Elaboration of a Domain Model for Migrating the Monolithic Software Architecture of a Data Management Server into a Microservice Architecture

Christian Zaschke[a]
Fraunhofer IOSB, Karlsruhe, Germany

Keywords: CSD, Coalition Shared Data, DDD, Domain-Driven Design, Domain Model, Microservice.

Abstract: This paper presents the necessary steps for creating a useful domain model for the Coalition Shared Data (CSD) domain. After describing the basic procedure of creating a domain model following the principles of domain-driven design (DDD) some concrete results achieved and experiences gained throughout this process are depicted. The domain model is the backbone for further development activities to migrate the monolithic software architecture of an existing data management server into a loosely coupled one according to the principles of microservice architectures. Major challenges with the creation of such a domain model and the benefits of loosely coupled software components in the CSD environment are outlined.

1 INTRODUCTION

Collection, analysis and processing of huge amounts of data nowadays are crucial abilities to meet the demands of a digital and globalized world. These capabilities help to gather necessary information about people and organizations, their specific actions and intentions, with the aim to get superior knowledge. As a basis for making data available to the right people at the right time with the right granularity (Zaschke et al., 2016) a process for storing, disseminating and maintaining data has to be defined, implemented and established. To involve a critical mass of stakeholders in this process, participation has to be made possible in a beneficial, cost-efficient and sustainable way. One way to achieve this is to establish common standards.

One of these interoperability standards is the NATO Standardization Agreement (STANAG) 4559 ((NSO), 2018) which defines the interface of a distributed data storage for enabling data sharing between different NATO nations and their partners.

The STANAG 4559 specifies standardized interfaces to allow ingesting and retrieving relevant Joint Intelligence, Surveillance and Reconnaissance (JISR) products into a so called Coalition Shared Data (CSD) Server. Whether data is relevant (for oneself or for other partners) or not has to be determined by the people who collect and process them. The designated JISR products follow standardized data formats as well to make sure that CSD client systems requesting those products know how to interpret and process them. CSD client systems using the STANAG 4559 interfaces are able to ingest new products and to place specific requests on the server to get the appropriate products in return.

As the STANAG 4559 doesn't specify architectural aspects of an implementation of the interface, a typical STANAG 4559 implementation is built up monolithically and, thus, closely coupled.

The paper addresses the necessary steps, challenges and benefits of a migration from a monolithic software structure to a loosely coupled microservice architecture following the principles of domain-driven design (DDD) as a precondition for a behavior-driven development (BDD) approach in a later phase.

2 FROM DOMAIN KNOWLEDGE TO A DOMAIN MODEL

According to the principles of DDD, a domain model for a specific area of application has to be developed to solve a problem in the real world. According to (Evans, 2015) a domain is defined as "a sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of

[a] https://orcid.org/0000-0002-6351-491X

Zaschke, C.
Elaboration of a Domain Model for Migrating the Monolithic Software Architecture of a Data Management Server into a Microservice Architecture.
DOI: 10.5220/0008319402120218
In *Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2019)*, pages 212-218
ISBN: 978-989-758-382-7

the software."

The domain model is the basis for a software specification which could be created following a BDD approach.

The features defined in the software specification represent the real world requirements. By implementing those features the corresponding problem of the real world is being addressed and solved appropriately.

2.1 Ubiquitous Language

For the development of a domain model both domain experts and technical experts have to be involved. The domain experts have a comprehensive knowledge and their own business-specific language describing business-specific terms and concepts. Technical experts, on the other side, typically use a very technical and highly specific language. They have great experience in the development of complex systems and are adept at using state-of-the-art methods and technologies.

As a first step towards a domain model, it is crucial to define a common and compulsory vocabulary - a ubiquitous language. To achieve that, all involved stakeholders enter into intense interactions with each other. In a continuous, iterative and collaborative process technical and domain experts determine a collection of common terms and specify a set of features. This creative process is called knowledge crunching.

