Empirical Software Engineering (2023) 28:94
https://doi.org/10.1007/s10664-023-10310-1

Refactoring with domain-driven design in an industrial context

An action research report

Ozan Özkan[1] · Önder Babur[1,2] · Mark van den Brand[1]

Accepted: 27 February 2023 / Published online: 15 June 2023
© The Author(s) 2023

Abstract

Context Software developers need to constantly work on evolving the structure and the stability of the code due to changing business needs of the product. There are various refactoring approaches in industry which promise improvements over source code composition and maintainability.

Objective In our research, we want to improve the maintainability of an existing system through refactoring using Domain-Driven Design (DDD) as a software design approach. We also aim for providing empirical evidence on its effect on maintainability and the challenges as perceived by developers.

Method In this study, we applied the action research methodology, which facilitates close academia-industry collaboration and regular presence in the studied product. We utilized focus groups to discover problems of the existing system with a qualitative approach. We reviewed the subject codebase to construct our own expert opinion as well and identified problems in the codebase and matched them with the ones raised by engineers in the team. We refactored the existing software system according to DDD principles. To measure the effects of our actions, we utilized Technology Acceptance Model (mTAM) questionnaire, and also semi-structured interviews with the development team for data collection, and card sorting methodology for qualitative analysis. For minimizing bias that might affect our results with the existing software engineers in the team, we extended our measurement with three new joiner software engineers in the team through the think aloud protocol.

Communicated by: Daniel Méndez

✉ Ozan Özkan
o.ozkan@tue.nl

Önder Babur
onder.babur@wur.nl

Mark van den Brand
m.g.j.v.d.brand@tue.nl

1 Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands

2 Information Technology Group, Wageningen University & Research, Wageningen, The Netherlands

Results We have identified that engineers mostly gave positive answers to our interview questions, which are mapped to software maintainability metrics defined by ISO/IEC 25010. Our DDD refactoring scored 85 in PU and 83 in PEU, leading to an overall mTAM score of 84. This means *acceptable* on the acceptability scale, *B* on the grade scale, and *good* on the adjective rating scale.
Conclusion Our research led us to conclude that a powerful design approach, like DDD, is an effective tool for restructuring and resolving software issues in this situation. It offers standardization to the software and the refactoring efforts. We realized that DDD entails a certain degree of complexity and cognitive load, which is a barrier for software engineers, but they are aware of its benefits.

Keywords Domain-driven design · Action research · Software refactoring · Software architecture · Microservices

1 Introduction

Software systems must constantly change over time to implement new functional and non-functional requirements. Thus, the size and complexity of the software system continuously increase, which eventually leads to a decrease of the quality of the software (Lehman 1980). To keep the quality level of a software system, it should be maintained and adapted rigorously. This is a continuous and vital challenge for software developers and companies, and in fact it is reported in the literature that software maintenance is as high as 70-80% of the software development costs (Schach 2011).

Refactoring is one of the major maintenance activities in the software development process to ensure quality. It is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure (Fowler 1999). As Mens and Tourwé (2004) also state, it is a tangible process which solves known issues but at the same time has significant side effects such as changing design patterns, responsibilities of software components and complexity. These side effects can create new challenges that software engineers might face in the near future while introducing new features. It might also introduce Tech Debt, such as bugs, and require a very high number of code transformations. Hence, refactoring should be well-designed and executed in a disciplined way (Abid et al. 2020).

Various approaches in the literature and industry exist to design and execute refactoring (Abid et al. 2020). They present multiple claims on the types of problems solved, suitability to various (especially industrial) contexts and effectiveness in practice. One of the examples is utilising Domain-Driven Design (DDD) as an approach for systematic refactoring. Our hypothesis that DDD has a proven record as a good tool for refactoring in software systems similar to our studied product and increase effectiveness as reported in literature (Evans 2003; Fowler 2020, 2003; Oukes et al. 2021; Wade and Salahat 2009; Braun et al. 2021; Kapferer and Zimmermann 2020; Landre et al. 2007).

In this paper, we address a refactoring case in industry to improve software maintainability using DDD. We performed our research in an industry setting with a software product that is released and used on a daily basis by customers. During this research, we were embedded in the industry directly, justifying design decisions and methodologically evaluating the effectiveness of our DDD refactoring. We adopted the Action Research methodology (Staron 2020) for this research because it investigates how an intervention affects a real-life context.

Our contribution is as follows. We (1) identified a maintainability problem at our industrial partner, (2) proposed a solution plan through refactoring using DDD, (3) implemented the solution in its real context, and (4) evaluated its perceived effectiveness and usefulness by the stakeholders. We focus on perceived effectiveness and usefulness, as we believe the problem is a socio-technical one rather than purely technical (Mens 2016).

Therefore, in this action research study, we aim to obtain empirical answers to two research questions (RQ) in an industrial context to clarify the mentioned objectives:

RQ1: How can we improve the maintainability of the current software system?
RQ2.1: How does DDD-based refactoring affect maintainability of the software system, as perceived by the developers?
RQ2.2: What are the challenges with refactoring using DDD?

The goal of RQ1 is to understand what actions we need to take to increase the maintainability of the current software system and what will be the effects of those actions, especially on a codebase that belongs to an actual product which customers actively use. Our RQ2.1 is to understand how DDD plays a role in our refactoring efforts to increase maintainability. Since DDD is an opinionated approach, we also would like to know the challenges of implementing it in an existing codebase, which is addressed in our RQ2.2. Does it work as-is, or should software developers perform specific approaches? Does it work cooperatively with the latest technologies? In this study, we aim to obtain answers to our RQs in real-life by applying Action Research Methodology.

The rest of this paper is organized as follows: Section 2 summarizes background information regarding our two main topics, DDD and Action Research. Section 3 provides details regarding our research design with industry background, product, action team and our discovery and measurement methodologies. Section 4 lays down the existing system, following by identified problems in Section 5. Section 6 describes our actions in detail and provides information about the refactored system. The results are analysed in Section 7 followed by our learnings and recommendations for other companies in Section 8. Section 9 discusses threads and validity of this study, and Section 10 summarizes related work in the literature. Finally, concluding remarks and future work can be found in Section 11.

2 Background

In this section, we are briefly sharing background information about two main pillars of this study: Domain-Driven Design in Section 2.1 and Action Research in Section 2.2.

2.1 Domain-Driven Design Essentials

DDD is a software development approach where the structure of the software source code such as class, method and variable names should match the *business domain*. It has arisen as a powerful solution to leverage complex needs of the business and effective abstraction on software design and implementation. It is introduced by Eric Evans in his book (Evans 2003) as an extension to the domain concept. Fowler defines DDD as an approach that centers the development on programming a domain model that has a rich understanding of the processes and rules of the domain (Fowler 2020).

To describe what DDD is and what does it mean in detail, first we need to establish what *domain* represents in the context of software engineering. The definition of *domain* by the dictionary is *"A specified sphere of activity or knowledge."* (Domain 2022), hence *domain* in

the context of software engineering refers to the knowledge area, where software will operate. Each software program created for solving specific problems or relates to some activity or interest. That particular problem, activity or interest, which can be named as subject area, where the created software program has been applied is the *domain* of the software. Formally, it represents the target subject of a specific programming project, whether narrowly or broadly defined (Bjørner 2017). Evans claims *domain* simply as *"it's the thing the software is about"* (Evans 2003). In software engineering, the *domain* is sometimes referred as *problem domain* as well (Evans 2003).

To make the definition more clear, we could use the following example: a particular software might have an objective to manage the supply chain of a particular car manufacturer. In this case, all the techniques, rules, specifications and concerns required for managing the supply chain of a particular car manufacturer, which we could also define them as knowledge, would be the *domain*. At this level, the *domain* is only a sphere of knowledge, a definition of the terminology, problem and complexity, not an implementation (Evans 2003).

In the phase of software design, all the knowledge and terminology of the related *domain* are applied to the software architecture and functionality. With this approach, designed software could be also expanded in scope to include supply chain operations of all car manufacturers as their *domain*.

In other words, the *domain* is all the things that need to be understood about the context in order to develop the software, but it has nothing to do with the language used to develop the software. If we consider embedded software for telecommunications equipment, the *problem domain* of this software might be Ethernet, voice and video protocols. If a firmware will be developed for military aircraft, the *problem domain* is weapons, sensors and control systems.

In consideration of given definitions and examples, we could define the *domain* as an area of expertise which is abstracted from the implementation details of the software.

DDD is an expansion of the *domain* concept (Evans 2003). Its main objective is creating and maintaining complex applications more easily by an evolving model, which helps software developers and teams better produce high-quality software modelling. With the DDD approach, software design becomes a more appropriate description of how the software works at a high level (Fowler 2003).

DDD is based on three core principles:

- Placing the application's primary focus on the core *domain* and *domain logic*.
- Basing complex designs on a model of the *domain*.
- In order to improve the application model and resolve any *domain* related issues, continually collaborating with *domain experts*.

The above-mentioned principles potentially alter the common practices of conventional software design processes, where the primary focus is on software architecture and implementation details such as classes and their dependencies and also software functionalities and features, to an understanding where the sphere of knowledge and activity about the subject area are more important and play a significant role in software design and development.

With the DDD approach, software is being developed closer to the mental model of the business. DDD brings domain experts, business experts and software developers together which facilitates a joint software development in order to deliver a piece of software that is the most useful to the business. This effort unifies the domain knowledge, strengthens the understanding of the problem and the solution among the entire team, even improves inter-team relationships (Fowler 2003).

DDD defines four concepts that consolidate the backbone of the terminology:

- *Context*: The setting for an event, statement, or idea, and in terms of which it can be fully understood. Statements regarding a *model* can be only understood in a context.
- *Model*: A system of abstractions that describes aspects and processes of a *domain*, designed and used to solve problems related to the same *domain*.
- *Ubiquitous Language*: A structured language specially crafted for the *domain*, contains terms and actions of the *domain*, and it is used by all team members to connect activities of the team with the software. It is basically a shared team language, every team member should use, no matter what role they have in the team.
- *Bounded Context:* A conceptual boundary of a system, subsystem or a work of a specific team within a particular *model* is defined and applicable. When a large project which has multiple *models* are in play, the project becomes difficult to understand and unreliable. Explicitly setting boundaries in terms of usage of specific models, parts of the application or a team organization will help in keeping the *model* strictly consistent and prevent it from distractions and confusions. Every *bounded context* contains one *ubiquitous language* which defines the meaning of *domain* terms, phrases, sentences and actions in the *bounded context*. Usage of those terms outside the particular *bounded context* has probably different meaning.

Since its introduction by Evans in 2003, DDD is widely used to tackle the complexity of the business domain and also a popular subject for publications in software engineering and design (Fowler 2003).

