Towards a UML Profile for Domain-Driven Design of Microservice Architectures

Florian Rademacher[1(✉)], Sabine Sachweh[1], and Albert Zündorf[2]

[1] Institute for Digital Transformation of Application and Living Domains, University of Applied Sciences and Arts Dortmund, Dortmund, Germany
{florian.rademacher,sabine.sachweh}@fh-dortmund.de
[2] Department of Computer Science and Electrical Engineering, Software Engineering Research Group, University of Kassel, Kassel, Germany
zuendorf@uni-kassel.de

Abstract. Domain-driven Design (DDD) is a model-driven approach to software development that focuses on capturing the application domain, its concepts and relationships in the form of domain models for architecture design. Among others, DDD provides modeling means for decomposing a domain into Bounded Contexts and expressing the relationships between them. With the recent emergence of Microservice Architecture (MSA), DDD again gains broad attention because a Bounded Context naturally maps to a Microservice, which enables the application of DDD for MSA design.

However, DDD is not a formal modeling language. Instead, it leverages informal UML class diagrams to express domain models, which prevents model validation and transformation. In this paper we address this limitation by providing an initial UML profile for Domain-driven MSA Modeling. Together with a survey on the UML constructs used in DDD, the profile denotes a foundation for validating domain models and deriving Microservice code from them.

Keywords: Domain-Driven Design · Microservice architecture UML profile

1 Introduction

Domain-driven Design (DDD) [3] is an approach to software development that focuses on the application domain, its concepts and their relationships as primary drivers for architecture design. Core principles of DDD comprise (i) capturing relevant domain knowledge in *domain models* that might comprise structural and behavioral aspects; (ii) collaborative modeling of domain experts and software engineers; (iii) fostering experimental design by strictly aligning model and implementation throughout the software development process as well as continuous model refinement; (iv) fostering communication between domain experts and software engineers by jointly defining an explicit *ubiquitous language*, which

© Springer International Publishing AG 2018
A. Cerone and M. Roveri (Eds.): SEFM 2017 Workshops, LNCS 10729, pp. 230–245, 2018.
https://doi.org/10.1007/978-3-319-74781-1_17

consists of relevant domain-specific terms and is used in both, domain models and implementation.

As a set of model-driven practices, techniques and principles for software design, DDD has been defined by Evans in 2004 [3]. With Microservice Architecture (MSA) as an architectural style for distributed, service-based software systems [9], that is gaining broad attention of both practitioners and scientists as of 2014 [13], the relevance of DDD recently increases. This is due to DDD providing various modeling patterns and techniques for the identification of coherent domain concepts and their encapsulation within conceptual boundaries that might serve as foundation for MSA-based service decomposition [9].

Thereby, Evans proposes to express domain models that capture structural domain knowledge in the form of UML class diagrams [3]. He therefore leverages a subset of standard UML elements, which are partially enriched with DDD-specific semantics to define *DDD patterns*. However, the pattern definitions lack a formal, UML-based foundation and sometimes differ in their notations. While the absence of a formal foundation leads to a high degree of freedom concerning syntaxes and semantics of DDD-specific modeling elements [3], it prevents structured model operations like validation [16] and transformation [8]. Hence, further usage of domain models next to being integral parts of stakeholder communication and domain documentation is hampered.

To overcome this limitation, we present an initial, twofold contribution towards a formalization of DDD for *Domain-driven MSA Modeling* (DDMM). First, we provide a survey regarding syntaxes, semantics and frequency of UML elements applied in DDD for capturing domain models. This defines a basic set of modeling constructs to consider when processing domain models for validation or transformation purposes. Second, we define a UML profile for DDMM and discuss its usage to model Microservices and derive interactions between them.

The remainder of the paper is organized as follows. Section 2 gives an overview of DDD and how it is applied for MSA. Section 3 presents the findings of a literature survey regarding syntaxes, semantics and occurrences of UML constructs in DDD domain models. In Sect. 4 we introduce the UML profile for DDMM. Section 5 presents related work and Sect. 6 concludes the paper.

2 Domain-Driven Design

In this section we elaborate on DDD as an approach to abstracting a domain in the form of *structural domain models* that describe structure and relationships of domain concepts [3]. We also describe the *Bounded Context pattern* that is commonly proposed for modeling services in MSA [9].

2.1 Structural Domain Models

In DDD, a *domain model* is a rigorously organized, selective abstraction of conceptual knowledge about a domain or relevant parts of it [3]. Basically, the notation to express domain models is not bound to a certain modeling language. However, Evans proposes to use UML class diagrams to capture *structural domain*

models, which leverage UML classes, attributes and methods to model domain concepts, and UML associations, multiplicities and collection specifications to express concept relationships. Figure 1 shows a preliminary structural domain model for a cargo shipping system described in [3].

