Ulm University | 89069 Ulm | Germany

Faculty of Engineering, Computer Science and Psychology
Institute of Databases and Information Systems

Evaluating Domain-Driven Design for Refactoring Existing Information Systems

Master Thesis at Ulm University

Submitted by:
Hayato Hess
Hayato.Hess@uni-ulm.de

Reviewers:
Prof. Dr. Manfred Reichert (Ulm University)
Dr. Jan Scheible (MERCAREON GmbH Ulm)

Advisor:
Nicolas Mundbrod (Ulm University)

2016

Revision May 2, 2016

Abstract

When modifying existing information systems, one might face some difficulties at best, a nightmare at worst. This is commonly caused by the *Big Ball of Mud* design anti-pattern [FY97], leading to "*code that does something useful, but without explaining how*" [Eva04]. To tackle this problem, Eric Evans suggests *Domain-Driven Design* (DDD) for the creation of new software systems [Eva04]. Having a model based on a shared language at its core, DDD helps to improve the understanding and thereby facilitates communication between the involved parties. As DDD was designed for new systems, the question naturally arises at this point, whether and how DDD-based architectures can be created from existing information systems. The application of DDD to an existing information system might result in benefits in communication and maintenance while minimizing development risks. To evaluate the feasibility, the existing *Time Slot Management* system of MERCAREON GmbH is used as a research object for this thesis.

To evaluate DDD for refactoring existing information systems, this work first extracts business knowledge from an existing information system. This knowledge is comprised of the used terms, supported operations and the data accessed by these operations. Based on this, DDD models are created both manually, and automatically to be used for refactoring towards an architecture with a domain model at its core. In order to automatically generate models, several transformation approaches are utilized, artifact-model transformations generate the initial DDD model, multiple model-model transformations transform the initial model into a more abstract DDD model, and model-artifact transformations ultimately generate source code based on the DDD model.

The approach has to be flexible enough to support continuous changes as the Time Slot Management system by MERCAREON is constantly maintained and improved.

To handle the challenges arisen from continuous maintenance, a *Java prototype* for the creation of DDD models has been developed accordingly and is presented in this work as well. It not only supports the creation of the model but also helps to track and communicate the impact of changes on the model. Furthermore, bubble strategies, which are also suggested by Eric Evans [Eva13], were evaluated as a strategy to evolve from an original architecture to one with a domain model at its core.

Based on a proof of concept dealing with a *Live Yardview*, a new view for the current Time Slot Management system implemented as a DDD style, the approach of this work shows promise for the evolution of an existing system towards a DDD-based one.

Acknowledgments

First, I would like to thank Prof. Dr. Manfred Reichert, my supervisors Dr. Jan Scheible and Nicolas Mundbrod for their interest and invaluable guidance.

Secondly, I would also like to thank the Institute of Database and Information Systems as well as the MERCAREON company for their support making this thesis possible.

Last but therefore not least important, I thank my parents, my sister and my friends (including MERCAREON's staff) supporting me with their company, advice and delicious meals.

Contents

1 Introduction

1.1 Motivation

One way to tackle complexity involved with information systems[1] is through abstraction, problem decomposition, and separation of concerns. Software architects aim to achieve this by moving the focus from programming to solution modeling resulting in a more human-friendly abstraction [SK03]. Over time, several different model solutions have been proposed, such as *Domain-Driven Design* (DDD) [Eva04]. Having a model at its core, DDD supports the creation of a more safe and sound software architecture as well as it aims to be as human-friendly as possible.

Software architecture, as defined by Ralph Johnson, is a subjective, shared understanding of a system's design by the system's expert developers [Joh02]. This understanding ranges from knowledge what the major components of a system are to how they interact. Also, it contains early design decisions that are perceived to be important and are hard to revert in the later stages of the project [Joh02].

The question arises what steps have to be taken when software architecture outgrows it's original purpose, slowly drifting towards a *Big Ball of Mud* [FY97] which is an anti-pattern for software systems lacking perceivable architecture and therefore increasing maintenance efforts and costs. Architectural refactoring, as suggested by Stal [Sta07] was designed to solve this problem. Though it was suggested in 2007, this refactoring method is still in its infancy today [Zim15].

[1]An *information system*, in this context, is a software system "*for collecting, storing, and processing data and for providing information, knowledge, and digital products*" [Bri16]

This thesis examines the possibility of applying architectural refactoring towards DDD on an existing information system. The challenge faced is that DDD was meant for the creation of new green or brown field systems but not for refactoring existing ones.

1.2 Problem Statement

Architectural refactoring is a common topic in today's software development. When information systems have evolved over time, they become difficult to maintain. This is due to the fact that system's architectures become obsolete as most were not designed to be adapted to new requirements. By utilizing architectural refactoring, the obsolete architectures can be adapted to the new needs. This leads to the problem that the new refactored architecture will also become obsolete over time. For this reason, the refactored system should have an adaptable architecture removing the need of repetitive refactorings and therefore reducing costs in the future. Domain-Driven Design provides such an architecture by being based on an adaptive model.

The MERCAREON company (see Section 2.1) decided to incorporate a new software architecture in their *Time Slot Management System* (see Section 2.2). This brings MERCAREON in the difficult situation of having to maintain the system while adding new desired features. By choosing *Domain-Driven Design* (see Chapter 3) as the future architecture, MERCAREON aims to reduce maintenance time and cost by making the system's architecture more human-friendly.

The problem faced with DDD is that it was not designed for architectural refactoring process. Therefore, a mapping of the old architecture towards DDD is required in order to extract the knowledge for Domain-Driven Design from the existing information system in an efficient way. This mapping must be created in a feature complete way supporting fast adaptation, thus, the question arises whether parts often being subject to change like *Entities* (see Section 3.4.1), *Value Objects* (see Section 3.4.2), *Aggregates* (see Section 3.4.3), and *Services* (see Section 3.4.5) could be obtained automatically whereas parts, seldom changed, like *Modules* (see Section 3.3.4) and *Bounded Contexts* (see Section 3.3.2) are maintained manually. Furthermore, protective layers must be created preventing the leak of unrelated information into the new architecture (see Section 3.4.7).

1.3 Goal

The goal of MERCAREON is to obtain a Domain-Driven Design based software architecture which can immediately be used for the development of new features integrated in the old system's architecture while the old architecture is step by step refactored to the new, DDD-based architecture. When the architectural refactoring process is complete, the *Time Slot Management system* (TSM system) will mainly be powered by an architecture having a domain model at it's core. The goal of deploying this new architecture, driven by an ubiquitous language, is to impact communication, to reduce misunderstandings, and providing improved communication channels. Additionally, a model based, better structured, and self-explanatory code base facilitates the code maintenance. This in turn, improves the shipping time of new features due to the ability to adapt the model to new requirements while assuring an overall high quality of code.

Therefore, the goal of this work is to find and establish an approach to refactor the old system's architecture to the new DDD-based architecture by extracting knowledge such as *Ubiquitous Language* (see Section 3.2) and *Business Operations* (see Section 5.1.2) from the old system used as a basis for the new DDD-based system. The approach must support that system changes can easily be verified and translated to the new DDD driven system architecture later on. As discussed in Section *Problem Statement* (see Section 1.2), the parts being subject to frequent changes should be translated to the DDD-based architecture automatically. Finally, the DDD-based architecture is to be translated to Java code automatically utilizing template based code generation.

1.4 Structure of Work

Firstly, the Chapter *Context* (see Chapter 2) discusses the reason and circumstances in which this thesis was created highlighting the problems solved by this work. The Chapter *Domain-Driven Design* (see Chapter 3) explains DDD and its components, such as the ubiquitous language, laying the theoretical groundwork for the heart of this work, the architectural refactoring process. The Chapter *Related Work* (see Chapter 4) qualifies other applicable modeling approaches and discusses a distributed database approach showing similarities to this work's approach. Then, the Chapter *Domain-Driven Design for an Existing Information System* (see Chapter 5) describes the automatic creation of DDD models ultimately enabling architectural refactoring. The Chapter *Prototype* (see Chapter 6) utilizes the previously defined model-model transformation theory showcasing the utilization and generation of fragments used by the architectural refactoring technique. In addition, the refactoring is evaluated in a case study of the Live Yardview in the company of MERCAREON. Finally, the Chapter *Conclusion and Future Work* (see Chapter 7) summarizes and concludes this thesis and points out possible future work.

Figure 1.1 shows the methodology used in this work. It starts with researching the foundation of the topic (see Chapter 3) and continues with evaluating related work (see Chapter 4). Next, combining the found approach with the techniques suggested by Eric Evans [Eva04], the architectural refactoring approach is presented (see Chapter 5). Finally, the approach is prototyped (see Chapter 6) and realized (see Section 6.5).

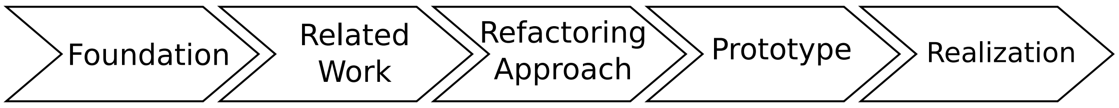

Figure 1.1: Methodology

2 Context

The goal of this chapter is to highlight the context in which this work was created. For this, an overview of the MERCAREON company (see Section 2.1) is provided. Then their *Time Slot Management System* (see Section 2.2) is presented showcasing its context and challenges which ultimately lead to the creation of this work.

2.1 About MERCAREON

MERCAREON GmbH, part of the TRANSPOREON Group, is a software company located in Ulm with a branch office in Poland, having currently 28 employees and being founded in 2009. As the name indicates, the companies of the TRANSPOREON Group are creating logistic software supporting the transport and management of goods. The company of MERCAREON provides a software system which supports the delivery process of ordered goods. As it is based on time slots, the system created by MERCAREON is called the Time Slot Management system and was decided to be architectural refactored towards Domain-Driven Design.

2.2 Time Slot Management System

The *Time Slot Management system* (TSM system) is a web based system enabling a *Carrier* to book a specific time slot to deliver the goods he was ordered to transport.

When a *Retailer* orders goods from a *Supplier*, the *Supplier* procures the goods and assigns the transport of the goods to a *Carrier*. As shown in Figure 2.1 in red, there was no definitive communication channel between the Carrier and the Retailer before the TSM system was in effect. A Carrier usually used phone calls or transmitted lists via fax to make delivery appointments leading to imprecise timed arrangements as there were usually no information about the unloading capacities at a certain time. For the Carrier, this situation meant inestimable waiting times until the transport vehicle was handled at its destination. The Retailer had a logistic problem since he had no information on the time of arrival nor the quantity of goods. Besides, the Supplier had no means of assessing his Carriers in terms of efficiency.

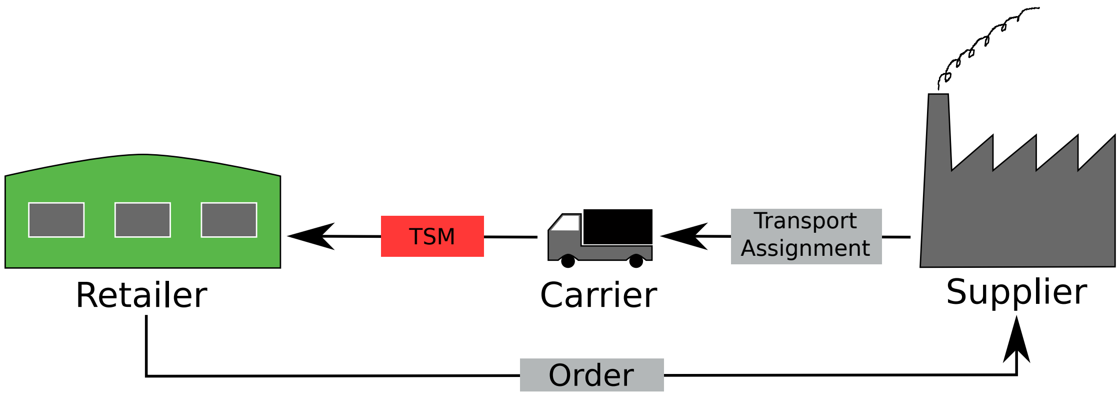

Figure 2.1: Environment of the Timeslot Management System[1]

With the help of a TSM system (see Figure 2.1), the Retailer enters the order, identified by an *order-number*, into the system. When the Carrier gets the delivery assignment, a time slot can be booked for the pending delivery by using the *order number* and providing additional information such as how many goods are being transported enabling the TSM system to calculate an estimated unloading duration. Thereby, the Retailer may perceive the important information when the Carrier arrives and how long it approximately takes to unload the delivery. Moreover, the Retailer can plan and additionally control deliveries by constraining the bookable time slots (e.g. limiting slots for beverage deliveries at a certain gate). Knowing when to deliver and how long the delivery takes, the Carrier in turn has lower idle times and, thus, will likely save money. The supplier can view statistics about the Carriers' deliveries and, therefore, rate their effectiveness.

[1]adapted from MERCAREON GmbH

2.3 Refactoring towards a Domain-Driven Design

Although the Time Slot Management system is running successfully, MERCAREON decided to change the system's architecture by introducing Domain-Driven Design.

The first reason originates from the TSM system's long history. Before MERCAREON was founded, TRANSPOREON had already worked on the creation of a TSM system in C#. MERCAREON took over in the year of 2009 and ported the application from C# to Java resulting in the system containing both MERCAREON and deprecated TRANSPOREON terms that may confuse developers. Furthermore, when communicating in-house or with customers, the staff of MERCAREON faces another type of communication problem: for example, a customer care member has to make sure that he will not mix up both customer-specific and in-house communication terms. In-house communication can thereby range from talking to other customer care members, to TRANSPOREON colleagues, speaking their own diverged dialect, or to the developers. This can be very daunting especially when the system is evolving and is constantly being influenced by external factors such as companies working with, or contributing to the TSM system. In addition, developers of the system are based partly in Germany and partly in Poland. The spatial distance in combination with fuzzy terms increases communication difficulties. To solve this issue, an *Ubiquitous Language* (see Section 3.2) needs to be introduced as suggested by DDD.

Secondly, in systems with ever changing requirements, complexity is likely to increase while maintainability decreases. The TSM system, in particular was created with a traditional layered architecture. Further, it is deployed in an environment containing various companies (*Retailer*, *Supplier*, and *Carriers*) continuously facing newly arising requirements. Hence, the TSM system will likely suffer from rising complexity in the future. As any architecture's maintainability, Domain-Driven Design architectures will also suffer from high complexity but not as much as traditional approaches (see Figure 2.2). This is due to the fact that *Domain-Driven Design* (DDD) helps coping with the complexity by using a domain model at its core abstracting reality to make it easier to grasp for the developers and other involved parties [Eva04; Avr07]. With DDD program code is simplified making it easier to grasp its meaning [Eva04] thus code starts to be part of the documentation [Fow05a].

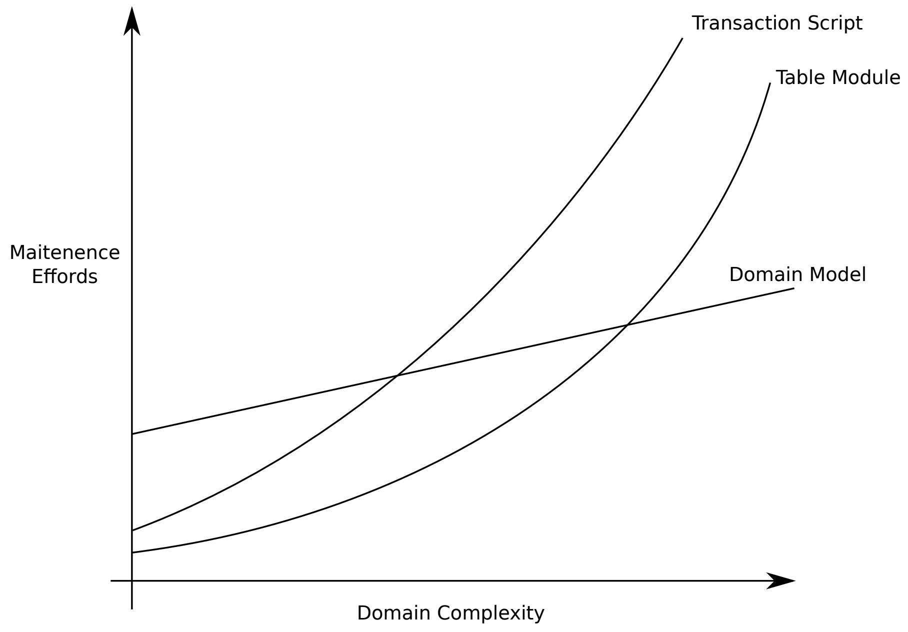

Figure 2.2: Maintenance Costs vs. Complexity of Domain Logic[2]

[2]adapted from [Fow02]

3 Domain-Driven Design

Domain-Driven Design (DDD) is a *Model-Driven Engineering* (MDE) software development approach designed "*for complex needs by deeply connecting the implementation to an evolving model of the core business concepts*" [Git07]. It aims to provide practices and terminology enabling design decisions which focus and accelerate the creation of complex software. It therefore is neither a technology nor a methodology. [Git07]

The goal of this chapter is to introduce the DDD approach (see Figure 3.1) as required for the architectural refactoring as stated in the Section *Goal* (see Section 1.3).

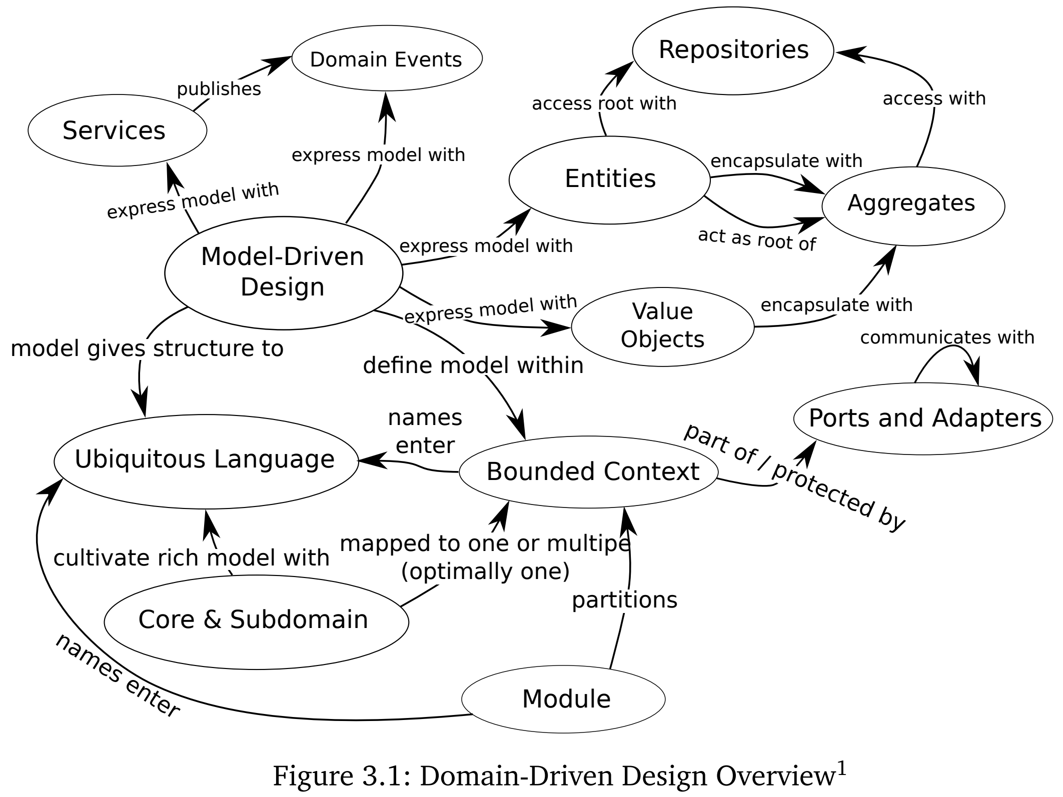

Figure 3.1: Domain-Driven Design Overview[1]

The approach has three premises: firstly, a collaboration between developers and domain experts is required to get the conceptual heart of the problem (see Section 3.1). Secondly, complex designs are based on models such as the ones suggested by Eric Evans (see Chapter 3) or presented in the *Related Work* (see Chapter 4). Thirdly, the main focus should be on the core domain and its domain logic (see Section 3.3.1) [Git07].

Eric Evans describes how models are utilized by DDD in three different ways [Eva04]:

The "*backbone for language*" – the *Ubiquitous Language* (see Section 3.2) – that is derived from the *Domain Model* (see Section 3.1) specifies the terms used by the participating parties and forms a foundation of DDD. [Eva04] Then models of the *Strategic Design* (see Section 3.3) serve as distilled knowledge. Additionally, the models convey how the domain model is structured while distinguishing the elements of most interest. They are furthermore used to break down and relate concepts helping to select terms defining the way of distributing the parts of the application and specifying the boundary for components and external sub systems. The strategic design contains parts which are seldom changed in later stages of the project and are therefore created manually. The shared language supports involved developers and domain experts to transform their knowledge into this second model usage [Eva04].
Lastly, the *Tactical Design* (see Section 3.4) is created by utilizing *Model-Driven Design* (MDD) which helps to reflect the domain model in the systems' software design. The model thereby serves as a bridge to the implementation. The code can therefore be grasped more easily as it is based on the model. The model contains the building blocks of the system surrounding parts that are subject of frequent changes [Eva04]. Therefore, their generation is automated as explained in Chapters 5 and 6.

When looking at the final software design, the *strategic design* represents the distribution of code, modules and high level packages in system whereas *tactical design* contains low level packages wrapping the actual different classes.

Ports and Adapters (see Section 3.4.7) then describes how a DDD-based architecture combining both *strategic* and *tactical design* could look like.

[1]adapted from [Eva14]

3.1 Domain Model

> *If the design, or some central part of it, does not map to the conceptual domain model, that model is of little value, and the correctness of the software is suspect.*
>
> – Eric Evans, *[Eva04]*

When creating complex business software, problems arise when an understanding of concepts is missing. For example, in order to create a booking software, proper understanding of the *domain* – the "*sphere of knowledge, influence or activity*" [Eva14] of booking – is required. Knowledge of the problem can be obtained through inquiry of *domain experts*. This approach leads to the problem, though, that an approach is required to abstract the acquired knowledge which will finally lead to working code.

Eric Evans suggests to create a *domain model* in order to tackle this problem, which is an internal representation of the domain [Eva04]. The domain model thereby stands for the solution of the problem, an abstraction of the reality. For this, abstraction, refinement, division, and grouping of the information gathered about the domain are required [Avr07]. The *domain model* is therefore the organized and structured distilled problem knowledge containing the domain's key concepts [Bro14]. It should be communicated to and shared with all involved parties ensuring its integrity and supporting a common understanding. When dealing with changing requirements, creating a perfect model covering all future requirements is impossible. However, it can be continuously evolved ensuring to be as close to the domain as possible [Avr07].

3.2 Ubiquitous Language

When working with a team of experts from multiple areas, a further challenge arises: the communication barrier. As they rely on different concepts, experts being from different areas face a problem of mutual understanding (see Figure 3.2). Experts from different areas are speaking *in their own language* (e.g. developers speak of databases, events, ...) [Eva04]. Moreover, people tend to have a different language when communicating

Figure 3.2: Communication Barrier between Stakeholders

in text or speech, they tend to create a “layman’s language”[2] to communicate difficult aspects [Avr07]. As an example for this, a developer could use the pictorial language of a computer reading and writing on a note-book when trying to explain operations on computer memory. This communication barrier proves to be a huge risk for projects since misunderstandings drastically reduce the chance of success [Eva04].

To circumvent this problem, an *ubiquitous language* derived from the domain model shared by all participating parties is required [Avr07; Eva04; Bro14]. As the word “ubiquitous” states, the language is used by the involved experts and (third) parties not only when creating the application but also when communicating with each other. It is of importance not to mistake the ubiquitous language as a global and company wide language. It is meant to have an ubiquitous meaning with unambiguous words and phrases only in a specific part of a domain (see Section 3.3.1). In fact, the larger the ubiquitous language boundary is, the higher the ambiguity making it more and more “fuzzy”. Therefore, its boundaries have to be explicit helping to make it precise and well defined [Vau13].

When developing the language, the domain’s key concepts are introduced whereby the language’s nouns are mapped to objects and their associated verbs become part of their behavior [Avr07].

[2]The use of simple terms that a person without specific knowledge in a complex area can understand.

During the development of the system, especially with changing requirements, the ubiquitous language must be continuously maintained and updated by the involved domain experts. Whenever a domain expert thinks a phrase or word sounds wrong, he should raise concerns so that the language can be further improved [Vau13; Eva04].

3.3 Strategic Design

Containing how to distill the domain into distinct parts small enough for the human mind to handle, the strategic design is important for handling complexity.

This Section first discusses the *Core and Subdomains* (see Section 3.3.1), partitioning the application based on the importance. In an optimal case, *Bounded Contexts* (see Section 3.3.2), serving as a ubiquitous language barrier would be directly mapped. In reality, however, they may intersect with one or more core or subdomains. Last, *Modules* (see Section 3.3.4) are presented partitioning a bounded context into smaller logical units.

