Domain-Driven Design Refactoring Tool

Manuel Baumgartner

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2019

© Copyright 2019 Manuel Baumgartner

This work is published under the conditions of the Creative Commons License *Attribution-NonCommercial-NoDerivatives 4.0 International* (CC BY-NC-ND 4.0)—see https://creativecommons.org/licenses/by-nc-nd/4.0/.

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 25, 2019

Manuel Baumgartner

Contents

Abstract

Nowadays, changes in a software model occur more often than expected. The simple modification of a single method can cause errors in different parts of the model. A boundary within the model and detailed diagrams are not available in most cases. It would be helpful to consider these before the implementation. Moreover, there might be misunderstandings between the domain experts (targeting the primary audience) and the developers. Domain-Driven Design was invented by Eric Evans in 2003 to make models more flexible and make clear decisions for the boundaries within them. Graphical visualisations and the definition of a ubiquitous language should help to develop the model. However, changing the current model to a DDD model is not a simple operation. The Domain-Driven Design Refactoring Tool was developed to support such an endeavour with a graphical view. The main focus is to load an existing model from the database and convert it, according to the DDD specifications. After that, the finished and converted model can be exported directly as a Java project and a database model.

Kurzfassung

Veränderungen in Software Modellen passieren heutzutage öfter als gedacht. Das einfache Entfernen von nur einer Methode kann dabei schon zu ungewollten Fehlern in einem ganz anderen Bereich des Modells führen. Eine Abgrenzung, sowie detaillierte Diagramme sind dabei meist nicht vorhanden, um sich schon vorab über die Implementierung Gedanken zu machen. Außerdem kann es auch zu Missverständnissen zwischen der Zielgruppe der Applikation (Domain Experten) und den Entwicklern kommen. Domain-Driven Design wurde 2003 von Eric Evans entwickelt um Modelle flexibler für Änderungen zu machen und auch klare Abgrenzungen zu treffen. Es soll dabei auch mithilfe von einer grafische Veranschaulichung und dem Definieren einer gemeinsamen Sprache der Entwicklung des Modelles und dem Aufbau der Software helfen. Es ist aber nicht trivial das aktuelle Modell auf DDD umzustellen. Dafür wurde das Domain-Driven Design Refactoring Tool mit grafischer Unterstützung entwickelt, um eine einfache grafische Veranschaulichung des Modelles zu ermöglichen und vor allem die Unterstützung ein bestehendes Modell aus einer bestehenden Datenbank abzuleiten und dieses anhand der DDD Spezifikationen umzuformen. Anschließend kann das fertige umgeformte Modell direkt als Java Projekt und Datenbank Modell exportiert werden.

Chapter 1

Introduction

Changing a complete software model is a cost-intensive and long process. Each class should be taken into account with different choices in the Domain-Driven Design (DDD) methodology. The goal of the thesis is to create a graphical supporting tool for such an endeavour, with choices and refactoring steps according to DDD. The import should work on any database model. Consequently, it converts it to a new DDD model.

Although the user can freely design in a graphical editor, some restrictions are necessary for the export of a real Java model. The export itself should create an Eclipse project containing the model classes together with a repository. Additionally, the system should create the necessary tables for the representation in a SQL-based database.

Changes of a model do occur sometimes and can be hard to handle because even small changes in the model can lead to side effects. DDD should make such modifications easier. The concept of *Bounded Contexts* helps to avoid unexpected side effects. Furthermore, defining a *Ubiquitous Language* supports the communication between different teams working on the software. However, changing the complete model by introducing this new approach may appear difficult. That is why a graphical tool should help.

Moreover, several operations might occur during the refactoring process. These can be typical DDD changes or also default changes which occur more often, such as creating a new class or properties in this class. The tool should also help with a general overview of the model by providing different zoom levels.

1.1 Structure

The thesis starts with the background topics *Domain-Driven Design*, *Refactoring* and *UML* in chapter 2. The main focus is on DDD and how it is used in a software model. Chapter 3 presents popular UML software. The three selected ones are *Visual Studio Entity Framework*, *Eclipse Modeling Framework* and also *UMLet*. *UMLet* is the most important one because it is the foundation of the project. The required modifications that are used are written in chapter 4, focusing on some typical DDD cases and the theory of the import and export. Chapter 5 shows the *Implementation* where some code snippets and screenshots are shown. In conclusion, the notes with some experts are summarised in chapter 6, including how they see the result of the refactoring tool. Finally, chapter 7 gives a quick review of the project and its outcomes.

Chapter 2

Background

The *Domain-Driven Design Refactoring Tool* builds on the concept of *Domain-Driven Design*. The user can import an existing model or design it from scratch. Some *Refactoring* methods support the user to convert a standard class model to a DDD model. All these operations are shown in a graphical way using the *Unified Modeling Language (UML)*. The foundations follow in this chapter.

2.1 Domain-Driven Design

The term *Domain-Driven Design*, or often referred to as DDD, was coined by Eric Evans in his book *Domain-Driven Design: Tackling Complexity in the Heart of Software* [16]. It talks about how to improve the model of a domain and the resulting software as well as making refactoring operations faster with fewer side-effects.

2.1.1 Domain

The term *Domain-Driven Design* rises the question of what a domain is. It is the entire area where a business is based on a realm, know-how and methodology [25, pp. 43–44]. Every person in the company works within this domain.

The domain can be seen as all-inclusive and is difficult to understand initially. However, almost always, the domain consists of different smaller areas called subdomains. In DDD each of the subdomains can consist of single or multiple bounded contexts. It should guarantee a separation, whereby none of them should stand alone and should have clearly defined connections to other subdomains.

With all these subdomains one of them should be of the highest importance. The team should think of the main success of the business – especially where the source of the most revenues is. An example is a customer booking a visit to a veterinary clinic. This important subdomain is called the *Core Domain*. The business should stand out with this core domain, which means that the business invests most effort in this subdomain.

However, the core domain should not stand on its own. All the other subdomains are still relevant. These surrounding subdomains can also be categorised in two areas. The *Supporting Subdomain* is, as it is written here, support for the core domain. With the veterinarian example, it can be an agenda for all the appointments of the clinic or a reminder function for the customer. All the other subdomains still being required for

the overall business are called *Generic Subdomains*. The billing for the customer is an example.

2.1.2 Model-Driven Design

Unlike the model-driven approach, there is the anaemic approach [28]. These models are widely used in enterprise applications alongside with the MVC pattern, whereby the controller handles all operations with the model, and the model itself consists only of the properties with public getters and setters. The structure with getters and setters is also required for ORM software such as *Hibernate*[1] because these frameworks have to access the properties for persistence and to read the data [25, p. 15]. This approach helps due to the separation, but it also brings some difficulties and can cause problems. Manipulation of the model is possible wherever an instance of the model is used in the entire application. For example, a veterinarian doctor stores the clients with a simple boolean flag for the first visit. The controller method for the visit `visitDoctor()` has to handle this flag, but if another method is added to the controller or this method is modified forgetting about this boolean flag, it will cause problems.

The model-driven design brings the behaviour directly to the model by declaring methods and leaving all the properties hidden with private visibility. The access is only possible through the defined behaviour avoiding manipulation from outside.

Another aspect is to make the model the central part of the software. In kind of small projects, the model might not matter, meaning that the model can be at several places of the software being the result of bad design. With the model as the central parts, it means that all the other parts, for example, the graphical user interface, has to access this model directly or via controller methods and a change of the model means that the parts depending on the model have to be changed as well.

2.1.3 Ubiquitous Language

Domain experts being specialised people in their area might use a different jargon than the software developers working on the same project. Developers usually use different terms and acronyms compared to domain experts, making communication difficult. Some people have to work as translators between them. It will be a bottleneck in communication. As a result, the exchange of knowledge usually fails or becomes at least very difficult.

Domain-Driven Design comes up with the idea of using the same terms across the project team. All these terms together should be part of this *Ubiquitous language* that is based on working with the model. It has to involve the domain expert's scope as well as the developers' scope, bringing them together. In cases of misunderstandings, domain experts have to point out the terms that are tricky or unsuitable for them. For example, the term *Class* might confuse people as they might think of a school class or a working class. In order to avoid confusion, a better term has to be defined. In this case, this might be the term *Template*. Nevertheless, the terms have to be used for the model construction including the name and properties and behaviour of the classes, although, it is vital that changes of the ubiquitous language automatically means changes in the

[1] http://hibernate.org/

model.

An example is a veterinary clinic with a customer and a patient. A doctor as the domain expert should know that the patient is the pet and the customer is its owner. However, this might be confusing for a software developer as he might not usually work in this area thinking that the customer is the pet and not the person owning it. Clarifying these terms among the team helps setting up the ubiquitous language.

2.1.4 Bounded Context

A bounded context is a specific area, where a dedicated project team works. These teams should work independently from all other teams. The full project context is the general domain that is separated into several subdomains. However, the borders of bounded contexts and subdomains can vary since the linguistic differences matter here, defined in the ubiquitous language. The ubiquitous language also means that each bounded context needs to define its terms and phrases that matter only within its boundaries. Moreover, it is possible to use the same term in two different bounded contexts with two different meanings. For example, the term "Account" can mean a social network account but also a bank account [25, p. 63].

Models usually consist of various elements, whereby each element has its own purpose. Relations between elements show that there are dependencies. In the view of design, a large number of different objects without any grouping might be confusing on first sight and takes more time to figure out the topic of this model.

Grouping elements together by drawing a boundary around them helps the project team to understand where the separations are. In a Java environment, packages are commonly used. In DDD these are specific elements, and most of the communication should be done in between these elements and coupling with different bounded contexts should be clearly defined and done with services.

A particular type of bounded context is the shared kernel. It should contain classes that are used in multiple other bounded contexts because replications of the classes/elements in two or more bounded contexts should be avoided.

According to Eric Evans, *Modules* are useful to split the responsibilities by using the package structure [16, pp. 109–116]. However, this is also the concept of bounded contexts. Nevertheless, he points out that the main differences between bounded contexts and modules are the patterns and the motivation but with the vital aspect that modules also organise the elements within a model. However, Vaughn Vernon means that modules are part of bounded contexts [25, p. 344]. According to this concept, all the connections on the outside should be controlled and done via services, whereby, the modules are located beside each other inside the bounded context also being loosely coupled, and more interactions can happen between them.

However, smaller sized grouping might not be necessary, in particular for small and medium-sized projects which the refactoring tool supports. Each bounded context has only one module or package. The main reason for this is simply because of the limited drawing area.

2.1.5 DDD-Based Architecture

In a developmental perspective, DDD is not an architecture by itself, but developers are using the following architectures in a model-driven context [25, pp. 113–168]. Additionally, it is also possible to combine them.

Layers: Putting all the logic in encapsulated layers is common in larger projects and even in networking with the ISO/OSI layers [23]. The key aspect is that each layer has distinctive responsibilities, whereby all the other layers should be encapsulated from it. However, there are some differences in DDD. Dependencies are allowed in one direction beyond the adjacent layers. These layers are the *Presentation Layer*, *Application Layer*, *Domain Layer* and *Infrastructure Layer* [16, pp. 68–71]. The top presentation layer has access to all lower layers, but the infrastructure layer has no access to the layers above. Also, the mediator and observer pattern can be used to access them. The aspect of encapsulation ensures easier maintenance efforts because changes should not affect other areas; however, it also means that the lower layers might have more dependencies than the higher layers influencing the maintenance costs.

Dependency Inversion Principle (DIP): Higher classes should not depend on lower subclasses. Both the superclass and the subclass should extend from the same interface or abstract class [21]. It means that the model should not depend on the repository, avoiding simple code lines using the `instanceof` operation. The repository depends on the model. An appropriate example for the DIP is the `InputStream` abstract class in the `java.io` framework. The subclass of the stream (e.g. `FileInputStream`) implements all the methods, whereby, the superclass reader does not need to know which implementation is used. It can be either a file or a network connection.

Hexagonal: This architecture has multiple names such as *Ports* or *Adaptors*. It separates to model to an inner and an outer part. The inner part is about the core domain. All the access from the outside works with adaptors and these adaptors handle the connection to the inside. The idea of using the hexagon is to create a category of the adaptors for each side with a maximum of six categories.

The drawback of this architecture is that it focuses more on the technical side and not on the business side [19].

Service Oriented: The service-oriented architecture (SOA) has multiple meanings and structures. It can be designed as abstraction, loosely coupled, or stateless depending on its purpose. The central concept of this service orientation is that business values are more important than the technical strategy and the strategic goals are more important than project-specific benefits [27]. It is about web services interacting with the model via SOAP or REST. The benefit is that it can be combined with a hexagonal architecture where the services are part of the outer part, besides the adaptors and other connectors. With this combination, it allows the developer to add new services for new purposes or customers. However, SOA does not support a technical definition. Just DDD defines the usage of bounded contexts, but influence by this architecture should be avoided as it may require to create a bounded context for each service, missing a primary purpose of DDD [25, p. 132]. However, if the business goals are considered higher than the technical

goals, as described in [27], it will work well with the bounded contexts, as they also focus on fragmenting the business needs.

REST: *Representational State Transfer* can be seen as an access method for the items in the model, which are connected via HTTP services. The access to the model is defined with four of the HTTP request methods [13]. They are *GET*, *POST*, *PUT* and *DELETE*, whereby, *GET*, *PUT* and *DELETE* can be called again without any problems, in case of an error. The key aspect here is that behind each request is a resource that will be modified or, if it is a *GET* request, it is only fetched.

REST also means that the web service does not store any state. This has to be handled on the client's side [22]. As a result, requests should not depend on previous requests or clients.

Although the architecture should not influence the model, it might help to create a separate bounded context for the REST operations in the interface layer. However, it decouples its core from the interface model, because changes of an object structure can influence the remote interface, and break the access for the clients [25, p. 137]. That means that changes in the object's structure of the core model will need a formal decision if they should also be added to the remote models.

A solution to this problem might be using the media type parameter in the HTTP header *(MIME-Type)*. Standard types can be used (e.g. text/html) or custom type names. The idea is here to provide a domain model for each format. Another solution might also be to use a version parameter in case of structural changes. An adaptor maps the current version of the object to the requested version and delivers the result.

Others: Besides all these underlying architectures, there are also some architectures that might be worth to be mentioned.

One of them is the *Command-Query Responsibility Segregation* [4, pp. 223–234]. The idea is to use two different databases. One of them is for gathering information (Query) and the other one for modifications (Command). The reading operation might be faster; however, as soon as a modification occurs, the command database has to push an update to the query database.

Another storage method is to store events with the *Event-Sourcing Architecture* [4, pp. 235–246]. On one hand, it stores permanent data; on the other hand, also all the modification operations are permanently stored. It can be compared with an undo/redo stack of any text editor, allowing the user to undo changes and helping to verify all the changes.

2.1.6 Entities

Entities can be seen everywhere in the world, and they contain multiple and sometimes thousands of attributes. It is also one of the fundamental elements of a database model alongside with relationships. That is why a database diagram is often called a *Entity Relationship Diagram (ERD)* [15]. These entities are very similar to those of DDD. In typical Java applications, the model is usually a representation of the database with minor adjustments (e.g. types), but they still use the same foundation of an entity.

Entities store a couple of properties and access methods to these properties. The

significant aspect that defines an entity is the identification. At least one property has to identify an entity uniquely. In database management systems (DBMS), this is defined as *Primary Key*. Java uses a default identification via the memory location [17, p. 35] (== operation with two objects) but it is not suitable as this identification is different on other machines and even instances and cannot be transferred to the database. Finding a unique identification inside the application is crucial for the entity. Failing to do so can lead to data corruption. There are several examples in the real world for identification. The social insurance number can identify a person but not everybody has an insurance number. In such a case, a symbol consisting of numbers and letters has to be added to the entity, ensuring a truly unique key. This symbol can be either generated on the database or the user side. Ensuring true uniqueness is more comfortable with an auto-increment number on the database. Nevertheless, Java already supports a generator for a unique identifier with the UUID (Universally Unique Identifier) class [17, p. 823]. They always have the same length of 128 bits represented as a zero-filled hexadecimal digit string [11]. Besides that, this ID must be immutable in all circumstances because a change can also lead to data corruption. Additionally, the ID can also be used to create references to other entities storing the foreign id in a property element. It creates a *Foreign Key* constraint that is also used in databases. The class structure can either use the ID or the direct reference to the object. Moreover, a reconstructing of the reference only needs the ID of the referenced element.

2.1.7 Value Objects

If an object only cares about its values without requiring an identification, it will be most certainly a *Value Object*. The goal is to reduce complexity because identification does not need to be tracked. A database usually automatically tracks the primary key with an auto-increment or a sequence, but on a Java platform, this might get a bit more complicated.

The absence of an identity is not the only aspect of value objects. The value objects can be reused to save memory. The problem is that several elements can use this object at the same time. If one value of this value object changes, the system might break. A solution to overcome this is to make them immutable. The life cycle is reduced to only two steps. The creation, being the first step, has to provide all the necessary values, and the second and last step is the deletion of the object. It must not allow changes during existence, avoiding side effects for multiple referencing objects. However, if a value has to be changed, a completely new value object can be created and reassigned.