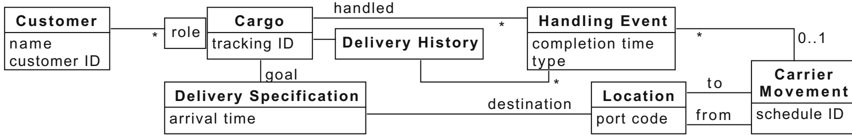

Fig. 1. Preliminary structural domain model for a cargo shipping system [3]

Each class represents a *domain object*, which in DDD is synonymous with "domain concept" [3]. The model contains the core domain objects and their relationships. For example, it shows that a `Cargo` has a `tracking ID` and is associated with a set of `Customers`, each distinguished by its `role`, e.g. "shipper" or "receiver". Assigned to a `Cargo` is a `Delivery History` that tracks cargo-related `Handling Events`, which might involve at most one `Carrier Movement` from a source to a target `Location`. Furthermore, a `Cargo` has a `goal`, i.e. a `Delivery Specification` with a destination `Location`.

On the basis of certain UML class diagram elements, DDD introduces a variety of patterns to enrich a structural domain model with further semantics for Model-driven Design [3]. These patterns and their definition by means of the UML 2.5 metamodel [12] are described in Table 1.

Figure 2 shows an excerpt of the cargo shipping model with refined associations and extended by a selection of DDD patterns [3].

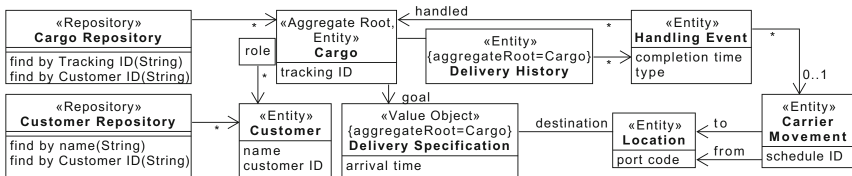

Fig. 2. Excerpt of refined cargo shipping model with additional DDD patterns [3]

All domain objects are annotated with pattern-specific stereotypes to identify them as Entities, Value Objects, Aggregate roots or Repositories [3]. For example, the Entity `Cargo` is also a root for the Aggregates `Delivery Specification` and `Delivery History`, which may only be accessed via the root object. The `Cargo Repository` models the retrieval of `Cargoes` by `tracking ID` and `customer ID`. While most domain objects are Entities, `Delivery Specification` is a Value Object to communicate that two `Cargoes` might share

Table 1. DDD patterns and their UML 2.5 metamodel [12] equivalents. Note: "Annotated" stands for any mechanism that allows to assign additional meaning to UML modeling elements, e.g. stereotypes or comments.

Pattern	UML metamodel equivalent	Description
Aggregate	Associated Classes with annotated root Class	Cluster of associated Entities and Value Objects. An Aggregate is treated as a whole when being accessed by referencing its root Entity
Closure of Operations	Annotated Operation	A Closure's return type is of the same type as its arguments and provides an interface without additional domain object dependencies
Entity	Annotated Class	An instance of the domain object is distinguished from other instances by its identity. Identity determination is domain-specific
Module	Annotated Package	Encapsulation mechanism whose primary goal is to reduce cognitive overload in domain models by partitioning cohesive sets of domain objects
Repository	Annotated Class with outgoing Associations to other Classes	Models access to persistent domain object instances via operations that perform instance selection based on given criteria
Service	Annotated Class containing only Operations	Services encapsulate processes or transformations that are not in the responsibility of Entities or Value Objects
Side-effect-free Function	Annotated Operation	Expresses that a domain object's Operation does not have any side effects on a system's state
Specification	Annotated Class depending on specified Class	Used to determine if a domain object instance fulfills a specification. Contains a set of Boolean Operations to perform specification checks
Value Object	Annotated Class	Typically immutable object without domain-specific identity. Might act as value container

the same `Delivery Specification`, but with most likely differing `Delivery Histories`. Otherwise the `Cargoes` would exhibit the same identity [3].

2.2 Domain-Driven Design for Microservice Architecture

In contrast to Service-oriented Architecture (SOA), MSA imposes explicit requirements on service granularity [14]. Each Microservice should realize exactly one capability of the software system that is clearly distinct from others. The goal of business-related service decomposition in MSA is to cluster related domain objects and functionalities in isolated *functional Microservices*.

For modeling functional Microservices and domain objects exchanged between these, i.e. *shared domain objects*, DDD's *Bounded Context* pattern is predestined [9] and has become a common means for determining and expressing MSA-based service granularity prior to service decomposition [2,4,5].

Next to Modules (cf. Table 1), Bounded Contexts are another encapsulation mechanism of DDD. While Modules solely structure domain objects in different namespaces, Bounded Contexts define *scopes* for enclosed domain objects, i.e. boundaries for object validity and applicability [3].

The boundaries of a Bounded Context typically impact team, code and application organization [3]. Only the team responsible for a context may change its internal structure. It is further responsible for context implementation and interface provisioning on the basis of shared domain object models. As these responsibilities correspond to those of a Microservice [9], a Bounded Context provides the foundation for the domain-specific implementation of a Microservice. Figure 3 shows a version of the cargo shipping model that has been decomposed into Bounded Contexts depicted as UML packages.

