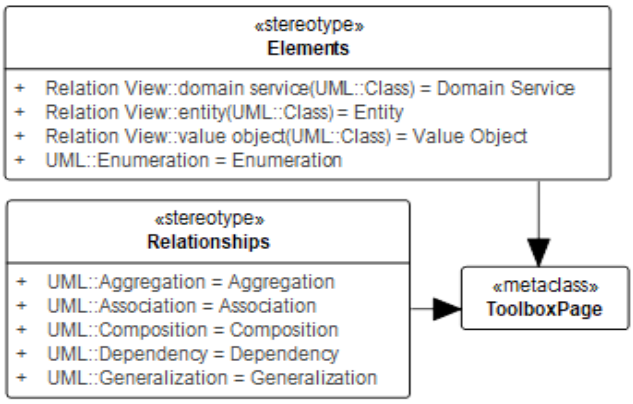


Figure 5. Toolbox profile for relation view

the user has only the necessary self-defined and predefined standard modeling elements available; this, in turn, simplifies the modeling process and reduces modeling inconsistencies.

### C. Modeling with EA

In order to model the different diagram types, the corresponding DDD profile is loaded. For example, the relation view is modeled by using the previously defined relation view toolboxes. This simplifies the modeling process because EA offers many modeling elements. The result of the toolbox defined in Section V-B is illustrated in Figure 6. The toolbox contains all the previously discussed elements as well as the relationships. For modeling purposes, the elements are simply dragged out of the toolbox into the diagram. In Section III, several domain objects are illustrated in Figure 1 – “TodoList,” “Todo,” “User,”

 ISBN: 978-1-61208-701-6

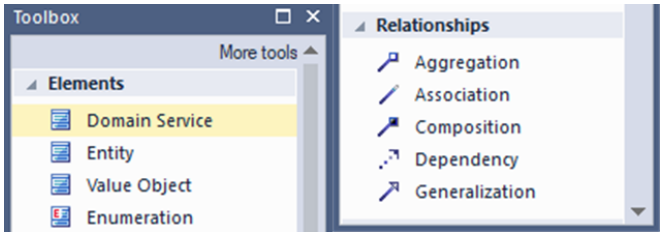


Figure 6. Resulting EA toolbox for the relation view

and “Address” – as well as several relationships. The diagram itself was created by using EA and the corresponding DDD profile.

## VI. Model-To-Code and Code-Model Equality

The tool-supported domain modeling based on an extended UML profile enables the possibility of generating code directly from the model. Enterprise Architect and other tools automatically generate classes, attributes, and methods (including parameters) from the model. Using this generation process ensures that the model and code are equal at the point of the code creation process. During implementation, there is a high chance that the developers will notice that the modeled methods, attributes, and classes are not enough or that they require changes. Therefore, the code should be adjusted according to the needs. Adding new classes, methods, and attributes to the implementation does not automatically adjust the model. If the model is not adjusted afterwards, the model is no longer useful as a convenient reference. This model state is not desired; therefore, an interface between the tool and the code is required that can automatically adjust the model when code changes happen. EA allows importing source code that can be used to automatically create a model. The imported source code creates a new model, but it does not adjust the model used to generate the code.

### A. Step-Wise Model-To-Code Transformation

For the implementation of the todo list domain, we used Java as a programming language and the framework Spring [14], which simplifies the developing of enterprise applications. The relation view that we created was transferred step by step into code. Figure 7 shows the different steps for implementing the domain model. To simplify the implementation process, we implemented an entity-base class that provides useful DDD functionality and can reduce the boilerplate code of the microservice implementation. As depicted in Figure 8 line 2, this base class is inherited and provides useful classes, configurations, and methods (for example, an ID and corresponding equals- and hashCode methods,) and it simplifies the microservice-based development with Spring Boot. The annotation *@Entity* in line 1 enables the mapping to a database by an ORM framework and is used for domain objects that are entities. In addition, the annotation *@ValueObject* is used for value objects.

