Formalizing Domain-driven Microservice Design with UML's Profile Mechanism

Florian Rademacher

University of Applied Sciences and Arts Dortmund
Institute for Digital Transformation of Application and Living Domains
44227 Dortmund, Germany
florian.rademacher@fh-dortmund.de

Abstract

In the context of Microservice Architecture (MSA), Domain-driven Design (DDD) denotes a model-driven approach for domain decomposition and service identification. However, DDD-based domain models are typically expressed as informal UML class diagrams, which hampers model operations like code generation. As a first step to overcome this limitation, we present a UML profile that formalizes DDD-based domain models and enables domain-driven Microservice design.

1 Introduction

With the recent advent of Microservice Architecture (MSA) [6], Domain-driven Design (DDD) [2] gains attention as a model-driven approach for identifying services by domain decomposition [3]. Therefore, DDD provides modeling patterns such as Bounded Contexts that enable clustering of coherent domain concepts, expressing data exchange between clusters and mapping clusters to Microservices. Additionally, DDD comprises means to enrich domain concepts with further semantics, e.g. for stating that a concept instance is distinguished from others by a domain-specific identity or that a concept enables access to persistent data.

However, DDD commonly leverages informal UML class diagrams to depict domain models with classes representing domain concepts [2]. This hampers the application of model operations like code generation. To reduce models' informality and increase their value for designing MSA-based software systems, we present our UML profile introduced in [7] (cf. Section 2) and elaborate on its use for MSA implementation (cf. Section 3).

2 A UML Profile for Domain-driven Microservice Design

In this section we first introduce patterns applied by DDD for enriching domain concepts with additional semantics [2] and identify their UML 2.5 metamodel [5] equivalents (cf. Subsection 2.1). Starting from these patterns and their UML counterparts, we conceptually define the UML profile for domain-driven Microservice design (cf. Subsection 2.2) and present its implementation (cf. Subsection 2.3).

2.1 Patterns of Domain-driven Design for Domain Modeling

Table 1 shows patterns of DDD to semantically enrich domain models and their UML 2.5 metamodel [5] equivalents according to [2]. The patterns are applied to domain models in that domain concepts, i.e. UML classes, concept parts, i.e. class methods or attributes, or clusters, i.e. UML packages, are "annotated", e.g. with UML stereotypes or comments identifying the patterns being used.

Table 1: DDD patterns and their UML 2.5 metamodel [5] equivalents from [7]. 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.
Bounded Context	Annotated package	Encapsulation mechanism for defining scopes for enclosed domain concepts, i.e. boundaries for concept validity and applicability.
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 concept dependencies.
Entity	Annotated class	An instance of the domain concept 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 concepts.
Repository	Annotated class with outgoing associations to other classes	Models access to persistent domain concept 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 concept'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 concept 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.

2.2 Definition of the UML Profile

In this section we present the modeling elements and constraints that constitute our UML profile for domain-driven Microservice design. We decided to apply UML's profile mechanism as *metamodeling technique* [5, 8] because (i) Evans, who defined DDD as a model-driven approach, expresses domain models as UML class diagrams and perceived them to be well understandable by domain experts [2]; (ii) it provides an approach for defining graphical modeling languages by extending UML's mature metamodel [9] and use complementary specifications, e.g. the Object Constraint Language (OCL) for constraint definition [4]; (iii) UML is a common modeling language, even for the design of MSA-based software systems [1]; (iv) it enables ad-hoc usage of existing UML toolchains.

Figure 1 shows the profile's modeling elements as extensions of the UML metaclasses `Class`, `Operation`, `Property` and `Package`. UML metaclass extensions are depicted as arrows with filled arrowheads pointing from extensions to metaclasses and applied to existing UML diagrams in the form of stereotypes [5].

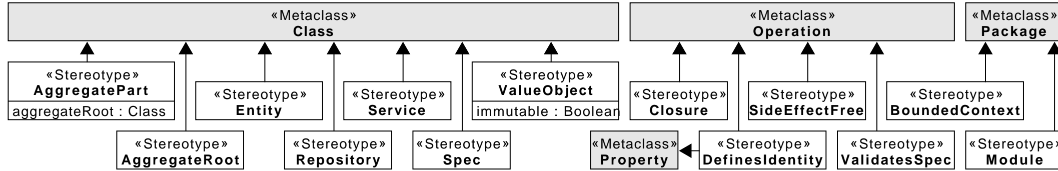

Figure 1: The profile's modeling elements defined as stereotypes, i.e. extensions of UML metaclasses [7]

The profile comprises a stereotype for each DDD pattern listed in Table 1. However, applying the Entity or Aggregate patterns involves additional stereotypes. First, the usage of the `DefinesIdentity` stereotype is mandatory for an Entity domain concept to identify which method or attributes are responsible for determining an instance's domain-specific identity. Second, for the definition of Aggregates in a domain model, the two stereotypes `AggregateRoot` and `AggregatePart` need to be applied in combination. Thereby, `AggregateRoot` specifies an Aggregate's root Entity, while `AggregatePart` is used to indicate that a domain concept is part of an Aggregate by referencing its root Entity class.

Furthermore, the profile comprises constraints to ensure well-formedness of profile-based domain models [4]. Table 2 lists the profile's constraints for each stereotype in textual form.