In knowledge crunching events all stakeholders are brainstorming all their terms, concepts and issues of the specific domain. The domain experts have to abstract their knowledge, whereas the technical experts have to understand and formalize the domain knowledge. Experiences of the stakeholders in similar domains or with earlier projects could be advantageous and complement the domain knowledge. Subsequently, the huge amount of input is being harmonized, distilled, restructured and streamlined.

The result of this process is a collection of terms and a feature list. All terms need to be clearly defined. The relevant features have to be described accurately. As a result, the ubiquitous language for the domain is defined and ready to be used by all stakeholders in the following development and operating phases.

When talking about a domain-specific term or concept every involved stakeholder uses the same word for it. Likewise, each involved stakeholder should have a clear comprehension about every defined term. As a consequence, in conversations and written texts every stakeholder has the same understanding of a specific term or concept. This helps avoiding ambiguities and misinterpretations and leads to a clear and efficient communication and documentation.

In case the team realizes that the definition or name of a term or a concept is not useful or not correct anymore, it needs to be revised and again collectively defined and agreed upon. This procedure helps keeping the ubiquitous language clean and up-to-date and gives the flexibility to react on changes throughout the whole development. The implications of a change in the ubiquitous language and thereby in the domain model will be examined in the next section.

2.2 Domain Model

The domain model is the pivotal point of a software project following the principles of DDD. It describes the interrelationship of central terms and concepts of the determined common language and can therefore be seen as the abstraction of the underlying domain. According to (Evans, 2003) these terms and interrelationships provide the semantics of a language that is tailored to the domain while being precise enough for technical development. The software specification, forming the basis of the implementation, is build up on the defined domain model which leads to a strong binding between the domain model and the derived implementation - the "crucial cord that weaves the model into development activity and binds it with the code" (Evans, 2003).

The elaboration of the domain model is an iterative process in which the experiences and insights gleaned throughout the project are reflected. Thus, the domain model may be subject of change all the time. These changes are typically based on contributions from domain experts and developers. If the software project needs, at a given moment, some kind of conceptual or linguistic change, e.g., additional terms, new wording of existing terms or new relations between concepts, this change is applied at domain model level first. This change has implications to the software specification (e.g., the class diagram) and in turn to the code which again emphasizes the strong binding between model and implementation.

2.2.1 Conceptual Modeling

DDD is not a technology or a methodology - it is a way of thinking and a set of priorities for accelerating software projects that have to deal with complicated domains (Landre et al., 2007).

Effective modeling as defined in (Evans, 2003) is obtained by performing the following five activities.

- Binding the model and the implementation;
- Cultivating a language based on the model;

- Developing a knowledge-rich model;
- Distilling the model;
- Brainstorming and experimenting.

As the implementation is strictly based on the domain model they are ”bound” with each other. The domain model is used by the developers throughout the whole development process. New insights and experiences gained during the development could lead to changes in the domain model. Whereas a change in the model implies changes in the implementation.

The model contains terms which are part of the ubiquitous language described in section 2.1. This language is common to all participants involved in the software project including domain experts and developers. Through the cultivation of a common language base concepts become linguistically describable. In this way, concepts could be communicated unambiguously and translations between different notions and concepts depending on the expert domains could be prevented. The domain model is thereby used as the backbone of the ubiquitous language.

The domain model is more than just a representation of data structures, it must contain the whole domain knowledge. It contains data structures and objects but also ideas, behaviors and enforced rules. To a certain extent this so called knowledge-rich model also contains system behavior and various knowledge relevant for the domain and the system.

As the domain model contains all, but also only, the relevant knowledge, relevant knowledge has to be added continuously and knowledge that is not relevant any more has to be removed from the model. Through this process of adding and removing knowledge from the model the desired domain model distills from the total amount of domain knowledge.

The domain model is subject to changes throughout the development process and could be improved through brainstorming sessions and experimentation meetings between domain experts and developers. The output of the massive experimentations and discussions taking place is the distilled and knowledge-rich model - the domain model.