In the next step, the infrastructural annotations were added. As shown in line 7 of Figure 8, domain database annotations define the relationships and cardinalities between the domain objects for the database. These database-specific annotations are sufficient in this simple case since the domain can be

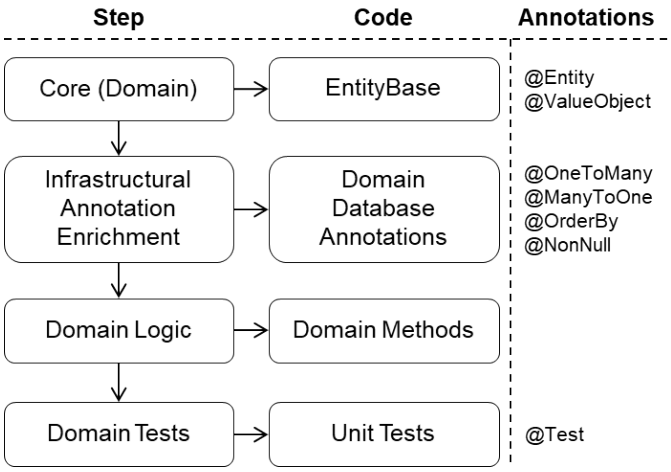


Figure 7. Implementation steps and code annotations

mapped to the database as it is–a structural adjustment of the class by database restrictions did not take place accordingly. The annotations were added to relating domain objects, for example, *TodoList* and *Todo*. After this step, the methods that contained the domain logic were implemented. It should be noted that the class does not contain any "getters" and, especially, no "setter" methods because business methods are not covered with simple setter methods [3]. Getters are only used if they are really necessary for fulfilling business capabilities and cannot be queried indirectly by the use of business methods.

```
1  @Entity
2  public class TodoList extends EntityBase {
3
4   @Column(nullable = false)
5   private String title;
6
7   @OneToMany(cascade = CascadeType.ALL,
8    mappedBy = "todoList",
9    orphanRemoval = true)
10  private List<Todo> todos;
```

Figure 8. Excerpt of the TodoList implementation

Once the implementation of the domain model were completed, the implementation was tested. At the domain level, unit tests were used for the testing purpose that focused on the formal correctness of the domain at a technical level and ensured the correct behavior of the domain implementation. For aggregate elements, the root element methods were called in order to test the domain logic within the aggregate.

### B. REST-based Web API

According to [15] and with consideration for the Richardson maturity model (RMM) [16], several questions have to be answered to provide a REST-based Web API: 1) Which domain objects should be exposed? 2) Which information from these selected domain objects should be exposed? 3) Which methods of the domain should be provided?

 ISBN: 978-1-61208-701-6

TABLE I. END POINTS OF THE TO-DO LIST EXAMPLE

| Entity | Collection | Resource ID |
|---|---|---|
| TodoList | /todo-lists | /{id} |
| Todo | /todo-lists/{id}/todos | /{number} |

Once these questions have been clarified, an initial specification of the Web API can be derived. For the specification, the domain objects will be mapped onto so-called "resources" that act as data transfer objects (DTOs) that contain (partial) information of the respective domain objects. By using such an approach, we introduce an abstraction layer so that the domain objects can develop independently of each other without a necessary Web API change. A Web API change can result in a negative side-effect for existing service customers if the provided methods are changed. The domain object methods are mapped to corresponding HTTP methods to reflect on the operation semantically (create, read, write, update, delete, execute). The positive impact of a good Web API is uncontroversial especially when offering the underlying service to a wide range of possible service customers. That is why several companies apply dedicated review cycles and also create guidelines on how to build Web APIs with quality in mind [17][18]. There is also an aggregation of well-known best practices that should be kept in mind during the design process [6][19][20]. For the purpose of formalization and further processing, dedicated specific languages, such as OpenAPI can be used, which, in turn, come with corresponding tool support. For instance, dedicated client libraries can be automatically generated to simplify the integration.

