Modernization of Monolithic Legacy Applications towards a Microservice Architecture with ExplorViz

Master's Thesis

Stephan Lenga

June 26, 2019

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
M.Sc. Alexander Krause

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 26. Juni 2019

Abstract

With the rapid advance of the digitization in nearly every branch of industry, the scope and complexity of required software systems have reached new heights. In order to keep up and adapt to the ever-changing requirements of the fast-moving markets, it is often not enough to only rely on state-of-the-art technologies when developing software applications. Additionally, software developers have to adjust their development process for staying competitive. As a matter of fact, a monolithic software system, which was and still is a popular architecture style among software developers, tends to struggle with meeting these standards. Especially legacy software monoliths suffer from their over the time convoluted and tightly coupled inner structure. Their often outdated technology stack as well as their cumbersome deployment and delivery process pose a great risk for successful maintenance and further development. Consequently, within the last years, companies have been extensively investing in modernizing the software architecture and development processes of their products. By moving away from software monoliths towards the emerging microservice architecture style, an agile and robust software system with a compatible development process can be realized. However, this modernization can be highly challenging from a practical point of view. Available software solutions, which aim at supporting the restructuring of software monoliths into microservices, often do not satisfy the arising needs of the developers. While these products offer numerous static analysis functionalities, dynamic analysis aspects are mostly missing. Yet, especially a dynamic analysis can provide invaluable information about overlooked characteristics of the software system.

In this thesis, we investigate the beneficial impact of the live trace visualization tool ExplorViz when it comes to supporting developers during the software modernization process. Therefore, we combine the dynamic analysis toolkit of ExplorViz with commonly used static analysis tools for discovering a feasible microservice architecture within the monolithic online lottery application *in|FOCUS*. Based on relevant research results and fundamental domain-driven design concepts, a well-structured software modernization process is presented and executed. Furthermore, we exploit the self-contained systems architecture style as an intermediate step towards a microservice architecture. After discussing opportunities and challenges that are encountered during this modernization endeavor, we evaluate the supporting features of ExplorViz. To this end, a questionnaire containing qualitative open-ended questions is developed and used for conducting a guided interview with the software developers of the *in|FOCUS* application. Besides providing critical feedback on the current development state of ExplorViz, in particular its capabilities to support software modernization process, this evaluation gives rise to possible future development ideas for the live trace visualization tool.

Acknowledgments

I would first like to thank Prof. Dr. Wilhelm Hasselbring for giving me the opportunity to write my thesis at the Software Engineering Group of the Kiel University and for establishing the contact with the adesso AG. Moreover, I am grateful to Alexander Krause for his most valuable support and various interesting technical discussions.

I want to thank Uwe Lutter for making it possible to write this thesis within the scope of a Master's thesis internship at the adesso office in Hamburg. Furthermore, I would like to thank the members of the *in|FOCUS* development team for providing insights into their software framework and for participating in the evaluation process. In particular, I really appreciated the guidance provided by Dan Kröger.

Moreover, I would like to thank my parents and my brother for their loving support throughout my life, for their guidance and all the opportunities they have provided to me.

Contents

Chapter 1

Introduction

1.1 Motivation

Monolithic software systems that are being developed over a longer period of time not only tend to increase in complexity but also become more and more convoluted in their inner structure. Previously modular components become tightly coupled with one another [Thönes 2015]. Formally defined best practices and design decisions are being disregarded for certain reasons, that is, a lack of discipline or the necessity to reduce development expenses. Therefore, it becomes more challenging as well as more costly to continue developing and maintaining these systems. Their often outdated technology stack continuously rises the technical debt which leads to an increased risk for any future successful development. Since a technology update would affect the whole application and is therefore costly and time-consuming, a monolithic system is often unable to successfully and sustainably reduce its technical debt [Woods 2016].

Moreover, monolithic applications struggle when employing agile development and delivery processes which would increase the effective productivity and flexibility of developers [Bennett and Vaclav 2000]. Even small changes to a single part of the code can result in the redeployment of the whole application. Accordingly, this limits continuous deployment [Kalske et al. 2018]. When it comes to monolithic applications, it is not easy to provide additional resources for certain overloaded modules without duplicating the whole application by scaling horizontally. In certain high-stress situations, some parts of the system might have a much higher workload than usual. A common example are online shops during the Christmas season. The inability of a monolithic software system to scale efficiently in order to handle increasing workload increases operating cost [Newman 2015a]. Another downside of monolithic systems is its proneness to error propagation throughout large parts of the application.

Consequently, companies started to invest in modernizing their software systems and rebuild their applications with a more robust and flexible software architecture. The *microservice architecture* is an emerging architecture style with an rapidly increasing popularity among software engineers. Hence, it is often the target architecture for these modernization processes. Its concept is characterized by the interaction of modular, autonomous services called *microservices*. Each of these microservices implements a set of distinct features or functionality and collaborates with other microservices in providing some service opera-

tion. Their autonomous design avoids tight coupling with other services of the software system and streamlines individual development, testing, and deployment. As a result, agile software development concepts such as *DevOps* and *Continuous Delivery* are supported [Newman 2015a]. Besides, this modernization makes the automation of system building out of a version control system easier. The same applies to the automation of unit testing and performance benchmarking, as well as the automation of deployment in test and production environments. This significantly increases development productivity and strengthens the ability to react to changing requirements and is of utmost importance in order to stay competitive in today's software market.

Nevertheless, the modernization process of a monolithic software system to a microservice architecture is challenging and time-consuming. A bottom-up approach by starting over from scratch and reimplementing the whole application as microservices is rarely possible. Too many resources are spent while the actual functionality of the software system remains unaltered. Therefore, it is often more practical to incrementally decompose the monolith into microservices. However, this task is far from being trivial. The first step for the extraction of microservices is the identification of separate modules, each with specific responsibilities with explicit borders within the monolith, so-called *bounded contexts*. These bounded contexts will outline the organizational structure of the resulting microservice architecture. Afterwards, the implementation, migration and deployment of the microservices takes place [Knoche and Hasselbring 2018].

The software modernization process of a large-scale application is the focus of this master's thesis. In cooperation with the German IT consulting and software development company *adesso AG*, we analyze a large-scale lottery software application *in|FOCUS*[1]. By combining domain-driven design concepts, static analysis technologies, and dynamic monitoring solutions such as *Kieker*[2] and *ExplorViz*[3], a modernized software architecture of *in|FOCUS* is developed and thoroughly discussed. In particular, the above mentioned tools are used for discovering and visualizing the actual architecture of the *in|FOCUS* software system and monitoring its run-time behavior. The domain of the application is analyzed and its bounded contexts are identified. Each context serves as the basis for partitioning the legacy architecture into well-defined *self-contained systems*. As a next step, these self-contained systems are split up further into microservices with the help of ExplorViz.

Furthermore, the usability of the ExplorViz tool for discovering a microservice architecture from a monolithic software system is assessed with the help of qualitative guided interviews which are conducted in cooperation with a group of software developers working at adesso. This investigation does not only evaluate the current degree of support for modernization projects but also reveals new ideas for future extensions and possible improvements of ExplorViz.

[1] `https://www.adesso.de/de/branchen/lotteriegesellschaften/leistungen/loesungen/index.jsp`, accessed 29.05.2019
[2] `http://kieker-monitoring.net/`, accessed 15.04.2019
[3] `https://www.explorviz.net/`, accessed 15.04.2019

1.2 Goals of this Thesis

After analyzing the domain and the legacy architecture of the large-scale monolithic *in|FOCUS* software system, we aim at exploring a possible division into self-contained systems. In order to identify suitable boundaries for these independent services, we combine static analysis tools with the dynamic analysis and visualization capabilities of ExplorViz. A further division of the newly developed architecture into microservices is conducted by dynamically analyzing the behavior of the software application with ExplorViz. Finally, the ability of ExplorViz to support this kind of modernization process is assessed. A detailed exposition of these goal is provided below.

1.2.1 G1: Discovery of the Software System *in|FOCUS* and its Domain

In order to identify a practicable division of the legacy architecture into distinct services, the current code and database architecture of the monolithic application has to be explored. Therefore, the domain of the application should be statically analyzed and divided into bounded contexts with the help of static analysis and domain experts, as suggested by recent research. These bounded contexts will serve as a foundation for separating logic and data of the services from each other in order to create autonomous, self-contained components, each with its own independent logic and data.

1.2.2 G2: Identification of a Suitable Decomposition into Self-Contained Systems

Here, the goal is to identify a possible decomposition of the existing architecture in order to restructure it into well-defined self-contained systems. A suitable modernization process, which should be based on up-to-date expert literature will be chosen and presented to the reader. Static analysis tools should provide the necessary insight into the legacy architecture of *in|FOCUS*.

1.2.3 G3: Refinement of the Self-Contained Systems into Microservices

We aim at discovering possible microservices within the previously defined self-contained systems. The live trace visualization tool ExplorViz will be the main tool to support this step by monitoring and dynamically analyzing the behavior of the analyzed software application.

1.2.4 G4: Assessment of the Ability of ExplorViz for Exploring Self-Contained Systems and Microservices

This thesis will provide an assessment of the supportive features of ExplorViz for exploring feasible boundaries of self-contained systems or microservice within a large-scale mono-

lithic software system. Hence, it will be investigated how ExplorViz helps in overcoming certain challenges of the software modernization process. To this end, software developers with in-depth knowledge of the *in|FOCUS* application should be interviewed with an appropriately chosen questionnaire. It can be assumed, that invaluable information on possible improvement of ExplorViz is gained by giving the participants the opportunity to utilize ExplorViz for performing certain architectural analysis tasks themselves.

1.3 Document Structure

The remainder of this thesis is organized as follows. Chapter 2 gives a brief overview on the technologies which will be used throughout this thesis. Chapter 3 outlines the approach of how to achieve the previously described goals. It presents the software modernization process of *in|FOCUS* as well as the derived architecture decomposition. It is argued that this decomposition is indeed suitable for the *in|FOCUS* application. Next, Chapter 4 presents the assessment of ExplorViz capabilities to support the software modernization process of transforming a monolithic software system into a microservice architecture. The assessment setup and the results of the conducted qualitative interviews are discussed. Furthermore, Chapter 5 presents projects that went through a similar modernization process but utilizing analysis tools other than ExplorViz. Chapter 6 concludes the thesis and presents an outlook on future work.

Chapter 2

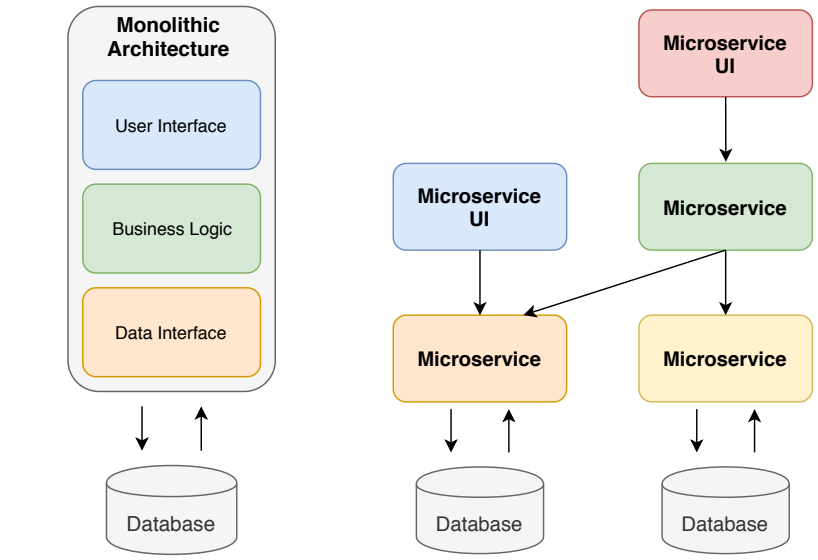
Foundations and Technologies

In this chapter, the reader is introduced to the key technologies and theoretical foundations which are relevant to this thesis. First, Section 2.1 and Section 2.2 introduce main ideas and characteristics of a monolithic as well as a microservice architecture. Then, Section 2.3 familiarizes the reader with the benefits of self-contained systems when it comes to modernizing a monolithic software application. Here, we identify this architecture style as being suitable for an intermediate step when transitioning from a monolithic to a microservice architecture.

Thereafter, the concept of domain-driven design is presented in Section 2.4 as a basis of modeling such self-contained systems. The analysis approach of this thesis combines static and dynamic analysis methods for discovering a modernized architecture within a legacy application. Hence, Section 2.5 continues with the explanation of the main features of the employed static architecture analysis tool Structure101. Next, the dynamic monitoring framework Kieker and the live trace visualization tool ExplorViz are presented in Section 2.6 and Section 2.7. Since modernizing a legacy architecture also requires the analysis of the system's database, the database administration tool DBeaver is introduced in Section 2.8 for this reason. Furthermore, Section 2.9 presents the database logging library log4jdbc for listing executed SQL queries. Lastly, an overview of the to be analyzed monolithic software system *in|FOCUS* in Section 2.10 is given and main concepts of Docker are discussed in Section 2.11.

2.1 The Monolithic Architecture Pattern

In the past, software systems were generally designed as a single-tiered software system which is organized in a centralized way by sharing resources on a single machine, such as memory and data. As it is shown in Figure 2.1a, it is usually divided into a three layers, namely *User Interface*, *Business Logic* and *Data Interface* which connects the monolith to its database. Even though the application itself can be organized in different components, modules, or services, the whole application is built and deployed as one artifact. Hence, different components of the application cannot be executed independently but are tightly coupled. Usually, these architectural choices are often termed as a *Software Monolith* [Richardson 2014].



(a) Monolithic software architecture **(b)** Microservice software architecture

Figure 2.1. Comparison of the monolithic and microservice architecture style.

This architecture style has several advantages [Kharenko 2015]. First, it simplifies the development and deployment of less complex systems. Especially for small teams with no prior experience with alternative architectures such as microservices, the monolith often seems as the more expedient choice. Secondly, monolithic systems tend to be easier to test since there are less different sources of possible errors. For clarification, when it comes to testing the internal communication between modules, package loss due to networking failures does not need to be considered as a possible reason for a failed test. Furthermore, a monolith is easily horizontally scalable by replicating the whole application behind a load balancer which distributes the requests to each instance.

However, monolithic applications also come with a price. The internal structure of software systems that evolved over time often becomes convoluted and tightly coupled. The boundaries of initially clearly defined modules within the monolith tend to be broken [Kharenko 2015]. The entanglement of the components results in a chain of synchronous deep nested calls as well as multiple points of change when updating the application. After modifying the code, the whole application needs to be redeployed. This increases development efforts and decreases the ability of continuous deployment as well as the flexibility of the development process.

The control flow inside a monolithic architecture is difficult to understand and failures might propagate through big parts of the system [Bonér 2017]. Due to its intricate complexity, it is challenging to maintain the application while still keeping up with the

environment and its changing demands and requirements [Sneed and Seidl 2013]. Usually, monolithic applications are scaled horizontally by duplicating the entire system behind a load balancer in order to serve the increasing requests directed to the system. However, the increased load does often only affect certain subsystems and not the entire application. Therefore, resources are wasted by scaling the entire software system rather than targeting only the overloaded subsystems.

Furthermore, employing new technologies to the monolith is often not straightforward since changes to the technology stack usually affect the whole application. This significantly increases the required resources for the development process [Kharenko 2015]. Consequently, developers undertake a long-term commitment when initially choosing the technology stack of the application. This often leads to the use of outdated technologies in the future and therefore to an accumulation of technical dept [Sneed and Seidl 2013]. The sheer size and complexity of the monolith also impedes on the familiarization of the system for developers who are new to the team. When the software application reaches a certain complexity, experience shows that a division of the team into specialized groups of developers increases the overall productivity and quality of the product. This introduced development process structures the communication between the teams and allows developers to specialize in used technologies of the application. In contrast, a monolithic architecture does not naturally reinforce an independent development and deployment of software components. Hence, the communication effort and therefore the development costs increase [Kharenko 2015].

2.2 The Microservice Architecture Pattern

A *Microservice* architecture enforces the goal-oriented interaction of modular and autonomous software services, called *microservices* [Lewis and Fowler 2014]. Each microservice is a highly specialized unit which should only have a single responsibility [Newman 2015a]. Therefore, each service has precisely defined, comprehensible tasks within a given boundary. Bonér [2017] emphasizes the importance of the isolation as well as the autonomy of each microservice. This independence simplifies the adoption of *Continuous Integration* and *Continuous Deployment* (CI/CD) technologies. Further benefits are the targeted scaling of services under heavy workload while using the available resources of the system efficiently. The particular support of independent monitoring, testing, and debugging of specific services are additional advantages of this architecture style. Figure 2.2 depicts the flexible and targeted scaling of microservices. In contrast to monolithic architectures, the need of replicating the whole system for horizontal scaling purposes is alleviated. Instead, overloaded microservices can be individually targeted for vertical scaling.

Usually, a microservice is a stateful component [Bonér 2017]. This implies that an isolation of the behavior is insufficient for defining loosely coupled units. A microservice also needs to own its state exclusively, preferably inside its own database. An example is depicted in Figure 2.1b. Naturally, this is only the case for stateful and not stateless

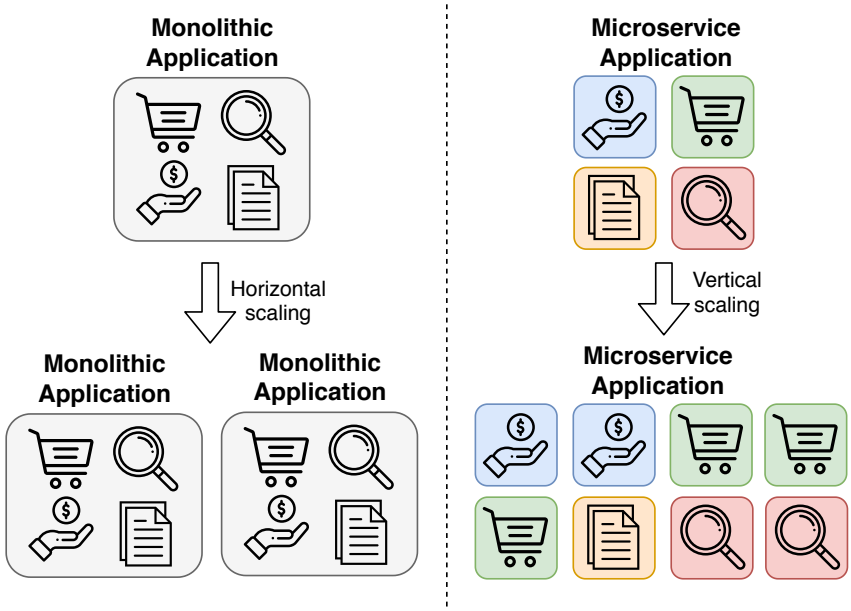


Figure 2.2. Comparison of the horizontal and vertical scaling possibilities of a monolithic and microservice software architectures.

services. Oftentimes, different microservices need similar data. The data on users of the application is a common candidate for this need. Multiple microservices require certain parts of this data set and commit changes to it within the same time period. Therefore, only eventual consistency can often be achieved throughout the software system while ensuring the independence of loosely coupled services [Kharenko 2015; Newman 2015a]. This leads to particular design challenges which are further discussed in Section 3.6.

The desired communication between microservices is led by the principle "smart endpoints, dump pipes" [Lewis and Fowler 2014]. Hence, clearly defined communication interface adapters provide an API, such as an HTTP RESTful API, that can be used by other microservices or third-party systems. Another frequently discussed approach is the use of a lightweight messaging bus for distributed messaging, such as RabbitMQ[1] or ZeroMQ[2]. Generally speaking, asynchronous network calls and transaction-less database calls should always be preferred in order to ensure loose coupling of the services. Furthermore, a microservice implements its own independent web user interface for communicating independently with its end user. Figure 2.1b illustrates this characteristic.

The strictly defined boundaries within the software system isolate a microservice from others. As a result, a failure within a service does not propagate outside the limits of this

[1] `https://www.rabbitmq.com/`, accessed 28.05.2019
[2] `http://zeromq.org/`, accessed 28.05.2019

service. Other microservices are unaffected by this fault and remain operational. Moreover, the high degree of autonomy and decoupling of each microservice enables the developer to select its technology stack independently [Newman 2015a]. As an example, each microservice can be implemented with a suitable programming language in order to increase the operating as well as the development efficiency. Moreover, database technologies can be individually selected to meet the requirements of the microservice and its specific data model.

Another important topic for discussion is the size of a respective microservice. The name *"microservice"* itself seems to emphasize the importance of the small size of a service. However, whether microservices should be intrinsically small is a highly controversial topic. Some argue that the actual size of the code base of each service relies on its organizational structure of the development process. However, the general consensus is that a single developer team should own and manage the code base for a single microservice [Lewis and Fowler 2014; Newman 2015a; Richards 2016]. This concept originates from *Conway's Law* which states that the products structure will at some point represent the structure of its organization [Conway 1968].

While microservices enable an agile and robust development process, it also leads to new challenges for the developers. This architecture style introduces the inherent complexity of a distributed system. Inter-process communication mechanisms have to be implemented to ensure a loss-free delivery of messages. Fallback strategies have to be developed which handle the case of a service or network failure [Kharenko 2015]. Furthermore, the previously mentioned eventual data consistency of the microservices can lead to severe problems if not handled correctly [Kharenko 2015; Newman 2015a].

When it comes to scaling, a certain level of expertise of the developer is required in order to efficiently take advantage of the flexible scaling possibilities of a microservice architecture. A microservice application usually consists of a set of services with possibly multiple runtime instances. Each instance has to be configured, scaled, and monitored. Automated service discovery mechanisms have to be implemented when the number of microservices exceed a certain threshold. Experience shows that manual approaches are often unable to handle this degree of complexity [Kharenko 2015].

2.3 The Self-Contained Systems Architecture Pattern

The *Self-Contained Systems (SCS)*[3] architecture is often used as an intermediate step when transitioning from monoliths to a microservice architecture [Hasselbring and Steinacker 2017]. The reason for that is the reduction of complexity while still benefiting from the main concepts of the microservice architecture style. This includes the enforcement of service isolation via independent units, the discharge of a centralized infrastructure, the support for a flexible adoption of the technology stack, as well as the alignment of the structure

[3] `http://scs-architecture.org/`, accessed 28.05.2019

of both the application and the organization. However, there are also a number of key differences in comparison to microservices. The main agreed differences are the following: Other than microservices, the communication between each SCS should ideally be reduced to a minimum. Each SCS represents a subdomain of the software application which should execute its use cases independently. Furthermore, an SCS can be split further into separate units, especially its business logic. This iterative divide-and-conquer approach is one of the main advantages of SCSs when using them as an intermediate step in the modernization process of a monolithic software architecture towards microservices. The initial coarse granularity of the first division of the monolith into SCSs, followed by a more fine-grained division into microservices is one possible approach in order to overcome the complexity of the modernization process. Last, the size as well as the number of SCSs inside a software architecture usually differ from the microservice approach. A SCS architecture commonly consists of not more than approximately 25 SCSs while a microservice architecture may possibly have hundreds of services [Wolff 2017].

2.4 Domain-Driven Design

Domain-driven design is a software development approach which emphasizes the importance of the exact definition and modeling of the software system's domain. It serves as the basis for the concept of microservices. The domain represents the field of application of the software. A domain model is created which represents key domain objects and their relationships. It is necessary to not only model the software itself but also its domain and the relations and interactions between the two to become aware of all possible issues that could later on pose problems for the development process. For achieving the needed quality of communication between software architects and domain experts, a *ubiquitous language* is developed which is structured around the domain model. The modeled architecture is characterized by a business logic layer which contains the domain classes of the application and separates those from other functions of the system to avoid coupling. Each domain class is autonomous, highly specialized and contains the business logic and data necessary to fulfill its single purpose [Evans 2003].

In order to find the boundaries of these domain classes, the domain is divided into *bounded contexts*. Each bounded context represents a self-contained subdomain of the system. Hence, the required encapsulation of the domain classes resembles the isolation of independent microservices. Therefore, strategies for developing domain-driven designed systems can also be applied in order to discover a microservice architecture [Evans 2003]. An example is given by Figure 2.3. It shows the division of the domain of a trading company into subdomains, presented as blue boxes and further into domain classes, presented as gray hexagons. These are grouped into bounded contexts, depicted as red ovals. Bounded contexts can contain any number of domain classes. Furthermore, they can even incorporate one or more subdomains within their boundaries.

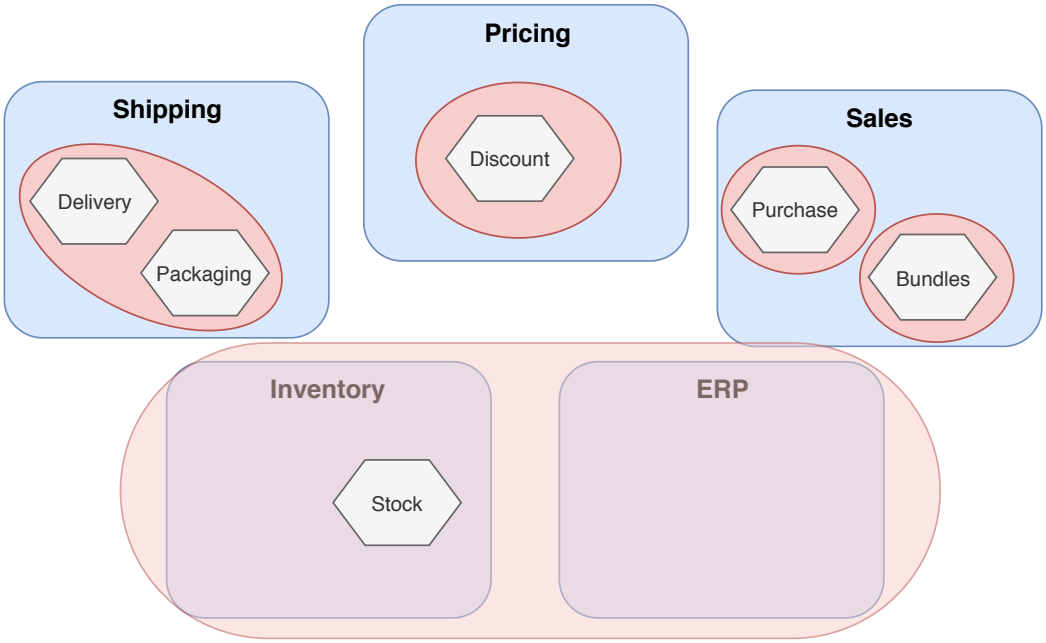


Figure 2.3. Division of the domain into domain classes and definition of bounded contexts.[4]

2.5 The Static Software Structure Analysis Tool Structure101

Structure101[5] is a static code analysis tool for the visualization of the source code packages and contained dependencies of a software project. It provides a workbench with an array of different tools which aim at supporting the software architect in understanding, analyzing, and refactoring large and complex software systems. This application automatically constructs a traversable hierarchical dependency graph of an imported software project. This model can then be manipulated by using pattern transformations, as well as reorganizing, filtering, and automatically grouping related modules. This aims at supporting the software architect to decompose monoliths and disentangle as well as decouple software components.

Figure 2.4 shows an example of the user interface of Structure101. The dependency graph presents the user with an overview of the packages of the analyzed software application, their dependencies, and the direction of said dependencies. The numbers next to the relation arrows indicate the number of relations of a package to another. Moreover, a package can be expanded to see its content, as illustrated by the orange box in the figure.

[4] `https://www.informatik-aktuell.de/entwicklung/methoden/ddd-context-is-king-kein-context-keine-microservices.html`, accessed 28.05.2019

[5] `https://structure101.com/`, accessed 22.06.2019

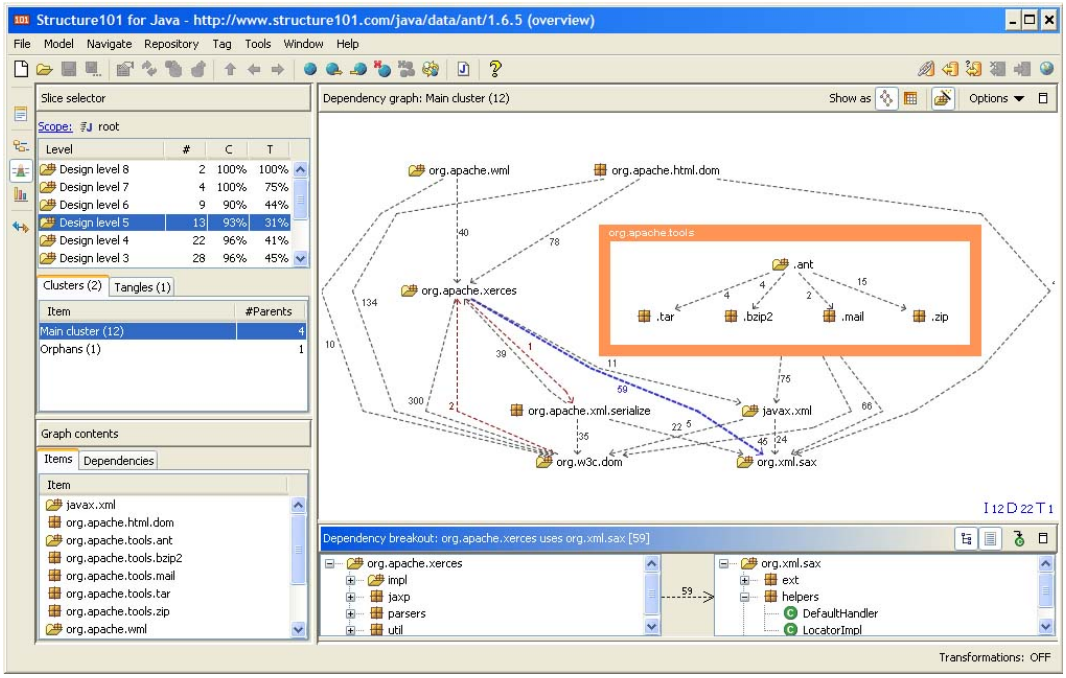


Figure 2.4. Static code analysis tool Structure101 shows code dependencies between classes and packages.[7]

Structure101 provides additional views for more detailed information about packages and classes, as well as their relations. A complete list can be found on the Structure101 website[6].

2.6 The Software Monitoring Framework Kieker

Kieker[8] is a dynamic analysis tool developed by the Kiel University and the University of Stuttgart. Its first main application is the performance monitoring and the dynamic analysis of the runtime behavior of large-scale software systems. This is realized by measuring operation response times, CPU utilization and memory usage. Additionally, the analysis of user sessions and traces provide an detailed insight into the behavior of the monitored application. The second main application of Kieker is the architecture discovery of a software system. Therefore, structural and behavioral architecture data is extracted by identifying architectural entities, such as packages and classes, as well as their interactions, such as procedure calls.

[6] `https://structure101.com/legacy/structural-analysis/`, accessed 24.06.2019
[7] `https://www.prweb.com/releases/structure101/java/prweb448397.htm`, accessed 15.04.2019
[8] `http://kieker-monitoring.net/`, accessed 15.04.2019

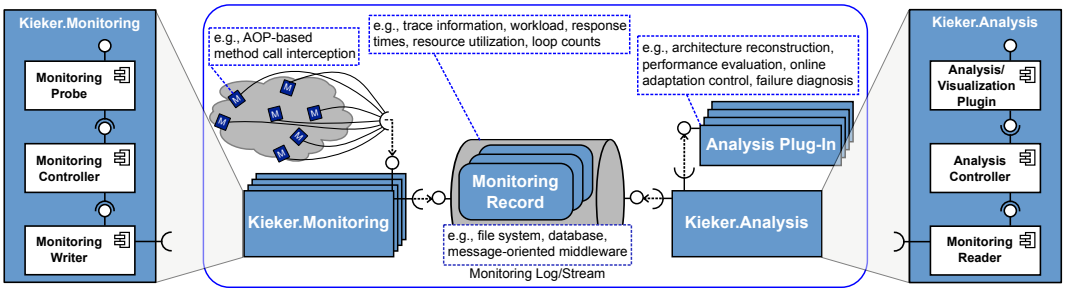


Figure 2.5. Overview of the internal structure of Kieker.[10]

In order to gather all the required information for the two presented use cases, Kieker provides several predefined, customizable probes for the application instrumentation and motorization. These probes can be weaved into the byte code with the help of *AspectJ*[9], an aspect-oriented extension to Java which allows the modularization of cross-cutting concerns such as monitoring and logging. The gathered data is then written and stored into monitoring records for further processing and analysis [van Hoorn et al. 2012]. Figure 2.5 presents an overview of the internal structure of Kieker for supplementary insight.

2.7 The Live Trace Visualization Tool ExplorViz

ExplorViz[11] is a live trace visualization tool which is developed by the Software Engineering Group of the Kiel University. It aims at improving the comprehension of the system and the ongoing communication within. A rendered 2D software landscape, exemplary depicted in the bottom part of Figure 2.6, shows the model of an analyzed software landscape and its software systems (gray boxes). The nodes of a software system (green boxes) contain applications (purple boxes) which communicate with each other (orange lines).

The top part of Figure 2.6 shows the 3D application visualization of the data model landscape of a chosen application. This visualization is in this case based on the city metaphor. It depicts detailed information about the architecture of the application and communication (orange lines) between components, such as packages (green boxes) and classes (blue boxes) [Fittkau et al. 2013; 2015; 2017]. Different extensions, such as a virtual reality mode allow for an even more immersive and interactive discovery of the internal structure of software systems. ExplorViz uses Kieker, which was presented in Section 2.6, to instrument software systems for a dynamical analysis of its runtime behavior. Instead of using Kieker's analysis methods, ExplorViz uses its own and instructs Kieker to send it the gathered monitoring data, called *records*. The Analysis service creates traces from

[9] texthttps://www.eclipse.org/aspectj/, accessed 15.04.2019
[10] http://kieker-monitoring.net/properties/, accessed 15.04.2019
[11] https://www.explorviz.net/, accessed 15.04.2019

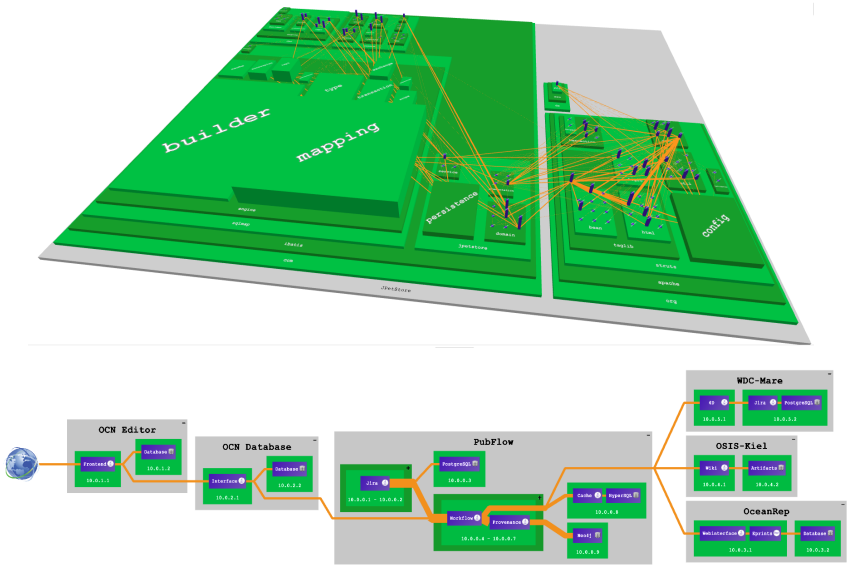


Figure 2.6. 3D application (top) and 2D landscape visualization (bottom) generated by ExplorViz.

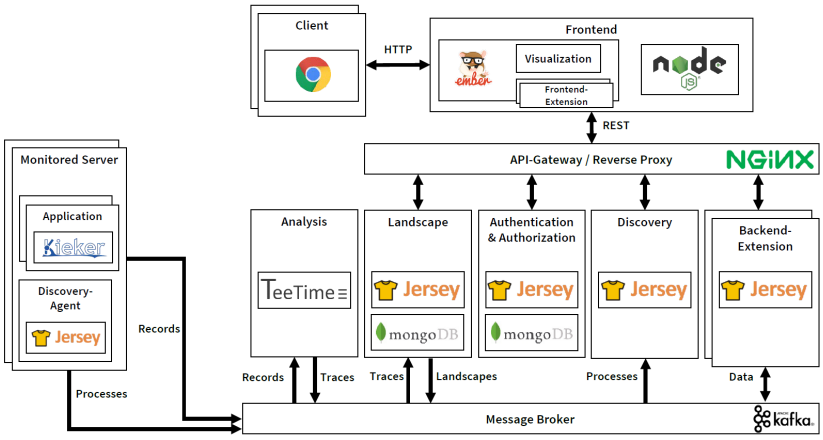


Figure 2.7. Overview of the microservice architecture of ExplorViz [Zirkelbach et al. 2019].

these records which are sent to the Landscape service which creates the models which are visualized in the Frontend service of ExplorViz [Zirkelbach et al. 2019]. An overview of the microservice architecture and the interaction of these services is presented in Figure 2.7.

2.8 The Database Administration Tool DBeaver

Not only the code base but also the database structure has to be considered for reconstruction when modernizing monolithic software systems. Database discovery and analysis tools enable the developer to understand and restructure the current database schema of legacy application for meeting the specific needs of the new architecture.

DBeaver[12] is an open source multi-platform SQL client and database administration tool. Its functionalities range from data browsing and editing to the visualization of the database schema with the help of entity-relationship (ER) diagrams. Furthermore, an SQL editor enables the execution of SQL queries. The application supports relational databases with a JDBC driver which is a software component that enables the interaction between a database and a Java application. For other databases, such as NoSQL, proprietary database drivers are used. Figure 2.8 shows the UI of DBeaver. After establishing a connection to a database, the user can browse its different components, such as its tables and views, on the left hand side of the UI. The main view in the center presents automatically layouted visualization of the database tables and their relations to each other.

2.9 The JDBC Driver log4jdbc

The open source library *log4jdbc* is a Java Database Connectivity (JDBC) driver that can log SQL and JDBC calls for other logging frameworks such as Apache Log4j. The JDBC API is part of the Java SE Platform [14]. Its task is to establish and manage a connection to relational databases as well as to transmit SQL queries to the database. The query responses are then transformed into a for Java usable format. Additional features of log4jdbc include the logging of both the query execution times as well as the result set of an SQL query.

2.10 The Lottery Web Plattform *in|FOCUS*

The software system *in|FOCUS* is key object for the architectural analysis of this thesis. It is a sales and management software solution for lottery operators, developed by adesso[15], a German consulting and software development company. This multi-channel portal platform enables the management of customer data, which is stored in a centralized database.

[12] `https://dbeaver.io/`, accessed 22.06.2019
[13] `https://github.com/dbeaver/dbeaver`, accessed 15.04.2019
[14] `https://www.oracle.com/technetwork/java/javase/tech/index.html`, accessed 15.04.2019
[15] `https://www.adesso.de/de/index.jsp`, accessed 01.06.2019

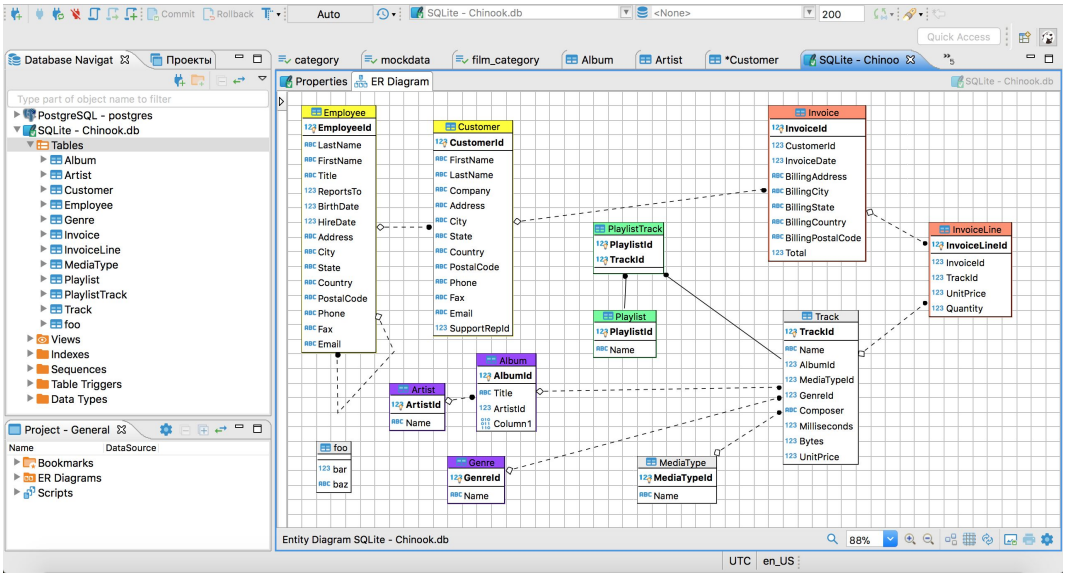


Figure 2.8. Database administration and analysis tool DBeaver shows an ER diagram of a connected database.[13]

Additionally, this application gives detailed insight into the customer behavior and shows invaluable information for sales and marketing campaigns. It provides access to a catalog of online lottery and instant lottery games as well as other games. While consisting of multiple core functionalities, the *in|FOCUS* application is furthermore customized for the specific needs of each customer. Adesso also makes *in|FOCUS* available as a *Software-as-a-Service* for their customers which is currently used by four different state lotteries.

2.11 The Enterprise Container Platform Docker

The enterprise container platform Docker[16] is a set of collaborating Software-as-a-Service and Plattform-as-a-Service products which employ operating-system level virtualization in order to standardize the development and the delivery of software applications. The target software system is therefore bundled into a standardized unit, called a *container*. Each container incorporates their own software, the necessary libraries as well as configuration files. These containers are created at runtime with the help of a *Docker images* which is an executable standalone and lightweight software package. The images are assembled with the help of a *Dockerfile* script which is written by the developer. Thereafter, the developer specifies the content of the containers within a *docker-compose.yml* file. Finally, the docker

[16] `https://www.docker.com/`, accessed 28.05.2019

containers are hosted by the *Docker Engine*. Thereby, the Docker containers isolate its software from the environment and will therefore always run the same, regardless of the infrastructure.

Figure 2.9. Overview of the Docker development workflow.

Chapter 3

Analysis of a Monolithic Application

The third chapter of this thesis presents the software modernization approach of the *in|FOCUS* application as well as its results. In order to ascertain the taken decisions of this process, Section 3.1 discusses applied key concepts of the software architecture modernization. After classifying the conducted project as brownfield, the benefits of exploiting self-contained systems as an intermediate step in the transition from a monolithic architecture towards microservices are discussed. Thereafter, the domain analysis of the software system is performed in Section 3.3. We divide the domain into a set of distinct bounded contexts. They serve as a basis for structuring the static analysis of Section 3.4 with the help of Structure101. The goal of the static analysis is the partitioning of *in|FOCUS* into well-defined self-contained systems. Next, Section 3.5 builds on the findings of the static analysis. The live trace visualization tool ExplorViz is then utilized to dynamically investigate the behavior of the *in|FOCUS* application. Based on these results, the previously defined self-contained systems are further divided into microservices. Section 3.6 explains the final step of the modernization process, namely the adaptation of the database to the new architecture style. This chapter concludes by discussing different alternatives when composing the newly developed services to a single, coherent software system.

3.1 Software Architecture Modernization

After a certain time in development, software systems tend to struggle with an overwhelming complexity and the software architecture oftentimes deteriorates into a highly entangled and coupled monolith. Thus, the ability to react to the requirements and needs of an environment of ever increasing speed is hindered by a non-agile development and deployment process. Moreover, developments of newly requested features could not often be delivered in time [Knoche and Hasselbring 2018]. Additionally, the inability to employ and benefit from emerging software technologies while being stuck with an outdated technology stack is troublesome. The resulting accumulation of technical debt [Sneed and Seidl 2013] is a vicious circle.

Consequently, more and more companies invest a considerable amount of resources for the modernization of their monolithic software systems towards more agile and maintainable software architectures such as microservices. As presented in Section 2.2 the microservice architecture excels in providing a highly scalable and flexible software architecture and

development process. However, the journey from a monolith to a microservice-based application is not trivial. During the last years, a number of different modernization processes were developed in order to guide the developers during this complex endeavor [Geitgey 2013; Richardson 2014; Martincevic 2016; Bonér 2017; Bindick and Stoye 2018; Fritzsch et al. 2018; Kalske et al. 2018; Knoche and Hasselbring 2018].

Generally speaking, it is usually distinguished between *greenfield* and *brownfield* modernization projects [Newman 2015b]. The first describes the complete redevelopment of the software system from scratch without any integration of legacy code. This clean-slate approach provides the opportunity to investigate and to chose the most fitting state-of-the-art technologies and redevelop the new application without any constraints. Nevertheless, this freedom often comes with a lack of clear direction and results in a comparatively high risk of success. Additionally, every aspect of the software system needs to be newly defined which further increases the challenge of finalizing a greenfield project in a reasonably timely manner [Admin 2018].

Alternatively, brownfield projects take a prior existing legacy system as the basis of the redevelopment of the software application. Hence, the newly developed software architecture has to take previously made design decisions into account. Furthermore, the modernized system usually has to coexist with the legacy application to a certain extent. Therefore, the general direction of brownfield projects is usually more structured because they orientate themselves towards the legacy system. By reusing business processes and parts of the already existing code base, valuable resources can be saved. This lets the developers focus their efforts on important pending design decisions. Besides, more resources can be utilized for introducing state-of-the-art technologies which were previously unknown to the involved developers [Newman 2015b]. However, the brownfield approach also requires in-depth knowledge of the existing legacy system in order to efficiently reuse parts of its code base or its architecture. Additionally, the endeavor of dealing with legacy code often gets underestimated and can lead to an initially overlooked time sink [Newman 2015b], [Admin 2018]. On top of that, brownfield projects can encourage corner cutting by the excessive reuse of the legacy code or outdated technologies.

When it comes to operating the legacy system during the modernization process, we usually distinguish between two general approaches. The first possibility is to stop any further maintenance or updates of the legacy system. It is even possible to shut down the software system completely until the reworked system is ready for operation. The new application is then released in a big bang and replaces the legacy system in one step. The second choice is to continue operating and further developing the legacy application alongside the modernizing efforts. However, it is recommended that the further development of the legacy system should be kept at a minimum [Blanch 2017]. Instead, components with the highest need for updates or maintenance should be considered as first candidates for the extraction and the redevelopment into microservices. Especially brownfield projects often enable such a more iterative modernization approach. Exemplarily, the development of every new feature for the monolithic software system is directly

developed as its own microservice. Over and above that, parts of the monolith can be extracted and realized as a microservice one after another [Fowler 2015; Blanch 2017]. In comparison to the big bang approach, this strategy reduces the risk of far-reaching service failures of the software system. Furthermore, invaluable experience can be gained by designing and testing a microservice by incorporating the service into the operative legacy system. Non-ideal design decisions can be reviewed and reworked. This knowledge can be transferred to the more successful development of the next microservice.

Nonetheless, the development of a microservice architecture can be overwhelming when the necessary experience with the design and implementation of distributed software systems is missing or insufficient. That is why, an intermediate step is often suggested [Fowler 2015]. One option is not to move directly from a monolith to microservices but rather towards a self-contained systems architecture instead [Hasselbring and Steinacker 2017]. Self-contained systems (SCSs) are based on the main concepts of microservices. They enforce service isolation, a missing centralized infrastructure and the organizational alignment with the application structure. Compared to microservices, SCSs tend to partition an application into fewer subsystems. Thus, the architecture is often less complex and more accessible to inexperienced developers. Furthermore, SCSs can be divided into smaller units which can be realized by microservices. Therefore, SCSs naturally support the iterative modernization process of a monolithic software system towards a microservice architecture by overcoming the complexity of the modernization in a divide-and-conquer fashion. For further details of SCSs, see Section 2.3.

3.2 Approach for Analyzing *in|FOCUS*

As outlined in Section 1.2, this thesis aims at performing a structural analysis of the architecture and the database of the software system *in|FOCUS* (Section 2.10). This analysis provides an initial division of the system into multiple SCSs. Thereafter, additional investigations partitions chosen SCSs into more fine-grained microservices. For that, we orientate ourselves on the proposed modernization processes of Bindick and Stoye [2018] and Martincevic [2016]. Both base their methods on main ideas of the domain-driven design. The key concept is the incorporation of the environment of the application, namely its *domain*, into the planning and the development process. The domain analysis and domain modeling provide the required in-depth information which is necessary for finding a suitable decomposition of the software system. Especially when it comes to legacy applications with insufficient documentation of the implemented business logic, this step is essential [Bindick and Stoye 2018].

Therefore as a first step, we analyze and model the *in|FOCUS* domain, namely the lottery application domain, in order to familiarize ourselves with the requirements and characteristics of this environment. The *domain model* visualizes and abstracts the involved business objects of the domain and their relationships to each other with the help of a *ubiquitous language*. For further information, see Section 2.4. As a next step, the domain

model is divided into multiple cohesive and independent subdomains, called *bounded contexts*, which aim to represent the boundaries of our SCSs. However, the exact dimensions of each bounded context within the domain is neither always clear nor unambiguous. Therefore, we divide this challenge into multiple sub-problems. As suggested by Knoche and Hasselbring [2018], we define a set of indispensable service operations of an application of the lottery domain. These service operations enable us to specify a certain provided behavior of the lotto software system whose boundaries and dependencies can be more easily analyzed. Next, each defined use case is assigned to fitting objects of the ubiquitous language. These objects represent possible subdomains. The resulted allocation provides us with a first impression of possible bounded contexts and their relationship to another. Section 3.3 discusses the process and the results of the domain analysis of *in|FOCUS*.

After gaining essential insights into the lottery domain and its bounded contexts, we utilize the static software structure analysis tool Structure101 for recognizing the previously defined service operations inside the *in|FOCUS* software application. Each service operation represents a set of to each other dependent Java classes and packages which implement the service operation functionality. Here, Structure101 is the main tool for the dependency analysis. Thereafter, the classes and packages can be mapped onto the subdomains of the lottery domain model. It is based on the previous assignment of the service operations to the subdomains of the domain analysis. The goal is to determine highly cohesive and loosely coupled components in order to draw the boundaries of the bounded contexts. This results in a first division of the *in|FOCUS* software system into bounded contexts which can be realized as SCSs. Section 3.4 examines the static analysis of *in|FOCUS* further.

Thereafter, we expand on the results of the static analysis with the help of the live trace visualization tool ExplorViz. By dynamically analyzing the behavior of the *in|FOCUS* application, we confirm the previously drawn borders and divide the SCSs of the system further into more fine-grained microservices. Section 3.5 presents the conducted dynamic analysis of *in|FOCUS* and its results.

A software modernization process does not only focus on the software architecture but also takes the data model of the software application into account. In order to develop independent services, each of them must own its own state on top of its logic. Consequently, the next step of this analysis process is to investigate a possible division of the data model of *in|FOCUS*. Accordingly, the accessed database tables of each service operation are identified with the help of both the database administration tool DBeaver and the database access logging of log4jdbc. Afterwards, the aggregation of all accessed database tables by the service operations of a bounded context summarizes the necessary data for this subdomain. By repeating this for all bounded contexts, the data model is mapped onto the domain model. Section 3.6 reviews the conducted database analysis of *in|FOCUS* in further detail.

3.3 Domain Analysis of *in|FOCUS*

The first step of the analysis process is the familiarization with the domain of the lottery software system *in|FOCUS*. The necessary knowledge is gathered with the help of provided customer requirements documentation by adesso. As these documents contain sensible business information, extracts cannot be shared in this thesis. On top of that, qualitative interviews and discussions with software developers and domain experts of the *in|FOCUS* system grant the required understanding for the object of study. This research process is guided by the following investigative questions:

Q1 Who are the actors in the lottery application domain?

Q2 How can you describe the relationship of these actors?

Q3 Which service operations does a lottery application need to provide to these actors?

Q4 How can these service operations be mapped onto domain business objects?

Q5 How can these business objects be grouped into bounded contexts?

By answering these questions, we do not only inform ourselves about the domain but also create an essential commonly defined vocabulary. This vocabulary is then seen as part of the ubiquitous language which is required for further DDD analysis.

Actors of the lottery application domain

Table 3.1 answers **Q1** and gives an overview of the involved actors in a lottery software system. Generally, we distinguish between the *Lottery Application Users*, the *State Lottery* and *Others*. *Customers* of the lottery application represent a subset of *Users* who actively engage with the offered products of the software system.

Relationship of the Domain Actors

Figure 3.1 models the relationships of these actors to each other (**Q2**). As expected, the lottery application represents the center of the relationship model as it connects every other actor with one another. The state lottery provides a game catalog to the application which offers the whole variety of games to the customers. Moreover, the lottery software submits the filled out lottery tickets to the state lottery which in response transfers the lottery drawing results back. Other than that, the customer makes use of other possible services of the application, such as special events or newsletters. Apart from the customers, there are other users who use different kinds of services of the software system. Exemplary, a marketing specialist is able to collect customer behavior data for research purposes from the lottery application. The lottery software system makes use of certain other services for collecting specific information on the users. As mentioned in Table 3.1, these services

Table 3.1. Actors in the Domain of a Lottery Software System.

Lottery Application	The lottery application provides lottery games and other lottery related services such as newsletters to their users.
User	Users incorporate every person that uses or administrates the lottery software system in any way.
Customer	Customers are a subset of users who buy products or other services of the application.
State Lottery	The state lottery provides lottery games for the lottery application that the customer can play. The state lotteries then determine the lottery winners on predefined dates and notifies the lottery software system accordingly.
Other	This group summarizes every other actor that is involved in the lottery process and not mentioned above. Exemplary members of this group is the state government which enacts lottery laws. An other central member of this group is the OASIS system which logs every banned player in order to counteract gambling addiction.

include the database for banned players OASIS or the German credit bureau SCHUFA. The only relation that does not include the application is the one between the state lottery and the "Other" group. Precisely, the government does not only enforce restrictions and policies on the lottery software, but also on the state lotteries themselves. However, later on, we are only interested in the relations of the lottery application since this is the target object of development. Further relationships between either others or the state lottery and the users are possible. But, neither are these relations part of the context of the lottery software system domain nor can we be sure about them due to a lack of information. Therefore, these relationships are not included in our model.

Service Operations of the Lottery Application Domain

For answering **Q3**, we identify the possible service operations (SO) according to the relationships of the lottery application to other actors of the domain. We group the service operations of the software system, depending on the involved actors. Table 3.2 depicts a sample of the service operation list for the lottery software system. The entries for each group show some of the related essential service operations. A customer has to be able to log into the customer account and transfer money from his or her bank account to the online wallet of the lottery application. As follows, the buyer can pay for a lottery ticket after having filled in a relevant form. An administrative user of the lottery software system might want to be able to assign administrator rights to other accounts and to gather customer behavior data.

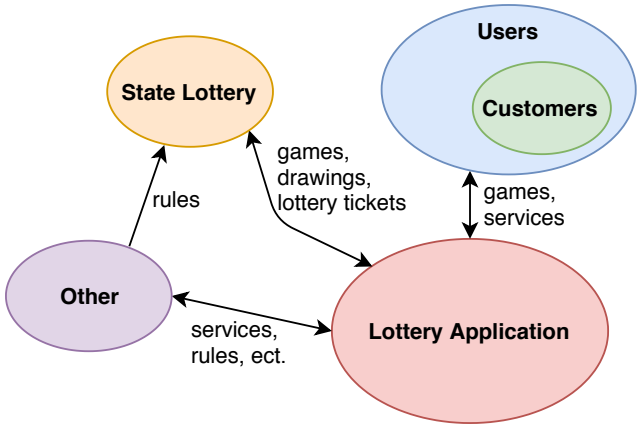


Figure 3.1. Actors of the lottery software system domain and how they are related to each other.

Furthermore, the lottery application submits the filled out lottery tickets to the state lottery and then awaits their notice to import the results of the drawings. Lastly, the lottery application has to check SCHUFA and OASIS in order to determine whether a new customer is allowed to play the lottery games. As soon as a customer approves and wants to transfer money to his lottery online wallet, the selected payment method gets invoked. For a complete list of the service operations, see Section A of the Appendix.

Table 3.2. An excerpt from grouped service operations (SO) of the lottery application.

Customer SO	Login customer Transfer money to online wallet Fill out lottery ticket
User SO	Assign role Gather customer data
State Lottery SO	Submit lottery tickets Receive lottery drawing results
Other SO	Request SCHUFA information Check OASIS entry Invoke payment method

Mapping the Service Operations onto Domain Business Objects

After determining the service operations, they have to be mapped onto domain business objects (BO). A BO is part of the ubiquitous language which represents parts of the functionality of a software system. Table 3.3 shows the SO-BO mappings of the previously examined SOs of Table 3.2 in order to answer **Q4**. It becomes apparent that certain SOs,

such as "Transfer money to online wallet" are mapped onto multiple BOs. These kinds of SOs often describe an application process that either is used in multiple scenarios or which is more far-reaching and therefore affects more than one BO. A complete list of the mappings is shown in Section B of the Appendix.

Figure 3.2 illustrates the domain BOs and their relations to each other. The domain is divided into twenty BOs. Each of them represents certain functionalities of the lottery application. In order to provide a comprehensive impression of the whole lottery application domain, this set of BOs is not limited to the SOs of Table 3.2, but includes every defined SO of Section B of the Appendix. The directed arrows in this figure illustrate the relation between two BOs. The meaning of the relation $A \rightarrow B$ differs slightly, depending on the context of both objects. It can mean that "*A provides B*", that "*A uses B*", or that "*A is connected to B*".

Table 3.3. An excerpt from mapped service operations to business objects.

Customer Account	Login customer
Online Wallet	Transfer money to online wallet
Payment Method	Transfer money to online wallet Invoke payment method
Lottery ticket	Fill out lottery ticket
User Account	Assign role Gather customer data
Ticket Submission	Submit lottery tickets Receive lottery drawing results
Customer Verification	Request SCHUFA information Check OASIS entry

Division of the Domain Model into Bounded Contexts

The final step aims at partitioning the domain model and its BOs into bounded contexts. The dependencies between each bounded context should be kept to a minimum. Consequently, the cohesion between BOs of the same bounded context should be higher than to other BOs. Most likely, multiple different divisions of the domain are possible. Figure 3.3 depicts a possible classification. The domain is described by the following five bounded contexts:

The *Customer* context contains every functionality of the customer account. It provides private information of the customer such as banking information or the balance of the personal online wallet.

The *Gaming* context combines every part of the application for providing, carrying out, and managing different lottery games. Furthermore, the winner determination and notification are also part of this context.

The *Marketing* context provides different marketing instruments such as creating newsletters or gathering and analyzing the customer behavior.

The *Payment* context handles all outgoing and incoming money transactions as well as provides different payment options to the customer.

The *Administrative* context encompasses organizational functionalities such as the user and role management of the software system.

This concludes the domain analysis. We established a solid foundational understanding of the lottery application domain, its involved actors, and required business objects. Furthermore, we developed an initial division of the domain into bounded contexts. The following sections take the above presented results as a basis for further static and dynamic analyses of the lottery software application *in|FOCUS*.

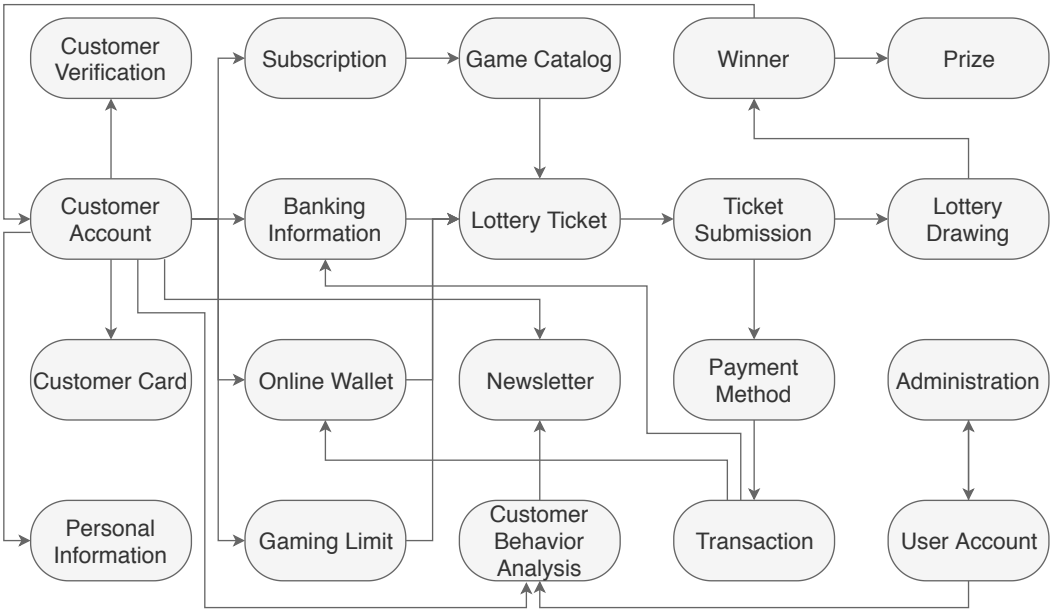


Figure 3.2. Business objects of the lottery software system domain and their relations to each other.

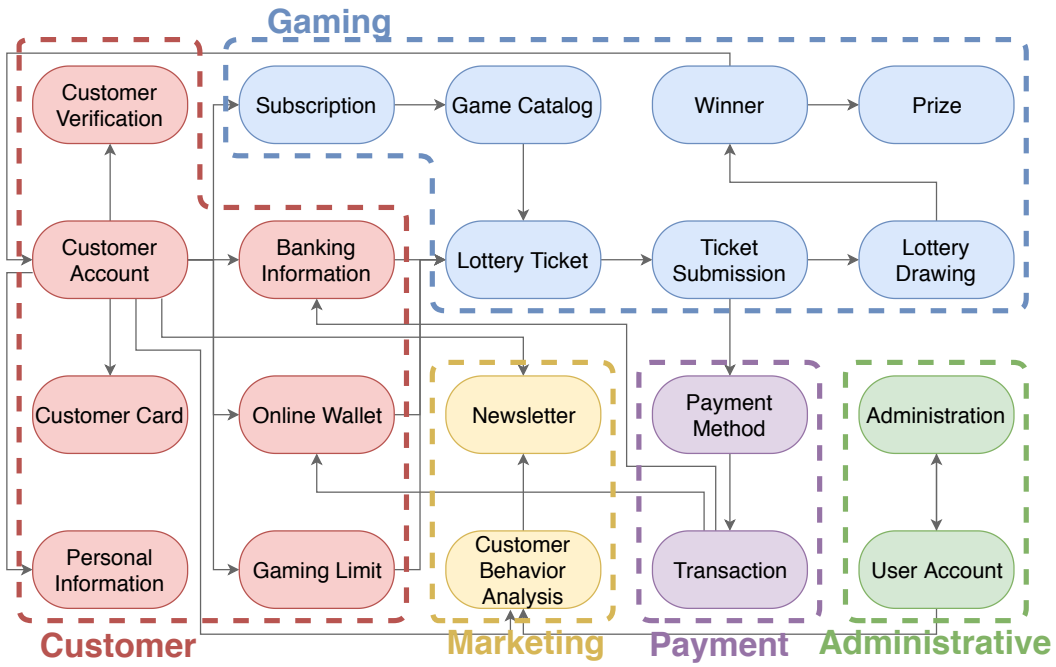


Figure 3.3. Partitioning of the domain model into bounded contexts.

3.4 Static Analysis of *in|FOCUS*

The previous Section 3.3 presented the results of the domain analysis of a lottery application. We base our further investigations on these findings. Therefore, the first goal of this static analysis process is to transfer the newly gained knowledge onto the existing *in|FOCUS* application. However, before analyzing the software architecture itself, the *in|FOCUS* project and its internal package structure is introduced.

3.4.1 Introduction to the *in|FOCUS* Project

Figure 3.4 gives an overview of the package structure of the *in|FOCUS* application. Each package and its main functionality of the `infocus-core` Java project is described in Table 3.4. It is noted that each of the presented packages has several sub-packages. Although, discussing every single package would go beyond the scope of this chapter. The application is deployed on a cross-platform JBoss Enterprise Application Plattform (EAP) 6.3[1], nowadays known as WildFly Application Server. A further introduction of the *in|FOCUS* software system is given in Section 2.10.

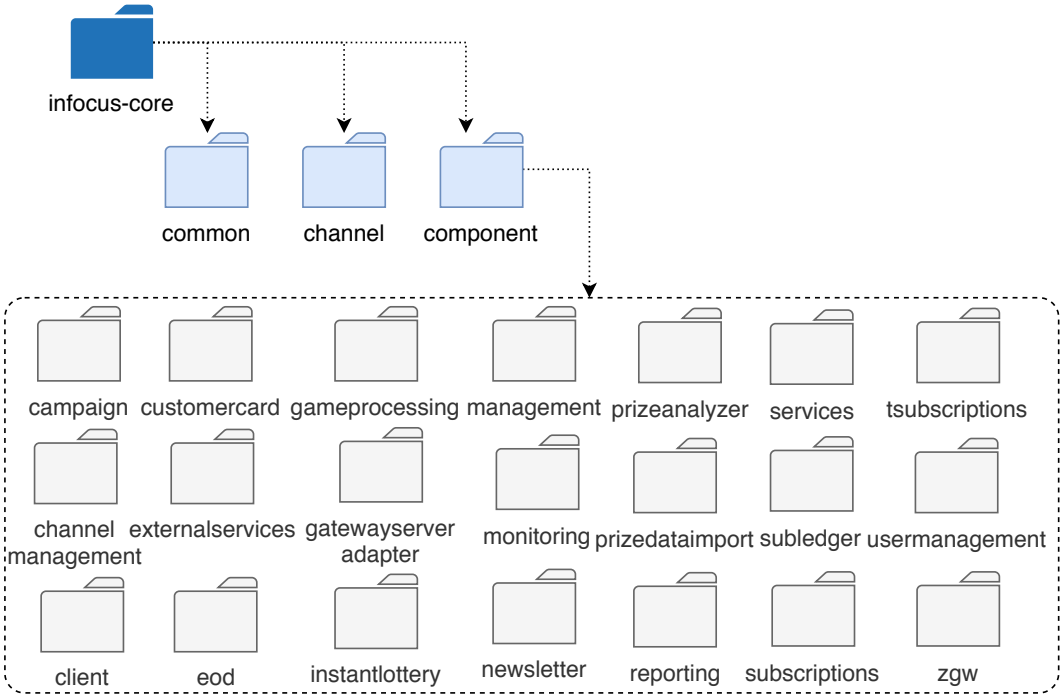


Figure 3.4. Structural overview of the *in|FOCUS* project.

[1] `https://www.redhat.com/de/technologies/jboss-middleware/application-platform`, accessed 05.06.2019

Table 3.4. Package descriptions of the *in|FOCUS* application.

Package Name	Description
`channel`	Collection of different SMS template requests as well as the functionality for SMS messaging to the *in|FOCUS* customer.
`common`	Functionalities which are commonly used throughout the project.
`component.channelmanagement`	Management of supported channels of *in|FOCUS*.
`component.client`	Management of clients and mandates of *in|FOCUS*.
`component.customercard`	Management and administration of customer cards which enable the customer to play which their *in|FOCUS* customer account at local lottery offices.
`component.eod`	Management of end-of-day functionalities.
`component.externalservices`	Connection and management of utilized external services. Processes incoming and outgoing requests for customer verification and customer cards. Additionally, it connects to available payment method services.
`component.gameprocessing`	Handles involved processes for lottery gaming and subscription, ticket submissions, and lottery results.
`component.instantlottery`	Functionality for playing instant lottery (e.g. scratch cards) and handling the winnings.
`component.management`	Managing different configurations and parameters for system monitoring.
`component.monitoring`	Handles the application monitoring.
`component.newsletter`	Creates, manages, and distributes newsletters to customers.
`component.prizeanalyzer`	Determines the prizes for the winners of a lottery.
`component.prizedataimport`	Receives and processes the prizes of different lotteries.
`component.reporting`	Manages the compilation and distribution of system reports.
`component.services`	Contains a set of services for different functionalities of the application (e.g. user verification, change game limits, customer notification).
`component.subledger`	Handles a set of payment methods as well as incoming and outgoing payment processes.
`component.subscriptions`	Functionality for managing of lottery game subscription.
`component.tsubscriptions`	Manages terrestrial subscriptions.
`component.usermanagement`	Manages user and customer accounts, as well as connected functionalities (e.g. login, identification, account data).
`components.zgw`	Management of material prizes or lottery winnings above 5000€.

3.4.2 Analyzing *in|FOCUS* with Structure101

After familiarizing ourselves with the internal structure of *in|FOCUS*, the next goal is to statically analyze the relations and dependencies between the *in|FOCUS* components. As previously mentioned in Section 3.2, the approach for this static analysis consists of two major steps. First, we discover the service operations defined in Section 3.3 withing the *in|FOCUS* project. Therefore, the *in|FOCUS* packages are assigned to the different SOs which *in|FOCUS* implements. Secondly, we investigate the dependencies of these packages with the help of Structure101 (Section 2.5).

In order to analyze the package dependencies, the *in|FOCUS* .jar-files are imported into a new Structure101 project. Structure101 comes with an array of different analysis tools for investigating the dependencies of the *in|FOCUS* packages. One of these tools is the *levelized structure map* which is depicted by Figure 3.5. Here, the blue arrows represent general dependencies. The direction of the arrow indicates whether a package either uses or is used by another package. As an example, Figure 3.5 shows the dependencies of `customercard` as well as the packages which use `customercard`. This structure map illustrates the complexity

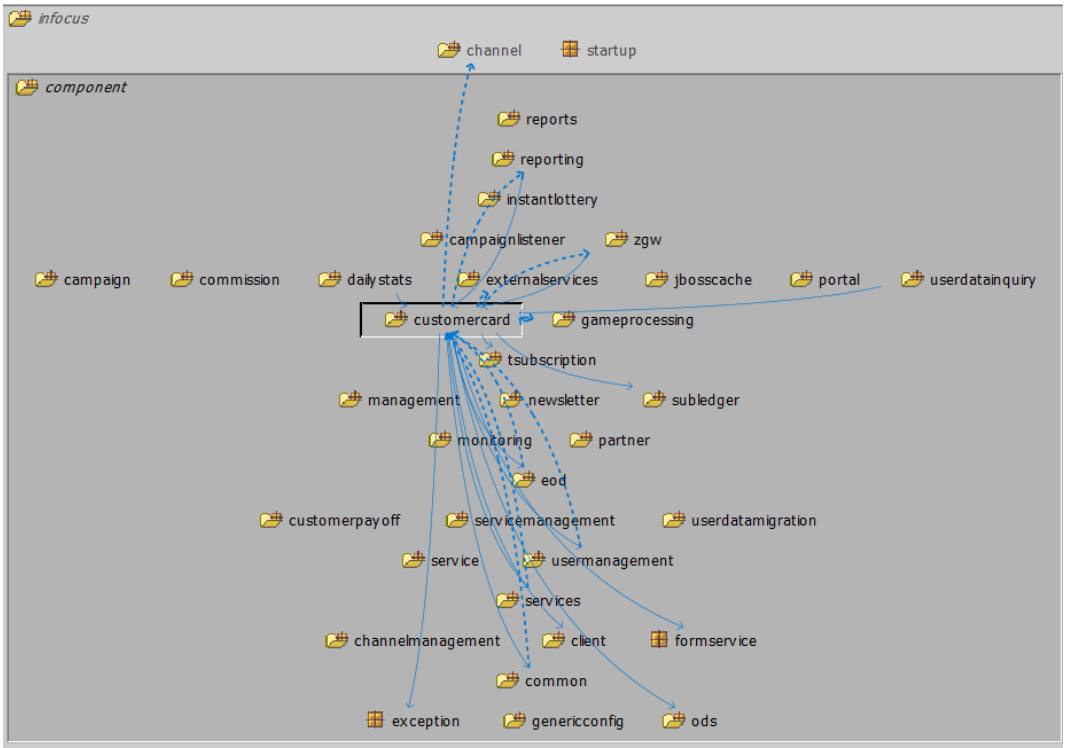


Figure 3.5. Levelized structure map for the `customercard` package.

and the entanglement within the inner structure of the *in|FOCUS* project. The `customercard` package is dependent on approximately half of the other `component` packages of the same level within the file structure. This is the case for most of the other `component` packages as well. Therefore, the main tool for the dependency analysis is the *collaboration list* of Structure101 which is illustrated by Figure 3.6. This feature lets us selectively choose information and simplifies the focused examination on single dependencies. Just like Figure 3.5, Figure 3.6 shows the dependencies of `customercard`, however, as a list. Here, the dependencies of the selected package on the left are highlighted on the right of the user interface. The number above the arrow in the middle represents the number of dependent calls within the code inside the selected package. Hence, there are 687 different points inside the source code of `customercard` which are dependent on parts of the highlighted packages on the right. The given number can indicate a rough estimate on the degree of entanglement of a package. For reference, the number of dependencies for the sub-packages of `component` range from 10 (`channelmanagement`) to 2500 (`gameprocessing`).

Table 3.5 presents an excerpt from the package assignments resulting from the static dependency analysis. It shows a list of the above discussed service operations (Table 3.2) and their involved *in|FOCUS* packages. For a complete list, see Section C of the Appendix. Since we already assigned the service operations to our BOs (Table 3.3) as well as organized these very BOs into bounded contexts (Figure 3.3), we obtain a direct assignment of the *in|FOCUS* packages to the bounded contexts. This leads to a first division of the *in|FOCUS* application after assigning every package to its respective bounded context. Figure 3.7 on page 35 illustrates the assignment and the therefore resulting division for the previously discussed packages. Moreover, Figure 3.8 on page 35 summarizes the resulting bounded contexts with their respective packages as well as the relations in between the bounded context.

Customer contains the main functionality of the customer account management, provided by the package `usermanagement`. This includes the account creation, login and logout,

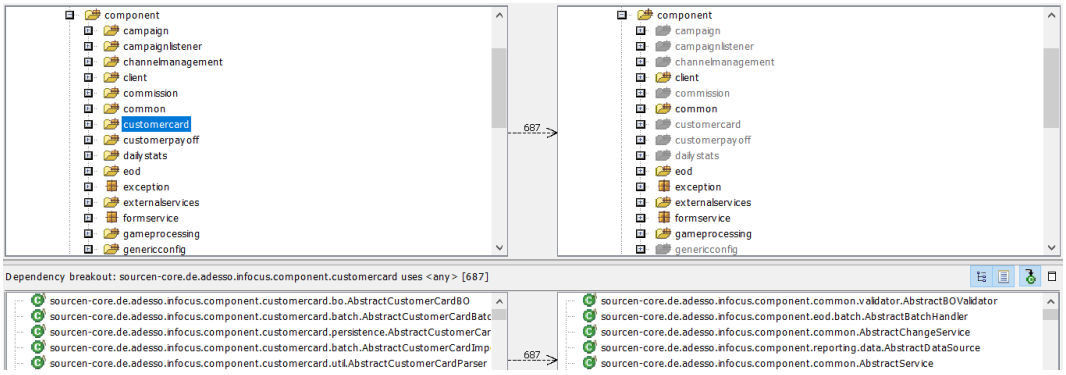


Figure 3.6. Collaboration list of the `customercard` package.

Table 3.5. An excerpt from mapping the *in|FOCUS* packages to the service operations.

Login customer	`component.usermanagement`
	`component.services`
	`component.common`
	`component.subledger`

Transfer money to online wallet	`component.subledger`
Fill out lottery ticket	`component.gameprocessing`
Assign user role	`component.usermanagement`
Gather customer data	`component.reporting` `component.monitoring`
Submit lottery tickets	`component.gameprocessing`
Receive lottery drawing results	`component.gameprocessing`

Request SCHUFA information	`component.services`
	`component.externalservices`
	`component.usermanagement`

Check OASIS entry	`component.externalservices`
	`component.usermanagement`
	`component.services`

Invoke payment method	`component.subledger` `component.externalservices`

as well as the editing of personal information. For these use cases, identity verification services as well as external SCHUFA or OASIS checkups are made accessible by `services` and `externalservices`. Furthermore, the customer card management and the personal online wallet are part of the *Customer* subdomain. A customer card entails the necessary customer account information for offline gaming at local lottery offices. Hence, the games played with the customer card are then credited to the online customer account. However, a customer card can exist without ever being used for gaming and can be seen as an extension of the customer account.
Consequently, we assigned the customer card management to the *Customer* bounded context, rather than to the *Gaming* bounded context. Since ordering and paying for a new customer card is also part of the customer card management, certain functionalities of the `subledger` are also part of the *Customer* subdomain. In order to complete the payment process, the *Customer* bounded context is dependent on the *Payment* context. Additionally, a customer is able to transfer money from his or her bank account to the personal online wallet to enable a straightforward gaming experience. This use case utilizes the same payment process as the ticket acquisition.

Gaming summarizes the gaming functionality of *in|FOCUS*. The customer chooses a lottery game from a provided game catalog, fills out a lottery ticket and creates a game order (`gameprocessing`). A game order can be seen as a receipt which tracks played games

not only for the customer but also for the software system. The ascription to the customer account creates the illustrated dependency between *Gaming* and *Customer*. The application also provides instant lotteries such as online scratch tickets. The instant lotteries are immediately resolved. The conventional lottery games however are resolved at a certain date. Thereafter, the drawing results are imported (`prizedataimport`) and the individual prize of each winner is determined (`prizeanalyzer`).
Alternatively, the customer can also subscribe to certain games. A subscribed game is repeatedly played for a defined period of time. Naturally, the lottery ticket and the subscriptions have to be paid for. Hence, the gaming process is dependent on the *Payment* context. The same dependency can be found in the package `zgw`, which stands for "central winnings management". As the name suggests, it manages the winnings of the customer. Winnings are either transferred back to the personal online wallet or to the bank account of the customer.

Payment is designed to handle and to take account of all incoming and outgoing money transactions (`subledger`) and provides different payment methods to the customer (`externalservices`). In order to receive the up-to-date banking information or the online wallet ID of the customer, *Payment* is dependent on *Customer*.

Marketing provides newsletters which the customer can subscribe to. As long as the customer is subscribed to a newsletter, a notice, either through the website or via e-mail (`services`), is received. Lottery operators can create or edit newsletters. In order to promote these newsletters to potentially interested customers, they can define a target group to which the newsletter is presented more vividly. This promotion process creates the shown dependency between *Marketing* and *Customer*.

Adminstrative comprises the reporting and monitoring services of the application. The customer activity as well as the performance and the load of the software system can be tracked. System reports are created continuously and distributed to the appropriate authority (`services`). Furthermore, this context manages the administrative user accounts and their rights (`usermanagement`). The collection and monitoring of the system's activity create the depicted dependencies to all other contexts of the *in|FOCUS* system.

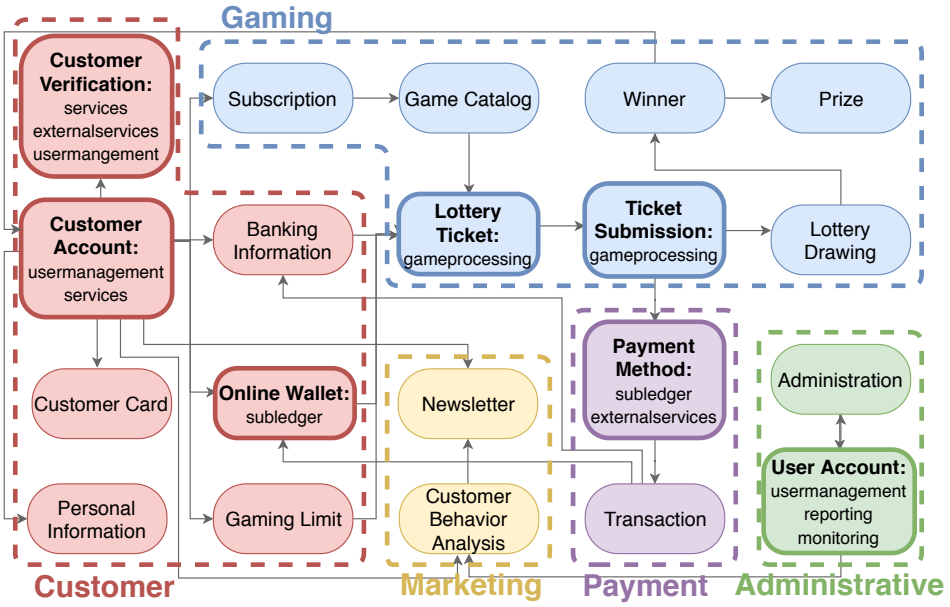


Figure 3.7. Assignment of a selection of *in*|*FOCUS* packages to the appropriate BOs (bold) of the domain.

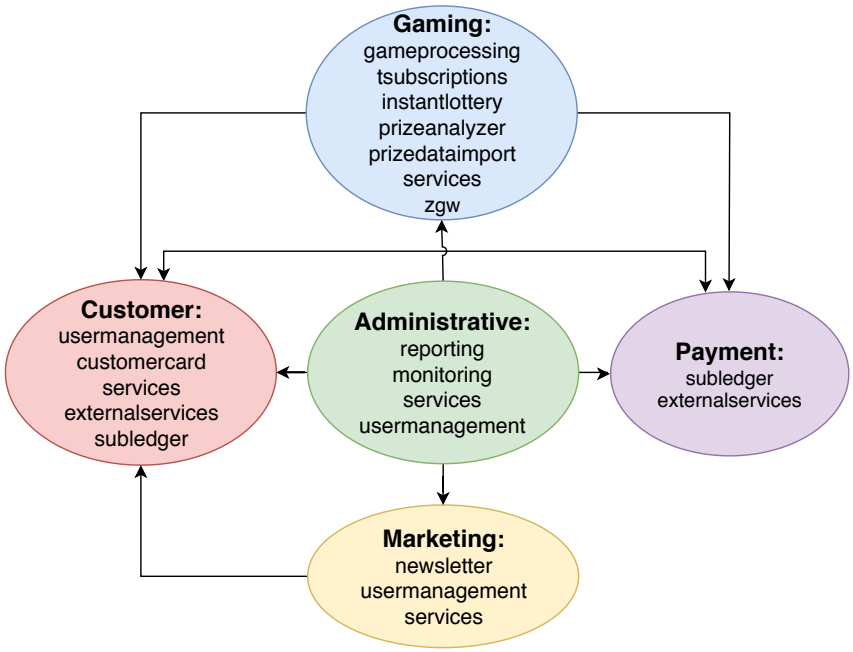


Figure 3.8. End results of the static analysis define the bounded contexts of the *in*|*FOCUS* application and their dependencies to each other. Each bounded context contains a set of *in*|*FOCUS* packages.

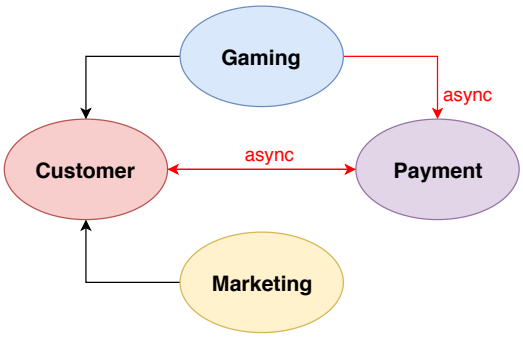
Now that the lottery application *in|FOCUS* is divided into distinct bounded contexts, the next step in the architecture modernization process would be to define the borders of the SCSs along the boundaries of said bounded contexts. However, the current partitioning does not fully support the main concepts of SCSs. In order to focus on the more prevailing challenges to the modernization process, the *Administrative* context is left out of the further discussion of the software architecture division.

Generally speaking, SCSs are supposed to be defined as highly independent, cohesive units which are able to perform their key functionalities on their own with only minimal dependencies to other subsystems. Here, both the *Customer* and the *Gaming* service are dependent on the *Payment* component and are unable to perform certain core use cases on their own. Hence, a customer cannot successfully acquire a customer card or play a lottery game when the *Payment* system is unavailable at that moment. One can argue that buying a customer card is not one of the primary functionalities of its subsystem. Therefore, it would be acceptable to asynchronously forward the buying request to the *Payment* service. Nonetheless, this position is harder to defend when it comes to playing a lottery game. In this case, it would be preferable to successfully play a game without any major dependencies to other services. Therefore, we present four possible alternatives to address this problem and discuss their pros and cons. Figure 3.9 depicts each alternative.

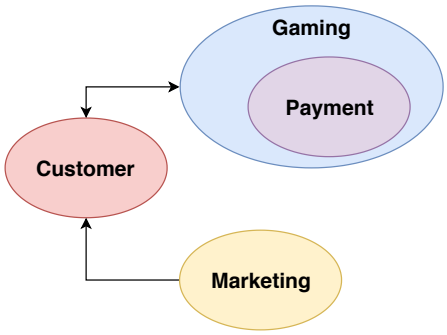
1. The first alternative is to redefine when the use case of filling out and submitting a lottery ticket is considered successful for the *Gaming* context. Up to then, it was considered to be completed after the game order of the lottery ticket was created and paid for. As an alternative, we can consider the use case for the *Gaming* service to be completed right after the game order has been created. Completing the payment process is then part of the *Payment* context. The game order fails when the payment process has not been completed until the ticket submission deadline of the lottery game. In this case, the *Payment* service informs the *Gaming* component. Therefore, an asynchronous communication protocol between *Payment* and the other services would in this case suffice. Figure 3.9a depicts this configuration. This way, the dependencies between the services do not change. However, each SCS can now independently complete their own use cases.

2. Secondly, we can merge the *Payment* and *Gaming* services as illustrated in Figure 3.9b. The resulting SCS can independently complete the whole gaming process, from game order creation, ticket submission to its payment. However, this SCS would be huge in size and would represent a small monolith itself. Also, the dependency between *Customer* and *Payment* when buying a customer card still remains.

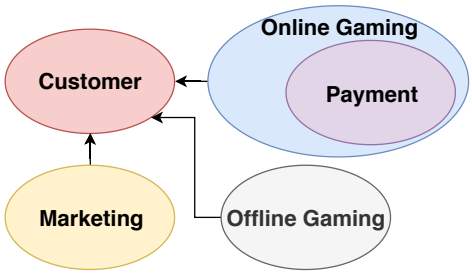
3. The resulting monolithic-like service of the second alternative (Figure 3.9b) can be counteracted by dividing the *Gaming* service itself into smaller parts. One possibility is the division into offline and online games. This breakdown is shown in Figure 3.9c. *Offline Gaming* manages every game which the customer played offline, e.g. with his or her customer card at a lottery office or in form of a TSubscriptions. A TSubscription is



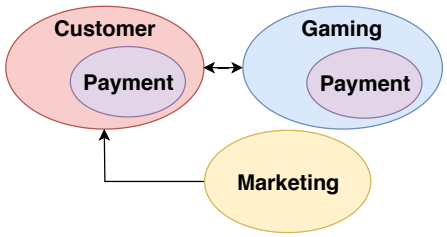
(a) Asynchronous communication to *Payment* service.



(b) Merging *Gaming* and *Payment* service.



(c) Dividing *Gaming* into *Online Gaming* and *Offline Gaming*.



(d) Multiple *Payment* components.

Figure 3.9. Four architectural alternatives for addressing the dependency problem with the *Payment* service.

an offline game subscription where the customer assigns a lottery offline to play a filled out lottery ticket repeatedly until the subscription is canceled. Since the *Offline Gaming* service does not need a connection to the *Payment* service, it reduces the size of *Gaming* without creating new dependencies which affect the successful completion of its own major use cases.

4. The last discussed alternative is the assignment of an individual payment service for both *Gaming* and *Customer*, see Figure 3.9d. The resulting modified SCSs can independently complete their core functionalities. Next to the size problem, the subledgers of both *Payment* components need to be synchronized at some point, e.g. when creating a bank statement. The highly frequented requests for the *Payment* components increase the challenge for providing a reasonably eventual data consistency.

This concludes the static analysis of the *in|FOCUS* architecture. To summarize, we divided the application into distinct bounded contexts which can be realized as SCSs. Thereafter, we saw potential for further improvement of the initial segmentation in order to reduce the dependencies of both the SCSs *Customer* and *Gaming*. Four alternative adjustments to

the architecture were presented. Section 3.5 discusses the next step of the modernization process. The dynamic behavior analysis of *in|FOCUS* with the help of ExplorViz is based on the previous findings of the static analysis and aims at verifying and refining previous architectural design decision.

3.5 Dynamic Analysis of *in|FOCUS*

The dynamic architectural analysis of the *in|FOCUS* software architecture builds on the previous findings of the static analysis. Furthermore, the live trace visualization tool ExplorViz is utilized to verify and refine the previously made architectural design decision. In order to enable the gathering of monitoring data with the help of Kieker probes and the following analysis as well as the visualization of ExplorViz, several configurations have to be made. The necessary setup is shown in the following section Section 3.5.1. For a more detailed introduction on how Kieker and ExplorViz work, see Section 2.6 and Section 2.7.

3.5.1 JBoss and Kieker Configurations

The software application *in|FOCUS* is deployed by a *Red Hat JBoss EAP 6.3* application server. We define the base directory of this JBoss as `jboss-eap-6.3` for the remainder of this section. The necessary setup for enabling the monitoring of *in|FOCUS* with ExplorViz is comprised of the following six steps:

Defining a JBoss module for Kieker

In order to integrate Kieker into the JBoss startup, Kieker first has to be defined as a JBoss module. Therefore, we create a `module.xml` in the directory

```
jboss-eap-6.3/modules/system/layers/base/kieker/main/
```

together with the `kieker-1.13.jar`. Listing 3.1 shows the content of the `module.xml`. The module is named `kieker` and defines the resource path of the Kieker .jar file.

Listing 3.1. Configuration of a JBoss module for Kieker of `module.xml`

```
1 <module xmlns="urn:jboss:module:1.3" name="kieker">
2     <resources>
3         <resource-root path="kieker-1.13.jar"/>
4     </resources>
5     <dependencies>
6     </dependencies>
7 </module>
```

Defining a JBoss module for AspectJ

A second JBoss module has to be defined for AspectJ in order to successfully weave the Kieker probes into the byte code of *in|FOCUS*. The module in the directory

```
jboss-eap-6.3/modules/system/layers/base/org/aspectj/main/
```

and includes a `module.xml` and the `aspectjweaver-1.8.9.jar`. Listing 3.2 shows the content of the `module.xml`. The module name is defined as `org.aspectj` and points to the path of the AspectJ .jar file.

Listing 3.2. Configuration of a JBoss module for AspectJ of `module.xml`

```
1 <module xmlns="urn:jboss:module:1.1" name="org.aspectj">
2     <resources>
3         <resource-root path="aspectjweaver-1.8.9.jar" />
4     </resources>
5     <dependencies>
6     </dependencies>
7 </module>
```

Defining a JBoss module for LogManager

We utilize the `JBoss LogManager` framework to enable logging processes for deployed applications. Therefore, we define a JBoss module located in the directory

```
jboss-eap-6.3/modules/system/layers/base/org/jboss/logmanager/main/
```

which contains the `module.xml` and the `jboss-logmanager-2.0.3.Final-redhat-1.jar`. Listing 3.3 shows the content of the `module.xml`. Here, the module `org.jboss.logmanager` points to the LogManager .jar file and defines its necessary dependencies.

Listing 3.3. Configuration of a JBoss module for LogManager of `module.xml`

```
1  <module xmlns="urn:jboss:module:1.1" name="org.jboss.logmanager">
2      <resources>
3          <resource-root path="jboss-logmanager-2.0.3.Final-redhat-1.jar"/>
4      </resources>
5      <dependencies>
6          <module name="javax.api"/>
7          <module name="org.jboss.modules"/>
8          <module name="org.jboss.as.logging" services="import"/>
9      </dependencies>
10 </module>
```

Declaring the Kieker and AspectJ modules as global

As a next step, it is necessary to declare both the above defined Kieker and the AspectJ modules as global JBoss modules in order to add these modules as dependencies for every Java application deployment[2]. To this end, the `jboss:domain:ee:1.2` subsystem entry of the configuration file

`jboss-eap-6.3/standalone/configuration/standalone.xml`

has to be edited. Here, both modules are defined as global JBoss modules, as depicted in Listing 3.4.

Listing 3.4. Specification of Kieker and AspectJ as global modules of `standalone.xml`

```
1 <subsystem xmlns="urn:jboss:domain:ee:1.2">
2     <global-modules>
3         <module name="kieker"/>
4         <module name="org.aspectj"/>
5     </global-modules>
6     <spec-descriptor-property-replacement>false</
          spec-descriptor-property-replacement>
7     <jboss-descriptor-property-replacement>true</
          jboss-descriptor-property-replacement>
8     <annotation-property-replacement>false</annotation-property-replacement>
9 </subsystem>
```

Setting up the required Kieker files

Furthermore, the required Kieker `aop.xml` and the `kieker.monitoring.properties`, are comprised in

`jboss-eap-6.3/kieker/ .`

The `aop.xml` file specifies the type of Kieker probe that should be used for the gathering of monitoring data. Additionally, it defines the application packages that the probes should be weaved into. Every to be monitored package is established with an `include` statements. Alternatively, the `exclude` instruction can be used to ignore a subset of to be probed packages.

The `kieker.monitoring.properties` file configures Kieker and the used file writers which transfer the gathered data to ExplorViz. For the monitoring of *in|FOCUS*, the `FullInstrumentationNoGetterAndSetter` probe of the type `flow.operationExecution` was utilized. Additionally, it is noted that not all but only the majority of *in|FOCUS* packages are able to be monitored with Kieker probes due to technical issues with internally used, customized loggers of the *in|FOCUS* application. Since these loggers also use AspectJ for

[2] `https://docs.jboss.org/author/display/WFLY8/Subsystem+configuration`, accessed 10.06.2019

weaving in own probes for data collection, it is assumed that an overwriting of AspectJ annotations is the cause of the problem. Moreover, these loggers cannot be disabled since they are essential for certain core functionalities of the *in|FOCUS* application.

Configuring the startup script of JBoss

The final step of the setup is setting the correct Java variables for the JBoss startup in order to allow a successful monitoring of the deployed application with ExplorViz. Hence, the JBoss startup script

`jboss-eap-6.3/bin/standalone.conf.bat`

has to be customized. The required additions, shown in Listing 3.5, are based on the official Kieker documentation[3].

Listing 3.5. Necessary additions of the JBoss startup script `standalone.conf.bat`.

```
1  rem ## Make these system packages visible for JBoss. The packages byteman,
       logmanager and manageenginie are necessary to enable monitoring for java
       applications.
2  set "JAVA_OPTS=%JAVA_OPTS% -Djboss.modules.system.pkgs=
3         org.jboss.byteman,org.jboss.logmanager,com.manageengine,org.aspectj,kieker"
4
5  rem ## Define the AspectJ weaver as a Java agent.
6  set "JAVA_OPTS=%JAVA_OPTS% -javaagent:/path/to/jboss-eap-6.3/modules/system/layers
       /base/org/aspectj/main/aspectjweaver-1.8.9.jar"
7
8  rem ## Set LogManager as the to be used logging manager.
9  set "JAVA_OPTS=%JAVA_OPTS% -Djava.util.logging.manager=org.jboss.logmanager.
       LogManager"
10
11 rem ## Set the classpath for the used modules logmanager, kieker and aspectj
       weaver.
12 set "JAVA_OPTS=%JAVA_OPTS% -Xbootclasspath/p:
13     /path/to/jboss-eap-6.3/modules/system/layers/base/org/jboss/logmanager/main/
           jboss-logmanager-2.0.3.Final-redhat-1.jar;
14     /path/to/jboss-eap-6.3/modules/system/layers/base/kieker/main/kieker-1.13.jar;
15     /path/to/jboss-eap-6.3/modules/system/layers/base/org/aspectj/main/
           aspectjweaver-1.8.9.jar"
16
17 rem ## Configure the Kieker monitoring by setting the required paths.
18 set "JAVA_OPTS=%JAVA_OPTS% -Dkieker.monitoring.configuration=
```

[3] `https://kieker-monitoring.atlassian.net/wiki/spaces/DOC/pages/24215574/Using+Kieker+in+Different+Java+EE+Environments`, accessed 10.06.2019

```
19     /path/to/jboss-eap-6.3/kieker/kieker.monitoring.properties"
20 set "JAVA_OPTS=%JAVA_OPTS% -Dkieker.monitoring.skipDefaultAOPConfiguration=true"
21 set "JAVA_OPTS=%JAVA_OPTS% -Dorg.aspectj.weaver.loadtime.configuration=
22     file:/path/to/jboss-eap-6.3/kieker/aop.xml"
```

3.5.2 Analyzing the Behavior of *in|FOCUS* with ExplorViz

For the dynamic analysis of the *in|FOCUS* application, we focus on certain parts of the architecture based on the static analysis results. There are two reasons for this decision. First, we want to verify and improve the previously made architectural design decisions. Secondly, certain technical limitations of the current state of ExplorViz at the time of this thesis requires us to limit the number of probed packages. Therefore, we are not able to monitor every package of *in|FOCUS* at once but only a subset. Therefore, only the most prominent and impactful packages for the examined service operations of the static analysis are identified and included for monitoring.

The dynamic analysis of *in|FOCUS* has the following two goals: First, we want to discover possible microservices within the software architecture to divide the resulting SCS architecture of the static analysis further. As an example, we present the discovery of two possible microservices. Secondly, we continue the idea of Figure 3.9a and further investigate its presented solution for the "Payment-Problem". To recapitulate, we address the problematic dependencies of both the *Customer* and *Gaming* services to the *Payment* service. The three service operations that were initially dependent on the functionality of *Payment* are "buying a customer card", "buying lottery tickets" and "transferring money to the online wallet". However, after statically analyzing the problem, we came up with a number of different solutions, shown by Figure 3.9. One possibility is to redefine the limits of the respective service operations and the responsibilities of each service. This enables us to let the involved services communicate asynchronously with each other while still meeting the independence requirements of SCSs. Therefore, we investigate this approach further in this chapter.

Discovering new microservices with the help of ExplorViz

One of the analyzed service operations is the selection and the completion of a lottery ticket. The monitored behavior of *in|FOCUS* is depicted in Figure 3.10. This use case can be divided into following four distinct steps:

① Inside the `gatewayserveradapter` package, a lottery manager is invoked for the appropriate lottery game which was selected by the customer. The necessary lottery information such as drawing dates is collected and the lottery ticket is created. Furthermore, the delivery of the required lottery ticket attributes is done by the internal functionality of the `persistence` sub-package in order to assemble the actual ticket. These attributes can represent different available jackpots or gaming options for the selected game.

② The involved classes of `lotterygameprocessor` manage the entirety of the gaming process of the selected lottery.

③ Here, we obtain the customer data such as the ID and the personal gaming limits which are required for successfully playing a lottery game as a customer.

④ The `lottery` package manages the filled out lottery ticket. It tracks the selected fields and specific gaming strategies of the played lottery ticket.

This use case can be split into two main phases. First, the selected lottery ticket is assembled by collecting the necessary gaming information. Thereafter, the customer fills out the ticket and selects different options for his or her game. Here, an opportunity can be found to define a new microservice. This microservice, which we call *TicketManager* from now on, handles the management and the composition of lottery game attributes for the creation of a specific lottery ticket. Hence, it holds all the available options for each lottery game. Depending on the customer's choice, it assembles the necessary information for creating a lottery ticket that the customer can then fill out. This microservice encapsulates the functionality of ① seen in Figure 3.10.

The introduction of this microservice enables the developers to easily define and add new games to the application. Furthermore, this part of the gaming process can be seen as a possible bottleneck, depending on the simultaneously requested games. Therefore,

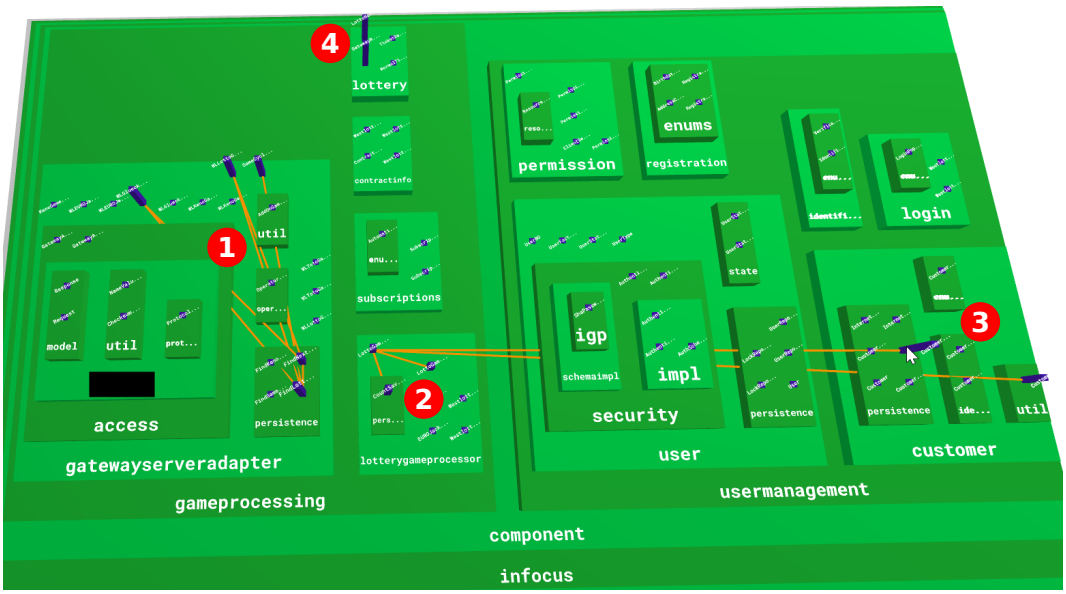


Figure 3.10. Monitoring the *in|FOCUS* behavior with ExplorViz of loading and filling out a lottery ticket.

the possibility of vertical scaling is desirable. Naturally, the *TicketManager* needs to submit the assembled attributes to the *Gaming* service which handles the further gaming process. One possibility is the asynchronous communication between services by publishing and subscribing to events. This communication pattern is called *Choreography* which is discussed in detail in Section 3.7.

The next service operation we focus on is accessing customer master data as an administrative user via the admin web interface of *in|FOCUS*. The monitored behavior of the software system is illustrated by Figure 3.11. As before, this invoked behavior can be broken down into the following four steps:

① The `permission` package is responsible for checking whether the user has sufficient rights for using internal administrative actions such as checking the customer master data.

② Here, the required user information is acquired for checking the user's permission.

③ The invoked behavior of the `identity` package confirms the logged-in status of the administrative user.

④ As the final step, the requested customer master data is gathered and presented.

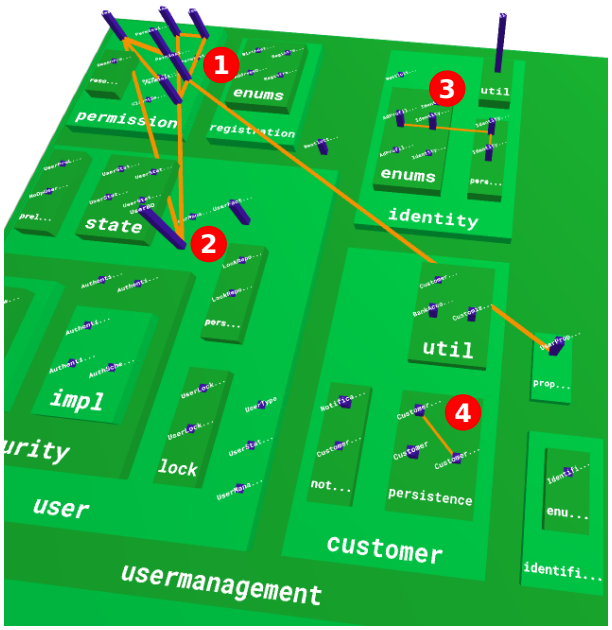


Figure 3.11. Monitoring the *in|FOCUS* behavior with ExplorViz of accessing customer data as an administrator.

When we compare the involved classes of an administrative user action to a customer user action, we can identify a dividing line between them. The main functionality of the administrative section is mostly handled by classes which are not involved in the customer functionalities. Therefore, it is possible to outsource separate *User* service from the already established *Customer* service. As a result, the internal size of the later is reduced. This counteracts a possible devolving of a over-sized SCS into a microlith and makes the *Customer* service easier to handle.

Addressing the "Payment-Problem" with the help of ExplorViz

As a next step, we continue the investigation of the idea shown in Figure 3.9a. The goal is to let the *Customer* and *Gaming* services communicate asynchronously with the *Payment* service. However, the independence of the owned service operation of each service has to be maximized. Since the current definition of the service boundaries leads to tight coupling of the three mentioned services when it comes to buying a product of the lottery application, we need to redraw these boundaries. Exemplarily, Figure 3.12 shows the system behavior when a customer transfers money from his registered bank account to his online wallet. This process can be broken down into the following seven distinct steps:

① These classes are the centerpiece of this service operation and direct the entirety of the payment process.

② Here, the necessary permissions for transferring money to the target online wallet are checked.

③ Furthermore, general user checks, such as sufficient user rights as well as the user status, are conducted inside the `user` package.

④ The `customer` package delivers the required information on the customer and his or her associated online wallet.

⑤ Inside the `workflow` package, the processing of the selected payment method is managed and directed.

⑥ This package is responsible for managing the appropriate business process. For that, business records are created in order to track the respective process.

⑦ Here, the account of the customer online wallet, onto which the money is transferred to, is managed.

In order to reduce the dependencies between *Customer* and *Payment* while enabling a feasible asynchronous communication between these services, we redefine the *Payment* service into an *Order* service. This service handles the payment process as well as the shopping cart which contains the items to buy. The shopping cart was previously part of *Customer*. This previous assignment of the shopping cart however led to unwanted

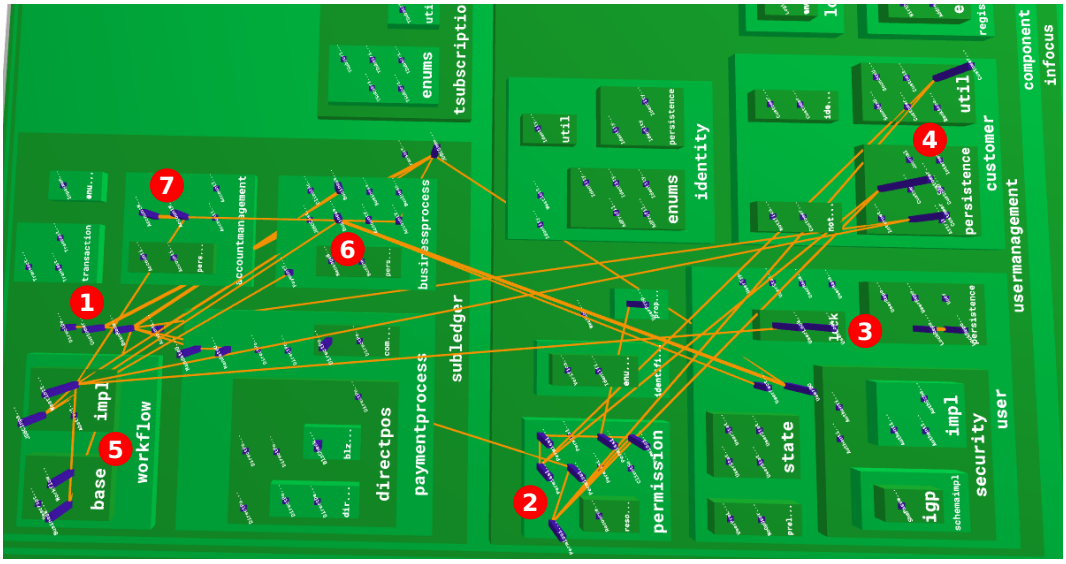


Figure 3.12. Monitoring the *in|FOCUS* behavior with ExplorViz of a customer transferring money to the online wallet.

communication between the *Gaming* service when buying lottery tickets. With the new assignment of the shopping cart component, both the *Customer* and *Gaming* service signal the *Order* service asynchronously to put a requested product into the cart. This can possibly be done through choreographed publish/subscribe events, which are discussed in Section 3.7. This has the advantage that all products can still be viewed or requested to buy even when the *Order* is not available at that time. Furthermore, no more inter-service communication has to take place in order to complete the payment process. The other services can then be informed via asynchronous communication to confirm the successful execution of the payment process.

This redefinition of the *Payment* service showcases the benefits of a dynamic analysis. By monitoring the runtime behavior of the software application, we gained new insights which were previously overlooked by the static analysis. Consequently, we were able to improve a previously made architectural design decision with the help of ExplorViz.

3.6 Database Analysis of *in|FOCUS*

After dividing the monolithic software architecture into distinct services, the data model of the application has to be adapted to the modernized architecture style. Otherwise, the flexibility and the vertical scalability of the services are limited. Furthermore, the enforcement of a single data model can lead to a bottleneck in the development process.

For large-scale software projects, the data model is often maintained by a separate team of developers. Consequently, the introduction of new features to the system oftentimes requires changes of the data model which have to be coordinated with the database team [Bindick and Stoye 2018]. This creates unwanted overhead and hinders an agile development process.

Therefore, the centralized data governance has to be replaced by a distributed data model. The goal is to follow the suggestions of both the SCS and microservice architecture style so that every service owns its own data. Usually, we distinguish between *pseudo-distributed* and *distributed data governance*. Both approaches are depicted in Figure 3.13. The first solution divides the centralized data model into separate data schemata within a single database. This enables a developer team of a specific service to independently own and change the respective schema without any additional communication to the other teams. Additionally, the data model can be adjusted to the specific needs of the service. In contrast to the first, the distributed data governance approach splits the data model further and places the different schemata into separate databases. This allows the employment of a diverse array of database types to exploit certain features of different database technologies, dependent on the characteristics of the internal data. The distribution of data of multiple machines also enables targeted scaling of each unit.

The approach of finding bounded contexts within the *in|FOCUS* of Section 3.4 is reused for the purpose of dividing the data model of the *in|FOCUS* application. Therefore, we statically analyze the data model alongside previously established service operations (see Table 3.2) of the lottery application. With the help of the JDBC driver *log4jdbc* (Section 2.9) and the database administration tool DBeaver (Section 2.8), the database tables which are

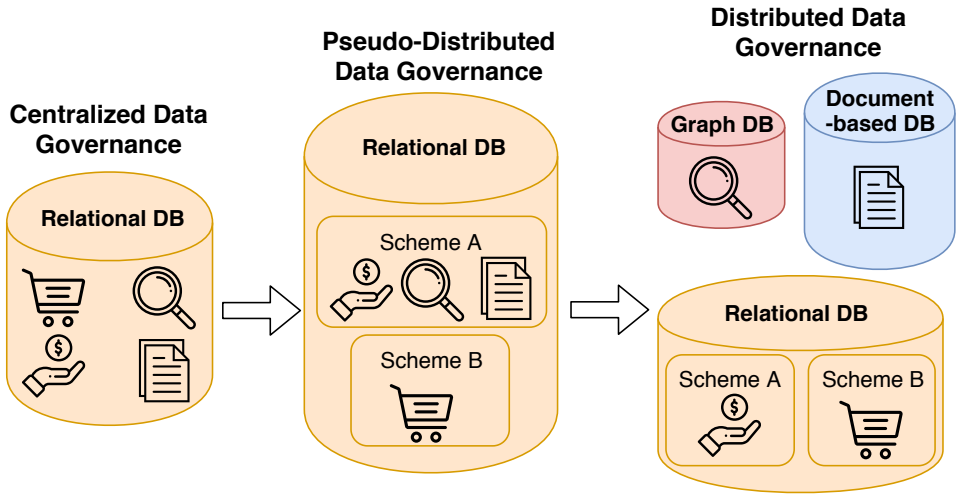


Figure 3.13. The transition process from a centralized to a distributed data governance.

accessed by each service operation are identified and mapped onto the respective bounded context. This investigation of transaction boundaries and the resulting assignment provides us with an individual schema for each service [Newman 2015a]. Afterwards, the tables are adjusted to their context.

Figure 3.14 exemplifies the identification of the required *in|FOCUS* database tables by the *Gaming* service. Here, we determine the utilized database tables for the service operation of loading and filling out a lottery ticket. The dashed lines between the tables indicate a foreign key relation. As an example, the table `USERS` as well as `CUSTOMERS` contain a foreign key of the `CLIENTS` table. Therefore, the two can be seen as a specialized subset of `CLIENTS`. On the one side, `USERS` handles the general attributes such as system locks and user roles. On the other side, `CUSTOMERS` contains specific information for *in|FOCUS* lottery customers. All the shown database tables can be assigned to the *Gaming* service.

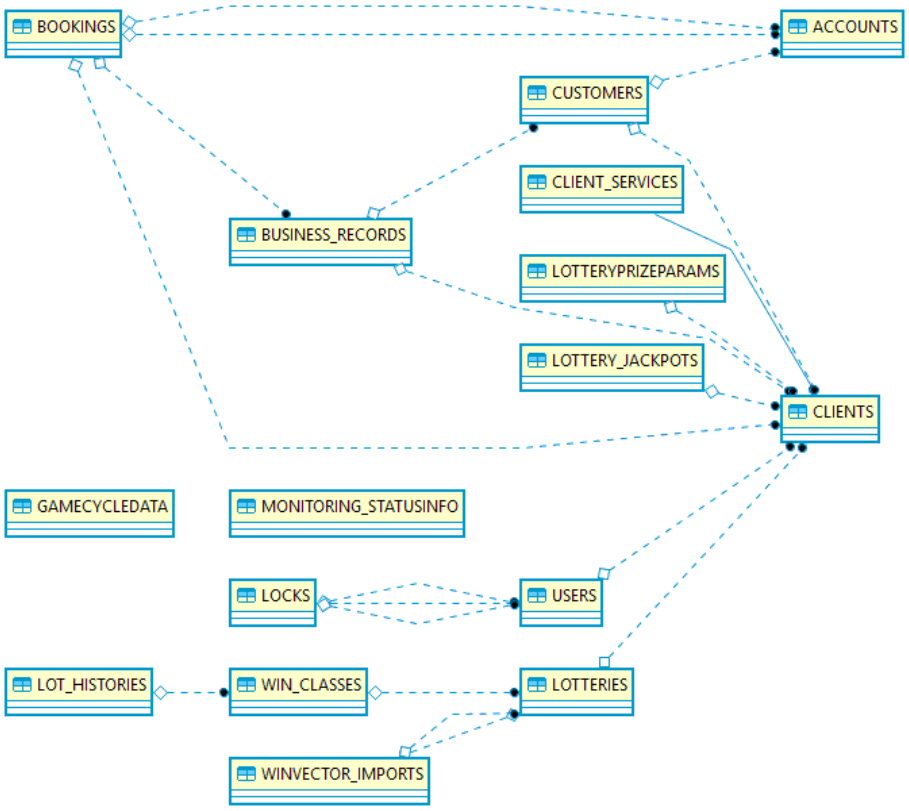


Figure 3.14. Identification of the required database tables by the *Gaming* service for loading and filling out a lottery ticket.

However, some tables are used by multiple service. Then, the data model of a service can be improved by customize the those tables to specifically fit the requirements of the table as well as conform to the ubiquitous language of its context.

One of these tables is `USERS` which is part of every defined SCS of the system. Though, the definition of a user differs depending on its context. In contrast to the customer management context of the *Customer* service, the *Gaming* service sees the user more a player of a game rather than a customer. As a result, the *Gaming* service may need different attributes of the player than the other services. Figure 3.15 illustrates the different definitions of a user and demonstrates a distribution of attributes depending on the context.

Admittedly, a distributed data governance comes with its challenges which have to be addressed. First, the introduction of multiple database technologies to the software system increases the complexity and the required know-how of the developers. Furthermore, a centralized database supports ACID (Atomicity, Consistency, Isolation, Durability) transactions. On the contrary, transactions over multiple databases and therefore over multiple distributed services cannot guarantee ACID. Though, it is possible to support these properties for transactions within a single service. In the following, we present design patterns which aim to address these challenges [Gonchar 2018].

The *Aggratage* pattern introduces so-called aggregates to the data model. Each aggregate summarizes business objects which are functionally related to each other. There can be multiple aggregates per bounded context. Then, data consistency has to be ensured for these aggregates. This pattern helps the developer to identify critical data dependencies

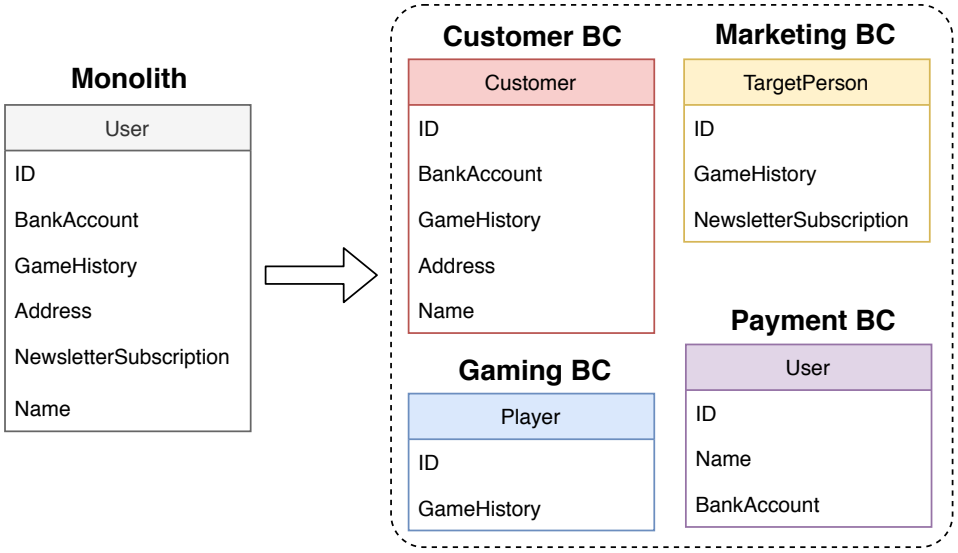


Figure 3.15. Division of the monolithic data model into separate data models for each bounded context (BC).

within their system. An example of an aggregate would be the bank account balance and the amount of money that the user wants to withdraw. Naturally, the withdraw amount should not exceed the available amount of the bank account.

Oftentimes, we need a procedure to gather data for a specific use case from a set of services which hold the required data, especially when it comes to microservice architectures. In order to keep the communication structured while ensuring loose coupling of the services, the clients should not directly communicate with the services. Instead, an *API gateway* is introduced which takes client requests and gathers the data from the involved units. More information on API Gateways can be found in Section 3.7.

Lastly, the *Saga* pattern presents a possibility to assure data consistency within distributed systems. This pattern, which is already applied in SOA architectures, can also be ported to SCS or microservice architectures. Generally speaking, an atomic business operation which involves multiple services is possibly comprised of multiple transactions. A saga is defined as a sequence of local transactions. Each of these local transactions updates the database and then publishes an event to trigger the next local transaction in the saga. In case a local transaction fails due to rule violations, the saga executes a series of compensating transactions that roll back the changes which were done by the preceding local transaction. The event management can be realized as either through orchestration or through choreography which are presented in the following Section 3.7.

3.7 Composition of the Services

In order to fully benefit from the decomposition of the monolithic software system *in|FOCUS* into multiple SCSs and/or microservices, these services have to collaborate with each other in an appropriate manner. Therefore, several integration patterns have been developed to meet different requirements of the software engineer. Even though the application consists of several distributed services, the user experience should not differ from a monolithic system. Therefore, the cooperation of the services should be seamlessly [Newman 2015a].

UI Fragment Composition

A commonly used pattern when it comes to the integration of SCSs is the *UI Fragment Composition* [Newman 2015a]. The UI of the application is composed of several fragments, each fragment being an individually provided UI component by an SCS. Figure 3.16 on page 52 illustrates this composition of the defined SCSs of Section 3.4 as an example. Then, these components can be represented as widgets in the web frontend. A more coarse-grained approach is the composition of a set of web pages which are provided by the SCSs. These are then connected via hyperlinks. Alternatively, the client can load different parts of the web page when requested with the help of JavaScript and *AJAX* (Asynchronous JavaScript and XML). This can be especially helpful when generating a page which is a

less prominent part of the frontend (e.g. shopping cart page for an online shop) with the help of different services [Steinacker 2015]. To ensure a uniform feel and look of each UI component for the user, an *asset server* can be used to manage static assets such as *Cascading Style Sheets* (CSS), HTML components or pictures for every UI fragment [Steinacker 2015]. A major advantage of this approach is the independence of the different developer teams. Hence, the team which develops a certain service can also change its UI independently. This separation of concerns is reinforced not only within the software system, but also in the development structure and therefore supports agile software development principles.

API Gateway

API gateways serve as an interface for the communication between the frontend and backend services [Newman 2015a]. This technical solution does not only promote the independence and the loose coupling of the services, but also enables the provision of varying content depending on different types of devices (e.g. mobile app, web app). Figure 3.17 depicts this concept which is often employed in microservice architectures. Here, an API gateway coordinates the communication between both a mobile and a web app and the determined SCSs of the *in|FOCUS* application. In general, the API gateway encapsulates the clients from the microservices and therefore supports the isolation concept of the architecture style. Furthermore, the API gateway can implement additional features such as load balancing, routing, and client authentication [Richardson 2018]. However, the centralized handling of the different UIs can negatively impact the ability to release the interfaces independently. It may not be possible anymore to change the UI of the mobile app without affecting the UI of the web app. A feasible solution to this problem is to split the API gateway into dedicated backends for each frontend, called *backends for frontends* [Newman 2015a].

Anti Corruption Layer

When it comes to industry software modernization projects, only certain parts of a software application might get targeted for rework due to a lack of resources. Consequently, the modernized system has to integrate parts of the legacy application which can pose great risks if not done correctly. In order to avoid an unwanted adoption of the legacy data model and enable the newly chosen concepts and design choices to be independent to the legacy system, an *anti-corruption layer* is introduced between the two. Figure 3.18 shows an application with two subsystems, a modernized and a legacy subsystem. This isolating layer functions as a translator for both subsystems and tailors the sent requests to fit the internal data model of the recipient. Usually, the anti-corruption layer is implemented as a standalone component which owns all the necessary logic for its task. Naturally, the introduction of a new layer always increases the complexity of the system. Furthermore, the design of the anti-corruption layer should support easy scaling. It is the crux of the communication between subsystems and therefore a bottleneck of the application architecture [Brown 2014].

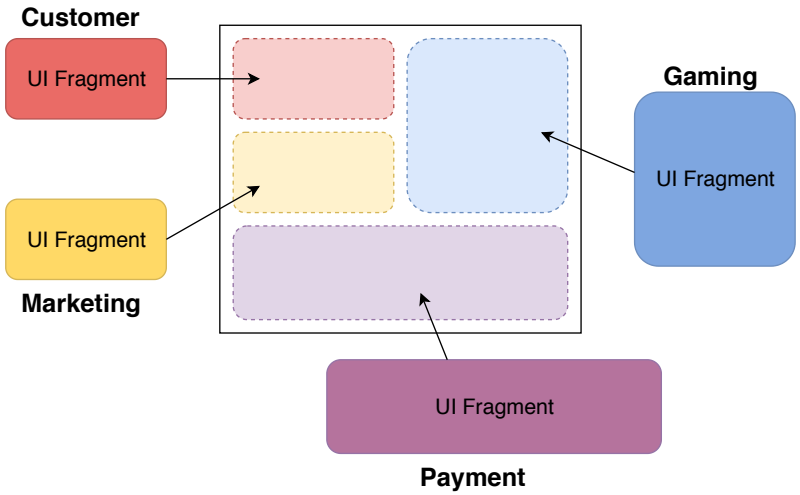


Figure 3.16. UI components of each SCS are composed into a single UI of the application.

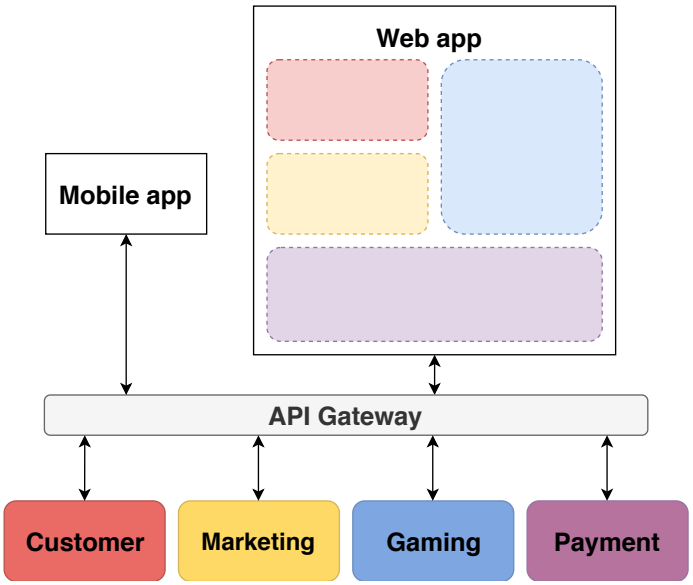


Figure 3.17. The API gateway handles calls to and from different frontends.

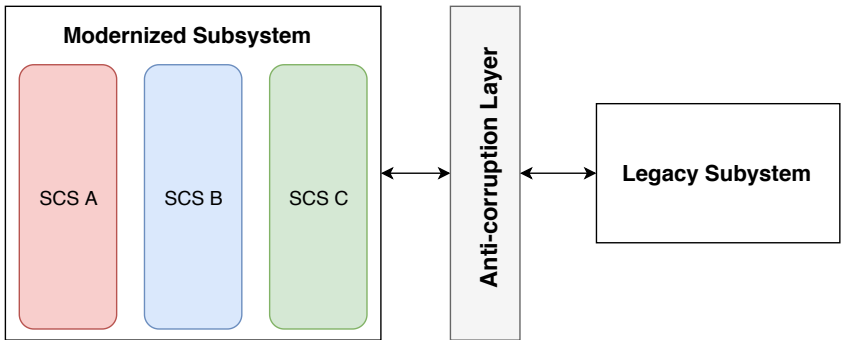


Figure 3.18. The anti-corruption layer manages the communication between the modernized subsystem and the legacy subsystem.

Orchestration

The organization of communication between the services can be realized in different ways. One of the often used service-oriented architecture (SOA) paradigms is the *orchestration* of services. Here, one of the services, called the *composer*, presents a focal point of the business logic. Through synchronous or asynchronous calls, the service composer directs the involved services through the invoked use case and tracks its progress. Therefore, it functions as a conductor to an orchestra of services, hence the name. Figure 3.19 shows a possible orchestration of the use case *"Create customer account"* of a lottery application. Here, the customer service acts as the composer and asynchronously directs three involved microservices to complete the customer creation process. However, it becomes apparent that this organization concept creates coupling between the composer and the other services. Additionally, the introduction of a central directing authority goes against the general principles of the SCS and microservice architecture concepts [Newman 2015a].

Choreography

The *choreography* approach also originates from SOAs and represents an alternative to the preceding orchestration concept. The directing unit is replaced by a service composer which publishes events to which services can subscribe to and react accordingly. Compared to the service orchestration, the choreography provides a higher degree of decoupling between these services. As a downside, additional effort needs to be made for monitoring and progress tracking. Figure 3.20 depicts the publishing and subscription process of the previously presented use case *"Create customer account"*.

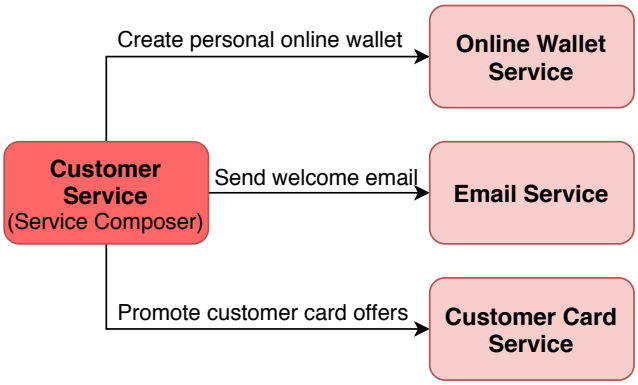


Figure 3.19. Orchestration of the customer account creation.

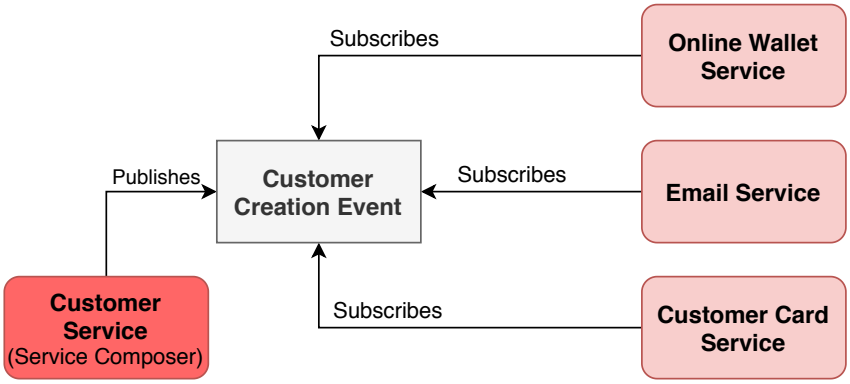


Figure 3.20. Choreography of the customer account creation.

Chapter 4

Evaluation

In order to assess the impact of the live trace visualization tool ExplorViz on the task of microservice discovery, a qualitative evaluation is conducted. To that end, four software developers who were involved in the development process of the software application *in|FOCUS* were asked to use ExplorViz for microservice discovery in *in|FOCUS*. Afterwards, a guided interview was conducted in order to capture the user experience. In this chapter, the answers giving during these interviews are qualitatively evaluated.

The broad goal of this endeavor is the gathering of new insights about the supporting features of ExplorViz which enable the division of large-scale software applications into microservices. The chapter is structured as follows: Details about the evaluation goals, the research questions, and the hypotheses of the evaluations are given in Section 4.1. Section 4.2 presents the preparation process of the interviews which is followed by the execution details described in Section 4.3. Thereafter, Section 4.4 presents and the discusses the results of the interviews. This chapter concludes with the discussion of possible theats to the validity of the evaluation results in Section 4.5.

4.1 Goals

The goal of this evaluation is the assessment of the capabilities for analyzing a large-scale software application and discovering a microservice architecture with the help of ExplorViz. We try to identify features of ExplorViz which support the developer during this discovery phase and which add value to the architecture analysis. Additionally, features are identified which in the current implementation of ExplorViz are considered to obscure the discovery functionality. Moreover, suggestions for future improvements, in particular requests of missing features, are collected and discussed below.

Typically, software modernization is a long-term commitment due to the complexity inherent in the associated processes. Therefore, an analysis software such as ExplorViz should not only be efficiently usable by software architects who have many years of experience with the target architecture. It should also support the training and incorporation of new developers to the architecture. Hence, the assessment of these capabilities of ExplorViz are also part of the interviews. Depending on the used modernization workflow, further development of the functionality of the application does not only happen after finalizing the software modernization, but also in parallel to the modernization process.

Consequently, an assessment of the support for the daily development work provided by ExplorViz is presented at the end of this chapter.

4.1.1 Research Questions

The following research questions are derived from the goals presented above:

RQ1 Does the software architecture visualization of ExplorViz have a positive impact on the software modernization process of a large-scale software architecture? In particular, does it do so by supporting the discovery and verification of bounded contexts in order to divide the existing software architecture into microservices?

RQ2 Does ExplorViz support the software architect's training and discovery of unfamiliar parts of the architecture?

RQ3 Does ExplorViz support the analysis of a large-scale software architecture in the daily developing work of software architects?

RQ4 How can ExplorViz be improved to increase the support of the software modernization process?

4.1.2 Hypotheses

The following proposed hypotheses will be discussed and either verified or disproved by the results of the interviews.

H1 ExplorViz makes it easier for developers with prior in-depth knowledge of the analyzed architecture to find and verify bounded contexts inside the existing software architecture.

H2 ExplorViz supports the introduction of previously unfamiliar parts of the system architecture to a software architect.

H3 ExplorViz enables its user to verify the existing architecture and therefore have a positive impact on the daily developing work.

4.2 Method

4.2.1 General Interview Guide Approach

For the evaluation of this thesis, the semi-structured *General Interview Guide Approach* [Gall et al. 2003] is chosen. Every participant of this interview is presented with the same task. Prepared open-ended questions aim to guide all participants towards the same general direction. However, the investigator still remains flexible in order to adjust to the course of

the interview which is based on the participant's assumptions and conclusions from the given task. This approach also provides the interviewees with the freedom to encourage creativity and express their own subjective opinions on the given matter. This flexibility is mandatory for the evaluation of a highly explorative task such as discovering microservices within a large-scale software architecture with the help of ExplorViz. There are countless possibilities of dividing an architecture without there being a right or wrong answer most of the time. It can expected that answers may vastly differ from one participant to the other because each of them might bring in a different level of experience and knowledge about the software architecture as well as the discovery and development of microservices. The given answers of the participants are recorded throughout the interview and later summarized in writing by the investigator.

4.2.2 Recruitment of Participants

Since the architectural analysis of *in|FOCUS* may provide a deep insight into possible sensible information and company secrets, it is necessary to limit the pool of possible participants to employees of adesso only. Furthermore, an advanced understanding of software architecture development is a prerequisite for all participants. The discovery of microservices is a convoluted task which requires a good overview over possible problems and opportunities of different architecture styles and patterns. Additionally, all participants should be familiar with the architecture of the *in|FOCUS* application. Without in-depth knowledge about the inner structure and behavior of the software system, it is unlikely to develop a substantially justified solution for the task at hand.

With the help of the current project owners of *in|FOCUS* of adesso Hamburg and adesso Dortmund, software developers, who are actively working on the development *in|FOCUS*, were selected and recruited via email. Due to the low number of participants, it was not deemed necessary for a prior classification of the participants.

4.3 Interview

4.3.1 Setup

The interviews were carried out within a single day at the adesso office in Dortmund. A separate room was reserved for the interviews to ensure an undisturbed working environment. Each interview was planned to take approximately 30 minutes in total. In order to maintain good performance of both ExplorViz and *in|FOCUS*, two notebooks were prepared. *Notebook 1* ran an in-development snapshot of ExplorViz. This version was provided as docker images[1]. Additionally, this notebook also ran a kieker-instrumented dummy application, called *kiekerSampleApplication* for introductory purposes of ExplorViz.

[1] `https://github.com/ExplorViz/docker-configuration`, accessed 24.06.2019

This Java application simply generates a constant stream of monitoring data by continuously calculating Fibonacci numbers and executing SQL queries[2].

Notebook 2 ran a released version of *in|FOCUS* which was instrumented by Kieker. Here, the number of probed packages had to be limited to a subset due to technical limitations of the used version of ExplorViz, as it was case for the architectural analysis of Section 3.5. This software application was deployed on a JBoss EAP 6.3 application server, provided by adesso. It was customly reconfigured to allow the instrumentation of Kieker. For further detailed information, see Section 3.5.1. Table 4.1 summarizes the relevant hardware configurations of both notebooks.

Table 4.1. Used notebook configurations for the evaluation

	Notebook 1 (ExplorViz)	Notebook 2 (*in|FOCUS*)
OS	Windows 10 Pro 64-bit	Windows 10 Pro 64-bit
CPU	Intel Core i7-8650U, 1.9GHz	Intel Core i5-4310U, 2.0GHz
RAM	32 GB	8 GB
Display	15 inch, 1920x1080	14 inch, 1920x1080
Peripherals	USB Mouse	USB Mouse

As the more powerful notebook, Notebook 1 was chosen for the execution of ExplorViz since the analysis and visualization of a large-scale software system such as *in|FOCUS* is very computationally intensive. The Kieker records of *in|FOCUS*, which were gathered on Notebook 2, were sent to Notebook 1 for further analysis of ExplorViz. At the time of this evaluation, no external displays were available at the adesso offices which could have been used for this study. It is noted that visually complex tasks, such as the microservice discovery and analysis with ExplorViz, may benefit from an increased display size.

4.3.2 Scenario

Each participant starts the interview with the same scenario in order to ensure equal preoperative conditions. On Notebook 1, the ExplorViz website is opened in the chosen browser. It shows the 3D visualization of the kiekerSampleApplication which is used for introducing the main features of ExplorViz. Afterwards, the kiekerSampleApplication is exited and the 2D landscape visualization of the *in|FOCUS* software application is opened on the ExplorViz website. No parts of the *in|FOCUS* architecture have yet been visualized as a 3D model, except the ones which are active during its startup process.

On the second notebook, the welcome page of the locally hosted *in|FOCUS* website is opened in the chosen web browser. Furthermore, the participant is provided with the login information of a dummy account for the *in|FOCUS* home page.

[2] `https://github.com/czirkelbach/kiekerSampleApplication`, accessed 24.06.2019

4.3.3 Execution

This subsection discusses the different steps throughout the course of the interview. Furthermore, the investigative questions are presented and discussed. They aim at guiding the participant through the semi-structured interview process towards the desired evaluation goals.

Initial Part

After obtaining the participant's approval for recording the interview, it begins by gathering relevant personal information. The interviewee is asked about the number of years of professional experience as a software developer, about their prior knowledge of the architecture of the software application *in|FOCUS* as well as any prior experience with ExplorViz. The answers to these initial questions enable a possible classification of the participants when discussing the results of the interview. They provide great insight into possible factors which are relevant to the discovery of microservices inside a large-scale software architecture with the help of ExplorViz. Thereafter, it is ensured that the participant does not suffer from any visual impairments such as color blindness which could affect the intended usage of ExplorViz since information is presented to the user via utilizing certain color codings in the visualization, see for example Figure 2.6.

This introduction is concluded with an overview of the upcoming steps of the interview. It is established that during the interview process, the people involved will both analyze the architecture of the software application *in|FOCUS* with the help of the live trace visualization tool ExplorViz and discover microservice boundaries in the given architecture.

Introduction to the Topic

Before commencing the main part of the interview, each participant will receive a technical introduction to ExplorViz and familiarize themselves with the basic functionality of ExplorViz. The web interface as well as all required features of ExplorViz for the given task will be presented with the help of the kiekerSampleApplication. This Java application generates monitoring data for the ExplorViz demo analysis. The introduction focuses on the 3D application visualization of ExplorViz, since the to be analyzed monolithic software application runs on a single system node and is not distributed onto multiple systems. Hence, the 2D landscape visualization shows only one node with no communication to others and is therefore not used for the architecture analysis.

The participants are introduced to the visual presentation of packages, classes, and their communication. The interaction with the architecture model by moving the model, opening and closing packages, and receiving additional information by mousing over different elements of the model is demonstrated. Moreover, the usage of the timeline and the trace replayer are presented. Finally, a common definition of microservices, as stated in Section 2.2, and their discovery approach with the help of bounded contexts, see Section 3.2, is established in order to avoid later misunderstandings.

Investigative Questions

The following part presents the investigative questions of the interview and discusses the reasoning behind each of them. Generally speaking, the seven questions **Q1**–**Q7** are associated to the four topics, namely *Area of Expertise*, *Architecture Discovery*, *Architecture Modernization* and *Feedback*, as described below.

1. Area of Expertise

   This topic establishes the participant's exact area of expertise in the development process and inside the software architecture of *in|FOCUS*. This initial question is essential for the successful execution of the interview. For the ongoing microservice discovery, it is desirable that the participant focuses on areas of the architecture that he or she is most knowledgeable of. Thus, a more informed decision can be expected when choosing boundaries of bounded contexts.

   Q1 *What is your part in the development process of in|FOCUS? Which parts of this software system did you develop?*

2. Architecture Discovery

   The second topic verifies two things, namely the participant's ability to use ExplorViz and the understanding of the presented information of the visualized 3D model. This gives the investigator the opportunity to clarify and solve possible misunderstandings before the main part of the architecture analysis commences. Furthermore, it shows another use case of ExplorViz to the participant, that is to say, the possibility for verifying the existing architecture. This verification could provide the following insights to the developer. Either there are problems in the current implementations, since parts of the application do not behave as expected, or the developer has a faulty or outdated knowledge of this part of the application. In both cases, this awareness is crucial for a successful software modernization process.

   Q2 *Execute a service operation on the locally hosted version of the in|FOCUS website, preferably one that you implemented. Can you identify the behavior of a service operation with ExplorViz? Is the behavior of this service operation as expected? If not, what is the difference?*

3. Architecture Modernization Process

   This topic represents the main part of the analysis. Its goal is to provide new insights into the ability of ExplorViz to support the discovery process of microservices inside a large-scale monolithic software architecture. This part comprises two steps. Firstly, the discovery of a possible microservice by drawing boundaries inside the existing monolithic software architecture, and secondly, the evaluation of these boundaries with the help of the dynamic analysis of ExplorViz. During this process, the participant can maneuver freely through the *in|FOCUS* website and its architecture. This procedure emphasizes an explorative way of discovering microservices and gives the interviewee

as much freedom as possible. Only when asked for, the investigator will give guidance to the participant.

Q3 *Your goal is to identify bounded contexts inside the in|FOCUS architecture. With your prior knowledge of the architecture, do you already have an idea where one of these boundaries could be in order to form a candidate for a microservice? If not, can you identify such a boundary with the help of ExplorViz?*

Q4 *Microservices should be loosely coupled and highly cohesive. With the help of ExplorViz, try to analyze service operations of your choice within your chosen boundary. Is it as loosely coupled and as cohesive as expected? What do you think about the quality of your defined boundary? Can you change it in order to improve its qualities for defining a microservice, such as loose coupling and high cohesion?*

4. Feedback

   These closing questions of the interview aim at gathering feedback from the participants after they gained first experience with the modernization process with the help of ExplorViz. The provided answers aim at giving an indication whether ExplorViz seems to be useful for the modernization of a monolithic software application towards a microservice architecture. In particular, possible extensions and missing features, which are deemed to improve the usability of ExplorViz for this use case, are discussed. After modernizing a software application into microservices, this reorganized software system still needs to be maintained and monitored. Therefore, it is also of value to address the continuous integration of ExplorViz into the daily work routine of a software developer.

   Q5 *In which way do you think does ExplorViz support the discovery of bounded context boundaries inside the architecture of an application?*

   Q6 *Could you think of a missing feature that would improve the usability for the architecture and microservice discovery?*

   Q7 *In which way do you think could ExplorViz support you in your daily work as a software developer?*

4.4 Results

This section presents and discusses the findings and results of the interview process described in Section 4.3. First, Section 4.4.1 introduces the four participants **(I1)–(I4)**. Thereafter, Section 4.4.2 presents the answers provided by the interviewees and last, **??** discusses these answers and the findings of this evaluation process.

4.4.1 Participants

Table 4.2 gives an overview over the participants' years of professional experiences as a software developer and with the software application *in|FOCUS*. Generally speaking, the

Table 4.2. Overview of the interviewees' experiences

	Experience software developer	Experience *in*|*FOCUS*
Participant 1 (**I1**)	9 years	3 years
Participant 2 (**I2**)	12 years	2 years
Participant 3 (**I3**)	10 years	3 years
Participant 4 (**I4**)	7 years	2,5 years

participants are experienced software developers with several years of experience with the *in*|*FOCUS* architecture. It remains to be determined whether the notable difference in years working as a software developer between participant **(I2)** and participant **(I4)** is of any significance. Also, the one-year experience difference of participant **(I2)** and participant **(I3)** should be kept in mind. Furthermore, none of the interviewees had prior experience with ExplorViz. Additionally, it is noted that neither of them suffer from visual impairments such as colorblindness that could impair the work with ExplorViz.

4.4.2 Answers

Next, the participants' answers to the previously formulated questions are presented.

Q1 *What is your part in the development process of in*|*FOCUS? Which parts of this software system did you develop?*

I1: The first participant described himself as a full-stack developer, having worked on both frontend and backend development of *in*|*FOCUS*. In the backend, the interviewee worked with most of the modules of the application and has great in-depth knowledge of the whole architecture. Besides the development of the functionality of *in*|*FOCUS*, the participant also worked as a software architect for past architecture development projects in the *in*|*FOCUS* context.

I2: The second interviewee works on the backend development of *in*|*FOCUS*. His daily work includes implementing change requests, doing error search and correction as well as redeveloping already existing functionalities. The participant mentions the customer payoff module as being part of his personal ongoing development work. This module transfers winnings to the customer online wallet rather than to the customer bank account.

I3: The responsibilities of the third participant include requirements engineering, communication with clients, and the definition of development goals and processes. Therefore, after being actively involved in the development of the management of customer cards inside the *in*|*FOCUS* software application, the interviewee is not implementing code for the analyzed application. Hence, the third participant's knowledge of the architecture can be considered as more general and less specific.

I4: The last interviewee develops the backend of *in|FOCUS*, implements change requests as well as error search and correction. Furthermore, other daily tasks include the specification of requirements and module testing. The participant exemplarily developed parts of the lottery ticket submission and the lottery gaming process.

Q2 *Execute a service operation on the locally hosted version of the in|FOCUS website, preferably one that you implemented. Can you identify the behavior of a service operation with ExplorViz? Is the behavior of this service operation as expected? If not, what is the difference?*

I1 The participant logged in on the *in|FOCUS* website with the previously provided login information of a dummy customer. Afterwards, the lottery system ticket was loaded as a second use case. The invoked behavior and the activated parts of the architecture of both use cases were successfully identified with the help ExplorViz. The interviewee confirmed that the general behavior was as expected and the correct modules were active during the use cases. However, the participant realized that the number of requests and active instances of the involved modules were higher than they should be. Therefore, it was concluded that this possibly needs further investigation and a possible patching.

I2 In the second interview, the customer login was executed with the given login information. Its behavior was identified with ExplorViz and the participant confirmed that it was depicted as expected.

I3 The participant logged in as a customer on the *in|FOCUS* website and opened the overview of the available customer cards of this customer. First the login process was analyzed with ExplorViz. While having no in-depth knowledge but only a higher-level expertise of the login functionality, the participant was surprised how many components were involved in this process. The trace replayer of ExplorViz was then used in order to discover the login process in detail. The interviewee concluded that it was very hard to understand the exact behavior with the help of ExplorViz without having prior in-depth knowledge of the actual behavior. While receiving information on the called packages and classes, the interviewee missed the possibility for gathering detailed information about the content of the communication, exemplarily the name of the sent object between two classes. After having monitored the second executed use case, as to say the presentation of all available customer cards of this customer, the participant stated that its behavior was as expected.

I4 As the previous participants, the final interviewee executed the customer login. Thereafter, a lottery game was chosen, its lottery ticket was filled out and submitted afterwards. While analyzing the behavior of the lottery ticket submission with ExplorViz, the participant quickly realized that the use case of submitting a filled out lottery ticket is too complex to be analyzed in the given time of the interview.

A considerable amount of packages from most parts of the architecture is needed to successfully fulfill this use case. Furthermore, the communication between them is convoluted which significantly complicated its analysis. Consequently, the participant chose the customer login process for further analysis whose behavior was as expected.

Q3 *Your goal is to identify bounded contexts inside the in|FOCUS architecture. With your prior knowledge of the architecture, do you already have an idea where one of these boundaries could be in order to form a candidate for a microservice? If not, can you identify such a boundary with the help of ExplorViz?*

I1 The participant presented several suggestions: The encapsulation of all the loading of all necessary parameters of the lottery tickets, such as the date of the next lottery drawing, the costs of a specific lottery ticket. Another mentioned possibility was the encapsulation of the user management or the ticket submission. The last suggestion was the extraction of the subledger of the lottery application.

I2 Initially, the participant could not suggest a boundary for a bounded context. The interviewee mentioned the high entanglement of the monolithic *in|FOCUS* application as the reason. Therefore, the participant executed several use cases of the customer management functionalities of the website to discover a bounded context. These executed use cases included several ones such as changing the customer address and the customer banking information. Finally, the customer management was suggested as a possible candidate for a microservice. However, is was also mentioned that the size of this service would possibly exceed the dimensions of common microservices.

I3 The customer card management was the participant's initial suggestion for a microservice candidate. As an alternative, the lottery ticket management was mentioned.

I4 The interviewee suggested two possible candidates for a bounded context. First, the user management and secondly the game processing. The user management service should not only contain the management functionality for the customer but also for other users, such as system administrators. The possibility for a further division of this service into distinct management services for each type of user and each type of distribution channel was mentioned. Exemplarily, different services for internet customer management, for customer card customer management, and for administrator management could be taken into consideration. To further break down a user management service, one could divide it into multiple small microservices such as a login service and a service for managing the rights of user accounts. Additionally, the participant suggested a lottery ticket microservice. This service would provide a specific lottery ticket for each different lottery game which is available on the *in|FOCUS* website. Furthermore, the task of the microservice should include the buying and the submission of the lottery ticket.

Q4 *Microservices should be loosely coupled and highly cohesive. With the help of ExplorViz, try to analyze service operations of your choice within your chosen boundary. Is it as loosely coupled and as cohesive as expected? What do you think about the quality of your defined boundary? Can you adjust its boundaries in order to improve its qualities for defining a microservice, such as loose coupling and high cohesion?*

I1 The participant first focused on the idea of a ticket parameter microservice. When loading a specific lottery ticket in the ticket office on the *in|FOCUS* website, every parameter such as the price of the ticket is initialized. Hence, the participant opened lottery tickets of different lotteries in sequence and monitored the invoked behavior in ExplorViz. The dynamic analysis confirms this thought that the packages which are responsible for loading the lottery tickets are already very loosely coupled with the rest of the application. Therefore, these results encouraged the interviewee to related back to his previous choice, which seemed to be a fitting candidate for a bounded context. However, the interviewee realized a possible problem with this bounded context as it was defined when it came to instant lottery games such as lottery scratch tickets. These games are handled differently in the current *in|FOCUS* architecture and require different modules than other lottery games. This could pose a problem for the clear definition of the single purpose of this microservice. Therefore, the participant suggested that it might be better to split the lottery games and instant lottery games into two different microservices. This however would require further analysis.

I2 The participant realized that the previously defined customer management service would be very extensive in its size and it would therefore qualify more as a self-contained system rather than as a possible microservice. With the support of ExplorViz, the participant analyzed possible use cases which would fit to the customer management service. Following discovery was made: The packages of the payment process of *in|FOCUS* are not only used when paying for a lottery ticket. However, multiple other use cases require the same parts of the payment process and are therefore and tightly coupled to it. Among others are the management of the customer online wallet, the management of customer winnings, and the transfer of money to the customer online wallet. Initially, all of the mentioned use cases were supposed to be part of the proposed customer management service. Since, while being tightly coupled to the customer management, the payment process should not be part of the customer management service. The interviewee's idea was to extract the dependent use cases from the customer management service and define them as their own microservices. However, the participant is certain that this would present new challenges. Therefore, further analysis would be necessary.

I3 The interviewee continued the analysis of the bounded context of the lottery ticket management by using the trace replayer function of ExplorViz. The participant discovered that the packages, which are responsible for gathering the necessary attributes for a lottery ticket inside the chosen ticket attribute service boundary, are also used in the ticket submission process. This posed a problem for the clear definition and the loose coupling of the developed microservice. The interviewee stated that the ticket submission process should not be part of the ticket management service. As a consequence, the ticket submission should either be its own or part of another microservice. Consequently, this creates unwanted dependencies between the ticket management service and the ticket submission process. Due to these issues, it is concluded that the chosen boundaries of the bounded contexts are insufficient and that there is no obvious fix for this problem. It will require further detailed analysis.

I4 The participant analyzed the previously defined bounded context of the lottery ticket microservice. It is realized that this lottery ticket microservice would include gateway service adapter in order to be able to independently fulfill its designated task. These gateway service adapters provide an interface for the connection and transfer of information to external software systems of the different online lottery gaming services, such as Eurojackpot and Toto. However, these gateway service adapters are also used by other tasks outside of the lottery ticket service. This would create unwanted dependencies between this microservice and other services. One idea was to define the gateway service adapters as a separate microservice. However, the interviewee was of the opinion that further analysis has to take place in order to make a qualified statement about this idea.

Q5 *In which way do you think does ExplorViz support the discovery of bounded context boundaries inside the architecture of an application?*

I1 The participant thought that ExplorViz provides a lot of useful information for the software architect. Even with very complex and highly tangled software systems, it is possible to extract critical information from the architecture visualization of ExplorViz, provided that the user has in-depth knowledge of this software system. The participant mentioned that the developer especially benefits from ExplorViz in regard to the verification of suggestions for bounded contexts.

I2 The interviewee saw potential in ExplorViz when it comes to supporting the software modernization process of large-scale software applications. The participant mentioned that the visualization of a large and complex software architecture such as the one of *in|FOCUS* can be overwhelming. Therefore, the user would need a significant amount of in-depth knowledge of the existing architecture for the efficient and goal-oriented use of ExplorViz.

I3 In the opinion of the third participant, ExplorViz supports the discovery of bounded contexts. However, analyzing complex use cases of highly entangled software systems could pose significant challenges. In this case, it is possible that ExplorViz lacks a way to filter the massive amount of presented information in a target-oriented manner. The interviewee mentioned the ticket submission process of *in|FOCUS* as an example of such a use case.

I4 The participant concluded that ExplorViz supports the discovery of microservices inside a large-scale software architecture. Especially the ability of ExplorViz to easily identify bottlenecks of a use case, can provide helpful clues for the software developer on where to find possible boundaries of a microservice.

Q6 *Could you think of a missing feature that would improve the usability of the architecture and microservice discovery?*

I1 The participant suggested a playback function for the history monitoring data which automatically executes a previously defined sequence of history monitoring data. This could improve the usability of the timeline function of ExplorViz by alleviating the user from clicking through the timeline him- or herself. Additionally, it was suggested that an introduction of a heat map, which shows most active parts of the architecture during this defined sequence, could be of great help for analysis tasks. It could greatly support the identification of possible bottleneck use cases in complex software systems. Furthermore, a heat map for the 2D landscape model could possibly highlight high-stressed services which are in need for vertical scaling.

I2 The participant could not think of a missing feature that would improve the usability of ExplorViz for the software modernization process.

I3 In order to possibly improve the architecture discovery capabilities of ExplorViz, the interviewee suggested a possibility for gaining more information about the communication between classes. By presenting details about sent objects, such as the object name, it might improve the support of ExplorViz for introducing and understanding yet unknown parts of the architecture. However, the participant admitted that it might be difficult to visualize all the information in an organized and clear manner when the amount of sent objects increases beyond a certain threshold.

I4 The interviewee wished for an option of the trace replayer to only show traces for marked communication lines. This would enable the developer to identify detailed information of a specific part of the communication within the software architecture by analyzing specific traces of the selected communication.

Q7 *In which way do you think could ExplorViz support you in your daily work as a software developer?*

I1 The participant stated that ExplorViz is more suited for architecture development and verification rather than for developing new functionalities of a software application. It should be faster to gather the needed information, which is necessary for further development, from the code rather than from ExplorViz.

I2 Similar to the first participant, the second interviewee was of the opinion that ExplorViz is probably not suited for the active development of new features inside a software application. Gathering necessary information inside the source code is much quicker. However, it was admitted that this could change if the developer is more familiar with and has more experience in using ExplorViz.

I3 The interviewee thought that ExplorViz has the potential to immensely support management tasks of the software development process as well as requirements engineering since it excels in giving a big picture overview of the architecture and its behavior.

I4 The final participant mentioned that ExplorViz could especially support the specification phase of a reworked functionality. Furthermore, the existing potential of ExplorViz of supporting the testing and debugging is stated. As a final benefit, the interviewee saw the opportunity of using ExplorViz for introduction and training purposes for newly employed software developers.

4.4.3 Discussion of the Evaluation Results

The results of the first question **Q1** show that every participant has in-depth knowledge about the *in|FOCUS* software architecture. They either have in the past or currently do actively implement parts of the functionality. Furthermore, it is to mention that the participant of the third interview **I3** is part of the development process as a software engineer. While possibly not having the same level of implementation details about certain modules as the others, this participant presents an opportunity for the evaluation process. This interviewee can provide a unique standpoint of view compared to the others. While still having enough experience with the target architecture in order to conduct its analysis, this participant provided an assessment of the usability of ExplorViz for the architecture planning process as a software engineer.

Next, three out of the four results of the interviews demonstrated different aspects of ExplorViz and the software architecture analysis. **I1** confirms that ExplorViz is able to identify unwanted behavior inside the software. While being the developer with the most experience with *in|FOCUS*, the first interviewee found information about possible yet unaware implementation errors. This partly confirms the hypothesis **H3** that ExplorViz can be successfully used for verifying developed functionalities of the analyzed application. **I3** shows another application of ExplorViz, the architecture discovery. This interviewee

utilized the visualization tool to gather missing in-depth information about certain parts of the architecture. Hence, this provides a case where the second established hypothesis **H2** is shown to be true. Last, **I4** reveals a problem which was also encountered during the architecture analysis part of this thesis. Even with the necessary knowledge and experience, the behavior of more complex use cases can be difficult to analyze inside a convoluted monolithic software architecture.

The results of the third and fourth question give several crucial insights into the ability of ExplorViz of supporting the discovery of microservice boundaries inside a large-scale architecture. While the participants of **I1**, **I3**, and **I4** were able to come up with a possible boundary for bounded contexts, **I2** had to discover such a boundary with the help of ExplorViz. This confirms that ExplorViz provides not only possibilities for the discovery of unknown parts of the architecture, as stated previously, but also supports the software modernization process in discovering bounded contexts. After the definition of boundaries, the participants had to verify their quality of being possible microservice candidates. This validation process is an important part of every development process. Here, every interviewee came to the conclusion that their first definition of boundary led to overlooked problems. They were able to come up with suggestions for improvement, however, every participant stated that these improvements possibly lead to other problems. In conclusion, it can be assumed that the definition of boundaries for microservices is highly complex and that even a plausible suggestion can imply numerous challenges that require further assessment.

When it comes to the ability to support the discovery and the verification of microservice boundaries, the participants agreed on the positive impact of ExplorViz in **Q5**. As previously expected, the prerequisite for that seems to be the in-depth knowledge of the software architecture. Generally speaking, the conducted interviews strengthen hypothesis **H1** that ExplorViz does seem to provide a positive impact on the discovery and verification of microservice boundaries.

The interviewees came up with compelling ideas for possible missing features of ExplorViz in **Q6**. As these features could be object for future work on the development of ExplorViz, Chapter 6 goes into further details in this matter.

For **Q7**, every participant saw ExplorViz not as a suitable software tool for supporting the daily implementation work of a software developer. However, it is explicitly stated that this might change when the user has more experience with this trace visualization tool. However, ExplorViz seems to prove itself very useful when it comes to software engineering tasks such as requirement engineering or for training purposes and the introduction to previously unknown or partly unknown architectures. Therefore, further investigations have to take place as future work in order to fully verify hypothesis **H3**. Exemplarily, future interviews with software developers with prior in-depth experience with ExplorViz could be conducted to provide further inside into this matter. Chapter 6 discusses this possibility in more detail.

4.5 Threats to Validity

The quality of the evaluation results could be compromised due to the following reasons. The prior experience with ExplorViz could greatly influence the quality of the results of the architecture analysis of the participants. It is possible that the lack of experience in combination with the time limit of 30 minutes of the conducted interviews made it not possible for the participants to be able the fully make use of the supporting features of ExplorViz. Furthermore, the experience with the software architecture of *in|FOCUS* as well as the software architecture development experience of the participants are two factors that could prove the results to be not valid. Discovering bounded contexts inside an existing architecture does not only require detailed insight into the application's domain but also in-depth knowledge of the majority of the application's architecture. Additionally, the limited number of four participants may not provide enough insight for drawing reasonable conclusions about the analyzed ability of ExplorViz. Due to technical limitations of the in-development version of ExplorViz which was used for these interviews, the number of probed packages had to be restricted and it was not possible to monitor the whole application at once. As it was the case for the main architecture analysis conducted in this thesis, high-impacting packages for the main service operations of the software system were identified which the help of static analysis. This subset of packages was then selected for probing prior to the interviews. Therefore, it is possible that some key packages were missing when dynamically analyzing the architecture. This could have either confused the participants or it could have made the discovery of certain microservices not possible.

Chapter 5

Related Work

5.1 Software Modernization Projects

5.1.1 Otto.de

Otto started to move away from a monolithic software architecture towards microservices [Hasselbring and Steinacker 2017]. The reasoning behind the company's decision was mainly non-functional. A modernized architecture would not only provide a more goal-oriented scalable software system as well as a more scalable development process, but also higher performing subsystems and better fault-tolerance throughout the system. Due to the immense complexity of the modernization process, they did not however go directly from a monolithic architecture to microservices. Instead, they chose so-called *verticals* as an intermediate step. These self-contained vertical systems restructure their single-application system along their business subdomain into initially four but eventually up to twelve verticals. Each vertical is a standalone application with its own separate frontend, backend, and data storage. Figure 5.1 gives an overview of the self-contained system architecture

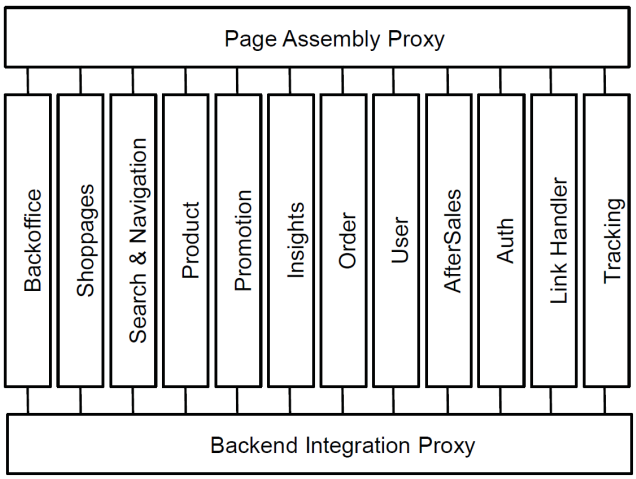


Figure 5.1. Decomposition of the monolith into verticals at otto.de [Hasselbring and Steinacker 2017].

structure. The different independent user interfaces of each vertical are assembled by the *Page Assembly Proxy* while the cross-cutting concerned services are deployed by the *Backend Integration Proxy*. Each vertical belongs to a single team in order to follow Conway's Law. Each team incorporated DevOps as well as CI/CD in their development process to stay agile and responsive to their development environment. The next step in the modernization process was to further divide the verticals into microservices. The definition of each microservice remains inside its domain boundaries and does not overlap with other domains. Figure 5.2 clarifies this decomposition within the domain boundaries, which are depicted as red lines in this figure. By keeping the general structure of the verticals, they combine the benefits of both self-contained systems and microservices. The verticals give the microservices their structure. In addition, the microservices ensure that each vertical does not devolve itself into monolithic applications.

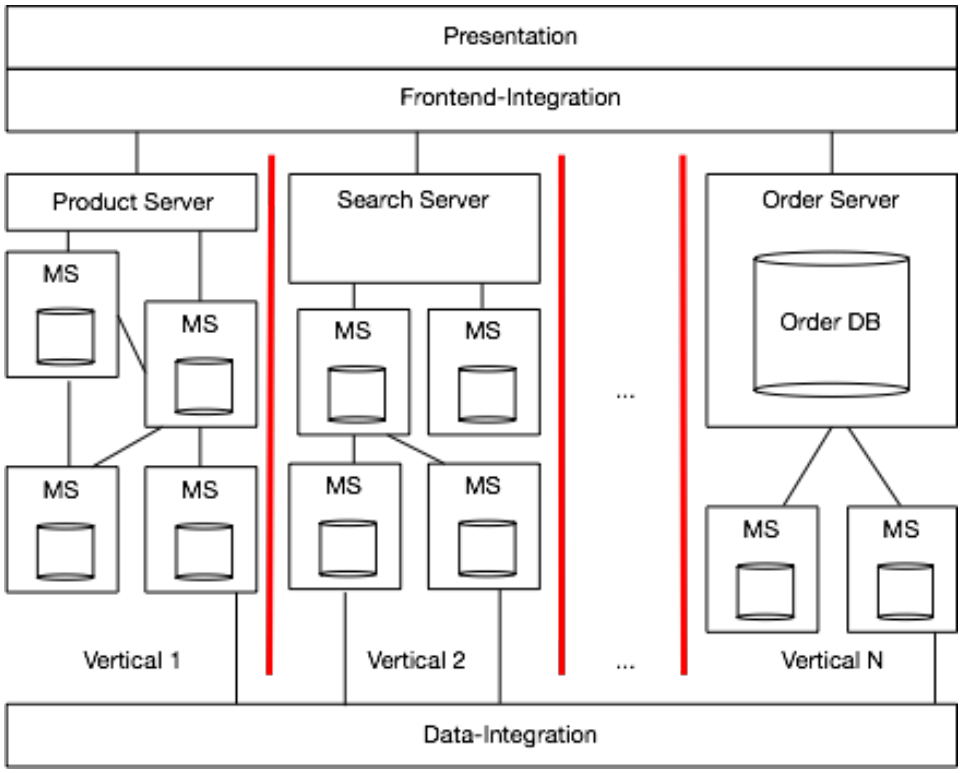


Figure 5.2. Further decomposition of verticals into microservices at otto.de [Steinacher 2014].

5.1.2 ExplorViz

The live trace visualization tool ExplorViz, which is used as one of the main analysis tools for this thesis, started off with a monolithic architecture. The legacy architecture utilizes the Google Web Toolkit (GWT) as a simple solution for developing Java code both frontend and backend in a single project. The frontend-related code is then compiled to JavaScript code for execution. GWT Remote Procedure Calls (RPC) enables the triggering of server-side actions or data transfer via HTTP. The goal of this chosen technology stack is to enable developers with limited experience, such as computer science students, to quickly understand the used technologies and actively contribute to the ExplorViz project.

The architectural modernization of the ExplorViz project was triggered by two drivers. First, by the development stop of GWT and secondly, by the increasing popularity of higher performant RESTful web services in comparison to the previously and widely used SOAP-based services (Simple Object Access Protocol) [Upadhyaya et al. 2011]. In order to ensure a flexible and scalable application, the monolithic ExplorViz architecture is modernized in a two-step procedure towards microservices. This process is illustrated in Figure 5.3. As a first step, the monolith, which can be seen in the top part of the presented figure, is divided into two independent self-contained microservices *Backend* and *Frontend*. The backend is implemented as a Java-based *Jersey*[1] web service. It provides a RESTful API for the client communication. The frontend utilizes the JavaScript framework *Ember*[2] for providing visualizations of software landscape models inside a web browser. Ember is executed inside a *Node.js*[3] environment. Additionally, the previously used custom-made monitoring component is replaced by the monitoring framework *Kieker*. Further information on Kieker can be found in Section 2.6.

The next step of the architecture modernization, depicted in the bottom part of Figure 5.3, further divides the backend into additional microservices. The frontend-backend communication takes place via a single endpoint, a *Nginx*[4] reverse proxy. This API gateway takes client requests and passes them to designated services of a server. Subsequently, the response of the server is then delivered back to the client. The communication between different microservices of the ExplorViz backend is handled by a *Apache Kafka*[5] message broker. This well-structured organization of the communication within the software architecture enforces lose coupling and therefore enables services to be developed, deployed, and executed independently. For a more detailed discussion about the modernization process as well as a in-depth introduction to the new microservice architecture, it is referred to Zirkelbach et al. [2019].

[1] `https://jersey.github.io/`, accessed 22.05.2019
[2] `https://emberjs.com/`, accessed 22.05.2019
[3] `https://nodejs.org/`, accessed 22.05.2019
[4] `https://www.nginx.com/`, accessed 22.05.2019
[5] `https://kafka.apache.org/`, accessed 22.05.2019

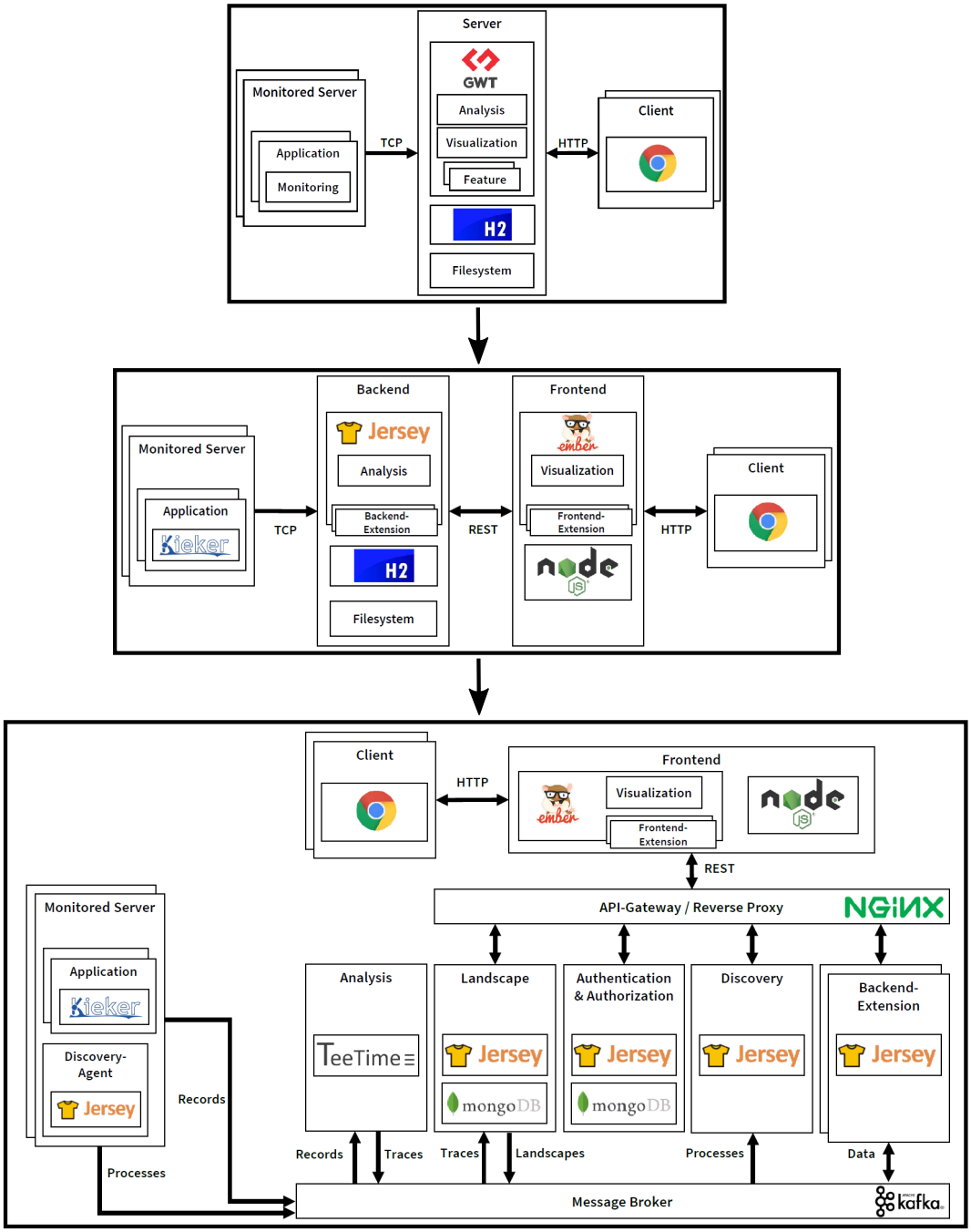


Figure 5.3. Overview of the architectural evolution of ExplorViz from a monolith (top) to a first modularization (middle) and finally to a microservice architecture (bottom) [Zirkelbach et al. 2019].

5.1.3 OceanTEA

The data analyzing software OceanTEA[6], developed by the Software Engineering Group of the Kiel University, supports scientists in interactively exploring and analyzing high-dimensional data sets. The different microservices of this architecture are structured in three verticals, which are marked in red in Figure 5.4. Each of the verticals group microservices together which are related. The OceanTEA Web Client communicates with the server-side part of OceanTEA via an API Gateway by using HTTP and REST calls. The API Gateway then calls the concerned microservices via REST calls. Microservices communicate with each other and exchange data, depicted as black arrows in Figure 5.4.

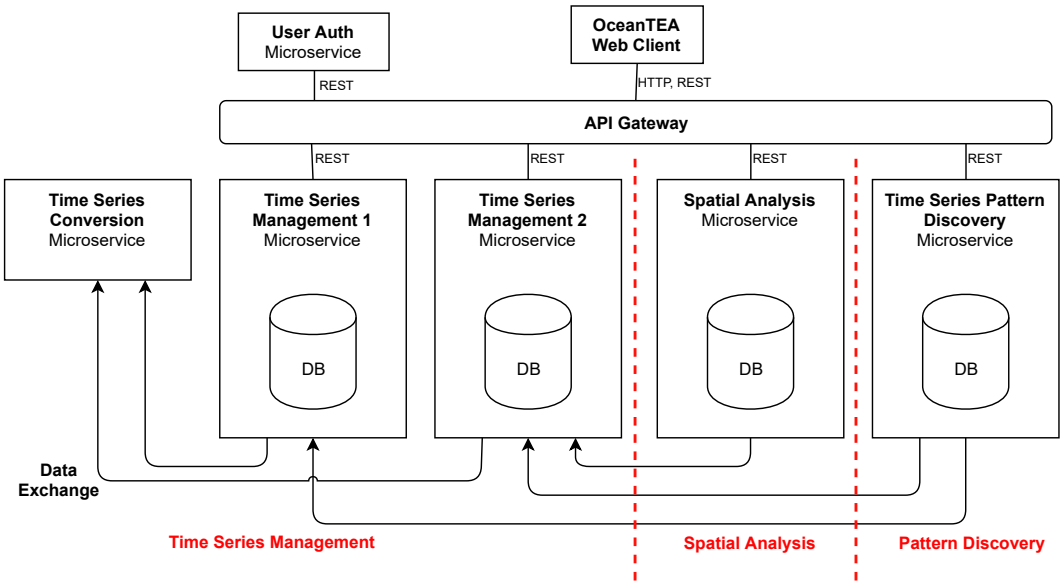


Figure 5.4. Overview of OceanTEA microservice architecture [Johanson et al. 2016].

5.1.4 GeRDI

The GeRDI project provides scientists with a sustainable Generic Research Data Infrastructure (GeRDI) in order to search, use and re-use external, multidisciplinary research data [Thomsen et al. 2018]. By combining concepts of domain-driven design and self-contained systems, the resulting architecture supports a flexible adjustment to changing requests and an easy integration of already existing software and external services such as cloud computing [de Sousa et al. 2018]. The application domain is clustered into eight distinct generic service domains which are realized as independent self-contained systems, as

[6] `https://github.com/cau-se/oceantea`, accessed 23.05.2019

illustrated in Figure 5.5. Each of them represents a required functionality of the concerned research cases and a step in the life span of research data. Each self-contained system contains its own business logic, data and, if required, its own UI. The self-contained systems communicate with each other via remote REST-interfaces, even though the inter-service communication should be reduced to a minimum. A backend integration layer deploys possible cross-cutting concerns such as a system monitoring infrastructure. Each separately implemented user interface of each self-contained system is deployed by the frontend integration layer.

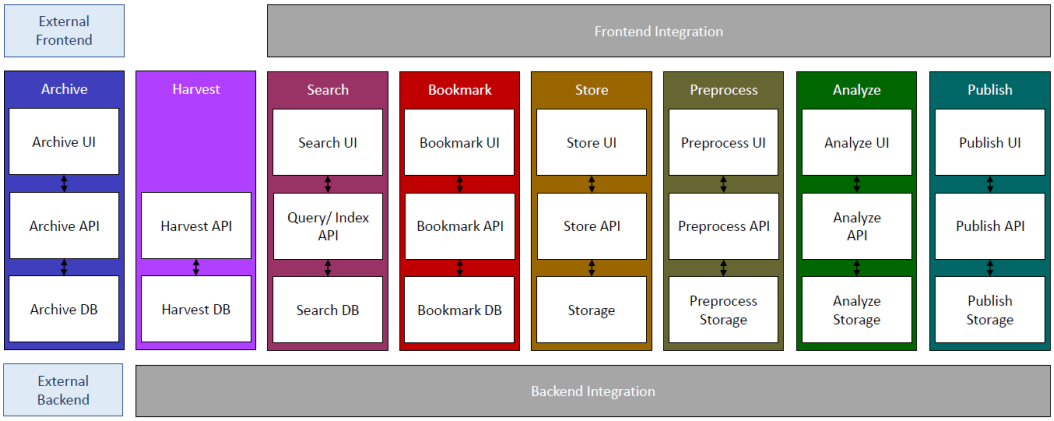


Figure 5.5. Overview of the GeRDI self-contained system architecture [de Sousa et al. 2018].

5.1.5 Galeria Kaufhof

In 2014, the German department store chain Galeria Kaufhof started modernizing their monolithic software architecture of their online web store towards a self-contained software architecture [Grotzke 2014; Kiessling 2015]. By dividing the business domain into five distinct subdomains, the boundaries of the self-contained systems were defined. Here, one domain can possibly contain multiple self-contained systems. In order to ensure loose coupling, each service owns its presentation layer, business logic layer, and persistence layer. Furthermore, the subsystems only communicate with each other via asynchronous REST calls when necessary. Besides the five self-contained systems *Explore*, *Search*, *Evaluate*, *Order*, and *Control*, the architecture also incorporates a *Foundation Services Layer* which deploys possible cross-cutting services. Furthermore, a *Frontend Ingertaion Layer* assembles the independent user interfaces of each service to a coherent website.

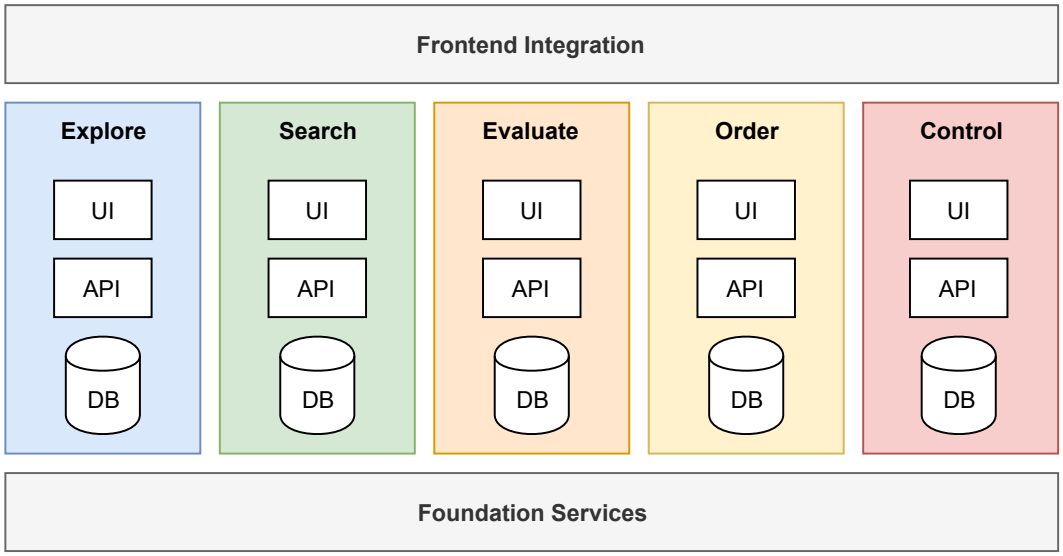


Figure 5.6. Overview of the Galeria Kaufhof self-contained system architecture [Grotzke 2014].

5.1.6 Groupon

Prior to the decision of Groupon, an online deal marketplace, the architecture of their software system consisted of multiple monoliths, each responsible for online marketplaces of different continents. Each monolith redundantly implemented the same functionality. Besides the increased development effort to serve both monoliths, the performance of the software system could not keep up with the rate of growth of the company [Geitgey 2013]. Hence, Groupon decided to completely rebuild their architecture to a microservice oriented architecture model.

Figure 5.7 shows the modernized architecture of Groupon. The application was reorganized into more than twenty separate web applications [Geitgey 2013]. A *Content Delivery Network* (CDN)[7] delivers different content to the customer, based on the customer's geographic location. The routing layer then forwards the user to the desired frontend of the application. The frontend layer communicates with the backend of the system via a designated API layer. The backend is divided into two domains, namely the *North America Backend* and the *European Backend*. Each of them includes multiple self-contained microservices. In the future, it is intended to merge the two backend domains into a single backend to further reduce redundancy throughout the system [Geitgey 2013].

[7] `https://www.webopedia.com/TERM/C/CDN.html`, accessed 23.05.2019

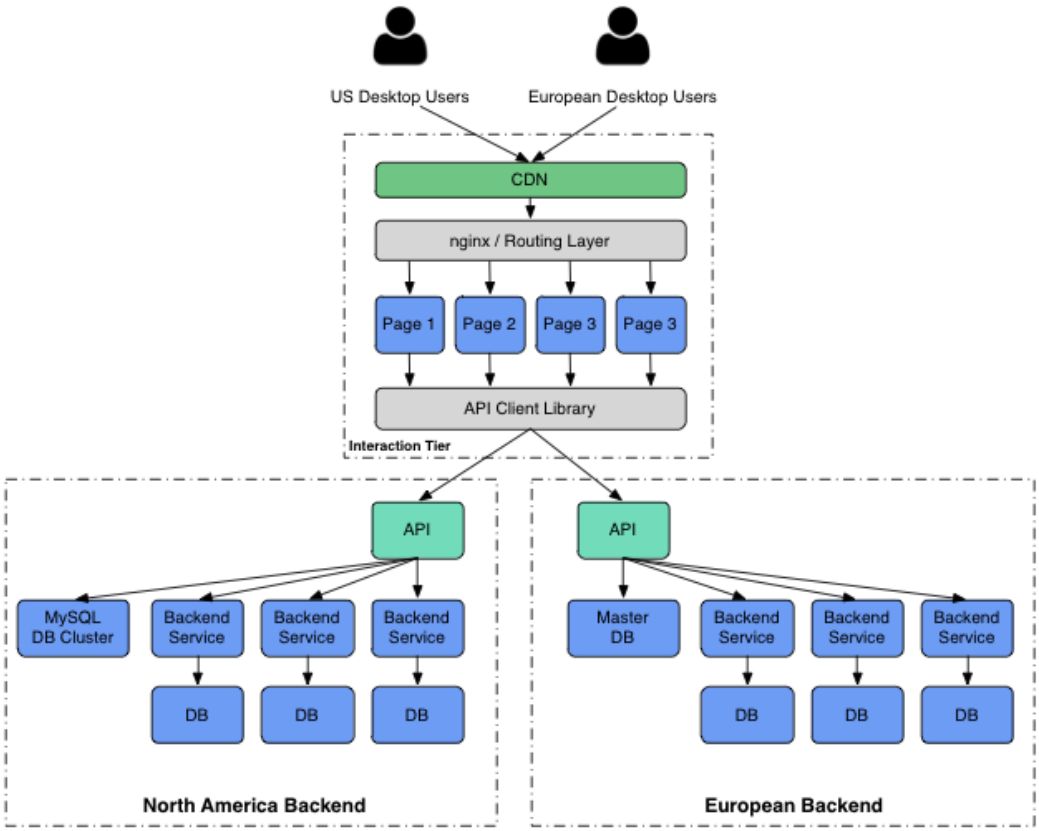


Figure 5.7. Overview of the Groupon microservice architecture [Geitgey 2013].

5.2 Software Modernization with the Help of Analysis Tools

5.2.1 Modernization of a Customer Management Application

A detailed software modernization process and best practices for the decomposition of the architecture into microservices are presented in the article of Knoche and Hasselbring [2018]. Furthermore, it reviews its implementation in a large-scale industrial modernization project. After defining a model of the application's domain in the initial phase of the modernization process, static analysis techniques with the help of custom made tools are employed in order to find communication entry points into this application. These entry points can for example be method calls or database queries which are invoked by outside applications. As it was stated in the article, this discovery step was crucial for successfully defining modular boundaries for future microservices due to the previous

unawareness of all possible entry points [Knoche and Hasselbring 2018]. However, they did not incorporate dynamic analysis in combination with static analysis. Therefore, this thesis aims at combining both analysis approaches for the modernization process of the large-scale industry software application. Furthermore, the goal is to receive further insights into the impact of the dynamic analysis tool ExplorViz.

5.2.2 Microservice Discovery for a Cargo Tracking Domain

Gysel et al. [2016] and Baresi et al. [2017] both perform a tool-supported domain analysis of a cargo tracking software application, an often used sample application which was introduced by Evans [2003]. Before discussing the analysis process and the results of both projects, this demo application is presented with the help of Figure 5.8 in order to gain the necessary understanding of the application and its domain. The figure shows the class diagram [Baresi et al. 2017] of a Java implementation[8] of this application. It provides the functionality to move a `Cargo` with a unique `trackingId` between two `Locations` via a `RouteSpecification`. When the `Cargo` is ready for shipment it gets associated with a `Itinerary` which is a list of `Legs`. Each `Leg` routes the delivery by selecting from existing `Voyages` and `CarrierMovements`. The progress of the `Cargo` on its `Itinerary` is traced by `HandlingEvents`. Any available general information about the delivery status or the estimated date of arrival is gathered by `Delivery`.

Gysel et al. [2016] utilizes the static software system and domain analysis tool *Service Cutter*[9] in order to discover a suitable decomposition into microservices within the software

[8] `https://cargo-tracker.gitbook.io/`, accessed 23.05.2019

[9] `https://github.com/ServiceCutter/ServiceCutter`, accessed 23.05.2019

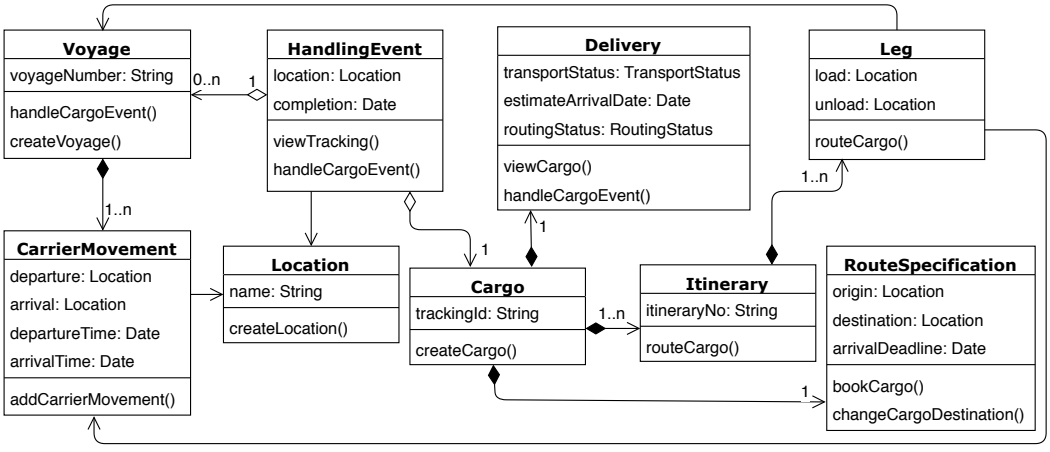


Figure 5.8. Class diagram of the cargo tracking application [Baresi et al. 2017].

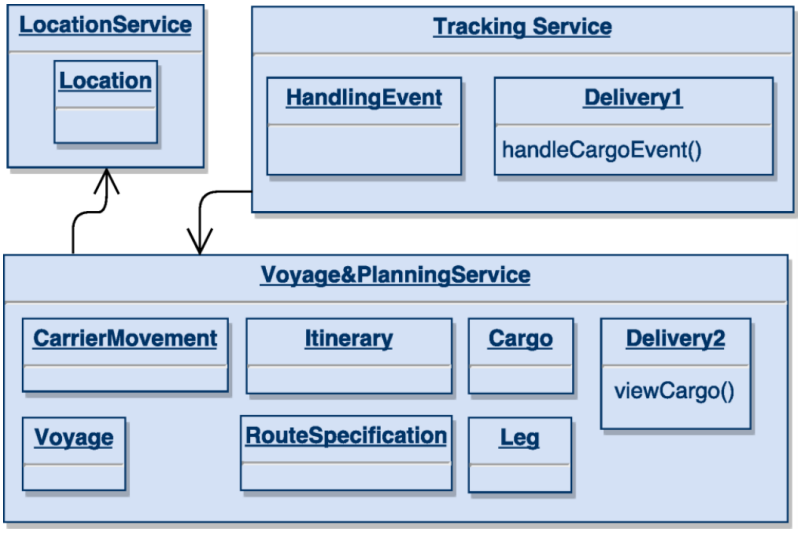


Figure 5.9. Decomposition of the cargo tracking domain by Service Cutter [Baresi et al. 2017].

architecture of the cargo tracking application. Service Cutter generates *candidate service cuts* from user-defined *System Specification Artifacts* (SSA). The candidate service cuts are chosen in a way that the internal structure of these candidates is highly cohesive and such that the services are loosely coupled to each other. The SSA, which are for this reason utilized, are data sets which contain information about coupling criteria. These criteria define significant architectural requirements and arguments why two distinct entities of a system should or should not be part of the same service. SSAs can be comprised of numerous data types, called *User Representations*, such as use cases or Entity-Relationship Models. A list of all possible data types can be found in the Service Cutter Wiki[10].

Baresi et al. [2017] aims to provide an automated solution for decomposing a given domain into candidate microservices with the help of *distributionally related words using Co-occurrences* (DISCO) [Kolb 2009], a pre-computed database of collections and distributionally similar words. The similarity between two words is measured based on their co-occurrences in large collections of text, such as Wikipedia, and is statistically analyzed. Hence, the goal is to convert the cargo tracking domain into a machine-readable format in order to find cohesive groups within. Therefore, Baresi et al. [2017] defines operations that the previously presented cargo tracking domain should offer to its user. These operations are specified with *Swagger*[11], a language-agnostic interface for RESTful APIs. Afterwards, the Swagger specifications are mapped onto the entries of a provided reference vocabulary, in this case Schema.org[12], a project for creating, maintaining, and promoting schemata for

[10] `https://github.com/ServiceCutter/ServiceCutter/wiki/User-Representations`, accessed 23.05.2019
[11] `https://swagger.io/`, accessed 23.05.2019
[12] `https://schema.org/`, accessed 23.05.2019

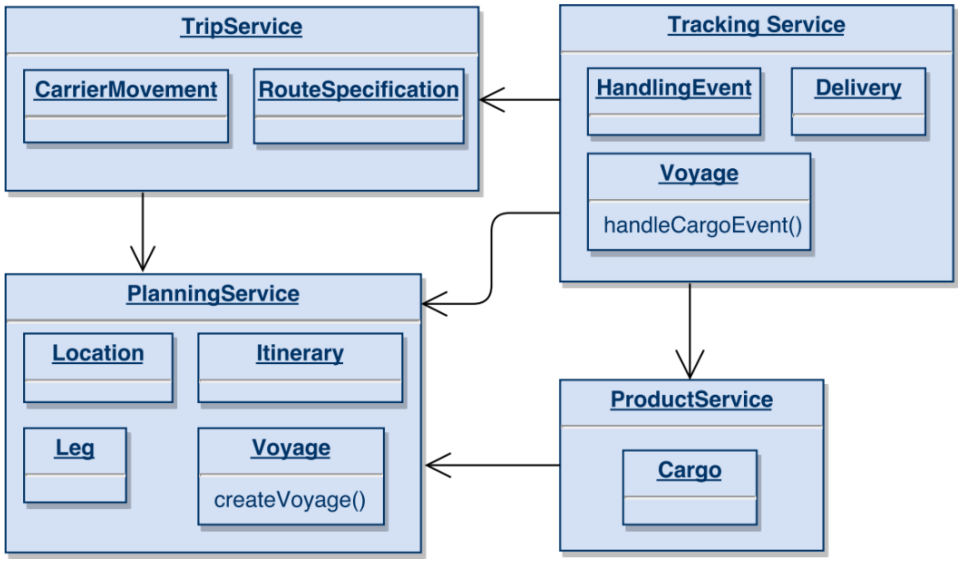


Figure 5.10. Decomposition of the cargo tracking domain by using the DISCO-based approach [Baresi et al. 2017].

structured data. The mapping process uses a fitness function which is based on DISCO entries. This term of the reference vocabulary, that the operation is mapped onto, should describe its *reference concept* which depicts the operation the most accurate. Next, the goal is to identify operations with the same reference concepts. At this point, it is assumed that operations which share the same reference concept are also similar to each other and should therefore be grouped in the same subdomain. The results of this algorithm suggest a decomposition of the given domain into multiple reference concepts. Each reference concept comprises the functionality of a microservice. For further details on the used algorithms of this decomposition approach, see Baresi et al. [2017].

Figure 5.11 shows the expected division of the cargo tracking domain into four services as defined manually by Gysel et al. [2016], namely *Voyage Service*, *Tracking Service*, *Location Service* and *Planning Service*. Each dotted line in this figure marks out a cohesive service of the software system. It is noted that the resources and the functionality of the `Delivery` class is divided between the *Tracking Service* and the *Planning Service*. When comparing the results of Figure 5.9 and Figure 5.10 to the expected results of Figure 5.11, it is apparent that both tool-assisted decomposition approaches do not generate the expected division. However, on the one hand, it can be argued that the previously defined expected results are not optimal. On the other hand, it is possible that the generated results are of the same quality as the expected decomposition and therefore provide a valid alternative. For a more in-depth look into the matter, see [Gysel et al. 2016] and [Baresi et al. 2017].

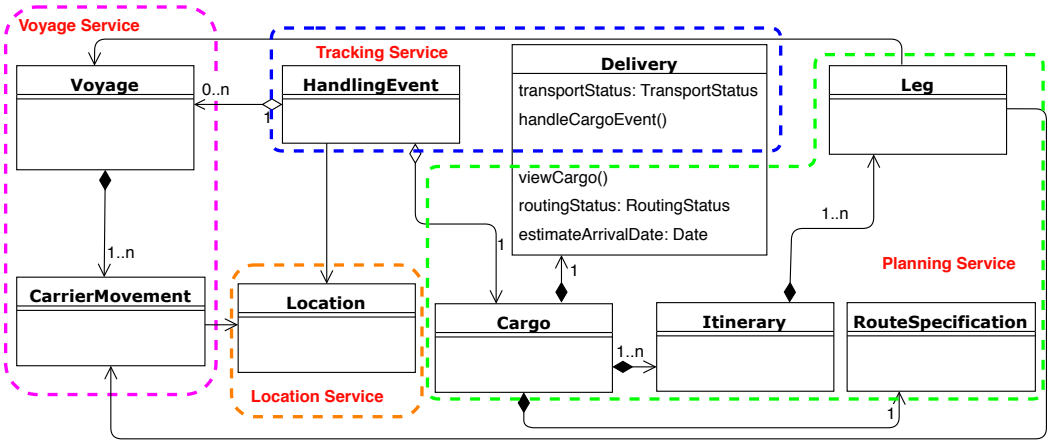


Figure 5.11. Expected decomposition of the cargo tracking domain [Gysel et al. 2016].

While both papers apply tool-assisted static analysis in order to discover microservices inside a software application, it can be argued that the analysis of this thesis operates on another level. The *in|FOCUS* software system is many times bigger and more complex than the cargo tracking application. Furthermore, the analysis process of this thesis combines static and dynamic analysis tools in order to combine the benefits of both. Especially the dynamical aspect of the architectural analysis, with regard to the discovery of microservices, is novel.

Chapter 6

Conclusions and Future Work

The final chapter of this thesis summarizes the analysis process whose goal was to modernize the *in|FOCUS* software architecture and concludes on its results. Furthermore, the gathered major insights of the evaluating qualitative interviews are presented. These evaluation results as well as the gathered experience throughout the modernization process itself give rise to future work and possible extensions.

6.1 Conclusion

Within the scope of this thesis, we presented and executed the modernization process of a large-scale monolithic software system towards a microservice architecture. For this work, static analysis technologies, which are commonly applied within these kinds of modernization projects, are supplemented by the dynamic analysis capabilities of the live trace visualization tool ExplorViz. Hence, we initially introduced main ideas of software architecture modernization and distinguished between the two general greenfield and brownfield approaches. Furthermore, the advantages of self-contained systems as an intermediate step before fully transitioning to microservices in the modernization process was discussed.

The modernization process was structured around three phases. We capitalized on the applicability of main domain-driven design concepts to the investigation of boundaries of loosely coupled and highly cohesive services within the software system. Therefore, we initially discovered service operations of the involved lottery domain actors. These service operations were then grouped into bounded contexts for providing a possible general division of the lottery domain. The second modernization phase based the static analysis of the *in|FOCUS* application on the previously defined bounded contexts. By employing static analysis techniques, we rediscovered the service operations and bounded contexts within the existing application architecture. The boundaries of the bounded contexts were adapted and redefined in order to improve their degree of loose coupling and cohesion. As a result, the application was divided into several distinct self-contained services. The third and final step was the verification of the static analysis results with the help of the live trace visualization tool ExplorViz. The invoked behavior of previously defined service operations was dynamically analyzed in order to evaluate the loose coupling qualities and cohesiveness of these services. Thereafter, a possible more fine-grained division of the

self-contained systems into microservices was presented.

Furthermore, we qualitatively assessed the ability of ExplorViz to support the modernization of a large-scale monolithic software system towards a microservices architecture by conducting guided interviews with the software developers of *in|FOCUS*. This gave us some key insights on the applicability of the current version of ExplorViz. Moreover, the detailed feedback of the development team lead to various noteworthy possibilities for extending the ExplorViz tool set.

The assessment confirmed that one of the central prerequisites for the effective usage of ExplorViz is a thorough prior knowledge of the entire monolithic architecture. What is more, the evaluation also demonstrated that finding a suitable division into self-contained systems or microservices is highly complex process which requires experience and time.

6.2 Future Work

Usability of ExplorViz for Microservice Software Modernization

Up to our knowledge, the qualitative assessment presented in this thesis is the first to investigate the impact of ExplorViz on the modernization of a monolithic application towards microservice architecture. Clearly, a more comprehensive evaluation involving more participants and allowing for a larger time frame would be desirable. In particular, repeating this kind of evaluation with participants who already have prior experience with the ExplorViz tool we deem as very interesting. For the sake of a better comparability, the software architecture used in such a future study should be similar to the *in|FOCUS* application in the means of complexity and scale.

Furthermore, the in-development version of ExplorViz, which was used throughout this thesis, limited the dynamical analysis due to technical issues. The number of *in|FOCUS* packages that could be monitored at once had to be reduced to a subset and consequently, it was not possible to observe the complete behavior of the monitored application. Therefore, we were unable to analyze the behavior of certain service operations, especially the ones which are far-reaching within the system architecture and utilizing a high number of components. However, exactly these service operations are possibly the most interesting and impactful use cases to analyze. These operations could affect a high number of self-contained systems or microservices. This could possibly result in unwanted dependencies and coupling between the involved services. Consequently, having the ability to analyze these high-impacting service operations in the future would certainly improve architecture analysis capabilities. Hence, we deem the extension of the ExplorViz tool to cover such use cases as crucial in order to support a boarder spectrum of application scenarios.

Supporting Features of ExplorViz

Analyzing the *in|FOCUS* architecture with ExplorViz as described in Section 3.5 as well as the evaluation process of this thesis gives rise to the following new functionalities and possible improvements of existing ExplorViz features.

ExplorViz already enables its users to revisit the past behavior of the monitored application within a certain time frame. The user can click through each available point in history. During the evaluating interviews, a playback function for history monitoring data was suggested. By automatically executing a previously defined sequence of points in history, the user could not only focus on analyzing the invoked behavior but could also revisit and label the behavior of specific executed service operations more easily. The ability to organize and revisit certain invoked behavior would improve recognizing opportunities for and threats to certain bounded contexts.

Furthermore, we consider that the introduction of a heat map, encoding the system work load within a certain color overlay, to ExplorViz tool set would improve its analytic capabilities. This heat map could enhance the currently available work load visualization of the system for the 2D landscape and 3D application model of ExplorViz. Bottleneck services as well as over-loaded services which are in need of possible vertical scaling would be more easily identified. This could prove to be highly useful for the microservice discovery as well as the maintenance of such microservice architectures.

Another suggestion which came up during the evaluating interviews was to enable more detailed communication lines between packages and classes within the model. Exemplarily, by showing the name of the sent objects directly on the communication lines, it could help to discover and to understand yet unfamiliar parts of the architecture. However, the challenge here would be to find a way to represent the possibly high amount of information in a clear and organized fashion.

Moreover, we suggest to extend the functionality of the trace replayer. Being able to filter the list of executed traces of the ExplorViz trace replayer could facilitate to focus on points of high interest during the architectural analysis. As an example, the user could selectively analyze certain parts of the system behavior by only presenting the traces of a specified sequence of communication lines.

Bibliography

[Admin 2018] I. Admin. *Greenfield vs. Brownfield Software Development. What's the Difference?* 2018. URL: http://www.indusa.com/articles/greenfield-vs-brownfield-software-development/. (Cited on page 20)

[Baresi et al. 2017] L. Baresi, M. Garriga, and A. Derenzis. Microservices Identification through Interface Analysis. In: Sept. 2017, pages 19–33. (Cited on pages 79–81)

[Bennett and Vaclav 2000] K. Bennett and R. Vaclav. Software maintenance and evolution: a roadmap. In: *Proceedings of the Conference on The Future of Software Engineering (ICSE 2000)*. ACM New York, June 2000, pages 73–87. (Cited on page 1)

[Bindick and Stoye 2018] S. Bindick and M. Stoye. *Von Monolithen zu Microservices: Domain-Driven Design als Schlüssel zum Erfolg*. 2018. URL: https://entwickler.de/online/development/microservices-domain-driven-design-579861186.html. (Cited on pages 20, 21, 47)

[Blanch 2017] R. Blanch. *Microservices: Strategies for Migration in a Brownfield Environment*. 2017. URL: https://medium.com/@rhettblanch_48135/microservices-strategies-for-migration-in-a-brownfield-environment-6c14335a8069. (Cited on pages 20, 21)

[Bonér 2017] J. Bonér. *Reactive Microservices. The Evolution of Microservices at Scale*. Sebastopol, CA: O'Reilly Media, 2017. (Cited on pages 6, 7, 20)

[Brown 2014] P. Brown. *Strategies for Integrating Bounded Contexts*. 2014. URL: https://culttt.com/2014/11/26/strategies-integrating-bounded-contexts/. (Cited on page 51)

[Conway 1968] M. E. Conway. How do committees invent. *Datamation* 14.4 (Mar. 1968), pages 28–31. (Cited on page 9)

[De Sousa et al. 2018] N. T. de Sousa, W. Hasselbring, T. Weber, and D. Kranzlmüller. Designing a Generic Research Data Infrastructure Architecture with Continuous Software Engineering. In: *Software Engineering Workshops 2018*. Volume Vol-2066. CEUR Workshop Proceedings. CEUR-WS.org, Mar. 2018, pages 85–88. (Cited on pages 75, 76)

[Evans 2003] E. J. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, Massachusetts: Addison-Wesley Professional, 2003. (Cited on pages 10, 79)

[Fittkau et al. 2017] F. Fittkau, A. Krause, and W. Hasselbring. Software landscape and application visualization for system comprehension with ExplorViz. *Information and Software Technology* 87 (July 2017), pages 259–277. (Cited on page 13)

[Fittkau et al. 2015] F. Fittkau, S. Roth, and W. Hasselbring. ExplorViz: Visual Runtime Behavior Analysis of Enterprise Application Landscapes. In: *23rd European Conference on Information Systems (ECIS 2015)*. May 2015. (Cited on page 13)

[Fittkau et al. 2013] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live Trace Visualization for Comprehending Large Software Landscapes: The ExplorViz Approach. In: *1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. Sept. 2013, pages 1–4. (Cited on page 13)

[Fowler 2015] M. Fowler. *MonolithFirst*. 2015. URL: https://martinfowler.com/bliki/MonolithFirst.html. (Cited on page 21)

[Fritzsch et al. 2018] J. Fritzsch, A. Zimmermann, and S. Wagner. From Monolith to Microservices: A Classification of Refactoring Approaches. *ArXiv e-prints* 3 (July 2018). (Cited on page 20)

[Gall et al. 2003] M. D. Gall, W. R. Borg, and J. P. Gall. *Educational Research. An Introduction*. Boston, MA: Allyn and Bacon, 2003. (Cited on page 56)

[Geitgey 2013] A. Geitgey. *I-Tier: Dismantling the Monolith*. 2013. URL: https://engineering.groupon.com/2013/misc/i-tier-dismantling-the-monoliths/. (Cited on pages 20, 77, 78)

[Gonchar 2018] G. Gonchar. *Data Consistency in Microservices Architecture*. 2018. URL: https://dzone.com/articles/data-consistency-in-microservices-architecture. (Cited on page 49)

[Grotzke 2014] M. Grotzke. *Jump - Ein Technologie-Sprung bei Galeria Kaufhof*. 2014. URL: http://tech.kaufhof.io/general/2014/09/20/jump-ein-technologiesprung-bei-galeria-kaufhof.html. (Cited on pages 76, 77)

[Gysel et al. 2016] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann. Service Cutter: A Systematic Approach to Service Decomposition. In: *5th European Conference on Service-Oriented and Cloud Computing (ESOCC)*. Volume LNCS-9846. Service-Oriented and Cloud Computing. Vienna, Austria: Springer International Publishing, Sept. 2016, pages 185–200. (Cited on pages 79, 81, 82)

[Hasselbring and Steinacker 2017] W. Hasselbring and G. Steinacker. Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. Apr. 2017, pages 243–246. (Cited on pages 9, 21, 71)

[Johanson et al. 2016] A. Johanson, S. Flögel, C. Dullo, and W. Hasselbring. OceanTEA: Exploring Ocean-Derived Climate Data Using Microservices. In: *Proceedings of the Sixth International Workshop on Climate Informatics (CI 2016)*. NCAR Technical Note NCAR/TN. Sept. 2016, pages 25–28. (Cited on page 75)

[Kalske et al. 2018] M. Kalske, N. Mäkitalo, and T. Mikkonen. Challenges when moving from Monolith to Microservice Architecture. In: *Current Trends in Web Engineering*. Edited by I. Garrigós and M. Wimmer. Basel: Springer International Publishing, 2018, pages 32–47. (Cited on pages 1, 20)

[Kharenko 2015] A. Kharenko. *Monolithic vs. Microservices Architecture*. 2015. URL: https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59. (Cited on pages 6–9)

[Kiessling 2015] M. Kiessling. *Die Architektur der Galeria.de Plattform im Kontext der Produktentwicklungsorganisation*. 2015. URL: http://tech.kaufhof.io/general/2015/12/15/architektur-und-organisation-im-galeria-de-produktmanagement. (Cited on page 76)

[Knoche and Hasselbring 2018] H. Knoche and W. Hasselbring. Using Microservices for Legacy Software Modernization. *IEEE Software* 35.3 (Apr. 2018), pages 44–49. (Cited on pages 2, 19, 20, 22, 78, 79)

[Kolb 2009] P. Kolb. Experiments on the difference between semantic similarity and relatedness. *NODALIDA 2009 Conference Proceedings* 4 (Jan. 2009). (Cited on page 80)

[Lewis and Fowler 2014] J. Lewis and M. Fowler. *Microservices: a definition of this new architectural term*. 2014. URL: https://martinfowler.com/articles/microservices.html. (Cited on pages 7–9)

[Martincevic 2016] N. Martincevic. *DDD: Context is King – Kein Context, keine Microservices*. Nov. 2016. URL: https://www.informatik-aktuell.de/entwicklung/methoden/ddd-context-is-king-kein-context-keine-microservices.html. (Cited on pages 20, 21)

[Newman 2015a] S. Newman. *Building microservices. Disigning fine-grained systems*. Sebastopol, CA: O'Reilly Media, 2015. (Cited on pages 1, 2, 7–9, 48, 50, 51, 53)

[Newman 2015b] S. Newman. *Microservices For Greenfield?* 2015. URL: https://samnewman.io/blog/2015/04/07/microservices-for-greenfield/. (Cited on page 20)

[Richards 2016] M. Richards. *Microservices vs. Service-Oriented Architecture*. O'Reilly Media, Apr. 2016. (Cited on page 9)

[Richardson 2014] C. Richardson. *Microservices: Decomposing Applications for Deployability and Scalability*. 2014. URL: https://www.infoq.com/articles/microservices-intro/. (Cited on pages 5, 20)

[Richardson 2018] C. Richardson. *Pattern: API Gateway / Backends for Frontends*. 2018. URL: https://microservices.io/patterns/apigateway.html. (Cited on page 51)

[Sneed and Seidl 2013] H. M. Sneed and R. Seidl. *Softwareevolution. Erhaltung und Fortschreibung bestehender Softwaresysteme*. Heidelberg, Germany: dpunkt.verlag, 2013. (Cited on pages 7, 19)

[Steinacher 2014] G. Steinacher. *Scaling with microservices and vertical decomposition*. 2014. URL: https://dev.otto.de/2014/07/29/scaling-with-microservices-and-vertical-decomposition/. (Cited on page 72)

[Steinacker 2015] G. Steinacker. *Von Monolithen und Microservices*. 2015. URL: https://www.informatik-aktuell.de/entwicklung/methoden/von-monolithen-und-microservices.html. (Cited on page 51)

[Thomsen et al. 2018] I. Thomsen, W. Haselbring, J. Schmidt, and M. Quaas. Integrated search and analysis of multidisciplinary marine data with gerdi. *International Conference on Marine Data and Information Systems* (2018). (Cited on page 75)

[Thönes 2015] J. Thönes. Microservices. *IEEE Software* 32.1 (Feb. 2015), pages 116–116. (Cited on page 1)

[Upadhyaya et al. 2011] B. Upadhyaya, Y. Zou, H. Xiao, J. Ng, and A. Lau. Migration of SOAP-based services to RESTful services. *13th IEEE International Symposium on Web Systems Evolution (WSE)* (2011). (Cited on page 73)

[Van Hoorn et al. 2012] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In: *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM, Apr. 2012, pages 247–248. (Cited on page 13)

[Wolff 2017] E. Wolff. *Self Contained Systems (SCS): Microservices Done Right*. 2017. URL: https://www.infoq.com/articles/scs-microservices-done-right/. (Cited on page 10)

[Woods 2016] E. Woods. Software architecture in a changing world. *IEEE Software* 33.6 (Nov. 2016), pages 94–97. (Cited on page 1)

[Zirkelbach et al. 2019] C. Zirkelbach, A. Krause, and W. Hasselbring. Modularization of Research Software for Collaborative Open Source Development. In: 2019. (Cited on pages 14, 15, 73, 74)

Appendix

A Lottery Domain Service Operations

Table 1. List of use cases of a lottery application

Customer login	Customer logout
Forgot password	Create promotion
Check OASIS entry	Check banking information
Edit promotions	Receive lottery promotions
Import lottery prizes	See product list
Transfer money to bank account	Transfer money to online wallet
Import lottery drawing dates	Block user account
Create new customer	Create new user
Edit user rights	Create new role
Admin portal login/logout	Deregister user
Check winnings	Request SCHUFA information
Change cash-out limit	Search promotions
Export customer balance sheet	Cancel game order
Search order	Determine winnings
Create account statement	Add credit card
Admin portal login	Check winners
Check customer identity	Invoke payment method
Search customer account	Edit debit lock
Cancel subscriptions	Edit customer card
Assign role	Compile master data set
Edit newsletters	Search customer card
Buy new customer card	Credit material prices
Receive lottery promotions	Fill out lottery ticket
Choose lottery ticket	Submit lottery tickets
Gather customer data	Edit personal information
New subscription	Receive lottery drawing results
Cash out winnings	Notify winners
Cancel customer card	Specify target group
Verify personal information	Change gaming limits
Determine Winnings	Check gaming history
Process incoming transaction	Process outgoing transaction
Subscribe to newsletter	Unsubscribe to newsletter

B Mapping of Service Operations to Business Objects

Table 2. Mapping of service operations (left) to business objects (right).

Business Object	Service Operations
Customer Verification	Check OASIS entry Request SCHUFA information
Customer Account	Customer Login Forgot password Change cash-out limit Export customer balance sheet Search order Create account statement Receive lottery promotion Customer logout
Customer Card	Buy new customer card Edit customer card Cancel customer card Search customer card
Personal Information	Edit personal information Verify personal information
Subscription	New subscription Cancel subscription
Banking Information	Transfer money to bank account Check banking information Add credit card
Online Wallet	Transfer money to bank account Transfer money to online wallet
Gaming Limits	Change cash-out limit Change gaming limits
Game Catalog	Choose lottery ticket See product list
Lottery Ticket	Choose lottery ticket Cancel game order Fill out lottery ticket

Table 2 (continued).

Newsletter	Change promotions Create newsletters Create promotion Search promotions Recieve lottery promotions Subscribe to newsletter Unsubscribe to newsletter
Customer Behavior Analysis	Gather customer data
Player	Receive lottery drawing results Notify winners Check gaming history
Ticket Submission	Create game order Cancel game order Check order Submit lottery tickets
Payment Method	Transfer money to bank account Invoke payment method Transfer money to online wallet
Transaction	Transfer money to bank account Transfer money to online wallet Process incoming transaction Process outgoing transaction
Prize	Credit material prices Determine winnings Check winnings Receive lottery drawing results Import lottery prizes
Lottery Drawing	Import lottery drawing dates Receive lottery drawing results Check winnings Check winners Credit material prices Transfer winnings to customer

Table 2 (continued).

Administration	Create new customer account Edit user rights Search order Assign role Block user account Create new user Create new role Deregister user Edit debit lock
User Account	Admin portal login/logout Search order Gather customer data Create promotion See product list Compile master data set Assign role

C Mapping of *in*|*FOCUS* Packages to Service Operations

Table 3. Mapping of *in*|*FOCUS* packages to service operations.

Service Operation	Involved Packages
Create new customer	component.usermanagement component.services component.newsletter component.portal component.subledger component.eod component.common
Login customer	component.usermanagement component.services component.common component.subledger
Logout customer	component.usermanagement component.services component.common
Forgot password	component.usermanagement
Change customer information	component.usermanagement
Change banking information	component.externalservices component.usermanagement component.subledger
Verify personal customer data	component.usermanagement component.services
Check OASIS entry	component.externalservices component.usermanagement component.services
Request SCHUFA information	component.services component.externalservices component.usermanagement
Change gaming limits	component.services component.subledger component.usermanagement
Buy new customer card	component.customercard component.usermanagement component.externalservices

Table 3 (continued).

Change cash-out limits	component.services component.subledger component.usermanagement
Export customer balance sheet	component.services
Check customer identity	component.usermanagement component.services
Search customer account	component.usermanagement
Cancel customer card	component.usermanagement component.customercard
Import lottery drawing dates	component.gameprocessing
Receive lottery drawing results	component.gameprocessing
Notify winners	component.gameprocessing component.services
Check gaming history	component.gameprocessing
New game subscription	component.usermanagement component.gameprocessing component.tsubscription
Cancel game subscription	component.usermanagement component.gameprocessing component.tsubscription
Check winnings	component.gameprocessing
Import lottery prizes	component.zgw component.gameprocessing component.externalservices
Submit lottery tickets	component.gameprocessing
Search order	component.gameprocessing
Cancel game order	component.gameprocessing component.services
Invoke payment method	component.subledger component.externalservices
Process incoming transaction	component.subledger
Process outgoing transaction	component.subledger
Transfer money to online wallet	component.subledger
Cash out winnings	component.subledger component.usermanagement

Table 3 (continued).

Transfer winnings to customer	`component.gameprocessing`
	`component.zgw`
	`component.subledger`

Determine winnings	`component.gameprocessing` `component.prizeanalyzer`
Subscribe to newsletter	`component.newsletter` `component.usermanagement`
Unsubscribe to newsletter	`component.newsletter` `component.usermanagement`
Send newsletter to subscribers	`component.newsletter` `component.usermanagement`
Create new user	`component.usermanagement`
Create new role	`component.usermanagement`

Deregister User	`component.newsletter`
	`component.usermanagement`
	`component.services`
	`component.customercard`
	`component.subledger`

Edit user rights	`component.usermanagement`
Fill out lottery ticket	`component.gameprocessing`
Gather customer data	`component.reporting` `component.monitoring`
Assign user role	`component.usermanagement`