2.3 Implementation of the UML Profile

The profile has been implemented leveraging Eclipse Papyrus[1] with its stereotypes being based on the Eclipse realization of the UML metamodel and constraints being implemented in OCL [4]. The current version can be found on GitHub[2].

Figure 2 shows an example of a profile-based domain model from [2] created with Eclipse Papyrus.

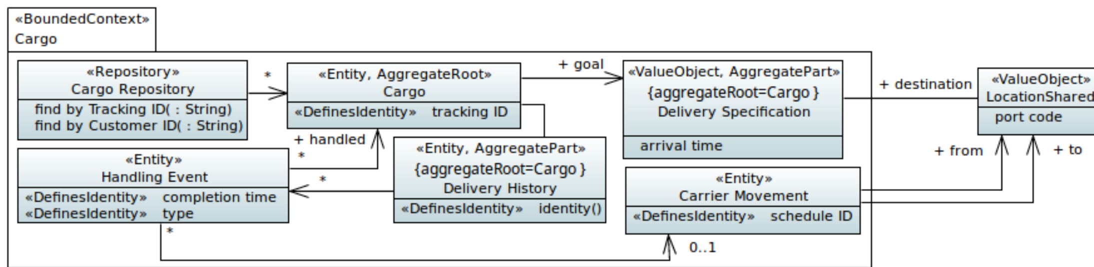

Figure 2: Example DDD-based domain model from [2] applying the profile with Eclipse Papyrus [7]

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

Table 2: Stereotype constraints of the profile [7]

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 concept is specified C16: Validation operation has parameter typed as specified concept
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

3 Mapping Profile-based Models to Microservices

In this section we present initial ideas on how to map profile-based domain models to MSA implementations [7]. Therefore, according to [3], we focus on deriving a Microservice for each Bounded Context in a domain model.

The following paragraphs describe certain aspects of transforming profile-based domain models into MSA implementations.

Service Interfaces Service interfaces may be deduced on the basis of relationships between Bounded Contexts. These relationships may be modeled *directly* as associations between domain concepts of different contexts or *indirectly* in the form of *shared Value Objects* [3]. Shared Value Objects explicitly define the structure of information exposed by a Bounded Context and might hence be used to tailor domain concepts for information exchange with other contexts. For example, in Figure 2 two concepts of the `Cargo` Bounded Context reference the shared Value Object `LocationShared`, which has been defined outside the context.

However, the deduction of service interfaces is independent of the kind of concept relationship, i.e. direct or indirect. Basically, a Microservice representing a Bounded Context needs to provide access to instances of concepts referenced from other contexts. The kind of relationship

then determines the return type of the interface's access methods, i.e. whether instances of original domain concepts or corresponding shared Value Objects are returned.

However, several open questions remain when determining service interfaces on the basis of relationships between Bounded Contexts. First, if associations between concepts of different contexts are non-navigable, e.g. between `Delivery Specification` and `LocationShared` in Figure 2, service provider and requester might not be unambiguously determinable. Second, it remains unclear which operations a service interface needs to provide. Thus, it cannot be identified whether a context and the resulting service allow read access for domain concept instances only or if they also enable instance creation or deletion. Third, mechanisms are needed to keep domain concept instances persistently associated [7]. Fourth, DDD lacks modeling constructs for specifying how a domain concept is transformed into a shared object representation.

Lack of Technical Characteristics Besides relationships between Bounded Contexts modeled as UML associations and fragmented, probably shared Value Objects, DDD does not comprise constructs that specify technical characteristics of prospective service interfaces. Among these are the assignment of protocols and message formats to interface operations, as well as an approach for stating the type of action performed by an operation, e.g. create or read.

Additionally, when two shared objects are modeled for one concept, it cannot be unambiguously determined which shared representation a derived service interface shall return [7].

Model and Code Consistency When deriving Microservices from domain models, means are needed to ensure consistency between model and implementation, e.g. when an original domain concept is extended by an attribute that also needs to be considered in shared Value Objects. This impacts existing code of interfaces, domain concepts, shared objects and their producers [7].

References

[1] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *Proc. of the 9th Int. Conf. on Service-Oriented Computing and Applications (SOCA)*, pages 44–51. IEEE, 2016.

[2] Eric Evans. *Domain-Driven Design*. Addison-Wesley, 2004.

[3] Sam Newman. *Building Microservices*. O'Reilly Media, 2015.

[4] Object Management Group. Object constraint language (OCL). Version 2.4 (formal/2014-02-03), 2014.

[5] Object Management Group. OMG unified modeling language (OMG UML). Version 2.5 (formal/2015-03-01), 2015.

[6] Claus Pahl and Pooyan Jamshidi. Microservices: A systematic mapping study. In *Proc. of the 6th Int. Conf. on Cloud Computing and Services Science (CLOSER)*, pages 137–146, 2016.

[7] Florian Rademacher, Sabine Sachweh, and Albert Zündorf. Towards a uml profile for domain-driven design of microservice architectures. In *Proceedings of the Second Workshop on Microservices: Science and Engineering (MSE)*. IEEE, 2017. to be published, a preprint is available at http://fmse.di.unimi.it/faacs2017/papers/paperMSE1.pdf.

[8] Alberto Rodrigues Da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems and Structures*, 43:139–155, 2015.

[9] Bran Selic. A systematic approach to domain-specific language design using uml. In *Proc. of the 10th Int. Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 2–9. IEEE, 2007.