2.2.2 Technical Modeling

In this section the development of the domain model is being described from a more technical perspective.

The desired output of the domain model development process is a formalized mapping of the technical links inside a domain.

A domain typically consists of several separable sub-domains in which it could be broken down. In a domain model sub-domains are represented through packages.

A self-contained sub-domain is called bounded context. Bounded contexts serve as candidates for the derivation of microservices in a later step (Millett, 2015). In the beginning, a concrete partitioning in bounded contexts is not known. The partitioning has to be developed in various iterations using an exploration and experimentation approach.
A context map defines the relations between bounded contexts by the use of shared entities, i.e. shared models and shared objects. The context map is an overview page and should be part of every domain model. It provides a brief overview over all relations between all bounded contexts. From a developer team's perspective a context map reflects the kind of relationship their bounded context has to another bounded context and indicates areas where inter-team communication is imperative (Vernon, 2013).

Domain views subdivide the model. They depict static and dynamic aspects of the domain and visualize relations between domain objects. Static aspects are depicted by relation views. Whereas process views depict the dynamic aspects of the domain.

Examples for a context map, a relation view and a process view were given in section 3.2.

3 ELABORATION OF THE CSD SERVER DOMAIN MODEL

In this section the creation process of the CSD Server domain model and the lived experiences will be explained.

3.1 Ubiquitous Language for the CSD Server Domain

The development of a domain model for the CSD domain started - similar to the approach described in section 2.1 - with several meetings of CSD experts and experienced technical personal. In the first meeting the general concept was presented and explained. Because of the abstract level of detail only a few questions were asked and answered.

The second meeting started with some questions from the developers about aspects that were not mentioned yet or not completely understood. After that, the developers presented their typical approach of developing a domain model for a new domain. At this point, both sides - the domain experts and the technical experts - had a rough idea about the other experts knowledge and procedures.

The next two meetings served as the communication platform for developing the ubiquitous language

explained in section 2.1. Every term mentioned in these meetings which was related to the domain was written down by the developers and explained by the domain experts.

During this process, even the domain experts learned something about their own domain by clearly defining and differentiating the considered terms. Fine-grained differences between the interpretations of some terms among the domain experts were revealed and discussed. The various questions of the developers trying to figure out the meaning of any term and his relation to other terms helped them to get used to the domain step-by-step. It also helped the domain experts to extract and transfer their domain knowledge to the developers.

The knowledge got formalized by establishing a common vocabulary in which all domain-specific terms were listed and clearly defined. This list of terms and their definitions was produced by the developers which also presented them to the domain experts. After the presentation all definitions were either accepted or discussed again. After two more iterations all definitions were accepted and agreed upon from both sides. The common vocabulary - reflecting the current level of knowledge - was finally established and committed on by all persons involved. From this moment on, every person used only these terms defined in the common vocabulary and only according to the common understanding.

3.2 Domain Model for the CSD Server Domain

After defining all terms and establishing a common understanding of the core domain, the development of the domain model for the CSD Server domain started. Based on their actual domain knowledge, the technical experts created a first draft version of the domain model. This first draft was explained to and discussed with the domain experts. The developed domain model was not correct and complete in all points and in general did not exactly represent all aspects of the domain. The further disclosure and integration of several new and previously not completely known details by the CSD experts led to new findings for some terms and their relations. With the feedback of the domain experts and the complemented and adjusted vocabulary and domain knowledge the model had been revised by the technical experts. After several development iterations and feedback meetings in which the domain model was gradually updated and going through several intermediate stages, the first complete and correct version of the domain model had been created.

3.2.1 Context Map

An extract of the CSD domain model showing the context map for the so called *Catalogue Entry Storage* is illustrated in Figure 2 (please see Appendix).