In our case, we derived two end points for our to-do list example, as illustrated in Table I. Using the tool *SwaggerUI*, we could visualize and interact with the Web API with no written code from client side. The end points of the to-do list example are displayed in Table I. Requests to the entity *Todo* are always passed over to-do lists because the entity *TodoList* is the aggregate root.

## VII. Conclusion and Limitation

We focused on the tactical modeling of a domain based on UML profiles in order to formalize the modeling process. Our approach has allowed us to divide the domain model into multiple models; this has allowed us to develop different models simultaneously, which reduces the complexity of the modeling process. In order to model the diagrams, we used EA as modeling tool; this enables an automatic translation of the model into code. For entities and value objects, we implemented base classes that provide useful functionality for these domain concepts. However, at this point we transferred the model into code manually. For an automatic code generation, additional work is required. Currently, we are only able to automatically generate the classes, methods, and attributes. Further research is required, because the annotations and mappings could be automatically added by the modeling tool as well. The annotations remove boilerplate code from the implementation, but the tool needs to automatically provide the annotation when code is generated. For example, the annotations *@Entity, @ValueObject* need to be automatically generated. In addition, relationships and database annotations need to be considered as well. In future work, we need to add the mappings that are required to automatically create the complete code from the toolset we presented.

Since we focused on tactical models only, strategical modeling and the implementation aspects should be investigated to a greater extend in further research.

## References


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

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

[3] V. Vernon, Ed., Implementing Domain-Driven Design. Addison-Wesley, 2013, ISBN: 978-0321834577.

[4] M. Usman and A. Nadeem, “Automatic generation of Java code from UML diagrams using UJECTOR,” International Journal of Software Engineering and Its Applications, vol. 3, no. 2, 2009, pp. 21–37.

[5] P. Giessler, “Domain Driven Design of Resource-oriented Microservices,” Ph.D. dissertation, Karlsruhe Institute of Technology, Germany, 2018.

[6] P. Giessler, M. Gebhart, D. Sarancin, R. Steinegger, and S. Abeck, “Best Practices for the Design of RESTful Web Services,” in International Conferences of Software Advances (ICSEA), 2015, pp. 392–397.

[7] F. Rademacher, S. Sachweh, and A. Zündorf, “Towards a UML Profile for Domain-Driven Design of Microservice Architectures,” in International Conference on Software Engineering and Formal Methods. Springer, 2017, pp. 230–245.

[8] A. G. Kleppe, J. Warmer, W. Bast, and M. Explained, The model driven architecture: practice and promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2003.

[9] K. Czarnecki and S. Helsen, “Classification of model transformation approaches,” in Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, vol. 45, no. 3. USA, 2003, pp. 1–17.

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

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

[12] Sparx Systems, “Enterprise Architect - Model Driven UML Tool,” URL: https://www.sparxsystems.eu/start/home/ [retrieved: 2019.03.15].

[13] O. OMG, “Unified Modeling Language (OMG UML),” Superstructure, 2007.

[14] Pivotal Software, “Spring Framework,” URL: https://spring.io/projects/spring-framework/ [retrieved: 2019.03.15].

[15] R. T. Fielding, “REST: architectural styles and the design of network-based software architectures,” Doctoral dissertation, University of California, Irvine, 2000, URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm [retrieved: 2019.01.31].

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

[17] A. Macvean, M. Maly, and J. Daughtry, “API Design Reviews at Scale,” in Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems. ACM, 2016, pp. 849–858.

[18] Zalando, “Zalando RESTful API and Event Scheme Guidelines,” 2017, URL: https://zalando.github.io/restful-api-guidelines/ [retrieved: 2019.01.31]. [Online]. Available: https://zalando.github.io/restful-api-guidelines/

[19] M. Masse, REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces. " O’Reilly Media, Inc.", 2011.

[20] P. Giessler, M. Gebhart, R. Steinegger, and S. Abeck, “Best Practices for the Design of RESTful Web Services,” International Journal On Advances in Internet Technology, vol. 9, no. 3 and 4, 2016.


 ISBN: 978-1-61208-701-6 