3.3.1 Core and Subdomains

The word *domain* is often misleading unfortunately. When used in the context of DDD, the word might lead to the conclusion that the goal is to create an all-knowing model of the whole business in DDD. This is not the case as already hinted in Section 3.2. Creating a DDD model, the domain is partitioned naturally into the *core domain* and several *subdomains* based on their business relevance. The former contains the heart of the application: the critical core that will get the most attention in the shape of resources, and experienced developers. It is supported by the subdomains, which can be divided into *supporting subdomains* and *generic subdomains*. Subdomains come in these two different forms helping to prevent the core domain to get overly complex and, thus, harder to grasp [Eva04; Vau13].

When partitioning, the core domain can be determined by asking the following questions [Oli09]:

- What makes the system worth writing?

- Why not buy it off the shelf?
- Why not outsource it?

Supporting subdomains offer supporting functions to the business or model aspects of the business. Supporting subdomains are required, but are not as important as the core domain. Therefore, more inexperienced developers can be assigned to the teams responsible for the supporting subdomains or they might sometimes even be outsourced [Vau13; Eva04; Oli09]. *Generic subdomains* contain parts that are not "core" to the business but are still required. They contain specialties and support the system in a generic way. However, they are still essential for the system's function. Usually these functions can be purchased or outsourced [Eva14; Eva04; Oli09].

The TSM system by MERCAREON is their main area of competence sold to their customers (see Figure 3.3). As no comparable system exists in Europe, it can't be bought off the shelf and it makes the system worth writing. Additionally, being most crucial to business and MERCAREON having the know-how, it makes no sense to outsource the TSM system. All in all, it is save to assume the TSM system is the *core domain* of MERCAREON.

The User Management connected to the TSM system manages its users and therefore directly supports the core domain. Therefore, User Management is a *supporting subdomain*.

In contrast, reporting is an (outsourced) component with which companies can access statistics of their bookings. It is also part of the business but not crucial to the core. Therefore, reporting is a *generic subdomain*.

3.3.2 Bounded Contexts

Especially in large projects, the domain can have words and phrases colliding with each other making the *Ubiquitous Language* (see Section 3.2) fuzzy and therefore hard to grasp. When one wants to merge different models into one big system, the result gets prone to bugs, is difficult to understand and therefore hard to maintain. To solve this dilemma, the use of *bounded contexts* is suggested by [Eva04]. They serve mainly as the ubiquitous language boundary and can contain multiple aggregates [Eva14; Vau13]. Each word or phrase has to be unique within one bounded context. Furthermore, the assigned team, code base, and resources like the database have to be differentiated by the bounded context in order to protect the model [Eva14]. When working with Java, for example, a project may be divided in separate JAR, WAR or EAR files or create multiple dependent projects [Vau13].

In a perfect (green field) environment, core and subdomains can be mapped to bounded contexts one to one. In reality, a bounded context can span multiple core and subdomains. It is also possible that multiple bounded contexts are part of one core or subdomain.

The communication between bounded contexts proves to be difficult because each context has its own ubiquitous language. Therefore, a translation layer is required translating messages between the contexts into their respective language (see Section 3.3.3).

Example 1. The TSM system was chosen to be a bounded context of MERCAREON. As seen in Figure 3.3, the bounded contexts are mapped to core or subdomains.

3.3.3 Bounded Context Communication

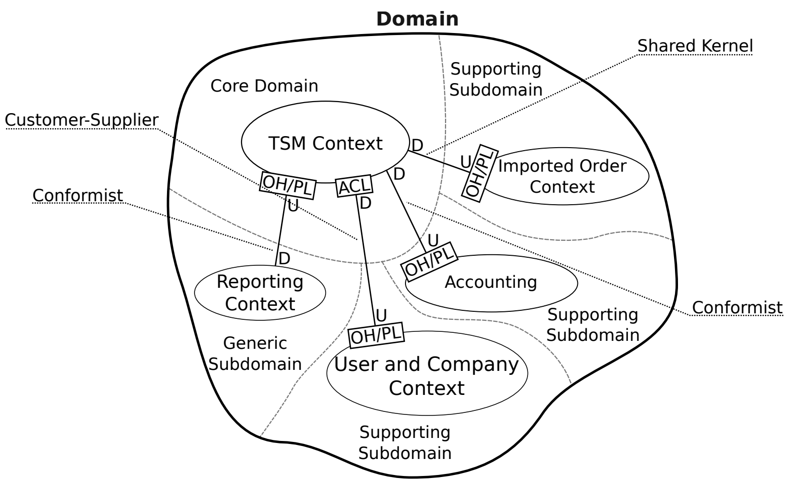

Figure 3.3: MERCAREON's Subdomains and Bounded Contexts

When two bounded contexts communicate by exchanging messages, they have two different kinds of interaction (see Figure 3.3):

Firstly, the way one bounded contexts influences the other is described. In [Eva04], this is modelled as an *upstream* (U) and *downstream* (D) relation arising from the picture of a city polluting a river. The city itself is not affected but affects cities down the stream of the river. Logically, cities upstream can not be affected by cities down the stream and, therefore, they have no direct incentive to avoid polluting the river. From a model point of view, the upstream model provides an interface to exchange information and the down stream mode has to cope with what kind of information it receives and how the information is represented [Eva04].

Secondly, the relationship that exists between two bounded contexts is discussed. When the teams maintaining two bounded contexts must cooperate since either both of their contexts succeed or fail together, the bounded contexts share a *Partnership* relation. This relationship requires coordinated planning of development and integration. The interfaces must be created in a way satisfying the needs of both contexts.

Forming an intimate relationship, *Shared Kernel* shares a part of the model and associated code. It is of importance to define small explicit boundaries defining which subset of the domain model is shared. When the shared part is changed, both responsible teams have to be consulted. It is suggested to define a continuous integration process keeping the shared model small aligning the ubiquitous language of the two involved teams.

Customer-Supplier relationships exists only for up and downstream relationships. The upstream team's success is mutually dependent of the downstream team's success. The downstream team's needs must be addressed by the upstream team.

Last but not least, the *Conformist* relation also only exists for up and downstream relationships in which the upstream team has no incentive to address the downstream team's needs. The downstream team has to eliminate the complexity of translation by using parts of the model created by the upstream team [Vau13].

There are three concepts enabling a regulated communication: The *Open Host Service* (OHS) as can be seen on the upstream contexts in Figure 3.3, defines the protocol for accessing subsystems as a set of services. The protocol has to be open for all parties who need to communicate with the system.

The *Published Language* is required as the translation (e.g. via Anti Corruption Layer) requires a common language. The common shared language should be well documented and expresses necessary domain information enabling the translation into and, when necessary, out of that common language. The published language is often combined with the open host service.

Last, the Anti Corruption Layer.

Anti Corruption Layer

When working with a domain model, special attention has to be paid that it stays pure. It has to be ensured that application and other domain logic does not leak into it, especially when systems must communicate over large interfaces. The difficulties in mapping these two systems' models can corrupt the resulting model [Eva14; Vau13].

The *Anti Corruption Layer* (ACL) is the protecting mechanism of the domain model. When communicating with with another bounded context or external systems such as databases, the ACL can be used as a two way translator translating between the external system and the current system's language [Eva14; Vau13]. For bounded contexts, it is used when having limited to no control over the communication. In a *Shared Kernel*, *Partnership*, or *Customer-Supplier* relationship the ACL translates between different domain models. The layer communicates to the other system through its interface requiring little to no modification to it. Then, internally, the communication is translated to the target's model.

3.3.4 Modules

When creating a complex application, bounded contexts can get too big to apprehend the relationships and interactions. In such a case, it is recommended to split them into *modules*. Modules' sole purpose is to "*organize related concepts and tasks in order to reduce complexity*" [Avr07]. As being used in most of the existing software projects, modules help to manage complexity and improve code quality. This is achieved by grouping related classes into modules. These modules contain a cohesive set of concepts, increasing code cohesion[3] and decreasing coupling[4].

[3]Measures the relationship between functional components, [SMC74].
[4]The strength of the relationships between modules, [Abr+01].

When deciding which parts of an application to be grouped into a module, it is recommended to select models separating high-level domain concepts and their respective code. Further, they should be given names from the ubiquitous language representing these [Avr07].

Example 2. The TSM system by MERCAREON is separated into several different modules (see Figure 3.4) interacting with each other in the bounded context.

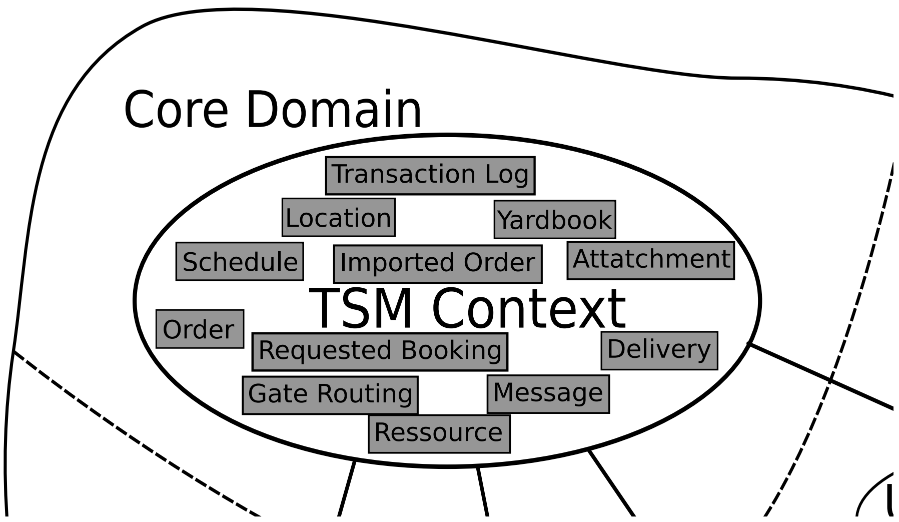

Figure 3.4: TSM System Modules

3.4 Tactical Design

The *tactical design* contains the building components that connect models to the implementation. The implementation is part of a module in a bounded context (see Section 3.3).

Entities (see Section 3.4.1) and *Value Objects* (see Section 3.4.2) are the smallest pieces of the tactical design. *Aggregates* (see Section 3.4.3) wrap both entities and value objects and are stored in *Repositories* (see Section 3.4.4). *Services* (see Section 3.4.5), in turn, hold operations performed on aggregates and *Domain Events* (see Section 3.4.6) inform about internal or external state changes.

3.4.1 Entities

Entities are objects in DDD defined by an unique identity (see Figure 3.5) remaining the same through and beyond the life cycle of the system. They are not defined by their attributes enabling multiple different entities with the same attributes (e.g. person entities sharing the same name) [Avr07].

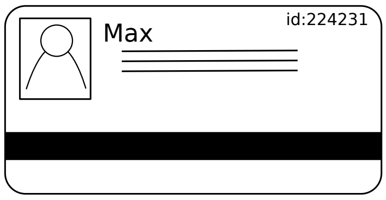

Figure 3.5: Unique Identity for the Person 'Max'

The system has to ensure the uniqueness of the entity's identity. A database could, for example, create these unique identities [Eva14].

The identities can range from technical entities to natural entities. For example, an *unloading gate* entity could have some sequential auto-generated identifier or its identifier could be constructed out of a set of human readable metadata (e.g. company – country – locationName – gate name) [Vau13].

Example 3. As an example, *Company*, *User*, *Role*, *Booking*, and *Task* are entities in the context of MERCAREON's TSM system.

3.4.2 Value Objects

Since many objects in a system have no conceptual identity, creating entities for each of these would bring no benefit. In fact, it would corrupt the system by introducing the required complexity to find unique identities for all these objects. Therefore Eric Evans suggested the so called *value objects*. Value objects have no identity and represent the objects of the system that don't apply for being an entity. Having no identity they can easily be created and removed which simplifies the design. Moreover, value objects are recommended to be modeled as immutable objects[5]. This brings the advantage of shareability, thread safety and the absence of side effects. Although value objects can

[5]The state of immutable objects can not be changed after creation. Therefore, the object has to be replaced by a new instance when its state is changed.

hold multiple attributes, it is recommended to split long lists of attributes into multiple value objects. The attributes held by a value object should be conceptual whole. For example a location can have GPS coordinates and a name but should not contain the colors of buildings located there. [Eva14]

Example 4. As an example *Order Number*, *Company Id* and *Delivery Quantity* are value objects in MERCAREON's TSM system.

3.4.3 Aggregates

> *A much more useful view at aggregates is to look at them as consistency boundaries for transactions, distributions and concurrency.*
>
> – Eric Evans, *[Eva09]*

Aggregates define object ownership and consistency boundaries. Aggregates gather entities and value objects into groups enforcing data integrity and abidance of invariants. They are globally identified and accessed by an ID. Every aggregate has one root entity (see Section 3.4.1). It is the only part of the aggregate that is accessible from outside and holds references to all other entities and value objects of the aggregate. As soon as a change to an inner part of an aggregate is required, the root entity has to be asked to apply these changes while maintaining the aggregate's invariants. Other objects can only hold references to the root. As value objects are immutable the root entity can decide to expose them to its accessors. The accessors of aggregates thereby have to pay attention to reference value objects only temporary or they are in danger of working with outdated values. Furthermore, holding references could lead to memory leaks since, as soon as the root entity is deleted, all inner objects are not supposed to be referenced any more and should be deleted too. [Avr07]

A problem faced when defining aggregates is that aggregates should both not be too large and too small. When they are designed too large, they will likely perform badly. Especially when lazy loading comes into play, a small change to an aggregate may require to load the whole aggregate into memory. In addition, as realizing a transactional boundary, modifying aggregates will lock all of its components [Vau13]. Since only aggregates can

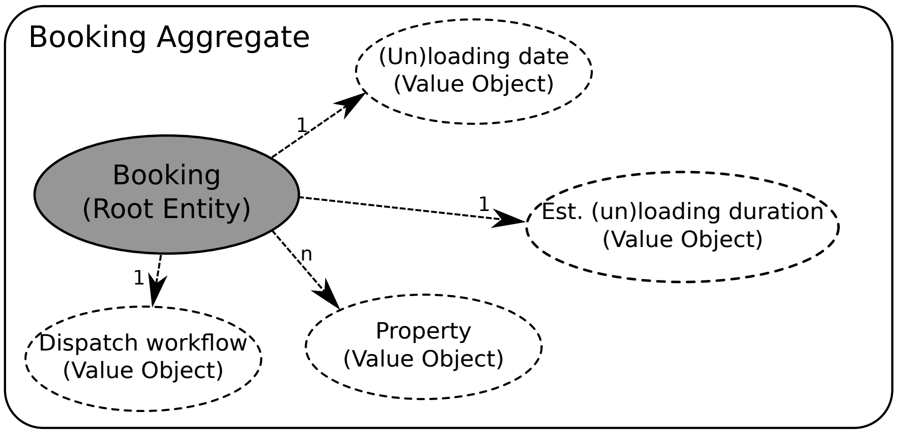
Booking Aggregate
Booking (Root Entity)
(Un)loading date (Value Object)
Est. (un)loading duration (Value Object)
Dispatch workflow (Value Object)
Property (Value Object)
1
1
1
n

Figure 3.6: Booking Aggregate

be obtained from repositories (see Section 3.4.4), they work as consistency gatekeepers for the data. One important rule regarding aggregates is, that only one may be modified during a transaction at a time.

Example 5. The *Booking* is represented as an Aggregate (see Figure 3.6) with the Booking as its root entity which provides the uniqueness and contains several value objects.

3.4.4 Repositories

The question of how instances of *Aggregates* (see Section 3.4.3) can be obtained obviously arises while working with DDD. One option is to trigger the creation operation giving us a reference to the root entity of an aggregate [Vau13].

Another option is to traverse entity references between aggregates. For this, a reference to any entity is required. *Repositories* can give us the reference to a root entity of an aggregate. From an object oriented point of view, these entities are newly instantiated through data retrieved from an external system (e.g. a database). From the DDD's point of view *existing* entities are referenced. Therefore, this operation is referred to as "*reconstruction*" [Eva14; Vau13].

Repositories can be seen as an *Anti Corruption Layer* (see Section 3.3.3) around databases [Vau12] and, as a rule of thumb, should not be accessed from within aggregates [Vau13].

Example 6. The *Booking Repository* enables access to *Booking Aggregates* by providing access to their root entities.

3.4.5 Services

> *For example, to transfer money from one account to another; should that function be in the sending account or the receiving account? It feels just as misplaced in either.*
>
> – Abel Avram, *[Avr07]*

When developing the domain model, there are typically behaviors that can not be incorporated into entities or value objects. However, they represent important requirements and, therefore, they can not be ignored. If these behaviors were added to entities or value objects, they would make them more complex than necessary and introduce functionality that does not belong to these objects. Furthermore, working with multiple aggregates would be impossible since repositories should not be called within aggregates [Avr07; Vau13].

Services solve this problem by providing stateless functionality important to the domain. They can access repositories and therefore refer to multiple aggregates in the domain. Another characteristic of services is that the operations performed in them refer to a domain concept whereas, as the quote above already states, they do not naturally belong to either entity or value object [Avr07].

Services are subdivided into two categories, the *domain services* and the *application services*.

Domain Services

Domain services implement functionalities required for the application. They require domain-specific knowledge for providing the functionalities. The domain service does not provide security or transactional safety since its operations are too fine grained for this purpose [Vau13].

Example 7. Calculating the amount of time slots for a booking contains domain logic and is therefore part of the domain service.

Application Services

> *Keep Application Services thin, using them only to coordinate tasks on the model.*
>
> – Vernon Vaughn, *[Vau13]*

Residing in the Application Layer (see Section 3.4.7), the *application services* contain no domain logic but directly communicate with the domain model. Application services offer all possible operations supported by the bounded context while remaining lightweight. Application services utilize repositories to operate on domain objects. In summary, they provide the execution environment where operations are coordinated to the domain model (including the domain services). Moreover, an application service controls transactions, and ensures the state transitions in the model are handled atomically. It is responsible for security and is in charge for event based messaging. When implemented, the application service has either method signatures consisting of primitive types (e.g. short, int, float, double, ...) and *Data Transfer Objects*[6], or, it alternatively uses the *command pattern*[7] [Vau13].

Example 8. To book an order in the TSM system, the application service is queried and asks the *imported order repository* for the *open booking* aggregate. Then, the application service uses a schedule aggregate instance for creating a new booking for the resulting imported order entity. The whole process is transactionally save which ensures that only one booking is created for the orders.

[6]Especially when calls are expensive, more data needs to be transfered with a single call. This is problematic as long parameter lists are not desired and programming languages as Java only support one return value. Therefore a transfer object can be used to assemble all required parameters or results for an operation, [Fow02].

[7]"*Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations*", [Gam95].

3.4.6 Domain Events

Domain events were not included in [Eva04]. Evans later added them to DDD due the benefit of decoupling systems and therefore supporting the creation of distributed systems by enabling different bounded contexts to communicate [Eva09]. Besides, highly scalable systems like high transaction finance software can be created using event sourcing [Vau13; Fow05b].

> “ *Something happened that domain experts care about.* ”
>
> – Vernon Vaughn, *[Vau12]*

As the quote states, *domain events* are created when something important—according to domain experts—has happened. The level of granularity is therefore of importance since not every event in the domain is important. For example, creating an event for every step a person makes might be of interest in the context of a step counter but not in the context of a navigation software.

Events generally have a timestamp, either when they actually took place or when they were recorded. They also have a person associated with them, let it be the person who recorded it or the person responsible for the event's creation. Like value objects, domain events are immutable since they record something that happened in the past [Eva09; Eva14].

When working with domain events, special attention has to be paid as systems might not be consistent all the time [Vau13; Eva09].

For example, the unloading of a truck could be separated into each pallet being moved. However, that might not be important to the domain experts and, therefore, only the start and end of the process are eventually tracked. As soon as one of these events is fired, services of the bounded context responsible for handling unloading eventually notify the interested bounded contexts. The system's user might not see the change directly after committing the unloading process as the change takes place asynchronously and the user's GUI is outdated until the responsible bounded context is notified accordingly.

3.4.7 Ports and Adapters

The *ports and adapters* architecture[8] utilizes the ACL and protects the domain model.

The architecture is comprised of three layers where inner layers are independent from the outer layers.

- *Domain Layer* – This layer contains the domain model (consisting of bounded contexts, entities and value objects), domain services, and repositories. [Vau13]
- *Application Layer* – This layer wraps the domain layer and utilizes its components using application services. It adapts requests from the infrastructure layer to the domain layer. In addition, it dispatches events raised in the domain layer to the outside [Vau13].
- *Adapters Layer* – This layer is the outermost layer wrapping the application layer. It contains adapters to external systems like databases, mailing systems, rest interfaces, messaging systems, but also 3rd party libraries. These adapters are ACLs enabling the system to utilize different protocols and systems without corrupting the domain with the knowledge of these systems. They relay messages from and to the application layer using the domain's language. When a message is received from outside, at a port, an adapter converts the technology specific message into a form suitable for the underlying layers. If an underlying layer wants to send a message, an appropriate adapter transforms the message to something the external system can work with and sends it out on a port [Coc05].
- *Ports* – Defines the exposed functionality to and the applications view of the outside.

In the implementation shown in Figure 3.7, Adapter A and B use *events* to communicate (in this case with another ports and adapters system), C is a message listener connected to a *Message Bus*, D accesses the *REST* API of a three layered system whereas E to H communicate with external or memory databases and are represented by repositories in the DDD context.

[8]Previously known as Hexagonal Architecture [Coc05; Coc06] or Onion Architecture [Pal08]

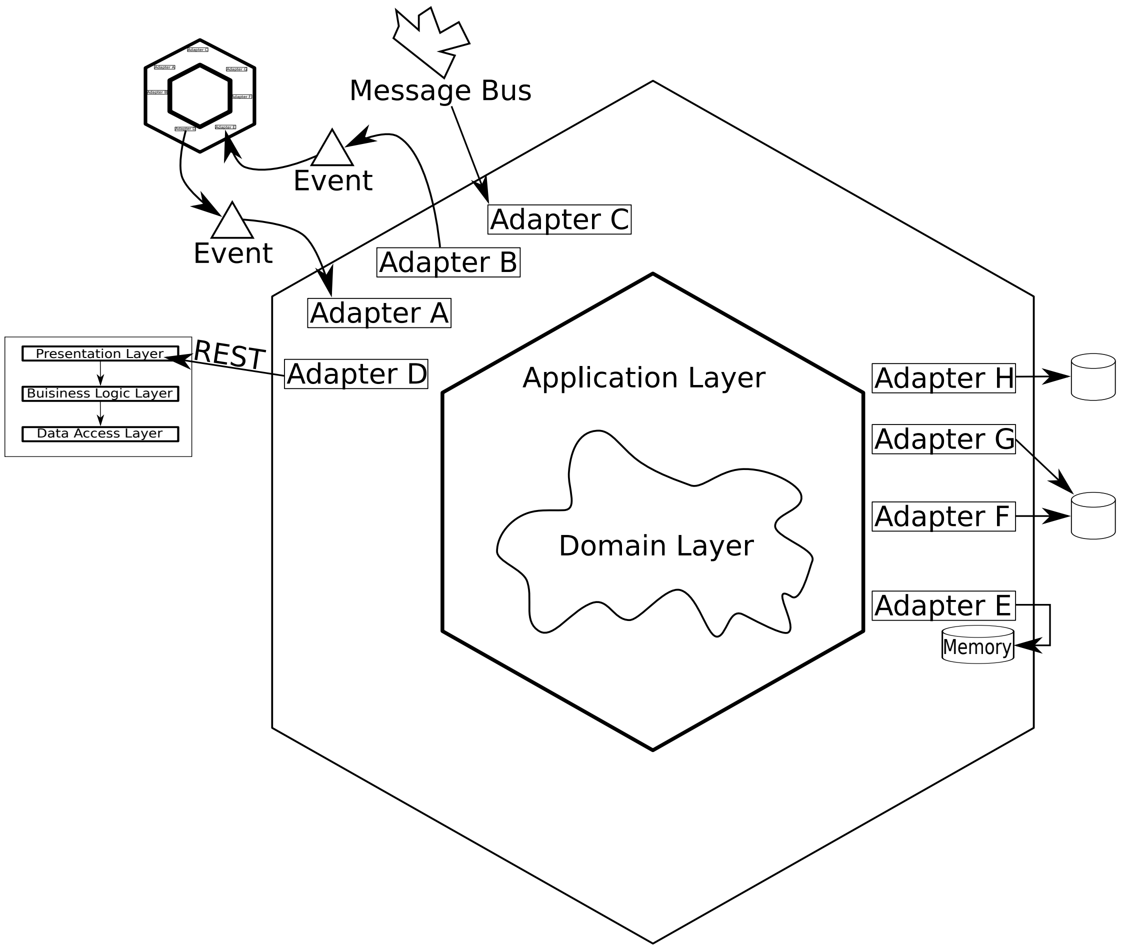

Figure 3.7: Ports and Adapters[9]

The interested reader might have noticed that the independence of inner layers does not fit with the need of inner layers to access parts of the outer layers. For example, an implementation of a repository in the domain layer, most likely requires any form of persistence—be it in file, memory, or database. For this reason, ports and adapters achieve minimal coupling [Fow02] by using the *inversion of control containers* paradigm [Fow04]. The paradigm utilizes dependency injection where outer layers implement interfaces defined by inner layers. For example, the domain layer provides an interface stating that it requires some repository with a given set of functionalities. The implementation in the infrastructure then provides these functionalities by implementing the interface.