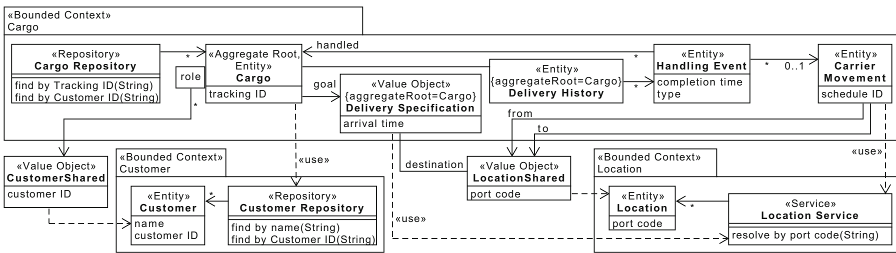

Fig. 3. Cargo shipping model decomposed into several Bounded Contexts

Relationships between the Bounded Contexts are expressed as shared Value Objects, i.e. instances of `CustomerShared` and `LocationShared` act as containers for exchanging values between contexts (cf. Table 1). Each shared object depends on the Entity it represents, i.e. `Customer` and `Location`. For the retrieval of shared object instances, `Cargo` uses the existing `Customer Repository`, while instances of `LocationShared` can be requested from the

`Location` context via the introduced `Location Service` (cf. Table 1). It dynamically resolves the `port code` from a given argument, i.e. `Locations` are not stored in a Repository.

3 Survey on UML Elements in Domain-Driven Design

In the following, we present an overview and characterization regarding syntaxes, semantics and frequency of UML elements used by DDD to model structural domain models. We identified the elements by surveying each of the 92 UML class diagrams in [3] representing real-world structural domain models. Thereby, we left out the 29 diagrams showing domain object interactions as they are (i) modeled with various notations differing in the degree of formality, e.g. object interaction, UML sequence and domain-specific diagrams; (ii) used to exemplify interactions between few selected objects rather than in a comprehensive architectural design; (iii) not applicable for identifying functional Microservices and their structural relationships (cf. Subsect. 2.2).

Together with the DDD patterns described in Sect. 2, the UML elements identified in our survey define a basic set of modeling constructs to be considered in UML-based DDMM, e.g. when validating domain models or deriving code.

Table 2 shows the results of our survey. It lists each UML element used in the domain models in [3] and classifies them on the basis of six categories representing basic UML concepts, i.e. "Associations", "Attributes", "Classes", "Constraints", "Methods" and "Multiplicities". We further state the occurrence count per element, that is the number of domain models comprising it at least once, as well as its representation with UML 2.5 metaclasses [12] to be considered in UML-based DDMM. Due to space limitations, we do not present survey results for modeling constructs used in DDD domain models that are not conform to UML 2.5 and hence might not be validly expressed leveraging its metamodel, e.g. abstract attributes or named extensions between classes. As only four out of 92 domain models (4.34%) comprise such elements, we view them as negligible.

Next, we describe category-specific characteristics of conform UML elements.

3.1 Classes

In structural domain models, DDD expresses domain objects as named Classes. Hence, every domain model contains at least one Class, which makes this element the predominant UML construct in DDD. Thereby, a Class might be modeled as being abstract to specify Methods (cf. Subsect. 3.5) that have to be realized by Sub-classes and enable polymorphism in domain model implementations. A special case of abstract classes are generalized Specifications (cf. Table 1) where different instances of a domain object need to satisfy different specifications, e.g. an `Invoice` for which two Specifications `DelinquentInvoiceSpec` and `BigInvoiceSpec` inheriting from a general `InvoiceSpec` are modeled, that specify a date or a threshold amount, respectively [3].

A DDD Class corresponds to the UML metaclass `Classifier`. For abstract Classes, `Classifier.isAbstract` is set to `true`.

Table 2. Results of surveying the 92 UML class diagrams in [3] for UML elements used in structural domain models. Table ordering is based on elements' occurrence count.