2.2 Action Research in Software Engineering

Action Research is one of the research methodologies that has been applied by many disciplines and gained popularity in the second part of the twentieth century (Brydon-Miller et al. 2003). The main reason for its popularity is that it is focused on organizational learning with on-hands experience as part of the research process. Reason and Bradbury define action research as *"a participatory, democratic process concerned with developing practical knowing in the pursuit of worthwhile human purposes, grounded in a participatory world-view which we believe is emerging at this historical moment. It seeks to bring together action and reflection, theory and practice, in participation with others, in the pursuit of practical solutions to issues of pressing concern to people, and more generally the flourishing of individual persons and their communities"* (Reason and Bradbury 2001). In this definition, Reason and Bradbury emphasize that the idea of action research is seeking innovations and improvements in a practical environment, bringing theory and practice together in collaboration with peers in the environment.

In software engineering, action research is a methodology that inherited from the field of information systems. According to the studies by Santos and Travassos (2009), (2011), the number of action research studies in software engineering field is rapidly increasing. The nature of software engineering is very suitable to execute action research, since software engineers construct their theories based on empirical observations and also apply new methodologies and improve them during the execution. Action research combines researching, learning and executing actions, applying action, thus, research in software engineering context is very straightforward.

Action research process is based on iterative research cycles in company's context (Staron 2020). A cycle starts with *problem diagnosis*, which is either done by exploring the problem or outcome of the previous research cycle. It continues with *action planning*, where we define specific actions and their expected outcomes. The *action taking* phase is the execution phase

of the planned actions. The action research cycle ends with documenting results and learning. This cycle approach is familiar for software engineers from contexts such as Requirements Engineering or widely-used iterative software development methodologies e.g. Agile or V-model. These characteristics make action research very suitable for research projects in software engineering.

Staron (2020) defines the team who conducts the action research as *action team*. The action team is responsible for planning, executing and evaluating the research. He describes the topology of an action team as a team consists of practitioners and researchers. Practitioners are members such as software engineers, architects, testers, designers are involved in planning and executing actions in the research project. On the other hand, researchers are responsible for providing external perspective with asking critical questions, bringing experience from other research projects and their expertise on state-of-the-art research results. In addition to the action team, there is also a *reference group* who is responsible for giving feedback and advice to the action team and a *management team* who is responsible for managing the project and providing support with applying the results of the study in their organization.

The primary motivation and the goal of the action research is improving practices in the industry and contribute to the knowledge by reporting the experience and results. The first part of the goal is achieved by executing the study in an industrial context and working together with industry professionals, which is relatively easy because of previously mentioned characteristics of action research and its similarities with software engineering processes. However, the challenging phase is the reporting phase, since the industrial context usually has sensitive information, and it’s not always possible to publish. While Staron defining basics of reporting action research studies in his book, he emphasizes that reporting should respect to sensitive information, so companies can maintain their businesses.

3 Research Design

In this study, we applied the *action research* methodology, which is one of the main research methodologies used for academia-industry research collaborations which facilitates close collaboration and regular presence in the studied product. We follow the action research design methodology specified by Staron (2020) and we follow their guidelines for reporting our study. We explain the industrial context of the study where we share information about the company, we also give information about the studied product with brief information about the product’s context and technical details. We further explain the characteristics of the team developing the studied product and our action research team. Steps we have followed during our action research can be seen in Fig. 1.

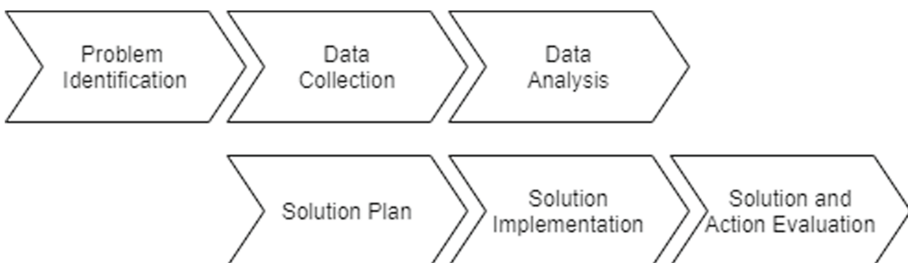

Fig. 1 Diagram showing the steps we follow during our action research

3.1 Industrial Context

The company that creates the product focused on this study is a worldwide leading independent location technology specialist based in The Netherlands. It employs around 5,000 diverse individuals in 30 offices around the world. It develops location technologies such as highly accurate maps, navigation software, real-time traffic information, APIs that enable smart mobility on a global scale and services for autonomous driving over 30 years. These technologies are being relied upon by hundreds of millions of drivers, businesses and governments. The products run in dedicated consumer devices, mobile devices, in-car head units and also in automated driving systems and the cloud infrastructure.

Since the company is developing multiple products, it is divided amongst product units and each unit usually has multiple sub-products and employs several hundred developers. Product development conducted according to the principles of Marty Cagan's Empowered Product Teams (EPT) (Cagan 2017) and Agile software development procedures. Since their products are mainly for the automotive industry, next to EPT and Agile procedures they are employing processes such as DFMA and Requirements Management to ensure safety and security standards.

Team formation in the company based on product and/or product components. Each product or product component has a development team mainly consisting Product Manager, Architect, Tech Lead and Software Engineers. In some cases, there are extra roles such as UX Designer, technical and business-oriented product managers and safety engineers.

The product unit within the company, which includes the product team that we worked together for this study, has over 350 developers divided into multiple countries. They work in a combination of EPT and Agile principles. They are developing systems for autonomous driving vehicles. And products they are developing are used by well-known car manufacturers with millions of cars on the road.

3.2 Studied Product

The studied product is an industry-first product that manages the Operational Design Domain (ODD) of autonomous vehicles, which allows carmakers to decide where it is safe for drivers to activate their vehicles' automated driving functions. The product has an intuitive web editor where carmakers create autonomous driving rules and restrictions. It also has a cloud-based service to process all updates and send them directly to vehicles. Updates are delivered to vehicles over the air through a safety-critical delivery product, which is being developed by another team in the same organization. The product is already released to car manufacturers, and it is continuously being improved with new features.

The product has multiple components such as frontend, backend, databases, tests and infrastructure as code. All the components are implemented by the same product team. The frontend is implemented in Typescript with AngularJS framework (Moiseev and Fain 2018) and communicates with the backend through REST API endpoints exposed by the backend. The backend is implemented in Java with Spring Framework and implements the entire business and domain logic and also communicates with 3rd party services and databases such as PostgreSQL. Communication with 3rd party services done by either client libraries or REST APIs. The frontend, backend and databases are deployed to Azure and working in the cloud.

3.3 Product Team

The team that develops the studied product has 13 diverse full-time developers including Product Manager, System Architect, UX Designer, Test Lead and Software Engineers. Like their organization, the team also works in a combination of Agile with two weeks sprints and EPT principles. They employ continuous integration pipelines, processes for managing requirements and safety compliance. The software engineers in the team possesses varying levels of experience both in the software industry and DDD.

3.4 Action Team

Our action research team consists of three researchers, including two faculty members at the university as advisors and a PhD candidate to execute the research in the subject product, who is also a senior software engineer in the subject company and working full-time in the subject product.

3.5 Data Collection and Analysis Methods

This industrial study utilizes several data collection and analysis methodologies in problem discovery and action measuring phases. Results of our analysis are reported and discussed in the *Results and Interpretation* section. However, to protect the intellectual property of the company, in this study we did not share materials we have used during the data collection such as software architecture diagrams and transcripts of the interviews. Any information linked to the company intellectual property, such as the real names of the components, is also redacted.

3.5.1 Problem Discovery

We discovered the problems of the subject software product with a qualitative approach following focus group guidelines (Kontio et al. 2004). Our motivation for using a focus group method was the following: (1) the product team already had more than 10 group meetings to identify and discuss technical problems, (2) we wanted to create a discussion environment with subject-matter experts to get their expert judgement, (3) it was fast and cost-effective due to the small number of subject-matter experts. To implement the focus group method, we chose to use a *mini focus group* setup because of the number of developers in the product development team. We held two mini focus group meetings with software engineers of the product team who work in the subject codebase on a daily basis. Meetings lasted for 60 minutes with 7 software engineers. All interviews were recorded by notes with the permission of the participants. After each focus group session, the recordings were semantically analysed following the *attribution analysis* methodology (Janis 1965), which examines the frequency with which certain characterizations or descriptors are used.

3.5.2 Action Taking

After problem discovery, we have decided that the subject software product requires refactoring to address identified problems. To make informed decisions, as researchers in the action team, we reviewed the literature to explore the suitable approaches.

We concluded that there are several software architecture approaches, including *Object-Oriented Design (OOD)*, *Service-Oriented Architecture (SOA)*, *Event-Driven Architecture* and *Domain-Driven Design (DDD)*. Each of these approaches has its own strengths and weaknesses, and the best approach for a particular project will depend on the specific requirements and constraints of the project. We took into account that the subject software system is a *product* (Section 3.2), which is actively developed by a *product team* (Section 3.3) in a company which has a *product* (Section 3.1) mindset. According to our findings that we discussed in DDD section (Section 2.1), we concluded the following: DDD is a way of thinking and designing software systems that aligns the code with the business domain, making the code easier to understand and maintain. By using DDD, companies can create software that is better aligned with the business domain, making it more effective at solving the problem it was designed to address. Additionally, using DDD can make it easier for developers to understand the business domain, which can help them create more maintainable code. Therefore refactoring with DDD would put the software close to the product requirements so that the architecture and internal organisation of the software would be more logical, understandable and maintainable.

After we decided to move forward with DDD for our refactoring, we started to design our action. After careful consideration in the action team, we decided to perform a refactoring which has a small footprint to (1) focus on the problems better, (2) have an end-to-end refactoring in a short period of time, (3) provide a self-contained example so that it will be possible to measure with less noise as possible, and (4) ensure that the rest of the system won't be affected by our changes so that we won't affect the business continuity.

To decide which part of the software we will introduce our refactoring, we used the business cases of the software product as a reference. We identified a business case which has an end-to-end implementation from the entry point of the software to the persistent storage; it consists of *2 controllers, 4 services, a repository implementation* and *9 data objects* of the existing software system that we discussed extensively in Section 4.

Considering the information at hand, we decided to exercise *one cycle of action-taking* where we introduced our refactoring changes and everything around it. Once our cycle is done, we measure the cycle with the stakeholders outside of our action team, which are software engineers with relevant information and experience with the system before our action. We discussed this in more detail in Section 3.5.3.

3.5.3 Measuring and Evaluating Actions

In order to answer our RQ2.1 and RQ.2.2, we utilised semi-structured interviews with the product team's subject-matter experts to qualitatively analyse our actions' effects. The group of interviewees is the same engineers that we conducted the focus group meetings in the problem discovery phase. We conducted a 90-minute written session where engineers answered 9 questions about software maintainability, challenges and wrap-up questions. We used ISO/IEC 25010 Software, and Data Quality standard (ISO/IEC 25010 2011) as a reference for software maintainability metrics, defined as Modularity, Reusability, Analysability *(Understandability)*, Modifiability *(Changeability)* and Testability. The mapping between questions and software maintainability metrics can be seen in Table 1. We applied *Card Sorting Methodology* (Zimmerman and Akerelrea 2002) to analyse the interview results to understand engineers' expert judgement about changes we introduced as part of this study. We created cards based on interview transcripts and grouped the cards regarding their category and merged if they were redundant. These categories served to identify the mental model of the participants regarding our refactoring actions.