Some of them are often used in Java. One of them is the `String` class for example or colours in the AWT toolkit. They are also immutable because all changes will create a new instance, and the original object stays the same.

Moreover, value objects can also reference other objects reusing the values directly. Connection of prices with the name of the product and the price itself is an example. The name of the product is the main value object referencing the value object of the price. However, a value object should not reference an entity as it can break the immutability [25, p. 222]. On the other hand, at least one entity has to reference the value object.

The storage of value object in the database is also an essential aspect because it has

to be a property of an entity. The database table has to have a column for the value object and stores a string representation containing all the values (e.g. JSON string).

2.1.8 Services

Some elements in the model cannot be clearly defined as an entity or a value object. It does not help to change the structure to fit them into one of these two categories. It means that only functions are used without any properties, and this is the definition of services. They are not encapsulated compared to the entities and value objects and provide access across the bounded contexts [26]. They should be stateless moving the state part either into the entities, value objects or the requester of the service. Although only functions are invoked with the service, it is possible to call methods of entities and value objects.

Services can also be categorised in domain services (microservices) and external services. A domain service provides access only within the system by either different bounded contexts, from a controller or the graphical user interface. Besides that, a service can be shared externally as a web-service (REST/SOAP).

2.1.9 Aggregates

Changes in the objects tend to happen quite often. In an entity, some records most certainly will be modified or deleted. It can influence the relations and the other referenced classes. An example is the removal of a user. All the user data should be cleared as well but the data might be shared with other users.

In this case, this should be handled by a single, specific *Entity* using a specific element as the root of a cluster that consists of several *Entities* and *Value Objects*. This element is the *Aggregate Root* with the primary purpose to provide access methods and saving references to the underlying *Entities* and *Value Objects*. The chosen entity may be referenced by classes outside of the aggregate and also for the services to provide access to other *Bounded Contexts* [16, p. 127].

2.1.10 Factories

In smaller software projects a class constructor can be created without parameters, and the settings of the values can be done afterwards via getters and setters, as it is default in anaemic models. Nonetheless, DDD uses rich client models transferring the operations in the method part of the model classes. It means that not all properties are accessible from outside. In this case, only the constructor is accessible, and this definition can be ambiguous in larger projects. Moreover, it might be the case for aggregate roots to create some entities or value object for several properties.

A solution for all these construction problems is to create factories [5]. They take the responsibility to create and instantiate new objects, including the references. An example is creating a new appointment to visit the veterinary doctor with the customer's phone number and the name of the pet. The factory for the appointment needs the phone number, name of the pet, name of the doctor and the time of the appointment. With all this information, it can automatically generate all the objects required to be stored in the database. With the phone number, the user can be searched in the system, and

with the user, the pet can be found. It is the same for the doctor as this person should also be on the doctor's table. As a result, this should all be combined in the aggregate root appointment and can be stored in the database.

In case of a new customer, the system might require additional information similar to a registration form when visiting a doctor for the first time. It also means that another factory should be created as it is for a different purpose, although, it is useful to make methods reusable for the factory steps trying to avoid redundancy.

2.2 Refactoring

Refactoring is also essential in software development. It is reworking bad-designed code into well-designed one avoiding changes to the external environment [9, p. 9]. Speaking about DDD, this means that the internal model should be changed including the database structure. However, all the services accessible from the outside should still receive the same requests and send the same response. To sum it up, refactoring can also be described in these three questions:

- What should be done?
- Why should it be done?
- When should it be done?

2.2.1 Refactoring Methods

It is starting with the question "what" connecting to processes that come from *Test Driven Development* because tests should be written before even starting the refactoring process. They should ensure the functionality of the code as changes may cause different results, even though they should not. After the tests have been defined, some changes can be made to the code with a test being executed after each modification. This loop should last until the code is well-designed meaning that no more changes seem to be necessary. Martin Fowler points out three general refactoring operations with an example [9, pp. 18–45]. They also indicate how refactoring can be done.

Extracting Methods and Classes: They are opportunities to avoid duplication and also points out that a method should have only one responsibility. Development IDEs such as Eclipse support this method using a simple dialogue and automatic creation of the method header, including a method call at the original position. Moreover, it keeps the code flow, and in case of a return value, it will also create the return statement and type. A similar method is also available for classes as they should also not grow too big.

Polymorphism over conditions: The goal here is to use the advantages of object-oriented programming. It is also combined with a class extraction because the superclass defines abstract methods, and the lower classes have to implement these methods. The advantage of this method is that adding extensions is easy and can be done by just adding new subclasses.

Move Method: Coming back to the responsibilities of methods, in the example of Martin Fowler, the full amount of video rentals is calculated for a customer. It is done by only one method in the customer class, although each rental has its price,

meaning that this calculation should be extracted and done within the rental class delivering the result to the customer class. Nevertheless, the price calculation of a single video is also not the responsibility of the rental. Therefore it should be in the video class. The advantage of the result can be seen after a change in the price calculation of a video.

The second question: "Why should it be done?" A reason might be the time that adding new functionality and bug-fixing wastes. If the new developer needs more than a week to understand the code, when it could have been learnt in a couple of hours, it will be an alarm for code improvement.

Even for companies, there is the aspect of time, answering the "when" question. It is useful to do it in regular cycles. On one hand, it might take too much time but, but on the other hand, it should not be a side task that will be done if there is nothing else to do. A process model has already been developed to include the refactoring process in regular cycles called *Extreme Programming* [14]. Another functional theory is the *Rule of Three*. In case of similar changes in row for three times, a refactoring process should follow [9, p. 49]. Another indicator is to see how long bug-fixes take. In case of very long duration and not understanding the concept after several days, refactoring should be considered as well. Intensive use of the copy-paste function also indicates redundancy that might cause troubles after some time. Extracting this copied code most certainly is useful.

2.2.2 Disadvantages

However, there are also some cases when a refactoring should not happen at all. If the code does not work at all or produces a wrong output, refactoring will not be useful as the semantic before and after should, in general, stay the same. If there are too many refactoring possibilities, it will be easier to rewrite the complete code part thinking about a new structure. A strong encapsulation might also make the refactoring difficult. Interfaces should also be an area of high importance due to their possible public access from the outside. They should not be published prematurely [9, p. 55], and changes have to be pointed out as well as the consideration of keeping the old signature of methods with a deprecated annotation. Thinking about publishing and releasing the product, refactoring should not happen if the team is already very close to the deadline, at least not at this particular moment.

2.2.3 Advantages

Besides these cases, refactoring might be useful, but this can also mean that maintaining more classes and methods might become more difficult. In particular, it may get more challenging to find them, and it can lead to more responsibilities. Nevertheless, creating new methods also has some distinct advantages.

Shared Logic: If lines of code are copied to other places because they have to be executed in different parts of the application, it will lead to code redundancy. This logic can be shared by creating a new method and call it from the original code location.

Isolate Changes: A method could be called in two different places and the behaviour

should be changed in one place. Changing the shared method changes both cases but can be avoided by creating a subclass implementation of this function and adding the additional code without changing the original method. The advantage is that the other case will not be affected.

Ensure Conditional Logic: This brings polymorphism over conditional structures by creating an abstract method and let the subclasses implement this method. The flexibility is higher for changes and extensions.

All these three advantages are also connected since either a subclass is created or new methods and classes. Refactoring mostly works with this methodology by using *Object-Oriented Programming*.

2.2.4 Designs

Also, the factor design in a software project is essential. Refactoring might get complicated if the current code should be changed to a new design guideline. That is why it is easier to start with a design approach and keep it for the project. Design is also essential for a program. However, it can also be done by creating a solution for the problem and redesign it afterwards, as this is more the case in *Extreme Programming*. Nevertheless, keeping the design might seem easier finding a reasonable and also a flexible solution, but sometimes it is the case that a flexible solution is not needed or not in this level, and a more straightforward solution might be better. It is mostly the case if a simple solution can be quickly changed into a flexible solution [9, p. 57].

2.2.5 Tools

With development IDEs such as *Eclipse* refactoring processes may have become faster and also simpler for the developers. An example is the *Extract Method* functionality as described in the section 2.2.1. Keeping it this way, Martin Fowler points out some practical criteria for them [9, pp. 331–332].

Speed: Putting a method from the one class to another seems to be comfortable with a copy/paste operations. However, access to variables are not considered, and new parameters have to be added. The tool should automatically scan them and create them automatically. Nevertheless, if this operation takes more time than just copying and pasting, including the variable access check, it will not be used by the developers.

Undo: The tool has been designed by the guidelines that the original developer of it had. It might not be the same for a new project. Considering the outcome after the refactoring process, the user might not be happy about the result. In such a case, it would be helpful to undo these changes.

Integration: The developer should also find the refactoring extension, and with a separation of the refactoring and the IDE, it might not be used as it cannot be found. For more extensive refactoring operations it might help to create them as an external application, especially for model creation and modification tools.

2.3 UML

In the DDD environment, the code itself is crucial, but sketching the software is an even more significant part. All the ideas are delighted to be designed on paper. However, it might be easier to design them in an editor. A UML diagram will open a way for communication and understanding [10, p. 6]. These UML diagrams usually do not affect the source code directly but should bring up the ideas for the software. The ideas should be much easier to understand and also helps to communicate between different teams in an *Object-Oriented Environment*, but therefore, standardisation is required.

In a DDD environment, the most valuable kind of UML diagram is the class diagram. However, this is not the only used diagram type. For example, *Bounded Contexts*, are usually not supported by default UML class diagrams but another kind of diagram can be used for this purpose. *Package Diagrams* are handy for grouping elements together. Moreover, some modifications can be done to display DDD diagrams.

2.3.1 Class Diagram

UML consists of multiple types of diagrams whereby the most widely used is the class diagram [10, p. 35]. In all object-oriented Programming languages, they are helpful in designing the properties and methods and having a quick overview. With UMLet, it is possible to create such diagrams by defining the properties and methods in the properties input-field. The specific rules about the structure of properties are:

Visibility: It defines the accessibility of the property. In Java classes and anaemic models, this is per default "private", but it can also be in "package" visibility, meaning that only classes from the same package can access the variable. Additionally, the "protected" keyword adds the subclasses to gain access. Finally, to be accessible from everywhere, the visibility has to be set to "public". Four symbols are used in the UML standard to display the visibility quickly. `-` is for private, `~` for package, `#` for protected and `+` for public [10, p. 38].

Name: The name of the property is a combination of letters and numbers, whereby the first character has to be a letter.

Type: The type makes the property either an elementary type (String, int, et al.) or a reference to another object (Date, Colour, et al.) including references to other classes in the same diagram.

Default Value: In case of an undefined property in the constructor, a default value can be declared here.

Property String: A string here is for additional properties defining the behaviour of the attribute. It can be a final declaration for a constant.

Figure 2.1 shows the properties in the central part of the class diagram. In a default UML environment, the user is not bound to this structure. The goal of it is to be flexible as not all the components have to be added to the property definition, but the name is always necessary [10, p. 37]. However, building a real model out of this design requires more elements of the standard with the right order. For that reason, the input of the already mentioned elements needs to have a better-controlled input.

It is also true for the relations because they have to be clearly defined by each prop-

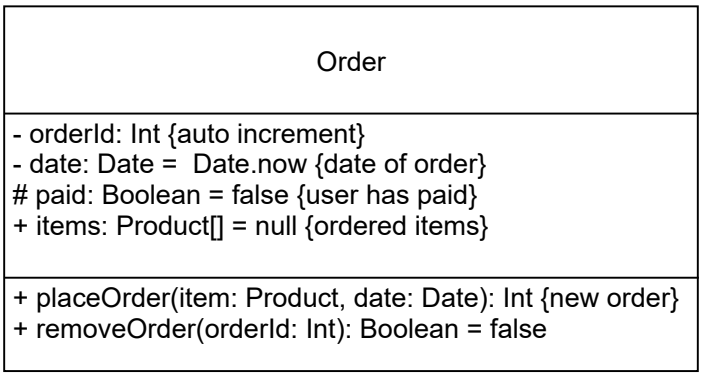


Figure 2.1: UML Class Diagram Example

erty using the reference to another class. This type of relation is called an association. The stored instance is the property using its type from the class it references. An arrow is drawn from the property to the desired class being the position of the arrowhead to show this reference.

Associations need to define a multiplicity or also called cardinality. It declares how many objects can be assigned to the property [10, pp. 38–39]. It can be compared to a foreign key constraint in a database column because the constraint only allows values from the other table's primary key being the unique identifier of the class. In general, it is a one-to-many relationship.

An example is a relationship between the pet and the owner visiting the veterinarian clinic. The pet has only one owner that is the customer, but the customer can have more than one pet.

If a collection for storing multiple instances in one property is used, the multiplicity will be many-to-many because the list can grow to an infinite amount and also the number of possible references are not limited.

Besides the properties, there is also a notation for the methods. They have a similar structure starting again with the visibility.

Visibility: The visibility is the same as the properties but with the difference that methods are usually declared as public.

Name: The name is also with a combination of letters and numbers.

Parameter List: The parameter-list as a list of parameters that are handed over to the method with each parameter having the notation `name: type` being separated by a comma.

Return Type: The return type is the type that will be returned after the method has been finished or `void` in case of a procedural method without a return value.

Property String: This is also for declaring additional properties.

Figure 2.1 shows the methods in the lower part of the class diagram.

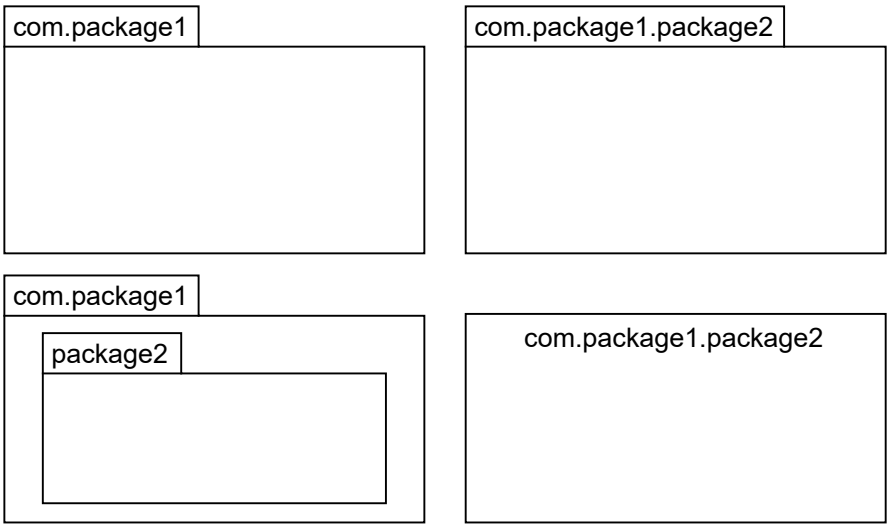


Figure 2.2: UML Package Diagram Examples

2.3.2 Package Diagram

Class diagrams show the structure of a model, but for larger models, more visualisations with more abstraction levels are needed [10, p. 73]. An approach is the package diagram because it allows elements to be split into several groups and only shows the group without its contents for a higher level of abstraction. Elements with more dependencies should be put together similar to packages in Java.

The default form for the packages is a tabbed folder, whereby, the name of the package is inside this tab which can be either the fully qualified path or just the relative name. Nonetheless, it is also possible to use a rectangle with the title either on top or in the centre. Sub-packages can also be included within the package to form a nesting package. An example of these different package diagram notations can be seen in figure 2.2.

It can be a difficult question to determine which classes should be put in which package. Whereas, the DDD approach already defines a useful way to determine the packages, concerning the classes, by creating bounded contexts, defining which elements should be in the same context, and which ones should be in another bounded context. A bounded context should be a package hiding the core model from others using visibility. Services and aggregate roots, however, should be visible. In Java, packages do not handle visibility. It depends on the classes inside. In this case, services and aggregate roots are public, and the rest is only visible within the same package.

Chapter 3

Related Work

UML design software is widely used in the software development process presented here. It can either be used for the documentation or communication across the project team. Some of them support to create and a database or code model out of it. The three selected tools are the *Visual Studio Entity Framework*, *Eclipse Modelling Framework* and *UMLet* due to its widespread use. Besides that, *UMLet* is the foundation for the refactoring tool created for the thesis.

3.1 Visual Studio Entity Framework

Visual Studio itself is a development software developed by Microsoft[1]. It supports multiple programming languages like Visual Basic, C++, C# and even more common .NET programming languages, but the most widely used one is C# with different technologies and frameworks available. One of these frameworks handles the interaction between the database and the model of the software, which is also called repository. In Visual Studio, it is called the *Entity Framework* [20].

The creation of the model, including the database and the code, works with three different approaches. They depend on the source for the entity framework and how to start.

Database First: This approach is used for projects with an existing and working database, including the model (tables, views, et al.) meaning that the project does not have to start from scratch. With an import dialogue, the user has to select the tables that should be part of the model. With the import, the framework automatically creates the UML representation allowing the user to customise the code that should be created out of it.

Code First: The code first means that instead of the database, the code is created first. The database tables are going to be created from this code, including the UML representation. The only drawback using this approach is that all changes in the code have to be pushed again to the database.