<table>
<tr><th>Element</th><th>Category</th><th>Occurrence Count</th></tr>
<tr><td>Class</td><td>Classes</td><td>92 (100%)</td></tr>
<tr><td colspan="3">UML Metamodel Representation: <code>Classifier</code> with <code>name</code>. <code>Classifier.isAbstract</code> may be set to <code>true</code>.</td></tr>
<tr><td>Attribute</td><td>Attributes</td><td>72 (78.26%)</td></tr>
<tr><td colspan="3"><code>Property</code> with <code>name</code>, <code>type</code> or both. <code>Property.isDerived</code> may be set to <code>true</code>. A <code>MultiplicityElement</code> may be used to specify that the <code>Property</code> value is optional.</td></tr>
<tr><td>Multiplicity</td><td>Multiplicities</td><td>67 (72.82%)</td></tr>
<tr><td colspan="3"><code>MultiplicityElement</code> with <code>ValueSpecification</code> for <code>lowerValue</code> and <code>upperValue</code>.</td></tr>
<tr><td>Non-navigable Association</td><td>Associations</td><td>58 (63.04%)</td></tr>
<tr><td colspan="3"><code>Association</code> with optional <code>name</code> and without specified navigability. <code>ownedEnd</code> comprises two possibly named <code>Properties</code>, of which one can have a <code>qualifier</code>. To add a collection specification, <code>ValueSpecification.isOrdered</code> of an assigned Multiplicity may be <code>true</code>.</td></tr>
<tr><td>Method</td><td>Methods</td><td>49 (53.26%)</td></tr>
<tr><td colspan="3"><code>Operation</code> with <code>name</code> that may have <code>isAbstract</code> set to <code>true</code>. <code>ownedParameter</code> may contain <code>Parameters</code> with <code>name</code>, <code>type</code> or both and <code>direction</code> typically set to <code>in</code>. One <code>Parameter</code> may have it's <code>direction</code> set to <code>return</code>.</td></tr>
<tr><td>Unidirectional Navigable Association</td><td>Associations</td><td>36 (39.13%)</td></tr>
<tr><td colspan="3"><code>Association</code> with optional <code>name</code> and one <code>Property</code> in <code>navigableOwnedEnd</code>. <code>ownedEnd</code> comprises two possibly named <code>Properties</code> with a possible <code>qualifier</code>. To specify a collection, <code>ValueSpecification.isOrdered</code> of an assigned Multiplicity may be <code>true</code>.</td></tr>
<tr><td>Non-navigable Aggregation</td><td>Associations</td><td>28 (30.43%)</td></tr>
<tr><td colspan="3"><code>Association</code> without <code>name</code>. <code>ownedEnd</code> comprises two possibly named <code>Properties</code> with one having <code>aggregation</code> set to <code>shared</code> and a possible <code>qualifier</code>. To specify a collection, <code>ValueSpecification.isOrdered</code> of an assigned Multiplicity may be <code>true</code>.</td></tr>
<tr><td>"extends" Relationship (Inheritance)</td><td>Associations</td><td>23 (25%)</td></tr>
<tr><td colspan="3"><code>Generalization</code> relationship between two <code>Classifiers</code> (single inheritance). The specific <code>Classifier</code> holds a <code>Generalization</code> with <code>general</code> pointing to the general <code>Classifier</code>.</td></tr>
<tr><td>Informal Constraint</td><td>Constraints</td><td>13 (14.13%)</td></tr>
<tr><td colspan="3">Informal domain-specific constraints like "Itinerary must satisfy specification" may be specified as <code>names</code> of <code>Associations</code> or <code>Dependencies</code> where <code>client</code> constrains <code>supplier</code>.</td></tr>
<tr><td>Unidirectional Navigable Aggregation</td><td>Associations</td><td>11 (11.95%)</td></tr>
<tr><td colspan="3"><code>Association</code> without <code>name</code>. <code>ownedEnd</code> comprises two possibly named <code>Properties</code>, with one being in <code>navigableOwnedEnd</code>, having <code>aggregation</code> set to <code>shared</code> and an optional <code>qualifier</code>. May have a collection specification like Non-navigable Aggregation.</td></tr>
<tr><td>Formal Constraint</td><td>Constraints</td><td>6 (6.52%)</td></tr>
<tr><td colspan="3">Formal domain-specific constraints like "Bucket.contents <= Bucket.capacity" may be specified as (i) <code>body</code> of <code>Comment</code> with <code>annotatedElement</code> being <code>Classifier</code> or <code>Property</code>; (ii) <code>name</code> of <code>Dependency</code> whose <code>supplier</code> is an <code>Association</code> and <code>client</code> is a <code>Classifier</code>.</td></tr>
<tr><td>Semi-formal Constraint</td><td>Constraints</td><td>3 (3.26%)</td></tr>
<tr><td colspan="3">Semi-formal domain-specific constraints like "sum of item <= approved limit" are specified as <code>names</code> of Unidirectional Navigable Associations or Aggregations.</td></tr>
<tr><td>Class Dependency</td><td>Associations</td><td>2 (2.17%)</td></tr>
<tr><td colspan="3"><code>Dependency</code> without <code>name</code> and both <code>supplier</code> and <code>client</code> being <code>Classifiers</code>.</td></tr>
</table>

3.2 Associations

The elements in this category are applied in 87 of the 92 domain models (94.56%), which makes Associations the second most occurring UML construct in DDD. Associations are used to specify relationships between exactly two domain objects. A special form of Associations are Aggregations in the sense of UML *shared aggregations* [12], whose semantics depend on application area or modeler. Aggregations group together a set of assigned domain object instances.

Most Associations lack an explicit specification of navigability, which otherwise is always unidirectional. An Association end may exhibit an "ordered" collection specification and be qualifying to partition a set of assigned instances, e.g. `Customers` by their `role` (cf. Fig. 1).

Next to Associations, the category comprises Inheritance relationships and Dependencies. Both establish Associations between Classes and are applied corresponding to the UML specification of `Generalization` and `Dependency`.

3.3 Attributes

Attributes represent *structural features* of domain objects. For the majority of Attributes, no type is specified, which increases the level of modeling flexibility but complicates domain model processing. For example, when generating code from domain models, e.g. in an object-oriented language like Java, untyped Attributes might be assigned a generic type like Java's `Object`. However, this prevents type-safety and relies on the semantics of an Attribute being sufficiently communicated by its name. A facing issue is DDD possibly specifying unnamed Attributes that only have a type. The meaning of an Attribute may then remain unclear, especially when its type is not domain-specific, e.g. `Double` instead of `MoneyAmount` [3].