Table 1 Our semi-structured interview questions

Maintainability
1. How easier or harder is it to understand the backend implementation after DDD refactoring? *[Understandability]*
2. When you compare DDD refactored backend with the previous implementation, how easy or hard is it to introduce a change to a business logic without affecting the rest of the implementation? *[Modularity]*
3. How complex is it to introduce new features to the current implementation after DDD refactoring? *[Changeability]*
4. When you compare DDD refactored backend with the previous implementation, how straightforward is it to create tests to a new feature after DDD refactoring? *[Testability]*
5. When you compare DDD refactored backend with the previous implementation, how possible is it to reuse the existing implementation while implementing a new feature? *[Reusability]*
Challenges
6. What are the most challenging aspects of using DDD?
7. How do you cope with these challenges?
8. How would you describe DDD affecting your development experience?
Ending
9. Having discussed some topics about DDD refactoring, would you like to add some thoughts?

We also utilized the Technology Acceptance Model (mTAM) questionnaire (Lah et al. 2020) to quantitatively measure effects of the actions we took during this study. The mTAM questionnaire consists of six perceived usefulness (PU) and six perceived ease-of-use (PEU) questions. mTAM questionnaire uses PU to measure "The degree to which a person believes that using a particular system would enhance their job performance." and PEU to measure "The degree to which a person believes that using a particular system would be free of effort." (Davis 1989). We employed mTAM results to support our interview results, which directly answers to RQ2.1 and RQ2.2.

In pursuance of minimizing bias that might affect our interview results with the existing software engineers in the team, we extended our measurement with three new joiner software engineers in the team. We have conducted a *Think Aloud Protocol* (Tirkkonen-Condit 1990) to analyse the effects of actions we took in the perspective of engineers who have no prior experience nor knowledge of the studied backend code and the changes introduced by this study. We have conducted our *Think Aloud Protocol* with providing software architecture diagrams of the studied backend code before and after DDD refactoring to the interviewees and asked them to vocalize their thoughts and actions they could take while implementing a real business case we have provided. We have recorded their statements by creating cards and used *Card Sorting Methodology* to analyse their judgements. Thereafter, we compare our results with the interview results that we have conducted with existing software engineers in the team to ensure an unbiased result.

3.6 Ethics

Since this study involves human participation, we went through the Ethics Board of the university to be reviewed and approved. We have reported which steps we are going to take and what kind of methodologies we are going to apply in order to collect the data from our participants. We have also reported what kind of data we are going to collect, how long we

are going to store them and where. Even though this study does not collect and store any personal data which can be mapped to the participants, we ensured our compliance with GDPR and data security. Before interviews, all the participants are informed about the study and how we will use and store the collected data. They participated in our sessions with their consent.

3.7 Data Availability

The datasets generated during and/or analysed during the current study are not publicly available due to company intellectual property restrictions but are partially available from the corresponding author on reasonable request.

4 Existing System

The studied component of the product is called `backend` by the product team, which is an online service deployed to a cloud infrastructure. The main responsibility of the backend is to implement all the business logic required for the product. It also has the responsibility for communicating with 3rd party services and repositories such as SQL databases. It is implemented in Java 11, and it is using the power of Spring Framework such as dependency injection and powerful REST API capabilities.

At the moment that we conducted our study, the backend consisted of around 10,000 lines of code (LOC), implementing 23 REST API endpoints with 11 controllers and services underneath. There were also several wrappers around 3rd party dependencies and self-contained, packaged implementations being used by several services.

4.1 Architecture

In order to understand how the existing system is designed, we should go beyond the high-level information mentioned above and elaborate more on the software architecture and components. The implemented software architecture is Controller-Service-Repository (CSR) architecture, which is one of the commonly used software architectures for online services which are exposing REST API endpoints. It is a layered architecture where responsibilities such as defining REST endpoints, implementation of the business related logic and database communications are positioned in different layers as it can be seen in Fig. 2. In the following subsections, we explain the components of the subject software system in more detail.

4.1.1 Controller

The controller is the top layer of the software, where REST API endpoints are defined and mapped to the method to be invoked when there is a call to the defined endpoint by the consumer. The URI of the endpoint, path parameters, request body, response type and the code is also defined in the controller implementation. The controller also implements the input validation of endpoints and, when the specified return data in the API specification is different from the original resource, it converts Value Objects provided by the service to the special Data Transfer Objects (DTO) before handing over the requested data to the consumer.

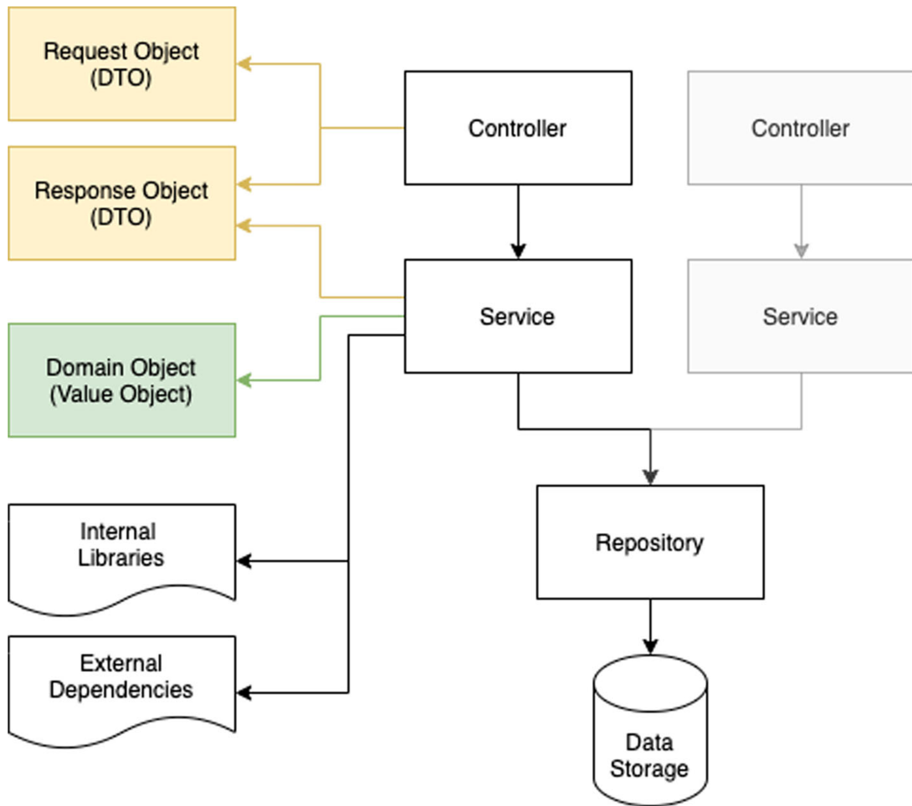

Fig. 2 The Software Architecture of the existing software system (backend), showing the dependency direction between classes and internal/external dependencies such as libraries and 3rd party dependencies

4.1.2 Service

The service is the middle layer of the software, and it implements the business logic of the controller. As shown in Fig. 2 the service also communicates with the repository or 3rd party dependencies if it is necessary for the implemented business logic. One of the design decisions of the engineers in the product team regarding the dependencies of the software components is, a controller can consume only one service, thus, each controller in the backend communicates with one service which implements the business logic for the controller.

4.1.3 Repository

The repository is the lowest layer of the backend, where communication with persistent data sources, such as databases is implemented. The implementation in the repository layer collects, saves or updates the data in the persistent data source and also converts the collected data into value objects. The service consumes the repository when the business logic in the service requires a read/write operation on the persistent data source. Once there is a read operation, conversion to value objects for raw records in the persistent data source is also executed in the repository.

4.1.4 Domain Object

A domain object is a small value object that represents the information defined by the business requirements of the product. They hold the information, and they can be transferred to any layer. Value objects in the backend only store the information and do not implement any logic.

4.1.5 Data Transfer Object

A data transfer object is a value object that represents the resource of the REST API endpoint. It does not have any behaviour except representing and temporarily storing the data. They are used if the domain object in the backend is not matching with the resource defined in the REST API specification.

4.1.6 Internal Libraries

Some implementation required by the core business functionalities of the product is encapsulated in libraries. Those libraries are implemented and maintained by the product team. Libraries have independent and standalone logic required to fulfil certain core business logic of the product.

4.1.7 External Dependencies

The studied product has dependencies to external services, which are services developed by the same company, to acquire and share certain information required by core product functionalities. Those dependencies are usually integrated by wrapping client libraries provided and maintained by the external services, or, consuming their REST API from the backend.

4.1.8 Data Storage

The backend uses a SQL compliant relational database management system (RDBMS) as a persistent data storage. This storage runs in the cloud infrastructure and is always available to the backend. It contains one database instance with several tables, which is designed to store the data that corresponds to the business and functional requirements of the product. Currently, the storage has more than 50 GB of data, which is produced in a year, and it grows rapidly in line with the product's usage trends by the customers.

5 Identified Problems

We have identified the following problems based on focus group interviews we have conducted with 7 software engineers who are working on the software system on a daily basis. We asked our focus group to state their ideas about issues they have encountered during the development process, the reasons for those problems, and how they can be resolved. Overview of the attribute analysis can be seen in Table 2.

We also correlated the results of the focus group interviews with our expert judgement based on our experience and software architecture and code reviews that we have conducted. We report our conclusion in the following subsections.

Table 2 Overview of the attribute analysis results of the focus group interviews

Attribute	Occurrences
Issue: Code duplication (mostly in services)	6
Issue: Domain objects with no logic	4
Issue: Code maintainability	5
Issue: Test maintainability	4
Solution: Refactor code	4
Solution: Improve Code Quality	3
Solution: Use better domain object design patterns	3
Solution: Use better tools and frameworks	4

5.1 Architectural Problems

According to the results of our evaluation, the current software system contains several architectural problems. It has *anaemic domain model* anti-pattern, which is considered contrary to the OOD principles (Fowler 2003) and needs to be addressed to improve responsibilities at the object level. We also identified that the current implementation *duplicates the business logic in different services* and *business logic is not encapsulated*, which is also an anti-pattern of code re-usability and needs to be addressed to improve the overall code quality and tests. The current software system *implements a centralised repository* in the data access layer, resulting in a god class for the entire database operation. It needs to be improved to address responsibility and scalability concerns. We also noticed that the *DO and DTO dependencies are not following a standard* which needs to be addressed to reduce the footprint of future changes and the risk of software failure.

5.1.1 Anaemic Domain Model

The current software design and the implementation of the backend can be described as an Anaemic Domain Model, where the domain objects contain little or no business logic. Martin Fowler, who for the first time described this pattern, considers the practice as an anti-pattern. He states that “The fundamental horror of this anti-pattern is that it’s so contrary to the basic idea of object-oriented designing; which is to combine data and process them together” (Fowler 2003). According to Fowler, this anti-pattern is a common approach in three layered Java and .NET service applications especially the ones using technologies like Entity Beans, in our case, it can be considered as Spring Framework. When we evaluate the current backend implementation, we can examine strong clues of the Anaemic Domain Model. Validations, calculations, business rules and the core logic for the business are always implemented in layers when they are needed for a particular business use case. Objects representing business objects, which can be also considered as domain objects, are value objects without implementing any validation, calculation or business rules, and they are only representing data. This situation potentially leads to the creation of unchecked domain objects and core business logic. The validity of these domain objects and the core business logic is also potentially compromised. Even though there are unit, integration and system integration tests checking the validity of the generated information and the business logic, since the studied product is safety-critical, there is still a substantial risk to have invalid objects and logic, which will produce wrong information. The maintenance of the code is

also quite difficult and making changes is potentially risky. Since the core business logic and its constraints are not encapsulated and occasionally implemented in multiple places, a small change on product requirements leads to complex and repetitive refactoring work in multiple places, where the business logic needs to be changed was previously implemented. Since multiple units are affected by this change, tests checking the integrity of those units should be also adapted to the recent changes. Most of the time, considering the logical complexity of the product requirement change, the magnitude of the modifications has to be done and potential failures those changes may cause are considerably extensive.

5.1.2 Duplicated Business Logic in Services

Code duplication is an issue in most software systems. According to Roy and Cordy, around 5-50% of the code in software systems can be contained in clones (Roy and Cordy 2007). In our case, the service implementation in the backend is highly endpoint oriented. Thus, they are implementing business use cases. Parts of some business use cases might share the same core business logic. This is a perfect place to take advantage of code re-usability and apply the same core business logic. But, since business use cases are encapsulated in services in the same layer, code re-usability in the same layer is almost impossible.

Considering our backend is a part of a safety-critical product, its business logic implementation must be well tested, and its integrity must be guaranteed. Given the situation where we have multiple places where the same or similar business logic has been implemented, maintaining and testing the business logic is very difficult. Also the risk of introducing a breaking change is very high since all occurrences of the same, or similar business logic should be changed.

When a new requirement is introduced to certain business logic or one of the requirements modified, the business logic which is affected should be modified according to the requirements. Since multiple places are implementing the same business logic for different business use cases, the team should refactor multiple services to apply the change. This is a highly expensive operation in terms of effort and the risk of creating inaccurate results.

5.1.3 Repository Implementation is Centralized

The repository implementation contains the entire functionality required to perform operations on persistent storage. As it can be seen in Fig. 2 it is shared amongst services, which makes it a large class with more than 3000 LOC. The primary issue of repository implementation is, it implements repository operations for every service in one single class. It is not only a perfect example for god class antipattern, this approach can be seen as an antipattern to the Single Responsibility Principle of SOLID, where every module, class or function in a computer program should have responsibility for a single part of that program's functionality, and it should encapsulate that part (Martin 2000). Having a repository implementation required for services in one repository class not only violates the single responsibility principle but also makes the repository implementation hard to maintain and hard to test. The repository also implements bits of business logic next to simply create, read, update and deactivate (CRUD) operations as well. We have identified a great number of methods performing queries to the persistent storage based on business logic, or, processing the obtained data according to certain business logic before handing it over to the consuming service.

5.1.4 Domain Object and DTO Dependencies are not Standardized

Fowler states that improper dependency management is one of the symptoms of rotting software design (Fowler 2003). When a software system has non-standardized dependency management, every change causes a cascade of subsequent changes in dependent modules. The software can break in many places every time it is changed.

We identified that in our studied software system, some internal dependencies are used in multiple layers and dependency strategy is not standardized. As we explained in previous sections and can be seen in Fig. 2, the backend defines and utilizes Data Transfer Objects (DTO) when the return information of the REST endpoint does not match with the Domain Object used by the rest of the implementation. DTOs are created from Domain Objects after the execution of the business logic and returned to the user as a response body of the particular REST endpoint. In this case, the problem starts with the place which implements the conversion from Domain Object to DTO, because the conversion is sometimes implemented in the controller, and sometimes in the service.

The same issue also applies to DTOs, which receive the information from the REST endpoint as a request parameter or body. We have identified that some methods implemented in services are receiving those DTOs as method parameters and carrying out the business use case logic for that specific controller based on given DTOs.

This non-standardization creates confusion on the responsibilities of controllers and services. We have identified that there are usually discussions in the development team regarding where Domain Object to DTO conversion should be implemented and why, which indicates that confusion is not only an issue in the software architecture, it is a time-consuming issue in the development team as well.

Since DTOs are very endpoint specific, employing DTOs in services is crippling the reusability of the service. DTOs make services endpoint oriented, thus, it makes certain services only usable by only one controller. Due to this approach, we identified that some services implement different methods but the same business logic and they are returning DTOs or domain objects. Those occurrences are only contributing to creating more confusion, hard to maintain and poorly testable code.

5.1.5 Business Logic is not Encapsulated

Fowler thinks that “repetition is the root of all software evil” (Fowler 2003) and, as we discussed previously, we have identified many repetition and responsibility issues in the entire backend codebase. Situations such as the Anaemic Domain Model, violation of SOLID principles (Fowler 2020, 2003) and inconsistent value object dependencies creates difficulties to have a common understanding of components’ responsibilities in the development team, and it hinders applying consistent practices and implementation. It is not only affecting the performance of the development team, it also affects tests and the overall consistency of the codebase and produced information, which is essential for such a safety-critical product.

As a result, the business logic is distributed among multiple places and multiple layers in the codebase and even on some occasions the implementation is duplicated, as we partially discussed in the statements of previous problems. In addition to current responsibility problems, domain objects representing the entities in the business logic are not implementing business constraints, thus, the integrity of their instances are not strongly checked, and they are not acting as enforcers during the processing of the business logic.

Certainly, the product is heavily tested by unit, integration, system qualification and system integration tests and surely the information produced by the business use case logic

implemented in several components of the backend is also heavily checked, and compliant to software standards of the company and the product. But, these tests are always checking the logic and results of the business use case, thus, we cannot clearly state that the core business logic is individually well tested and consistency of its logic and results are assured in the entire codebase.

5.2 Operational Problems

We have identified that architectural problems in the software system discussed in Section 5.1 raise operational problems where *tests and code are hard to maintain*. We are discussing these problems in more depth in the following subsections.

5.2.1 Tests are Hard to Maintain

We reviewed the test implementation of the studied software product against the concept of test smells by Van Deursen et al. (2001). Effects of issues we have discussed previously, such as the Anaemic Domain Model and distributed business logic, can also be seen in test implementation. We have identified that unit and system integration tests of the studied product contain 7 of 11 test smells defined by Deursen et al. (*Mystery Guest, Resource Optimism, General Fixture, Eager Test, Lazy Test, Indirect Testing, Test Code Duplication*).

We also identified that test codebase is hard to maintain due to long and complex mocking and assertion of the expected states, objects and methods. The integrity of domain objects used for the business case under the test is also tested and checked in each test, since domain objects are not implementing any tested business logic and business constraints. In the case of a new addition or a modification to a certain business use case, all the tests related to the certain business case logic needs to be modified and adapted to the new implementation. Software engineers in the development team realize this rework effort before or after the business case implementation, depending on their development strategy for the particular business case. They sometimes use Test Driven Development (TDD) methodologies, but most of the time tests are adapted after the implementation. Due to long and complex mocking and assertions, adaptation efforts are usually expensive and compromising the validity of tests.

The effect of having the business logic not encapsulated but distributed to multiple places can be also seen with the test scenarios. The backend currently implements around 1,200 unit test scenarios where most of the business logic is unnecessarily tested over and over again, but the essence of the particular unit test is testing the validity of a certain business use case, which also had to implement some business logic because the core business logic is not encapsulated and tested once.

5.2.2 Code is Hard to Maintain

One of the issues the development team suffers and also recognizes is the maintainability of the codebase. This is not an unexpected situation in consideration of the previous issues we have discussed. Each of them is contributing to hard to maintain codebase, which is also recognized by the development team.

The development team stated that most of the time they are having difficulties coming up with an implementation compliant with the current code base, and they need to rework or refactor what they implemented before implementing new requirements. They also stated that implementing new features to previously implemented business use cases or modifying

them according to the changed product requirements is most of the time difficult because of barriers the previous implementation and the software design creates, thus, considering it is a growing product, adding new features or modifying business use cases are getting more expensive each time.

We have identified that with the absence of SOLID principles (Martin 2000), centralized core business logic implementation and domain-oriented services (Fowler 2003), a small change on a business logic constraint or an addition to a business use case requires a rework on components that are completely irrelevant for the change we want to have, but it is a must-do for the consistency of applied same business logic in different components, thus, consistency of the produced information. Those must-do changes also requires modifications on their tests, and a small change becomes more expensive.

6 Refactored System

During our literature review, we noticed that DDD is used for converting monolithic software applications to MSA applications (Mazlami et al. 2017), identifying micro-services (Rademacher et al. 2018), enhancing separation of concerns and responsibilities by introducing domain objects and domain services (Kapferer and Zimmermann 2020). Since its introduction by Evans (2003) in 2003, DDD is widely used to tackle the complexity of the business domain and also a popular subject for publications in software engineering and design (Fowler 2003). Thus, we have decided that the DDD is the most practical approach to refactoring efforts.

The backend's new architecture is based on the DDD and layered architecture style. The new architecture is intended to completely replace the old architecture and deployed to the cloud with standalone and scalable instances. For now, in order to perform our study in a controlled environment, we identified a particular component in the backend to perform our refactoring. This section will cover how we refactored the backend by applying DDD practices in combination with layered architecture style, thus giving an idea of how DDD can be implemented in a setting where the project contains heavily opinionated frameworks such as Spring Framework (Dinh-Tuan et al. 2020) and industrial constraints play a crucial role.

To cover mentioned information, this section includes a description of the refactored architecture, a brief description of decisions and implemented techniques to solve identified problems and undertake challenges. We discuss how we improved the software architecture of the software system by introducing Domain-Driven Layered Architecture in Section 6.1. Following that, we describe how did we improved controllers in Section 6.2, services in Section 6.3 and repositories in Section 6.4. Later, we are briefly describe our changes in Domain Objects in Section 6.5 and Data Transfer Objects in Section 6.6. Afterwards, in Section 6.8, we briefly discuss the introduction of Intention-Revealing Interfaces to improve code understandability. We finalize with Section 6.9 where we discuss improvements in tests. An overview of the identified problems and their solutions can be seen briefly in Table 3.

6.1 Domain-Driven Layered Architecture

To implement software in a Domain-Driven approach, in his book Evans suggests utilizing an industry converged architecture, *Layered Architecture*, where the software implementation physically separated into multiple layers. Our backend already implemented in layered

Table 3 Overview of the problems and their corresponding solutions

Problem	Solution
Anaemic Domain Model	Implement Domain-Driven Layered Architecture
Duplicated Business Logic in Services	Introduce Application Service and Domain Service concept
Repository Implementation is Centralized	Introduce resource-specific repository implementation
Domain Object and DTO Dependencies are not Standardized	Refactor DTOs based on SOLID principles, use DTOs only in controller layer, extensively test DTOs
Business Logic is not Encapsulated	Implement Domain-Driven Layered Architecture, introduce Application Service and Domain Service concept
Tests are Hard to Maintain	Improve Unit Tests by testing DOs, DTOs, Services, Controllers and Repositories independently, introduce Blackbox Tests around unit of services and components
Code is Hard to Maintain	All of above

architecture and has Controller, Service and Repository layers, however, domain-related implementation diffused through layers. Thus, we focused on improving the current Layered Architecture of the backend where the domain-related implementation can be encapsulated in one place.

Our Domain-Driven changes in the Layered Architecture now naturally divide domain-related logic and business use case related logic, encapsulates and implements them in their own space and also loosely couples the domain logic from the higher-level layers of the backend. Changes in layers we have made to accommodate our Domain-Driven refactoring can be seen in Fig. 3 and also explained in details in their dedicated subsections.

6.1.1 Controller Layer

Our *Controller Layer* in the refactored architecture is almost similar to the pre-refactored controller layer. It stays as the top-level layer of the backend, and we strengthen its responsibility with defining and clarifying which objects and implementation will be part of the refactored Controller Layer. The Controller Layer responsible for exposing REST endpoints, accepting and validating requests and handing requests over to lower layers for further operation. This layer is thin and does not contain business logic or knowledge. According to the requirements of the exposed REST endpoint, this layer either communicates with the Application Services Layer or Domain Services Layer directly (Fig. 4).

6.1.2 Application Services Layer

We have introduced an *Application Services Layer* as a layer consists of our Application Services. As it can be seen in Fig. 3, the Application Services Layer is positioned between Controller Layer and Domain Services Layer and it is consumed by Controller Layer when it is necessary. This layer does not contain domain rules or knowledge, but it does contain

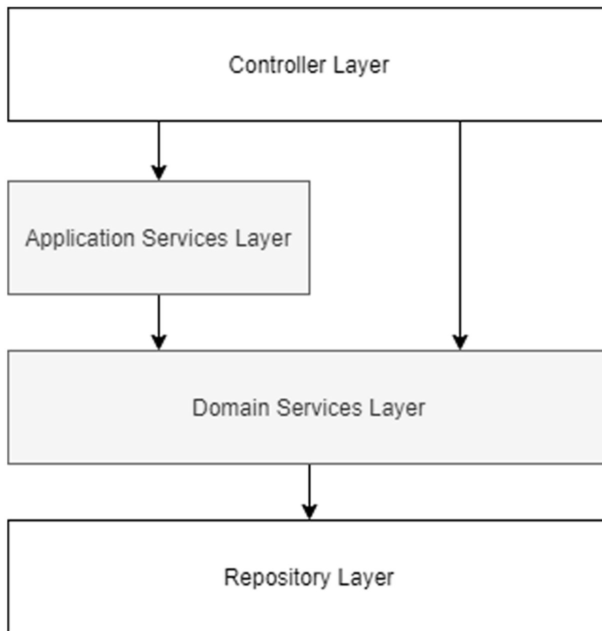

Fig. 3 Diagram showing layers of the new architecture. Arrows are showing the dependency direction. Application Services Layer implements the business use case when required, thus, Controller Layer can communicate with Domain Services Layer directly as well

the business use case knowledge and implements the coordination required by the business use case logic with aggregating and consuming Domain Services.

6.1.3 Domain Services Layer

The introduced *Domain Services Layer* is the layer of all the Domain-Driven core business logic encapsulated and implemented. This layer implements processes in the domain that are not the responsibility of domain objects and is responsible for solving domain-related problems for the user and completely agnostic to requirements such as API specifications and business use cases and operates with concrete domain objects. The domain services layer the layer also communicates with the repository, internal libraries and external dependencies when necessary.

6.1.4 Repository Layer

The *Repository Layer* is carried over to the refactored architecture as is. This layer will continue to have the implementation for the communication with persistent data sources such as relational databases.

6.2 Controllers

We redefined the scope of our controllers in order to have thin and action focused controllers. We followed norms of Single Responsibility Principle and removed all business use case and

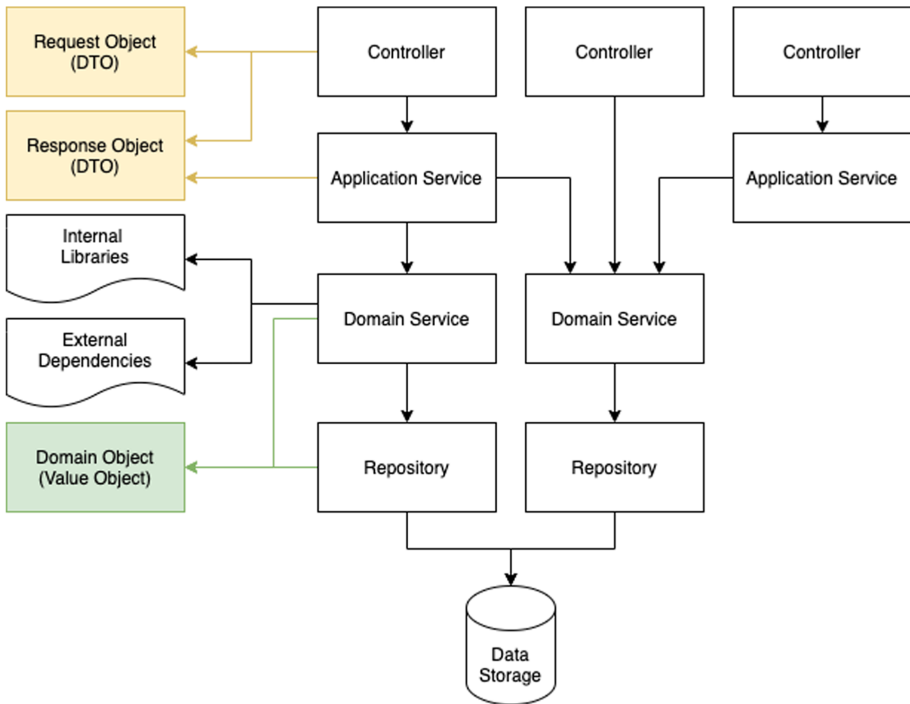

Fig. 4 The Software Architecture of the backend, showing the dependency direction between classes and internal/external dependencies such as libraries and 3rd party dependencies after DDD refactoring

core domain related logic from our controllers. With this approach, their responsibility is now exposing REST endpoints and acting as an interface to the application, perform validations in line with API specifications and dispatch requests to related Application Service or Domain Service. As a result of this approach, we shrink the controller's change surface and make them more adaptive, easy to test and maintain.

6.3 Services

Services are core of the backend, where we have most of the coordination and knowledge which produces the operational information. To ensure the separation and encapsulation of endpoint specific business use case knowledge and business domain knowledge and constraints, we have separated business use case knowledge and business domain knowledge into separate services.

6.3.1 Application Service

We have introduced an *Application Service* concept to encapsulate and implement business use cases specific for REST endpoints when it is needed. The Application Service implements the logic required by the business use case by aggregating multiple services from the Domain Services Layer. The Application Layer also carries out the conversions from a domain object provided by the domain service to a special data transfer object, which is required by the

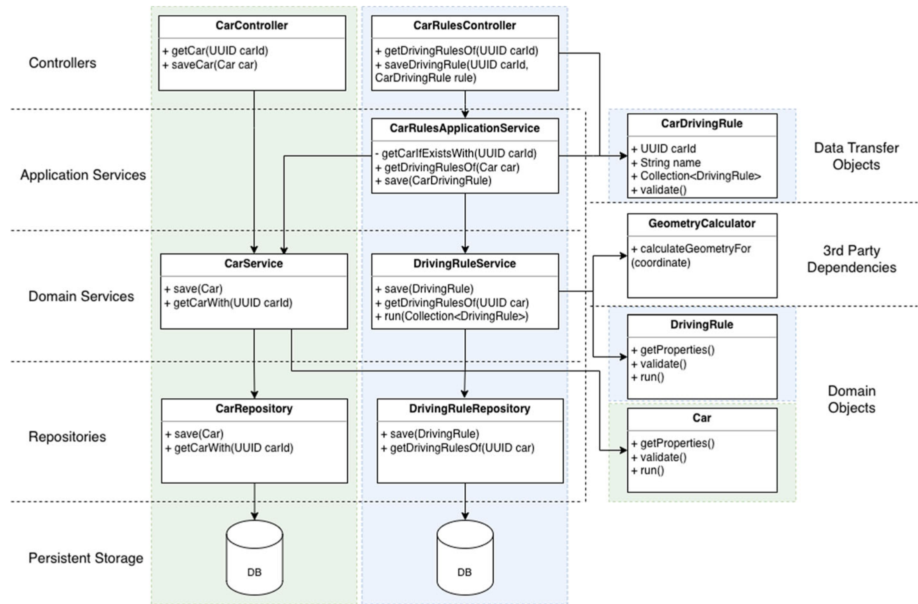

Fig. 5 The Software Architecture of the backend after DDD refactoring with classes, their relationships and the Bounded Context (with color codes). Green marked Bounded Context is *Car Context* and blue is *Driving Rules Context*. The mentioned class names, variables and methods are examples and do not reflect the actual code due to IP restrictions

controller depending on the return data defined by API specification. An example usage of the Application Service can be seen in Fig. 5.

6.3.2 Domain Service

Evans thinks that the service is a stateless layer implemented when there is a significant process or transformation in the domain is not the natural responsibility of a domain object or value object (Evans 2003). By following Evans' description, we defined our domain services as places that have a certain part of the domain knowledge, and implements a logic required by the domain, but not a responsibility of a single domain object. That implementation can be an aggregation of multiple domain objects or value objects, or orchestration and computation of a certain domain logic that involves repositories or external dependencies. As it can be seen on Fig. 5, our implementation of domain services is not very much different from a service concept in microservices. However, they are self-contained and responsible for only a certain domain logic.

6.4 Repositories

Evans suggests using the *Repository Pattern* to encapsulate data storage access to utilize more model-driven focus (Evans 2003). In the concept described by Evans, repositories are extensions to Domain Objects, which adds the ability to communicate data source to the Domain Object directly. Although, in our backend data source communication are already encapsulated in the *Repository Layer* and there are no any hard dependency between Domain

Objects. We carried out this approach to make our refactoring changes more compliant with the current architecture and frameworks that the backend uses, but we redefined the design and responsibilities of our repositories with a more domain focused approach.

With our domain-driven refactoring, we have introduced resource-specific repository implementation to encapsulate the context of the repository as can be seen in Fig. 5. The refactored repository implements the communication with the persistent data source and creates, reads, updates and deactivates resources. The refactored repository only implements CRUD operations, and it does not contain any core domain logic or business use case logic. They are consumed by services when data storage access is required for the core domain logic implemented by the service. The repository also creates instances of concrete domain objects while reading them from the data source.

6.5 Domain Objects

The Domain-Driven Design terminology defined by Evans describes domain objects as objects from the business specific area that represent something meaningful to the domain expert. Domain objects are mostly represented by entities and value objects. In our case, the backend already defined immutable objects to capture and represent domain-related information. However, we redesigned our domain objects according to *Ubiquitous Language* and added more responsibilities to our objects. With this approach, we aimed to capture the domain-related information and constraints better and encapsulate them to have a single source of truth. To achieve this idea, we have extended our domain objects with the knowledge of business domain constraints and validations. We have also encapsulated domain related state transitions and computations in domain objects. With this approach, we have introduced a reliable single source of truth for business domain representation in the entire backend code.

6.6 Data Transfer Objects

We extended the responsibility of our Data Transfer Objects by moving constraint validations and conversions from controllers to DTOs. With this approach, our DTOs became smarter, compliant with SOLID principles and their behaviours are strongly tested. Controllers also benefit from this improvement because we utilized DTOs to capture parameters and constraints specified by our backend's API specification. Since DTOs implements their own validations and conversions both for representation of incoming and outgoing data, we shrink the implementation surface of controllers more.

6.7 Bounded Context

A bounded context is a central pattern in DDD (Evans 2003). It is a delimited area in which a particular domain model applies. In our refactoring, we have implemented a bounded context for each of our domain services (Section 6.3.2). We cannot disclose the actual architecture, however as an example, we can talk about imaginary *Car Service* and *Driving Rule Service*. Each of these services has its own database, controller and REST API, which helps to isolate the data and operations within that service's bounded context. This means that each service can evolve and be modified independently of the others, without affecting the overall system.

In addition to these individual bounded contexts, we also have Application Services (Section 6.3.1) that aggregate data from multiple bounded contexts. For example, we might have a *Car Rules Application Service* that uses both the *Car Service* and the *Driving Rule Service* to provide which autonomous driving rules that particular car has to users. This relationship between the individual bounded contexts and the application services that aggregate them can be represented visually, as shown in the Fig. 5.

In the figure, the individual bounded contexts are represented by colored boxes, with the name of the service and its corresponding database and API. The application services are represented by arrows, indicating the flow of data between the bounded contexts. For example, the *Car Rules Application* service is shown as an arrow between the *Car Service* and the *Driving Rule Service*. This indicates that the car rules application service uses data from both of these bounded contexts to provide driving rules of a particular car to users.

By implementing our services as separate bounded contexts and using application services to aggregate them, we are able to clearly define the boundaries of each domain and ensure that the data and operations within each service are self-contained and independent of the others. This allows us to make changes to one service without affecting the others, and helps to ensure the overall maintainability and scalability of our application.

6.8 Intention-Revealing Interfaces

In his book, Evans (2003) suggests using *Intention-Revealing Interfaces* to capture meaningful domain logic in the code, to make the effect of the code clearly understandable to the user and anticipate the effect of a change while consuming the implementation. In order to ensure the backend code is easily understandable and maintainable by developers, we named methods in the backend the same as their domain-level operations defined by our *Ubiquitous Language*.

6.9 Tests

As a part of our refactoring efforts, we also reworked *Unit Tests* and *Integration Tests* of the backend in order to improve them and match them with the new design. Since we kept the mechanism and the model separated in the backend as Evans suggested in his book, now it becomes possible to unit-test our components such as domain objects, core domain logic and services, business use cases and controllers in isolation. With using the advantage of remarked improvements, we have introduced and reworked existing unit tests to verify our core domain logic and constraints in isolation against possible expected and unexpected cases. With this approach, we have achieved well-tested and verified strong core domain logic in the entire codebase. We have also introduced *Blackbox Tests* around our unit of services and components to ensure and verify their behaviour while working together.

7 Results and Evaluation

In this section, we are reporting the results and evaluation regarding the refactoring changes based on our questionnaire, interviews and think-aloud protocol data. We report our results and assessment following the order of the research questions as presented in the introduction.

In previous sections, we described how we identified problems and what they are. We reported the thought process of how DDD was selected as a methodology for our refactoring

efforts. We also reported how we implemented our solution in the studied software system. Therefore, the information we have shared in previous sections is answering to our **RQ1**. Thus, we will share our findings for the rest of our research questions in this section.

RQ2.1: How does DDD-based refactoring affect maintainability of the system, as perceived by the developers?

To answer our RQ2.1, we are interpreting the results of our interviews, mTAM questionnaire and think-aloud methodology. Since we are primarily emphasising the software system's maintainability, the structure of our reporting is based on the metrics defined by ISO/IEC 25010 standardization (Section 3.5.3).

According to the answers of our participants to **understandability** aspect, the DDD refactored code is more understandable in most cases due to its semantics and business-oriented architecture. Participants mentioned that they can easily follow the logic and connections between classes and methods because the implemented business case is more reflected in the code in software architecture, class responsibilities and selection of method and variable names. They also mentioned that the state of encapsulation and code re-usability after refactoring allows them to follow the code more easily because they are implemented in the respective scope once, and it is standardized in the entire code base. As we can see in Table 4 statements regarding tests are also scored high which shows that tests after DDD refactoring are easier to understand. One of our participants mentioned, *"the code actually consists of scenario actions, it speaks for itself. Once you know the business scenario, understanding the code is much easier. The logical division of the implementation is also very helpful to understand what is happening."* and it can be seen that the close relationship between the business scenario with the code plays an important role.

Four out of five participants were very positive on **modularity** aspect. They commonly mentioned challenges they were having with the previous code when they had to modify some parts of the implemented business logic and compared it with the DDD refactored code to recognize and appreciate introduced improvements. According to their answers, when there was a small change with a business case at the product level, the effect and footprint of the change they need to introduce were very big because of the monolithic structure of the old code base and poor encapsulation. According to participants, after DDD refactoring *"responsibilities and dependencies between classes are improved and logically correct"* so when they modify business logic, they can easily fit the changed scenario to the refactored code. The opinion of the new joiners also confirms that the modularity of the refactored

Table 4 Overview of the identified positive statements after the card sorting session of the semi-structured interview with 5 experienced engineers

Positive Statement	Participants
Code re-usability	5
Creating and maintaining tests are easier	5
Less mocks	5
Less refactoring	5
No code duplication	4
Good separation of concerns	3
Clear responsibilities	3
Useful domain objects	3
Following an industry standard	2
Logically correct	2

software system is high. According to the think-aloud session results shared in Table 6, all the new joiners agreed on statements such as *"No code duplication"* and *"Code re-usability"*, which positively contributes to modularity results.

It is seen that the answers of our participants are not unanimous for the **changeability**. The common opinion of participants is when they introduce a change, it is not affecting the rest of the implementation as it was before. This is a very important statement for us to conclude that DDD improved the changeability aspect. Nevertheless, we have noticed a difference in opinions about the difficulty of introducing a change. Participants who have more experience with OOP principles and especially a prior experience with DDD are more positive about the easiness of introducing changes in the refactored code. Participants with less or no prior experience with DDD finds the old system more straightforward to change, even though they think DDD introduced a lot of improvements. Considering the results of the changeability question, we think that the experience level is a very important factor to cope with the extra complexity introduced by DDD. Engineers who have less OOP experience or no prior experience with DDD might be overwhelmed with the introduced extra layers, the number of classes and methods involved for one business logic and their relations. On the other hand, engineers who have prior experience with DDD or define themselves as comfortable using it are not affected by the introduced complexity, thus, they are not having difficulties while introducing a change.

All of our participants gave very positive answers to the question related to **testability**. This fact is also visible in our interview results. As can be seen in Table 4 all participants of our semi-structured interview are mentioned creating and maintaining tests are easier, and they are maintaining fewer mocks after the refactoring. Our participants mostly mentioned scenarios such as the following: with the DDD refactored code, they can test and verify the core domain logic once, and then they do not have to test it for different scenarios again. They also create and maintain fewer mocks because of the improved architecture and good separation of concerns and encapsulation. We conclude that improved architecture and encapsulation make engineers able to create more self-contained tests in a focused context and fewer mocks improved the test maintainability (Table 5).

As it is shown in Table 4, all of our participants made positive statements about **reusability**. Our participants mostly mentioned the introduction of Core Domain Services and Smart Domain Objects have extremely improved the code reusability. Participants recognize the situation that previously they had to duplicate the code if they had to use some already implemented logic. Participants state that the introduction of DDD enabled them to reuse part of the implementation when it is needed. The results of our think-aloud sessions with new joiners, which can be seen in Table 6, also shows that code reusability was mentioned positively by all of our participants. During our interviews, we encountered statements such as, *"I remember I wrote the same logic three times to retrieve users from the database in different services. Can you imagine? Now we are using only one service which handles*

Table 5 Overview of the identified negative statements after the card sorting session of the semi-structured interview with 5 experienced engineers

Negative Statement	Participants
Requires experience	4
High level of abstraction	3
Not easy to understand	3
Logic is too much divided	2
Too much opinionated	2

Table 6 Overview of the identified positive statements after the card sorting session of the think aloud session with 2 experienced new joiners

Positive Statement	Participants
No code duplication	2
Code re-usability	2
Feels correct	2
Separate repositories	1

users and we are reusing it. This is how it should be!" which is a clear indication that the introduction of DDD improved the reusability in the entire product code.

We have also conducted a **mTAM Questionnaire** to quantitatively measure the impact of our DDD refactoring. Participants who were already part of the development team and had experience on code before refactoring and after DDD refactoring answered to questions can be seen in Table 7. Using the questionnaire results, we computed mTAM scores for PU and PEU, applying the approach in Lah et al. (2020), which is as follows:

$$PU = (AVG(TAM01, ..., TAM06) - 1)(100/6)$$
$$PEU = (AVG(TAM07, ..., TAM12) - 1)(100/6)$$

As a result of the computation, DDD refactoring scored 85 in PU and 83 in PEU, leading to an overall mTAM score of 84 out of 100. Lah et al. (2020) do not specify a way to interpret the mTAM values. However, mTAM is designed to be similar to the System Usability Scale (SUS), which is a robust and reliable tool for measuring the usability (Brooke 2020). Due to this fact, we use corresponding guidelines to interpret SUS scores proposed by Aaron et al. (2009). In reference to the scale proposed by Aaron et al. in Fig. 6 our refactoring with DDD ranks *acceptable* on the acceptability scale, *B* on the grade scale and *good* on the adjective rating scale.

Interpreting the total PU and PEU results of the mTAM questionnaire shows that engineers recognise the positive gains of the DDD refactoring and find it helpful overall. However, they also think that using DDD requires effort to get into it. We believe that the reasons for this can be found in Tables 4 and 5, where we published the card sorting results of the semi-structured interview sessions.

When we interpret individual results of the mTAM questions in Table 7, we can state that engineers perceive DDD as an approach that enables them to accomplish software development tasks more quickly, increases their productivity and enhances their effectiveness during implementation tasks. These statements are similar to the advantages of having a highly maintainable software (Martin 2000). We believe that positive results of the the semi-structured interview (Table 4) and think-aloud sessions (Table 6) can be considered as reasons. We can also see that the lowest scoring questions are only related to the easiness of learning and easiness of using DDD, which also correlates with the results of our semi-structured interviews and think-aloud sessions.

RQ2.2: What are the challenges of refactoring using DDD?

During this study, we found that there are several challenges to using DDD. These challenges are mostly related to adopting and perceiving the DDD principles by engineers and technical challenges such as using DDD in cooperation with other opinionated frameworks (Dinh-Tuan et al. 2020) which are already present in the code. Our findings regarding the challenges of using DDD are as follows:

Onboarding engineers to DDD is not easy Even though our participants are positive about DDD refactoring, the commonly mentioned aspect was DDD introduces a lot of encap-

Table 7 Our mTAM Questionnaire with the likert scale results of individual questions. Answers to questions 1 to 12 are on a scale of 1-7, and question 13 is on a scale of 1-10

Question	Result
Perceived Usefulness (PU)	
1. Using DDD enables me to accomplish software development tasks more quickly.	6
2. Using DDD improves my job performance.	6
3. Using DDD during development increases my productivity.	6
4. Using DDD enhances my effectiveness on the implementation tasks.	6
5. Using DDD makes it easier to do implementation.	6.25
6. I have found DDD useful in implementation.	6.25
Perceived Ease-of-Use (PEU)	
7. Learning to implement with DDD was easy for me.	5.75
8. I found it easy to get DDD to do what I want it to do.	6
9. My interaction with DDD has been clear and understandable.	6
10. I found DDD to be flexible to interact with.	5.5
11. It was easy for me to become skilful at using DDD.	6
12. I found DDD easy to use.	5.75
Ending	
13. How likely would you recommend DDD to others?	8.5

sulation and abstraction levels. Our participants stated that it is challenging for the developers who had no prior DDD experience to understand the dependency between classes, methods and how certain business logic is divided. Our participants also stated that, especially at the beginning, participants were having challenges in understanding how they are going to implement the solution they have in their mind and where to start. This fact can be also seen in Table 5, the negative statements are mostly about experience, abstraction and understandability. The answers of our participants to questions regarding challenges are consolidated around the fact that engineers without prior DDD experience have difficulties with the onboarding process, but after some time they start to think that DDD brings undeniable improvements to the software. On the other hand, engineers with prior DDD experience do not encounter difficulties during the onboarding, and they grasp the idea more quickly. Participants who had difficulties during the onboarding process stated that they coped with the situation by having training from online materials and studying the example implementation we have introduced as part of this study.

Refactoring in a codebase that is in development phase is challenging We carried out our study on a product that is in the development phase. Due to new business requirements and goals, the development team was constantly adding new features and also changing most of the business logic very often. This also affected our refactoring efforts from time to time.

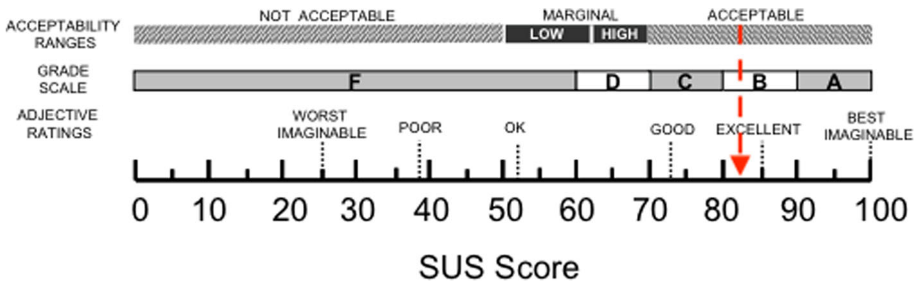

Fig. 6 A comparison of the adjective ratings, acceptability scores, and grades in relation to the average SUS score in Bangor et al. (2009) and the placement of our overall mTAM score

Our solution was carrying out our refactoring efforts in an isolated environment, which is also suggested by Evans, and then working on feature parity before we deprecate the old implementation and use the refactored one.

Implementing DDD exactly how it is described in the guidelines is not always possible if you are using modern frameworks Since the publication of Evans' book, there were a lot of changes and improvements in programming languages with the introduction of modern frameworks. The product we were introducing our refactoring was using several frameworks such as Spring Boot[1], Lombok[2] and jOOQ[3]. Some of them are very opinionated frameworks, and they create their abstraction, even inject their DSL into the code. Because of their architectural and coding style requirements, sometimes it was not fully possible to implement DDD as described in the book. Evans also recognizes this situation in this book, saying "In general, don't fight your frameworks. Seek ways to keep the fundamentals of domain-driven design and let go of the specifics when the framework is antagonistic." (Evans 2003). In this study, we also applied the same mentality, and we did not fight with the frameworks in the source code. We used them to implement the fundamentals of DDD in ways that they allowed us.

8 Lessons Learned

Our findings show that using an opinionated design approach such as DDD is a good tool to solve common software development issues, and it brings standardization to the source code. This standardization provides a clean structure and also guidance to software engineers developing the product. This guidance can be related to understanding how to design their solutions while implementing a new feature, as well as extending the existing features. As we reported the results in Section 7, software engineers who work for the subject product also appreciate the standardization we have introduced with the DDD approach. Engineers value adhering to a design methodology that is accepted in the industry, despite the fact that they acknowledge DDD as an opinionated approach.

We understood that DDD is a valid option for developing software systems in any situation where the business domain of the problem is complex and needs to be well understood in order to create effective software. DDD is a software development approach that focuses

[1] https://spring.io/projects/spring-boot

[2] https://projectlombok.org

[3] https://www.jooq.org

on the business domain of the problem that a software system is being built to solve. It is a way of thinking and designing software systems that aligns the code with the business domain, making the code easier to understand and maintain. By using DDD, companies can create software that is better aligned with the business domain, making it more effective at solving the problem it was designed to address. Additionally, using DDD can make it easier for developers to understand the business domain, which can help them create more maintainable code. Overall, DDD is a valid option in situations where the business domain is complex and needs to be well understood in order to develop effective software.

We also understood that DDD comes with a certain level of complexity and cognitive load, which creates a challenge for software engineers. Depending on engineers' prior experience with the OOP principles and especially with DDD, it can range from significantly challenging to easy to adopt. We think that is a crucial aspect that needs to be taken into account in an industrial setup where there are tight commitments.

One of our experiences is that refactoring with DDD is not limited to source code refactoring, it is also a refactoring on the software architecture. Due to its magnitude, it is very clear to us that DDD refactoring will create a significant amount of workload depending on the size of the subject codebase. This is surely a challenging aspect from both software development and business point of view. Refactoring efforts should be planned and executed very carefully.

8.1 Learning for the Company

To share our research and lessons learned, we set up multiple presentation sessions inside the organization. We discussed the benefits of using ubiquitous language to facilitate communication between technical and non-technical staff members and to match business cases with source code. Following the completion of our post-refactoring measurements, we hosted presentations and study sessions with the subject team's engineers to further elaborate on DDD, our refactoring and findings. We talked about how to put it into practice and found solutions to issues raised during the pre-refactoring data collection phase. We also held presentations in a company-wide environment to impart knowledge and share our findings and recommendations. More than 800 people attended our company-wide presentation sessions, which were held at the product unit and company levels.

The subject team came to the decision to continue using DDD principles in the software system's future development. They will keep refactoring to enhance their codebase and fully adhere to DDD principles. They also stated that they would plan their future activities and work while taking the DDD difficulties we reported into consideration.

8.2 Recommendations for Other Companies

Considering our learnings, we would like to recommend to companies, who would like to migrate to DDD principles, to consider the size of their codebase, experience of their engineers, their project commitments, requirements and budget before they commit to applying DDD refactoring.

We also recommend that companies be aware of the level of complexity of DDD principles. To ensure the quality of DDD refactoring and its corresponding architecture over time, companies should provide the necessary tooling and knowledge to development teams.

9 Threats to Validity

Our study is subject to several threats to validity with the classification scheme of Wohlin et al. (2012), which are elaborated in this section.

Conclusion validity We conducted our study in a software development team consists of small group of engineers. That means our conclusions based on the data collected from the small group of participants using interview and questionnaire methodologies. The amount of participants could raise the concern in terms of reliability of measures and random heterogeneity of respondents. However, according to the Empirical Standards of Action Research[4] it is the nature of an action research to have a small group of participants, because it is focused on a particular context and setting.

It should be noted that the participants of this study were software engineers who already had knowledge and experience with the subject codebase before the study. However, it should also be acknowledged that the engineers had varying levels of experience in the industry as well as with DDD. It is important to note that some participants may have had more or less experience with DDD, which could potentially impact their perceptions and responses during the study. Therefore, the construction of this study may be influenced by the varying levels of experience with both the software industry and DDD among the participating software engineers. However, efforts were made to ensure that all participants had sufficient familiarity with the subject codebase and DDD concepts to allow for meaningful and informed feedback.

Internal validity During our study, refactoring with DDD was not an architectural or business decision of the studied product. We conducted our study in an environment where new features and code changes were constantly introduced. Thus, we carefully isolated actions that we took as part of this study to ensure that observed results of the study will not be caused by other activities, not the one intended. We introduced our refactoring in an isolated component, a version of an Anti-Corruption Layer (Evans 2003) in the codebase (Section 6), which was not affected by the changes introduced to the rest of the system.

Our questions in the semi-structured interviews were carefully selected to gather engineers' own experiences and thoughts without any external effect caused by the study and also the action team. To eliminate possible biases caused by designed questions, we also utilized an mTAM questionnaire to verify our findings quantitatively.

To eliminate the maturation effect during our measurements, we also interviewed software engineers who recently joined the product team, who had a similar amount of exposure to the old and refactored code. Before our conclusions, we carefully compared the results that we have gathered from software engineers who were already working on the product and new joiners.

Finally, during our study, we utilized and used the same development environment, tooling, coding guidelines and business requirements with the old codebase. None of the instruments was changed over time.

Construct validity To identify problems without any bias, we first carefully explored them with the group of engineers in the product team, following focus group guidelines (Kontio et al. 2004). We created a discussion environment with subject-matter experts to get their expert opinions to identify their common problems regarding the codebase. We semantically analysed results by applying the attribution analysis methodology (Janis 1965), which examines the frequency with which certain characterizations or descriptors are used. As the action team, we reviewed the subject codebase to construct our own expert opinion

[4] https://acmsigsoft.github.io/EmpiricalStandards/docs/?standard=ActionResearch

(Section 4), identified problems in the codebase and matched them with the ones raised by engineers in the team (Section 5).

External validity We have conducted our study in one specific team and codebase. This means that we study only one specific instance of the setting. This is due to several important aspects of this study: (1) it is an action research study, which is focusing on a specific problem domain and actions by its design, (2) it is very difficult to study more teams and validate our findings there due to its industrial environment, where there are decisive concerns such as business commitments, decisions and tight schedules.

Our action research study conducted in a specific context by its nature and does not control the external environment (Staron 2020). This means that we cannot fully claim the generalizability of our findings and results to all kinds of projects and companies. However, it is worthwhile to emphasize that this study executed in an industrial context directly and not in an isolated laboratory environment. We hypothesize that provided one takes certain actions to address unique situations in different projects and companies, this study can be applicable to other projects and companies as well. As suggested by the action research guidelines (Staron 2020), we discuss learnings and recommendations for other companies in Section 8, which can help apply a similar study in a different context.