[9]adapted and extended from [Vau13]

This implementation is then injected into the inner layer at run time. Looking at the dependencies, the domain layer is independent from the outer infrastructure layer as it provides the interfaces implemented by the outer layer. In the testing phase, parts of the system can be easily replaced by other implementations due to this decoupling [Vau13]. For example, to test the application, the adapters communicating to external databases could be replaced with in-memory test databases.

4 Related Work

This chapter is comprised of two parts. First, *Modeling* (see Section 4.1), introduces and compares three alternative models to Domain-Driven Design for creating and refactoring information systems. Second, Section 4.2 introduces research on the field of *Distributed Databases* and compares a similar fragmentation concept to this work's approach.

4.1 Modeling

Domain-Driven Design (see Chapter 3) utilizes models for the creation of complex information systems. This Section discusses similarities, differences, strengths, and weaknesses of different existing modeling approaches in comparison to DDD. First off, the popular standard *Unified Modeling Language* (UML) (see Section 4.1.1) is introduced due to its strong connection to Model Driven Architecture. Based on this, *Model Driven Architecture* (see Section 4.1.2) is presented, which is, as DDD, a MDE approach and is strongly related to UML 2.0. In general, UML was specially tailored to fit MDA's needs.

4.1.1 Unified Modeling Language

The *Unified Modeling Language* (UML), introduced by the Object Management Group (OMG) in 1994, unifies the three object-oriented design methods *Booch Method*, *Object Modeling Technique*, and the *Objectory Method* providing a common visual notation for describing today's software [Pet13; Tho04]. UML is said to have been established as de-facto standard of software engineering supporting a variety of different diagrams from package diagrams to class diagrams [OMG04]. UML is used differently from company to company: some use its class diagrams, some use it to make a quick sketch on a white board, and some even use it for model-driven development [Pet13]. In a study of 2013 [Pet13], doubts were raised whether UML is really a standard. As of 50 practicing professional software developers, 35 did not use UML at all. They reasoned that UML would not "*offer them advantages over their current evolved practices*" and "*what was good about UML was not new, and what was new about UML was not good*". They further criticized the lack of context dealing primarily with the software architecture than the whole system. Furthermore, UML is reasoned to be unnecessarily complex as the notation is considered to have significant overheads and is too close to programming to be readable by all involved stakeholders. Moreover, it is argued that UML has no "*consistency, redundancy, completeness or quality*" checks leading to difficulties maintaining large project's UML models.

In comparison to Domain-Driven Design, some of the criticism targeted at UML is aligned to DDD's main goals as it focuses on the context while trying to be simple and as human-friendly as possible. As DDD does not specify the type of models to be used but only the content, it is possible though to use models similar to UML in the DDD design process. The usage of UML has its weaknesses though as DDD also utilizes models to work out contexts while UML is meant as a tool to model object oriented issues. For example, the ubiquitous language can not be reasonably modeled in UML as a different representation, such as a glossary, is required.

4.1.2 Model Driven Architecture

Model-Driven Architecture (MDA) is as DDD a MDE approach. It was defined by the Object Management Group and is a model-based approach to cope with complex systems specifying structure, semantics, and notations of models [OMG14]. Moreover, it has a domain model comparable to DDD (see Section 3.1) which is called *Computation Independent Model* (CIM) and specifies systems without constructional details at its core. Models, that are conform to these standards, are called *MDA Models*. These models can be used for producing documentations, generating artifacts, and executable information systems [OMG14]. UML, though formally not required, is used by almost all MDA projects as the 2.0 standard was tailored for MDA [OMG15]. The only exception are projects in specialized fields requiring a specifically tailored modeling language [OMG15].

Using meta-models, the foundation of MDE, MDA can utilize powerful model transformations [Tho04; MV06]. A meta-model is a model that defines the abstract syntax of modeling languages (e.g. UML, BPMN, ER) specifying the model boundaries in the language [OMG14] and serving as a necessary prerequisite for automated model transformation [MV06; OMG10].

A key aspect for the proposed DDD architectural refactoring approach (see Chapter 5) is the automated model transformation as provided by MDA. It is used to automatically generate DDD models that are being subjected to frequent changes (see Section 1.2). Meta-models are a prerequisite for these model transformations and therefore had to be created before (see Chapter 5).

Other than the DDD approach, MDA separates the models into three distinct layers (see Figure 4.1). The CIM layer which serves as a basis for the *Platform Independent Model* (PIM) and the *Platform Specific Model* (PSM) layer [OMG01]. Abstracting technical details, PIM provides a platform independent formal structural and functional specifications [OMG01]. PSM in contrary represents the target platform, such as JavaEE [Ora] or .Net [Mic07] enabling model transformations from PIM to source code [Tho04]. Moreover, PSM is criticized to be too complex, especially for describing target platforms containing a huge amount of APIs such as JavaEE or .Net.

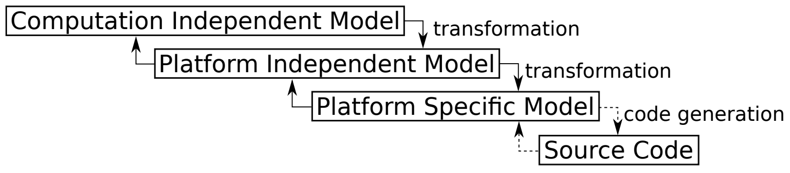

Figure 4.1: MDA Layers[1]

All in all, MDA can be used to create and refactor information systems. As DDD, it utilizes models but unlike DDD, it utilizes meta-models to enable model transformations. These model transformations are used to adapt to new requirements and allow the deployment of the system to different target platforms. For this, MDA requires the PSM which

[1]adapted from [SM07]

describes the target platforms. DDD in contrary is more abstract and its tactical design components (such as entities and value objects) can be deployed in any modern object oriented programing language. As utilizing model transformation, the target platforms supported by MDA are not limited to object oriented programing languages but, therefore, it has an increased complexity.

4.2 Distributed Databases

A *Distributed Database* (DDB) system is a system consisting of multiple, interrelated databases that are not sharing the same memory and that are distributed over a computer network. They have become the dominant data management tool for data-intensive information systems [ÖV96]. Data is distributed over several *data sites* by *fragmenting* and *replicating*. A fragmentation on a relational database scheme can be horizontal by partitioning the table rows using a selection operation or vertical by partitioning the table columns using a projection operation. The advantages of fragmenting the data are, among others, to improve the performance of database systems and to reduce transmission cost by placing the required data in close proximity of its usage. Further, fragmentation can speed up response times by reducing the amount of relations having to be processed in an user query. A replication fragmentation replicates data over multiple *data sites*. This is desirable when the same data is accessed over multiple sites and, therefore, a lower response time can be achieved by duplicating rather than to transferring the data each time [ÖV96; KH10].

For fragmenting horizontally, [KH10] suggested a *Create, Read, Update, and Delete Matrix* (CRUDM) based approach which does not require the frequency of queries, unlike previous horizontal fragmentation techniques. This is beneficial, as the frequency of queries is not available at the initial state of the database creation.

As a partition was required for the DDD approach and the operations of an information system (see Section 5.1.2) can also be subdivided into *Create, Read, Update, and Delete* (CRUD) operations [Fow02], a similar approach to the CRUDM fragmentation has been taken. The approach also utilizes weighted functions partitioning based on CRUD access.

However, the operations are not stored as a matrix but in a graph (see Section 5.8.3). Likewise, as an existing information system is to be architecturally refactored, the access frequency of business operations is available so that additional weighting is possible. As the DDB fragmentation approach, partitioning entities and value objects into aggregates wrongly can also impact performance negatively since business operations having to access more aggregates than necessary for a single operation. This negative performance impact is modeled with negative weighting function called the *Access Frequency Negative Weight Function* (see Section 5.8.3). Finally, like in the DDB fragmentation approach, the best partition is selected maximizing the sum of the weight functions.

5 Domain-Driven Design for an Existing Information System

Chapter 5 describes the conceptual key aspects of the approach (see Figure 5.1). First, this chapter outlines the *Refactoring Process* (see Section 5.1) containing an ubiquitous language and appropriate business operations based on the analysis of the domain model. As shown in Figure 5.1, the information of the ubiquitous language and the business operations are merged to create a *source model* (see Section 5.7) which contains entities, value objects, and modules. The *source model* is defined by a meta-model (see Section 5.2) called the *source meta-model* and contains entities, value objects, modules, and business operations. By utilizing different transformation rules (see Section 5.4), the *source model* can be transformed from the *source meta-model* to a *target meta-model*. The created *target model* can be used as a *source model* for the next transformation. Finally, after one or more transformations, the *final model*, such as the aggregate model (see Section 5.8.1) or the service model (see Section 5.9) is obtained. The process for generating the first model is called artifact-model transformation. The model transformation processes are called model-model transformations.

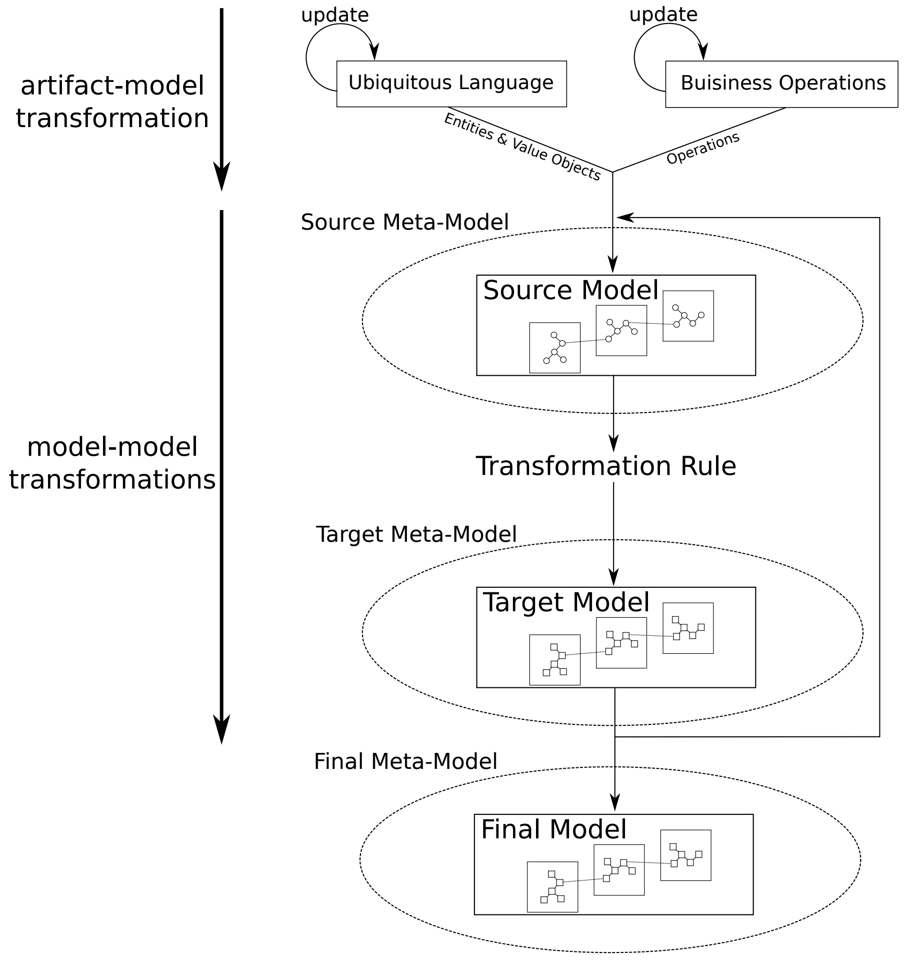

Figure 5.1: Transformation Process

5.1 Refactoring Process

Architectural Refactoring describes the process of changing the infrastructure of an existing system to a new one while reusing information and components of the old architecture if it is beneficial. The goal of Architectural Refactoring is to improve the overall software quality bypassing limitations of the old architecture [Ste16].

The benefit of this approach is clearly that one does not have to write the system completely anew but is able to utilize the old system's structure. However, when designing a Domain-Driven Design based system, one usually creates a new system and utilizes the experience of domain experts for its design. It was decided against a direct transformation as it seldom succeeds in practice because “*most such systems have multiple, intermingled models and/or the teams have disorderly development habits*” [Eva13]. The approach presented in the next chapter also creates a Domain-Driven Design by utilizing information about the business operations to support the refactoring process. The "detour" utilizing business operations for transformation was chosen, as their information can be collected even when dealing with a *Big Ball of Mud* [FY97] scenario.

The goal of this section is to describe the architectural refactoring process towards Domain-Driven Design. This process utilizes the strategic design and the tactical design as suggested by Eric Evans [Eva04]. The process to obtain the strategic design is close to the original process. The tactical design however utilizes automated model transformations incorporating business operations into the design process. After creating the important elements of the tactical design automatically, developers can modify the ubiquitous language and the business operations to see the impact of the change on the architecture.

5.1.1 Ubiquitous Language

The ubiquitous language (see Section 3.2) is the main pillar of DDD. It should contain the terms of the core and subdomains, bounded contexts, and modules. As the bounded contexts serve as a barrier of the ubiquitous language, the language has to be determined for each bounded context. Furthermore, it facilitates the terms of the tactical design (see Section 3.4) and serves as basis for the definition of business operations.

The *glossary* is used to capture the ubiquitous language. For the architectural refactoring process, it is created from the terms used in the old system. In addition, terms used by the people involved in the domain are collected. The benefit of this approach is that it does not alienate the new design. Terms are distilled and improved by finding unique terms and thereby tackling redundancy. Moreover, terms can be changed if they do not fit their use case. This can lead to a resistance of the employees as they are used to old terms and have to adapt. For this, it was found that involving all parties into the creation of the ubiquitous language is important. [Eva13] describes the ubiquitous language as a company wide language . If the ubiquitous language is created locally and only for a small part of the project, it will not gain this required coverage.

MERCAREON, for example, decided to use the ubiquitous language as defined in the glossary as a company wide language used for any type of communication.

As mentioned before, it is important to constantly update the ubiquitous language language whenever a change to the domain model occurs. Changes can thereby range from new customers to new requirements. In some cases, changes may require adjustments to the strategic design (see Section 3.3) but in most cases they require changes to the tactical design (see Section 3.4).

The ubiquitous language is a collection of entries defined in Definition 1:

Definition 1 (Ubiquitous Language)**.**
Let L be the ubiquitous language with $L = \{e_1, ..., e_n\}, n \in \mathbb{N}$
Then let $e = (\textit{term}, \textit{bounded context}, \textit{module}, \textit{Identity}, \textit{Has-a}, \textit{Is-a})$ be an entry of the ubiquitous language (with entry sets starting with a capital letter) where:

- ***term**: Term of the entry. Has to be unique within the bounded context.*
- ***bounded context**: Context to use the word in (see Section 3.3.2).*
- ***module**: Module in the bounded context the word is required for (see Section 3.3.4).*
- ***Identity**: (Possibly empty) Set of entries identifying this entry. The identity is required for separating entities and value objects (see Section 5.1.4).*
- ***Has-a**: (Possibly empty) Set of entries that are part of this entry. Has-a can be annotated with a quantity and is required to detect the root entity (see Section 3.4.1) and its components.*
- ***Is-a**: (Possibly empty) Set of entries that are parent of this entry. Is-A enables inheritance between entries of the language.*

Example 9 exemplifies Definition 1 with the ubiquitous language entry of a *booking*:

Example 9 (Booking entry)**.**
- **term**: booking
- **bounded context**: TSM system
- **module**: Booking

- **Identity**: {company, booking number}
- **Has-a**: {(un)loading date[1], gate[1]}
- **Is-a**: {expected delivery}

5.1.2 Business Operations

Transferring information directly from the old system's architecture to the new was not intended by DDD. By utilizing the business operations, information about the operations that should be supported are added using *Create, Read, Update, Delete, and Input* (CRUDI). In addition, the frequency annotation of each operation conveys the experience gained from the previous system helping to create tactical design DDD models for each bounded context.

Section 6.1.2 exemplifies how the business operations are stored. To gather the initial operations, domain experts need to categorize the bounded context's business operations into Create, Read, Update, Delete, and Input operations for each module. When the ubiquitous language changes, the business operations have to be updated as the business operations are based on the ubiquitous language. Moreover, the business operations are also subject to changes as soon as the requirements of the system have been changed. System maintenance therefore results in constant updates to the glossary and business operations.

The collection of *Business Operations* (BO) comprises operations that were categorized as important for the domain by its domain experts. Though not part of the original DDD concept, the gathering of business operations was introduced to enable a more powerful analysis, e.g. what performance the execution of a method has and if it requires transactional safety. The analysis is then used for transformations determining aggregates (see Section 3.4.3) and services (see Section 3.4.5). The supported operations for each business operations are *Create, Read, Update, and Delete* (CRUD) with the extension of *Input* (CRUDI). The Input extension enables to distinguish between data read by business operations (e.g. by accessing *Repositories* (see Section 3.4.4)) and data passed to the business operation as parameters.

Since "*any of the use cases in an enterprise application are fairly boring CRUD [...] use cases on domain objects*" [Fow02], CRUDI was chosen to support categorizing data elements (entities and value objects) and business operations into aggregates. The *frequency* determines how often a business operation is executed, is required to weight the occurrence of the CRUDI operations and, moreover, determines the performance impact of the business operation.

Definition 2 introduces the different components of the business operations.

Definition 2.
Let BO be the set of business operations with $BO = \{bo_1, ..., bo_n\}, n \in \mathbb{N}$.
Then let $bo = (name, bounded\ context, module, Precondition, Input, Create, Read, Update, Delete, frequency)$ be a business operation where:

- ***name**: unique identifier of the business operation.*
- ***bounded context**: context to use the business operation in (see Section 3.3.2).*
- ***module**: module in the bounded context the term is required for (see Section 3.3.4).*
- ***Precondition**: conditions that have to hold for the business operation to be executed.*
- ***Input**: data elements passed as parameters.*
- ***Create**: business operation that creates a data elements.*
- ***Read**: business operation that reads an existing data element.*
- ***Update**: operation that changes an existing data element.*
- ***Delete**: removes an existing data element.*
- ***frequency**: ranges between "always" and "almost never" and states how often a business operation is executed with: $1 \leq frequency \leq 5, frequency \in \mathbb{N}$*

5.1.3 Strategic Design

Strategic Design proposed in [Eva04] has to be determined once upon designing the system architecture. It was therefore decided to determine this design manually instead of creating an automated solution. The approach of strategic design for architectural refactoring is very similar to the traditional approach (see Section 3.3) and therefore only shortly discussed in the following.

Core and Subdomain

The first step in refactoring information systems is to divide the domain into core and subdomains (see Section 3.3.1). For this, domain experts have to evaluate which parts of the system is crucial to the domain and which parts are not.

This decision is important for the architectural refactoring process since it facilitates the decision of what parts of the old system should be transfered to the new architecture, what parts could stay in the old architecture, and what parts can be completely outsourced.

Bounded Contexts

Bounded contexts (see Section 3.3.2) mainly serve as the boundary for the ubiquitous language. Legacy systems communicating with the new architecture should be encapsulated with bounded contexts. The same is valid for outsourced components. In a perfect scenario, a bounded context should be matched to a single core and subdomain.

Domain experts can create a map of the bounded contexts showing the mapping to core and subdomains and the communication strategy between different contexts. Figure 3.3 shows the map created for the MERCAREON company. An important point to notice when creating a context map is that bounded contexts also hint how to distribute different teams. Therefore, it should be taken into consideration that the distribution of the old system's architecture also influences the team distribution.

As discussed in Section 3.3.3, the bounded contexts, having different ubiquitous language, may require a translation layer. Furthermore, bounded contexts can be encapsulated with the *Ports and Adapters* (see Section 3.4.7) architecture to communicate with databases, third party libraries, and other external systems.

Module

As soon as bounded contexts have been designated, they can be subdivided into smaller logical units—the modules. They are used to reduce complexity and can be created from high level domain concepts. Modules are created by domain experts responsible for the bounded context the modules are located in.

When the old architecture has not yet degraded to a *Big Ball of Mud*, modules may be partly extrapolated. Having familiar modules supports development process as developers can conjecture the modules' functionality.

5.1.4 Tactical Design

Tactical Design is affected by every change to the business operation and its underlying glossary. To face this challenge, it has been decided that the tactical design will be generated automatically using a Java prototype (see Chapter 6). Moreover, the prototype helps in creating the initial and following Domain-Driven Designs by validating the glossary and business operations. In addition, it provides a graphical overview of the tactical design. This can help to evaluate the positive and negative sides of the current design. Moreover, as the prototype can cope with any change to the business operations the involved developers can try out different variants getting immediate feedback how the tactical design is changed.

Entities and Value Objects

The entries in a ubiquitous language that have an identity or that are child of an entity having an identity are entities (see Definition 3 and Section 3.4.1).

Definition 3 (Entity).
Let E be the set of all entities *in the ubiquitous language L,*

$E = \{e_1, ..., e_n \mid hasIdentity(e, L) = 1\}\ k \in \mathbb{N}, k \leq n,$

$$hasIdentity(e, L) = \begin{cases} 1 & \textit{Identity}(e) \neq \emptyset \wedge \textit{Identity}(e) \subset L \\ hasIdentity(\textit{Is-a}(e), L) & \textit{Is-a}(e) \neq \emptyset \wedge \textit{Identity}(e) = \emptyset \\ 0 & (\textit{Identity}(e) = \emptyset \vee \textit{Identity}(e) \not\subseteq L) \\ & \quad \wedge \textit{Is-a}(e) = \emptyset \end{cases}$$

Entries in the ubiquitous language that have neither an identity nor are child of an entity having an identity are value objects (see Definition 4 and Section 3.4.2).

Definition 4 (Value Object).
For the ubiquitous language L and the set of entities E,
$V = \{e_1, ..., e_n \mid hasIdentity(e, L) = 0\}\ k \in \mathbb{N}, k \leq n$ *is the set of all value objects of L.*

Example 10 showcases the difference between entities and value objects on the case of the booking entry:

Example 10 (Booking Entry Entity)**.** The booking entry of Example 9 is an entity because it has the identity of *company* and *booking number*. If it had not, it still might be an entity if one of its parents connected to it via an is-a relation is an entity. In case of the booking, the *expected delivery* is an entity. Otherwise it would be a value object.

Example 11 presents the business operation which is being used to create a reservation in the MERCAREON company. Reservations are created when timeslots are assigned to a *carrier* company.

Example 11 (Create reservation)**.**
The business operation entry used for the *create reservation* operation:

- **name**: create reservation
- **bounded context**: TSM system
- **module**: Schedule
- **Precondition**: $\emptyset$
- **Input**: {gate, date, time, property values, carrier id}

- **Create**: {reservation, transaction log entry}
- **Read**: {schedule}
- **Update**: $\emptyset$
- **Delete**: $\emptyset$
- **frequency**: 2

Aggregates

The aggregate model (see Section 5.8) is created from the source model using automatized model-model transformations by utilizing transformation rules (see Section 5.4). The generated model helps the developers to see object ownership and transactional boundaries. When a change to the source model occurs, it affects the created aggregates and their manual counterparts. Developers are warned if their manual definition becomes outdated and whether they have to review the generated aggregates. Like the source model, the generated visualization supports the developers to get an overview of how a change affects the ownership and boundaries.

Moreover, as the aggregates are obtained from repositories, the aggregate model sheds light on which repositories exist. Using a model-artifact transformation, it is possible to create the method stubs for the required repositories creating the aggregates.

Services

By creating the service model (see Section 5.9) utilizing a model-model transformation, the prototype supports developers to categorize business operations into object methods and services. This categorization is especially important to prevent *anemic domain models*. Anemic domain models [Eva04; Fow03] is an anti-pattern where hardly any behavior resides inside the objects making them "*little more than bags of getters and setters*" [Fow03]. Anemic domain models are especially problematic as they resemble valid domain model implementations. With the prototype creating a visual representation, developers can immediately identify potential anemic designs for each bounded context.

5.2 Defining Modeling Languages using Meta-Model

The best way to define a modeling language is by employing a meta-models. A meta-model, however, is also being defined by a modeling language itself [OMG14].
The goal, as can be seen in Figure 5.1, is to establish *model-to-model* transformations by applying a *transformation specification* consisting of multiple *transformation rules* [OMG10].