Attributes may be modeled as derived or optional. Thereby, a derivation specification is missing and optional Attributes' names are terminated by "(opt)".

Attributes correspond to the UML metaclass `Property`. For derived Attributes, `Property.isDerived` is set to `true`. Optional Attributes may be specified by assigning a `MultiplicityElement` to the `Property` with a `lowerValue` of `0` and an `upperValue` of `1` or `*`.

3.4 Multiplicities

All occurrences of Multiplicity specifications in domain models conform to UML. Typically, an Association or Attribute is provided with Multiplicities. Multiplicity specifications correspond to UML's metaclass `MultiplicityElement`, whose properties `lowerValue` and `upperValue` reference instances of `ValueSpecifications` that represent an `Integer` and an `UnlimitedNatural`, respectively.

3.5 Methods

DDD leverages Methods to model the interfaces of domain objects' *behavioral features*. Thus, concrete behavior specifications are omitted and Methods are only represented by their type signatures. Methods correspond to UML's metaclass `Operation`, possibly comprising a set of `Parameters`. Parameter names and types are mutually optional, which is conform to UML but raises the same issues as for unnamed and untyped Attributes (cf. Subsect. 3.3). `Parameters` are modeled as incoming or returning, i.e. with `direction` set to `in` or `return`.

3.6 Constraints

We classify Constraints used by DDD depending on their degree of formality.

Informal Constraints are formulated in natural language and modeled as names of Associations or Dependencies. In the latter case, the dependent domain object always constrains the independent object, e.g. a `Route Specification` depends on an `Itinerary` stating that it must satisfy the Specification (cf. [3] and Table 1). This semantically makes the dependency bidirectional, because logically `Itinerary` depends on `Route Specification`, which it otherwise could not satisfy. This can be resolved in that the direction of the modeled Dependency is reversed, i.e. the specified object depends on the Specification (cf. Sect. 4).

Semi-formal Constraints are stated in natural language mixed with formal notations. Like Informal Constraints, they are modeled as names of Associations.

Formal Constraints leverage a formal notation for their constraint expression. In DDD, they are modeled in the form of Class or Attribute Comments, or, in one occurrence, as name of a Dependency in which a domain object depends on the Association between two other domain objects to express that an `Overbooking Policy` (the dependent object) ensures that the sum of `Cargo` sizes does not exceed a `Voyage's` capacity by more than 10% [3].

Alternatively, all Constraint types could be modeled as UML `Constraints`. This would make their existence more explicit and allow to formally specify the Constraint's type. For example, to identify Informal and Semi-formal Constraints, an instance of `OpaqueExpression` with `language` set to "Natural language" could be assigned to `Constraint.specification`. Formal Constraints could analogously be expressed in the form of automatically evaluable expressions, e.g. by leveraging the Object Constraint Language (OCL) [11].

4 A UML Profile for Domain-Driven Microservice Architecture Design

This section presents an initial UML profile, which enables the expression of structural domain models as UML class diagrams by providing stereotypes and constraints for DDD patterns (cf. Sect. 2).

We decided to apply UML's profile mechanism [12] as *metamodeling technique* [15] for DDMM because (i) in [3], when introducing DDD, Evans expresses

structural domain models as UML class diagrams because he perceived them to be well understandable by domain experts; (ii) it provides an approach for defining graphical modeling languages by extending UML's mature metamodel [17] and use complementary specifications, e.g. OCL [11] for profile-specific constraint specification (cf. Subsect. 4.2); (iii) UML is a common modeling language, even for the design of Microservice architectures [1]; (iv) it enables the usage of existing UML toolchains suitable for domain experts or software engineers for DDMM.

A UML profile comprises *extensions* of UML metaclasses like `Class` or `Property` in the form of *stereotypes* [12]. Instances of extended metaclasses might then be semantically enriched with profile-specific stereotypes. A UML profile might also define formal constraints that enable automatic validation of profile-based models, e.g. to verify that stereotypes have been used as intended.

Figure 4 shows all stereotypes of our UML profile for DDMM. The relationship between a stereotype and the metaclasses it extends is depicted as an arrow with filled arrowhead pointing from stereotype to metaclass [12].

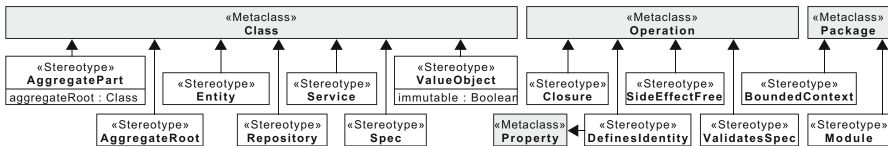

Fig. 4. Stereotypes of the DDMM UML profile as extensions of UML metaclasses

The profile provides stereotypes for all DDD patterns presented in Sect. 2, i.e. the ones listed in Table 1 as well as Bounded Contexts. It therefore extends the UML metaclasses `Class`, `Operation`, `Package` and `Property`. In the following, we discuss characteristics of the profile concerning differences between pattern definitions in DDD and the profile, constraints, mapping between profile-based models and MSA, and the profile's implementation.