Model First: This is a different approach via the UML diagram designer. An example of this designer showing the example of the veterinarian clinic for the appoint-

[1] https://visualstudio.microsoft.com

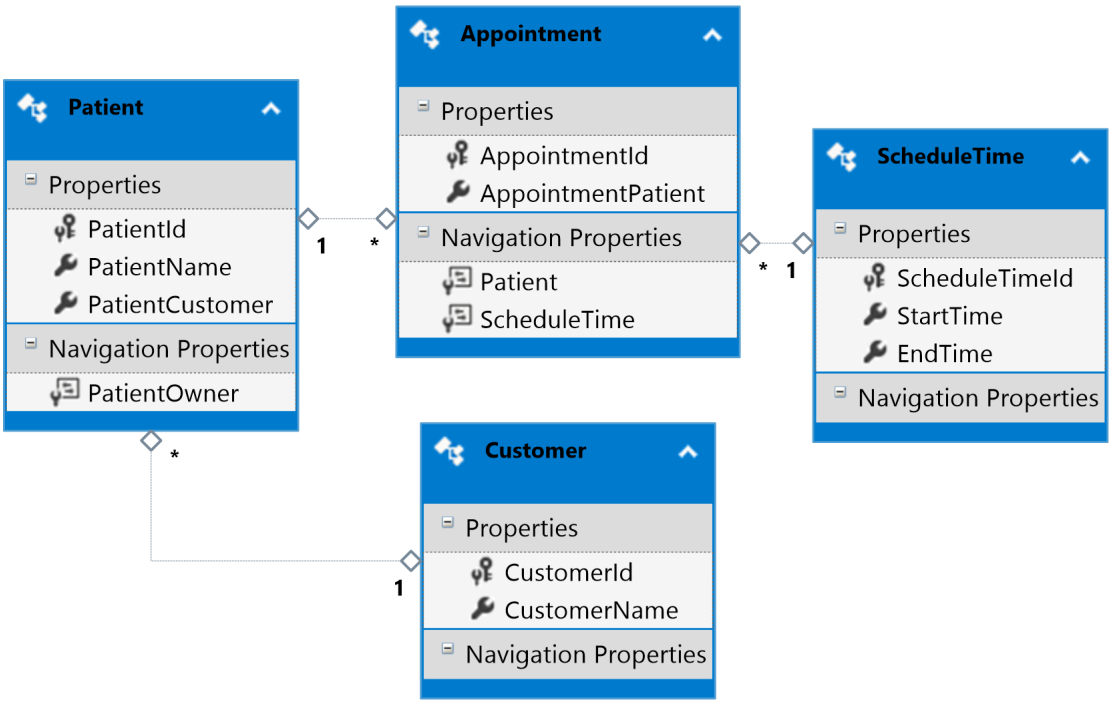


Figure 3.1: *Visual Studio Entity Framework* example

ment can be seen at figure 3.1. The user creates all the classes in this designer, including the relations between the entities. The elementary properties are listed in the properties pane, and the foreign key properties are listed in the *Navigation Properties* panel. The code representation of the model will always be created if a change occurs, but this is not true for the database. After having finished the design, the *Generate Database from Model* task has to be executed via the context menu pushing the changes to the database; however, it just creates the SQL script that the user has to execute.

With all these three approaches, the *Model First* approach is used to create the conceptual model by using *Domain-Driven* patterns and let the database and code created out of it. Additionally, the creation of the model can be supported via the database input using the *Database First* approach.

Besides the task of creating the model, the entity framework brings more advantages for the developers. After having created a model from the database or designed the model from scratch, all the code, representing the model, is automatically generated. Moreover, all changes in the designer that are made afterwards are also transferred to the source code, keeping it up-to-date. Moreover, the connection from the program to the database is handled by the framework, decreasing the amount of code that has to be written for the database requests [20].

The designer itself is both for checking the current state of the data model as it has been imported from the database or the code and refactoring this model. Most of the refactoring operations start from the context menu and the toolbox. Adding a new

entity and relations are examples for typical refactoring operations. Additionally, the creation operations are mostly in dialogues determining the name of an entity and the key property (primary key). This is also true for the associations because the dialogue here needs both ends of the relation with the multiplicity being `0`, `1` or `*` and the names of the properties for the access called the navigation properties.

3.2 Eclipse Modeling Framework

Eclipse[2], being a free IDE mostly for Java, is using a lot of tools and frameworks to support the developers. The particular framework *Eclipse Modeling Framework* is used for the class model creation. It is primarily made for the modelling first approach. First, the model is designed. Second, other tasks are considered, such as the user interface. The *Eclipse Modeling Framework* should help with designing this model and providing code generation functionality [1]. It is a unification of UML, XML and Java [3]. XML here can be used for the description of a model and also for storing it as a file. This unification also means that the definition of the model has to be in one of these formats and the modelling framework will create the other two formats from it. With DDD, an example is to create the UML diagram first and the EMF creates the Java implementation as well as the XML representation.

The model to describe models is defined in the `ECore` model that contains information about the defined classes. This is also called a meta-model [12]. This meta-model is based on XML, and the root element is the `EPackage` storing the package name. All the classes are children of this `EPackage`. Additionally, each class can consist of these four elements:

EClass: This is for the EMF representation of a `class` with at least a name. Attributes and references are optional, but in most cases, it should have a couple of attributes, whereby, there might be no references or very few of them. The attributes are described as an `EAttribute`.

EAttribute: They contain a name and a type but mostly primitive types such as int, char and long or object types extending from the `EDateType`.

EDateType: They are used for the type representation and are a bit different to default Java data types because all of them use an "E" prefix. Examples are `EString`, `EInt` and `EDate`. However, in the case of referencing another class in a diagram, the `EReference` is used for it.

EReference: They replace the `EAttribute` as a definition of the relationship between two classes by associating the type of the destination class. The main attributes of a reference are also the name with the target type, which is here the type of another class in the same diagram. Additionally, it requires a boolean flag indicating the representation of a containment.

These *ECore* models can be created in the editor as a tree structure or via the XML file, but for DDD the UML designer is the main area of interest. An example of the designer can be seen in figure 3.2. The designer allows the user to create new classes, adding attributes and handle the association between the elements. All these operations

[2]https://www.eclipse.org/

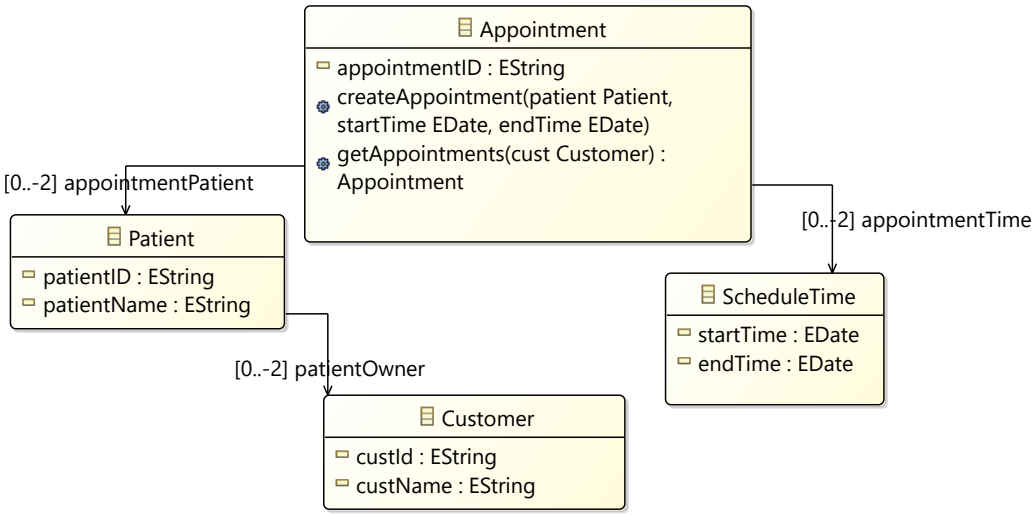


Figure 3.2: *Eclipse Modelling Framework* example

can be done with the toolbox. After the selection of an element, the user has to select the desired location for a new class or the class for new attributes or associations.

The next step is to build the Java model out of the designed UML model.

3.3 UMLet

UML is the de facto standard for graphical class model design [6]. Most products here are part of a framework or an IDE and they work only in their development environment. *UMLet* is an open source software for designing UML diagrams. It is written in Java also with the functionality of gathering information from Java source or object files using external libraries such as "JaPa" (Java Parser)[3]. This is kind of a code first approach, whereby, the code will not be modified. Although this part is limited to Java, it should help without being restricted to any development area. It does already contain the elements to create a graphical representation of the model by quickly drawing them with a simple mark-up and without input dialogues [8, p. 1].

The GUI is using several containers and controls of the Java Swing [7] library. An example can be seen in figure 3.3. The main drawing area can open up multiple files with different tab-folders. The toolbar on the right side is similar to the main drawing area with different toolbar types. Furthermore, it allows the user to drag and drop an object from the toolbar into the drawing area. A copy will be created in this process.

UML itself supports multiple types of diagrams. The most important one is the class diagram. In all object-oriented programming languages class diagrams are very helpful in designing the properties and methods by having a quick overview. With *UMLet*, it is possible to create such diagrams by defining the properties and methods in the properties input-field on the right bottom corner. The specific rules about the structure

[3] https://javaparser.org/

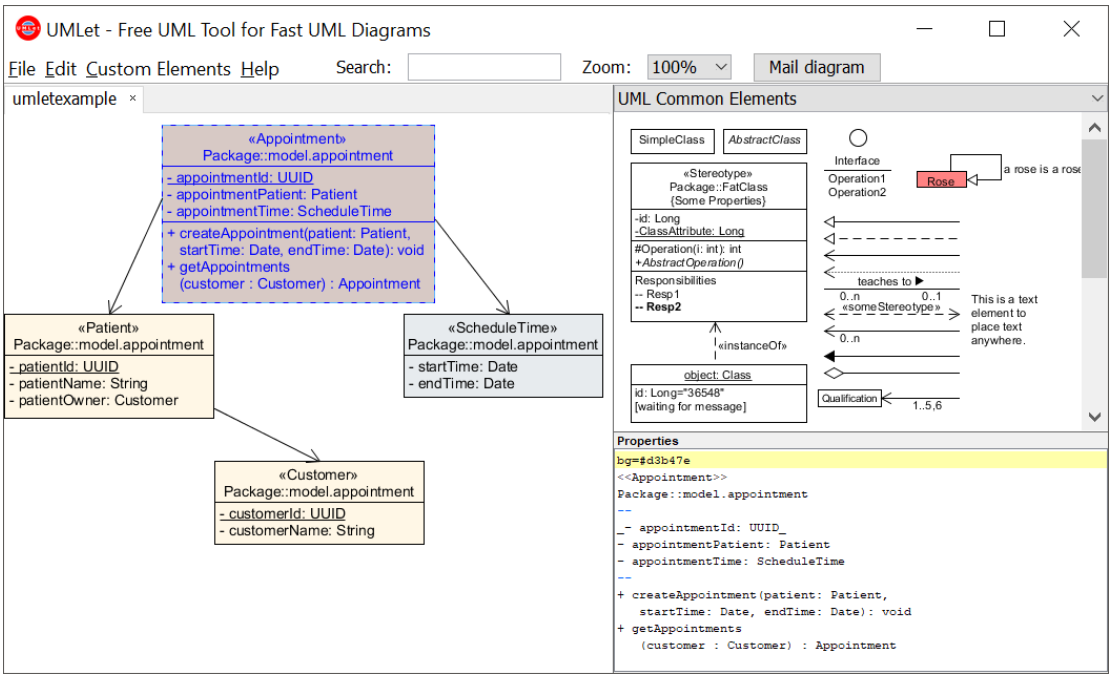


Figure 3.3: *UMLet* example

of properties are:

- visibility (private, package, protected, public),
- name and
- type (String, int, et al.).

This is described in detail in section 2.3.1. However, in *UMLet* this structure is just a recommendation because the goal of the software is to be a lightweight editor for UML diagrams.

Relations can be drawn to connect elements. They will keep the same relative position of the bounding polygon after the element has been moved to another position. Moreover, types can also be changed more quickly compared to different platform dependent modelling tools by the properties text-input. All the other defined elements, such as the multiplicity, are reused.

This is also true for whole classes and all other elements because a copy of them has the same properties and size. Nonetheless, it is an entirely new element, and changes will not affect the original one [8, p. 4]. As a summary, this means that almost all of the user-defined definitions and customisations happen in this text form supporting default keyboard shortcuts and avoiding the usage of multiple dialogues.

It is also useful to share these diagrams across the team or use them in presentations. Communication is also an essential aspect in UML [10, p. 6]. The user of this diagram might require this in a specific format such as a high or low-resolution raster graphic (`JPEG`, `PNG`) or a vector graphic (`SVG`, `PDF`). External open-source libraries are used, such as Batik for SVG files, making it extendable for more formats.

Table 3.1: UML Tools Comparison

	Visual Studio Entity Framework	Eclipse EMF	UMLet
Start Approaches	Model, Code, DB	Model	Model, Code
Create DB	Yes	No	No
Read from Code	Yes	No	Yes (Java only)
Generate Code	Yes	Yes	No
Supported Languages	C#, Visual Basic, .NET	Java	Independent
Export Model Image	No	Yes (.png, .svg)	Yes (.png, .pdf, .svg)
Customised Elements	No	No	Yes
Open Source	No	Yes	Yes

3.4 Comparison

Table 3.1 shows these three UML tools with different aspects that are required for the thesis project. UMLet has been chosen as the foundation because it is open source and uses the UI framework "Java Swing". The model can be exported as an image file, but it does not support the export to a real Java or database model. Although Eclipse EMF supports it, there are no customisable elements. The Visual Studio Entity Framework is closed source, that is why extensions and modifications might not be possible. Besides that, Visual Studio does not support Java as a programming language.

Chapter 4

Own Approach

This chapter describes how the concepts of DDD and UML methodology are used in the project for the thesis (DDDrt). There are several steps to create a DDD model from a usual object-oriented one. In the planned application, a model is created either from scratch or by loading an existing model from the database. This model will be refactored via the methods described in section 4.3. In the end, a Java project should be exported with a suitable database and code for a repository, as described in section 4.5.

4.1 Class Structure

The DDD approach requires a different kind of class elements, for example, entities or value objects, and almost all of them can be seen as regular Java classes. The general structure of them are:

Name: A unique name for the class,

Properties: variables and

Methods: control the changes of variables and add behaviour.

An example of an entity can be seen in figure 4.1.

In general the three elements (*Entities*, *Value Objects* and *Aggregate Root*) contain

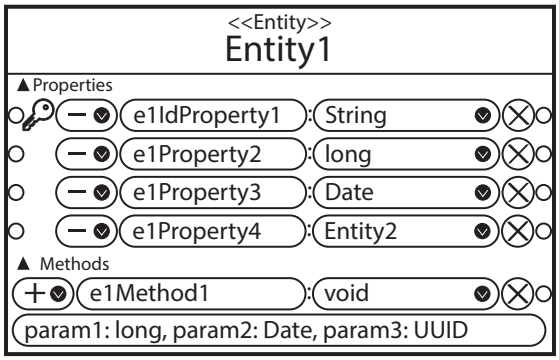


Figure 4.1: *Entity example* example

these three parts. *Services*, however, do not require properties because they provide only methods for external access. Moreover, *Entities* and *Aggregate Roots* need to define a unique identity as a primary key. Only the name of the property is required to keep diagrams clear and readable, details like visibility and type specifications are omitted. Nevertheless, they are required in Java.

4.2 Import

Using an already existing model is a major aspect for refactoring existing code. In the case of this project, this model can be imported either from Java classes or directly from the database. The advantage of using a Java model is that it only requires the Java files without any database connection, but it requires extensions for other programming languages. On the other hand, working with a database is independent of the programming language. For example, the task is to change from a simple CRUD (Create, Read, Update, Delete) application, written in Oracle APEX, to a Java-based technology. This means that a local Java class model is not available. Business software and websites usually use a database behind the scene with all the entities and relations. The goal is to reuse them for the new DDD approach. [24] describes the reverse engineering process in two major steps.

Model Extraction: The first step is gathering the name and the type of the tables and its columns. Methods such as PL/SQL functions, procedures and triggers do not have to be considered.

Constraints Extraction: The second step is loading the constraints of the tables and especially of the columns. These are *Primary Keys* for a unique property and *Foreign Keys* for the relations.

Additionally, the reverse engineering of a database works with the meta-tables of the database. The three major elements for the loading procedure are *Tables*, *Columns* and *Relations*. This means that three loading steps are required due to this database structure. The real steps for a real database are:

1. the name of all tables from the current user trying to avoid loading other meta-tables,
2. the name, data type, length, et al. of the columns and
3. finally the constraints.

Both step one and step two are separate steps here. An example import can be seen in figure 4.2. It takes the model from the ERD (Entity Relationship Diagram) and converts it to the DDD diagram.

The database tables have to be converted to Java classes. This, in particular, includes the name of the class and the properties but also the type of each property. In particular, property types show some differences. For example, the type NUMBER as in table 4.1 can be a different type in Java depending on its length, but the length in the database can have customised limits for some types, particularly numbers. However, in Java, there is only a general limit depending on the byte length of the given type. Nonetheless, it can be simulated either with an array or storing the length in annotations and checking this length in the setter.