5.3 Semantic Network Meta-Model

The concept of a *Semantic Networks* (sNets) model is defined as follows:

> “ *A [model] is a set of nodes. Any sNet node belongs to a [model]. A sNet node cannot belong to many [models]. [Models] bring modularity to the sNets.* ”
>
> – University Nantes, *[Nan98]*

The sNets meta-model (see Figure 5.2) enables to draw models representing semantic relationships between concepts [Nan98]. Within a sNets model, a model is create out of several nodes. As shown in Figure 5.3, a node has a type which is a subtype of the node *Root*. This is illustrated by a *meta relation* between a node and its root node. This relationship represents an *is-a* relationship that is to be distinguished from the business operation's is-a relationship (see Definition 2). Furthermore a node has a name represented by a character sequence. Each node is part of a *model* (originally *universe*). Each *model* is described by a *meta-model* (originally *Semantic Universe*). *Root* nodes are part of the *meta-model* which is connected to the *model* of the root's child node by a semantic relation (*sem*). The *meta-model* in sNets is a subtype of a *model* consisting of *Node* and *Link* Nodes. Each *model* **must** be linked to its *meta-model* by a *sem* link. Since a *model* is also a *Root* node in sNets nesting of models is also possible (see Figure 5.2) [CST00; Lem98a].

The sNets models were extended to support weighted edges required for modeling *Has-a* relations and business operations' *access frequencies*: a *meta-model* in sNets can contain

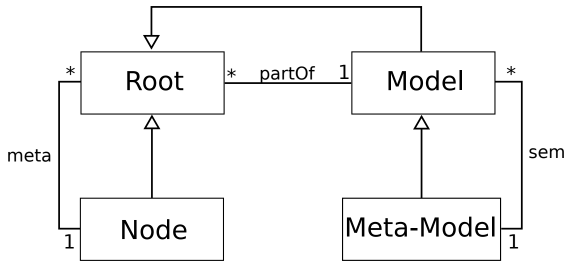

Figure 5.2: sNets UML Representation[1]

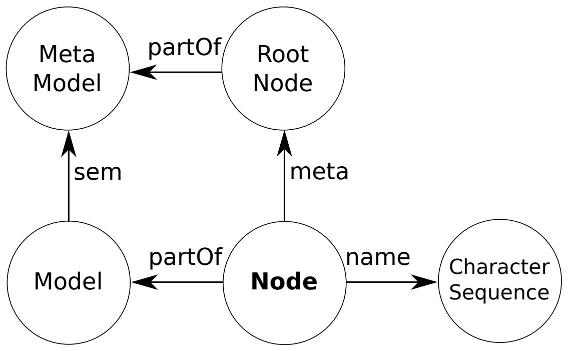

Figure 5.3: sNets Node[2]

*Link** Nodes which represent a weighted link. The weight may be any numeric number, + for one or more, and * for zero or more.

To create sNets models more easily, [Lem98b] proposed a simplified notation as shown in Figure 5.4. In this notation, the *model* and *meta-model* are represented by squares with rounded edges. The *model* has a *sem* relation to its *meta-model*. Nodes are displayed as halved circles exposing the *type* on the top and the *name* on the bottom part. The *partOf* link is modeled by the node being drawn inside of a model. It is of importance that this is only another graphical representation and, even though not in the figure, the *name* and *partOf* links still exist and they therefore can be accessed by transformation rules (see Section 5.4).

Example 12 shows how a booking entry of the TSM system is represented in sNets.

Example 12 (Booking entry)**.** Figure 5.5 is a simplified sNets representation of Example 9. It shows how entities and value objects are defined and also how the definition of edges

[1]adapted from [CST00]
[2]adapted from [Lem98b]

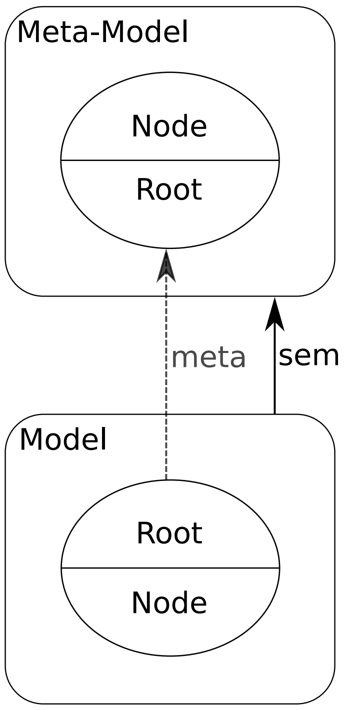

Figure 5.4: Simplified sNets Notation[3]

between them are modeled. Additionally, it exemplifies how models are being nested by nesting the Booking module into the TSM system context. Moreover, the example is a simplified version of reality since it contains entities without a matching identifier.

[3]adapted from [Lem98b]

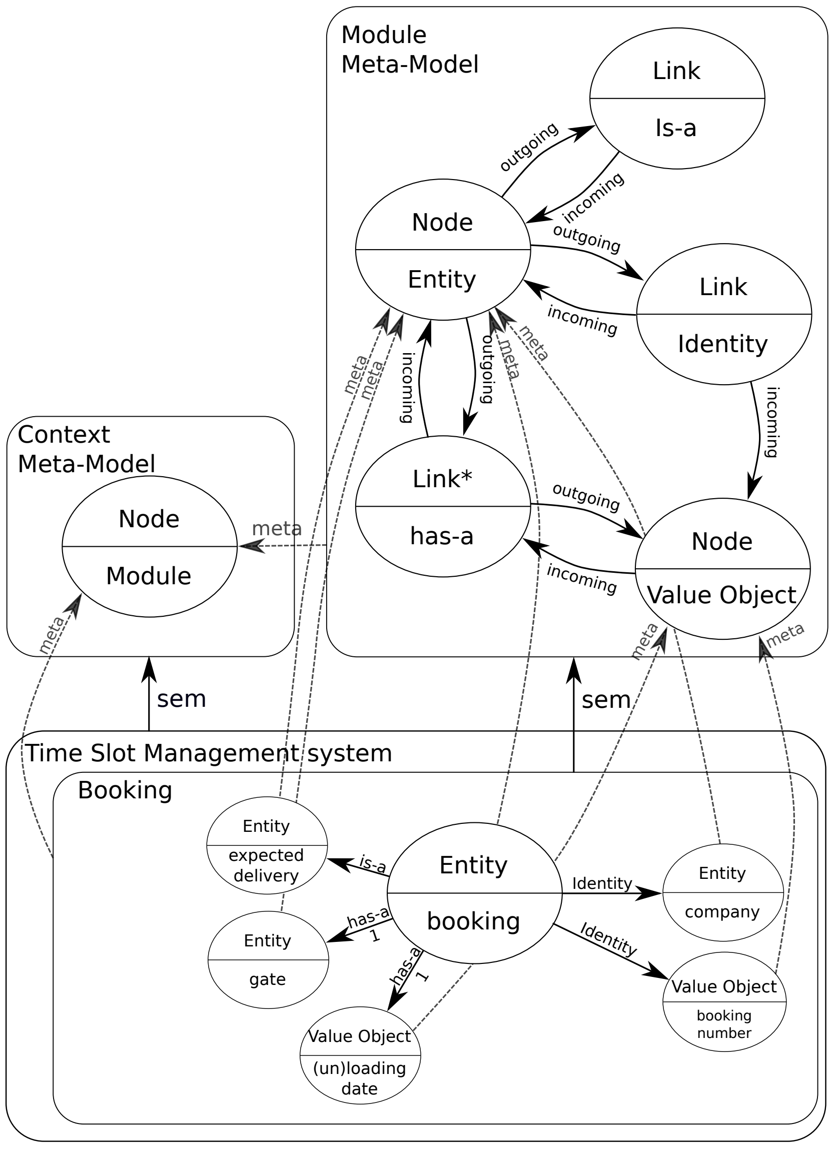

Figure 5.5: Time Slot Management System sNets Example

5.4 Transformation Rules

As [Lem98b] explains, a transformation between models of two different meta-models can be achieved by applying a set of rules.

The transformation rules are defined in *EBNF* [Lem98b]. Firstly, every transformation rule contains the source and target *meta-model*. Secondly, a transformation rule contains a set of statements. The latter is executed until no statement is applicable any more and, therefore, the transformation is finished. A statement can have multiple variables bound to either nodes or edges. Statements can not be applied multiple times with the same bounded variable whereby each bounded variable can not be bound to an already bound node or edge. The transformation rules's *EBNF* definition was adapted to fit the needs of transforming DDD models.

The *EBNF* transformation rule is defined as follows:

⟨*TransformationRule*⟩ ::= ⟨*Header*⟩ ":"{⟨*Statement*⟩};

⟨*Header*⟩ ::= ⟨*SourceMetaModel*⟩ → ⟨*TargetMetaModel*⟩;

⟨*Statement*⟩ ::= ["Priority " ⟨*Priority*⟩ ":"] ⟨*Conditions*⟩ → ⟨*Conclusions*⟩ ";" ;

⟨*Conditions*⟩ ::= ("("⟨*Conditions*⟩ ")") | (⟨*Condition*⟩ {("∧"|"∨") ⟨*Conditions*⟩});

⟨*Conclusions*⟩ ::= ("("⟨*Conclusion*⟩ ")") | (⟨*Conclusion*⟩ {"∧"⟨*Conclusion*⟩});

As defined in the *EBNF*, transformation rules have a $\langle Header \rangle$ and a set of $\langle Statements \rangle$. The following Section 5.5 explains the header of a transformation rule. Thereafter Section 5.6 discusses the different $\langle Conditions \rangle$ and $\langle Conclusions \rangle$ available in a $\langle Statement \rangle$.

5.5 Transformation Rule – Header

The *Header* of a transformation rule describes the *source meta-model* and the *target meta-model* for which the statements of the transformation rule are applicable [Lem98b]. A model can be transfered from the *source meta-model* to a *target meta-model* by applying the statements defined by the transformation rule.

5.6 Transformation Rule – Statement

A $\langle Statement \rangle$ defines a set of positive or negative conditions (see Section 5.6.1) that, if satisfied, will result in the fulfillment of the conclusion (see Section 5.6.2) which creates a set of new nodes and links [Lem98b].

Definition 5 defines the sNets model mathematically and is used for defining the different conditions and conclusions available in the transformation rule's statement.

Definition 5 (Semantic Network)**.**

- $G \in \{\mathit{SourceMetaModel}, \mathit{SourceModel}, \mathit{TargetMetaModel}, \mathit{TargetModel}\}$
- *$V(G)$ are vertices of the graph G*
- *$E(G)$ are edges of the graph G*
- *Each edge $e \in E(G)$ has a type(e) and each node $n \in V(G)$ has a type(n).*
- *$V_{\mathit{VT}}(G) := \{v \in V(G) \,|\, \mathit{type}(v) = VT\}$ are the vertices of the type VT. The set of VT is defined according to the meta-model.*
- *$E_{\mathit{ET}}(G) := \{e \in E(G) \,|\, \mathit{type}(e) = ET\}$ are the edges of the type ET. The set of ET is defined according to the meta-model.*
- *Each node $n \in V(G)$ has a character sequence name(e).*
- *Each edge $e \in E_{\mathit{Link}*}(G)$ has a natural number $w(e)$, called its weight.*

Note: In the definition, the sNets model is called a graph G. sNets nodes are called vertices while sNets links are called edges.

Priority (Extension)

The $\langle Priority \rangle$ is used to give the statements of a transformation rule an execution order. The priority is a natural number. Statements with a higher priority will be executed before the ones with a lower priority. Statements with the same priority will be executed in a random order. If no $\langle Priority \rangle$ is specified, the statement can be executed randomly and, therefore, behaves like the statements that were suggested by [Lem98b].

5.6.1 Statement – Conditions

Formally, a *condition* constraints the type of a variable or requires a link between two variables. The conditions have been extended to support weighted links and arithmetic formulas.

A condition, therefore, is defined as follows:

⟨*Condition*⟩ ::= ⟨*TypeConstraint*⟩
| ⟨*LinkConstraint*⟩
| ⟨*TempLink*⟩
| ⟨*WeightedLinkConstraint*⟩
| ⟨*Formula*⟩

The following Sections introduce the different available constraints of a *statement condition*. For this, the negation extension is firstly presented as each constraint also has a negative variant.

Negation (Extension)

Especially with the arithmetic formulars and the temporary links, there is the need for negation. The negation is expressed with a logical negation (¬) symbol. It allows more powerful statements such as, for example, filter for all nodes of the source model without a link to a certain different node in the source model.

Type Constraint

The $\langle TypeConstraint\rangle$ are used in statements to filter a variable for nodes of a specific node type.

A $\langle TypeConstraint\rangle$ has the following form:

⟨*TypeConstraint*⟩ ::= ["¬"] ["target_"]⟨*Type*⟩ "("⟨*Node*⟩")"

In case the $\langle TypeConstraint\rangle$ condition does not start with "target_", the condition is true iff the node of the *source model* specified by $\langle Node\rangle$ has a meta link to a node with the type $\langle Type \rangle$ in the *source meta-model*.

This is equivalent to:

$$\forall \textit{Node} \in V(\text{SourceModel}) \, \exists \, \text{t} \in V(\text{SourceMetaModel}) : \\ (\text{Node}, \text{t}) \in \{E_{\text{meta}}(G) \mid name(t) = Type\}$$

In case the $\langle TypeConstraint \rangle$ condition starts with "target_", the condition is true iff the node of the *target model* specified by $\langle Node \rangle$ has a meta link to a node with the type $\langle Type \rangle$ in the *target meta-model*.

This is equivalent to:

$$\forall \textit{Node} \in V(\text{TargetModel}) \, \exists \, \text{t} \in V(\text{TargetMetaModel}) : \\ (\text{Node}, \text{t}) \in \{E_{\text{meta}}(G) \mid name(t) = Type\}$$

In turn, the negated variant is true iff the node has no meta link to a node with the name $\langle Type \rangle$ in the respective models:

$$\forall \textit{Node} \, \nexists \, \text{t} : (\text{Node}, \text{t}) \in \{E_{\text{meta}}(G) \mid name(t) = Type\}$$

Link Constraint

$\langle \textit{LinkConstraint} \rangle$ ::= ["¬"] ["target_"]$\langle \textit{LinkType} \rangle$ "(" $\langle \textit{FromNode} \rangle$ "," $\langle \textit{ToNode} \rangle$ ")"

In case the $\langle LinkConstraint \rangle$ condition does not start with "target_", the condition is true iff there are links with the type $\langle LinkType \rangle$ between the nodes $\langle FromNode \rangle$ and $\langle ToNode \rangle$ of the *source model*.

This is equivalent to:

$$\forall \, \text{FromNode} \in V(\text{SourceModel}) \, \forall \, \text{ToNode} \in V(\text{SourceModel}) : \\ (\text{FromNode}, \text{ToNode}) \in E_{\text{LinkType}}(G)$$

In case the $\langle LinkConstraint \rangle$ condition does start with "target_", the condition is true iff there are links with the type $\langle LinkType \rangle$ between the nodes $\langle FromNode \rangle$ and $\langle TargetNode \rangle$ of the *target model*.

This is equivalent to:

$$\forall\, \text{FromNode} \in V(\text{TargetModel})\ \forall\, \text{ToNode} \in V(\text{TargetModel}) : (\text{FromNode}, \text{ToNode}) \in E_{\text{LinkType}}(G)$$

The negated variant is true, iff no link of the type $\langle LinkType \rangle$ exist between $\langle FromNode \rangle$ and $\langle ToNode \rangle$ in the respective *models*:

$$\forall\, \text{FromNode}, \text{ToNode}\ :\ (\text{FromNode}, \text{ToNode}) \notin E_{\text{LinkType}}(G)$$

Temporary Link Constraint

Temporary links start with an underscore (_) character in order to distinguish them from normal links. The temporary link constraint is true iff there are temporary links of the type $\langle LinkType \rangle$ between the nodes $\langle FromNode \rangle$ and $\langle ToNode \rangle$.

Temporary links are defined as follows:

⟨*TempLink*⟩ ::= ["¬"] "_" ⟨*LinkType*⟩ "(" ⟨*FromNode*⟩ "," ⟨*ToNode*⟩ ")"

The formal definition is equivalent to the definition of normal link constraints (see Section 5.6.1 – Link Constraint). Temporary links are used to associate created nodes in the *target model* with nodes out of the *source model*. For example, when creating a node of the type *table* in the *target model* from a node in the *source model*. In order to add new nodes and connect them to the table node for each node connected to the node of the *source model*, a temporary link between these two nodes is required. Without the temporary link, the creation of the connected node in the *target model* would prove difficult as the association to the node in the *source model* is lost. Moreover, temporary links can be used to indicate the previous execution of a statement to statements executed later on.

As the name already indicates, temporary links are removed after the execution of the transformation rule concluding a model transformation.

Weighted Link Constraint (Extension)

To support the addition of weighted links (*Link**), the following constraint was added.

⟨*WeightedLinkConstraint*⟩ ::= ["¬"] ⟨*LinkType*⟩ "("⟨*FromNode*⟩ "," ⟨*ToNode*⟩ "," ⟨*Weight*⟩ ")"

The condition is true iff there are links of the type $\langle LinkType \rangle$ between the nodes $\langle FromNode \rangle$ and $\langle ToNode \rangle$. In addition, $\langle Weight \rangle$ has to match the link weight.

This is equivalent to:

$$\begin{aligned}&\forall\, \text{FromNode} \in V(\text{SourceModel})\ \forall\, \text{ToNode} \in V(\text{SourceModel})\ \text{let}\\ &\qquad x := (\text{FromNode}, \text{ToNode}) \in E_{\text{LinkType}}(G)\ \text{where}\\ &\qquad \exists\, y \in \{V_{\text{Link*}}(\text{SourceMetaModel}) \mid \text{type}(x) = \text{name}(y)\}\ \text{and}\\ &\qquad \text{weight}(x) = Weight\end{aligned}$$

The negated variant is true, iff there are no links at all or only links with a different weight between *FromNode* and *ToNode*:

$$\begin{aligned}&\forall\, \text{FromNode} \in V(\text{SourceModel})\ \forall\, \text{ToNode} \in V(\text{SourceModel})\ \text{let}\\ &\qquad x := (\text{FromNode}, \text{ToNode}) \in E_{\text{LinkType}}(G)\ \text{where}\\ &\qquad \nexists\, y \in \{V_{\text{Link*}}(\text{SourceMetaModel}) \mid \text{type}(x) = \text{name}(y)\}\ \text{or}\\ &\qquad \text{weight}(x) \neq Weight\end{aligned}$$

Temporary Weighted Link Constraint (Extension)

The *temporary weighted link constraint* start with an underscore (_) character in order to distinguish them from normal weighted links. It behaves like the temporary link (see Section 5.6.1 – Temporary Link Constraint) with the addition of a link weight (see Section 5.6.1 – Weighted Link Constraint (Extension)). Therefore, the temporary weighted link is also removed when the transformation is finished.

⟨*WeightedLinkConstraint*⟩ ::= ["¬"] "_"⟨*LinkType*⟩ "("⟨*FromNode*⟩ "," ⟨*ToNode*⟩ "," ⟨*Weight*⟩ ")"

Functions (Extension)

One drawback of the sNets formalism is that there is no option to constrain the transformation based on sophisticated mathematical formulas. For example, one only transforms if the income of the employee is greater than 1000 Euro. Since the aggregate detection is based on heuristics, it was necessary to add a way of performing calculation into transformation rules. Moreover, helper functions could be defined to traverse the graph and, for example to get the root node of a tree-based structure.

A $\langle Function\rangle$ condition starts with a hash (#) character and is expressed as follows:

$\langle$*Function*$\rangle$ ::= ["¬"] "#"$\langle$*FunctionName*$\rangle$ "("$\langle$*Parameter*$\rangle$ ("," $\langle$*Parameter*$\rangle$) * ")"

Let Parameter$_i \in V(SourceModel) \cup E(SourceModel), i \in \mathbb{N}$. The $\langle Function\rangle$ condition is true iff the result of the function is greater than zero for all parameter permutations:

$\forall\, \mathrm{p}_1 \in \mathrm{Parameter}_1 \ldots \forall\, \mathrm{p}_n \in \mathrm{Parameter}_n : f_{\text{FunctionName}}(\mathrm{p}_1, \ldots \mathrm{p}_n) > 0$

The negated variant is true iff the result of the function is less or equal than zero:

$\forall \mathrm{p}_1 \in \mathrm{Parameter}_1 \ldots \forall \mathrm{p}_n \in \mathrm{Parameter}_n : f_{\text{FunctionName}}(\mathrm{p}_1, \ldots \mathrm{p}_n) \leq 0$

5.6.2 Statement – Conclusions

A *conclusion* may be the creation of new nodes or the creation of a (temporary) link between two bound variables. A conclusion, therefore, is defined as follows:

$\langle$*Conclusion*$\rangle$ ::= $\langle$*NodeCreation*$\rangle$
| $\langle$*LinkCreation*$\rangle$
| $\langle$*WeightedLinkCreation*$\rangle$
| $\langle$*TempLinkCreation*$\rangle$

Node Creation

Nodes are created in the *target model* by using a $\langle NodeCreation\rangle$ conclusion.

$\langle$*NodeCreation*$\rangle$ ::= $\langle$*TypeName*$\rangle$ "("$\langle$*Variable*$\rangle$")"

The created node has the name specified by $\langle Variable\rangle$ and is of the type $\langle TypeName\rangle$. Therefore, it has a *meta* link to the node of the corresponding type in the *meta-model*.

This is equivalent to:

$$\exists\,\text{Node} \in V(\text{TargetModel}), \text{t} \in V(\text{TargetMetaModel}) : \\ (\text{Node}, \text{t}) \in \{E_{\text{meta}}(G) \mid name(t) = Type\}$$

Link Creation

Links are created between two nodes of the *target model* by using a $\langle LinkCreation\rangle$ conclusion.

⟨LinkCreation⟩ ::= *⟨LinkType⟩* "(" *⟨SourceVariable⟩* "," *⟨TargetVariable⟩*")"

The $\langle LinkCreation\rangle$ conclusion creates a link of the type $\langle LinkType\rangle$ between the nodes $\langle FromNode\rangle$ and $\langle ToNode\rangle$.
This is equivalent to:

$$\exists\,\text{FromNode} \in V(\text{TargetModel})\exists\,\text{ToNode} \in V(\text{TargetModel}) : \\ (\text{FromNode}, \text{ToNode}) \in E_{\text{LinkType}}(G)$$

Weighted Link Creation (Extension)

As the extended sNets meta-models support weighted links (*Link**), the creation of weighted links between two nodes of the *target model* was added to the statement's conclusions.

⟨WeightedLinkCreation⟩ ::= *⟨LinkType⟩* "("*⟨SourceVariable⟩* "," *⟨TargetVariable⟩* ","
(*⟨Weight⟩* ["+"*⟨Weight2⟩*]) | (*⟨Function⟩*) ")"

The $\langle WeightedLinkCreation\rangle$ conclusion creates a weighted link of the type $\langle LinkType\rangle$ with the weight $\langle Weight\rangle$ between the nodes $\langle SourceVariable\rangle$ and $\langle TargetVariable\rangle$. $\langle Weight2\rangle$ is set to zero if not specified. Alternatively the result of a $\langle Function\rangle$ can be used to set the weight of the link.

This is equivalent to:

$$\begin{aligned}&\exists\,\text{FromNode} \in V(\text{TargetModel})\ \exists\,\text{ToNode} \in V(\text{TargetModel})\text{ with}\\ &\qquad x := (\text{FromNode}, \text{ToNode}) \in E_{\text{LinkType}}(G)\text{ where}\\ &\qquad \exists\, y \in \{V_{\text{Link}^*}(\text{SourceMetaModel}) \mid \text{type}(x) = \text{name}(y)\}\text{ and}\\ &\qquad ((\text{weight}(x) = Weight + Weight2)\text{ or}\\ &\qquad (\text{weight}(x) = f_{\text{FunctionName}}(p_1, ..., p_n))\end{aligned}$$

Temporary Link Creation

Temporary links can be like normal links created between two nodes. The difference between temporary links and a normal link creation is, that temporary links can be created in the source model, the target model or even spanning both the source and the target models (see Section 5.6.1 – Temporary Link Constraint).

⟨*TempLinkCreation*⟩ ::= "_" ⟨*LinkType*⟩ "(" ⟨*SourceVariable*⟩ "," ⟨*TargetVariable*⟩ ")"

Temporary Weighted Link Creation

The temporary weighted link creation is similar to a normal weighted link creation (see Section 5.6.2 – Weighted Link Creation (Extension)). The difference between temporary links and a normal link creation is, that temporary links can be created in the source model, the target model or even spanning both the source and the target models (see Section 5.6.1 – Temporary Weighted Link Constraint (Extension)).

⟨*TempWeightedLinkCreation*⟩ ::= "_"⟨*LinkType*⟩ "("⟨*SourceVariable*⟩ "," ⟨*TargetVariable*⟩ ","
⟨*Weight*⟩ ["+"⟨*Weight2*⟩] ")"

5.6.3 Examples

The following examples showcase the *EBNF* for different condition and the conclusion.

Example 13 (Statement Conditions).

- Entity(e) is true for all nodes of type “Entity”
- Entity(e) $\wedge$ Identity(e,v) $\wedge$ Entity(v) is true for all “Entity” nodes connected with an “Identity” edge to a node of the type “Entity”.

Example 14 (Statement Conclusions)**.**

- Entity(e) $\wedge$ name(e,"booking") creates a node of type “Entity” with the name of “booking”.
- has-a(a,b) creates a has-a link between the nodes a and b.