4.1 Differences Between Pattern Definitions in Domain-Driven Design and UML Profile

While most of the profile's stereotypes correspond to their textual definition in [3], few of them were accompanied by additional stereotypes to formally enable their DDD-conform application. For example, the `DefinesIdentity` stereotype was added to specify which Attributes or Method of an Entity provide identity.

Moreover, the definition of an Aggregate involves the combined application of the `AggregateRoot` and `AggregatePart` stereotypes. The latter is necessary because in [3] the boundaries of an Aggregate are sketched informally by freehand drawings that enclose the Aggregate and its parts. When leveraging the profile for DDMM, the root of an Aggregate is annotated with `AggregateRoot`. Aggregate objects are then assigned to the root by means of the `AggregatePart` stereotype and specifying the Aggregate's root in the `aggregateRoot` property.

Another difference between its definition in DDD and in the profile exists for the Specification pattern. First, due to a name conflict with the UML

metamodel [12], the profile's stereotype for Specifications is abbreviated as `Spec`. Second, all predicate-like validation Methods of a Specification [3] need to exhibit the stereotype `ValidatesSpec`.

4.2 Profile Constraints

To ensure consistency between profile application and DDD, we added constraints to the profile, i.e. restrictions that need to be satisfied by a profile-based structural domain model to be considered valid. Table 3 describes them in natural language.

Table 3. Stereotype constraints of the profile following DDD pattern definitions in [3]

Stereotype	Constraints based on UML metamodel
`AggregatePart`	C1: Only Entities and Value Objects may be Aggregate parts
	C2: Assigned Aggregate root must have `AggregateRoot` stereotype
	C3: No incoming Associations from outside the Aggregate
	C4: Must be in same Bounded Context as Aggregate root
`AggregateRoot`	C5: Only Entities may be Aggregate roots
	C6: Aggregate must contain at least one part
`Entity`	C7: One Operation or at least one Property defines the identity
`Repository`	C8: Class has no other stereotypes
	C9: Class contains only Operations and at least one
	C10: Outgoing Associations must point to Entities or Value Objects
`Service`	C11: Class has no other stereotypes
	C12: Class contains only Operations and at least one
`Spec`	C13: Class has no other stereotypes
	C14: Class contains at least one validation Operation
	C15: At least one domain object is specified
	C16: Validation Operation has Parameter typed as specified object
`Closure`	C17: Must not be specification validation or identity Operation
	C18: Return Parameter type must conform input Parameter type
`DefinesIdentity`	C19: Must not be specification validation Operation
	C20: May only be applied within Entities
`SideEffectFree`	C21: Operation must have a return Parameter
`ValidatesSpec`	C22: Must have Boolean-typed return Parameter
	C23: May only be applied within Specifications
`BoundedContext`	C24: Must not have `Module` stereotype
	C25: Must not be nested, i.e. part of another Package

According to [12], all constraints have been formalized by expressing them in OCL [11]. However, due to lack of space, we only present the OCL code for constraints C4 and C25 in Listings 1 and 2. The OCL expressions for the remaining constraints are part of the profile's implementation (cf. Subsect. 4.4).

```
let partPackage = self.base_Class.package in
let root = self.base_Class
    .extension_AggregatePart.aggregateRoot in
partPackage <> null and
root <> null and
partPackage.extension_BoundedContext <> null and
partPackage = root.package
```

Listing 1. OCL code to ensure Aggregate parts being in same context as root (C4)

```
let nestingPkg = self.base_Package.nestingPackage in
let pkgStereotypes = nestingPkg.getAppliedStereotypes() in
nestingPkg = null or
pkgStereotypes->isEmpty() and
nestingPkg.nestingPackage = null
```

Listing 2. OCL constraint preventing nested Bounded Contexts (C25)

4.3 Mapping of Profile-Based Structural Domain Models to Microservice Architecture

In the following, we present initial ideas on how to map structural domain models applying the profile to conceptual elements of MSA for DDMM. We thereby focus on coherences between modeled Bounded Contexts and Microservices [2, 4,5] for the purpose of transforming profile-conform domain models into code. While [3] describes the implementation of the DDD patterns listed in Table 1, a possibly automatic derivation of Microservice code from structural domain models remains an open question.

An important aspect of mapping a Bounded Context and its encapsulated domain objects to a Microservice implementation is the determination of the service interfaces on the basis of context relationships. For example, in Fig. 3 the `Customer` Bounded Context shares a reduced form of its `Customer` Entity, which is modeled as a shared Value Object named `CustomerShared` that depends on the `Customer` Entity and is outside the context. As the shared object is referenced from the `Cargo` context, a Microservice for the `Customer` context needs to provide an interface that exposes `Customers` as instances of `CustomerShared`.