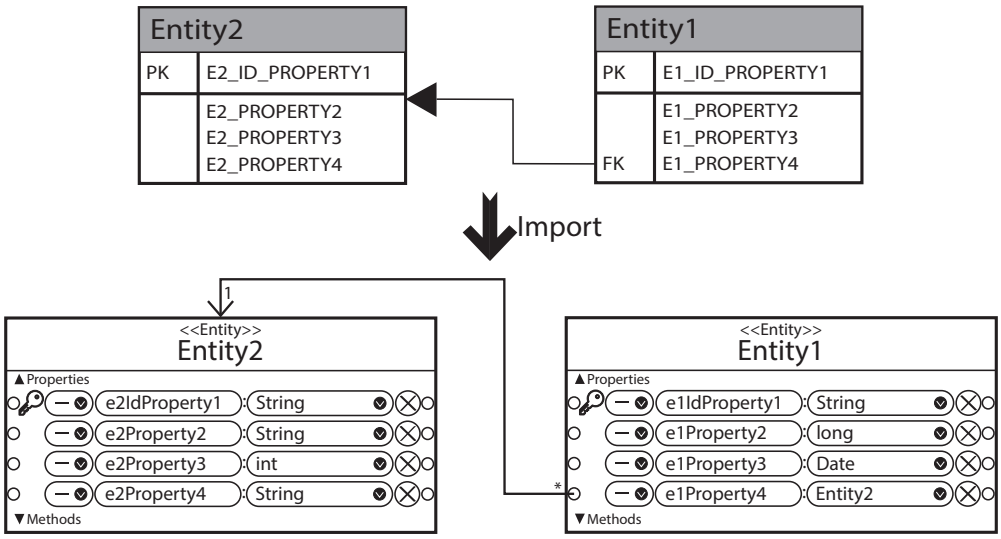


Figure 4.2: *Database Import* example

Table 4.1: Java Types versus Database Types

Java Type	Database Type
byte	NUMBER(<3, 0)
short	NUMBER(<5, 0)
int	NUMBER(<10, 0)
long	NUMBER(≥10, 0)
double	NUMBER(X, X)
java.util.Date	DATE
String	VARCHAR2(X), CLOB, CHAR(X), ...

The names of both the database table and the column are usually written in upper case letters with underline separators. This might be a bit unusual in Java. That is why the names should be converted to camel case notation for the Java classes, properties and methods. An example of this conversion from database tables to Java classes can be seen in figure 4.2.

4.3 Refactoring

The central part of the tool is to refactor the model from the database or Java classes and convert it to be DDD conform. For this several steps have to be taken by the user, and the tool supports this. The operations can be done in a custom order, but some operations can only be performed on particular classes, whereby, the *Change type* operation can influence it, as it changes the type of a particular class. For example, a value object does not have a primary key.

Also, the refactoring methods are categorised in the sections of DDD. These are

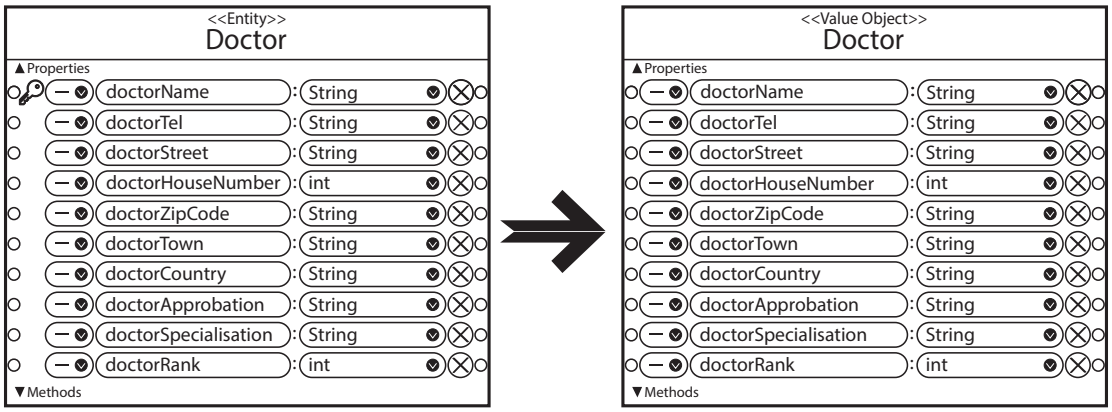


Figure 4.3: Class Type Change

elements, properties, methods, bounded contexts and relations. The refactoring of one section should not affect others except for the "Change type" operation.

4.3.1 Classes

In the first step, the user needs to decide whether a class should be an *Entity*, a *Value Object*, an *Aggregate Root* or a *Service*. It enables more or less functionality for the given element. Entities and aggregate roots require a unique property being also the primary key column in the table. This type can also be changed later.

Nevertheless, changing back to a value object or a service removes the unique identity. Besides that, an aggregate root is very similar to an Entity. It can be seen as an associative table in the database for a many-to-many relationship. It is also the case here to make relations easier. Another part of the aggregate root is to provide methods for the services. Services only contain methods without any properties. Services and aggregate roots should provide almost all of the public methods and the changing objects within the bounded context.

Adding new classes: This can also be the case if the user decides to create a new model from scratch. A new element of the three types *Entity*, *Value Object*, *Aggregate Root* or *Service* can be created.

Removing classes: In case an element is not required anymore, this particular element can be removed from the model.

Change type: After the import from the database, a dedicated entity should become the element root; therefore, it should be possible to change the type of an element. Another case is to change an entity to a value object as in figure 4.3.

4.3.2 Properties

All the properties have to be checked. The type in the database model might be a different one than the user expects. *NUMBER* is an example of such a type because it can be both a floating point number or an integer number. Besides the type, the name

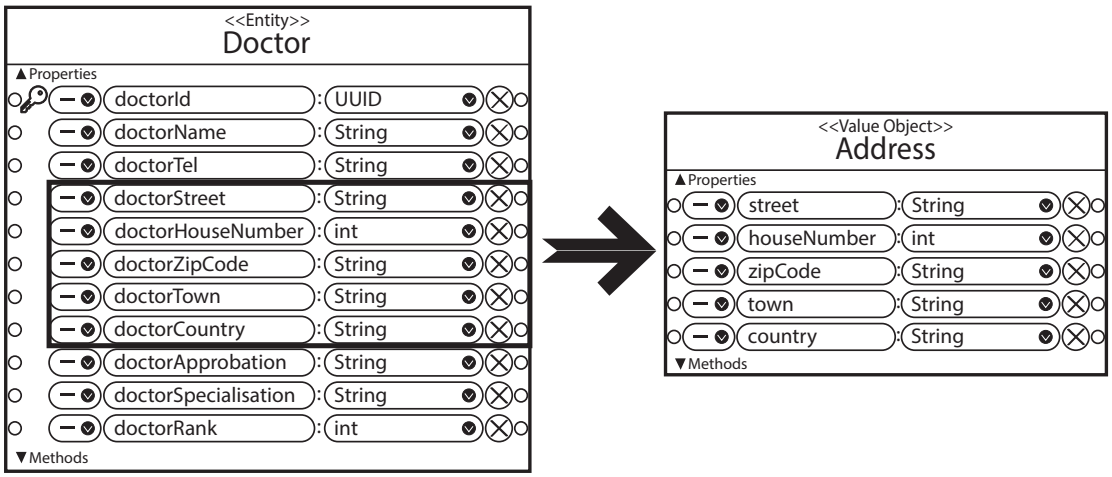


Figure 4.4: *Properties extraction* example

can also be changed. Additionally, it is possible to add other properties or remove some of them.

New properties: It should be possible to add new properties with a simple plus button. Both imported classes and new classes support this function.

Property modifications: Besides adding and removing properties, it is possible to modify each part of the property. This also means that the name from in the database form (underline case) should be changed to the standard Java notation (camel case).

Property extraction: The columns of the database columns can grow in size, and after the import, the user will see a large number of properties in the entity. Keeping an eye on it might become difficult. A solution might be to create another entity or a value object out of a group of properties. Extraction from an entity to a new value object can be done, for example, the case in figure 4.4.

4.3.3 Methods

Besides the properties, the user can define the headers of methods. The parts of them are visibility, name, return type and also the parameters. With these arguments, the method header is defined, but the method body has to be defined by the user after the export in the Java class. The idea is to declare simple CRUD (Create, Read, Update, Delete) methods for a class, ensuring the encapsulation and the access to the properties. They should be created automatically via all the current properties as the example in figure 4.5.

It is not recommended to create only getter and setter methods because the model should not become an anaemic model again. All the access with the class should be with behaviour functionality that is defined in sentences. An example of such behaviour is "The customer arranges an appointment with a doctor". The method for this behaviour is `arrangeAppointment(Customer customer, Doctor doctor, Date time)`.

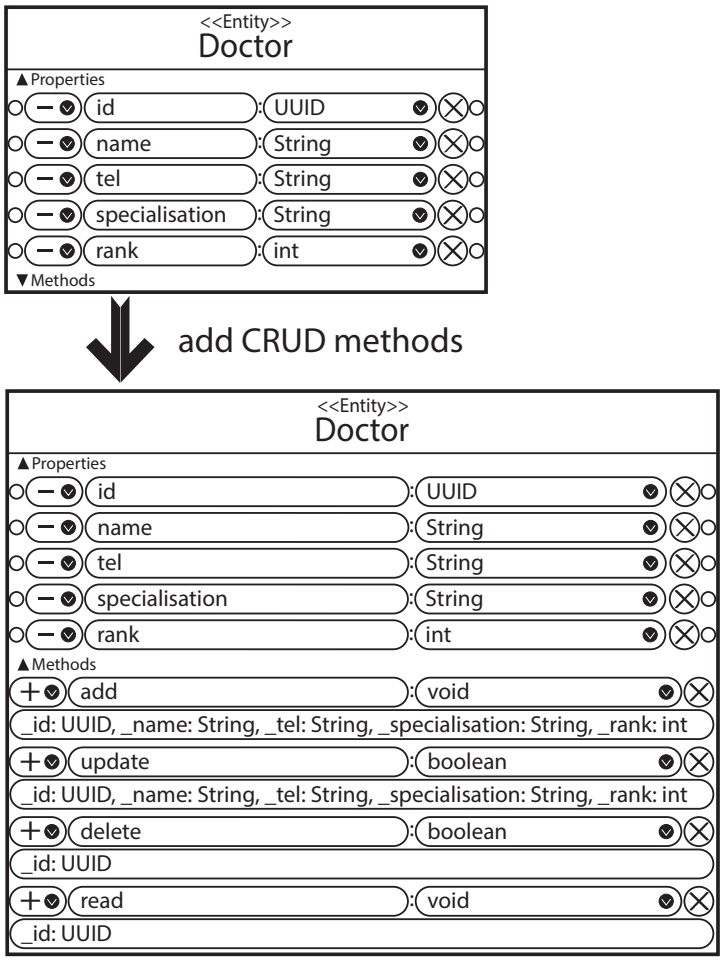


Figure 4.5: *CRUD* methods example

Additionally, most methods should be in a package or protected visibility because the aggregate root and the services should provide the functionality to the outside to ensure a side-effect free functionality.

From the database, only the properties, relations between properties and other classes are loaded. That is why the user can declare new methods of the element, including the parameters.

4.3.4 Bounded Context

All the elements have to be organised in at least one bounded context. This means an encapsulation of the class elements by avoiding too many dependencies to elements of other bounded contexts. For DDD, it helps to avoid unwanted side-effects after any modification. For a start, all elements can be put in the same bounded context, but the user should think about this separation. It might be possible that a particular element

has to be used in more than one bounded context. Replication should be avoided by copying the same element in two or more bounded contexts. As a result, they should be put into a shared kernel [25, p. 92].

It should also be possible to move an element to another bounded context or even copy this element to be used in a different version. An example is the *User* class. In the *Appointment* bounded context only the name and the e-mail address have to be known. However, for payments, the system requires data such as a credit card number. This encapsulates these usages of the user, but the database table still keeps the order. If some data is required in almost all bounded contexts, this class should probably be moved to the shared kernel. This kernel stores all the classes used by more than one element of a different bounded context. It can be seen as the core part of the model. On the other hand, the core classes are less flexible to changes as they might be used in several other places of the software.

Add elements: An important operation concerning bounded context is to add new elements to it. A simple way to do so is dragging them inside the bounded context. There might be a situation when more than one element should be added or moved from another bounded context. A simple multi-select should help. In the example in figure 4.6, two elements without a context are selected and added to a new bounded context.

Move elements: It can also be the situation that an element should be in another bounded context. In one case, it should be really in this different bounded context, but in another example, the desired element should get to the shared kernel. The operation can be almost the same but with a different outcome for the exported Java project. It can be done with merely dragging the element to another bounded context or via selecting this operation in the context menu.

Copy elements: In some situations, a similar class should be used in another context and retyping all the properties and methods can be much work. A simple copy operation should make this faster. This can be done in the context menu deciding to which bounded context the element should be copied. As in figure 4.7, an entity is copied from bounded context 1 to bounded context 2.

4.3.5 Relations

A relation means that property uses the type of another element in the diagram. Relations across the application should be kept to a minimum and used with the services.

Before moving and configuring the relation, all the class elements of the application have to be added to the type dropdown element. If the user selects a type from this dropdown element or enters the name of the other element, the relation will be created as a drawn arrow from the property to the destination class.

Besides that, there are two types of relations concerning multiplicity. Typically it uses a one-to-many relation, meaning that only one element is used in the other class, but there are several decisions which element is going to be used. The other relation type comes up with collections (e.g. list, array) with more than one element being stored in the property of the starting class. This is a many-to-many relationship. Several elements can be used, and there are still the decisions which element should be added to this collection.

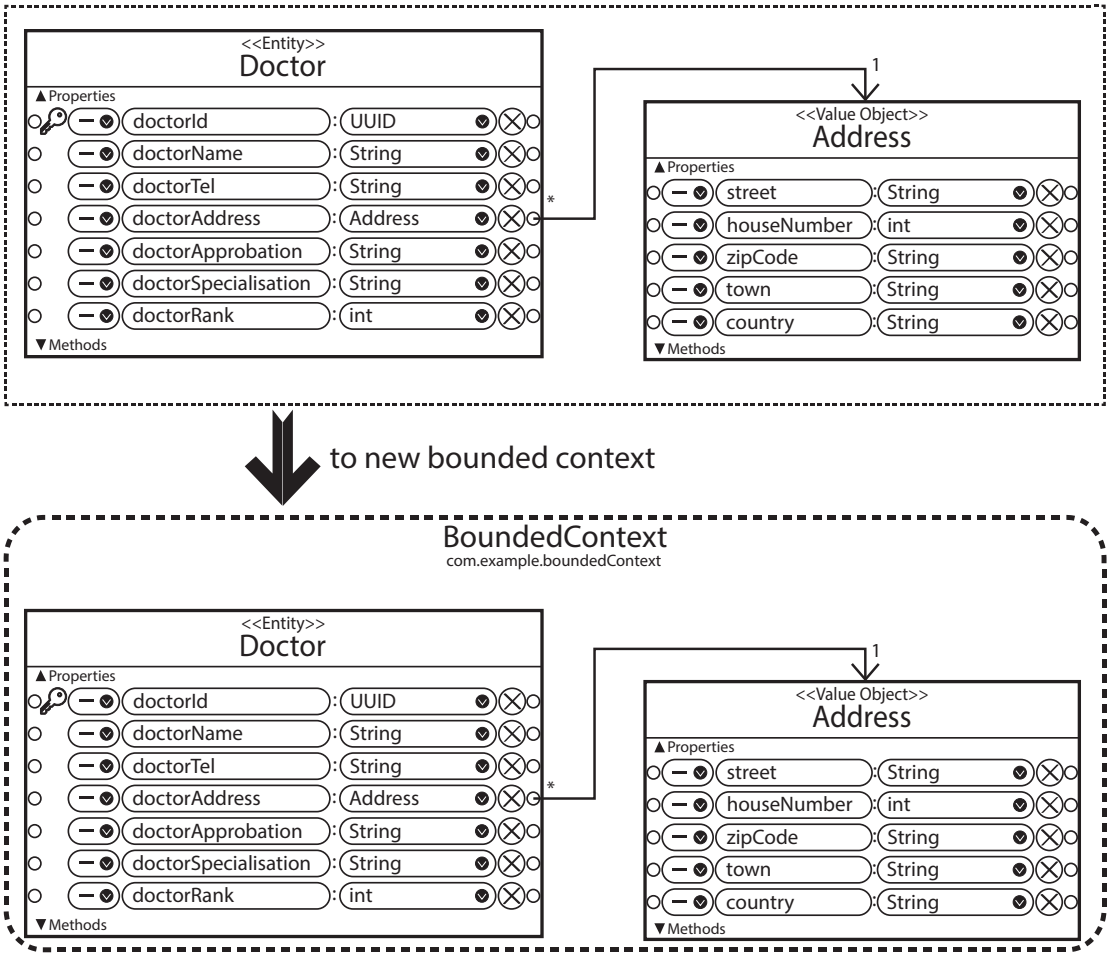


Figure 4.6: Add to new *Bounded Context*

4.3.6 Ubiquitous Language

Another key aspect of the DDD approach is to create a ubiquitous language across each project team working with the model. However, supporting the creation and development of it is difficult to handle. Nevertheless, each element can store notes in the property table. It ensures to keep these notes about the ubiquitous language for each element. After the export, all these notes will be added as comments to the specified element keeping it still in the model.