The context map of the *Catalogue Entry Storage* shows its three sub-domains represented as the packages *FileManagement*, *CatalogueManagement* and *DataModel*. The package *FileManagement* consists of the bounded context *File*. The *CatalogueManagement* package consists of the bounded contexts *Catalogue* and *CatalogueQuery*. The package *DataModel* contains the bounded context *DataModel*.

The context map defines the relations between the mentioned bounded contexts by displaying their shared entities. For example, the bounded context *File* uses the shared object *FileFormat* and the bounded context *CatalogueQuery* uses the shared object *CatalogueEntry*.

3.2.2 Relation View

The relations between the bounded contexts and the shared objects are described in relation views in further detail. Figure 1 depicts the relations between the shared objects *File* and *FileFormat* in the bounded context *DataModel*.

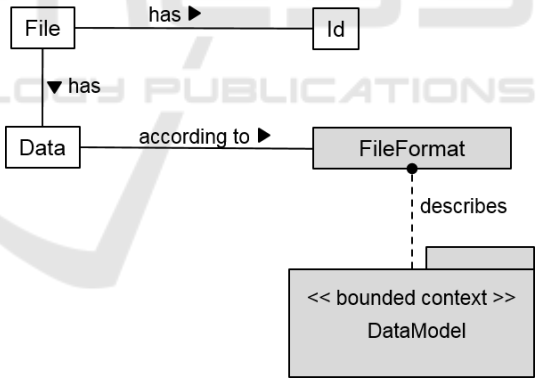

Figure 1: The relation view describing the *File*.

The relation view shows that the *File* has an *Id* and *Data*. The *Data* is according to a *FileFormat*. The *DataModel* describes the *FileFormat*. In this way, relation views help to understand the relationships between different domain model entities in connection with the bounded contexts they belong to.

3.2.3 Process View

Process views are used to model processes in the domain. Processes could for example be part of business workflows within the domain or technical processes taking place between systems. Figure 3 (please see

Appendix) visualizes the interaction between a *Client*, the *CatalogueEntry*, the *Catalogue* and the *FileManagement*.

The *Client* could ingest one or more files by providing a *FileId* and the *File* itself. The *File* is provided using an upload mechanism. The *FileManagement* stores the *File* and makes it available for distribution. This step is optional and could also be executed several times if there is more than one file to be provided.

The next mandatory step is the creation of the *CatalogEntry*. This step is again initiated by the *Client* and validated by the *CatalogueEntry*.

After the creation of the *CatalogueEntry* the *Client* initiates the storage of the *CatalogueEntry*. The *Catalogue* distributes the *CatalogueEntry* and afterwards notifies all subscribers about the new *CatalogueEntry*.

4 CHALLENGES AND BENEFITS OF USING A MICROSERVICE ARCHITECTURE IN THE CSD DOMAIN

Restructuring an existing system is in many cases a system design decision as a result of a cost-benefit analysis in favor of the potential benefits. But that does not mean that there are only few problems to be solved. Restructuring the architecture of an existing system in an efficient way does not only need a clear vision of the target structure but also a good knowledge of the existing architecture. A new structure has to be designed by technical experts who are experienced in the development of the desired structure. But the technical experts also have to understand the domain the system is being used at.

In case of the data management server considered in this paper, the underlying standard that is being implemented by that server is very complex and ambiguous. As a consequence, it is not sufficient only to understand this standard but it is also necessary to get a common understanding of how some aspect of the ambiguous parts of the standard should be interpreted in this context. Even the domain experts have to argue about details and finally need to agree - in consultation with the technical experts - to a common interpretation of the relevant aspects for the development of the new structure.

After designing the desired target architecture a migration path has to be elaborated. Because the restructuring process needs a certain amount of time in which the software still has to be used, maintained and developed further, in some cases a gradual alteration approach with several intermediate stages is being applied. All of these intermediate stages have to be planned and need to be aligned with the general development plan of the software.

As described in section 3.2 the elaboration of a domain model is an iterative and very communicative process. It is a good way for technical experts involved to get to know the domain as a whole and also in detail. Moreover, it is a chance for the domain experts to learn how to formalize their domain knowledge and to identify and eliminate ambiguities in their own domain knowledge. The established domain model helps all involved persons to get used to a potentially complex domain as easy as possible and with the usage of an unambiguous common vocabulary. Thus, even the design and development process itself generates a valuable output, e.g., the list of terms along with their definitions and the domain model, for further design and development tasks.

From a more technical view a loosely coupled software architecture following the principles of microservice architectures helps to efficiently maintain and quickly adapt and roll out new versions of specific parts of the software. This is a major advantage for a system that is constantly evolving but typically only step-by-step and only in specific, isolated parts of the software (the bounded contexts). While implementing, testing and deploying the different parts of a system separately the overall effort for software engineering is significantly reduced.

An important assumption to do so is that the interfaces between the bounded contexts are fixed and don't change over time.

A system consisting of such loosely coupled services that interact with each other over fixed interfaces provides a good basis for an implementation following a microservice approach.

5 CONCLUSION

The paper introduced the best practices for the development of the domain model for a specific domain.

It started with a brief motivation why interoperability is the key issue in the CSD domain and introduces the goal of migrating an existing data management server using a domain-driven design approach.

Then the typical best practice approach of working out an ubiquitous language as a basis for the creation of a domain model was presented. Afterwards, the development of a domain model was described on a conceptual level on one hand and on a more technical level on the other hand.

To provide some practical experiences the concrete approach of elaborating a domain model for the CSD domain was described. The concrete approach also followed the typical steps of first creating a ubiquitous language for the CSD domain followed by the development of several different diagrams that also serve as examples for the previously defined design artifacts.

As a last major topic the challenges and benefits of the usage of microservices in the CSD domain were pointed out.

REFERENCES

Evans, E. (2003). *Domain-Driven Design – Tackling Complexity in the Heart of Software*. Addison-Wesley.

Evans, E. (2015). *Domain-Driven Design Reference – Definitions and Pattern Summaries*. Dog Ear Publishing.

Landre, E., Wesenberg, H., and Olmheim, J. (2007). Agile enterprise software development using domain-driven design and test first.

Millett, Scott with Tune, N. (2015). *Patterns, Principles, and Practices of Domain-Driven Design*. John Wiley & Sons, Inc.

(NSO), N. S. O. (2018). STANAG 4559 Edition 4.

Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley.

Zaschke, C., Essendorfer, B., and Kerth, C. (2016). Interoperability of heterogeneous distributed systems. In *Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security, Defense, and Law Enforcement Applications XV*, volume 9825, page 98250Q. International Society for Optics and Photonics.

APPENDIX

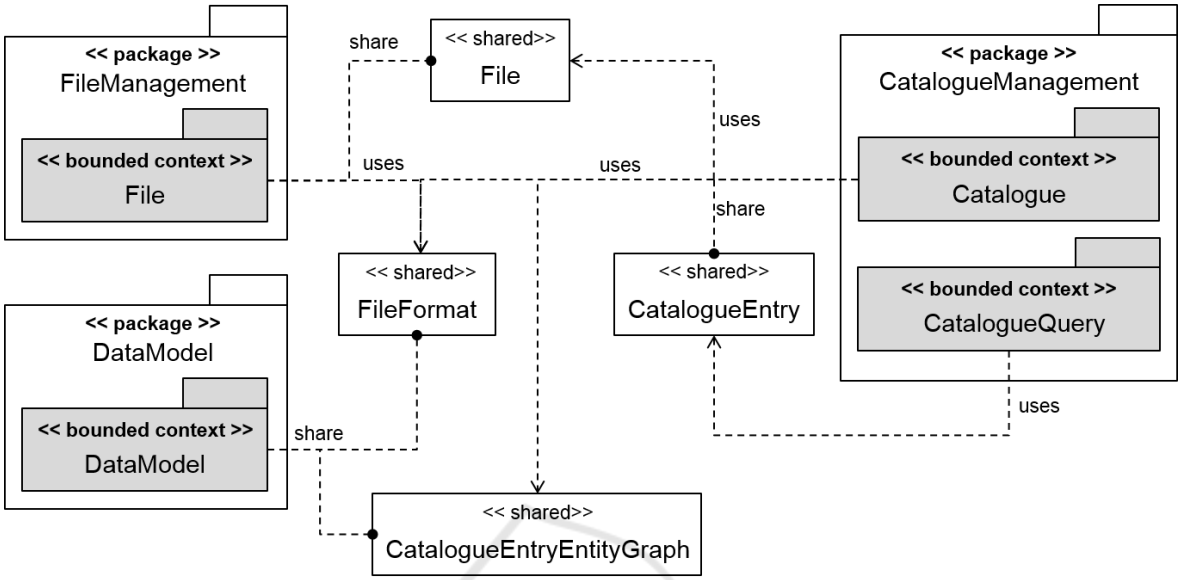

Figure 2: An extract of the CSD domain model showing the context map for the *Catalogue Entry Storage*.

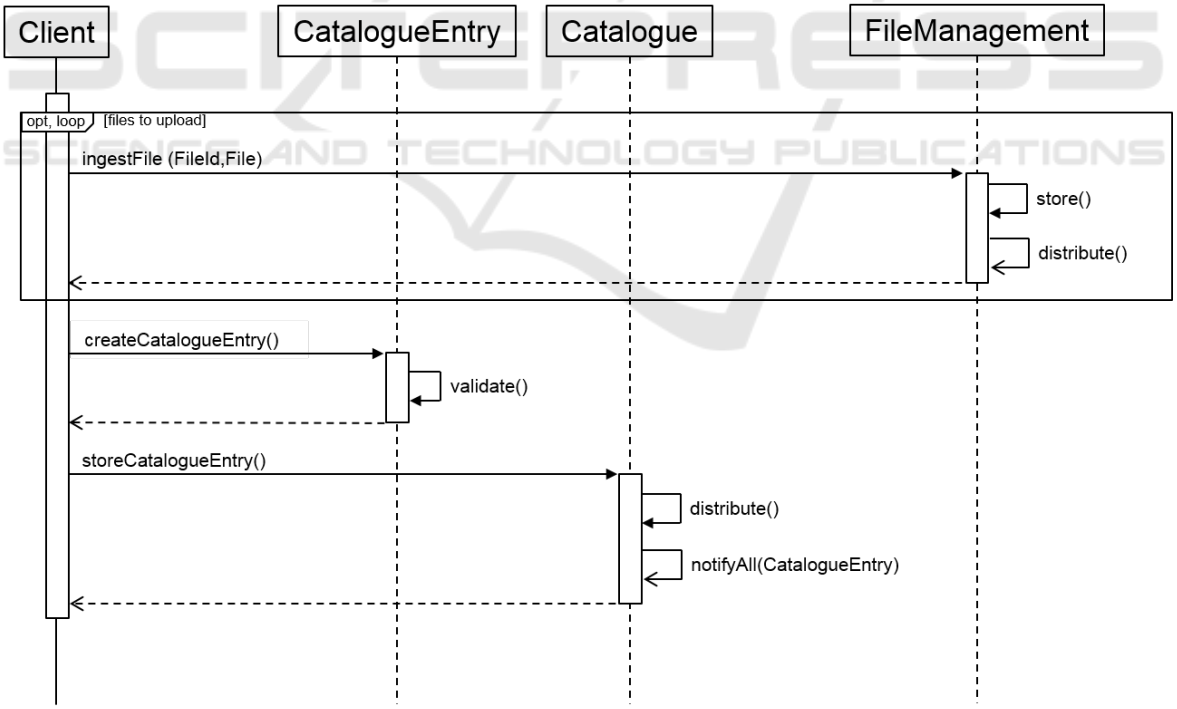

Figure 3: The process view describing the ingest process.