Moreover, in the domain model in Fig. 3, the signatures of interface operations as well as service provider and requester may be identified on the basis of usage dependencies between Bounded Contexts. For example, `Carrier Movement` from the `Cargo` context uses the `Location Service` from the `Location` context, which has access to a set of `Location` Entities, to retrieve shared model instances of these. Thus, a service for the `Location` context needs to provide an interface to a `Cargo` service that adapts the signature of the `Location Service`.

While the described mappings of Bounded Context relationships to Microservice interfaces are intuitive, several questions arise when taking the potential informality of structural domain models in DDD into account. First, besides Bounded Context relationships in the form of Associations between fragmented, probably shared domain objects, none of the surveyed domain models comprises constructs that specify technical characteristics of context interfaces for subsequent service implementation (cf. Sect. 3). Among these are the assignment of protocols and message formats to prospective interface operations, as well as an approach for stating the type of action performed by an operation, e.g. read or update. Additionally, when modeling service calls as ≪use≫-Dependencies in which the supplier has more than one Operation that returns the same shared model type, it cannot be unambiguously determined, which of the Operations the client invokes for shared model instance retrieval, e.g. for the `Cargo` object in Fig. 3 `find by name` or `find by Customer ID` from `Customer Repository`.

Another open question concerns the handling of Associations between context-internal domain objects and shared models. For example, in Fig. 3 `Carrier Movement` is associated with `LocationShared`. However, as `Carrier Movement` is an Entity and probably gets persisted when a `Cargo` Microservice proceeds [3], an approach is needed for keeping `LocationShared` (and hence `Location`) and `Carrier Movement` instances persistently associated. This includes retrieval of shared model instances when `Carrier Movement` is reinstantiated, with considering that the `Location` might since have been deleted by a `Location` service.

Furthermore, DDD lacks a construct for specifying how a domain object is transformed into a shared model representation, e.g. in Fig. 3 `CustomerShared` does not comprise the `Customer's` name. As an initial approach, a code generator for the UML profile could yield stubs for operations that transform domain object instances into their shared representations. However, then the consistency between model and code needs to be ensured for future model refinements.

A last aspect stems from DDD being a modeling technique focused on expressing core domain parts rather than achieving model completeness. In case two shared models of the same domain object are modeled, it cannot be unambiguously determined which shared representation a derived service interface returns when the retrieval operations of the underlying provider objects, e.g. Repositories or Services, do not specify a return type.

4.4 Implementation

We have implemented an initial version of the UML profile that comprises all stereotypes and constraints presented in Fig. 4 and Subsect. 4.2 on the basis of Eclipse and the Papyrus modeling environment[1]. The current version can be found on GitHub[2]. Figure 5 shows the `Cargo` context from Fig. 3 and its relationship to `LocationShared` modeled with Papyrus and applying the profile.

[1] https://www.eclipse.org/papyrus.
[2] https://github.com/SeelabFhdo/ddmm-uml-profile.

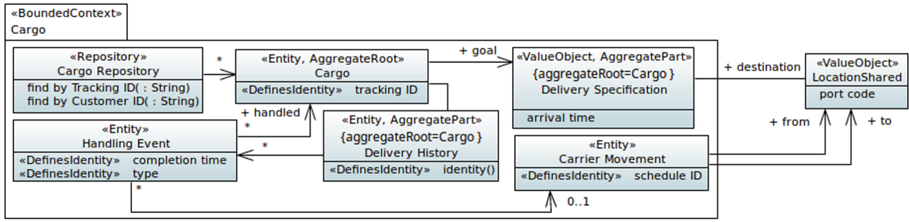

Fig. 5. Excerpt of the `Cargo` context modeled with Eclipse Papyrus and the profile

5 Related Work

We discuss work related to employing UML class diagrams for DDD as well as the design of service-based software systems leveraging UML profiles and DDD.

In [7], a DDD-based approach is presented that leverages *meta-attributes* (MAs) as annotation mechanism for UML class diagrams representing structural domain models with the goal to enable capturing domain-specific requirements. MAs reflect domain-specific abstractions that may be directly mapped to code by using built-in extension mechanisms of the programming language, e.g. Java annotations. They are modeled as classes with attributes, whose values correlate to property values of code annotations, and are associated with domain model elements. This approach differs from the application of our UML profile. First, MAs instead of stereotypes are used to annotate domain models. While this is UML-conform, extending domain models with additional MA classes enlarges their structure and may complicate understanding. This effect is mitigated when using a UML profile as its application might be flexibly hidden from a UML diagram [12]. Second, MAs do not enable constrained expression of DDD patterns (cf. Table 1). Instead, MAs map to UML metaclasses like `Classifier`. Third, to foster semantic understanding of domain experts, MAs partially capture information being already part of the model, e.g. the `name` Property of a `DAttr` MA specifies the name of a modeled domain object Attribute. Our approach assumes that domain experts are already able to read basic UML class diagrams. Fourth, existing UML tools may not semantically differentiate between MAs and domain objects as both are UML classes without specific stereotypes. This hampers automatic validation of annotated structural domain models.