Usually, the ubiquitous language has already been decided across the project team. With these notes, all the decision can be written down to remember them later. For example, why this class has been called with this name or why there are these properties. The shared kernel should have a higher priority because a change here can cause more changes than other bounded contexts.

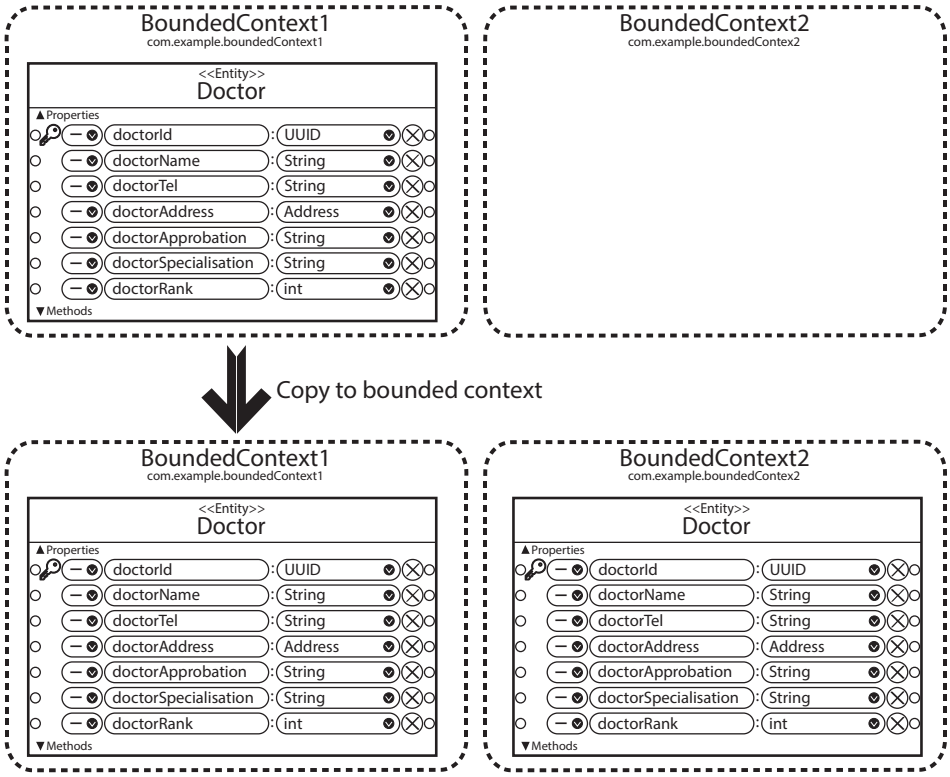


Figure 4.7: Copy to *Bounded Context*

4.4 Validation

Very important for the export is that the model is valid for this operation. Several names, types and relations have to be considered for this validation.

Each name of bounded contexts must be unique and all elements within it. This is even more important for the package name as two equally named packages are also considered equal. In case of a duplicated name, the context name or the package name has to be tagged with a red colour and a tooltip text telling the user about the validation error.

All the classes, including the entities, value objects, aggregate roots and services, have to be checked. This means the names of each class as well as its contents. The name has to be unique in each package, and all the characters must be valid. This should comply with the rules of naming variables and classes in Java. For example, only alphanumeric letters may be used without the usage of a keyword and each variable has to start with a letter. It can be checked with a regular expression and checking if the name is not equal to a keyword.

The types of these properties and methods have to be checked. All the types can be either selected via a dropdown or via text input. If the entered text is also visible in the dropdown, it can be seen as a valid type. In case of a custom type, it is difficult to say if this is a valid one. After the export, another library might be imported using some of

these types. In case of the project, it should reduce the error rate with this validation and therefore it is considered as an error but, in this particular case, the user can select the `Object` type to use a different type later.

The start and end points of the relations are also checked. If a relation line is drawn across more than one bounded context accessing a class not being an aggregate root, it will be considered as a violation error. Services should be used. However, the relations only show the dependencies of properties and relations of methods are not shown. That is why the relations to services will not be shown here.

4.5 Export

After the refactoring process follows the export process with the creation of a new Java model, the database representation with SQL statements and a repository for the connection between them. The export format should be a Java project as an archive. As a result, a basic zip file already containing the relevant information for the Eclipse IDE and the Java build path is loaded. This archive has already been generated by the IDE before and lies in the resources folder. It also includes the repository.

The user sees all the export information in a dialogue window. An example can be seen in figure 4.8. The Java project export requires the output file path, and the database requires all the connection information. These are the connection string, username and password. The export process itself starts after the user clicks the start button.

The export process starts with the Java project. All the necessary data from the graphical representation has to be loaded and converted to a Java source code form. This process is done for every single entity, value object, aggregate root and service. After that, each class will be added to a specified package folder of the zip archive. Entities and aggregates should be a subclass of a generic *Entity* class that creates and handles the operations in the repository. Additionally, another generic class is generated to support *Value Objects*. Both of them should support the import and export of all the containing properties enabling access through the repository. Besides that, the main difference between entities and value objects is that all elements in the value object should not be modified having the same state all the time and changes will result in a new instance.

The types of properties are automatically added as they are written in the graphical representation. Additionally, the properties containing a relation need to store the full path of the related one. This means that the full package name is written before each type.

After that, Java export is finished. The zip archive will be stored where the user has defined it. This is also the only setting required for the Java export. The structure of the finished zip files is similar to the one in figure 4.9. All the other information is used for the database export that is the following step.

The export to the database should also be done in a programmed function that creates SQL modification statements. In particular, these are statements for creating or replacing the tables in the database. However, all the required tables may have to be deleted before the actual creation. The user should decide whether this is required. For that reason, the user sees the full text of all the SQL statements and has to press the *Start* button to start the export process.

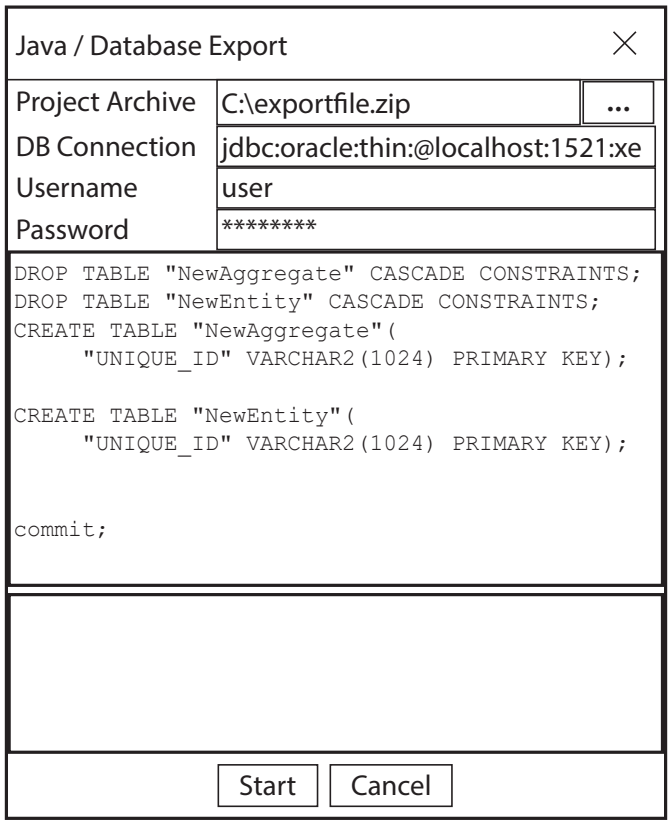


Figure 4.8: Java/Database Export Dialogue

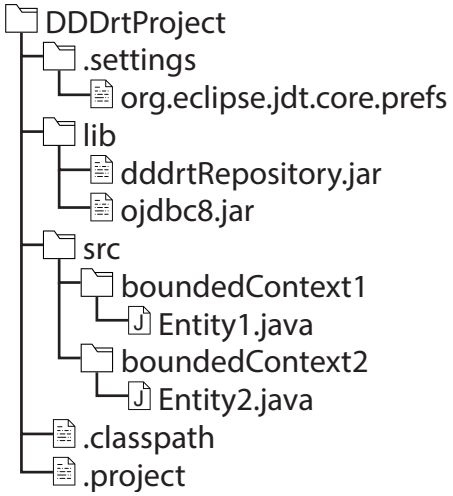


Figure 4.9: Exported ZIP Structure

Also meaningful is the connection between the database and the Java model. For this purpose, the repository has to know which table and column are responsible for a particular Java class and property. A solution for this might be annotations giving information about the database table/column. The repository can load it and build the SQL statements with this information.

Chapter 5

Implementation

The first part of the project is the integration with *UMLet* described in section 5.1. It uses the DDD class diagrams, and section 4.1 describes their structure. Classes can be either created from scratch or via an import from a database model, mentioned in section 4.2. The resulting SQL statements with the mapping are shown in section 5.2. After the import, the refactoring methods were implemented. These are the methods that are listed in section 4.3. The resulting code with some screenshots can be found in section 5.3. Although the restriction in the model creation and modification process, a validation is executed after any operation and modification. A part of the validation process is shown in section 5.4. A validation error results in the background colour change to red for the particular element. After successful validation, the project can be exported. This is described in section 5.5.

5.1 Integration with UMLet

In general, the integration is all the modification that is necessary to convert UMLet to DDDrt. The original idea behind UMLet is to quickly draw a UML diagram, including the textual representation for both the attributes and texts inside elements. The design here is in the foreground here, whereby, creating code out of the model, has not been considered at all. All the designing elements have a standard class called `NewGridElement`. The original software supports the creation of customised elements. However, all the elements do not support any direct text input such as text fields or combo boxes. That is why a new a connection from the `NewGridElement` to the `ComponentSwing` is necessary. The `ComponentSwing` is an extension of `JComponent` and allows to add these direct input elements. The reason for this extension is that it is also possible to export the current version of UMLet as an Eclipse plugin but, in this case, the standalone version is adequate. For each case, there is an own particular class extending from the `GridElement` class. The `NewGridElement` class is used in the standalone version.

The new parts of the tool are the bounded contexts as a package container and the field composite for the classes, whereby, all the four DDD class types extend from the field composite. These four subclasses here are aggregate root, entity, service and value object. Each of them consists of two collapsible panels for both the properties and methods, except for services because they do not have properties. Almost always

using the same superclass for all of these elements helped for the modifications and extension of them. Also, the properties and methods are considered to have common elements because only value objects are different here. Bounded contexts were quite different from all the class elements because they are a container. However, they are not supported by UMLet. Package diagram elements are used for the bounded contexts. Nevertheless, any movement operation of the package will not change the position of the containing elements making this a required extension. An example of a diagram with the four DDD class types in the bounded context `Core` can be seen in figure 5.1.

All of the class elements (Field Composites) use the superclass `NewGridElement` having the abstract method `drawCommonContent` where all the drawing happens. The new field composite adjusts all the position and lines of it, and the framework handles the drawing of them. Besides that, the elements should appear in the toolbox on the right side, and the definition of them has been written in particular UXF files in the `palettes` folder, located within the project directory. All the available elements are defined there, For new DDD elements, a new UXF file (`Domain-Driven Design.uxf`) has to be added as separate toolbox category. An example of the DDD entity class:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<diagram program="umlet" version="13.3">
  <zoom_level>10</zoom_level>
  <element>
    <id>DDDEntity</id>
    <coordinates>
      <x>320</x>
      <y>0</y>
      <w>300</w>
      <h>218</h>
    </coordinates>
    <panel_attributes></panel_attributes>
    <additional_attributes></additional_attributes>
  </element>
</diagram>
```

After that, the registration of the ID (here DDDEntity) must be added in the `com.baselet.control.enums.ElementId` enum. The new class has to return this ID in the `getId()` function. The last step is the integration of the new element in the `ElementFactory` class by adding a new CASE entry.

```
protected static NewGridElement createAssociatedGridElement(ElementId id) {
    switch (id) {
        ...
        case UMLClass:
            return new Class();
        //New DDD Entity element
        case DDDEntity:
            return new EntityComposite();
        ...
    }
}
```

From now on the new DDDEntity element is visible in the `Domain-Driven Design` toolbox and elements can be dragged from it to the main drawing area.

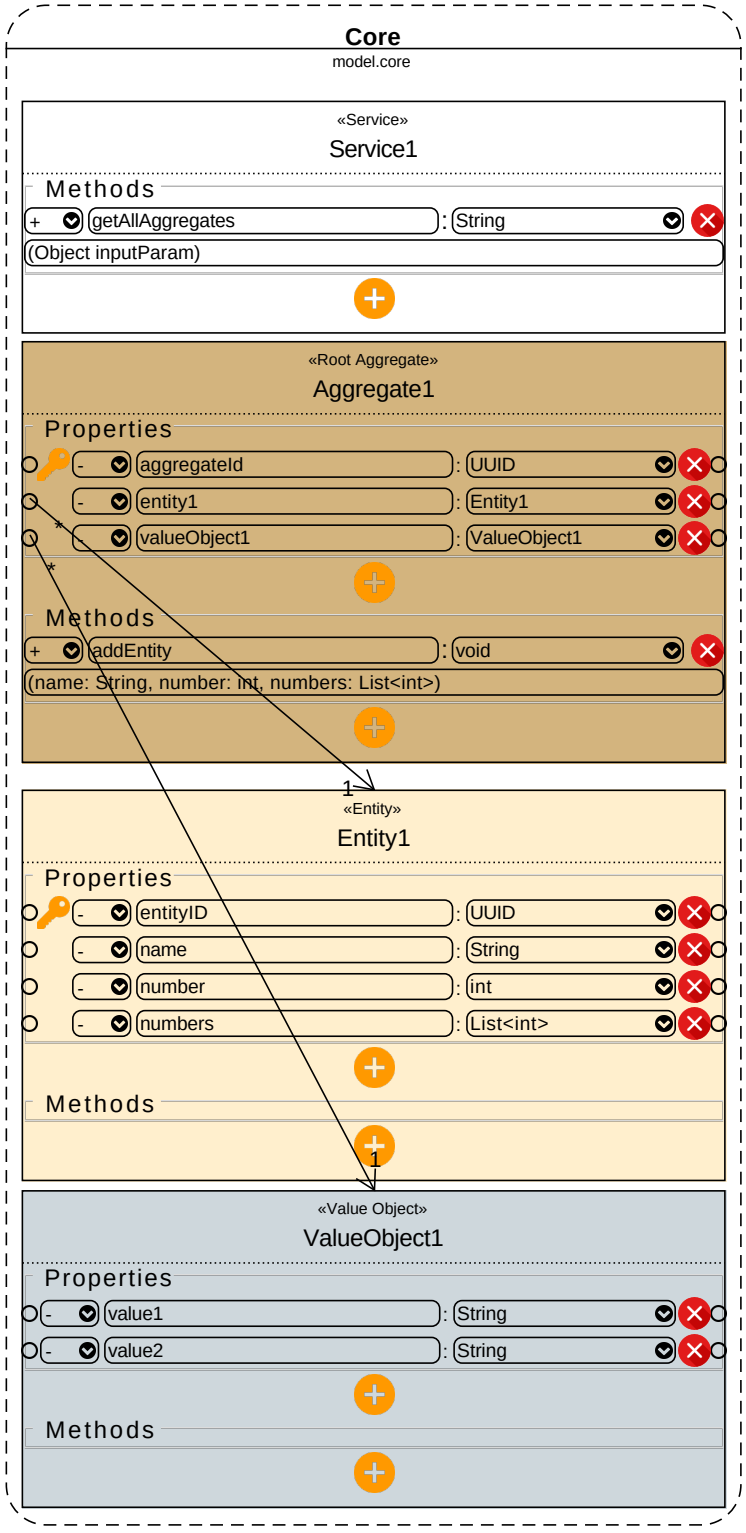


Figure 5.1: UML example diagram with the four DDD class types

5.2 Import a Database Model

The database most likely stores only the model and it might be difficult to define where the model is in the Java application. That is why importing a database should be considered.

The import takes two steps as described in section 4.2. Starting with the model information, in particular, the tables and its columns. After that, the constraints for the relations are loaded. The database itself stores more tables than a default user can see. They are called meta-tables, and in an Oracle database, they can be used for gathering information from the tables, column and constraints [2].

ALL_TABLES: It stores information about all tables accessible by the user.

ALL_TAB_COLUMNS: Every table contains columns. All the columns of tables, views and clusters are stored here referencing the name of the parent element (e.g. name of the table).

ALL_CONSTRAINTS: All the constraints are shown here, such as *Primary Key* and *Foreign Key*.

ALL_CONS_COLUMNS: The information of this view are the references to the column and the name of the constraint. This name is equal to the name in `ALL_CONSTRAINTS`.

With these four meta-tables, it is possible to retrieve the information needed for the class diagram and converting it to DDD. The import from the database in the DDDrt is performed via a `JDBC` connection with `SELECT` statements from these meta-tables. The table names are loaded via this statement on the `ALL_TABLES` meta-table.