10 Related Work

Action Research in Domain-Driven Design At the time we were conducting this study, there were only a handful of published action research studies in DDD (Oukes et al. 2021; Wade and Salahat 2009; Braun et al. 2021; Kapferer and Zimmermann 2020). The studies we have identified were classified as action research studies, however, we noticed that none of them followed Staron's action research approach (Staron 2020). These studies have topics that are mainly focused on technical areas, such as creating systems and frameworks using DDD. Their approach is usually based on case studies and reporting the final implementation and researchers' experiences during the utilization of DDD. To the best of our knowledge, none of these studies methodologically measures or validates engineers' perception of DDD, either in a controlled or an industrial environment. Instead of publishing a technical report and measurements, our study aims to capture and evaluate how a software development team perceives DDD changes, by integrating those engineers into the process by utilizing the action research methodology.

Architectural Refactoring with Domain-Driven Design There are several studies and books available regarding architectural refactoring with DDD with Micro-service Architecture (MSA). One study employed a Model-Driven Design approach to software developed with MSA and proposed DDD as an approach to identify microservices (Rademacher et al. 2018). Their proposed example had two microservices, however, the methodology they applied to identify their services was not covered in their study.

A real-life study can be found in Kapferer and Zimmermann (2020) where researchers proposed a DDD approach for a microservice-oriented architecture. They also followed an approach where they separated concerns and responsibilities by introducing domain objects, domain services and aggregates. They have also introduced new concepts on top of Evans' and Fowler's DDD approach to be able to fit DDD principles into their reality. They proposed a tool to help software engineers and architects to design DDD applications by utilizing UML diagrams, however, they did not test their proposed DDD approach with its users.

Another real-life study related to DDD and MSA can be found in Hasselbring and Steinacker (2017). This study proposes the best practices applied to one of the biggest e-commerce websites in Europe. In their architecture, there is a single-bounded context supported by several domain services. However, they did not explain the purpose of their domain services clearly.

There are also studies focusing on the extraction of microservices from monolithic applications and leveraging DDD as an architectural design approach. One noticeable study is Mazlami et al. (2017) where researchers tried three different approaches to refactor a monolithic application to MSA. They designed their refactoring based on (1) grouping classes that are regularly changing together, (2) grouping classes that are changing due to the same contributors, and (3) grouping classes based on DDD principles. Out of three methods, researchers found that DDD gave the best result.

Considering that microservice architecture is also a fairly new architectural approach (Carrasco et al. 2018), the literature is quite hollow with studies that combine microservices with DDD architecture. The architectural refactoring proposed in this study is designed based on Evans' (Evans 2003) and Fowler's (Fowler 2003) DDD suggestions, with taking the state and requirements of the studied product into account. We tried to apply their DDD suggestions to a microservice-based product with our best abilities to explore its possibility and identify challenges in the technical and human aspects with scientific measurements.

11 Conclusions and Future Work

In this research, we explored several aspects of refactoring with DDD. We investigated (1) how we can improve maintainability using DDD, and (2) what would be the challenges of DDD refactoring in an industrial context, including engineers' perception of the DDD. We did an action research study in a company with real-world conditions and performed refactoring on software which is actively used by the customers and still in the active development phase. We have collected engineers' opinions on this effort by conducting semi-structured interviews, think-aloud sessions and an mTAM questionnaire.

We identified that utilizing DDD improved the software maintainability according to engineers' perception. Engineers gave positive answers to our interview questions, which are mapped to software maintainability metrics defined by ISO/IEC 25010 Software and Data Quality Standard. The results of the mTAM questionnaire also support the improvement. Our DDD refactoring scored 85 in PU and 81 in PEU, leading to an overall mTAM score of 83. This means our refactoring with DDD ranks *acceptable* on the acceptability scale, *B* on the grade scale, and *good* on the adjective rating scale.

We have experienced that using a design approach such as DDD has a good tool to solve common software development issues, and it brings standardization to the source code. Software engineers who are working on the subject product also appreciate the standardization we have introduced with the DDD approach. We understood that DDD comes with a certain level of complexity and cognitive load, which creates a challenge for software engineers. We also understood that refactoring with DDD is not limited to source code refactoring, it is also a refactoring of the software architecture. Due to its magnitude, it is very clear to us that DDD refactoring will create a significant amount of workload depending on the size of the subject codebase. Due to this fact, companies who are planning to work on such a refactoring effort should create a plan carefully before the execution.

This action research was conducted on one product and software development team using particular software development procedures and technology stack. Hence, one of the future work for extending this research might be applying our refactoring ideas and strategy to a different but similar software product to discover its outputs. With this action research, we emphasised the human factor of our change to explore reactions and acceptance of such change in the engineering team. We based our conclusions upon the data gathered by software engineers working on the studied product through interviews and questionnaires. Thus, another future work for extending this research might be measuring these changes with tools such as static code checkers to gather more quantitative data, analyse the output and compare it with the findings of this research.

Acknowledgements We want to thank our subject company and engineers in the software development team for their contribution and support during this study. We also would like to thank Alexander Serebrenik and Michel Chaudron for their support and TU/e Ethical Board for their approval to conduct and publish this study.

Data availability The datasets generated during and/or analysed during the current study are not publicly available due to company intellectual property restrictions but are partially available from the corresponding author on reasonable request.

Declarations

Ethics approval and consent to participate This study is approved by the TU/e Ethical Review Board.

Consent for publication All the participants in this study anonymously took part in this study with their consent.

Conflict of interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Abid C, Alizadeh V, Kessentini M, Ferreira TDN, Dig D (2020) 30 years of software refactoring research: a systematic literature review. arXiv preprint arXiv:2007.02194

Bangor A et al (2009) Determining what individual SUS scores mean: adding an adjective rating scale. J Usability Studies 4(3):114–123

Bjørner D (2017) The Tryptych of Software Engineering, Software Engineering 3 - Domains, Requirements, and Software Design, I. Springer Verlag

Braun S, Bieniusa A, Elberzhager F (2021) Advanced Domain-Driven Design for Consistency in Distributed Data-Intensive Systems. In: Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '21), Association for Computing Machinery, New York, NY, USA, Article 9, pp 1–12

Brooke J (2020) SUS: A "Quick and Dirty" Usability Scale. Usability Evaluation in Industry, 207–212

Brydon-Miller M, Greenwood D, Maguire P (2003) Why Action Research?

Cagan M (2017) INSPIRED: How to Create Tech Products Customers Love. Wiley

Carrasco A, Bladel BV, Demeyer S (2018) Migrating towards microservices: Migration and architecture smells. In: Proc. 2nd Int. Workshop Refactoring, Montpellier, France, pp 1–6

Davis FD (1989) Perceived usefulness, perceived ease of use, and user acceptance of information technology. MIS Quarterly: Management Information Systems 13(3):319–339
Dinh-Tuan H, Mora-Martinez M, Beierle F, Garzon SR (2020) Development frameworks for microservice-based applications: Evaluation and comparison. In: Proceedings of the 2020 European Symposium on Software Engineering
Domain (2022) In: Oxford Online Dictionary, Retrieved from https://en.oxforddictionaries.com/definition/domain
dos Santos PSM, Travassos GH (2009) Action Research Use in Software Engineering: An Initial Survey, In Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, IEEE Computer Society pp 414–417
dos Santos PSM, Travassos GH (2011) Action Research Can Swing the Balance in Experimental Software Engineering, In: Advances in Computers, vol 83. Elsevier, pp 205–276
Evans E (2004) Domain-Driven Design: Tackling Complexity in the Heart of Software. Pearson Education
Fowler M (2003) AnemicDomainModel. Retrieved from http://www.martinfowler.com/bliki/AnemicDomainModel.html
Fowler M (2003) AnemicDomainModel. Retrieved from http://www.martinfowler.com/bliki/DomainDrivenDesign.html
Fowler M (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA
Hasselbring W, Steinacker G (2017) Microservice architectures for scalability, agility and reliability in E-commerce. In: Proc. IEEE Int. Conf. Softw. Archit. Workshops (ICSAW), Gothenburg, Sweden, pp 243–246
ISO/IEC 25010 (2011) Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. https://www.bibsonomy.org/bibtex/25951b0998b7eaea346d826fd77110a48/bcoldewey
Janis, IL (1965) The problem of validating content analysis, In: Lasswell HD, Leites N & Associates (Eds.) Language of politics
Kapferer S, Zimmermann O (2020) Domain-Driven Service Design. In: Dustdar S (ed) Service-Oriented Computing, SummerSOC, Communications in Computer and Information Science, vol 1310. Springer, Cham
Kontio J, Lehtola L, Bragge J (2004) Using the focus group method in Software Engineering: Obtaining practitioner and user experiences, ESEM'04, pp 271–280
Lah U, Lewis JR, Šumak B (2020) Perceived usability and the modified technology acceptance model. Int J Human-Comput Interaction 36(13):1216–1230
Landre E, Wesenberg H, Olmheim J (2007) Agile enterprise software development using domain-driven design and test first. Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion (OOPSLA '07). Association for Computing Machinery, New York, NY, USA, pp 983–993
Lehman MM (1980) Programs, life cycles, and laws of software evolution. In: Proceedings of the IEEE, vol. 68, no. 9, pp 1060–1076
Martin RC (2000) Design principles and design patterns. Object Mentor 1(34):597
Mazlami G, Cito J, Leitner P (2017) Extraction of microservices from monolithic software architectures. In: Proc. IEEE Int. Conf. Web Services (ICWS), Honolulu, HI, USA, pp 524–531
Mens, T (2016) An ecosystemic and socio-technical view on software maintenance and evolution. 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE
Mens T, Tourwé T (2004) A survey of software refactoring. IEEE Transactions on software engineering 30(2):126–139
Moiseev A, Fain Y (2018) Angular Development with TypeScript. Simon and Schuster
Oukes P, van Andel M, Folmer E, Bennett R, Lemmen C (2021) Domain-Driven Design applied to land administration system development: Lessons from the Netherlands, Land Use Policy, vol. 104
Rademacher F, Sorgalla J, Wizenty PN, Sachweh S, Zündorf A (2018) Microservice architecture and model-driven development: Yet singles, soon married (?). In: Proc. 19th Int. Conf. Agile Softw. Develop., Companion, Porto, Portugal, pp 1–5
Reason P, Bradbury H (2001) Handbook of Action Research: Participative Inquiry and Practice. Sage
Roy CK, Cordy JR (2007) A survey on software clone detection research. Queen's School of computing TR 541(115):64–68
Schach SR (2011) Object-oriented and classical software engineering. McGraw-Hill Companies
Staron M (2020) Action Research in Software Engineering: Theory and Applications. Springer
Tirkkonen-Condit, S (1990) A Think-Aloud Protocol Study, Learning, Keeping, and Using Language: Selected Papers from the 8th World Congress of Applied Linguistics, vol. 2. John Benjamins Publishing Company

Van Deursen A, et al. (2001) Refactoring test code. Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001). Citeseer
Wade S, Salahat M (2009) Application of a Systemic Soft Domain-Driven Design Framework, Proceedings of the World Academy of Science, Engineering and Technology
Wohlin C, et al (2012) Experimentation in software engineering. Springer Science & Business Media
Zimmerman DE, Akerelrea C (2002) A group card sorting methodology for developing informational web sites, IEEE International Professional Communication Conference, IEEE

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.