SoaML [10] is a UML profile and metamodel from the OMG for model-driven engineering of service-based systems. It defines modeling elements to describe, e.g., services, interfaces and data exchange, and addresses SOA, which characteristically differs from MSA [14]. However, SoaML provides an extensive set of constructs for modeling service interfaces and interactions our profile might draw on (cf. Subsect. 4.3). Thereby, it would be crucial to balance the technical needs of MSA architects and developers with the profile's applicability for domain experts, which is central to DDD but not one of SoaML's primary goals.

In [6] the Romulus approach for the development of service-based software systems is presented. It integrates a metaframework that enables the enrichment

of Java-based domain models with annotations to provide services. Thereby, the first step in Romulus is to identify domain objects that reside in different Bounded Contexts. Like with the presented UML profile, domain models are expressed as class diagrams and, conceptually, a Bounded Context may be mapped to a service. However, no specific UML notation on how to express DDD elements in domain models is presented. Instead, domain models are implemented as semantically annotated plain Java objects. Thereby, annotations do not express DDD concepts but complement a domain model with technical aspects like view representation and validation. MSA is not explicitly covered.

6 Conclusion and Future Work

In this paper we introduced an initial UML profile that aims at enabling the modeling of Microservice systems by leveraging Domain-driven Design [3]. Therefore, we first presented DDD and its patterns, with Bounded Context being central for modeling Microservice candidates (cf. Sect. 2). In Sect. 3, DDD was characterized by means of a literature survey, which comprised each of the 92 structural domain models in [3]. It identified syntaxes, semantics and occurrences of UML class diagram constructs used to capture domain models. Together with the DDD patterns, these UML elements define an initial set of modeling elements, which need to be considered in UML-based DDMM, e.g. for model validation or transformation purposes. In Sect. 4 we presented a UML profile for DDMM, which integrates constrained stereotypes for all mentioned DDD patterns. We also discussed initial thoughts on how to map profile-based domain models to Microservices with considering the findings of our survey (cf. Subsect. 4.3).

In future works we plan to implement a code generator for producing MSA code from profile-based domain models. We therefore focus on transforming Bounded Contexts into services with regard to deriving service interfaces from associations between domain objects of different contexts. With the code generator, we plan to evaluate the profile's applicability for both software engineers and domain experts, as well as the generators efficiency.

References

1. Alshuqayran, N., Ali, N., Evans, R.: A systematic mapping study in microservice architecture. In: Proceedings of the 9th International Conference on Service-Oriented Computing and Applications (SOCA), pp. 44–51. IEEE (2016)
2. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables DevOps: migration to a cloud-native architecture. IEEE Softw. **33**(3), 42–52 (2016)
3. Evans, E.: Domain-Driven Design. Addison-Wesley, Boston (2004)
4. Gysel, M., Kölbener, L., Giersche, W., Zimmermann, O.: Service cutter: a systematic approach to service decomposition. In: Aiello, M., Johnsen, E.B., Dustdar, S., Georgievski, I. (eds.) ESOCC 2016. LNCS, vol. 9846, pp. 185–200. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44482-6_12

5. Hassan, S., Ali, N., Bahsoon, R.: Microservice ambients: an architectural metamodelling approach for microservice granularity. In: Proceedings of the International Conference on Software Architecture (ICSA), pp. 1–10. IEEE (2017)
6. Iglesias, C.A., Fernández-Villamor, J.I., del Pozo, D., Garulli, L., García, B.: Combining domain-driven design and mashups for service development. In: Iglesias, C.A., Fernández-Villamor, J.I., del Pozo, D., Garulli, L., García, B. (eds.) Service Engineering, pp. 171–200. Springer, Vienna (2011). https://doi.org/10.1007/978-3-7091-0415-6_7
7. Le, D.M., Dang, D.H., Nguyen, V.H.: Domain-driven design using meta-attributes: a DSL-based approach. In: 8th International Conference on Knowledge and Systems Engineering (KSE), pp. 67–72. IEEE (2016)
8. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci. **152**, 125–142 (2006)
9. Newman, S.: Building Microservices. O'Reilly Media, Sebastopol (2015)
10. Object Management Group: Service oriented architecture modeling language (SoaML) specification version 1.0.1 (formal/2012-05-10) (2012)
11. Object Management Group: Object constraint language (OCL) version 2.4 (formal/2014-02-03) (2014)
12. Object Management Group: OMG unified modeling language (OMG UML) version 2.5 (formal/2015-03-01) (2015)
13. Pahl, C., Jamshidi, P.: Microservices: a systematic mapping study. In: Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER), pp. 137–146 (2016)
14. Rademacher, F., Sachweh, S., Zündorf, A.: Differences between model-driven development of service-oriented and microservice architecture. In: International Conference on Software Architecture Workshops (ICSAW), pp. 38–45 (2017)
15. Da Silva, A.R.: Model-driven engineering: a survey supported by the unified conceptual model. Comput. Lang. Syst. Struct. **43**, 139–155 (2015)
16. Seidewitz, E.: What models mean. IEEE Softw. **20**(5), 26–32 (2003)
17. Selic, B.: A systematic approach to domain-specific language design using UML. In: Proceedings of the 10th International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), pp. 2–9. IEEE (2007)