5.7 Source Model

The DDD source model is directly converted using an artifact-model transformation (see Figure 5.1) on the ubiquitous language and the business operations.

As required by the aggregate and service determination, the initial *source meta-model* can be seen in Figure 5.6 and contains entities and value objects with their relations and categorization by modules. They are being accessed by business operations also belonging to the module. The *source model* is created for each bounded context of the DDD adhering to the source meta-model. Besides, as explained in Section 5.1.2, the business operations accesses (represented as edges of the corresponding types in the source model) the entities and value objects with one of the *Create*, *Read*, *Update*, *Delete*, and *Input* operations.

The goal of the source model is not only to be able to determine aggregates and services, but also to graphically validate the ubiquitous language and the business operations. To create this model, an *artifact-model transformation* [OMG10] is performed to transform both the information of the ubiquitous language and the business operations to the source model. Moreover, as ranging from the distribution in modules to evaluating the bounded context borders, the source model also helps to evaluate strategic design decisions.

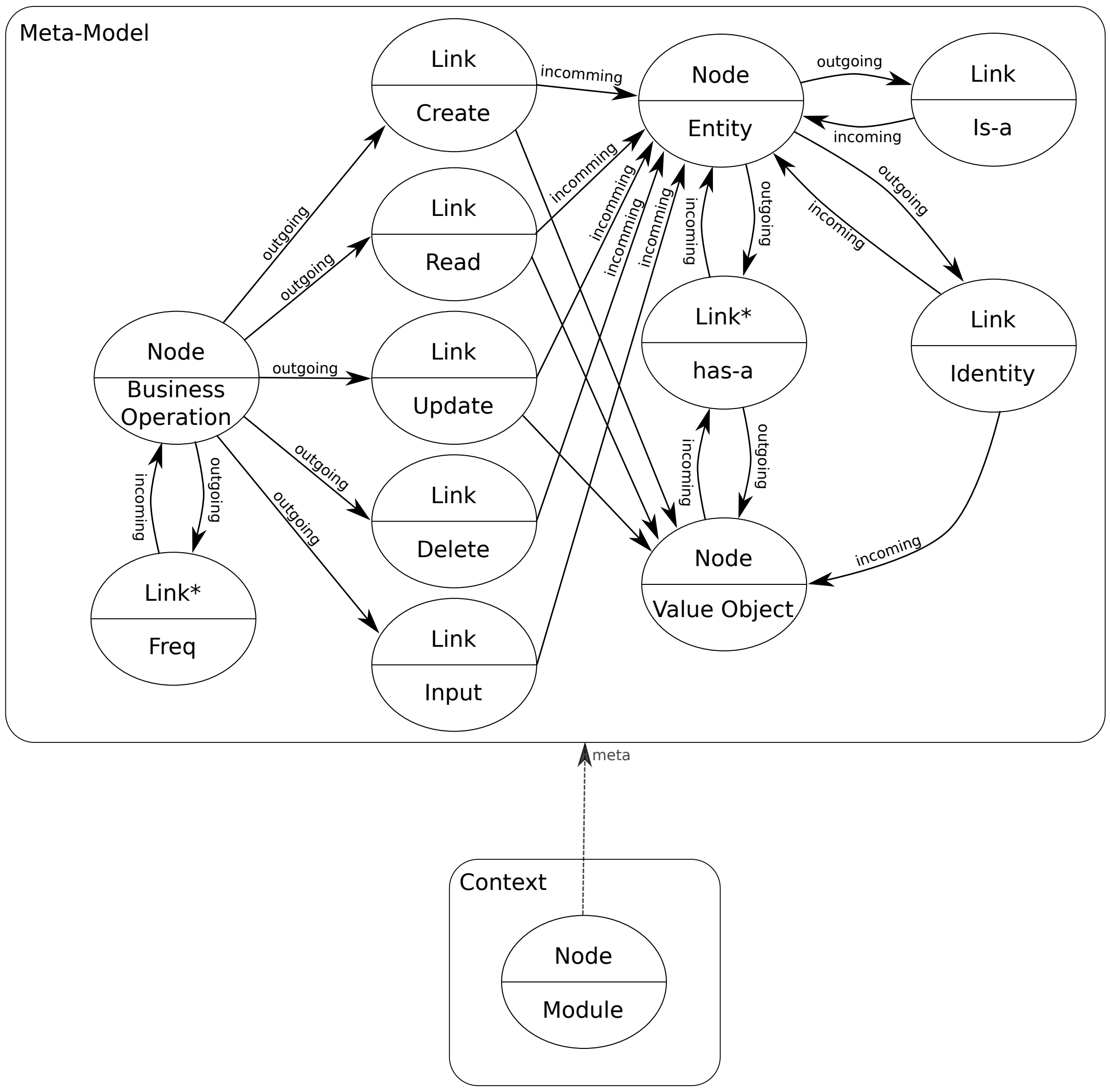

Figure 5.6: Source Meta-Model

5.8 Aggregate Models

Aggregates are being determined by two *model-model transformation* steps (see Figure 5.1). First, a *source model* is generated. This model is then transformed into the potential aggregates *target model* containing all candidates for aggregates by the appliance of a transformation rule. Then, by reusing the created *target model* as a *source model* for the next transformation rule, the potential aggregate ***source** model* is transformed into the *final aggregate model* containing the actual aggregates. First, the three models (see Section 5.8.1) and then the transformation rules (see Section 5.8.2) are presented.

5.8.1 Models

This section describes the different models involved in the transformation in order to finally determine aggregates.

Potential Aggregate Model

The potential aggregate model (see Figure 5.7 for the meta-model) contains all candidates applying for being an aggregate in the final *aggregate model* grouped by potential aggregate models. In this model several potential aggregate nodes can contain the same entities having the same value objects. Since sNets only support one node being in one model at a time, entities and value objects being in different potential aggregates are represented by different nodes consequently. Furthermore, the is-a relation of the *source model* is replaced by inheriting all has-a relations from the parent.

For example, if a booking is a delivery, and the delivery has a delivery date, the booking inherits this date. The business operations are left out of the model as being part of a service,an entity, or a value object which are being analyzed by the *Service Models* (see Section 5.9).

Potential aggregates are transformed from the *source model*. The first part is transformed by creating the transitive closure over the has-a relations in the *source model* and then walking through every node adding it and its children to the model.

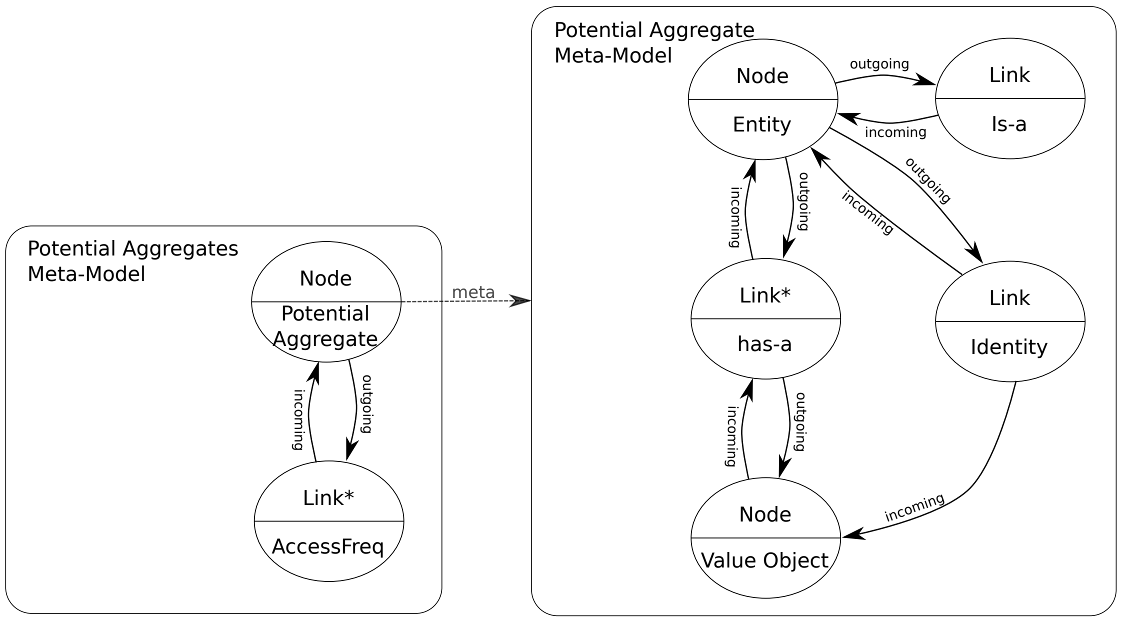

Figure 5.7: Potential Aggregate Meta-Model

In the second part, the nodes accessed together by business operations in the *source model* are also added to the potential aggregates since it is likely that they share the same transactional boundary.

Final Aggregate Model

The *aggregate model* (see Figure 5.8 for the meta-model) contains the selected aggregates from the *potential aggregate model*. It is obtained through the application of transformation rules on the *potential aggregate model* reducing the set of potential aggregates until all entities occur once in the *aggregate model*. This results in *potential aggregates* being left out since one of their entities was part of a *potential aggregate* that was chosen before by a transformation rule to be part of the *aggregate model*. The main difference to the *potential aggregate meta-model* is the absence of the frequency link* which was only required for transformation (see Section 5.8.3)

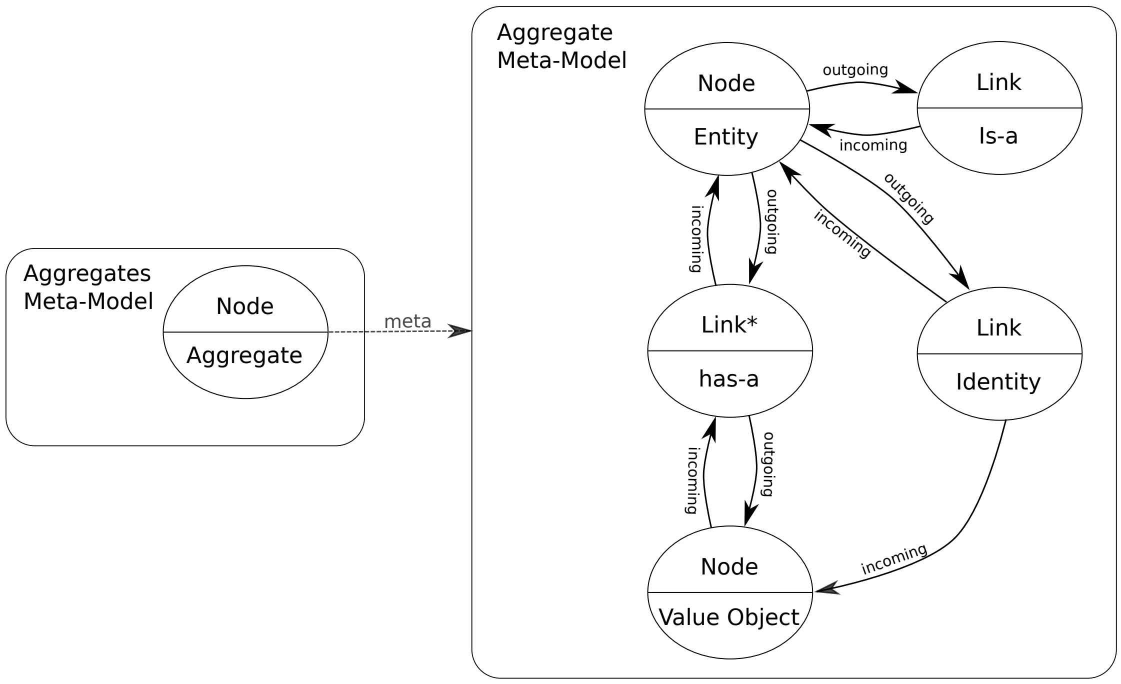

Figure 5.8: Aggregate Meta-Model

5.8.2 Aggregate Transformation Rules

This section describes the two step transformation of the *source model* to the *potential aggregate model* and then from the latter to the final *aggregate* model.

From Source Model to Potential Aggregate Model

The first two statements create a transitive closure in the source model over the has-a edges using temporary has-a edges. The first statement, named $\langle tempHasA \rangle$, creates temporary has-a edges for each has-a edge and has the highest priority (see Section 5.6 – Priority (Extension)). This step simplifies the other statements because as temporary has-a edges are created later on, the statements don't need to distinguish between normal and temporary edges.

The second statement, $\langle transitiveClosure \rangle$ creates a temporary edge A$\rightarrow$ C if an (temporary) edge exists from between A $\rightarrow$ B and from B $\rightarrow$ C, therefore, after all statements are executed, a transitive closure exists in the graph.

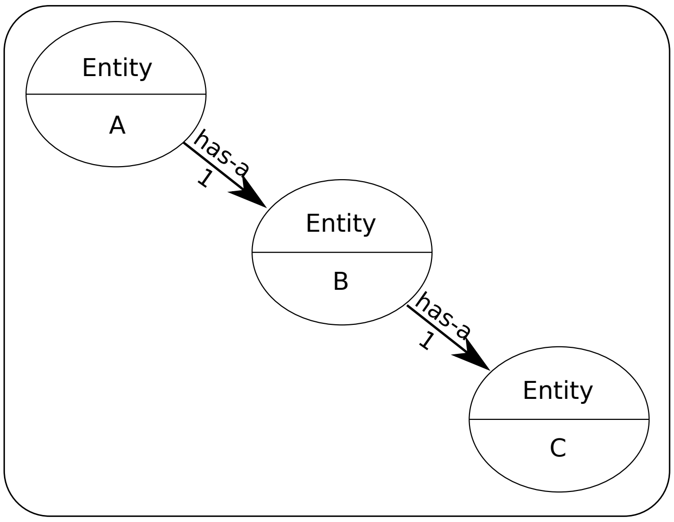

Figure 5.9: Has-a Constellation

For creating the potential aggregate entity permutations, intermediate steps of the transformation to a transitive closure are required. Therefore, the $\langle newPotAggregate\rangle$ and $\langle hasA1\rangle$ statements have a higher priority than the $\langle transitiveClosure\rangle$ statement. This results in copying the nodes from source to the target model as soon as a new temporary has-a edge has been created. The $\langle newPotAggregate\rangle$ statement creates the new *potential aggregate* with only one entity. Then, the $\langle hasA1\rangle$ statement copies all entities connected to this entity. This approach ensures to have all has-a permutations in the model, from a single has-a node to its has-a childrens and their children.

In Figure 5.9, the approach would result in the combinations $\{\{A\},\{A,B\},\{A,B,C\},\{B\},\{B,C\},\{C\}\}$. Lastly, the *hasA2* statement creates all regular has-a links spanning *potential aggregate* boundaries. These are required for the *External HasA Function* (see Section 5.8.3).

⟨*tempHasA*⟩ ::= Priority 4 : Entity(e) $\wedge$ has-a(e,v) $\rightarrow$ _has-a(e,v);

⟨*transitiveClosure*⟩ ::= Priority 1 :
    Entity(e) $\wedge$ _has-a(e,v) $\wedge$ _has-a(v,w) $\rightarrow$ _has-a(e,w);

⟨*newPotAggregate*⟩ ::= Priority 2 :
    Entity(e) $\wedge$ name(e,eName) $\wedge$ Entity(v) $\wedge$ _has-a(e,v)

$\rightarrow$ Enitiy(root) $\wedge$ name(root,eName) $\wedge$ PotentialAggregate(p) $\wedge$ partOf(root,p) $\wedge$ _createdFrom(e,root);

⟨*hasA1*⟩ ::= Priority 3 : _source(e,v) $\wedge$ _has-a(e,n) $\wedge$ name(n,nName) $\wedge$target_partOf(v,p) $\rightarrow$ Enitiy(new) $\wedge$ name(new,nName) $\wedge$ has-a(v,new) $\wedge$ partOf(new,p) $\wedge$ _source(n,new);

⟨*hasA2*⟩ ::= Priority 1 : _source(e,v) $\wedge$ _source(f,w) $\wedge$ has-a(e,f) $\rightarrow$ has-a(v,new);

The $\langle move\rangle$ statement accounts for the business operations and adds *potential aggregates* for multiple entities being modified or accessed by business operations. Note that with these statements, duplicate *potential aggregates* could be created. This does not lead to a later aggregate multiplication because if one of the similar *potential aggregates* is chosen in the transformation to the *aggregate model*, the duplicates are marked with a temporary edge and, therefore, ignored by statements executed later on.

⟨*move*⟩ ::= Priority 1 : BusinessOperation(bo) $\wedge$ Entity(e) $\wedge$ (Create(bo,e) $\vee$ Read(bo,e) $\vee$ Update(bo,e) $\vee$ Delete(bo,e) $\vee$ Input(bo,e)) $\wedge$ name(e,eName) $\rightarrow$ Enitiy(newE) $\wedge$ name(newE,eName) $\wedge$ PotentialAggregate(p) $\wedge$ partOf(new, p) $\wedge$ _source(v,new);

Last, the $\langle accessFreq\rangle$ statement provides information how often an entity is accessed by a business operation is required by the *Access Frequency Positive Weight Function* (see Section 5.8.3). The information is stored in a weighted edge on the *potential aggregate* containing the sum of all frequencies of business operations that access all entities of the *potential aggregate*.

⟨*accessFreq*⟩ ::= Priority 1 :
$\neg$target_Entity(te) $\wedge$ $\neg$target_partOf(te,p) $\wedge$ target_PotentialAggregate(p) $\wedge$ BusinessOperation(bo) $\wedge$ ($\neg$Create(bo,e) $\wedge$ $\neg$Read(bo,e) $\wedge$$\neg$Update(bo,e) $\wedge$ $\neg$Delete(bo,e) $\wedge$ $\neg$Input(bo,e)) $\wedge$ _source(se,te) $\wedge$ freq(bo,bo,f) $\wedge$ target_accessFrequency(p,p,fOld)
$\rightarrow$ target_accessFrequency(p,p,fOld+f);

The $\langle accessFreq\rangle$ statement can be read as follows: When there is no entity in the target model that is part of the potential aggregate p that is not being CRUDI accessed by the business operation b, then add the freq value of the business operation b to the accessFrequency of the potential aggregate p.

From Potential Aggregate Model to Final Aggregate Model

First of all, weights for *potential aggregates* are being calculated by the $\langle weight\rangle$ statement to be able to determine the order in which potential aggregates are transformed to aggregates. Therefore an edge with the weight of the *weightPotentialAggregate* function (see Section 5.8.3) is created, self referencing the *potential aggreagte*. Note that although the statement has not the highest priority, it is executed before the other statements. This is because the other statements can not be executed before the *stage* statement is executed.

⟨*weight*⟩ ::= Priority 2 : PotentialAggregate(e) $\rightarrow$ _weight(e,e,#$f_{weightPotentialAggregate}$(e));

After weighting all *potential aggregates* (ensured by a lower priority than the weight statement), the aggregate which has not been finished yet and has the highest weight is determined and staged for transfer to a newly created aggregate of the *final aggregate model* by the $\langle stage\rangle$ statement.

⟨*stage*⟩ ::= Priority 1 : PotentialAggregate(e) $\wedge$ PotentialAggregate(v) $\wedge$ _weight(e,e,we)
$\wedge$ $\neg$_weight(v,v,wv) $\wedge$ #f_{higher}(wv, we) $\wedge$ $\neg$ _finished(e,e)
$\rightarrow$ Aggregate(a) $\wedge$_stage(e,a);

The $\langle stage\rangle$ statement can be read as follows: There is no potential aggregate v with a weight higher than the weight of the potential aggregate e.
The function f_{higher} is thereby defined as follows:

$$f_{higher}(a,b) = \begin{cases} 1 & a > b \\ 0 & otherwise \end{cases}$$

When a potential aggregate has been staged for transfer, all *potential aggregates* containing an entity node created from the same entity of the source model are marked as finished.

The selection of the same node is based on the nodes' names. Hence the names have has to be unique in DDD within a single-bounded context. For this, the $\langle markDuplicate\rangle$ statement finds and marks duplicate entity names.

⟨*markDuplicates*⟩ ::= Priority 3 : Entity(e) $\wedge$ _stage(p,a) $\wedge$ partOf(e,p) $\wedge$ Entity(v) $\wedge$
$\neg$ partOf(e,v) $\wedge$ name(e,aName) $\wedge$ name(v,aName) $\wedge$ PotentialAggregate(p2) $\wedge$
partOf(v,p2)
$\rightarrow$ _finished(p2,p2);

With the same priority, the entities are copied by the $\langle moveEntities\rangle$ statement from the *potential aggregate model* to the *final aggregate model*. Additionally, a *createdOf* temporary link is created helping to identify which entity was created by which entity in the source model.

⟨*moveEntities*⟩ ::= Priority 3 : Entity(e) $\wedge$ _stage(p,a) $\wedge$ partOf(e,p) $\wedge$ name(e,eName)
$\rightarrow$ Entity(new) $\wedge$ name(new,eName) $\wedge$ partOf(e,a) $\wedge$ _createdOf(new,e);

After entities have been copied to the *final aggregate model*, the has-a relations have to be transfered as well. The first statement copies the value object for each created entity and creates a has-a relation between the entities and their value objects. The second statement creates has-a relations between entities of the same aggregate.

⟨*moveValueObjects*⟩ ::= Priority 2 :
_createdOf(new,e) $\wedge$ partOf(e,a) $\wedge$ has-a(e,v) $\wedge$ ValueObject(v) $\wedge$ name(v,vName)
$\rightarrow$ ValueObject(newV) $\wedge$ name(newV,vName) $\wedge$ partOf(v,a) $\wedge$ has-a(e,v);

⟨*restoreEntityHasA*⟩ ::= Priority 1 :
_createdOf(ae1,pe1) $\wedge$ has-a(pe1,pe2) $\wedge$_createdOf(ae2,pe2)
$\rightarrow$ has-a (ae1,ae2);

5.8.3 Weight Potential Aggregate Function

The $f_{weightPotentialAggregate}$(e) function calculates the sum of six distinct weight functions returning positive or negative infinity, or a value in the range of $-4, ..., 0, ..., +4$. Zero is returned when the no decision can be made based on the evaluated *potential aggregate*.

Minus infinity is returned when it is absolute certain that the evaluated potential aggregate can not be an aggregate and overrules other decimal values. Lastly, plus infinity is returned when it is absolute certain that the potential aggregate is an actual aggregate also overruling other decimal values.

Prefer Single Entity Cluster Function

The *prefer single entity cluster function* rates aggregates based on the involved amount of entity clusters in an potential aggregate. An entity cluster is a set of entities where one entity is the root entity of the cluster reaching all other cluster entities over a set of has-a links. Entity clusters can overlap, but they may not contain each other's root entity. The *prefer single entity cluster function* returns a fixed positive number iff only one cluster exists and a negative number for potential aggregates with more than one entity cluster. This reflects that aggregates with multiple clusters can not be completely obtained using a single repository access.

Access Frequency Positive Weight Function

The *access frequency positive weight function* weights by summing up the times entities of the potential aggregate were accessed by business operations. This is due to the fact that entities which are often accessed together, are more likely to be part of the same transaction and, therefore, belong to the same aggregate.

Access Frequency Negative Weight Function

In contrast to the positive version, the *access frequency negative weight function* calculates a negative weight based on different business operations changing entities or value objects of the evaluated *potential aggregate* while also changing different *potential aggregates*. The *access frequency negative weight function* was created as these operations would infringe the statement not to modify multiple aggregate at a time (see Section 3.4.3). Moreover, the function acts as a counterweight to the above-mentioned positive version.

HasA Weight Function

The *HasA weight function* traverses has-a relationships of entities and multiplies their weight. The higher the weight, the lower the function's result. The purpose of this function arises from the downside of having high has-a relationship counts in an aggregate. The high has-a relationships results in many entities having to be loaded when retrieving the aggregate from its repository. In addition, the transactional boundary is increased as more entities have to be locked. The performance of the aggregate therefore decreases with every entity [Vau13].

Prefer Small Aggregate Function

The *prefer small aggregate function* returns a positive result, if the aggregate contains only few entities and, in turn, a negative result, if the aggregate contains many entities. This function can be regarded as an extension to the above mentioned *HasA weight function* which is only based on the weight between entities and not on the amount of different entities.

External HasA Function

If the *potential aggregate* has an entity with no incoming has-a edges, but every other entity can be reached from this entity, it is considered as the *aggregate's root entity* (see Section 3.4.3). If the root aggregate's root entity can't be determined or is not referenced from the outside, the *external HasA function* will return zero. It will return a positive number if entities from outside of the aggregate have has-a edges to the root entity and a negative number if they reference entities other than the aggregate root.

5.9 Service Models

Figure 5.10 shows the final service meta-model for the model-model transformation (see Section 5.4) from the source meta-model. Unlike the *Aggregate Models* (see Section 5.8), determination of the service model can be achieved in one single model-model transformation.

As business operations can reside as services outside, or as object methods inside of entities, the meta-model allows for nesting of business operations into entities.

Comparing the source meta-model to the service meta-model, the main difference, aside from entities having object methods, is that, unlike the business operations, services only have a read and write link leading to a more simple diagram. Moreover, the transformation rules are shorter as they have not to deal with each of the CRUDI links.

5.9.1 Service Transformation Rules

As the service meta-model and the source meta-model are highly comparable, the first set of statements copy entities, value objects, and their relations from the source to the service meta-model.

⟨*moveEntities*⟩ ::= Priority 12 : Entity(e) $\wedge$ name(e,eName)
    $\rightarrow$ Entity(new) $\wedge$ name(new,eName) $\wedge$ _createdOf(new,e);

⟨*moveValueObjects*⟩ ::= Priority 11 :
    _createdOf(new,e) $\wedge$ has-a(e,v) $\wedge$ ValueObject(v) $\wedge$ name(v,vName)
    $\rightarrow$ ValueObject(newV) $\wedge$ name(newV,vName) $\wedge$ has-a(e,v);

⟨*restoreEntityHasA*⟩ ::= Priority 10 : _createdOf(target_e1,source_e1)
    $\wedge$ has-a(source_e1,source_e2) $\wedge$ _createdOf(target_e2,source_e2)
    $\rightarrow$ has-a (target_e1,target_e2);

As soon as the copying procedure is complete, the statements for creating the services and object methods are executed.

The first statement, namely $\langle createService1\rangle$, creates a service for business operations that only deletes entities as these business operations require repositories (which are not accessible from the object methods). The second statement, $\langle createService2\rangle$, handles business operations creating entities. The third statement, as can be seen in $\langle createService3\rangle$, handles business operations updating two or more entities.

⟨*createService1*⟩ ::= Priority 9 : BusinessOperation(bo) ∧ ¬_createdOf(bo,_) ∧
 name(bo,boName) ∧ delete(bo,_) ∧ ¬create(bo,_) ¬update(bo,_)
 → Service(newService) ∧ name(newService,boName) ∧ _createdOf(newService,bo);

⟨*createService2*⟩ ::= Priority 8 : BusinessOperation(bo) ∧ ¬_createdOf(bo,_) ∧
 name(bo,boName) ∧ create(bo,_) ∧ ¬delete(bo,_) ¬update(bo,_)
 → Service(newService) ∧ name(newService,boName) ∧ _createdOf(newService,bo);

⟨*createService3*⟩ ::= Priority 7 : BusinessOperation(bo) ∧ ¬_createdOf(bo,_) ∧
 name(bo,boName) ∧ update(bo,x) ∧ update(bo,y) ∧ ¬delete(bo,_) ¬create(bo,_)
 → Service(newService) ∧ name(newService,boName) ∧ _createdOf(newService,bo);

Next comes the $\langle createObjectMethod1\rangle$ statement which creates object methods. Firstly, business operations which only read exactly one entity are transformed to an object method of the entity as they only require the entities information. This statement utilizes the enforcement that multiple variables may be bound to the same element at a time (see Section 5.4).

⟨*createObjectMethod1*⟩ ::= Priority 6 : BusinessOperation(bo) ∧ ¬_createdOf(bo,_)
 ∧ name(bo,boName) ∧ read(bo,source_e) ∧ ¬read(bo,v) ∧ ¬update(bo,_)
 ∧ _createdOf(target_e,source_e) ∧ ¬update(bo,_) ∧ ¬delete(bo,_) ¬create(bo,_)
 → ObjectMethod(newMethod) ∧ name(newMethod,boName) ∧
 part-of(newMethod,target_e) ∧ _createdOf(newMethod,bo);

The other way around, business operations reading multiple entities are represented as a service because object methods have no access to repositories.

⟨*createObjectMethod2*⟩ ::= Priority 5 : BusinessOperation(bo) ∧ ¬_createdOf(bo,_)
 ∧ name(bo,boName) ∧ read(bo,source_e) ∧ read(bo,v) ∧ ¬update(bo,_) ∧
 _createdOf(target_e,source_e) ∧ ¬update(bo,_) ∧ ¬delete(bo,_) ¬create(bo,_)
 → Service(newService) ∧ name(newService,boName) ∧
 part-of(newService,target_e) ∧ _createdOf(newService,bo);

The following $\langle createObjectMethod3\rangle$ statement covers business operations updating exactly one entity. The statements utilizes the fact that updates, of two or more elements have already been processed by statements with a higher priority.

⟨*createObjectMethod3*⟩ ::= Priority 4 : BusinessOperation(bo) ∧ ¬_createdOf(bo,_)
  ∧ name(bo,boName) ∧ update(bo,source_e) ∧ _createdOf(target_e,source_e) ∧
  ¬delete(bo,_) ∧ ¬create(bo,_)
  → ObjectMethod(newService) ∧ name(newService, boName) ∧
  part-of(newService,target_e) ∧ _createdOf(newService,bo);

A special case are methods updating exactly one entity, but also modifying others. These methods are created as object methods but also require support of an external service. Therefore, they are modeled as a separate set of object methods called *supported object methods*. The following statement shows how they are created.

⟨*supportedObjectMethods*⟩ ::= Priority 3 : BusinessOperation(bo) ∧ ¬_createdOf(bo,_)
  ∧ name(bo,boName) ∧ update(bo,source_e) ∧ _createdOf(target_e,source_e)
  → SupportedObjectMethod(newService) ∧ name(newService,boName)
  ∧ part-of(newService,target_e) ∧ _createdOf(newService,bo);

Next, the $\langle createTempLinks \rangle$ is used to cover business operations modifying multiple entities (e.g., *creating* an entity and *updating* another). This type of business operations are either services utilizing *Domain Events* (see Section 3.4.6) or parts of the same aggregate. In order to prevent the listing of permutations, the $\langle createTempLinks \rangle$ statement creates temporary links (see Section 5.6.1 – Temporary Link Constraint) for *create*, *update*, and *delete* operations. The $\langle createMulti \rangle$ statement is then executed for business operations with more than two temporary links.

⟨*createTempLinks*⟩ ::= Priority 2 :
  BusinessOperation(bo) ∧ (create(bo,e) ∨ update(bo,e) ∨ delete(bo,e))
  → _writeLink(bo,e);

⟨*createMulti*⟩ ::= Priority 2 : BusinessOperation(bo) ∧ _writeLink(bo,a) ∧_writeLink(bo,b)
  → Multi(mult) ∧ name(mult,boName) ∧ _createdOf(mult,bo);

Business operations which do not match any of the previous statements, are transformed into uncategorized nodes. The low priority ensures all other categorization statements are executed before this statement comes into play.

$\langle$*createUncategorized*$\rangle$ ::= Priority 1 :
    BusinessOperation(bo) $\wedge$ $\neg$_createdOf(bo,_) $\wedge$ name(bo,boName)
    $\rightarrow$ Uncategorized(uncat) $\wedge$ name(uncat,boName) $\wedge$ _createdOf(uncat,bo);

Last but not least, after all operations and services are determined, read and write access operators are inserted.

$\langle$*createReadLinks*$\rangle$ ::= Priority 0 : BusinessOperation(bo) $\wedge$ _createdOf(service,bo) $\wedge$
    (read(bo,e) $\vee$ input(bo,e)) $\wedge$ Entity(source_e) $\wedge$ _createdOf(target_e, source_e)
    $\rightarrow$ read(target_e,service);

$\langle$*createWriteLinks*$\rangle$ ::= Priority 0 : BusinessOperation(bo) $\wedge$ _createdOf(service,bo) $\wedge$
    (write(bo,e) $\vee$ delete(bo,e)) $\wedge$ Entity(source_e) $\wedge$ _createdOf(target_e, source_e)
    $\rightarrow$ write(service,target_e);

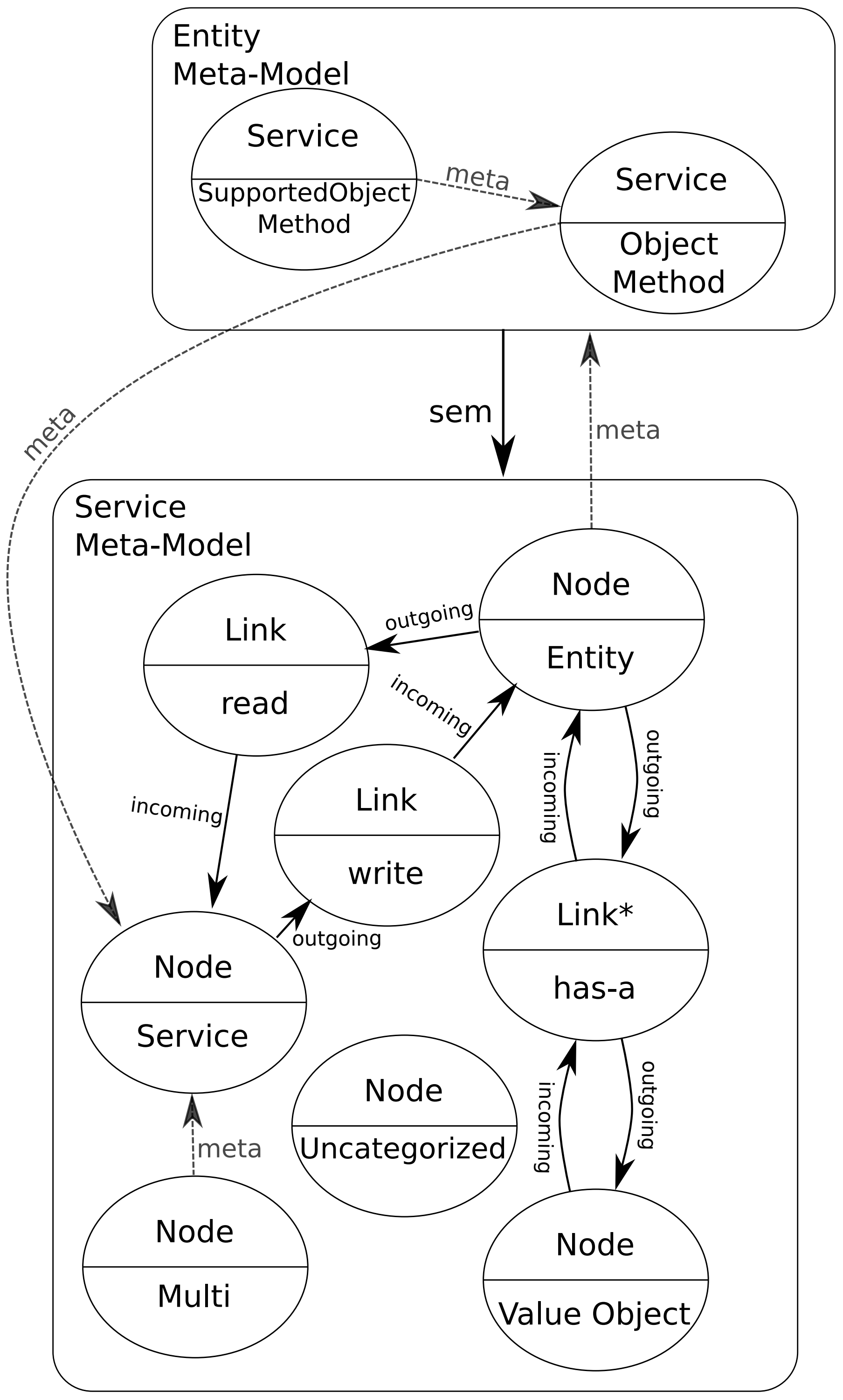

Figure 5.10: Final Service Meta-Model

6 Prototype

In Figure 6.1, the proof-of-concept is introduced implementing the artifact-model, model-model, and model-artifact transformations. To begin with, the structure of the artifacts (namely the business operations and the glossary) used in the first transformation is explained in Section 6.1.

Utilizing these artifacts, the artifact-model transformation (see Section 6.2) creates the *source model*, a tactical DDD model with modules containing entities, value objects, and their relations. In the java implementation, the DDD model is stored through an object graph. By analyzing this model (see Section 6.2.4), the additional information of access frequency to the DDD data types is gathered and added to the model.

As described in Section 6.3, multiple model-model transformations are then applied. This transformations translate the source model created by the artifact-model transformation to the aggregate model (see Section 6.3.1) and the service model (see Section 6.3.6).

Using the created models, the model-artifact transformation can be executed (see Section 6.4). This last transformation step transfers the different models to artifacts such as *exports for media wiki* (see Section 6.4.2), different visualizations of the model (see Section 6.4.1), and additionally, generates source code by utilizing a template engine(see Section 6.4.3).

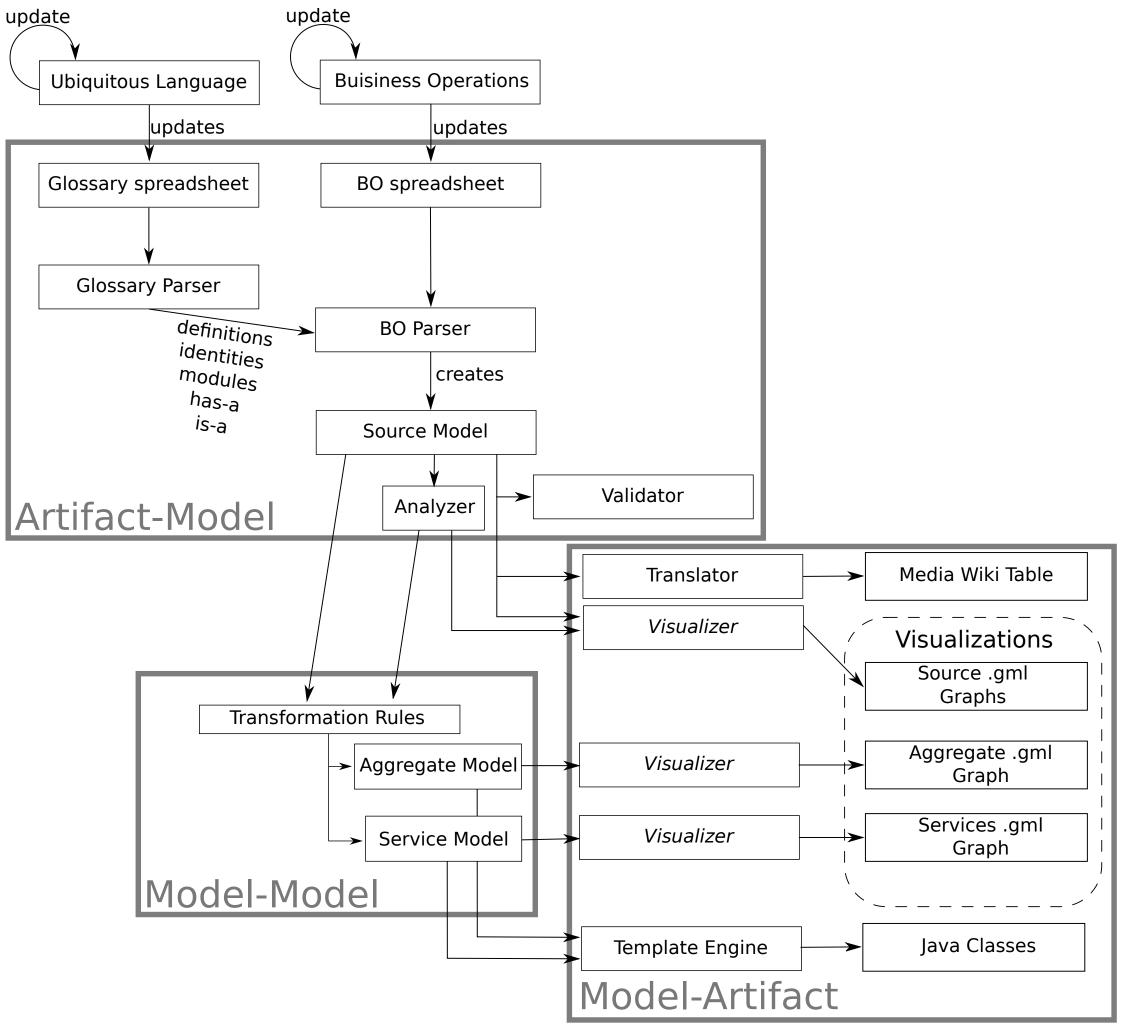

Figure 6.1: Prototype's Architecture

6.1 Artifacts

“*An artifact is a piece of information that is used or produced by a system development process, or by deployment and operation of a system.*”

– OMG Architecture Board, *[OMG10]*

In context of the prototype, artifacts are information stored in different formats such as xlsx (Excel spreadsheet), text, gml, or log messages. They serve either as an input for the prototype or are created by the it as an output.

The prototype receives two artifacts as input, the *Glossary* (see Section 6.1.1) and the *Business Operations* (see Section 6.1.2) containing the ubiquitous language and the business operations. After multiple transformation steps the input is finally converted to different types of output artifacts (see Section 6.4). These artifacts then represent a distilled knowledge representations of the input artifacts.

6.1.1 Glossary

The glossary is an artifact used to grasp the *Ubiquitous Language* (see Sections 3.2 and 5.1.1) as suggested in [Eva04]. At first, only the significant terms, their definitions, and deprecated alternatives were collected and stored in a simple *text* document. While the glossary had been growing, multiple problems with this solution were identified: firstly, the glossary has to be consistent in itself. For example, when having the word "gate" defined, all usages of the concepts of gates must refer to this definition. This led to the problem that changes applied to one term in the document would require that the complete document is edited manually. Secondly, as the used text processors have the tendency to insert special characters (e.g. non breakable spaces, different hyphen characters,...), parsing the document proved challenging. Lastly, as the number of terms grew, the document became increasingly complicated and confusing.

To cope with this, a $\LaTeX$ (LaTeX) document was evaluated as words could be defined in macros. This provided the ability to reuse them in descriptions of other terms. In addition, if properly used, it allows to refactor terms easily always providing a document-wide consistency. By using the *hyperref* LaTeX package, links from terms to their definitions can be automatically integrated in the resulting PDF document. Furthermore, LaTeX text processors usually does not include special characters in the document. As LaTeX stores its information in text files, it is easily parsed with both traditional and modern programming languages, it is easily deployable on multiple target platforms, and it can be naturally shared using version control systems.

However, there is a problem with using LaTeX for MERCAREON. As MERCAREON relies on Microsoft Office^{TM} products and the LaTeX toolchain is not installed on the personal computers by default, the usage of LaTeX for a company-wide language definition proved difficult. Furthermore, as LaTeX has a complex syntax, it is more difficult to write for the involved domain experts. This would endanger its required company wide acceptance.

Therefore, spreadsheets based on Microsoft Excel^{TM} has been selected as a more company compliant document type. With the spreadsheets' tabular layout, it offers a better readable representation of the glossary. The downside is, that as spreadsheets were not made for text representations, cross references were hard to accomplish, and even harder to maintain.

In total, three spreadsheets were used for each bounded context where each spreadsheet contains one data table. The first spreadsheet contains a summary of the other two spreadsheets created by a macro[1]. The second sheet can partly be seen in Example 15 and is specified in Definition 1. It contains terms, the module of the terms, optional identifiers, descriptions, has-a relations, and is-a relations. The third table contains deprecated terms, their modules, and their descriptions.

Example 15 (Excerpt of the second spreadsheet)**.**

Term	Module	Identity	Has-a	Is-a
gate	Location	gate group, name	booking cond{*}, schedule cond{*}	-
user	Authentication and Authorization	company, login	company{1}, role{*}	-
activated order	Order	order number, properties, activation number	-	imported order

6.1.2 Business Operations

Based on Definition 2, the business operation spreadsheet contains the following columns: the name of the operation, the operation's module, the precondition for executing the

[1]The macro firstly copies the complete second spreadsheet and then adds an alternative terms column for each term of the second spreadsheet. This additional column contains the alternative words gathered from the third spreadsheet.

operation, the execution frequency between zero and five, and the column for each CRUDI operation respectively. Example 16 shows an excerpt of the business operation spreadsheet. The business operation spreadsheet is defined for each bounded context.

Example 16 (Excerpt of the business operation spreadsheet)**.**

Operations' Module: Order

Name	Freq	Input	Reads	Creates	Updates	Deletes
activate order	4	activation criteria	imported order	activated order	-	-
delete activated order	2	activated order	-	-	-	activated order

6.2 Artifact-Model Transformation

The *artifact-model transformation* (see Figure 6.2) creates the source model, represented by a Java object graph, from the previously mentioned spreadsheets (see Section 6.1).

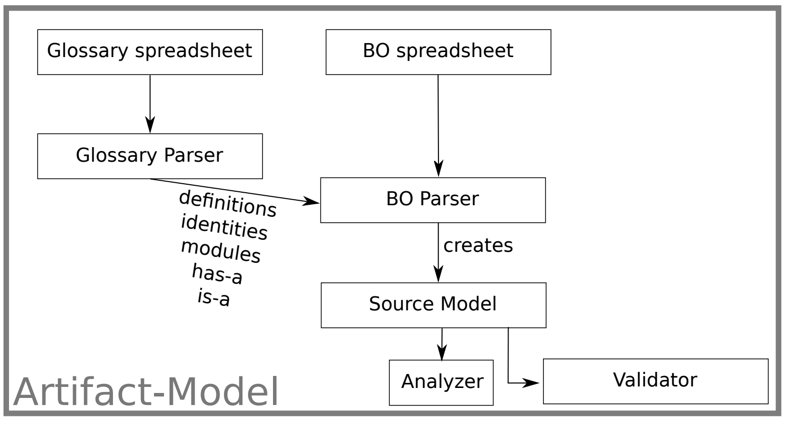

Figure 6.2: Artifact-Model Transformation

The transformation is performed in two steps: First, *Glossary Spreadsheets Parser* (see Section 6.2.1) parses the glossary entries into a Java data-structure. Then, the *Business Operation Parser* (see Section 6.2.2) is executed creating the DDD-based object graph and thereby constructs *Modules* (see Section 3.3.4), *Entities* (see Section 3.4.1), *Value Objects* (see Section 3.4.2), and their relations to each other. Additionally, the graph contains the *Business Operations* (see Section 5.1.2) which are not specified by the DDD models but required for performing *Model-Model Transformations* (see Section 6.3). As this transformation requires meta-models, the object graph created from the artifact-model transformation adheres to the source meta-model as defined in Section 5.7.

6.2.1 Glossary Spreadsheets Parser

To enable the artifact-model transformation, the glossary spreadsheet has to be parsed into a format suited for this task. Therefore, first, the spreadsheets' tables were parsed row by row creating a Hashmap containing the column's name mapping to the column's data. This map is then passed to the constructor of the *GlossaryEntry* class (see Listing 6.1) creating an instance with matching columns from the map. As some columns in the spreadsheet are not mandatory, they might be set to *null*. The *identity* which is required to identify entities, is not used to identify glossary objects as value objects have no such information. Instead, the combination of module and term was chosen for the object identity. As the prototype is meant to be executed on every bounded context seperately, the *GlossaryEntry* class has no bounded context affiliation entry. Last, the *deprecatedTerms* map is filled to enable the export of deprecated terms into other formats such as *Media Wiki* (see Section 6.4.2).

```
public class GlossaryEntry implements
   HasID<ElementIdentifier> {
        @NotNull private final String module;
        @NotNull private final String term;
        @Nullable private String identity;
        @Nullable private final String description;
        @NotNull private final String[] hasA;
        @NotNull private final String[] isA;
        @NotNull private Map<String,DeprecatedTerm>
           deprecatedTerms;
        @Override
        public ElementIdentifier getID() {
                return ElementIdentifier.of(module, term);
        }
}
```

Listing 6.1: Excerpt of GlossaryEntry Class

6.2.2 Business Operation Parser

The module parser uses *GlossaryEntry* instances created by the *Glossary Spreadsheets Parser* (see Section 6.2.1) to create entities and value objects connected by *isA* and *hasA* relations. For this, the business operations's CRUDI accesses of data elements, and recursively their *hasA* and *isA* relations, are traversed to create the respective entities or value objects. The decision whether a data element is regarded as an entity or a value object, is based on the *identify* column (value objects have no unique identities).

Listing 6.3 shows the super class `MethodImpl` of entities and value objects. The implementation differs significantly from the entity and value object representations used in the *source code artifact* created by the code generation unit (see Section 6.4.3). This difference arises as the implementation in the prototype is meant to represent different nodes in a graph where each node needs to be differentiated and therefore has an *node id*. Value objects, as used later in artifacts, don't require such an *node id* as being part of an entity.

Next, the module's business operations are created using the *Builder Pattern* [Gam95] adding their CRUDI data accesses to entities and value objects. As can be seen in Listing 6.2, the class representing the business operations holds: a reference to the business operation's *module*, a list of edges encapsulating the access to entities and value objects providing the information of the access type, the *frequency* of how often the business operation is executed, and the *name* of the operation. The *precondition* field is missing in the example since it was not required for the later performed model-models transformations.

```java
class MethodImpl implements Method {
        @NotNull private final Module module;
        @NotNull private final List<Edge> accessMap;
        @NotNull private final AccessFrequency frequency;
        @NotNull private final String name;
}
```

Listing 6.2: Excerpt Class Representing a Business Operation

```java
public class DataElementImpl implements DataElement {
        @NotNull protected ElementIdentifier id;
        @NotNull protected String name;
        @NotNull protected Map<DataElement, HasAFreq> hasA;
        @NotNull protected Set<DataElement> hasAChildOf;

        @NotNull protected Set<DataElement> isA;
        @NotNull protected Set<DataElement> isAParentOf;
}
```

Listing 6.3: Excerpt of the Entities' and Value objects' Superclass

6.2.3 Artifact Validation

The validation of the artifacts is performed in two distinct steps. At first, while parsing the data to an object model (see Section 6.2), the validity of relationships (isA, hasA) is checked. Since the business operations are based on the glossary, it is also checked whether every operation is based on an actual glossary entry.

As soon as the source model has been created, it is piped through additional validation steps. This second validation is based on the *Decorator Pattern* [Gam95]. Enabling programmers to extend validation routines as required. The simple validation program at the core of the pattern checks whether the parser has created a valid output so that all mandatory fields are set. This validator is then decorated with more sophisticated validators checking if there are data elements that are being written, but not read, and if there are is-A cycles in the object graph.

Listing 6.4 shows the source code of the cycle detection decorator. Has-A relationship cycles are detected and the user of the prototype is warned as circular "*are tricky to maintain*" [Eva04].

```java
class NoCycleValidator extends ValidatorDecorator {
public NoCycleValidator(ModuleValidator parent) {
  super(parent);
}
@Override
public boolean validate(Set<Module> modules) {
  boolean parentResult = super.validate(modules);
  if (!parentResult)
    return false; // skip if previous validation failed
  // iterate all existing data elements (entities or value
     objects) searching for cycles
  boolean cycle=false;
  for(Module module:modules){
    for (DataElement dataElement :
       module.getDataElements(DataElementType.ALL)) {
      cycle = cycle || cycleDetection(dataElement);
    }
  }
  return !cycle;
}
// ...
}
```

Listing 6.4: Has-A Cycle Detection

6.2.4 Analyzer

After the documents have been successfully transfered into a DDD object graph (see Section 6.2) and passed the *Artifact Validation* (see Section 6.2.3), the graph is analyzed. Entities and value objects are tagged with the access frequency metric telling how often they are being accessed. This information is important for the later detection of aggregates. Aggregates, which contain many frequently accessed elements, perform worse from a transactional point of view than aggregates that are accessed less frequently. Furthermore, as this information is also visualized (see Section 6.4.1), it can be used to determine which data elements are important for the domain.

6.3 Model-Model Transformations

The model-model transformations (see Figure 6.3) utilize meta-models and transformation rules (see Chapter 5). The goal is to transform from the source model created by the artifact-model transformation (see Section 6.2) to an aggregate model containing the domain models aggregates (see Section 3.4.3) and a service model containing the services (see Section 3.4.5).

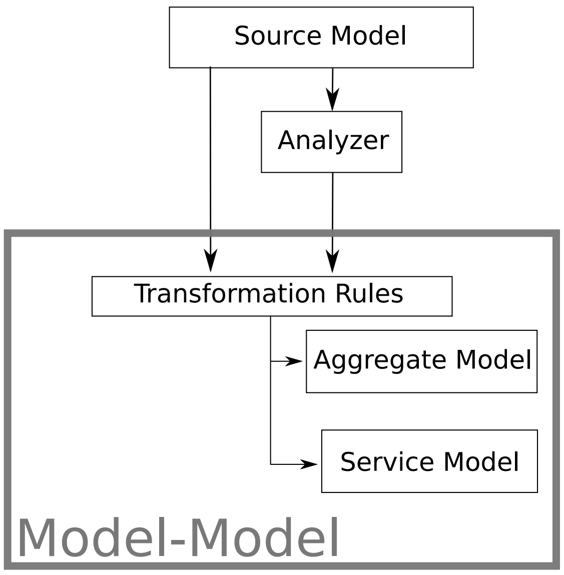

Figure 6.3: Model-Model Transformation

6.3.1 Aggregates

The aggregate determination algorithm is based on the model-model transformation as defined in *Aggregate Transformation Rules* (see Section 5.8.2). As described, aggregates are determined in two transformation steps: firstly, potential aggregates are generated (see Section 6.3.2) and weighted (see Section 6.3.3). Secondly, based on the weighting, a set of potential aggregates is selected and converted to aggregates (see Section 6.3.4).

Section 6.4.1 shows the result of the model-model transformation as exported from yED—a Java based graph visualization tool.

6.3.2 Generating Potential Aggregates

The first step described by the *Aggregate Transformation Rules* (see Section 5.8.2) is to transform from the source model to a *potential aggregates* (see Section 5.8.1) object model. During the determination process, the *potential aggregates* are collected in a map with the contained entities *HashSet* as key and the *PotentialAggregate* instance as value. The *HashSet*, has a specialized implementation calculating the *hash code* only once on creation and not every time accessed. As the map invoke the *hash code* method multiple times, the proposed *HashSet* implementation is faster as it has not to iterate all contained entities for each access.

As described in Section 5.8.1, the potential aggregates determination algorithm consists of several different steps.

To summarize: firstly, has-a relations are inherited over the is-a relations. Then, the root entity (see Sections 3.4.3 and 5.8.3) is determined in the module boundary. Starting from the root entity, the transitive closure over the has-a relation is determined and potential aggregates are created. The process is described in Section 5.8.2 by the $\langle tansitiveClosure \rangle$ rule. Lastly, entities accessed together by business operations are also grouped in potential aggregates. When the set of entities already exists as a potential aggregate, the new potential aggregate is (unlike in Section 5.8.2) not a duplicate but is merged with the existing one as it makes gathering access frequencies for potential aggregates easier.

6.3.3 Weighting Potential Aggregates

The weighting process, as described in Section 5.8.3, is implemented as a set of classes implementing the *Heuristic* class (see Listing 6.5). Each Heuristic class' *calcAggregate-Likelihood* method is executed. The parameters of the execution are the entities of the potential aggregate, methods accessing the whole entity set (also containing the access frequency), and invariants representing a groups of entities and should be protected by an aggregate (see Section 3.4.3). The methods finally return a likelihood enum ranging from 'impossible' over 'unlikely', 'maybe', and 'likely' to 'definitely'. After the likelihood is being calculated, it is converted into a numeric value from -∞ over values from -4 to +4 to ∞(see Section 5.8.3). This values are then weighted using the result of the method's *weight()* function (usually 1.0) and then collected. When all weighted results are gathered, the potential aggregate is simply tagged with the sum of all positive or negative results. The sum enables zero likelihoods indicating that the method could not decide for or against the potential aggregate. By choosing an average function instead, potential aggregates with a lower overall certainty might be chosen, as their average is higher.

```
public abstract class Heuristic {
  public abstract Likelihood calcAggregateLikelihood(
     @NotNull Set<Entity> entities,
     @NotNull HashSet<Method> accessingMethods,
     @Nullable HashMap<DataElement, List<Invariant>>
        invariantMap);
  public abstract double weight();

  \\...
}
```

Listing 6.5: Abstract Heuristic Class

6.3.4 Creating Aggregates

Concluding the aggregate model-model transformation, aggregates are created from the weighted set of potential aggregates (see Section 5.8.2). For this, potential aggregates are sorted first by their weight and second by their size (preferring smaller aggregates (see Section 3.4.3)). Then the potential aggregate with the highest weight is drawn and converted to an aggregate. The process is repeated as long as potential aggregates exist with no entity in an already created aggregate.

As a final step, value objects are added to the aggregate. Multiple aggregates may hold the same value objects. This leads to the problem that elements are being distinguished using Java's object equality. Two different value objects would be painted as one node as they share the same *hash code* and are *equal*. Thus, the *ElementIdentifier*, as described in Section 6.2.2, contains a default Java *Universally Unique Identifier* enabling the creation of randomized identifier copies when needed.

6.3.5 Manually Defining Aggregates

As the heuristics for determining of aggregates are not perfect and sometimes one even have to choose between variants that have the same weight, a feature was build in to manually define aggregates.

For this, a row can be added to a Java property file (see Listing 6.6, Line 2 and 3) containing the name of the aggregate and its entities. An aggregate is generated from this definition and its entities are excluded from further transformation steps. It is possible to let the prototype name the aggregate by adding "auto_" as a prefix to the aggregate's name.

```
1 #AggregateAssignment.properties
2 auto_location = location,gate group
3 company = company
```

Listing 6.6: Manual Aggregate Definition

6.3.6 Services

Services (see Section 3.4.5) are obtained using a model-model transformation which translates the source model to the service model. For this, transformation rules (see Section 5.9.1) are applied to the source model. Unlike the aggregate determination, the model-model transformation can be performed in one single step. This and the fact that the required transformation rules are straightforward simplify the transformation process.

The prototype contains a simple loop for the transformation iterating over all business operations and checking which statement applies. As soon as a matching statement is found, the business operation is converted to a service or an object method continuing with the next loop iteration (see Listing 6.7).

```
// Create without updating or delete
if (created > 0 && deletedOrUpdated == 0) {
   serviceMethods.add(method);
   continue;
}
```

Listing 6.7: Service Transformation Loop Excerpt

6.4 Model-Artifact Transformation

The last step of the prototype utilizes a model-artifact transformation to create artifacts from the different created models (see Figure 6.4). The first type of artifacts (see Section 6.4.1) are graph files used to visualize the different models created by model-model transformations (aggregate and service models) and the artifact-model transformation (source model). The second type of artifact, namely a media wiki table (see Section 6.4.2), is a simple representation of the source model. It contains the identical information like the glossary and is used for communicating the ubiquitous language. The third and last artifact is source code generated from the aggregate and service models. Section 6.4.3 shows how FreeMake, a template generator, can be utilized for creating this code artifact.

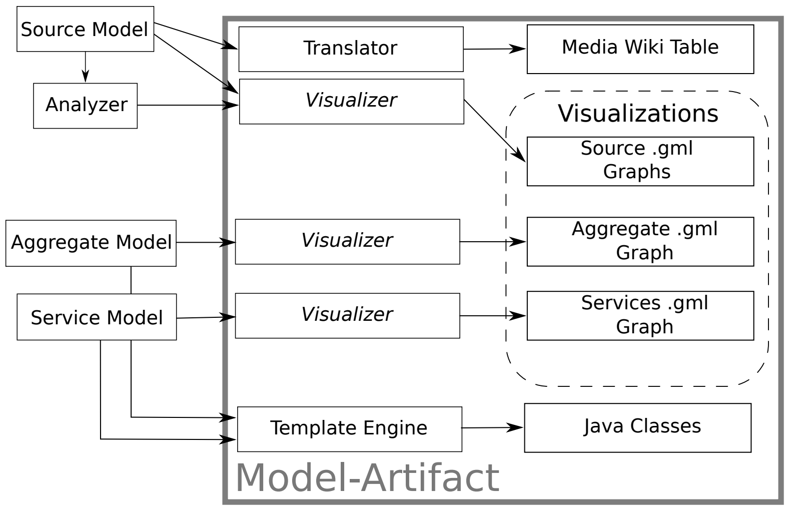

Figure 6.4: Model-Artifact Transformation

6.4.1 Visualization

To create a human friendly representation, *yED* (see Section 6.4.1) was chosen as it supports user friendly tools to interact with the graph and has various layouting mechanisms. To be able to utilize yED, a custom Java *Graph Modeling Language* (GML) exporter has been written (explained in the following). Lastly, this Section presents the result of transforming the object graph created in the *Artifact-Model Transformation* (see Section 6.2) to a human-friendly visual graph representation.

yED

Since the free version of *yED* [yWo] offers a good layout algorithm, it was chosen as the program of choice to display the graphs. To export the graph for yED, the GML [Him97] export function of the Java library *jgraphT* [Nav+] was chosen. Therefore, the created object graph is transformed into a jgraphT graph and, from there, into the gml format. Since the jgraphT's GMLWriter only offers limited support for the GML format as used by yED, a custom GML writer was created supporting different node types, borders, colors, edge types, labeled edges, arrows, and groups which can hold multiple nodes.

Listing 6.8 shows the interface created to define the layout of a graph for the custom GML writer. For each node V, edge E and (if exist) group G, the corresponding method is called where the formatting definition for the graph element is created and returned.

```
public interface YedGmlGraphicsProvider<V, E, G> {
  NodeGraphicDefinition getVertexGraphics(V vertex);
  EdgeGraphicDefinition getEdgeGraphics(E edge,
                    V edgeSource, V edgeTarget);
  @Nullable NodeGraphicDefinition getGroupGraphics(G group,
                    Set<V> groupElements);
}
```

Listing 6.8: Graphics Provider

Example 17 (Custom Entity Style)**.** Listing 6.9 shows an implementation painting every entity node white, with a dashed border, and a red font of the size 18. A simple *if statement* checks for entities in the *getVertexGraphics(V vertex)* method. When the if statement holds, an immutable *NodeGraphicDefinition* is returned with respective parameters set, otherwise a default *NodeGraphicDefinition* is returned.

```
@Override
public NodeGraphicDefinition getVertexGraphics(GraphNode
   vertex) {
if(vertex instanceof Entity){
  return new NodeGraphicDefinition.Builder(
      Color.white,GraphicDefinition.LineType.dashed)
        .setLabelColour(Color.red)
        .setFontSize(18)
        .build();
}else{
  return new NodeGraphicDefinition.Builder().build();
}
}
```

Listing 6.9: Custom Entity Style

Source Model Visualization

Figure 6.5 shows the result of the conversion of the source model to GML. The GML representation was then loaded it into yED where the organic layout was applied. The graph shows the different modules, e.g., Order, Location, Company. Each model contains entities as circles, value objects as dashed circles and business operations as yellow squares. Entities and value objects are colored in a scale from green to red according to the access frequency (see Section 6.2.4). Additionally, has-a edges are represented by lines with black arrows which are annotated with the weight of the relationship. Is-a relationships are drawn with dashed lines and white arrows.

This initial graph helps to get an overview of the domain model. It also helps to graphically validate the created glossary and business operations. In addition, as frequently accessed nodes are drawn in a red color, the visualization gives some indication which data is often accessed. Furthermore, business operations accessing many different data elements can be identified by analyzing the outgoing edges.

Figure 6.6 shows a visualization of module dependencies. Every time a method reads, writes or creates entities or value objects, a directed dependency edge is created. The dependancy edges are drawn between the modules pained as yellow squares. Furthermore, a has-a or is-a relation between the modules' data elements also results in drawing a dependency edge. This visualization helps to get an overview over the dependencies in the system at an early stage of the refactoring process. With this, unwanted dependencies (which partly exist in the figure) can be removed more easily. Especially when comparing to a self drawn dependency model, unwanted dependencies can be found and traced by looking at cross module accesses (see Figure 6.5). For this, realizing the graph in yED is helpful as modules could be moved freely. To compare the accesses of two modules, they can be simply moved next to each other in the source model visualization.

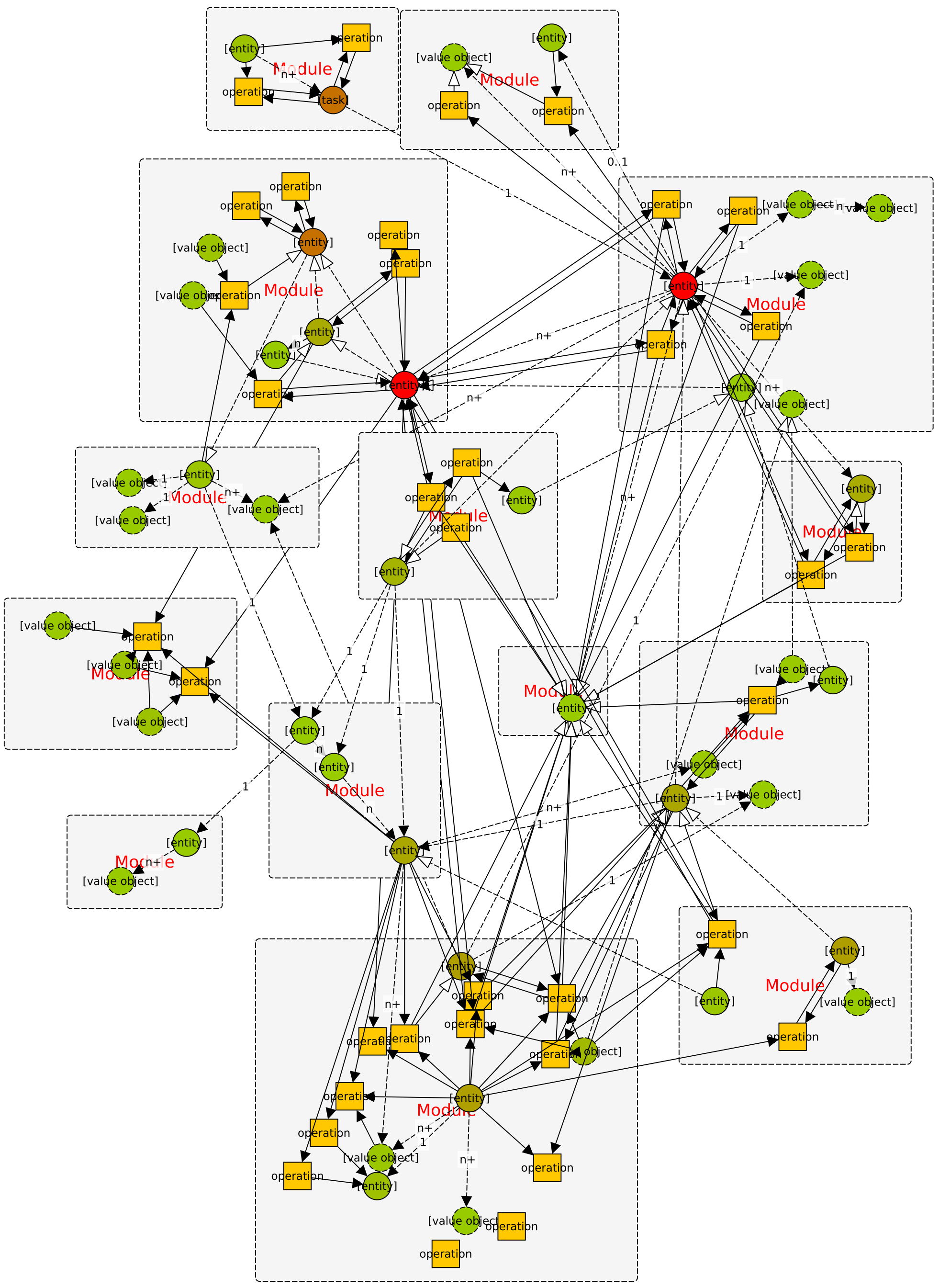

Figure 6.5: Source Model Visualization

Aggregate Visualization

The aggregate visualization can be seen in Figure 6.7. Aggregates are represented as grey dashed boxes containing entities and their value objects. The name of every aggregate is determined by its root entity. The number behind the name represents the confidence calculated from the summed up results of the weighting methods.

Service Visualization

The service visualization (see Figure 6.8) shows the business operations converted either to object methods or to services. When converted to object methods, they reside in an entity drawn as a green circle. Otherwise, they are visualized as yellow circles representing services. Value objects are also drawn as green circles but with a dashed line. A dotted line on an object method indicates the support of an service for its function.

For example, the object method *uncombine combined order* in *completed order* requires a service to delete the *combined order* entity. Services with dotted lines represent those accessing multiple entities in a way that these entities must either be part of an aggregate or be maintained by utilizing *Domain Events* (see Section 3.4.6). Last but not least, grayed out circles represent business operations for which the algorithm could not decide their category.

6.4.2 Media Wiki

With the spreadsheets set as the preferred document type for managing the ubiquitous language, the desire arose to communicate the content of the spreadsheets over the company-wide Wiki. This was achieved by transforming the source model (see Sections 5.7 and 6.2) to a text file artifact. Listing 6.10 shows an excerpt of the exported text file. The export itself is very basic: the object structure is iterated using ‘for’-loops and the required data strings are concatenated including separators such as “||” and “!!”. The document starts with a class definition which sets the design for displaying the data as html. The start of each line is annotated with “|-” whereby the first line is treated as the header of the table.

```
{| class="wikitable"
|-
! Term (en)                  !! Term (de)                  !!
   Module                     !! Identity                   !!
   Description                !! has-a                      !! is-a
|-
| company                    || Firma                      ||
   Company                    || company id                 || External
   assosiation.                        || relation{*}                ||
|-
| company id                 || Firmen ID                  ||
   Company                    ||                  || Companies Unique
   identifier.            ||                  ||
|-
| relation                   || Beziehung                  ||
   Company                    ||                  || Connection
   between two companies.               ||                  ||
|-
| relation type              || Beziehungstyp                        ||
   Company                    ||                  || Relation type
   between companies defining their roles.
   ||                 ||
|}
```

Listing 6.10: Media Wiki Export

6.4.3 Code Generation

To generate source code from the previously created aggregate and service models, *Apache FreeMaker* [Fou] was chosen as a tool of choice. FreeMaker is a template engine implemented as a Java library. It enables the creation of text files based on templates combined with the data accessible by a Java application.

Templates

FreeMaker templates can be used to generate Java sourcecode. For this, templates can be created for the elements of the *Tactical Design* (see Section 3.4). These templates contain special directives annotated with ${...} to be replaced with data from a Java data model.

Listing 6.11 shows a template which is used to create entity classes. At the top (Line 1 to 7), there is a function definition which converts the space separated names of entities and value objects into a *CamelCase format*. This function is first used to define the Java identifier of the class. In the class, references to value objects (Line 8 to 12), entities (Line 14 to 17) and the object methods (Line 21 to 25) are created. As some information is unknown, such as how the method operates, the developer has to fill in the missing information after the creation of the sourcecode.

```
1  <#function CamelCase word> <#assign result = "">
2          <#list ${word?split(" ")} as w>
3                  <#assign result = result + w?cap_first>
4          </#list>
5          <#return result>
6  </#function>
7  public class ${CamelCase(Entity.name)} implements Entity {
8  // value objects
9  <#list valueObjects as valueObject>
10 <#assign valueObj = CamelCase(valueObject.name?>
11 private ${valueObj} ${valueObj?uncap_first};
12 </#list>
13 // entities
14 <#list entities as entity>
15 <#assign ent = CamelCase(entity.name?>
private ${ent} ${ent?uncap_first};
</#list>
public ${Entity.name}() { // todo implement constructor
}
// object methods
<#list objectMethods as method>
public Object method(<#list method.inputs as
   parameter>${parameter},</#list>){
        return null; //todo implement
}
</#list>
//...
}
```

Listing 6.11: Template for an Entity Class

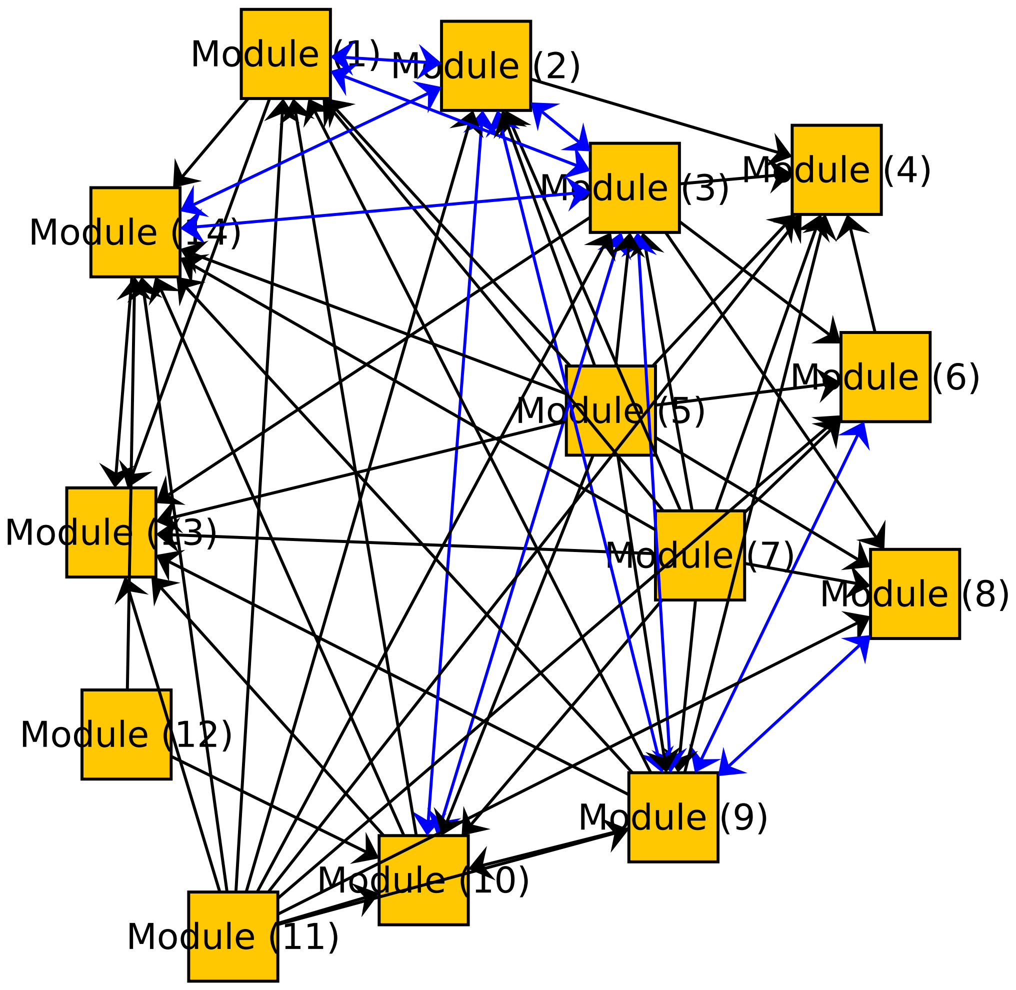

Figure 6.6: Module Dependency Visualization

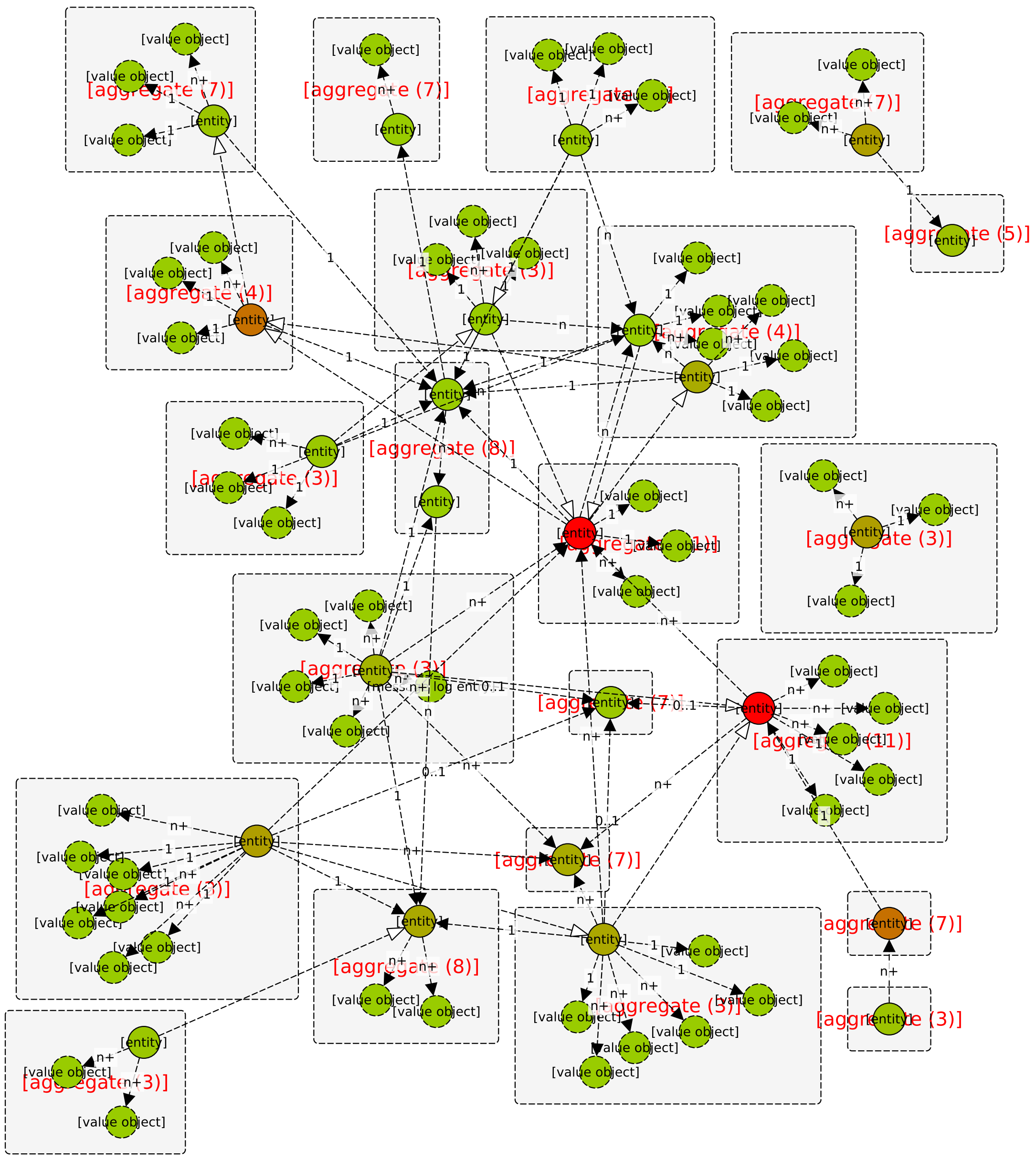

Figure 6.7: Aggregates

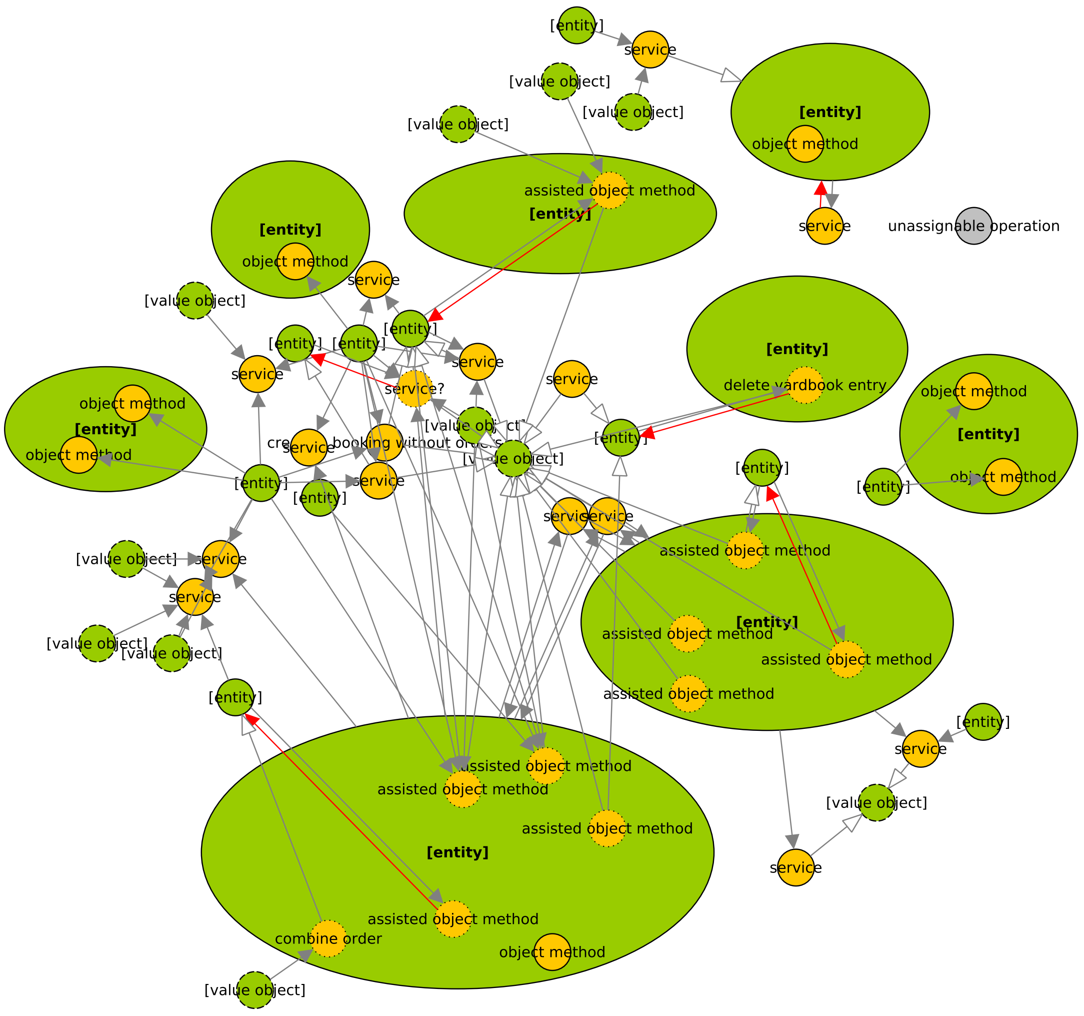

Figure 6.8: Services

6.5 Refactoring Realization

As the development team had no previous experience with DDD and still has to maintain the current TSM system it was decided against architectural refactoring of the whole TSM system. For this reason, MERCAREON decided that as a first step only a small module, the Live Yardview, is created using the proposed refactoring techniques based on DDD to establish an well maintainable architecture. This module is going to coexists with the existing TSM system. When completed, the module's bounded context is to be continuously expanded until reaching a fully-fledged architectural refactoring.

The Live Yardview's purpose is to act as a timeline resource planing tool for the retailer where *resources* have multiple *tasks*. The *resource* stands for an individual capacity for unloading. The *task* represents a delivery that is planned to be executed by a *resource*. The delivery is an *unloading task* expressed by a *workflow* comprised of a series of *dispatch states*. The *active dispatch state* hereby represents the current unloading status of a delivery.

For example, when goods are delivered, resources such as *fork lifts* or more abstract *gates* can be assigned to the *(un)loading task* of the goods. The progress of an *(un)loading task* is reflected by a change to the *active dispatch state*.

As described in Chapter 5, for creating a Domain-Driven Design for the Live Yardview, the collection of the ubiquitous language and the business operations is required to utilize model-model transformations. Therefore, the development started with collecting the glossary and the business operations of the TSM system's bounded context. The collection went through several iterations in cooperation with the management of MERCAREON. They are bound to go through additional iteration in the future as being maintained by the MERCAREON staff.

Next, the strategic design was determined for the existing systems architecture by following the advices of [Eva04] and [Vau13] to talk to the involved domain experts. After the strategic design had reached an acceptable state, the tactical design was created by gaining insights from the visualizations created by model-artifact transformations. Moreover, utilizing these visualizations, the different involved components were validated.

When the glossary for the ubiquitous language and the business operations are collected and the strategic and tactical design is created, the question arose how to fuse the existing TSM system with the refactored DDD-based architecture. For this, the *bubble context* approach was chosen.

The *bubble context* approach suggested by [Eva13] is used for the creation of DDD systems which are being surrounded by legacy systems. For this, a small, clean bounded context, as required by DDD, is created. This small bubble isolates the team's work from the existing legacy system. Additionally, the bubble context strategy does not require a big commitment to DDD. It allows a small team to create a working DDD design.

As can obtained of Figure 6.9, the bubble context used for the Live Yardview which utilizes *Ports and Adapters* (see Section 3.4.7) and the dependency inversion principle. The Live Yardview's *Domain Layer* contains the interfaces for repositories (R_1,...,R_n) and services (S_1,...,S_n) which are being implemented in the *Application Layer*. The repositories accesses the persistence layer of the TSM system over its *DAO interface*. The services accesses the logic layer of the TSM system over its *Service interface*. The translation between the two different ubiquitous language of the TSM system and the Live Yardview bounded contexts is realized in the repository and service implementations. These implementations act as an ACL in order to protect the newly created bounded context.

The goal is ultimately to evolve the bubble constantly by importing existing business operations of the TSM system's bounded context to achieve the desired architectural refactoring. For this, the ACL has to be adapted and maintained to cope with the growing bounded context. Furthermore, the bubble will, at some point, require its own data storage becoming an *autonomous bubble* [Eva13] and, thereby, resolving its dependencies to legacy parts.

By creating the bubble context, MERCAREON's development team has obtained the required experience to create further Domain-Driven Designs at minimal cost and risk [Eva13]. Additionally, parts of the bubble such as the glossary will continue to improve the experience within the company. Moreover, the developers of MERCAREON will gain a better insight into the domain improving continuous developments.

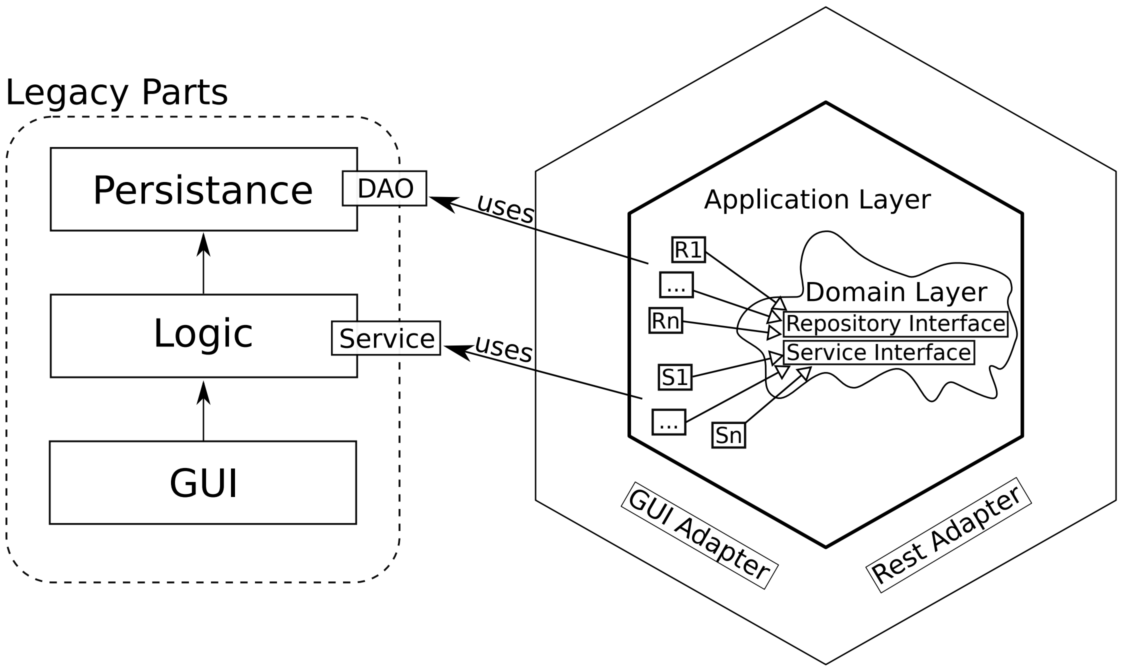

Figure 6.9: TSM system Architecture

7 Conclusion and Future Work

Domain-Driven Design (DDD) was not intended to be used for architectural refactoring. In [Eva13] it was even stated that the old model can be considered, but transformation of the model does not often succeed in practice as most systems have multiple and intermingled models. By utilizing the Create, Read, Update, Delete and frequency based business operations abstraction, the suggested approach was able to circumvent these problems by providing a graph transformation approach not directly coupled to the models of the old system.

The graph transformation approach, which was devised from the Model-Driven Architecture approach, utilizes artifact-model, model-model, and model-artifact transformations in order to support the creation of the DDD. A prototype has been created which utilizes these transformation steps to generate visual representations of the DDD tactical design from spreadsheets containing the ubiquitous language and the business operations. Moreover, based on the Model-Driven Architecture transformation approach, the prototype showcased how Java source code can be generated from DDD models.

The architectural refactoring approach was then realized by the MERCAREON company. For this, a small new module was created by the development team—the Live Yardview. The new module was created with a refactored DDD based architecture, which communicates with the legacy three-tier architecture. The design utilizes the ports and adapters architecture to protect the domain model. The communication with the legacy system is realized with bubble contexts [Eva13] in mind whereby repositories and services are utilized as an Anti Corruption Layer.

The created DDD based architecture will likely be extended in the near future. The functionality of the legacy TSM system is planned to be transfered step by step to the new architecture. This is possible as DDD is based on an adaptable model which, in turn, is adapted to the new needs by utilizing the prototype. As a future work, the benefits of continuous architectural refactoring towards the new DDD information system can be evaluated.

Another aspect that could be reviewed in the future, is whether the creation of the strategic design could be supported further. Currently the strategic design can only be evaluated after its creation, utilizing the automatically created tactical design artifacts. As for the tactical design, the selection of aggregates from the set of potential aggregates utilizes a greedy approach. This greedy approach certainly has some limitations, as selecting one particular aggregate might block several others. Choosing the wrong aggregate boundaries could lead to a suboptimal solution. Alternative approaches to solve this problem should be investigated.

All in all, it was observed that the presented architectural refactoring approach based on model transformations as suggested by Model-Driven Architecture is promising but also has some drawbacks. These drawbacks have to be addressed when dealing with systems, that have no clear models to work with.

Acronyms

ACL Anti Corruption Layer. 17, 25, 101, 103

API Application Programming Interface. 31

BO Business Operation. 39

CIM Computation Independent Model. 30, 31

CRUD Create, Read, Update, and Delete. 32, 39

CRUDI Create, Read, Update, Delete, and Input. 39, 40, 65, 69, 79, 81

CRUDM Create, Read, Update, and Delete Matrix. 32

DDB Distributed Database. 32, 33

DDD Domain-Driven Design. 2, 5, 7, 9, 10, 13, 19, 21, 24, 25, 29–32, 36, 37, 39, 42, 49, 58, 66, 75, 79, 80, 84, 100, 101, 103, 104, 111

GML Graph Modeling Language. 90, 91

GUI Graphical User Interface. 24

MDA Model-Driven Architecture. 30–32, 103, 104

MDD Model-Driven Design. 10

MDE Model-Driven Engineering. 9, 29–31

OHS Open Host Service. 16

PIM Platform Independent Model. 31

PSM Platform Specific Model. 31

sNets Semantic Networks. 45–48, 50, 54, 56, 60, 111

TSM system Time Slot Management system. 5–7, 14, 15, 18–20, 23, 38, 43, 46, 47, 100–103, 111, 112

UML Unified Modeling Language. 29–31

Bibliography

[Abr+01] Alain Abran et al. *Guide to the software engineering body of knowledge-SWEBOK*. IEEE Press, 2001

[Avr07] Abel Avram. *Domain-driven design Quickly*. Lulu, 2007

[Bri16] Encyclopædia Britannica. *Information System*. by Zwass Vladimir. 2016. URL: `http://global.britannica.com/topic/information-system` (visited on 02/15/2016)

[Bro14] Philip Brown. *What is the Domain Model in Domain Driven Design?* 2014. URL: `http://culttt.com/2014/11/12/domain-model-domain-driven-design/` (visited on 03/26/2016)

[Coc05] Alistair Cockburn. *Hexagonal architecture*. 2005. URL: `http://alistair.cockburn.us/Hexagonal+architecture` (visited on 11/23/2015)

[Coc06] Alistair Cockburn. *Ports And Adapters Architecture*. 2006. URL: `http://c2.com/cgi/wiki?PortsAndAdaptersArchitecture` (visited on 02/02/2016)

[CST00] W. Cazzola, R.J. Stroud, and F. Tisato. *Reflection and Software Engineering*. Lecture Notes in Computer Science. Springer, 2000

[Eva04] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004

[Eva09] Eric Evans. *What I've learned about DDD since the book*. NYC Domain-Driven Design User Group. 2009. URL: `http://www.dddcommunity.org/library/evans_2009_2/` (visited on 11/13/2015)

[Eva13] Eric Evans. *White paper: DDD Surrounded by Legacy Software*. Tech. rep. Domain Language, Inc, 2013

[Eva14] Eric Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. Dog Ear Publishing, LLC, 2014

[Fou] Apache Software Foundation. *FreeMarker*. URL: `http://freemarker.org` (visited on 04/16/2016)

[Fow02] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002

[Fow03] Martin Fowler. *Event Sourcing*. 2003. URL: `http://www.martinfowler.com/bliki/AnemicDomainModel.html` (visited on 04/19/2016)

[Fow04] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. 2004. URL: `http://www.martinfowler.com/articles/injection.html` (visited on 02/23/2016)

[Fow05a] Martin Fowler. *Code As Documentation*. 2005. URL: `http://martinfowler.com/bliki/CodeAsDocumentation.html` (visited on 03/11/2016)

[Fow05b] Martin Fowler. *Event Sourcing*. 2005. URL: `http://martinfowler.com/eaaDev/EventSourcing.html` (visited on 02/23/2016)

[FY97] Brian Foote and Joseph Yoder. “Big ball of mud”. In: *Pattern languages of program design* 4 (1997), pp. 654–692

[Gam95] Erich Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education India, 1995

[Git07] Vladimir Gitlevich. *What is Domain-Driven Design?* 2007. URL: `http://dddcommunity.org/learning-ddd/what_is_ddd/` (visited on 04/04/2016)

[Him97] Michael Himsolt. *GML: A portable Graph File Format*. Tech. rep. Universität Passau, 1997

[Joh02] Ralph Johnson. *Architecture*. 2002. URL: `https://groups.yahoo.com/neo/groups/extremeprogramming/conversations/messages/49656` (visited on 02/17/2016)

[KH10] Shahidul Islam Khan and ASML Hoque. “A new technique for database fragmentation in distributed systems”. In: *International Journal of Computer Applications* 5.9 (2010), pp. 20–24

[Lem98a] Richard Lemesle. “Meta-modeling and Modularity: Comparison between MOF, CDIF and sNets formalisms”. In: *OOPSLA98 Workshop# 25: Model Engineering, Methods and Tools Integration with CDIF*. Vancouver, Canada: Citeseer, 1998

[Lem98b] Richard Lemesle. “Transformation rules based on meta-modeling”. In: *Second International Enterprise Distributed Object Computing Workshop,EDOC*. La Jolla, CA, USA: IEEE, Nov. 1998, pp. 113–122

[Mic07] Microsoft. *.NET Framework*. 2007. URL: `https://www.microsoft.com/net/` (visited on 04/27/2016)

[MV06] Tom Mens and Pieter Van Gorp. “A taxonomy of model transformation”. In: *Electronic Notes in Theoretical Computer Science* 152 (2006), pp. 125–142

[Nan98] University Nantes. *sNets formalism*. Tech. rep. University of Nantes, 1998

[Nav+] Barak Naveh et al. *JGraphT*. URL: `http://jgrapht.sourceforge.net` (visited on 02/02/2016)

[Oli09] Jonathan Oliver. *DDD: Strategic Design: Core, Supporting, and Generic Subdomains*. 2009. URL: `http://blog.jonathanoliver.com/ddd-strategic-design-core-supporting-and-generic-subdomains/` (visited on 01/19/2016)

[OMG01] OMG – Architecture Board ORMSC. *Model Driven Architecture (MDA)*. Tech. rep. (draft) ormsc-01-07-01. Object Management Group (OMG), 2001

[OMG04] Object Management Group (OMG). *2.0 Superstructure Specification*. Tech. rep. Object Management Group (OMG), 2004

[OMG10] OMG – Architecture Board ORMSC. *MDA foundation model*. Tech. rep. (draft) ormsc-10-09-06. Object Management Group (OMG), 2010

[OMG14] OMG – Architecture Board ORMSC. *MDA Guide*. Tech. rep. Version 2.0 ormsc-14-06-01. Object Management Group (OMG), 2014

[OMG15] OMG – Architecture Board ORMSC. *Model Driven Architecture (MDA)*. 2015. URL: `http://www.omg.org/mda/` (visited on 04/05/2016)

[Ora] Oracle. *Java EE at a Glance*. URL: `http://www.oracle.com/technetwork/java/javaee/overview/index.html` (visited on 05/01/2016)

[ÖV96] M Tamer Özsu and Patrick Valduriez. “Distributed and parallel database systems”. In: *ACM Computing Surveys (CSUR)* 28.1 (1996), pp. 125–128

[Pal08] Jeffrey Palermo. *The onion architecture*. 2008. URL: `http://jeffreypalermo.com/blog/the-onion-architecture-part-1/` (visited on 02/02/2016)

[Pet13] Marian Petre. “UML in Practice”. In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE ’13. San Francisco, CA, USA: IEEE Press, 2013, pp. 722–731

[SK03] Shane Sendall and Wojtek Kozaczynski. *Model transformation the heart and soul of model-driven software development*. Tech. rep. Helsinki University of Technology, 2003

[SM07] Pankaj Saharan and Carlos Martinez. “Service-Oriented and Model Driven Architectures”. 2007

[SMC74] W.P. Stevens, G.J. Myers, and L.L. Constantine. “Structured design”. In: *IBM Systems Journal* 13.2 (1974), pp. 115–139

[Sta07] Michael Stal. *Architecture Refactoring*. 2007. URL: `http://stal.blogspot.de/2007/01/architecture-refactoring.html` (visited on 02/17/2016)

[Ste16] Martin Stefano Di. *Code refactoring vs Architecture Refactoring*. 2016. URL: `http://www.refactoringideas.com/code-refactoring-versus-architecture-refactoring/` (visited on 04/20/2016)

[Tho04] David Thomas. “MDA: Revenge of the Modelers or UML Utopia?” In: *Software, IEEE* 21.3 (2004), pp. 15–17

[Vau12] Vernon Vaughn. *Effective Aggregate Design Part 3*. DDD Meetup. 2012. URL: `http://www.dddcommunity.org/library/vernon_2012/` (visited on 11/13/2015)

[Vau13] Vernon Vaughn. *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013

[yWo] yWorks. *yEd*. URL: `https://www.yworks.com/products/yed` (visited on 02/02/2016)

[Zim15] Olaf Zimmermann. “Architectural Refactoring: A Task-Centric View on Software Evolution”. In: *IEEE Software* 2 (2015), pp. 26–29

List of Figures

Name: Hayato Hess Student id: 698230

Declaration

I hereby declare that I have developed and written the enclosed Master Thesis by myself and have not used sources or means without declaration in the text. This Master Thesis has never been used in the same or in a similar version to achieve an academic grading or was published elsewhere. Furthermore, I declare that gummy bears might have lost their lives for the creation of this thesis.

Ulm, May 2, 2016 ....

Hayato Hess