```
1 SELECT
2     table_name
3 FROM all_tables
4 WHERE
5     owner='__USERNAME__' AND
6     tablespace_name IS NOT NULL AND
7     table_name not like '%$%'
8 ORDER BY
9     TABLE_NAME asc;
```

Retrieving the columns gets more sophisticated because the primary key is loaded in the same request; however, this requires a join of three meta-tables. The first one is for the central column information in the `ALL_TAB_COLUMNS`. The second one is the `ALL_CONS_COLUMNS`, and `ALL_CONSTRAINTS`, which are used to get the constraints for the column of this table but, in this case, only to get the primary key.

```
1 SELECT
2     a.column_name as COLUMN_NAME,
3     a.data_type as COLUMN_DATA_TYPE,
4     a.data_length as COLUMN_DATA_LENGTH,
5     a.nullable AS COLUMN_NULLABLE,
6     CASE WHEN c.constraint_type = 'P' THEN 'Y' ELSE 'N' END as
7         PRIMARY_KEY
8 FROM ALL_TAB_COLUMNS a
9 LEFT OUTER JOIN
10    all_cons_columns b ON a.owner = b.owner AND
11    a.table_name = b.table_name AND
```

```
12     a.column_name = b.column_name
13 LEFT OUTER JOIN all_constraints c ON b.owner = c.owner AND
14     b.constraint_name = c.constraint_name
15 WHERE
16     a.OWNER = 'AFACI' AND
17     a.TABLE_NAME = 'SPACESHIP'
18 ORDER BY
19     a.column_id ASC;
```

Similar to [24], the second part of reverse engineering is to load the relations. They are gathered and recreated on the DDD model. The statement uses three meta-tables again. The `ALL_CONS_COLUMNS` table is used to see the columns with constraints with a join to the `ALL_CONSTRAINTS` table to figure out the related tables and columns.

```
1  SELECT
2      a.table_name start_table,
3      a.column_name start_column,
4      b.table_name end_table,
5      b.column_name end_column
6  FROM all_cons_columns a
7  JOIN
8      all_constraints c ON a.owner = c.owner AND
9      a.constraint_name = c.constraint_name
10 JOIN
11     all_cons_columns b ON c.owner = b.owner AND
12     c.r_constraint_name = b.constraint_name
13 WHERE
14    a.owner = '__OWNER__'
15    AND a.table_name = '__TABLE_NAME__';
```

With all these statements, the entities and relations are loaded and are converted to DDD classes. This, in particular, means the names of the classes itself, the property names as well as property types. The type conversion is described in the table 4.1. In particular, the import considers the number type while all the other types are converted to a string, except for the `DATE` type.

5.3 Refactoring

The main purpose of the application is the conversion of a typical database generated model to a DDD model. This process consists of various steps, where each step is in a specific section of DDD. These sections are classes, properties, methods, bounded contexts and relations. One section should not influence another one. For example, changing a property of a class should not change the surrounding bounded context.

5.3.1 Classes

A class has to be one of four types. These types are entities, value objects, aggregate roots and services. The refactoring methods, as mentioned in section 4.3.1, consists of the class being added, removed or changed. Another part is to change the type of the class.

The UMLet tool itself already supports the add and remove operations and therefore, does not require adjustments. The focus here lies in changing the type of the

element because database tables are usually entities, and the user may want to change one of them to an aggregate root or a value object. In the graphical user interface the changed element looks quite similar to the original one, as in figure 4.3. However, the operation behind it is more sophisticated because the operation loads the data from the first class and creates a new class by calling the static creation method of the `ElementFactorySwing` class, providing the original rectangle (location, size), the panel attributes, the additional attributes and the UUID. In particular, the additional attributes are essential here, as they store all the element data, including all the properties and methods of the class. Besides that, the relations have to be changed, and the class has to be added again to the same bounded context. Afterwards, the last operation is to remove the original (old) element from the drawing panel. An example of the full operation with the original and new type can be seen in figure 5.2. It shows how new type changes in the type specification text on top as well as its background colour.

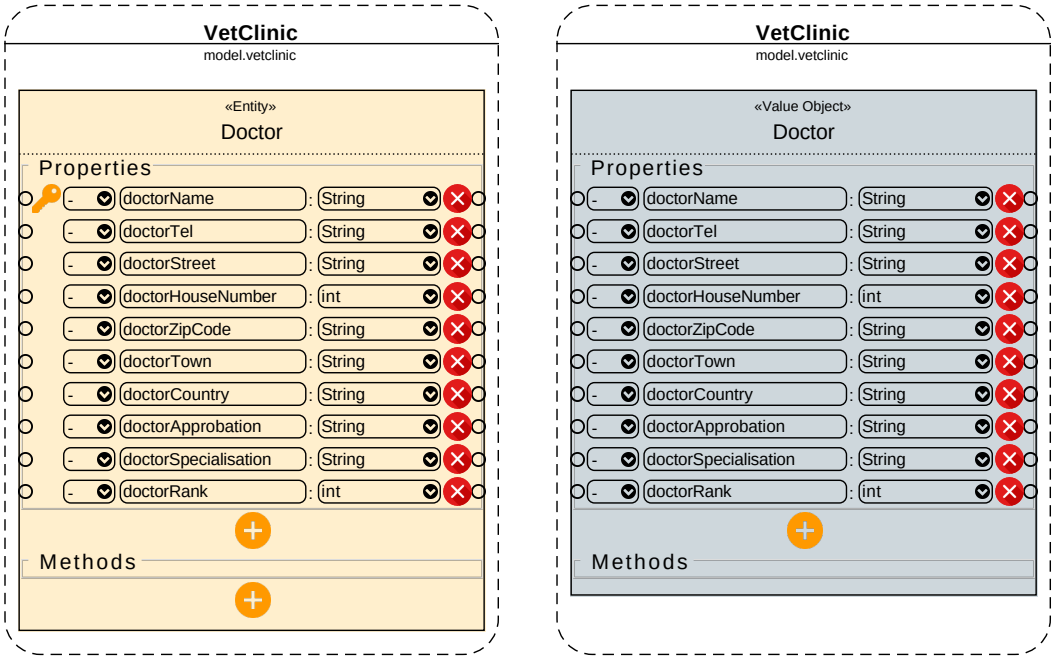
```
1 private void changeType(DiagramHandler handler, FieldComposite
      originalFieldComposite, ElementId newElementType) {
2     DrawPanel drawPanel = handler.getDrawPanel();
3     newFieldComposite = (FieldComposite) ElementFactorySwing
4             .create(
5                     newElementType,
6                     originalFieldComposite.getRectangle(),
7                     originalFieldComposite.getPanelAttributes(),
8                     originalFieldComposite.getAdditionalAttributes(),
9                     handler, originalFieldComposite.getUUID());
10    List<DDDRelation> relations = drawPanel.getRelationsOfFieldComposite(
      originalFieldComposite);
11    for (DDDRelation relation : relations) {
12        relation.changeFieldComposite(originalFieldComposite, newFieldComposite);
13    }
14    drawPanel.addElement(newFieldComposite);
15    drawPanel.removeElement(originalFieldComposite);
16    drawPanel.getSelector().deselectAll();
17    drawPanel.getSelector().select(newFieldComposite);
18    newFieldComposite.updateBoundedContext(originalFieldComposite.getBoundedContext
      ());
19 }
```

5.3.2 Properties

Each class element, except for services, contains properties. The import method converts the column names and types to the new property names and types by using the camel case notation and Java types.

All the other operations for DDD are adding new properties, modifying or removing them. The UMLet tool does not cover the handling of these operations with properties yet. It means that the project has to handle them. For the layout, both properties and methods are shown in a collapsible panel using a grid layout within this panel. The advantage of this collapsible panel is to have a general overview of all class elements without the properties and methods, but they can be opened if required.

After opening a panel, properties can be modified inline. Besides that, properties can be selected to perform modifications in the properties table on the right-bottom. With the plus button on top of the panel, it is possible to add a new property, and the



(a) Original Type (b) New Type

Figure 5.2: Change Type

remove button on the right side of each property removes it.

Coming to the selections, the user can select multiple properties holding down the control key, and extract them to a new value object. It helps to reduce the number of properties for an element.

The function for the extraction takes the selected properties from the class, creates a new value object with the element factory and adds the selected properties to it. After that, a relation property has to be created in the original entity to keep the connection between the original element and the new value object. This relation property in the entity is called `extractedValueObject` and the name of the value object `ExtractedValueObject`, which is also the type of the property. Afterwards, the connection is created between them. The example in figure 5.3 shows the original doctor with the extraction of the address, being a new value object.

```
1 private void extractPropertiesToValueObject(final FieldComposite fieldComp, final
      List<FieldProperty> properties) {
2     Rectangle rect = fieldComp.getRectangle();
3     rect.y += 10;
4     FieldComposite valueObjectComp = (FieldComposite) ElementFactorySwing.create(
5             ElementId.DDDValueObject,
6             rect,
7             "",
8             null,
9             CurrentDiagram.getInstance().getDiagramHandler(),
10            null);
```

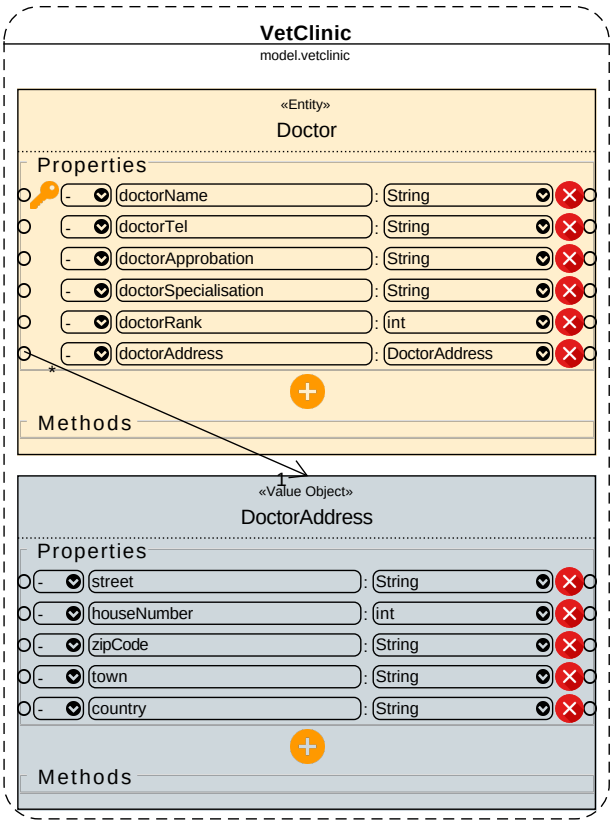


Figure 5.3: Extraction of an Entity

```
11      DiagramHandler handler = CurrentDiagram.getInstance().getDiagramHandler();
12      DrawPanel drawPanel = handler.getDrawPanel();
13      drawPanel.addElement(valueObjectComp);
14      valueObjectComp.removeAllFieldProperties();
15      valueObjectComp.addFieldProperties(properties);
16      fieldComp.removeFieldProperties(properties);
17      valueObjectComp.setName("ExtractedValueObject");
18      FieldProperty startProperty = EntityProperty.createFromName("
        extractedValueObject", "ExtractedValueObject");
19      fieldComp.addFieldProperty(startProperty);
20      createRelation(valueObjectComp, drawPanel, startProperty);
21      updateBoundedContext(fieldComp, valueObjectComp);
22  }
```

5.3.3 Methods

The methods are displayed directly below the properties with almost the same structure. The only difference here is that a list of parameters can be typed-in providing method headers for the classes. Relations are also not shown as these relations are usually handled in the method body (implemented code) afterwards. The view here is also in a

collapsible panel that is in the collapsed state per default after its creation.

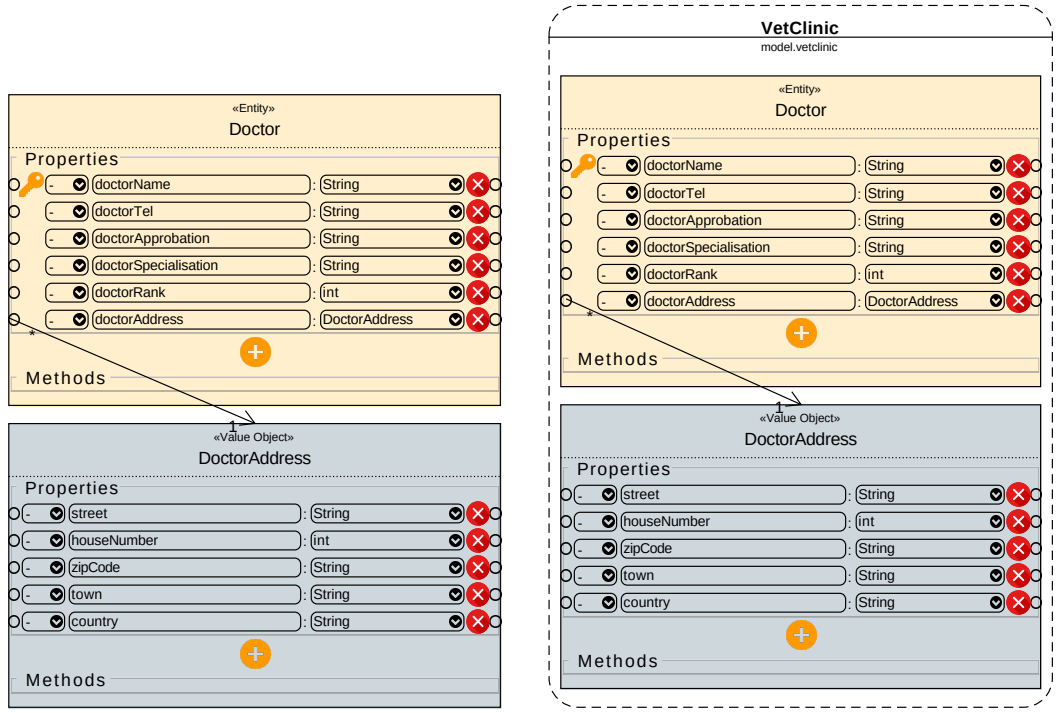
The methods should define the behaviour of the underlying class with its properties and might change their state. However, only the aggregate root should call these methods. A typical case is the usage of CRUD methods (`CREATE, READ, UPDATE, DELETE`) modifying the entity except for the read method. These headers are created with the `addCRUDMethods` operations started after pressing the context menu entry.

```
1 private void createCRUDMethods(final FieldComposite fieldComp) {
2     FieldMethod.Builder readMethod = new FieldMethod.Builder("#", "read", fieldComp.
      getName());
3     FieldMethod.Builder deleteMethod = new FieldMethod.Builder("#", "delete", "void"
      );
4     FieldMethod.Builder createMethod = new FieldMethod.Builder("#", "create", "void"
      );
5     FieldMethod.Builder updateMethod = new FieldMethod.Builder("#", "update", "void"
      );
6     FieldProperty idProperty = fieldComp.getIDProperty();
7     if (idProperty != null) {
8         readMethod.addParameter(idProperty.getPropertyType(), idProperty.
      getPropertyName());
9         deleteMethod.addParameter(idProperty.getPropertyType(), idProperty.
      getPropertyName());
10    }
11    for (ExportProperty property : fieldComp.getProperties()) {
12        createMethod.addParameter(property.getType(), property.getName());
13        updateMethod.addParameter(property.getType(), property.getName());
14    }
15    fieldComp.addMethod(createMethod.build());
16    fieldComp.addMethod(updateMethod.build());
17    fieldComp.addMethod(deleteMethod.build());
18    fieldComp.addMethod(readMethod.build());
19    fieldComp.updateModelFromText();
20 }
```

5.3.4 Bounded Context

The group where most of the communications and relations should happen is within these bounded contexts. The typical refactoring steps here are the creation, removal and changing the name or package name of it. Moreover, the more widely used operations here are for the class elements. For example, adding them or removing them from a bounded context. It should also be possible to copy a class element to another context. Two operations are available from the context menu for both moving the element to a bounded context and copying it. This function described here is for the move operation. The menu item is created with a loop over all bounded contexts to provide a selection of them. The user chooses the desired bounded context, and with the selection of it, the `actionPerformed` method is called. It forwards the event to the `moveSelectionTo(BoundedContext bc)` method, and this one moves the selected class elements to the selected bounded context.

```
1 private void moveSelectionTo(final BoundedContext bc) {
2         DiagramHandler handler = CurrentDiagram.getInstance().getDiagramHandler();
3         int startY = 60;
4         int defaultElementHeight = 120;
```



(a) without bounded context (b) with bounded context

Figure 5.4: Select elements and create them in a bounded context

```
5         startY = bc.organiseBoundedContextElements(startY);
6
7         List<GridElement> selection = handler.getDrawPanel().getSelector().
      getSelectedElements();
8
9         for (int i = 0; i < selection.size(); i++) {
10            GridElement copy = selection.get(i);
11            int width = bc.getRectangle().width;
12            Rectangle rect = copy.getRectangle();
13            rect.x = bc.getRectangle().x + 10;
14            rect.y = bc.getRectangle().y + startY;
15            startY += defaultElementHeight;
16            rect.width = width - 20;
17            copy.setRectangle(rect);
18            copy.dragEnd();
19            copy.updateModelFromText();
20        }
21    }
```

The copy operation is similar, but with the difference that classes are copied before the move operation. Additionally, it is also possible to add the classes to a new bounded context, but this requires its creation in advance. The example in figure 5.4 shows how to create the new bounded context.

The method called `organiseBoundedContextElements` creates the structure for all

bounded contexts to be in a row beside each other. This newly created bounded context will be the furthest right one.

5.3.5 Relations

The usage of a type in the property creates a relation. It means that this class is used within another class either in a one-to-many or a many-to-many relationship. The first step is to add all the available types to the dropdown selection of the property. If the user selects a relation type, it will create the relation by drawing the arrow from the property to the destination class. The create relation method `createRelation` needs the starting property, end class (FieldComposite) and if the relationship is a one-to-many (single type) or a many-to-many relationships (collection type). The `DDDRelation` is a new element type that extends from the `Relation` type from UMLet. The original relation type needs the surrounding rectangle with absolute positions. This rectangle knows two corners at the start and the end point of the relation. The calculation of the parameters for the full rectangle works via the more left and higher point and adding the width and height to the more right and lower point. This class also requires the arrow type, including the multiplicities in the property string or also called the relation text. The new relation is created via the default factory class `ElementFactorySwing`. Afterwards, this new class requires both the start field property and the destination field composite for future calculations after any position change. The method for updates of the positions of the relation is the `createRelationLine` method.

```
1 public static DDDRelation createRelation(FieldProperty startProperty, FieldComposite
       endComposite, boolean manyToManyRelation) {
2     java.awt.Point startPoint = startProperty.getAbsolutePosition(false);
3     java.awt.Point endPoint = endComposite.getAbsolutePosition(false);
4     if (startPoint.x < endPoint.x) {
5         startPoint = startProperty.getAbsolutePosition(true);
6     }
7     if (startPoint.y > endPoint.y + endComposite.getRectangle().height / 2) {
8         endPoint = endComposite.getAbsolutePosition(true);
9     }
10    Rectangle boundingRectangle = createBoundingRectangle(startPoint, endPoint);
11    DDDRelation dddRelation = (DDDRelation) ElementFactorySwing.create(
12            ElementId.DDDRelation,
13            boundingRectangle,
14            manyToManyRelation ? "lt=<-\nm1=*\nm2=*" : "lt=<-\nm1=1\nm2=*",
15            null,
16            CurrentDiagram.getInstance().getDiagramHandler(),
17            null);
18    dddRelation.startProperty = startProperty;
19    dddRelation.endComposite = endComposite;
20    dddRelation.collection = manyToManyRelation;
21    dddRelation.createRelationLine();
22    return dddRelation;
23 }
```

5.4 Validation

Before the export, the validation has to be done. In particular, the checks performed here are using names without any keyword and duplication, but also includes the type check and checking that every class is within a bounded context.

The most sophisticated check might be the parameters of the methods because they can be entered freely. This check also includes the parsing of the parameters as they will be used later for generating the method header. As for UML, the default writing style is `name":" type` but in Java it is `type name`. The parser here allows both of them. First of all, the whole string is split by the commas because they separate every single parameter. After that, it checks if there is a colon present. If true, the parameter is split by the colon and the text before it is the name and after it is the type. If false, the string splits by a blank space and the first part is the type and the second part is the name. In both cases, the resulting types and names are removed from blank spaces and the process checks if there are exactly two text elements after the split. Otherwise, it throws an exception. The validation catches all exceptions and might return a false validation state.

```
1 public String validateParameters() {
2     try {
3         Set<String> names = new HashSet<String>();
4         for (Parameter parameter : parseParameters()) {
5             boolean unique = names.add(parameter.name);
6             if (!unique) {
7                 throw new Exception(parameter.name + " has duplicates. Please rename
       them.");
8             }
9         }
10        return null;
11    } catch (Exception ex) {
12        return ex.getMessage();
13    }
14 }
```

The validation check also includes more general issues. The property names are checked for their uniqueness considering all the other properties within the same class. For this purpose, hash maps are used with the name as a key element and the field composite itself as a value element. This is also true for the field composites because they use the same check methods within the same bounded context. A bounded context should be a package with a unique bounded context name and package name.

Another aspect of the validation is the time and the frequency of its execution. In the case of the export, it is essential to do a complete validation before the export to the database and the Java model. Naming might cause a problem in the creation, and therefore, the execution has to be prohibited. Moreover, the tool also helps to check the naming even before the export. After a keyboard type in a naming field, it checks the current naming and marks the text field red in case of an invalid name.

The method for checking the names is implemented in the `validateElementNames` method and uses two hash maps because the database column names are validated at the same time. A duplicated name here means that this value has already been put in the map returning its value in the put method. The property method `setNameValidity`

uses this return value and handles the error message with the decoration of the text field. In case of an invalid state, the field adds a red frame and a tooltip text showing the error message. Figure 5.5 shows some typical validation errors.

```
1 public boolean validateElementNames() {
2     boolean validationState = true;
3     HashMap<String, FieldProperty> propertyNames = new HashMap<String, FieldProperty
      >();
4     HashMap<String, FieldProperty> databaseNames = new HashMap<String, FieldProperty
      >();
5     for (java.awt.Component comp : propertiesPane.getComponents()) {
6         if (comp instanceof FieldProperty) {
7             FieldProperty fieldProperty = (FieldProperty) comp;
8             FieldProperty previous = propertyNames.put(fieldProperty.getPropertyName
      (), fieldProperty);
9             FieldProperty previousDB = databaseNames.put(fieldProperty.
      getDatabaseName(), fieldProperty);
10            boolean newValidation = fieldProperty.setNameValidity(previous,
      previousDB);
11            if (!newValidation) {
12                validationState = false;
13            }
14            newValidation = fieldProperty.validateType();
15            if (!newValidation) {
16                validationState = false;
17            }
18        }
19    }
20    HashMap<String, FieldMethod> methodNames = new HashMap<String, FieldMethod>();
21    for (java.awt.Component comp : methodsPane.getComponents()) {
22        if (comp instanceof FieldMethod) {
23            FieldMethod fieldMethod = (FieldMethod) comp;
24            FieldMethod previous = methodNames.put(fieldMethod.getMethodName(),
      fieldMethod);
25            boolean newValidation = fieldMethod.setNameValidity(previous);
26            if (!newValidation) {
27                validationState = false;
28            }
29        }
30    }
31    return validationState;
32 }
```

5.5 Export

After successful validation, the project is ready to be exported to the database and the Java project, including a repository.

5.5.1 Model

The model is, according to Eric Evans, the heart of the software [16]. The database also represents the model. Data types might be different. In particular *Value Objects* are stored inside the table as `String` representation.

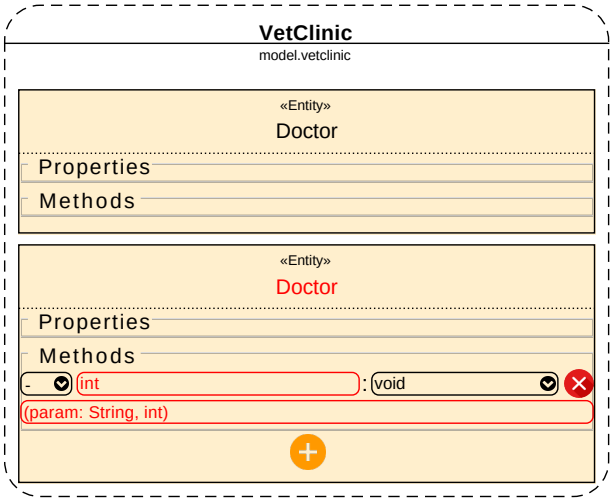


Figure 5.5: Validation Example

The properties itself can be written down sequentially with the desired visibility, type and name. Non-elementary types such as `Date` might require an additional import or they are written with the full package definition, e.g. `java.util.Date`. This is also true for the relations as they might be part of another package but should be in the same bounded context. The conversion from the Java data in this DDDrt project to real Java code representation works with the JavaParser API[1]) because it also supports exporting Java classes to real Java source code.

The packages also reveal the structure of the new project. Development IDEs use a folder structure for all the projects that can be both exported and imported via ZIP-archives. The idea is to create and export an almost empty project with only the necessary information from the IDE. The application creates a copy of it and adds the new Java source files. After the export, this archive can be imported in the IDE, and the new project can start. An example for the Eclipse IDE is in figure 4.9.

5.5.2 Database

`SQL`, used for the import statements, is a commonly used query language for databases. The import function itself only retrieves data without doing any changes. However, this is part of the *Data Manipulation Language* because all these requests only affect the data in a particular table without changing the table structure.

Nevertheless, SQL is also for the creation of the model on the database with the manipulation requests that are also part of it in the *Data Definition Language* used for changing the structure of the tables or creating new ones. Each entity or aggregate root is a table. The creation of it works with these `CREATE TABLE` requests, as it can be seen here:

```
1 CREATE TABLE entity1 (
```

[1] https://javaparser.org/

```
2     property1 VARCHAR2(1024) PRIMARY KEY,
3     property2 CLOB,
4     property3 CLOB
5 );
```

Each entity requires a table name, and all the properties need a name and a type. The *Primary Keys* or *Foreign keys* do not have to be set inside this statement because a statement can create them later. Nevertheless, foreign keys need primary keys.

SQL supports the alter statement after the creation of the table. In this case, it is about adding a foreign key to the table.

```
1 ALTER TABLE AGGREGATE_1
2 ADD CONSTRAINT fk_startentity_endentity
3     FOREIGN KEY (startentity)
4     REFERENCES endentity (id_endentity);
```

The foreign key requires the start point and the end point of the relation. The start point requires the name of the table and the property using the reference, and the endpoint requires the name and the primary key of the table it references. Additionally, the `ALTER TABLE` statement supports even more manipulation, such as adding and removing properties.

A property containing more than one element, using a relation type, is called a many-to-many relationship. The starting property is a list or an array using the type of the related class. There are multiple methods of how to bring this relationship to the database. Typically a table is added between these two tables called an associative table. Another more straightforward approach is to store a list of foreign keys in the table column with either a separator or using a JSON-array.

5.5.3 Repository

The repository is the connection between the Java model and the database. There are already several frameworks or libraries to manage a repository. However, the concept of value objects is a distinct use-case that needs extra support because Eric Evans suggests storing them as a column of the entity table [16, p. 102]. Moreover, classes of the model should not always allow universal access via getters and setters to the properties because it should not become an anaemic model again.

In this case, a JSON is used by converting all the value objects to JSON arrays and store its string representation directly within the database column. In this particular case, all elements within the value object are added sequentially to the JSON-array, and afterwards, this JSON-array is converted to a string. The database column uses the CLOB type to store it. The loading process in the select statements reconverts this CLOB back to the JSON-array and then to the value object.

For this particular case, a new Java ORM concept has been developed using annotations and their information in JDBC statements. The main idea of a repository is connecting the database with the Java classes. All the operations such as `SELECT`, `INSERT`, `UPDATE` and `DELETE` are used to synchronise the current Java model to the database and also read the current model after the start of the application. A repository is introduced to handle these operations. This connection also requires some smaller aspects. The Java class name needs the database table name, and all the properties need to know the corresponding column name, including the database type. The storage of

the meta-data works with annotations because they provide more and detailed information. The mapping requires two different annotations. One is for the class connecting it with the table, and the other one is for the property connecting it with the database column. An example class can be seen here:

```
@DDDEntity(tableName="ENTITY1")
public class Entity1 {
    @DDDProperty(primary=true, columnName="PROPERTY1", columnType="VARCHAR2(1024)")
    private String property1;

    @DDDProperty(columnName="PROPERTY2", columnType="CLOB")
    private double property2;
}
```

The synchronisation with the database is possible via the `Repository` class in the Java project. Example code can be seen here:

```
Repository repository = new Repository(
    "jdbc:oracle:thin:@localhost:1521:xe",
    "afaci",
    "afaci");
Entity2 entity2 = new Entity2("Entity2", null);
List<Entity2> listEntity2 = new LinkedList<Entity2>();
listEntity2.add(entity2);
Entity1 entity1 = new Entity1(
    "Test1",
    listEntity2,
    new Date(System.currentTimeMillis()));
repository.update(entity1);
List<Entity1> resultList = repository.selectAll(Entity1.class);
for(Entity1 entity : resultList) {
    System.out.println(entity.toString());
}
repository.disconnect();
```

Chapter 6

Evaluation

The evaluation process was done both during and after finishing the implementation. It led to some changes in the development process. The refactoring part is particularly noteworthy because it is the primary focus of the application. This is especially true for refactoring methods. The three parts of this evaluation of the refactoring tool are the functionality, the usability, as well as the diagram layout, according to DDD. In addition, some possible extensions are listed.

6.1 Functionality

This evaluation was done with a team of five senior software developers who are experts in implementing software tools with over ten years of experience. The tool was shown on a screen with this default use case: the user imports a model from the database, converts this to a DDD model with the provided functionality, and finally, the person uses the export service to create a new Java and database model. Afterwards, a feedback talk was held about existing refactoring methods, including a discussion about new methods as a sensible complement.

The main idea of the project is to bring the software development team closer to DDD. It should convey the basic concepts such as using more value objects instead of entities, and why it is important to separate the domain in several bounded contexts. The main focus of this evaluation is the functionality, especially with regard to finding the required elements, in order to make the developer's work easier. The evaluation team is also familiar with some similar UML tools such as the *Eclipse Modeling Framework* (in section 3.2), which is beneficial for comparison. A toolbar shows icons on the left side and double-click or drag-and-drop creates the new element in the working area. However, *UMLet* has a different approach. It displays the same graphical representation in the toolbox as in the main drawing area. The feedback here was that the elements in the toolbar should be smaller as in the main drawing area, due to the limited space. That is why all DDD elements there are shown with both panels collapsed.

Furthermore, for large entities, refactoring is an essential operation. It helps to make them smaller by extracting them, creating several entities or value objects from them and then linking them together. This can be done with fundamental functions, such as adding a new entity and moving some of the properties to it.

Nonetheless, the evaluation during the development process showed that these fundamental operations take much time, and certain faster-refactoring operations were necessary. Examples are the extraction of larger classes to value objects, or selecting one or more classes that are outside of any bounded context and combining these elements in either a new or an already existing bounded context. Besides that, there are also some nice-to-have features, such as adding the default CRUD methods (create, read, update, delete) to an element.

Moreover, services are also a significant aspect of DDD, because they are part of the application layer, as well as being the interface between the domain model and the rest of the application. The full definition of services can be added later because an extensive description of services is only necessary to create them automatically. The tool only supports the basic description with the name of the service and method headers.

6.2 Usability

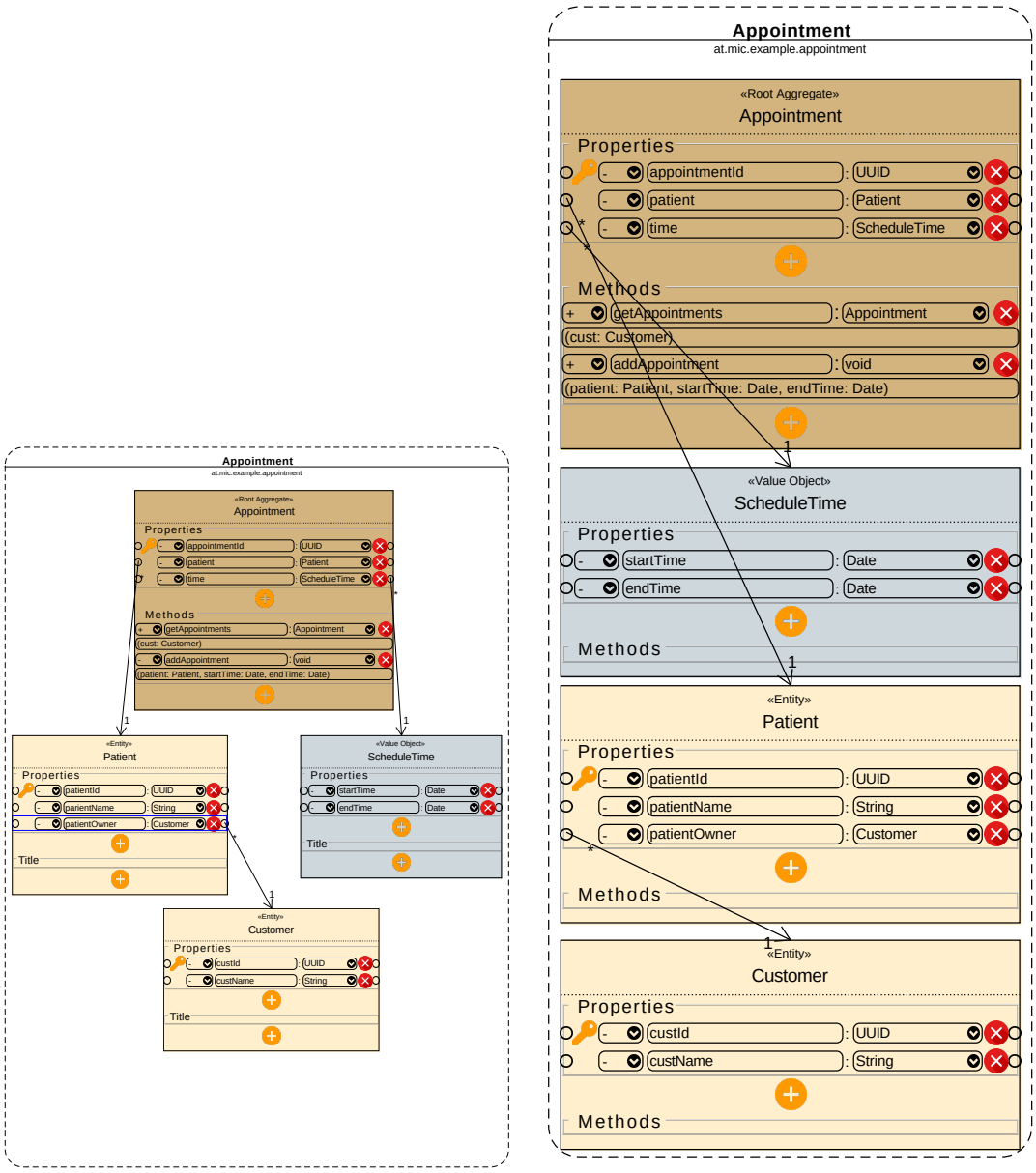
The usability evaluation was done with a UI expert. This person is not a software developer but is trained in UI design, UX design and usability.

The application should still use the UML format with class diagrams for the entities, value objects, aggregate roots and services. Package diagrams are used for the bounded contexts. In the beginning, the idea was to show the bounded contexts as ovals and the classes within them as rectangles. However, the circular shapes took too much space, and space was limited. That is why the bounded contexts changed to rounded rectangular shapes with dashed border lines because they should be distinct from the classes. The classes still use standard rectangular shapes with solid lines. A class or field composite can have one of four different types. In the beginning, the design was the same for all of the four types, with different text at the top to show the current type. However, it was difficult to see a type of change. For example, when changing from `«Entity»` to `«Aggregate Root»`, only the change of the text at the top was difficult for the user to notice. As a result, it became necessary to find a better approach to this usability aspect. The solution was to use four different background colours for each type.

Another part of the tool is the structure. In *UMLet*, the user can freely design the diagram with fewer restrictions. Although this is helpful for the design phase, it might be difficult for a code generation tool, because of code and structural limitations in Java, databases and the DDD approach itself.

Despite these limitations, the structure is still essential for giving a quick overview of the model. The location of the classes and the bounded contexts were not restricted in the version before the review. It helped to design the new Java model freely, but as the model grew to a larger size, it became more challenging to keep a structure. The solution for this was to introduce a strict structure for the locations and sizes of both the bounded contexts and the containing classes. All bounded contexts are located next to each other horizontally, and all the classes inside a bounded context are ordered vertically. The example in figure 6.1 shows the veterinarian clinic with the bounded context `Appointment`.

Due to the restricted space within the drawing area, not all elements should always be shown with the properties and method headers. There has to be a possibility to hide them, and they should be hidden after starting the application. The user should only



(a) free approach (b) structured approach

Figure 6.1: Veterinarian example

see the class names and can choose a class to open its properties and methods.

Another aspect is that the domain might also contain stories, or, even in the case of DDD, it is defined in a universal (ubiquitous) language. The design should bring this information, adding some notes to the class or property. The story here can be: "a veterinarian treats the patient (pet), and the customer owns the patient." Out of this sentence, the developer can create three classes. However, this sentence is only visible after selecting the element, and the description is only in one line. For the future, it

might be better to show this textual representation in multiple lines.

6.3 Diagram Layout

The layout evaluation was done with a DDD expert. The person knows the concept and how to use it. The layout here is important because the tool's primary subject is to help the user with the concept of DDD. The first concept of a value object might be difficult to understand because most developers with databases think of entities only. However, value objects should be unchangeable, creating a new instance for every operation similar to the String class. This aspect is shown after the export because no setters will be created, and the call of the constructor has to provide all properties. The separation of entities and value objects are also shown with different background colours and the aspect that they have to be part of an entity.

Another more specific DDD part is the grouping and encapsulation with bounded contexts. Most developers may know packages or modules. Nevertheless, bounded contexts are a bit different than them because they can contain multiple packages (described in section 2.1.4). However, smaller sized grouping might not be necessary, in particular for small and medium-sized projects which the refactoring tool supports. Each bounded context has only one module or package. The main reason for this is simply because of the limited space. These are some of the challenges a developer might face when a person starts working with DDD.

Nevertheless, each software design has a particular structure. The structure of DDD as in figure 6.2 shows the central aspect lying the in the core model with value objects and entities connected via aggregates. Repositories and factories provide access to this core domain, but the more important aspect is the interface between the domain and the application controlled via services.

6.4 Possible Extensions

There are still some extensions that are optional and useful for DDD. Services are also typical class elements in the refactoring tool. To be considered here is that the user does not see this separation in these two different layers. A solution for this problem is to show them with different background colours in the bounded context, including a hierarchy. In particular, services are put above all other classes using the sorting process in the `organiseBoundedContextElements` method. It is also helpful to provide a full definition of services to make an automated export possible.

In addition to the three main components of the domain layer, there are two more elements in this layer. These are factories, which are for creating new elements with the support of design patterns, and repositories, which provide access to infrastructure. An infrastructure can be a file system or a database.

The refactoring tool provides a repository with simple database access operations. Typical operations are inserting a new record; modifying an already existing record, and reading all the records from the database table. This single repository supports all of these and is extendable. A possible extension is to customise the repository and create one for each bounded context.

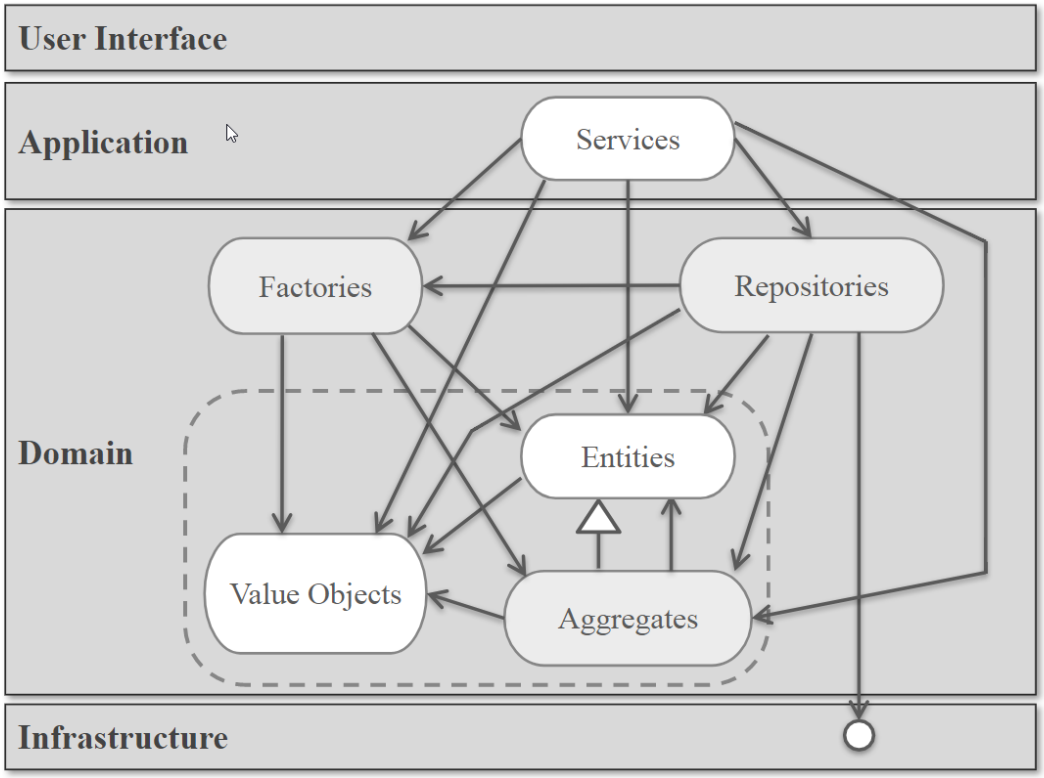


Figure 6.2: DDD General Structure

However, factories are more customisable. That is why this is not part of the DDD refactoring tool. A factory can, for example, create an aggregate root, and within the same creation process, it adds some entities or value objects and automatically creates the relations between them. Services might also interact with them as in figure 6.2.

Chapter 7

Summary

The thesis is about the modification of an existing model to DDD and its process. It began with with looking at *Domain-Driven Design*, *Refactoring* and *UML* as theoretical background to this research. After that, UML tools were described, explained and compared. *UMLet* was found to be the most useful one because it was modifiable and extendable.

The part of the project was about modifying this tool in order to refactor an existing project to DDD. The import of an existing model could be done from an Oracle database model with the JDBC driver. Nonetheless, the central part was the refactoring process within the software tool, including the designing of the new model. After the modification, the project was exportable as a new Java project and an automatically generated database script that is executable via the database connection.

The idea of DDD was also to help the developers in cases of ongoing changes in the software model. A particular part was the usage of bounded contexts because they brought clearly defined borders and changes should not affect other parts in different bounded contexts. As a result, it avoided side-effects after any modification.

Nevertheless, a software model does not usually stay the same for its entire usage. Some changes might be urgent without focusing on the code quality. These could be either unnecessary code in the model with help variables as new properties, or additional methods using properties from other classes. DDD took into account the access to the model, unlike in the anaemic model, getters and setters were more sophisticated. Help variables were modified in the model and not in the controller.

The separations in layers, as in figure 6.2, showed that services were responsible for connections between various bounded contexts. Additionally, they had an essential role because changes could impact other elements. Nevertheless, it kept the responsibility to a minimum, and a versioning system helped with compatibility to all API users.

Additionally, factories and repositories were also part of DDD. The refactoring tool added a single repository to the new Java project. JPA would have been an alternative for the repository using annotations for the model. These annotations were read by the parser and the information provided could be accessed with the meta-class object (`Class<?>`) [18]. However, value objects were not included in the model as they were part of the entity stored as a string representation. The Javascript Object Notation (JSON) already supported this. That is why it also played a role in the repository and the database. The repository was also an Object Relational Mapping (ORM) that

automatically serialised complete entities, including the value objects — likewise; it deserialised the JSON string from the database back to the value object.

DDD was a good start for proper software development because it dealt with the typical problems of growing projects. Companies tended to use *SCRUM* instead of the *Waterfall Model* because it was more flexible to changes. DDD was also more flexible to changes and should work fine with SCRUM.

In the future companies should use more strategic and conceptual models in order to keep them flexible because time and money is still an essential factor. Modifications and extensions will be quicker and more straightforward. Nonetheless, it helps to start in DDD because changing from a default anaemic model is still more challenging than already using DDD in the early stages of development. As a result, the project also supports creating models from scratch. The concept of a ubiquitous language should also be considered as necessary due to the communication with domain experts. They know the targets of the project beyond technical aspects. That is why they should be included even after the model has been created. For example, they should test the software before the release to ensure its quality.

Appendix A

CD-ROM Contents

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 Thesis File

Path: /Thesis

DDDrt-Thesis.pdf . . .	Master Thesis Document

A.2 Project File

Path: Project/

dddrt-1.0.1.zip	All the *Domain-Driven Design Refactoring Tool* project in a single zip file

A.3 Online Sources

Path: /Online-Sources

[27]SOA_Manifesto.pdf	Printed version of: Thomas Erl et. al – SOA Manifesto (`http://www.soa-manifesto.org/`)
[28]Anaemic_Domain_Model.pdf	Printed version of: Martin Fowler – Anaemic Domain Model (`https://martinfowler.com/bliki/AnemicDomainModel.html`)

References

Literature

[1] Bill Moore et al. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, Feb. 2004 (cit. on p. 17).

[2] Coral Calero et al. "Measuring Oracle Database Schemas". *The 3rd IMACS/IEEE International Multiconference on: Circuits, Systems, Communications and Computers (CSCC'99)* 99 (1999), pp. 7101–7107 (cit. on p. 36).

[3] David Steinberg et al. *Eclipse Modeling Framework: A Developer's Guide*. Addision Wesley, Aug. 2003 (cit. on p. 17).

[4] Dominic Betts et al. *Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure*. 1st ed. Microsoft patterns & practices, 2013 (cit. on p. 6).

[5] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994 (cit. on p. 8).

[6] James Rumbaugh et al. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2005 (cit. on p. 18).

[7] Marc Loy et al. *Java Swing*. O'Reilly Media, 1998 (cit. on p. 18).

[8] Martin Auer et al. "A Flyweight UML Modelling Tool for Software Development in Heterogeneous Environments". In: *Proceedings of the 29th EUROMICRO Conference*. (Washington, DC, USA). IEEE Computer Society, 2003, pp. 267–272 (cit. on pp. 18, 19).

[9] Martin Fowler et al. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999 (cit. on pp. 9–11).

[10] Martin Fowler et al. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2004 (cit. on pp. 12–14, 19).

[11] P. Leach et al. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. Network Working Group, July 2005. URL: https://tools.ietf.org/html/rfc4122 (cit. on p. 7).

[12] Philip Langer et al. "EMF Profiles: A Lightweight Extension Approach for EMF Models." *Journal of Object Technology* 11.1 (2012), pp. 1–29 (cit. on p. 17).

[13] Tim Berners-Lee et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. Network Working Group, June 1999. URL: https://tools.ietf.org/html/rfc2616 (cit. on p. 6).

[14] Kent Beck. *Extreme Programming Explained*. Addison-Wesley, Oct. 1999 (cit. on p. 10).

[15] Peter Pin-Shan Chen. "The Entity-Relationship Model – Toward a Unified View of Data". *ACM Transactions on Database Systems* 1.1 (Mar. 1976), pp. 9–36 (cit. on p. 6).

[16] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Pearson Education, 2003 (cit. on pp. 2, 4, 5, 8, 45, 47).

[17] David Flanagan. *Java in a Nutshell*. 5th ed. O'Reilly Media, 2005 (cit. on p. 7).

[18] M. Keith and M. Schincariol. *Pro EJB 3: Java Persistence API*. Expert's voice in Java. Apress, 2006 (cit. on p. 54).

[19] Sebastian Łaskawiec. "The Evolution of Java Based Software Architectures". *Journal of Cloud Computing* 2.1 (2016), pp. 1–17 (cit. on p. 5).

[20] Julia Lerman. *Programming Entity Framework*. 2nd ed. O'Reilly Media, 2010 (cit. on pp. 15, 16).

[21] Robert Cecil Martin. "The Dependency Inversion Principle". *C++ Report* 8.6 (June 1996), pp. 61–66 (cit. on p. 5).

[22] M. Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, 2011 (cit. on p. 6).

[23] David M. Piscitello and A. Lyman Chapin. *Open Systems Networking: TCP/IP and OSI*. Addison-Wesley, 1993 (cit. on p. 5).

[24] Fouad Toufik and Mohamed Bahaj. "Reverse Engineering of Object Relational Database". In: *Proceedings of the 2018 International Conference on Software Engineering and Information Management*. (New York, NY, USA). Casablanca, Morocco: ACM, Jan. 2018, pp. 73–76 (cit. on pp. 22, 37).

[25] Vaughn Vernon. *Implementing Domain-Driven Design*. Pearson Education, 2013 (cit. on pp. 2–7, 27).

[26] Eberhard Wolff. *Microservices: Flexible Software Architecture*. Addison-Wesley, 2016 (cit. on p. 8).

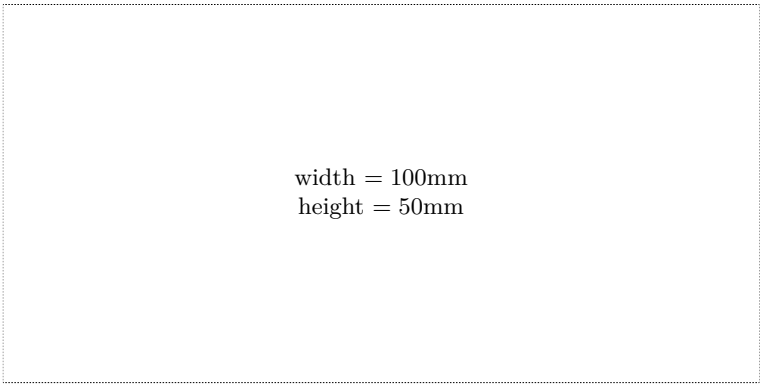
Online sources

[27] Thomas Erl et al. *SOA Manifesto*. 2009. URL: http://www.soa-manifesto.org/ (visited on 06/16/2019) (cit. on pp. 5, 6).

[28] Martin Fowler. *Anemic Domain Model*. 2003. URL: https://martinfowler.com/bliki/AnemicDomainModel.html (visited on 06/19/2019) (cit. on p. 3).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —