The present work was submitted to the Research Group Software Construction

of the Faculty of Mathematics, Computer Science, and Natural Sciences

Master Thesis

Towards a Catalog of Refactorings for Microservices

presented by

Vitalii Isaenko

Aachen, June 13, 2019

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Dipl.-Inform. Andreas Steffens

Statutory Declaration in Lieu of an Oath

The present translation is for your convenience only.
Only the German version is legally binding.

I hereby declare in lieu of an oath that I have completed the present Master's thesis entitled

Towards a Catalog of Refactorings for Microservices

independently and without illegitimate assistance from third parties. I have use no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Official Notification

Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whosoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 to 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

I have read and understood the above official notification.

Eidesstattliche Versicherung

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Masterarbeit mit dem Titel

Towards a Catalog of Refactorings for Microservices

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, June 13, 2019 (Vitalii Isaenko)

Belehrung

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicher ung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen.

Aachen, June 13, 2019 (Vitalii Isaenko)

Acknowledgment

I would like to thank Prof. Dr. rer. nat. Horst Lichter for the possibility to write my master thesis at the chair. I also thank him and Prof. Dr. rer. nat. Bernhard Rumpe for reviewing the thesis. I am particularly thankful to my supervisor, Dipl-Inform. Andreas Steffens for his input and continuous feedback.

I thank my parents for providing me with unfailing support throughout this thesis. This accomplishment would not have been possible without them.

Vitalii Isaenko

Abstract

Attracted by the competitive advantages of newly emerged architectural style many organizations have adopted microservices practices in recent years. While the style has valuable benefits it comes with several tradeoffs. The major problem in adopting the architectural style is high complexity on the architectural level that leads to severe problems with system's maintainability. The thesis presents a catalog of refactoring techniques that help to cope with degrading software paying off the technical debt. The evaluation of the results demonstrated validity of the chosen approach as well as applicability of the proposed refactoring techniques in industry. Chosen approach guarantees reliability of the techniques as they are based on the proven traditional refactorings. Overall, the presented catalog is usable in practice and the work brings value for research and industry due to evaluation results.

Contents

List of Tables

List of Figures

List of Source Codes

1. Introduction

Contents

Martin Fowler gave his opinion on refactoring for microservices in an interview: "This is a recent trend which does not necessarily have such a big impact. Refactoring affects monolithic applications as well as microservices. Microservice architectures have an impact in that it becomes more difficult to make changes beyond the boundaries of microservices. With monoliths, such changes can be made more straightforward. On the other hand, it is much easier to make transformations within a small service because of the smaller code base." [Mfi]. We completely agree with Martin Fowler and claim that the work has big impact. As he noticed, "it becomes more difficult to make changes beyond the boundaries of microservices" and that is exactly the gap we are trying to solve in the thesis. We do not aim to propose refactorings that affect only the service under refactoring but ones that work on the system level and solve architectural problems. It also means that most of such techniques involve more that one service.

Refactoring is a crucial concept that is adopted by developers and used heavily on a daily basis. Refactoring techniques are applied to improve quality of a system without affecting its external behaviour. There are different refactorings. Some of them work on different levels - code or design. Many of them consist of several trivial techniques. There are catalogs that describe existing refactorings. There are books written to collect, describe and organize them better.

Refactoring is an important and evolving research topic. There exist refactorings for monolithic systems. With dramatic boost of popularity of microservices architectural style there are also some new refactorings introduced that help to move from monolithic to microservices system. However, refactorings for microservices systems are still missing.

The goal of the work is to build a catalog of refactoring for microservices. We try to refine techniques specifically for this architectural style. We approach the problem starting with microservices smells. We want the techniques to be reliable, therefore, we base our work upon existing traditional refactoring catalogs with proven in practice techniques.

The catalog is inspired by existing catalogs collected by Fowler, Lippert and Roock and others. The described techniques are used in industry extensively. Great description of the techniques contribute their use. We aim to make the catalog usable in practice by developers. Therefore, in the thesis, we create a description template and stick to it to present all the techniques.

The work includes evaluation of the results. We publish a questionnaire for this purpose and collect opinions of practitioners, researchers and students.

1.1. Structure of the Thesis

There is much foundation knowledge required to discuss the topic. Therefore, chapter 2 introduces all the necessary base knowledge on microservices, technical debt, bad smells and refactoring. Chapter 3 defines the scope of the thesis and challenges to tackle. In the following chapter 4 we discuss work done on microservices smells by now. We also organize the identified smells into table that is used through the thesis. The next step is building a concept described in chapter 5. There, we define a process to follow to build a high quality catalog. We also define a description template. Afterwards, we propose and choose an approach of refactorings selection. As the last part of concept, we define a type of evaluation the results. In the chapter 6 we perform the planned in the concept selection. The main results of the thesis - refactoring techniques for microservices - are presented in the chapter 7. Each section of the chapter presents one refactoring technique following the defined template. In the chapter 8 we present a developed questionnaire as a method of evaluation and analysis of its results. The last chapter - chapter 9 concludes the work, underlying the main results achieved. It also contains a discussion of possible future work on the topic.

2. Background

Contents

The chapter introduces necessary information for the master thesis.

2.1. Microservices

Microservices or microservice architecture - is an architectural style that structures an application as a collection of small, autonomous services that work together [New15] [Wha].

The main feature of microservices system is componentization via services. In most of the cases when a high level component (library or package) is needed - a service is created instead. The services in the system communicate via network. Usually REST communication style is used for this purpose.

One more significant feature that differs applications adopted the style a lot from monolith applications is decentralized data management. Each of the service has its own data storage with data required to cover defined business responsibilities [Mic].

Services in a microservices system have certain characteristics [New15]:

- Highly maintainable and testable. All the services should have high quality. It influences the overall quality of the system.
- Loosely coupled. Each service should be considered as a separate component. To achieve replaceability of services - they should be loosely coupled.
- Autonomous, independently deployable. The more autonomous the services are - the more organizational and operational benefits will be achieved. In this case, teams are able to work in isolation without frequent interventions. The characteristic allows to reduce time for integration considerably.
- Organized around business capabilities. Single responsibility principle works on the services level - every service should have only one reason to change. In a microservices system the reason is the same as everywhere - change of business

requirements. Therefore, the services should be split based on business capabilities to increase their autonomy.

- Small, doing one thing well. While size of a service cannot be precisely defined - it is important to keep service from doing several things. It also correlates with single responsibility principle.

All the mentioned characteristics bring benefits for the system that is built using microservices architectural style. The main of them are the following [Ric19]:

- Scaling. Microservices systems are easier and cheaper to scale. Having business capabilities split clearly among different services there is a possibility to scale only those parts that need it.
- Resilience. Failing service in a microservices system does not mean the system's crash. There are many more opportunities for developers and architects to build solutions that can handle the failure of a service gracefully.
- Technology heterogeneity. Having several services that communicate with each other via network, there are no restrictions on used technologies for them internally. Therefore, development teams can pick the right tool for every task (business capability) rather than be restricted to use only one technologies stack for the whole application.
- Ease of deployment. Making changes to one service allows developers to deploy this only service immediately without large integration overhead. It also makes deployment less risky - if any problem occurs it can be quickly localized. Smaller changes that are isolated in one service also allow making a fast rollback.
- Composability. With microservices approach we have an opportunity of better reusability of the written functionality. Such a system can serve different clients of any platforms requiring only internet connection.
- Services replaceability. There can be several reasons to replace a service completely - drastically changed business requirements, legacy code base that is easier to throw away and rewrite from scratch than refactor or emergence of new better suitable or more powerful technologies. Having the application split among small services it is cheaper and easier to complete.
- Organizational alignment. Smaller teams working on smaller codebase tend to be more productive. In microservices system this benefit is present as the size of each service is always appropriate for one small team.

2.2. Technical Debt and Bad Smells

Bad smell is a crucial concept for the research. As refactoring helps to improve system's quality - it is necessary to identify its disadvantages upfront. The bad smells are indicators of these disadvantages.

There are different levels on which smells can appear. Smell can be seen on code, design and architecture level. They are all interesting for the research. Fowler gives definition of smells on code level: "a code smell is a surface indication that usually corresponds to a deeper problem in the system" [Fow99]. Suryanarayana, Samarthyam, and Sharma give the following definition of design level smells in their book *Refactoring for Software Design Smells: Managing Technical Debt*: "Design smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality" [SSS14]. In the article [Gar+09] authors define architectural smell as a "commonly used architectural decision that negatively impacts system lifecycle qualities"[Gar+09].

A term 'bad smell' is closely related to technical debt. "Technical debt is the debt that accrues when you knowingly or unknowingly make wrong or non-optimal design decisions" [SSS14]. There are different dimension of technical debt. The following are the most popular:

1. Code debt.
2. Design debt.
3. Architectural debt.
4. Test debt.
5. Documentation debt.

Technical debt appears in every project and for different reasons. The solution that is provided by developers is not always optimal. It can be due to lack of time, high domain complexity, delayed refactoring or lack of test suite. As some of the factors are unavoidable, project gains technical debt with time. There are several means to manage technical debt in the project. The first step is always the same - increasing awareness of technical debt including awareness of the concept, its different types, the impact of technical debt on a system and the main factors that contribute to technical debt. Next, it is important to prevent accumulation of technical debt. As it is not possible to prevent it completely due to the factors mentioned above, developers should be able to detect, analyze and repay technical debt form time to time. On every step there exist useful tools - for measuring metrics, for technical debt quantification and visualization and for refactoring. During the analysis, it is useful to create a plan to follow for debt repayment.

Technical debt means degrading quality. There are different qualify factors that can decrease with time. However, most of the qualify problems that are identified by smells are maintainability issues. There are understandability, modifiability, extensibility, reusability, testability quality factors that are degrading with time. In other cases, performance can be an issue that would drive refactoring process.

The first step to pay the debt and increase quality factors of the system is to find, identify and analyze bad smells. Afterwards, starting from the smells that reveal problems for the most important quality factors, refactoring on different levels is performed. It is required to know what exact parts of the system should be refactored and the smells are those indicators of problems that developers should focus on.

2.3. Refactoring

Performing refactoring is the most useful way of technical debt management. Refactoring (as a noun) is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. [Fow99]. Refactor (as a verb) - restructure software by applying a series of refactorings without changing its observable behavior [Fow99].

There are several reasons to refactor. First of all, it helps to improve the design of software [Mot]. The most dangerous kind of technical debt that accumulates with time is debt on design and architectural level. Bad design decisions and lack of comprehension of existing design prevent developers from working productively, delay development of new features and force to use workarounds that adds even more technical debt. Moreover, decisions made on design and architectural levels cannot be isolated because they form a framework of the whole application. Refactoring helps to clean up the source code that clarifies the software structure and makes it easier for developers to evolve [RFG05].

Another reason in favor of refactoring is that it makes software easier to understand. An important user of the source code is a developer. The ease with which developers can grasp the code intention defines how much time they spend on maintenance of existing and implementation of new functionality. Shipping fast is crucial for business and therefore development speed is an important variable.

Refactoring can also help to find bugs in software. To refactor a piece of code developer needs to understand it deeply, clarify its intention and also validate their assumptions.

Before beginning refactoring it is necessary to identify problems that are meant to be tackled. There is a set of common problems that are called bad smells. Bad smells help to find deeper problems in software. Knowing them helps to perform high quality refactoring because time spent on identifying problematic code fragments will be reduced and awareness of higher level problems will help to prioritize refactoring tasks and find a better solution that solves the root problem.

Since there are different levels of bad smells and technical debt - there are different levels of refactoring. There are techniques that help to tackle problems on code, design and architectural level. Usually, higher level techniques are based on lower level refactorings. Such techniques are called composed techniques.

Testing is closely related to refactoring. To perform safe refactoring it is required to have tests for components. While during refactoring it is necessary to preserve behaviour, testing is indeed highly valuable part of the process. Since refactoring is very often done in legacy systems (that usually implies systems without good test coverage) - characterization tests can help developers to cope with the problem. [Fea04]

2.4. Summary

This chapter introduces some necessary background information required for the rest of the thesis. We started with defining what is microservices architectural style, pointed characteristics of systems with microservices architecture and demonstrated main benefits of adopting the style. The next section presented the concepts of bad smell and technical debt. The terms were defined as well as relation between them. Different types of technical debt were discussed. The last section discussed refactoring. The main reasons to refactor software were discussed together with benefits it brings. We also discussed importance of testing for refactoring.

3. Problem Statement

Contents

While the first chapter was meant to introduce the thesis and in the second chapter necessary background information was provided, this chapter is meant to detail the problem that we try to solve.

Each research needs a well-defined goal and a description of a process to follow. In the thesis we try to build a catalog of refactoring techniques for systems with microservices architectural style.

However, the questions to be answered are not defined yet and, therefore, the definition of done for the research is not clear. In the following sections we define research questions as well as discuss challenges and scope of the thesis.

3.1. Research Questions

The thesis is dedicated to an important research gap - refactoring of microservices systems. We claim it due to popularity of the architectural style and high attention from researchers and practitioners to the topic of refactorings of such systems. While refactoring towards microservices is already discovered well, there are no accepted refactoring catalogs so far that could be used in industry for refactoring existing microservices systems. The overall goal is to answer the question - "What are adequate refactorings for microservices?". The question is complex and consists of several parts. Therefore, we define the following research questions to build a catalog:

- RQ1: - How to identify potential refactoring techniques?
- RQ2: - How to describe refactorings for microservices?
- RQ3: - What are potential refactorings?
- RQ4: - Which of the identified refactorings are valid?

The first questions is to be answered by providing a reliable approach of refactorings identification. The approach will determine the reliability of the results - identified refactorings. The answer for the second research question is necessary element of any catalog - description of the identified techniques. Proposed refactorings should be understandable by practitioners and researchers. While we will take into consideration existing description templates, they will most likely require adaptation. The architectural style has specific aspects that should be reflected in techniques description. An example would be prerequirements for technique application that are usually not present for traditional refactorings.

And answer for the third question is based on a chosen approach and has to lead to the initial catalog of refactorings. The last research question has to be answered to make sure the catalog is valid and applicable.

3.2. Challenges and Scope

Microservices architectural style is still new and not discovered in all details. One of the main open questions is the topic of the thesis - refactoring of systems with such an architecture. The topic is challenging and has different aspects to be considered. Therefore, in the section we define the challenges and scope of the thesis.

Refactoring is a complex process and besides techniques application there are many organizational issues present. We do not investigate the issues and work with techniques in isolation. Moreover, the process itself can be different from the one used for modification monoliths. We do not define the new process and leave it for future work.

Refactoring is closely related to software testing. Testing is one of the most important parts of the process that helps to make sure the behaviour of a system has not changed after techniques application. However, we do not discuss how testing process is changed for microservices systems while it is an extensive separate topic that requires thorough investigation.

For traditional refactorings one can observe presence of different supportive automation tools. Such tools are important in practice helping to avoid errors and increasing developers' productivity. Even though we believe that there will be need in such tools for refactoring microservices systems in the future, we do not propose their implementation and leave it out of the scope.

Restricting the scope helps us to concentrate on the defined research questions. However, on the way to answer them we will still face challenges. One of them is testing the refactoring techniques in industry. Most of the traditional techniques were derived in practice with time. They were verified by thousands of developers and refined by most valuable professionals in industry. Creating a catalog that has to be applicable without pretesting it on different projects involving many professionals is a big challenge.

Another challenge is to make general-purpose refactoring techniques. It is not that difficult to create a solution for a very specific case. However, it is much more difficult to propose universal techniques that can be used for any microservices systems independently from technologies used, existing architecture and other aspects. It is a crucial attribute for any refactoring technique.

Description of a refactoring should demonstrate how the system is affected by the technique application. For techniques that work on a design or architectural level it is important to discuss what qualities are affected and how they are affected. It is a challenging question because a practical answer to this requires a thorough analysis of an existing system where refactoring was applied before and after its application.

3.3. Summary

In the chapter we set the research questions to be answered. We also defined the challenges and scope of the research. We also discussed main tasks to do and open questions to be answered in the master thesis. The main challenges to tackle are refactorings selection, their description and then - evaluation.

4. Related Work

Contents

The chapter presents related work on the topic. The work is also used in the thesis later on.

4.1. Microservice Smells Catalog

While developing microservices, as for any other system, developers often get into technical debt. The main indicators of the debt are bad smells. Microservices systems also have such indicators. In their article “On the Definition of Microservice Bad Smells”, Taibi and Lenarduzzi call these smells "Microservice smells" [TL18]. They identified 11 microservices smells and presented them in the mentioned article. Afterwards, based on this and other researches, Bogner et al. published their paper “Towards a Collaborative Repository for the Documentation of Service-Based Antipatterns and Bad Smells” [Bog+19] together with online catalog [Onl]. Smells from these two main resources are presented in the table 4.1.

Catalog of Microservice Smells	
Microservice smell	Description
API Versioning [TL18]	APIs are not semantically versioned. A lack of semantically consistent versions of APIs (e.g., v1.1, 1.2, etc.) [TL18]
Cyclic Dependency [TL18]	A cyclic chain of calls between microservices exists. The existence of cycles of calls between microservices; e.g., A calls B, B calls C, and C calls back A. [TL18]
ESB Usage [TL18]	The microservices communicate via an enterprise service bus (ESB). An ESB is used for connecting microservices. An ESB adds complexities for registering and deregistering services on it. [TL18]
Hard-Coded Endpoints [TL18]	Hardcoded IP addresses and ports of the services between connected microservices exist. [TL18]

Microservice smell	Description
Inappropriate Service Intimacy [TL18]	The microservice keeps on connecting to private data from other services instead of dealing with its own data. A request for private data of other microservices or direct connection to other microservices' databases exists. [TL18]
Microservice Greedy [TL18]	Teams tend to create new microservices for each feature, even when they are not needed. Common examples are microservices created to serve only one or two static HTML pages. It leads to microservices with very limited functionalities. [TL18]
Not Having an API Gateway [TL18]	Microservices communicate directly with each other. In the worst case, the service consumers also communicate directly with each microservice, increasing the complexity of the system and decreasing its ease of maintenance. [TL18]
Shared Libraries [TL18]	Shared libraries between different microservices are used. [TL18]
Shared Persistency [TL18]	Different microservices access the same relational database. In the worst case, different services access the same entities of the same relational database. [TL18]
Too Many Standards [TL18]	Different development languages, protocols, frameworks, etc. are used. [TL18]
Wrong Cuts [TL18]	Microservices are split on the basis of technical layers (presentation, business, and data layers) instead of business capabilities. [TL18]
Timeout [TL18]	The service consumer cannot connect to the microservice. Mark Richards recommends using a time-out value for service responsiveness or sharing the availability and the unavailability of each service through a message bus, so as to avoid useless calls and potential time-outs due to service unresponsiveness. [TL18]
Mega-Service [TL18]	A service that is responsible for many functionalities and should be decomposed into separated microservices. [TL18]
Leak of Service Abstraction [TL18]	Designing service interfaces for generic purposes and not specifically for each service. [TL18]
Greed [TL18]	Services all belonging to the same team. [TL18]
Sloth [TL18]	Creating a distributed monolith due to the lack of independence of microservices. [TL18]
Envy [TL18]	The shared-single-domain fallacy. [TL18]
Pride [TL18]	Testing in the world of transience. [TL18]

Microservice smell	Description
Ambiguous Service [Onl]	A service that includes interface elements (e.g., port types, operations, and messages) with unclear names, i.e. names that are very short or long, include too general terms, or even show the improper use of verbs. [Onl]
Bottleneck Service [Onl]	A service that is being used by too many consumers and therefore becomes a bottleneck and single point of failure. [Onl]
Business Process Forever [Onl]	Business processes have been strictly defined and are now static and cannot be easily changed. [Onl]
Chatty Service [Onl]	A high number of operations is required to complete one abstraction. Such operations are typically attribute-level setters or getters. [Onl]
Connector Envy [Onl]	Services implement large amounts of low-level interaction-related functionality, e.g. for communication, coordination, conversation, or facilitation. These functionalities should be implemented by a connector instead. [Onl]
Low Cohesive Operations [Onl]	A service that provides many low cohesive operations that are not really related to each other. [Onl]
Scattered Parasitic Functionality [Onl]	Multiple services are responsible for the same concern and some of these services are also responsible for orthogonal concerns. [Onl]
Service Chain [Onl]	A chain of service calls fulfills common functionality. [Onl]

Table 4.1.: Catalog of Microservices Smells

Some of the presented in the catalog microservices smells have aliases. These aliases are presented in the appendix B.

The most common approach to manage technical debt is refactoring. The biggest problem caused by technical debt is degrading quality of the system. Refactoring helps to improve the software quality and pay off the technical debt.

The presented catalog will be used throughout the thesis. The microservice smells will be attached to the refactoring techniques proposed, so developers will be able to find an appropriate refactoring technique to manage technical debt based on identified bad smell.

4.2. Microservices Patterns and Practices

There are several design patterns identified for the systems with microservices architectural style. Many of them are collected in the book *Microservices Patterns* by Richardson [Ric19]. Not all of them are required to know in the thesis. Only the patterns that will be referenced throughout the catalog are described in the section. Knowing these patterns will help reader to benefit the most from the proposed catalog.

4.2.1. API Gateway

The are many problems with accessing system's services directly. It leads to the lack of encapsulation that misleads developers from changing the service decomposition and their APIs. It also leads to disadvantages for clients as they have to collect necessary data from responsible services making multiple requests for one use case.

An API Gateway is a service that is an entry point of a microservices-based application for external API clients. [Ric19]. The service is usually responsible for various functions that are listed below.

- Request routing. Key function of an API gateway. All the requests from external clients first meet the API Gateway. Then it routes the requests to appropriate services in the system.
- API composition. Encapsulates invocation of multiple API methods of different services of the system to enable external clients to get their data with single request.
- Protocol translation. API gateway might also let clients use the protocols different from those used in the system internally.
- Provide client-specific API. API gateway can help developers to provide each client with own API. This way there will be no problems of general solutions that should fit every client. It also increases usability of the systems for external clients.

Besides the mentioned main responsibilities typically present in an API gateway, the service often provides even more functionality. Among those additional responsibilities that are commonly implemented and also very useful in practice are the following:

- Authentication.
- Authorization.
- Rate limiting.
- Caching.
- Metrics collection.
- Logging.

The main benefit of using the pattern is that it helps to encapsulate internal structure of the microservices system. In this case, external clients always have only one service to communicate with. An API Gateway provides clients specific APIs that can reduce the number of requests and simplifies client code.

Among the drawbacks of the pattern is that there is one service more in the system that has to be developed and maintained. It can also happen that the API gateway becomes a development bottleneck in the system. Nevertheless, in most real world applications it is reasonable to follow the pattern.

4.2.2. API Versioning

When API of a system starts expanding beyond original intent - there is an option of exposing a new version of API. The practice to attach versions to APIs of an application is called API Versioning. Versioning helps to iterate faster when the needed changes are identified. [Res].

Usually, the point when it becomes necessary to make another version of API is a milestone of system development. It implies presence of many different changes of the system, new features or complex structural modifications. However, addition of new features that do not change existing calls does not require new API version. It happens as system evolves and part of its natural growth without radical modifications. Those changes that that absolutely require API to be up-versioned are called "breaking changes". Among breaking changes are the following:

- changes in the format of the response;
- removing part of API or features;
- renaming or removing data fields/changes of their types in response body or any other information restructuring in existing resource representation.

The most common approaches to API versioning are: [Bae]

- URI Versioning. The most common approach where version number is present in URI of the endpoint. It can be version number, date or any other meaningful identifier.
- Versioning using Custom Request Header. A custom header can be introduced to make request on different versions of API. Once clients decide to move to use of new API version - they change the header value.
- Versioning using Accept header. The same idea as in the previous approach, but using existing header.

5. Concept

Contents

The chapter presents the concept of the thesis. It demonstrates how we will answer the research questions. To answer the first research question we should define an approach for refactorings identification. It is done in the first section of the chapter. The second question - refactorings description - is answered in the second section. We also discuss an approach of validation of the refactorings in the last section.

5.1. Initial Refactorings Selection Process

To build a useful catalog of refactorings, each technique should solve a specific problem. The common practice to tackle software quality problems is to detect bad smells first. Having the problem signifier found, an appropriate refactoring technique should be applied. This process of tackling the quality issues leads us to the main idea of building a useful catalog - refactorings should be suggested for bad smells in the system.

However, to achieve the goal there are at least three different approaches to choose from. The first one makes us consider the existing bad smells first, adapt and abstract them to microservices and assess their relevance. The existing refactoring techniques for the smells then will be adapted for microservices on the next step to tackle the microservices smells. The second approach is to find already identified microservices smells and match existing bad smells to them that again will require us to adapt original refactoring techniques. The third approach would be to ignore the existing bad smells and consider only the microservices smells identified. For each found microservices smell an appropriate refactoring technique will be invented on the next step of the process.

As choosing a way of discovering microservices smells determines how to propose refactoring techniques, each approach has a name according to the first step. We explain and assess the three approaches in the section.

5.1.1. Abstract Bad Smells to Microservices Smells

The approach implies starting from described bad smells and abstracting them to find the applicability on, usually higher, microservices architecture style level. While most of the existing smells to start with are very concrete and applied on the code level only, they always signify a higher level design issue. That means that they can be abstracted to find the general idea of the problem that leads to decreased quality and then they can be adapted to microservices systems. That is the main concern of the proposed approach, to discover an appropriate abstraction that represents the same general idea bud does not go beyond it our of the boundaries. It is important not to overgeneralize and stay within the same design issue bounds, as later step will be to apply respective refactoring(s) to tackle the bad smell. Those smells will become our microservices smells. The refactoring technique will have to be adapted as well to match the new microservices smell. The adaptation is discussed in subsection 5.1.6.

Steps

The approach can be represented by the following steps:

1. Collect bad smells from defined reliable resources.
2. Take one bad smell from the bank.
3. Extract the general idea of the design issue the bad smell represents.
4. Find a characteristic of microservices that indicates the same design issue as the original bad smell.
5. Adapt the bad smell to microservices smell based on the identified design issue.
6. Adapt the bad smell refactoring techniques to microservices refactorings.

Research Data

There exist catalogs and individually described bad smells. They form the basis for the approach. The most famous catalog with widely known code smells is collected by Fowler in [Fow99]. It has 22 code smells described. There is a catalog by Lippert and Roock presented in a book *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully* [LR06]. Lippert and Roock describes smells that can be faced on architectural level and proposes refactorings to solve them. Book *Refactoring for Software Design Smells: Managing Technical Debt* is dedicated to design smells [SSS14]. Authors divide the smells into categories and also propose some existing techniques to tackle them.

5.1.2. Match Bad Smells to Microservices Smells

The proposed approach suggests to work with existing bad microservices smells in parallel to identify matches. To find a match, we could use components matching or try to extract the general design issue the smells represent as in the first approach. After the matching is identified, the refactoring techniques for the code smell should be adapted to refactorings for microservices. The main challenges of the approach are the matching procedure and further refactorings adaptation.

Another important aspect of the approach is to reduce the set of microservices smells to consider only the relevant ones. Some microservices smells currently identified shift the problem space towards organizational aspects. As these aspects are beyond the research scope defined in 3.2 and we consider only software bad smells, there is no matching between the respective microservices and bad smells. Such preprocessing can help to save time and concentrate better on potential candidates for matching.

Steps

The described approach requires the following steps to be done:

1. Collect bad smells from defined reliable resources.
2. Collect microservices smells from defined reliable resources.
3. Reduce the set of microservice smells to relevant onces.
4. Match the bad smells to microservices smells.
5. Adapt the bad smells refactorings to microservices refactorings.

Research Data

There is an article on microservices smells [TL18] by Taibi and Lenarduzzi where they identified and described a set of 11 microservices smells. This catalog could be used for matching. There is also work from Garcia et al. where some of the microservices smells mentioned [Gar+09]. Another interesting fresh article is “Towards a Collaborative Repository for the Documentation of Service-Based Antipatterns and Bad Smells” where smells for SOA-based systems are collected with separation between traditional SOA architectural smells and microservices smells [Bog+19]. For the matching resources with traditional bad smells that were discussed for Abstract Bad Smells to Microservices Smells approach 5.1.1 can be used.

5.1.3. Use Identified Microservices Smells

The last proposed approach does not require much work with smells. It shifts the most workload to refactorings exploration. We start with identified and described microservices smells and on a latter step try to invent a refactoring technique for each of them. It requires a method of inventing the refactoring technique for a bad smell instead of a guideline for adaptation existing ones.

The same concern as for the previous approach still applies here - we should reduce the set of microservices smells to relevant ones, as many identified smells represent an organizational or any other problem that should not be solved by refactoring of a system.

Steps

The approach has the following steps to follow:

1. Collect microservices smells from defined reliable resources.
2. Reduce the set of microservice smells to relevant onces.
3. Invent refactoring techniques for identified microservices smells.

Research Data

For the approach we can use the same articles with identified microservices smells as for Match Bad Smells to Microservices Smells approach 5.1.2. Resources for traditional bad smells are the same as presented for the approach Abstract Bad Smells to Microservices Smells 5.1.1.

5.1.4. Approaches Assessment and Choice

All the three approaches have one important aspect in common. They allow us to find solutions (refactorings) for specific problems (bad smells), instead of, for instance, adapting refactoring techniques for a non-existent problem. However, each of the approach has advantages and disadvantages and analysis will help us to choose the right approach.

The Abstract Bad Smells to Microservices Smells approach requires the most work with existing code smells and their refactorings. It includes smells' abstraction and refactorings' adaptation. It has many reliable resources, but there is not that much relevance to microservices systems, while most of the refactorings are very granular - working on the code level and other smells were discovered only in monoliths.

The Match Bad Smells to Microservices Smells approach helps to solve the problem and brings more relevance to microservices system. However, it requires smells matching and adaptation of refactorings that are two big challenges to tackle. Nevertheless, it has the most resources to work with. It is crucial for any research to have a sound basis.

In opposite, the Find Refactorings for Microservices Smells approach has the least information to base our work on. Not using the existing refactoring techniques will reduce the reliability of the research. It also means that the work to do will be shifted from processing existing to building completely new knowledge.

Based on the assessment, it is decided to choose the Match Bad Smells to Microservices Smells approach.

Thus, we have the final process established that is demonstrated on the diagram 5.1. A solid line shows the goal, our main challenge - to find refactorings for microservices smells.

The dashed lines show our actual flow to follow. The process starts with matching bad smells to microservices smells. Then we discover existing refactorings for bad smells that have the matching with microservices smells. And finally, we adapt these refactorings for microservices smells. Those steps will lead us to the main goal - we will have refactoring techniques for microservices smells.

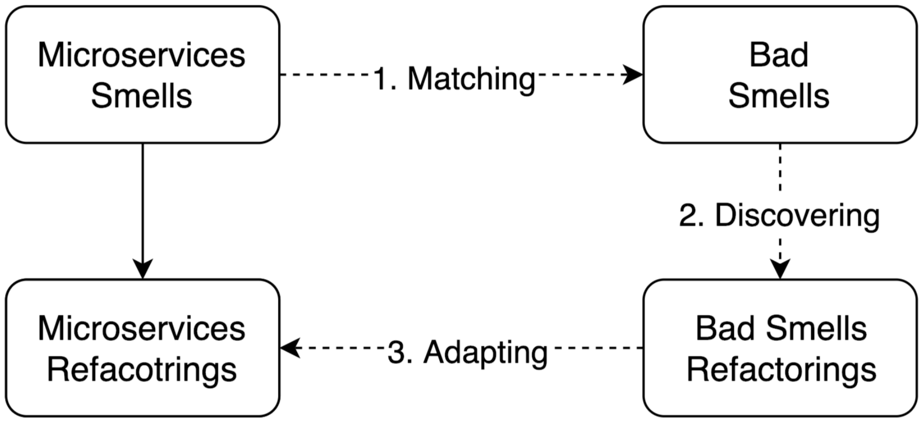


Figure 5.1.: Refactorings Selection Process

As we can see from the diagram and its description above, two steps out of three are quite challenging. The matching process requires guidelines to follow as there are no established rules for such matching exist. Discovering code refactorings for bad smells is straightforward - we will base our work upon existing resources on refactoring that are mentioned in section 2.3 where refactorings already presented as solutions to the bad smells. The third step is also unique in our work. The adaptation guidelines should be established to follow, moving from code to microservices refactorings. The sections below will fill the gaps in the process.

5.1.5. Smells Matching

To use the previously chosen approach Match Bad Smells to Microservices Smells in initial refactorings selection process, we have to define a way of matching the smells. It is important to document the process to be followed in this research as well as for future work in this direction.

The main part of matching is identifying elements used in smells description that will be considered equal. For example, a class in a code smell description could be interchanged with service in microservices smell. To find this, we should consider the common properties between used elements in existing bad smells description and elements in microservices smells description. The chosen way is to look at the elements as components.

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition" [Szy02]. In his book, Szyperski defines the following characteristic properties of a component:

- is a unit of independent deployment;
- is a unit of third-party composition;
- has no (externally) observable state.

While we have bad smells identified on different levels (code, design, architecture) we should consider these specificities. Lau and Di Cola in their book [LD17] define three main component models:

1. where components are objects;
2. where components are architectural units;
3. where components are encapsulated components.

Following this distinction, we can determine that code smells discuss elements of first category where some of them (such as methods) are not even under this category. Design smells also consider elements of the first category, while architectural smells describe problems related to architectural units - the components of second category. Microservices smells usually take services and their relation into consideration, therefore we deal with components from the third category.

Further in the section mappings between elements of different categories are provided with argumentation. It will help us to match the smells and the refactorings on a latter step.

First Component Model Matching

The relation between classes and services as the most important components of the first and the third category is the basis for further mapping. They share the same properties, having internal implementation, public interface and incoming and outgoing dependencies. They both can accept and send messages to other classes/services and possess other characteristics of components.

The second match that was identified is between public methods and services. The foundation for this matching is the same as for class-service. We cannot say that every method can be viewed as a component following the formal definition, but there exist methods that have similar properties with services. They should be public so they represent a part of interface of a class where they belong to say that there are clients as in case of services. They can use other methods, private or public of other classes that represent dependencies of service. They might also accept parameters that is similar to passing information to service in an abstract way. Methods can also react to events that is done by subscription as well as for services. All the common characteristics give us a basis and a good reason to try to match public methods to services aiming to find matches between existing bad smells and microservices smells.

The next match is pretty straightforward that is between public method and API method of service. They share the most common characteristics.

Another possibly useful observation is a matching between field of class and a piece of data in a persistance layer of service. This match could be useful in case of moving data that class or microservice is responsible for while refactoring to tackle a bad smell. They both represent a piece of data that belong to a class or a service. The only aspect that is important for us to define is the matching between the class and service that we already did above. The rest distinction falls down to the implementation details.

The last proposed non-trivial math is parameter to REST message field. There is no many properties to extract from these two, except that they both represent a piece of data and accepted from client. However, there are aspects that are common about the context of their use. In both cases the represented information is used by API method of service or method in a class. That is why we care about matching between public method and service that we defined before.

After analysis, the following matches are left considered as the most useful ones in our context:

- Class to service.
- Public method to service.
- Public method to API method.
- Field to record in persistance of service.
- Parameters of public method to REST Message body.

Elements that represent the same software entity (classes in monolith system and classes in microservices system, for instance) are possible to use as well during the smells matching. It does not require any analysis to do, though. However, they are not considered very useful, as most of the microservices smells concentrate on a system and architectural level problems rather than problems in each service.

Second Component Model Matching

The next step is to analyze architectural and microservices smells. There are considerably more similarities between architectural units and the elements in microservices that are considered to be encapsulated components.

The first identified match is package to service. Packages as architectural units each representing a part of a larger system that provides certain functionality. Each service in microservices system falls under the same definition with more restrictions.

The second match that can help in identifying similar architectural and microservices smells is a subsystem. We can match a subsystem to a separate service that basically is a subsystem in a microservices system. Moreover, as some piece of functionality in a system might be split into several microservices, we can match a set of microservices to a subsystem in monoliths.

After analysis, the following matches are proposed to be used:

- Package to service.
- Subsystem to service.
- Subsystem to a set of services.

Other units used in architectural smells but not discussed in the matching to elements in microservices systems can also be used but on the same level of abstraction. Layer is one example - we can find layers in monolith systems as well as in each service of a microservices system. Such matching is not considered useful for the same reason as described for first component matching above.

The proposed matching comes from the code, design, architectural and microservices smells analysis. In the first step the most obvious matches were chosen based on common sense and experience. After that, each of the software elements in each match was analysed from a component-perspective or based on previously defined match. Viewing the elements as components brought the most benefits in identifying commonalities.

However, not all the matches are possible to identify only based on the used components. Sometimes it is necessary to discover the general issue that the bad smells represents. Particularly in the case of smells that are driven by software design principles violation.

Revealing the main idea for both existing bad smells and microservices smell will help us to identify a match. To discover the general issue a bad smell represents, the 'problem' section of it should be examined. Such section usually presents discussion of design issues bad smells leads to. It is more creative task than matching by components. Most of the time the communication and connection between the components that leads to a problem should be examined. Concrete matches derivation will be discussed further in the thesis.

5.1.6. Bad Smells Refactorings Adaptation

The final step in the above defined process of initial refactorings selection is to adapt bad smells refactorings to microservices refactorings. Those adapted versions will be evaluated afterwards to decide on their applicability.

The major part of the initial adaptation will rely on the previously defined smells' elements matching. So the components correlation will be taken into consideration as the first step of transformation.

To derive motivation of the refactoring, the matched microservices smell will be explored in details. Parts of description that reveal problems that the bad smell can cause are the most useful to set the motivation.

Mechanics to follow will be derived from the original technique so that components will be interchanged with those from matching and the order will stay the same.

Nevertheless, initial adaptation results can differ from the final result. Evaluation can influence the part of refactoring description that discusses impact of the techniques application and other sections. Thus, techniques will evolve iteratively during all the phases of the refactoring selection and evaluation process.

5.2. Description template

The catalogs of best practices, such as refactoring techniques or design patterns, should be well organized to be usable. It helps to unify techniques description and has many other benefits. First of all, user of the technique will have to get used to description template only once, and after that, working with the catalog will be very easy. The need in grasping meta information will be eliminated and the reader will be able to concentrate on the content only. Secondly, this enhances communication among developers. Communication plays an important role in software development. Team members should be able to understand each other quickly to work effectively and concentrate on concrete specific problems to solve instead of explaining general technique each time in details. It is particularly important in our context, knowing that there is always little time left for quality. The third reason why defining a sound description template is important is because its structure can save developers time when well organized. The chosen structure organization will be thoroughly discussed in sections order passage.

5.2.1. Structure

The proposed template description was derived from existing ones, used in books *Refactoring: Improving the Design of Existing Code* [Fow99], *Refactoring for Software Design Smells: Managing Technical Debt* [SSS14] and *Refactoring to Patterns* [Ker04]. Below, the template is described. It consists of 6 sections. Their purpose and requirements that are claimed to content are stated for each section.

Technique name

Name should give the first impression of what the refactoring technique is all about. It can contain the name of the affected elements (such as service), or more general concept. It all depends on the intention of the technique. In the same time, to enhance technique communication and its use, the name should be catchy and concise. Another requirement to the name is to have similarity with original technique name. It will help developers that know the original technique when exploring the proposed catalog.

Summary

The section provides a very short summary It consists of a couple of sentences that remind the reader who is already familiar with the refactoring about what is it about. It does not have an intention to describe the whose technique for new readers. It also contains a reference to the original technique giving credit to the author.

Related Microservice Smells

The section provides a set of microservice smells that are related to the refactoring technique. The set is organized as keywords that reader can look through to match them to found smells in the system under refactoring.

Intent

Intention of the technique answers the question 'what will be done by applying this technique?'. In a concise way, the main goal of the refactoring should be described. So, as the section must clarify the results of the technique application, use of concrete terms and common patterns that the refactoring will lead to is preferable.

Motivation

While in the intent section the reader should get the question 'what?' answered, motivation will help to understand 'why?'.

The section presents possible reasons to use the described technique. This should help to understand whether the motivation of the developer matches any of the described ones. There can be also several different motivation factors, or reasons, that might encourage to use the technique.

There are some requirements that the content of the section should meet. First of all, the motivation factors presented should be very concrete. This way, it will be much easier for developer to match own motivation with one of those that are stated in the section. In the same time, it should not be bound to any particular domain. That will help to not to filter out this technique when it is indeed applicable.

Prerequirements

The section must answer the question 'what is required to use the technique?'. As there can be conditions present to use the microservices refactoring technique, they should be well defined. This section is the place for describing a context in which the technique can possibly be used with all the prerequirements that are claimed to the system. The prerequirements should be described using common concepts, pattern names and terms.

Impact

The Impact section is indented to demonstrate consequences of the refactoring application. As the goal of any technique is improving quality of a system, the section is describing what quality factors are affected and in which way. The section will be organized by listing the affected quality factors with description of the impact. While the main purpose of techniques application is improving quality factors, there are always tradeoffs present. Those tradeoffs are also mentioned in the section under each quality factor. It is important to know them in advance.

Furthermore, there are different perspectives on quality of microservices systems - from the overall system's and from the service under refactoring's point of view. Thus, they will be considered separately for each quality factor. In most cases, the tradeoffs happen to be for the different levels, improving a quality factor from one perspective and worsen it from another.

Mechanics

After answering all question that help to decide whether to use the proposed technique, the mechanics will be demonstrated. The mechanics should be organized as a list of concrete steps. Those steps, with motivation in mind to tackle some quality issues, within applicable context, will lead us from a system, that has prerequirements for technique application towards a modified system that has the defined impact on its quality.

Discussion

The main intention of the section is to give some notes on the use of the refactoring. It should help to understand its advantages and disadvantages, weight tradeoffs and decide on its use. The discussion is also meant to bring better understanding of technique's consequences in different contexts. The section should reveal situations in which the refactoring has more advantages or disadvantages. It will also be referring design principles and best practices when describing known problems of techniques consequences.

Example

In the section a small visual example of the technique application is given. It is crucial for the reader's understanding to see a diagram that depicts the initial and refactored states of the system. The examples are synthetic to keep it very concise and convey only what is important.

5.2.2. Sections order

The sequence in which the description template is presented is meant to save time for users of the catalog. The main idea behind is to enable developers to filter out irrelevant refactoring techniques as quickly as possible. Irrelevant techniques are those that either do not satisfy developer's needs or not applicable in the system under refactoring.

The entry point to each refactoring technique is its name. As was discussed previously, it will already speak its relevance to developers familiar with original refactoring. Right after the name a short, concise summary will be given. It is not meant to describe the technique to new readers, but can remind the main idea of the refactoring to people that used it before. In the summary, there is a reference to original technique. Investigating it will be interesting for those who will try to find other options of using the technique, not described in mechanics. It can also inspire developers to further adaptations of the refactoring. Following the short summary, a set of related microservice smells is given. It should help reader to match given smells to identified ones in the own system. After the first impression is given, the intent and motivation sections should clarify whether technique can solve an existing problem in a system that developer is trying to tackle. At this point, it is assumed that there are no further questions whether proposed refactoring can help developer. Then prerequirements passage will present additional conditions to use the refactoring. In case the system works within applicable context and fulfills all prerequirements it can be immediately used. Otherwise, only if the motivation to use it is strong enough developer will try to fulfill the requirements. The impact section is meant to demonstrate the way quality factors affected by the technique application, that is the main aspect to accept or reject refactoring use. In the following discussion section more thoughts of the author are presented that might contribute the reader's understanding as well as pointing our some aspects not mentioned in any of the previous sections.

5.3. Evaluation

The refactoring applicability and validity is the most important aspect while building a refactorings catalog. It is pointless how good the description of the technique is or how good arguments in favor of the refactoring are if it is not applicable. Applicability of every refactoring technique can be defined only on the evaluation phase.

Except of checking the possibility to apply the technique, evaluation brings many other benefits. This stage can help to correct mistakes in assumptions about its impact. It will also help to improve the mechanics description of the collected techniques. All the results of evaluation will be used to improve the catalog.

There are several different options on how to performs evaluation. The main requirement for evaluation is presence of expertise. Software professionals on different positions that have worked or anyhow faces with microservices is the target group that has the necessary expertise. Therefore, evaluation to be considered valid should include their assessment of proposed refactoring catalog.

To validate refactoring techniques applicability it is decided to perform questionnaire. A questionnaire is a research instrument consisting of a series of questions (or other types of prompts) for the purpose of gathering information from respondents. [Que] The target group has to consist of professionals in software development on different positions to assess the refactorings from different perspectives. Therefore, the questionnaire should have questions on the background of interviewees.

To understand better the interviewee's expertise - we should also understand what experience they have. While the information about general experience in software engineering can be implied from answers on background information questions - the more detailed experience with microservices should be questioned separately.

The refactoring catalog construction follows the predefined approach. Reliability of the final results depends on the validity of the approach. Therefore, the questionnaire should help to assess the chosen approach.

Afterwards, we need the assessment of the main result - refactorings catalog. There should be questions on each technique providing detailed explanation of each of them. Besides questions that help to estimate current results validity - we can also benefit from asking professional an advice. This contribution will support later work on the catalog and each separate technique improvement.

As the questionnaire is voluntarily - it should be kept short and all the questions should be possible to omit. It should also increase the validity of answers, as meaningless answers will not be counted (the questions interviewee has no opinion on will be skipped).

5.4. Summary

The chapter presented the concept of the thesis. First, the approach to follow was defined that answers the first research question. There were three different approaches presented. Then, all the approaches were assessed and the one providing the most reliable results was chosen. The final process was derived from chosen approach. Afterwards, the most important and complex approach steps were discussed in details.

Next section demonstrated the description template. Every section of the template was defined with argumentation of its mission. The chosen section order was discussed as well.

The last section - evaluation - showed the chosen way to validate the results. The refactoring catalog and each technique separately was chosen to assess using questionnaire. The target group and general structure of the questionnaire was presented.

6. Initial Refactorings Selection

Contents

Due to defined process in the chapter 5, the first step in the initial refactorings selection is matching code smells to microservices smells. It is the most important and creative step. Depending on the quality of identified matches the resulting catalog of refactoring techniques will be more or less applicable. The better matches are identified, the less time will be spent on inapplicable techniques.

We already defined possible ways and gave some instruction on how to perform the matching in section 5.1.5. In the same section, we also discussed use of bad smells of different levels. We also defined the catalog of microservices smells in section 4.1 that we are intended to match to bad smells.

The second step presented in the chapter is discovering refactorings for bad smell.

6.1. Microservices Smells Matching

The matching results are presented in table view. While there can be several different names for the same bad smell, the matching table does not contain all possible variants. The aliases to the used smells are provided in appendix A. While some aliases represent different levels of generality, the most specific ones were chosen for each specific case.

The presented table 6.1 with results of microservices smells to bad smells matching also has bad smells description. It does not contain all the microservices smells from the catalog for the reasons mentioned in section 5.1.5. Besides smells that are not related to the system under development (etc. organizational ones) we also do not include smells that do not have any matches from set of discovered bad smells. Some of them are Too Many Standards [TL18], Ambiguous Service [Bog+19] and Connector Envy [Gar+09]. They are too specific and the problems arise only in microservices systems. To identify refactorings for such smells another approach is required that is part of future work.

Microserivces Smells Matching		
Microservice smell	Bad Smells	Bad Smells Description
Bottleneck Service [TL18]	Hub-like Modularization [SSS14]	An abstraction has dependencies (both incoming and outgoing) with large number of other abstractions
Cyclic Dependency [TL18]	Cyclically - Dependent Modularization [SSS14]	Two or more abstractions depend on each other directly or indirectly (creating a tight coupling between the abstractions)
	Dependency Cycles Between Packages [LR06]	Cycles between packages can be created through use, inheritance, or through a combination of use and inheritance
	Cycles between Subsystems [LR06]	Cycles between subsystems can be created via use, inheritance or through a combination of use and inheritance
Inappropriate Service Intimacy [TL18]	Inappropriate Intimacy [Fow99]	Sometimes classes become far too intimate and spend too much time delving into each others' private parts
	Deficient Encapsulation [SSS14]	The declared accessibility of one or more members of an abstraction is more permissive than actually required
	Subsystem - API Bypassed [LR06]	Since the popular programming languages do not offer generic mechanisms for the definition of subsystems, projects must fall back on conventions. Consequently the subsystem's public interface – the API – will be defined through conventions
	Feature Envy [Fow99]	Data and behavior that acts on that data belong together. When a method makes too many calls to other classes to obtain data or functionality, Feature Envy is in the air
	Indecent Exposure [Ker04]	The smell occurs when methods or classes that ought not to be visible to clients are publicly visible to them
Chatty Service [TL18]	Broken Modularization [SSS14]	Data and/or methods that ideally should have been localised into a single abstraction are separated and spread across multiple abstarctions
	Feature Envy [Fow99]	Described above

Microservice smell	Bad Smells	Bad Smells Description
Microservice Greedy [TL18]	Middle Man [Fow99]	Delegation is good, and one of the key fundamental features of objects. But too much of a good thing can lead to objects that add no value, simply passing messages on to another object
	Lazy Class [Fow99]	A class that isn't doing enough to pay for itself should be eliminated
	Too Small Packages [LR06]	Packages with one or two classes are often not worth the effort of introducing them: the complexity created by the package is not offset by its additional structuring
	Subsystem Too Smal [LR06]	Too small subsystems shift complexity from subsystems into the dependencies among the subsystems themselves
	Broken Modularization [SSS14]	Described above
Not Having an API Gateway [TL18]	Subsystem-API By-passed [LR06]	Described above
	Deficient Encapsulation [SSS14]	Described above
Mega-Service [TL18]	Large class [Fow99]	Fowler and Beck note that the presence of too many instance variables usually indicates that a class is trying to do too much. In general, large classes typically contain too many responsibilities
	Long method [Fow99]	In their description of this smell, Fowler and Beck explain several good reasons why short methods are superior to long methods. A principal reason involves the sharing of logic. Two long methods may very well contain duplicated code. Yet if you break those methods into smaller methods, you can often find ways for the two to share logic. Fowler and Beck also describe how small methods help explain code. [Inc05]
	Multifaceted Abstraction [SSS14]	An abstraction has more than one responsibility assigned to it [SSS14]
	God Class [Rie96]	A god class is a class that performs most of the work, leaving minor details to a collection of trivial classes

Microservice smell	Bad Smells	Bad Smells Description
	Too Large Package [LR06]	Packages with a high number of classes indicate that they serve more than one specific responsibility
	Subsystem Too Large [LR06]	The phenomenon that no subsystems are defined is a special case of too large subsystems
	Subsystem-API Too Large [LR06]	When the API of a subsystem becomes too large in relation to the implementation, the main purpose of the subsystem is not served
	Hub-like Modularization [SSS14]	Described above
Leak of Service Abstraction [TL18]	Overgeneralization [LR06]	In order to assure that subsystems provide the greatest extent of reusability, they must be flexibly applicable. This generalization can be overdone though,which will result in the subsystem's overgeneralization. It will become more flexible than it actually needs to be
	Insufficient Modularization [SSS14]	An abstraction (such as a class or interface) exists that has not been completely decomposed and a further decomposition could reduce its size, implementation complexity, or both
Low Cohesive Operations [TL18]	Divergent Change [Fow99]	Occurs when one class is commonly changed in different ways for different reasons
	Multifaceted Abstraction [SSS14]	Described above
Scattered Parasitic Functionality [TL18]	Combinatorial Explosion [Ker04]	A subtle form of duplication, this smell exists when numerous pieces of code do the same thing using different combinations of data or behavior
	Duplicated Code [Fow99]	Explicit duplication exists in identical code, while subtle duplication exists in structures or processing steps that are outwardly different, yet essentially the same.

Microservice smell	Bad Smells	Bad Smells Description
	Solution Sprawl [Ker04]	When code and/or data used in performing a responsibility becomes sprawled across numerous classes, solution sprawl is in the air
	Broken Modularization [SSS14]	Described above
Service Chain [TL18]	Message Chains [Fow99]	Occur when you see a long sequence of method calls or temporary variables to get some data. This chain makes the code dependent on the relationships between many potentially unrelated objects
Wrong Cuts [TL18]	Data Class [Fow99]	These are classes that have fields, getting and setting methods for the fields, and nothing else. Such classes are dumb data holders and are almost certainly being manipulated in far too much detail by other classes.

Table 6.1.: Microserivces Smells Matching

6.2. Refactorings Discovering

After the smells matching, refactoring techniques for each microservices smell should be selected for further evaluation. Such filtering helps us to reduce amount of work and eliminates irrelevant techniques.

Besides the refactorings techniques that were discovered for each matched bad smell, the resulting set of techniques for microservices smell will include those, that were chosen based on proposed solution presented in section 4.1 (proposed in “On the Definition of Microservice Bad Smells”) in front of each smell. It can help us to expand the amount of techniques to evaluate.

Initially selected refactorings are presented in the table 6.2 with corresponding microservice smell. While some of the refactorings can be familiar to the reader, other are not that popular. Therefore, a short summary of each refactoring with some notes is provided afterwards.

It is worth mentioning that some of the refactorings were filtered out because they do not work with the same elements that are mentioned in smells they tackle. For example, Pull Up Method [Fow99] is to be matched to Scattered Parasitic Functionality [TL18] smell but its application requires to change inheritance relation between components, while this relation does not exist on an architectural level. Other refactorings work with OO-patterns. While they may be useful in tackling a problem, they are not chosen because there are no corresponding patterns exist for microservices. A concrete example would be technique called Replace State-Altering Conditionals with State [Ker04] that is, due to the chosen approach, matches Mega-Service smell [TL18] Adaptation of those patterns for microservices might be possible, but lie outside of scope of this thesis.

Refactorings Matching	
Microservice smell	Refactorings
Bottleneck Service [TL18]	Split responsibilities up across multiple new/old abstractions [SSS14]. Assign misplaced hub members to appropriate abstractions [SSS14].
Cyclic Dependency [TL18]	Break the dependency cycle [SSS14]. The Classic Removing of Cycles [LR06]. Introducing a Dependency Graph Facade [LR06]. Hide Delegate [Fow99].
Inappropriate Service Intimacy [TL18]	Remove Middle Man [Fow99]. Inline Class [Fow99]. Inline Method [Fow99]. Move Method [Fow99]. Move Field [Fow99]. Change Bidirectional Association to Unidirectional [Fow99]. Extract Class [Fow99]. Extract subset of cohesive members to separate abstraction [SSS14]. Hide Delegate [Fow99]. Moving Classes [LR06]. Introducing a Dependency Graph Facade [LR06]. Encapsulate Method [SSS14]. Encapsulate Field [Fow99].
Chatty Service [TL18]	Move method [Fow99] Encapsulate Field [Fow99]
Microservice Greedy [TL18]	Inline Method [Fow99]. Move Method [Fow99]. Encapsulate Field [Fow99]. Inline Class [Fow99]. Moving Classes [LR06].
Not Having an API Gateway [TL18]	Introducing a Dependency Graph Facade [LR06] Encapsulate Field [Fow99]. Encapsulate Method [SSS14]. Introduce Gateway [Fow19].
	Moving Classes [LR06]. Extract Class [Fow99]. Extract subset of cohesive members to separate abstraction [SSS14].

Mega-Service [TL18]

Microservice smell	Refactorings
	Extract Method [Fow99]. Compose Method [Ker04]. Introduce Parameter Object [Fow99]. Move Accumulation to collecting parameter [Ker04]. Replace Method with Method object [Fow99]. Introduce private helper methods [SSS14]. Apply ISP to make client-specific interfaces [SSS14]
Leak of Service Abstraction [TL18]	Replace Parameter with Explicit Method [Fow99]. Apply ISP to make client-specific interfaces [SSS14].
Low Cohesive Operations [TL18]	Extract class [Fow99]
Scattered Parasitic Functionality [TL18]	Extract method [Fow99] Extract class [Fow99] Move method [Fow99] Move field [Fow99]
Service Chain [TL18]	Hide Delegate [Fow99] Extract Method [Fow99] Move Method [Fow99]
Wrong Cuts [TL18]	Move method [Fow99] Extract method [Fow99] Encapsulate field [Fow99]

Table 6.2.: Refactorings Matching

Now we need to introduce refactoring techniques that were identified during the discovering process. Those refactoring are applicable to original bad smells that were matched to microservices ones. They will require adaptation later on. Below we give a short summary for the original refactorings. More complete description can be found in the original sources.

- **Break the dependency cycle** [SSS14]. In case one of the dependencies is unnecessary and can be safely removed, then remove that dependency. If possible, move the code that introduces cyclic dependency to an altogether different abstraction. In case the abstractions involved in the cycle represent a semantically single object, merge the abstractions into a single abstraction. The refactoring proposes three different more atomic techniques: remove class, inline class and extract class and therefore the three will be considered separately for tackling the smell.
- **The Classic Removing of Cycles** [LR06]. Having artifacts A and B it is necessary to split one of them (e.g. B) such a way that B1 depends on A and B2 whereas A depends only on B1. The refactoring proposes a technique that uses more atomic extract class refactoring.
- **Introducing a Dependency Graph Facade** [LR06]. In order to structure

dependencies between packages, subsystems and layers hide a number of classes behind a facade that is employed to simplify the handling of multiple classes, we can also use a facade to hide a subsystem's dependency graphs from the client of that subsystem.

- **Hide Delegate** [Fow99]. Create methods on the server to hide the delegate.
- **Inline Class** [Fow99]. Move all classes features into another class and delete the former one.
- **Inline Method** [Fow99]. Put the method's body into the body of its callers and remove the method.
- **Move Method** [Fow99]. Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.
- **Move Field** [Fow99]. Create a new field in the target class, and change all its users.
- **Change Bidirectional Association to Unidirectional** [Fow99]. Drop the unneeded end of the association.
- **Extract Class** [Fow99]. Create a new class and move the relevant fields and methods from the old class into the new class.
- **Extract subset of cohesive members to separate abstraction** [SSS14]. Take a cohesive part of members that belong together and extract them.
- **Moving Classes** [LR06]. Move classes to follow SRP.
- **Encapsulate Method** [SSS14]. Make the method private (or protected if necessary).
- **Encapsulate Field** [Fow99]. Make it private and provide accessors.
- **Remove Middle Man** [Fow99]. Get the client to call the delegate directly.
- **Extract Method** [Fow99]. Turn the fragment into a method whose name explains the purpose of the method.
- **Introduce Parameter Object** [Fow99]. Replace group of parameters that naturally go together. with an object.
- **Move Accumulation to collecting parameter** [Ker04]. Accumulate results to a Collecting Parameter that gets passed to extracted methods.
- **Replace Method with Method Object** [Fow99]. Turn the method into its own object so that all the local variables become fields on that object.

- **Introduce private helper methods** [SSS14]. Introduce private helper methods that help simplify the code in that method.
- **Apply ISP to make client-specific interfaces** [SSS14]. Break down the interface of class implementing it into several.
- **Replace Parameter with Explicit Method** [Fow99]. Create a separate method for each value of the parameter.

6.2.1. Refactorings Merge

Many of the discovered and presented in the section refactorings also have aliases. They will be taken into consideration while adapting a technique to use for tackling microservices smell. Other refactorings are not aliases, but represent the same idea on the architectural level based on the previously performed components matching in section 5.1. Therefore, they need to be merged and the table with results of the merge is shown in the table 6.3. In the table the proposed names of refactorings adaptation are mentioned. They will be also used in the catalog.

Refactorings Merge	
Microservice Refactoring	Bad Smells Refactorings
Inline Service	Inline Method Inline Class Break the dependency cycle
Extract Service	Extract Class [Fow99] Extract Method [Fow99] Extract subset of cohesive members to separate abstraction [SSS14] Introduce private helper methods [SSS14] Replace Method with Method Object [Fow99] Split responsibilities up across multiple new/old abstractions [SSS14] The Classic Removing of Cycles [LR06] Break the Dependency Cycle [SSS14]
Replace Parameter with Explicit API Method	Replace Parameter with Explicit Method [Fow99] Apply ISP to make client-specific interfaces [SSS14]
Move Responsibility	Move Class [Fow99] Move Method [Fow99] Move Field [Fow99] Moving Classes [LR06] Split responsibilities up across multiple new/old abstractions [SSS14] Assign misplaced hub members to appropriate abstractions [SSS14]

Microservice Refactoring	Bad Smells Refactorings
Introduce API Gateway	Introducing a Dependency Graph Facade [LR06] Hide Delegate [Fow99]
Require Data for API Method	Introduce Parameter Object [Fow99] Move Accumulation to collecting parameter [Ker04] Apply ISP to make client-specific interfaces [SSS14]
Encapsulate Responsibility	Encapsulate Method [SSS14] Encapsulate Field [Fow99]
Remove Middle Service	Remove Middle Man [Fow99]

Table 6.3.: Refactorings Merge

6.3. Summary

In the section we described an execution of the process of initial refactoring selection defined in the concept chapter earlier. It means that we performed the steps of smells matching and refactorings discovering that answers the third research question. We began with the first step - matching bad smells to microservices smells. Afterwards, we demonstrated the refactorings that were found based on the matchings. We also added definitions of the refactorings. After that in the Refactorings Merge table we combined some of the closely related on architectural level refactorings.

7. Refactoring Catalog

Contents

The chapter presents the main results of the thesis - selected refactoring techniques. Describing each techniques, we will follow the template defined in the concept chapter. All the refactorings were selected on the previous step of the general process - initial selection.

The catalog presents final results of the refactorings adaptation. We defined and described the whole process of adaptation in section 5.1.6. We already defined matched smells for each refactoring that are mentioned in the description. We also partially derive motivation section from the original refactorings, while extending it to consider microservices-specific cases. The prerequirements section is completely unique for the catalog. We took into consideration specific aspects of systems with microservices architectural style to understand what state of the system can allow performing each refactoring. Impact describes consequences of the technique application and was derived by analysis and comparison systems state before and after refactoring.

7.1. Inline Service

Inline Service is an adaptation of Inline Class - a refactoring technique proposed by Fowler in *Refactoring: Improving the Design of Existing Code* [Fow99]. Its main goal is to change the scope of services - when all the responsibilities of one service (source service) are moved to another service (absorbing service).

Inappropriate Service Intimacy, Cyclic Dependency, Microservice Greedy

7.1.1. Intent

Intention of the technique is to merge services that communicate to each other when serving most of their responsibilities. Smaller service, considered as a delegate is to be merged into bigger one. The goal is to remove the need of the component to communicate over network and the need of developers to work with separate services while maintaining these system's responsibilities.

7.1.2. Motivation

The use of the technique can have different motivations. First motivation matches the motivation of the original technique. We use the technique because the source service does not do much anymore and there is no need or benefits to have it as a separate service. In other words, it does not deserve to live and more important - be maintained. Another concern that can motivate to apply the technique is because the service's scope and its responsibilities are not well defined. During the work with the service it can become clear that requirements that affect the service always require changes in another service as well. That is why it will be beneficial to merge the smaller service (that is our source service) into bigger one (absorbing service). The third motivation would be to reduce chattiness. That is, using the technique there will be one less service in the system for absorbing service to communicate with that will bring performance improvement.

7.1.3. Prerequirements

Having an **API Gateway** in place is the only prerequirement for this technique. If the source service is not exposed to any external clients there are no prerequirements for the technique.

7.1.4. Impact

The main impact of the adapted technique stays the same as in original. It is maintainability that we would like to improve. However, some quality factors that are not affected while applying Inline Class are among the impact of the Inline Service.

- **Maintainability**. Moving features from one service to another means affecting **modifiability** of the system. In case, when the absorbing service is responsible for the features the methods provide, will mean that we adhere to CCP. It helps us, as we will deal only with this one service when requirements change. Having the two services separately would mean changing the two services for one change in requirements. It also could mean having work for two different teams that will have to communicate to solve the task. This would reduce speed of introducing functionality for the new requirement. From the service's perspective, the modifiability is decreasing, as maintaining thick services is more difficult.

The modification also means testing the changed code that is a concern of system's **testability**. Having two separate services would make it more difficult. In places, where unit testing would be enough, integration tests will be required.

However, the **modularity** of the system is decreasing. The services that utilized one thin service before will now have to deal with thicker service.

Analysability of the code will also benefit from this refactoring. When the CCP is followed carefully, we always have one service to look at for identifying issues and only one service to assess the impact of the intended changes. On the other hand, it could decrease maintainability. This situation is analyzed in the discussion section.

- **Performance efficiency**. One of the main motivations to apply Inline Service is to reduce chattiness in the system. The problem of having too small services is system's decreased **time behaviour**. Using the proposed technique leads to thicker services, that can increase performance efficiency, but under condition that the absorbing service used the source service. In other cases, the calls made to source service will be redirected to the absorbing service that will not bring any performance benefits.

  Another aspect of performance efficiency that is improved is **resource utilization**. This aspect of quality is improved while there is less services in the system after the refactoring technique application.

- **Usability**. Having thicker services can decrease **interoperability**. Depending on thinner services, as following ISP, is always less risky. However, if the two services were always used together due to their common responsibilities, that is not the case, as changes in one services would anyway affect both services.

  Applying the technique would even bring benefits for **learnability**, as using the one service is more intuitive.

7.1.5. Mechanics

1. Declare the api methods of the source service onto the absorbing service. Delegate all the methods to the source service.
2. Change all references from the source service to the absorbing service across services in the system.
3. Redirect calls in the API Gateway (if there are any) from the source service to the absorbing service.
4. Use Move Responsibility to move features from the source service to the absorbing service.

7.1.6. Discussion

All the motivation factors are reasonable enough to use the described technique. However, it cannot be guaranteed that the benefits will overcome disadvantages. To win most from this technique, one should think thoughtfully about services' responsibilities. The SRP and CCP are two important principles that can help to decide whether to use this technique or not. Developers should also consider dangers of Mega-Service smell. It leads to many disadvantages that are difficult to recover from. Therefore, special techniques should be applied to reason the use the refactoring.

7.1.7. Example

In the example depicted on figure 7.1 we see the problem mentioned in motivation section - the service A does not do much work. We also see that service B is the only direct user of service A. Service B delegates its job to service A to serve the requests that come from other services.

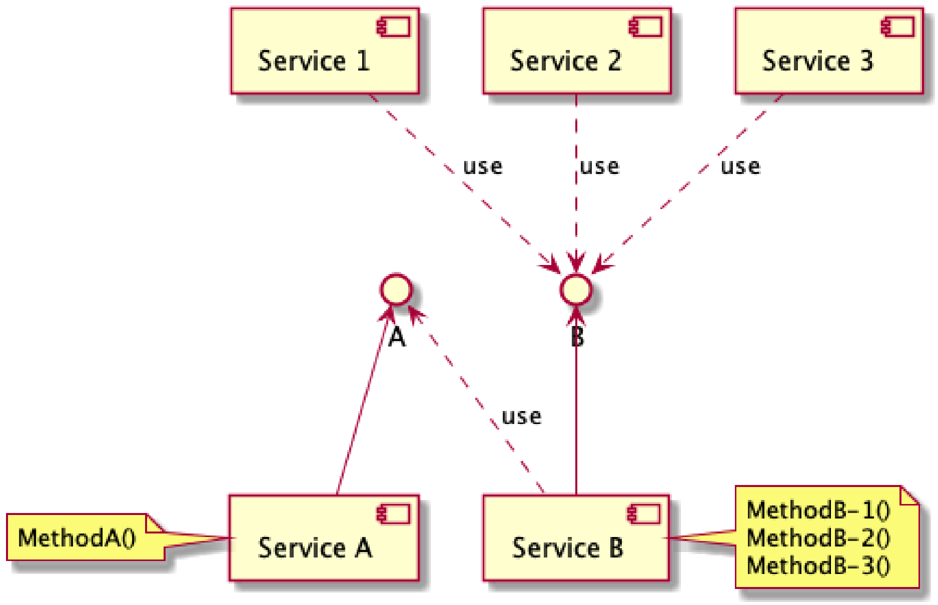


Figure 7.1.: Before Inline Service

Examining the figure 7.1 one can recognize the microservices greedy smell. There is not much that is left for service A to do except of handling the only type of request from service B with its only api method. Application of inline service as shown on figure 7.2 will reduce the number of services by merging service A into service B.

In case when service B accesses the data from service A to serve requests instead of dealing with its own data, we have an inappropriate service intimacy smell. Applying inline service for service A will move all the data to service B as demonstrated on figure 7.2 and there will be no need to access other services to answer the requests.

If we had a situation when service A sends requests to any of the services from upper part of the figure 7.1, we would deal with cyclic dependency smell. The proposed refactoring technique would help to cope with the problem, though not fully solve it. After merging the services A and B it will be more clear what responsibilities are missing for service B to keep working. It is likely that some of the services depicted in the upper part of the figure 7.2 would also require refactoring to fully avoid the cyclic dependencies.

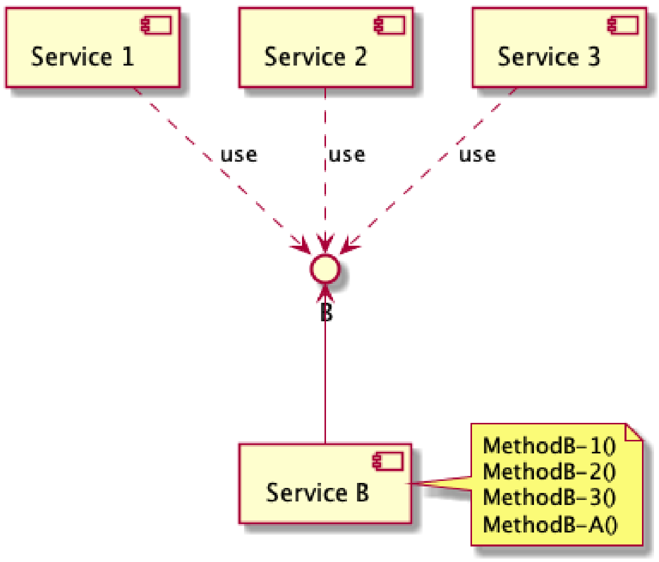


Figure 7.2.: After Inline Service

7.2. Replace Parameter with Explicit API Method

Replace Parameter with Explicit API Method is a specific refactoring technique that belongs to a group of refactorings where signature of method is changes. Original technique was proposed by Fowler in his *Refactoring: Improving the Design of Existing Code* [Fow99]. The main goal of the refactoring is to make the interface of service more specific for a client. It also reduces complexity of method. This leads us to a simpler and clearer interface.

Leak of Service Abstraction

7.2.1. Intent

The technique's intention is to fight the overgeneralization of the interface of service under refactoring. Microservices system should have specialized services that have one responsibility and, therefore, should have specific interface to help its users to understand and use functionality for this only responsibility.

7.2.2. Motivation

Use of refactoring has the only motivation - make the service easier to use for its clients. It is especially important in big microservices systems where many services serve for their purposes. Overgeneralization misleads the service intention and can lead to its misuse. While services are encapsulated components - it should be clear what they do from the interface without looking at internal implementation. Another reason to support the idea is difficulty to check the internal implementation while using a service. Current IDEs do not support navigation to methods outside of the project.

7.2.3. Prerequirements

It is required to have an **API Gateway** in the system to perform the refactoring without affecting existing clients.

7.2.4. Impact

The major part of the impact of the technique application is related to usability of the system. However, the maintainability and analysability of the system are affected as well.

- **Usability**. Having api of service tuned for their clients is increasing **interoperability** of the service. The more specific the methods are - the easier it is for clients to get the intention of each method as well as responsibility of the service.

  Applying the technique brings benefits for **learnability** of the service and the system in general, as all the methods will have more descriptive names with better communication of their purpose to the client.

7.2.5. Mechanics

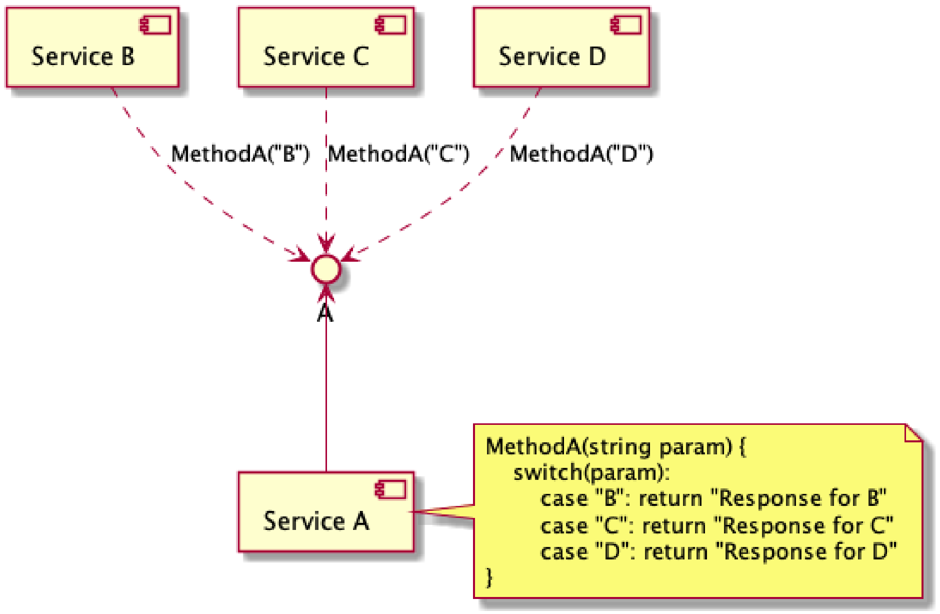
1. Create an explicit API method for each value of the parameter.
2. Replace each call of the conditional method with a call to the appropriate new API method.
3. Replace calls of the conditional method in API gateway with a call to the appropriate new API method.
4. When all callers are changed, remove the conditional method.

7.2.6. Discussion

While in general this technique helps to improve usability of the service, there are certain cases when it decreases the maintainability considerably. In such cases parameter could have too many options and therefore too many separate API methods would have to be created to serve the same functionality. Keeping all the methods up-to-date and reducing the code duplication with requirements changes can be much of workload.

7.2.7. Example

In the simple example depicted on figure 7.3 we see the problem presented in motivation section - service has the only method that is serving many different cases based on parameter value.



```
MethodA(string param) {
    switch(param):
        case "B": return "Response for B"
        case "C": return "Response for C"
        case "D": return "Response for D"
}
```

Figure 7.3.: Before Replace Parameter with Explicit API Methods

We can see that figure 7.3 demonstrates a Leak of Service Abstraction smell. The service is too general and therefore its responsibilities are not visible from the interface. Replacing the parameter with several explicit method solve the issue. As shown on figure 7.4 the enabling clients to grasp the responsibilities of the service easier and use it more intuitively.

7.3. Remove Middle Service

The refactoring is an adaptation of Remove Middle Man technique described by Fowler in *Refactoring: Improving the Design of Existing Code* [Fow99]. The main intention is to get rid of unnecessary delegation and use the service directly. The refactoring improves maintainability of the system. It can also improve performance.

Inappropriate Service Intimacy

7.3.1. Intent

The intention of the refactoring is to get the client service to call the delegate service directly. Service that serves many requests by simply delegating all work to other services

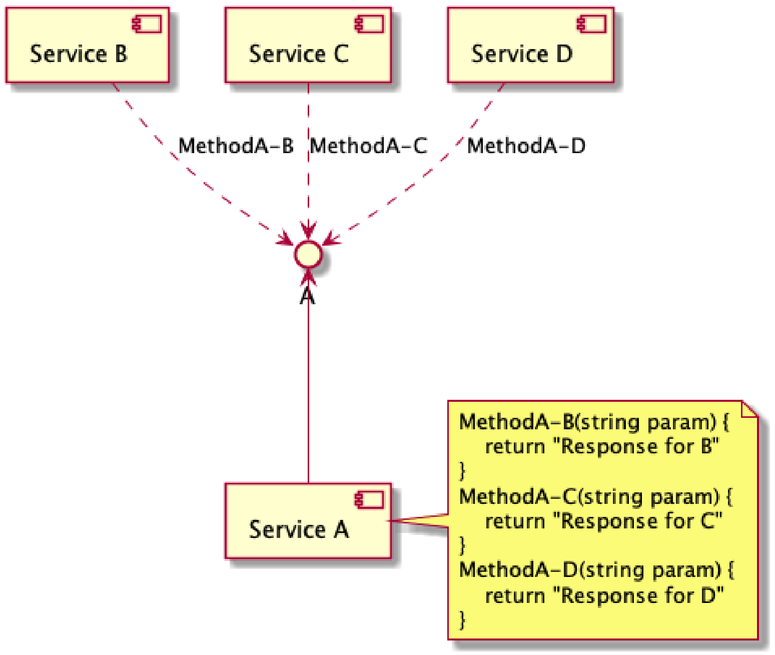


Figure 7.4.: After Replace Parameter with Explicit API Methods

should not be responsible for the requests handling.

7.3.2. Motivation

The main motivation of the technique application is reducing system complexity. We need to refactor a microservices system once we have a clearer understanding of proper application decomposition into services. The understanding comes with time and with system evolution. The separation of responsibilities leads to quality improvements and decreases operational complexity as well.

Another motivation could be increasing system's performance. Reducing communication between services by not passing requests through the middle service can lead to faster response times.

7.3.3. Prerequirements

In case the service to remove is used by external clients the system should expose its API via **API Gateway** to perform the refactoring safely.

7.3.4. Impact

Two main quality factors that the technique is aiming to improve are maintainability, performance of the system and usability of the service under refactoring.

- **Maintainability**. As the refactoring application leads to the rearrangement of responsibilities in the system, the maintainability of the system in general improves. As in the refactored version of the system services that do actual job are responsible also for requests handling, **analysability** of the system is increased. **Modularity** of the system is also increasing as responsibilities are not spread among different services anymore, but encapsulated in each service. The service that did simple delegation will not contain non-relevant to its main responsibilities API methods.
- **Performance efficiency**. The second main quality attribute that is affected by the refactoring application is **time behaviour**. The less communication between services there are - the faster the requests are handled. Bypassing the middle service helps to lessen the communication.
- **Usability**. Applying the technique contributes system's **learnability**. It is easier to learn the system when API methods are provided by responsible services. **Interoperability** is increased for the same reason.

7.3.5. Mechanics

1. Move API method from middle service that is provided to use delegate to delegate service itself
2. For each client using delegate directly replace calls to newly moved API method
3. For each client using middle service change service reference to use delegate.
4. Combine old API method of delegate with newly moved method.
5. Remove old API methods.

7.3.6. Discussion

This technique involves changing an interface of the service. It does bring much benefits for system quality, but in real world situation might require additional work to be done, namely application of section 7.7 The combination of the two techniques might help to achieve better results in separating concerns.

7.3.7. Example

A simple example on figure 7.5 demonstrates a situation where application of proposed refactoring is reasonable. Service A - the middle service - has an API method that is used by other microservices but internally simply delegates work to another service - Service B.

Exploring figure 7.5 we see that method MethodB actually does not belong to Service A as all it does is delegation to Service B. Therefore, application of the proposed refactoring will bring us all the benefits described in impact section and will bring us to the state shown on figure 7.6

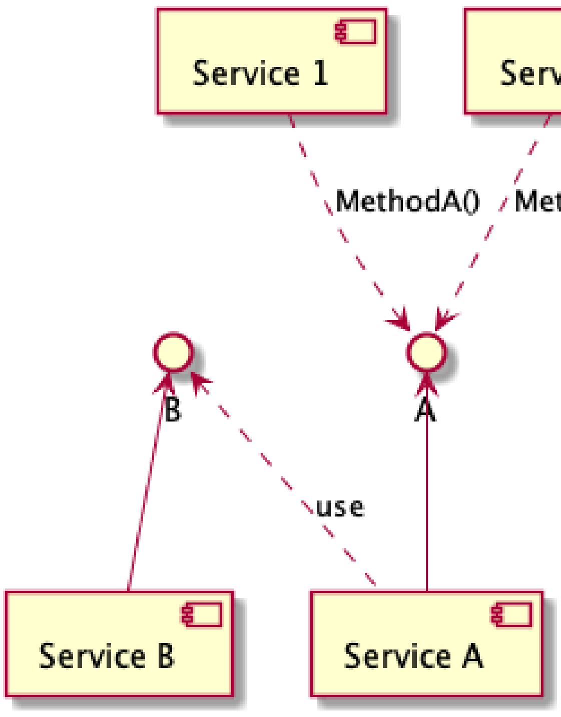


Figure 7.5.: After Remove Middle Service

7.4. Require Data for API Method

Require Data for API Method is an adaptation of merge of several refactorings. Original techniques are Introduce Parameter Object [Fow99], Move Accumulation to Collecting Parameter [Ker04] and Apply ISP to make client-specific calls [SSS14]. The refactoring helps to manage responsibilities by removing business logic code from service that is not responsible for it and make it accept the data that is required for service to serve its requests.

Mega-Service, Leak of Service Abstraction

7.4.1. Intent

The main intention of the refactoring is to remove business logic code form a service that the service is not responsible for. We substitute the code that gets and processes some data for service to work properly by accepting the data from client services.

7.4.2. Motivation

There are several microservices smells that the technique tackles. One of the motivations is to remove duplicated code that is spread across services. In a situation where service instead of accepting data from clients try to get and process the data itself, it contains

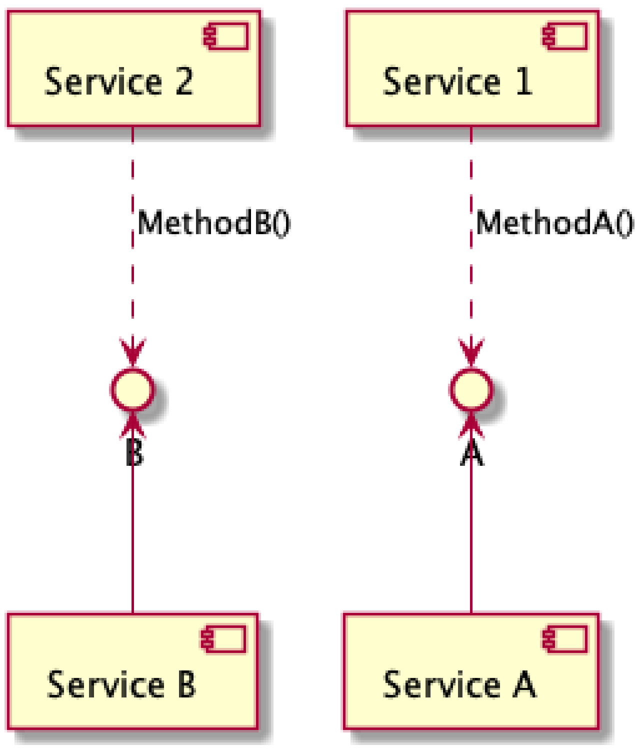


Figure 7.6.: After Remove Middle Service

the same code that is written in services that are actually responsible for this part of business logic. By concentrating the code only in the responsible service and simply require the required data by the service under refactoring we get less duplication.

Another motivation is to reduce the number dependencies for service under refactoring. By publishing a contract of requiring additional data we do not need to care about how this data will be acquired and do not need a dependency on the responsible service anymore.

7.4.3. Prerequirements

First prerequirement for the refactoring use is to have an **API Gateway** in the system. Secondly, it is also necessary to have a versioning policy for the system. If the service used only by clients that are under control, these prerequirements are not relevant.

7.4.4. Impact

The main consequences of the refactoring application is increased maintainability and usability.

- **Maintainability**. Removing the code duplication leads to increased **modifiability** of the system. Whenever requirements change, there will be no need to do the same modifications in several methods, but only one method will be affected.

  The same applies to system's **testability**. Testing several methods that are responsible for the same feature means changing these several tests each time methods change. The maintainability of tests is to be reduced considerably by applying the technique.

- **Usability**. Simplifying API affects usability of the system. It means affecting **interoperability** and **learnability**. However, it does not always increase those factors. It can be the case, that several more concrete methods will be more intuitive to use for clients, even if they are responsible for the same feature. It is required to think thoroughly when changing API applying the technique. The tradeoffs are described in the Discussion section.

7.4.5. Mechanics

1. Create a parameterized method that can be substituted for each repetitive method.
2. Replace one old method with a call to the new method with appropriate parameters across services of the system under refactoring.
3. Repeat for all the methods under control, testing after each one.
4. Redirect calls in the API Gateway (if there are any) from the old methods to the new method with appropriate parameters.

7.4.6. Discussion

Trying to reduce code duplication in the system follow the DRY (Don't Repeat Yourself) principle. It is an extremely important principle for system's maintainability. However, it is important to identify code that is indeed duplicated. For that, we should know where is fake and where is real duplication [Mar17].

When trying to simplify API by using the proposed refactoring, it is important to weigh advantages and disadvantages that it could bring. In many cases, it has a positive influence on system's usability. However, there are two aspects to consider. First of all, the achieved API will be more general. You can identify Leak of Service Abstraction afterwards. Secondly, the api might not match the domain after the modification. Not everything that has the same code is the same on the domain level. It is crucial to follow the domain understanding of system's users while working on public API. This eases the API use, makes it more intuitive.

7.4.7. Example

In the example depicted on figure 7.7 we see that Service B needs additional data to serve a request from Service A. However, it is not responsible for the data. It can do the job itself or delegate the job to Service C that is presented on the picture.

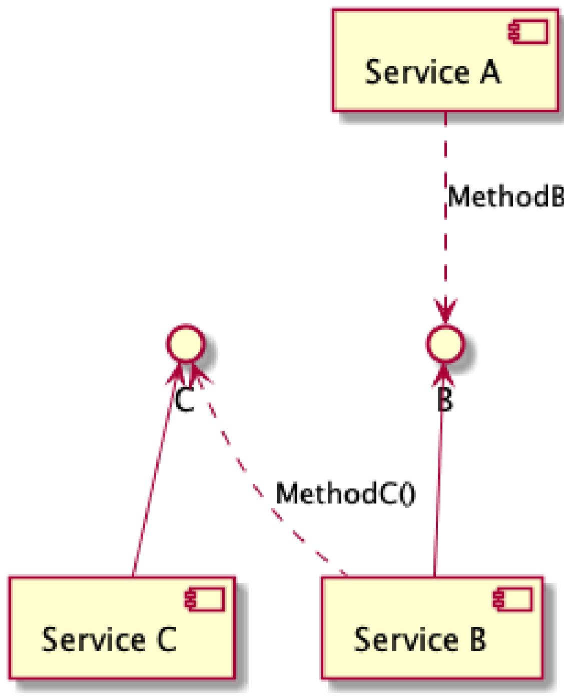


Figure 7.7.: Before Require Data for API Method

The figure 7.7 demonstrates an unnecessary coupling between Service B and Service C. Application of the refactoring can remove this coupling. On the figure 7.8 we see that Require Data for API Method technique application removes the unnecessary coupling. In the final state of the system, Service B requires Data C from client to serve the request. Client is now responsible to provide the data. Most likely, client should go to Service C to ge the data.

7.5. Encapsulate Responsibility

Encapsulate Responsibility is a technique that comes from merge of Encapsulate Method [Fow99] and Encapsulate Field [Fow99] techniques adaptation. The main goal of the refactoring is to hide method and corresponding data of the service. It increases maintainability and usability. It can also increase performance efficiency of the system.

Inappropriate Service Intimacy, Chatty Service, Wrong Cuts, Not Having an API Gateway

7.5.1. Intent

The refactoring's intention is to hide part of its provided functionality. Sometimes, it is done to preserve service's intimacy while other times several methods are hidden providing

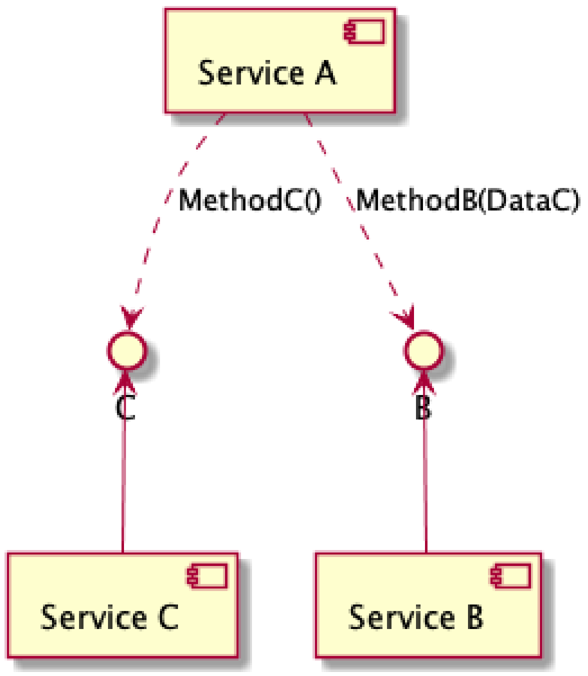


Figure 7.8.: After Require Data for API Method

one facade method that helps to achieve the same results as consequent invocations of those methods.

7.5.2. Motivation

There are several motivations to use the technique. We might want to reduce clients' confusion about the service use in case of unnecessary publishing methods of the service under refactoring.

We might want to reduce chattiness in the system in case, when we provide several methods that cover one functionality. Then, we can encapsulate those methods and provide clients a single method that will perform the task using those hidden methods internally.

We might also want to restrict direct access to data of the service. Then, instead of providing methods which only responsibility is to access the database, methods with business logic should be added that will use those database accessors internally. The less work is done with the service's data by its clients, the less chances are to have code duplication that increases maintainability.

7.5.3. Prerequirements

It is necessary to have an **API Gateway** in the system to use the technique.

7.5.4. Impact

There are different quality factors affected in different cases of the technique's application. Three main factors are maintainability, usability and performance efficiency.

- **Maintainability**. One of the main benefits of the refactoring is increased **modifiability** of the system. The technique helps to improve encapsulation of the service under refactoring that leads to reduced duplication of code throughout the system. It is easier to modify the service when its internals are not revealed through the interface.

  Testability of the system also increases when shrinking the service's interface. It will require less integration tests that involve several services.

  Analysability of the system benefits from the refactoring while smaller interfaces speak the service's purpose better. It lessen cognitive load while analyzing the service knowing that there are less possible ways for clients to use the service.

- **Performance efficiency**. **Time behaviour** problems arise when service provides several methods to cover one piece of responsibility. In this case, these methods are better to encapsulate providing one method.

- **Usability**. Having thicker services interfaces increases **interoperability**. It is easier to work with such services and less integration work is required.

  Learnability of the service under refactoring and the system in general also increase, as there are less public API methods that have higher-level purposes.

7.5.5. Mechanics

1. Make API methods to encapsulate usual class methods
2. Add method that will provide better abstraction to use service's responsibility
3. Use hidden methods in the new method
4. Change references from previous methods to new one. Change the way it is used. Remove unused API invocations
5. In the API Gateway change invocations of set of hidden methods to the new method invocation

7.5.6. Discussion

The main danger of the proposed technique's application is **Leak of Service Abstraction** smell. Before applying the technique, it is important to understand what methods should be covered with the facade. In the simpler case when methods are not yet used - the technique is a very good preventative mean to avoid service **Inappropriate Service Intimacy**.

7.5.7. Example

On the figure 7.9 we can see that Service B uses three methods of Service A to cover some functionality. These methods in the example are meant to be executed together to achieve results. As they are not used separately, it is reasonable to apply the proposed technique encapsulating those methods and providing a facade method.

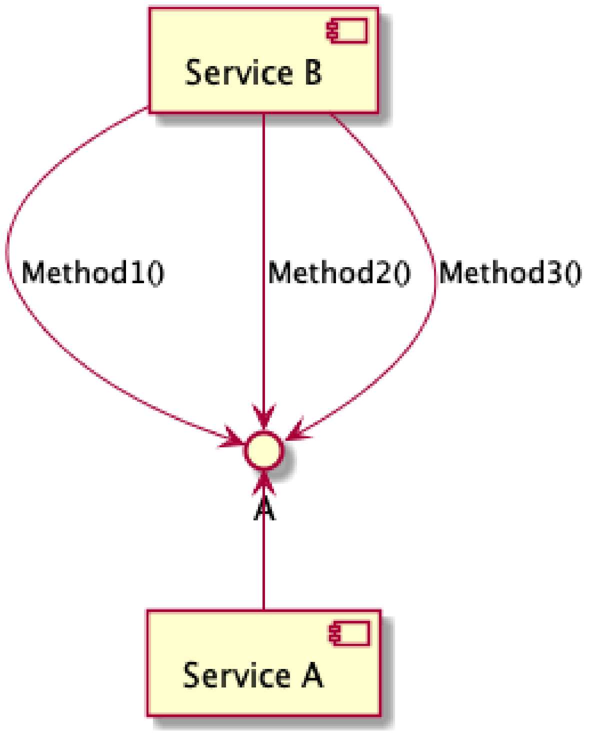


Figure 7.9.: Before Encapsulate Responsibility

Figure 7.10 depicts the final state of the system after technique's application. The facade method is called Method123 in our case and internally it uses the hidden methods to provide the same functionality.

7.6. Extract Service

Extract Service is an adaptation of Extract Class - technique proposed by Fowler in *Refactoring: Improving the Design of Existing Code* [Fow99]. The main intention of the refactoring is to extract part of responsibilities from one - source service into another - extracted service. It leads to increased system's maintainability due to better separation of concerns, but has a tradeoff of reduced performance efficiency.

Low Cohesive Operations,Scattered Parasitic Functionality,Service Chain,Bottleneck Service,Mega-Service,Inappropriate Service Intimacy,Cyclic Dependency

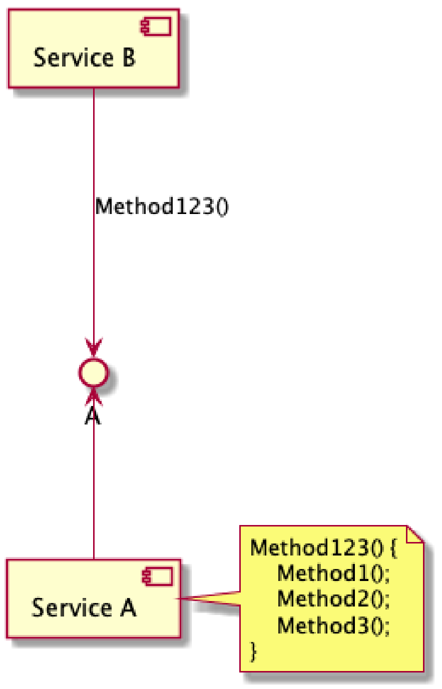


Figure 7.10.: After Encapsulate Responsibility

7.6.1. Intent

The intention of the refactoring is to extract responsibilities from service that it is not responsible for to newly created service. All the methods together with data are extracted and all the references to these methods are changed to new ones. It makes sense when SRP is violated and there is no appropriate service that would take those foreign responsibilities, otherwise Move Responsibility technique would be helpful.

7.6.2. Motivation

The technique helps to tackle many microservices smells. All the smells except of Cyclic Dependency can be split into two cases - two main motivations. There are either many services containing the whole or parts of the common responsibility or there is a foreign responsibility in some service. In the first case we want to improve maintainability of the system and in the second - maintainability of the service. In case of cyclic dependency the main motivation is to remove the cycle.

7.6.3. Prerequirements

The first prerequirement for the technique is to have **API Gateway** in place. Having an **API versioning policy** can ease the refactoring application.

7.6.4. Impact

Consequences of the refactoring's application are increased maintainability of the system and the service under refactoring and reduced performance efficiency. The technique also increases usability of the service.

- **Maintainability**. Increased maintainability is the main goal of the refactoring. **Modifiability** of the system and of the service under refactoring is increased. Extracting common responsibilities from several services in the system leads to modifications only in this service later on once requirements change. Extracting foreign responsibilities allows to arrange a separate team for the new service that helps in case when the service is growing to big.

  Testability of the system increases in case of extracting responsibility from several services as there will be only one set of tests for the responsibility. It becomes to maintain the tests.

  Modularity of the system increases. The number of services in the system increases where each service has one responsibility.

  Analysability of the services in the system increases with the refactoring. It is easier to identify services and their responsibilities when there is only one responsibility per service.

- **Performance efficiency**. The problem of having more services in the system is decreased **time behaviour**. In many cases, extracting common responsibility from several services will lead to more communication in the system. However, in case of extracting responsibilities from one service time behaviour is not affected.

  There is one more aspect of performance efficiency that is worsen - **resource utilization**. As we have more services in the system - there will be need for more resources.

- **Usability**. Having smaller services with only responsibility in the system increases **interoperability**. Such redistribution of responsibilities can help to follow ISP better.

  Learnability aspect of the whole system is worsen by the refactoring's application, while for the source services, learnability is improved.

7.6.5. Mechanics

1. Create a new service for responsibility to extract
2. If the responsibilities of the old service no longer match its name, rename the old service.
3. Use Move Responsibility on each one you wish to move for methods and data.
4. Review and refactor the interfaces of source and extracted services.

5. Change links in the API Gateway from the old services to the extracted one.

7.6.6. Discussion

While considering the refactoring's application it is crucial to avoid Microservice Greedy smell. The more services there are, the more complexity of the system is pulled up to operational level. The balance should always be kept to achieve the best maintainability of the system.

7.6.7. Example

As can be see from the example on figures 7.11 and 7.12, the states of the system before the refactoring matches the state after inline service refactoring application. The problem that is shown here is that Service B has two separate responsibilities that decrease maintainability.

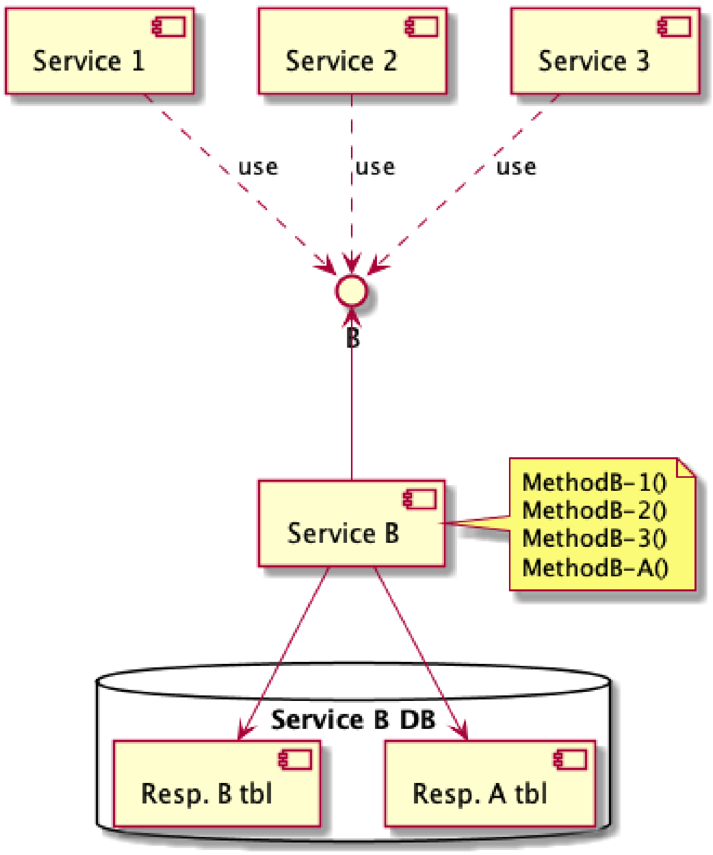


Figure 7.11.: Before Extract Service

There is only one case from motivation depicted on the figure. Another case would be having several services with same MethodA and Resp. A tbl. As shown on the figure

7.12, the method and respective data is moved to Service A by applying the technique. The final step shows exactly what we were aiming for - the responsibility is extracted into newly created service.

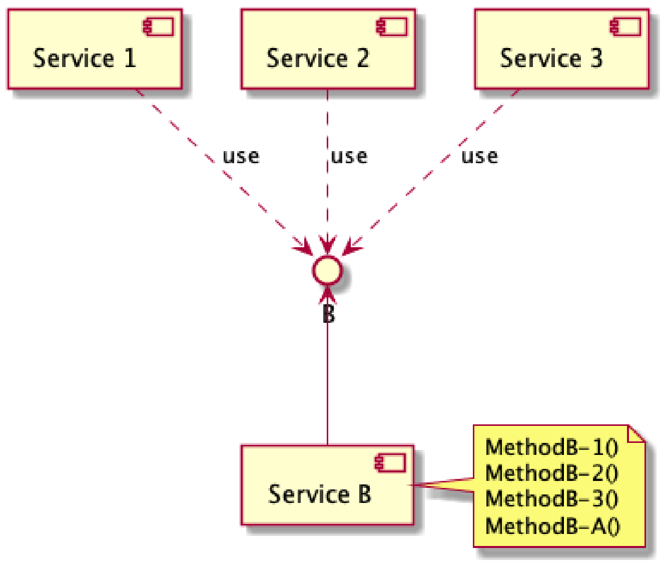


Figure 7.12.: After Extract Service

7.7. Move Responsibility

The technique is an adaptation based of several existing techniques such as Move Method [Fow99], Move Class [Fow99] and Move Field [Fow99]. With this refactoring it is intended to distribute responsibilities in a microservices system properly to follow SRP and CCP. The refactoring describes atomic steps of moving a responsibility and demonstrates consequences and possible dangers.

Service Chain, Scattered Parasitic Functionality, Chatty Service, Bottleneck Service, Mega-Service, Microservice Greedy, Inappropriate Service Intimacy, Wrong Cuts

7.7.1. Intent

The refactoring's intention is to better organize responsibilities distribution among services. Taking service under refactoring with foreign responsibility, we move its method and corresponding data that this service should not be responsible for out of it to more appropriate service.

7.7.2. Motivation

Having different smells this refactoring can help to tackle, we can define several motivations to use it. The main case when application of the technique is beneficial is when the service under refactoring has foreign responsibilities. It is the most general case and can be split into several. The service under refactoring might be used by another service to cover responsibility of the client service. It creates high coupling between the services and leads to organizational problems requiring more communication and coordination between teams responsible for the services. There also can be a case when the service under refactoring is used by external clients but for different reasons. It also leads to maintainability issues.

Another case is when there is a service that does not do much but provides access to data in its database. Then other services use it to access this data. To tackle this problem we can move the data together with data access methods to the client services of the system.

One more motivation to use the refactoring is to reduce the chattiness between services in a system. When responsibilities are not distributed right, it is often a case that there is far too much needless communication.

7.7.3. Prerequirements

API Gateway should be present in the system if the service under refactoring is used by external clients. **API Versioning** policy is also beneficial while applying the technique.

7.7.4. Impact

The refactoring affects maintainability, performance efficiency and usability of the system.

- **Maintainability**. Reorganizing responsibilities affects **modifiability** of the system. Once they are organized well, there is less effort from developers to coordinate the changes and most of the requirement changes will affect only one team. For the same reason the **testability** of the system is increased, it will be possible to cover more functionality with unit tests instead of involving multiple services for integration or functional testing.

  Analysability of the system is increased with better separation of concerns. It is easier to understand the cause-effect relation among services as well as between classes in each separate microservice.

- **Performance efficiency**. Improved **time behaviour** of the system is one and often just side benefit that this refactoring brings. Proper rearranging responsibilities of the services always leads to less communication between them.

- **Usability**. Applying the technique contributes system's **learnability**. First of all, is easier to learn the system when API methods are provided by responsible services. Secondly, assumptions on the service responsibilities based on previous experience with it should be met to increase learnability. **Interoperability** is increased for the same reasons.

7.7.5. Mechanics

1. Examine all features used by the source method that are defined on the source service. Consider whether they also should be moved.
2. Declare the method in the target service.
3. Move the data to the target service if needed.
4. Copy the code from the source method to the target. Adjust the method to make it work within the new service.
5. Turn the source method into a delegating method.
6. Remove the data from the database of service under refactoring.
7. Remove the source method or retain it as a delegating method.
8. Replace references from API method to newly created one in other services in the system and in the API Gateway.

7.7.6. Discussion

The main intention of the technique is to fix SRP violation in the system. As SRP is a principle - it is difficult to say how reasonable it is to apply the refactoring. In the most cases, there should be a strong reason, as the technique is challenging and quite radical. A reoccurring maintenance problem in cases described in motivation section could drive the techniques application.

7.7.7. Example

As there are many cases when the refactoring can be applied, only the abstract general idea is given below. We see state of the system before on figure 7.13 where the problem is depicted by naming only. The Service A on the picture has methods and data for responsibility that belongs to Service B.

Figure 7.14 on the other hand, shows how the violation of SRP can be fixed with the refactoring. It is a state of the system after the technique's application. It does not show how it affects client, because there are too many cases but it shows that now there is no data and no methods in Service A that it is not responsible for. The change of the situation has all the quality factors consequences described in impact section.

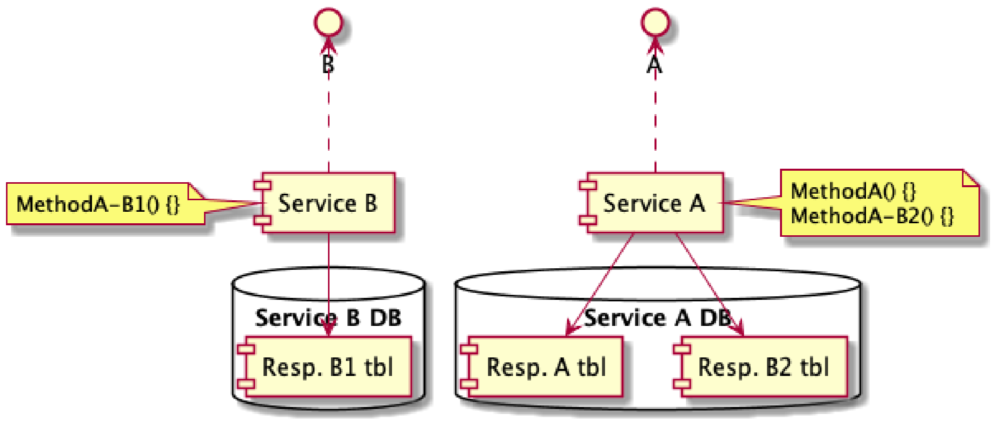


Figure 7.13.: Before Move Responsibility

7.8. Introduce API Gateway

The proposed refactoring is an adaptation of several matched techniques - Introducing a Dependency Graph Facade [SSS14] and Hide Delegate [Fow99]. The main intention is to hide complexity of using the system - several services in combination - behind one single entry-point service. This refactoring technique is towards the pattern API Gateway described in section 4.2.1.

Not Having an API Gateway, Service Chain, Inappropriate Service Intimacy, Cyclic Dependency

7.8.1. Intent

Introduce API Gateway is meant to create a new service in the system that provides a facade interface for external clients. The service has no business logic and simply delegates to other services in the system.

7.8.2. Motivation

There are several motivations to use the technique driven by mentioned smells. There can be a problem of having too difficult API for clients that can be make easier with gateway. Another case is when a chain of methods possibly form different services should be invoked to cover some functionality. These methods invocation can be combined in one method of gateway. In other cases, it is harmful to provide access to some methods to external clients that might reveal implementation details. Having a cyclic dependency can be another motivation to use the technique.

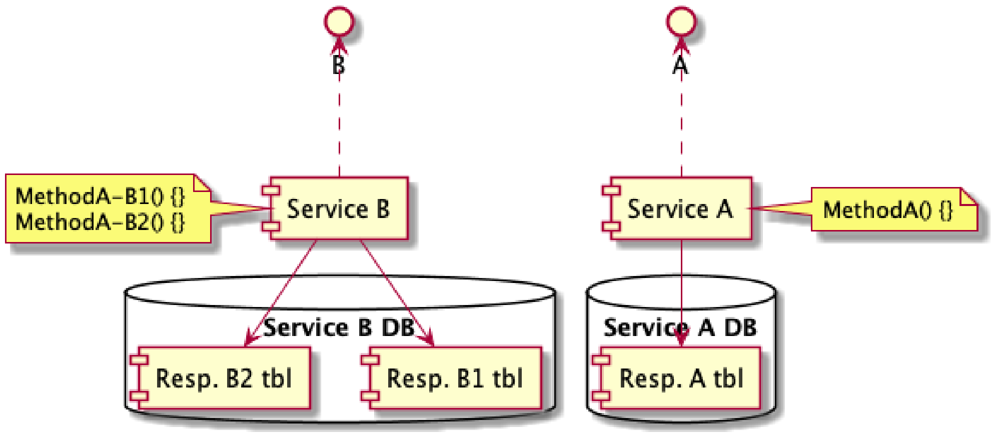


Figure 7.14.: After Move Responsibility

7.8.3. Prerequirements

There are no prerequirements to use the technique. Having an **API versioning** policy in place can help to perform the refactoring.

7.8.4. Impact

The refactoring impacts the system's usability the most. The technique also increases maintainability and can decrease performance efficiency.

- **Maintainability**. The proposed technique increases **modifiability** of the system. It is much easier to modify the services and their interface when there is the only facade service that external clients are aware of.

- **Performance efficiency**. Communication between services within the system can be faster than between external client and the system. Therefore, when we combine several methods of different services of the system in one API gateway service method increases **time behaviour** of the system. It decreases latency.

  Resource utilization is decreased as there is need for one more service to perform this refactoring.

- **Usability**. Often, increasing usability of the system in the main reason to use the refactoring. Proposed technique increases **interoperability** of the system. It is easier to work with system when it provides clearer, smaller and client-specific interface.

  For the same reason, **learnability** of the system increases by the technique application.

7.8.5. Mechanics

1. Create gateway service
2. Use Encapsulate responsibility for methods of each service that should not be directly available to clients

7.8.6. Discussion

The API Gateway pattern is meant to solve many problems and involves different concepts such as authentication or monitoring that is often part of API Gateway in practice. In the description of the technique we concentrate only on the main responsibility of API Gateway - introducing a facade for a system.

The refactoring is complex and involves the use of encapsulate responsibility (and sometimes move responsibility). Intention of the proposed technique is similar to what encapsulate responsibility is used for, but on a higher - system level.

As the most important quality factor that the refactoring impacts is usability of the system - it is crucial to develop a good, client-specific API. In some cases, it makes sense to use the refactoring towards Backends for Frontends.

7.8.7. Example

The example demonstrates the technique application on two services, while it can be extended to many more. On the figure 7.15 a situation where client works with two services of a microservice system is demonstrated. It is assumed that requests that client performs are meant to solve single task. Therefore, it would be easier for the client to make only one request. It would also increase performance.

The solution is Introduce API Gateway. The state of the system after the refactoring is depicted on figure 7.16. Here, client easily solves its task with only request and internally, API Gateway service delegates parts of the task to the Service A and Service B that are responsible for it.

7.9. Summary

The chapter presented the main results of the thesis - refactoring techniques. We described each of the selected techniques following the template. We defined their intent, motivation to use, prerequirements, impact on system's quality and mechanics. We also added discussion section where we demonstrated potential problems and tradeoffs. The last subsection of the refactoring description was an example with visualization of the technique impact.

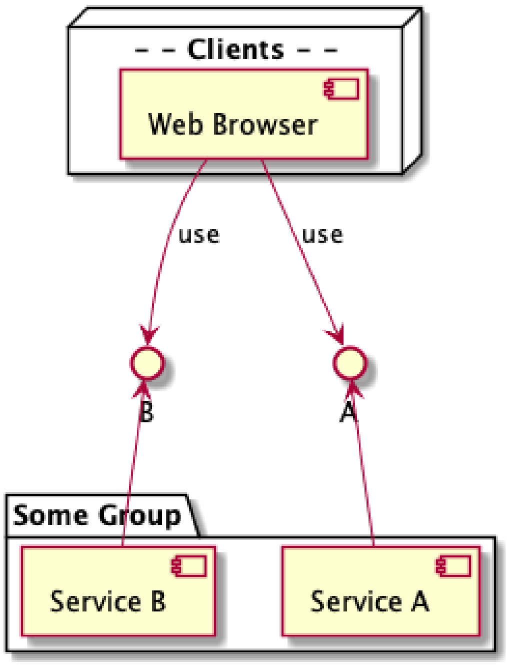


Figure 7.15.: Before Introduce API Gateway

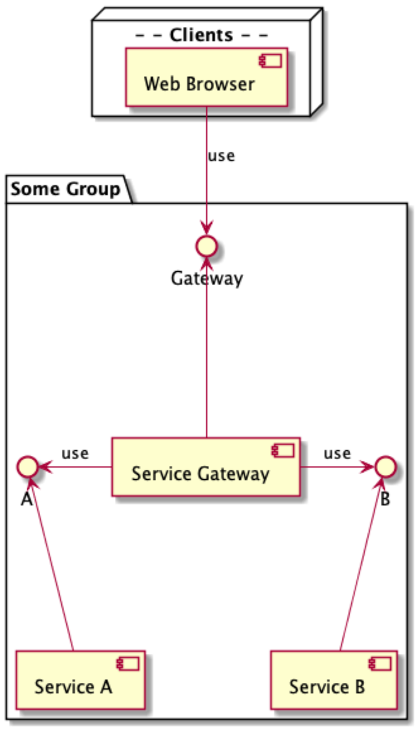


Figure 7.16.: After Introduce API Gateway

8. Evaluation

Contents

The chapter presents evaluation of the thesis results. The following sections demonstrate the prepared questionnaire, derived results and the conclusions made.

8.1. Questionnaire

In the chapter 5 it was defined to perform evaluation for the research results using questionnaire. We also discussed rough structure of it. In the section we demonstrate prepared questionnaire in details.

The structure is formed of three main sections. These are Background, Microservices expertise, Approach, and Refactorings Catalog sections.

8.1.1. Background

The section is meant to better understand the interviewee's background. It helps later during results analysis phase. Based on the answers it is easier to assess what weight (multiplier) should be applied to answers of the person on specific questions. For example, people on research positions might assess the approach with more expertise. At the same time, answers of the interviewees holding the post that involve much practical experience will be weighted more for refactoring techniques application questions.

The section consists of questions about experience in software engineering. Specifically, the amount of years in software engineering and current position are in question. The current position question is of multiple choice type. There are two reasons for it. First, it happens that worker has several responsibilities. Therefore, interviewee can associate themselves with multiple positions. Secondly, among the options there are some that can be combined. For example, full-stack developer and mobile developer. Therefore, it is useful to provide this opportunity to choose several options and also gather the more specific information.

8.1.2. Microservices Expertise

The questions in the section are meant to understand what smells interviewee has met before. They also help to discover interviewees' experience with microservices.

To assess the validity of the main results of the thesis - refactoring techniques applicability - it is necessary to know what is the microservices architectural style in depth. Especially, it is required to know what disadvantages are common for systems that adopt the style. The problems are visible on the surface through microservices-specific bad smells. While the bad smells are not widespread and still much research going on - the problems exist for quite some time already. Therefore, we present the smells with description of underlying problem in the section. The smells are used later in the survey in questions about refactoring catalog, specifically, what smells and to what degree the presented technique can tackle. The disadvantages come with decreased quality that refactoring techniques are meant to increase. In the fourth section there will also be questions about affected quality factors by proposed refactoring.

8.1.3. Approach

In the section we ask interviewees to assess the chosen approach (section 5.1) to build the refactoring catalog. We ask interviewees to assess the validity and also ask for a comment. The main importance in our case has the reliability of the chosen approach, as it is not the main result of the thesis and while we want it to be repeatable and enable researches to produce the same results with the same initial data, we still need to keep the survey concise and help interviewees to concentrate on the main results.

8.1.4. Refactorings Catalog

The section is the most valuable - it is meant to assess the main results of the master thesis. The section has its own structure. In the beginning there is a catalog overview. It mentions all 8 refactorings and presents the questions structure for each technique.

Every subsection dedicated to one refactoring presents the technique in the beginning. It consists of summary, mechanics and example sections from the catalog. Afterwards, the smells the refactoring helps to tackle are listed. For each smell it is asked to estimate how applicable the refactoring is. Then, the next question asks how the proposed technique affects listed quality criterion. The last subsection gives an interviewee an opportunity to leave comments.

8.2. Results

During the survey period we were able to attract 116 respondents having more than 500 clicks in total. We filtered out the cases and found 36 useful ones. We defined the validation criteria based on the fact that respondent reached the section of refactorings evaluation giving answers. In the section we demonstrate results of the questionnaire. We start with analysis of respondents' background. Then, we present and discuss evaluation of the approach. Afterwards, we show how the proposed refactoring techniques were evaluated. In the discussion section, different perspectives on the refactorings evaluation are be given and given an analysis of free input answers. There, we also discuss stratification where applicable.

8.2.1. Participants

We reached people with different background. We received feedback from 26 people from industry, 18 people of them are from development positions and 8 are involved in management. We also have 6 researchers and 4 students among participants. Worth noticing that all of the student respondents have work experience and have been working in software development between one and three years. We demonstrate this gathered information on participants position with chart 8.1.

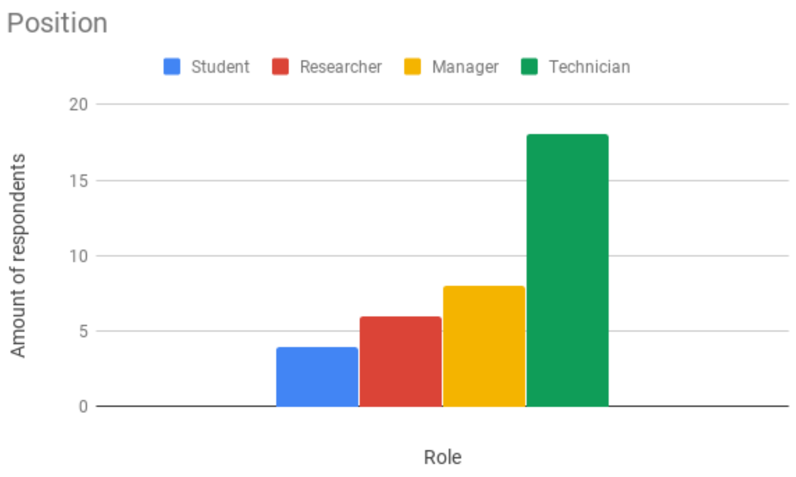


Figure 8.1.: Position

We gathered people with different experience levels. Seven people have been working between one and three years, four people between three and five years. Seven people have work from 5 to 10 years. Sixteen respondents have work experience more thant ten years. This data is presented on the chart 8.2.

Exactly 50 percents of the respondents have practical experience with microservices. Other either investigated on the topic themselves or developed toy programs for different

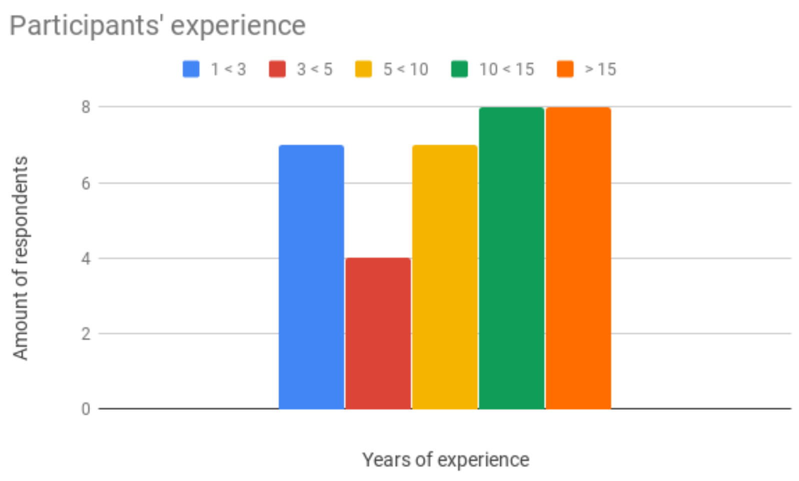


Figure 8.2.: Participant's Experience

microservices concepts. The difference in experience with microservices is demonstrated on the chart 8.3.

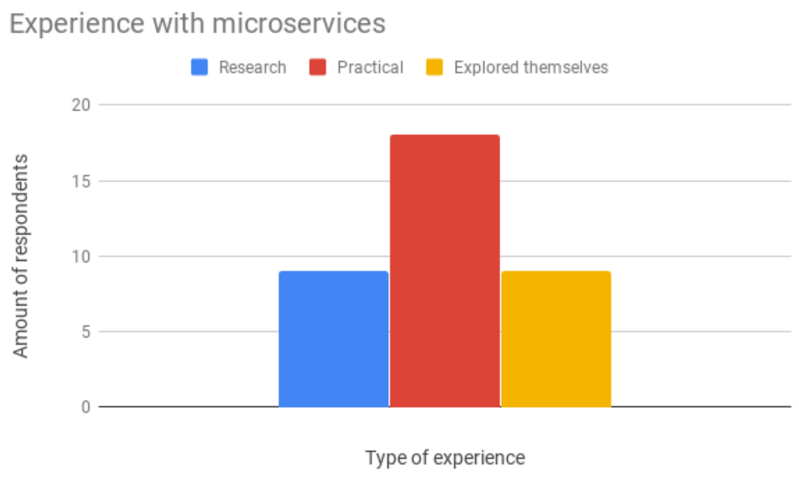


Figure 8.3.: Experience With Microservices

8.2.2. Approach

The approach was evaluated by 33 people. Among them, 15 respondents completely agreed that the approach is valid and promising. Partially agreed with the approach 11 people. It gives us a very good results of almost 80 percents of respondents supporting the approach validity. Only four interviewees have doubts that the approach is good while none of the respondents disagreed with the chosen approach completely. The chart 8.4 presents the information visually.

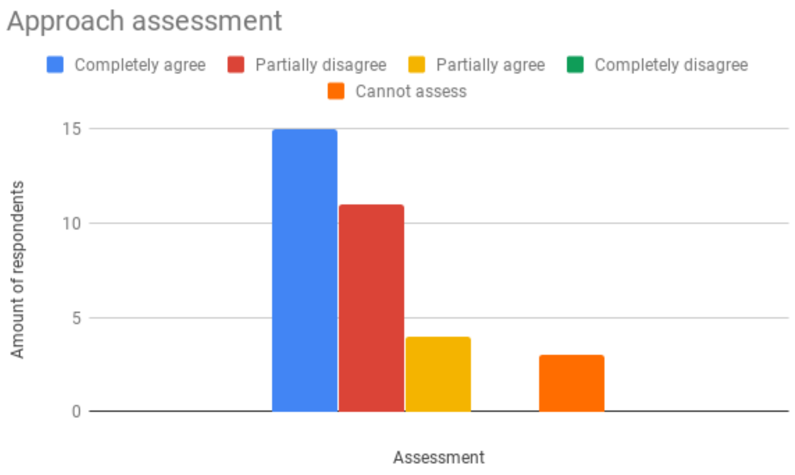


Figure 8.4.: Approach Assessment

Only few of the respondents were not able to asses the approach. This minor number demonstrates that the approach is transparent and understandable to people.

It is crucial for the research that the approach is accepted and valid. It is the basis for master thesis and quality and validity of the approach influences all the following results. It is also an answer for the first research question "How to identify potential refactoring techniques?". Based on the analysis we can make a conclusion that the chosen approach is promising and the results that we gained are reliable.

8.2.3. Refactorings Evaluation

The first question about refactoring was about its applicability to different proposed smells. As for every technique analysis presented later in the section, we start with demonstration of the applicability graph that is box plot with depicted median and weighted average values. The graph shows range where one is the minimum value of applicability and five is the maximum.

The second question asks participants to assess how different quality factors are affected by the proposed refactoring. They can be increased as well as decreased by the refactoring application. As was mentioned in the catalog itself, almost any change in software has its tradeoffs. So, we will see on the graphs that some of the quality factors are decreased or not affected at all by a proposed technique. The results are shown with the same box plot charts with minimum value of -2 (highly decreased) and maximum of 2 (highly decreased). Value zero means that the quality is not affected by the technique.

Encapsulate Responsibility

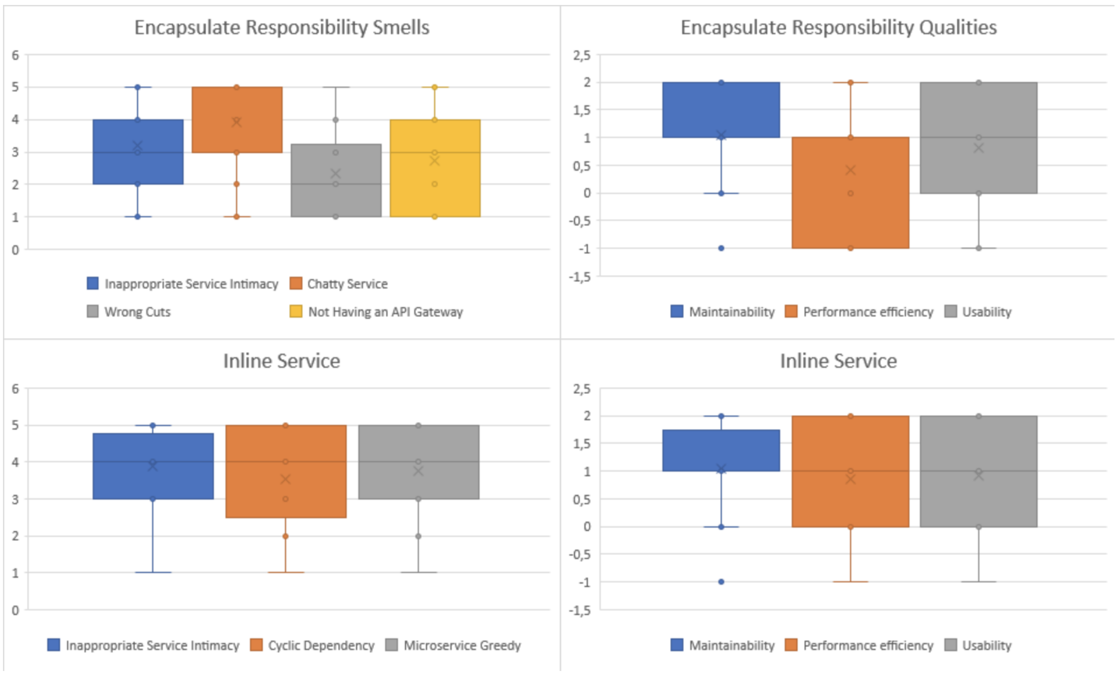


Figure 8.5.: Encapsulate Responsibility Smells, and Qualities. Inline Service Smells and Qualities

As can be seen on the figure 8.5, the proposed refactoring helps to tackle several smells. The most appropriate this refactoring is for tackling smell Chatty Service with median on maximum value - five. The next smell that the proposed refactoring is good for is Inappropriate Service Intimacy with median on three. While median value is the same for smell Not Having an API Gateway - the weighted average is smaller and some people voted that the smell is not applicable at all. Among presented, the refactoring is least applicable to tackle Wrong Cuts.

The chart 8.5 also demonstrates that encapsulate responsibility refactoring increases several quality factors. It increases maintainability the most with median on maximum value - plus two. Usability is also increased considerably by the refactoring application with median on 1. However, there are different perspectives on how it is affected. From the free text input we have the following feedback: "Usability of more fine-grained controls of a service offering might be higher because the service can be controlled on a finer level. However, then a developer/operator/consultant needs to understand each of the single methods instead of the big function.". It is indeed a valid argument. As was mentioned in the catalog, quality factors can be affected differently depending on a concrete case and implementation.

Performance stays the same having median on one and with weighted average of 0.5 with quite a few responses on quality factor decrease. It differs from the assumption made in the refactoring description in the catalog. We claimed that performance efficiency is increased due to less requests from client. On the other hand, time behaviour of this one new heavy request that combines several encapsulated ones will indeed decrease.

Inline Service

Inline service technique is among the best techniques by its applicability to proposed smells based on interviewees opinion. It is equally applicable to Inappropriate Service Intimacy and Microservice Greedy smells with a little less applicability to Cyclic Dependency smell. As we can see on the graph, respondents had quite different opinions on the technique applicability to the Cyclic Dependency smell. We can see that it can be due to the given example in the questionnaire that better demonstrated case for first two microservices smells.

The refactoring generally affects all the quality factors. It increases maintainability, performance efficiency and usability. However, due to the chart, respondents were more sure about increase of maintainability quality factor. During the discussion of the affected qualities by the refactoring in the catalog, we mentioned that the service's usability (specifically - interoperability) can be decreased. However, looking at the system as a whole, usability is increased and participants of the survey also support the assumption.

Extract Service

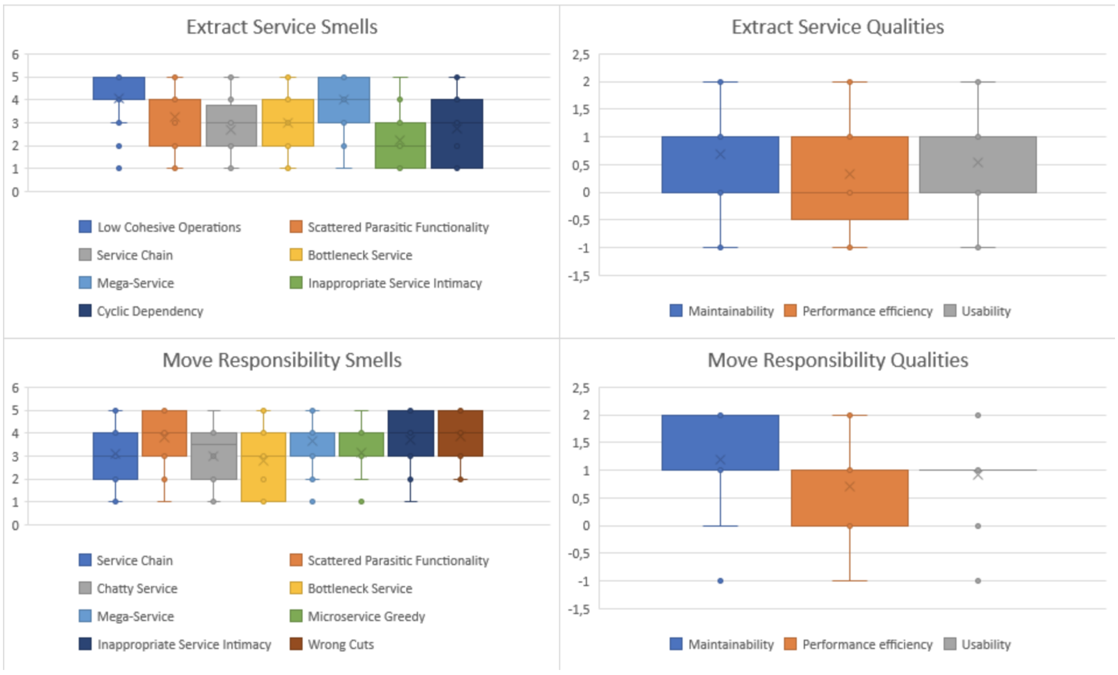


Figure 8.6.: Extract Service Smells, and Qualities. Move Responsibility Smells and Qualities

Extract service in the catalog is proposed to be used to tackle many different smells. As the technique is quite universal, it indeed can be used to solve different problems. However, based on the answers of the survey participants there is a clear separation between smells that can be tackled by the refactoring better and those that cannot be completely solved by the refactoring. We see on the chart 8.6 that the refactoring is not applicable to Inappropriate Service Intimacy at all and barely applicable Bottleneck Service smell. Cyclic dependency is not solved by this smell either but in the catalog it was also presented as only one step towards the complete avoidance of such a dependency. We can also see how different people opinions are on applicability to this smell.

The most applicable the refactoring is to Low Cohesive Operations and all the respondents agreed in that. Mega-Service and Scattered Parasitic Functionality can be tackled by Move Responsibility as well with high median value of four.

The results of the assessment of the quality factors affected by the refactoring application go along with the discussion in the catalog. We mentioned that the performance efficiency is not affected or can even be decreased by the refactoring application. we see that people agreed having median value of zero. We also have one opinion supporting the discussion in the catalog that was taken from analysis of free text input "Performance: Now that the service/method is isolated its performance can be tackled independently from the original service. This MIGHT increase the performance but not automatically does". On the graph 8.6 we see that none of the factors have a considerable increase. The most affected quality factor is maintainability based on the average value.

Move Responsibility

Move responsibility is the most general refactoring technique proposed in the catalog. Therefore, there are so many smells proposed that the technique could potentially tackle. We see a split into two groups on the chart 8.6. The first group consists of the smells Scattered Parasitic Functionality, Mega-Service, Inappropriate Service Intimacy and Wrong Cuts. Participants assessed the refactoring to be most applicable to those smells with median value four. The refactoring regarding its applicability to other four smells was assessed as less applicable, though still not inapplicable at all. The most confusion among participants was about smell Bottleneck Service. We see that some people have opinion that Move Responsibility is not applicable to this smell at all.

On the same chart 8.6 we see that maintainability is the main quality factor affected by the technique. It supports the assumption made in the catalog. All of the three qualities are increased with the refactoring application while the performance efficiency is affected the least. We can observe how sure were people about moderate increase of usability with the technique. It was also mentioned in the refactoring description that having improved time behaviour of the system is just a side effect and not the main purpose of the technique.

Analysis of the comments to the refactoring brought us valuable insight. It was highlighted that "it may require additional work [to perform the refactoring] if services use different languages or frameworks". It is indeed an important point to consider and another prerequirement for the technique.

Remove Middle Service

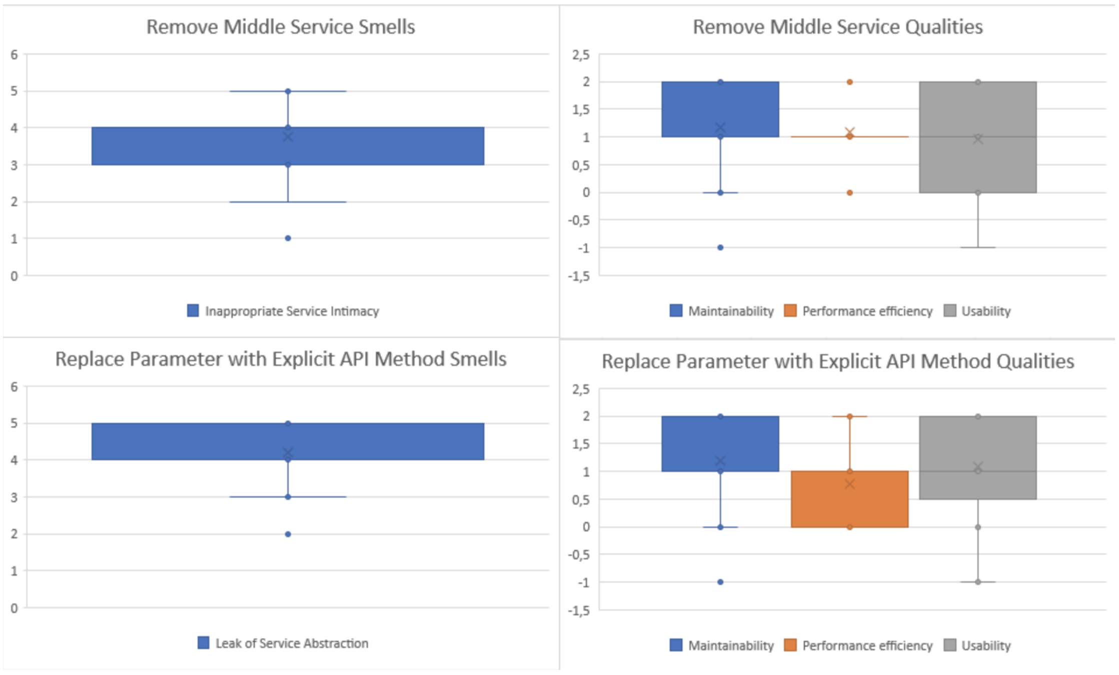


Figure 8.7.: Remove Middle Service Smells, and Qualities. Replace Parameter with Explicit API Method Smells and Qualities

Remove middle service is a very specific technique. We can see the results on its applicability and affected quality on figure 8.7 Only one smells is meant to be tackled with this refactoring. Based on the results that we received, people from industry and research support its applicability to Inappropriate Service Intimacy smell with median value four.

The refactoring affects quality factors a lot. The performance is increased due to less communication in the system as was discussed in the catalog. We see that all of the respondents were sure about moderate increase with value one. Due to the fact of having less useless services we see that participants of the survey approve that maintainability and usability are also improved. However, respondents were quite unsure about usability. There are interviewees who believe that usability is not affected by the technique application. Nevertheless, almost none of the people chose the option of decreasing any of the quality factors.

Replace Parameter with Explicit API Method

The technique is meant to tackle only one smell - leak of service abstraction. We got approval of the assumption that the refactoring is indeed applicable to solve this specific smell. People are sure, that is visible on the graph, that the technique is highly applicable.

Replace Parameter with Explicit API Method increases or slightly increases all the presented quality factors. It considerably increases maintainability and usability. While we got the same median values we see that people were more sure about increase in maintainability of the system. However, in the catalog only usability was mentioned as the main quality factor that could potentially force people to use the refactoring. Increased performance efficiency is positive side effect in this case. As can be seen on the graph people voted for the quality factor being unaffected or only slightly increased.

Require Data for API Method

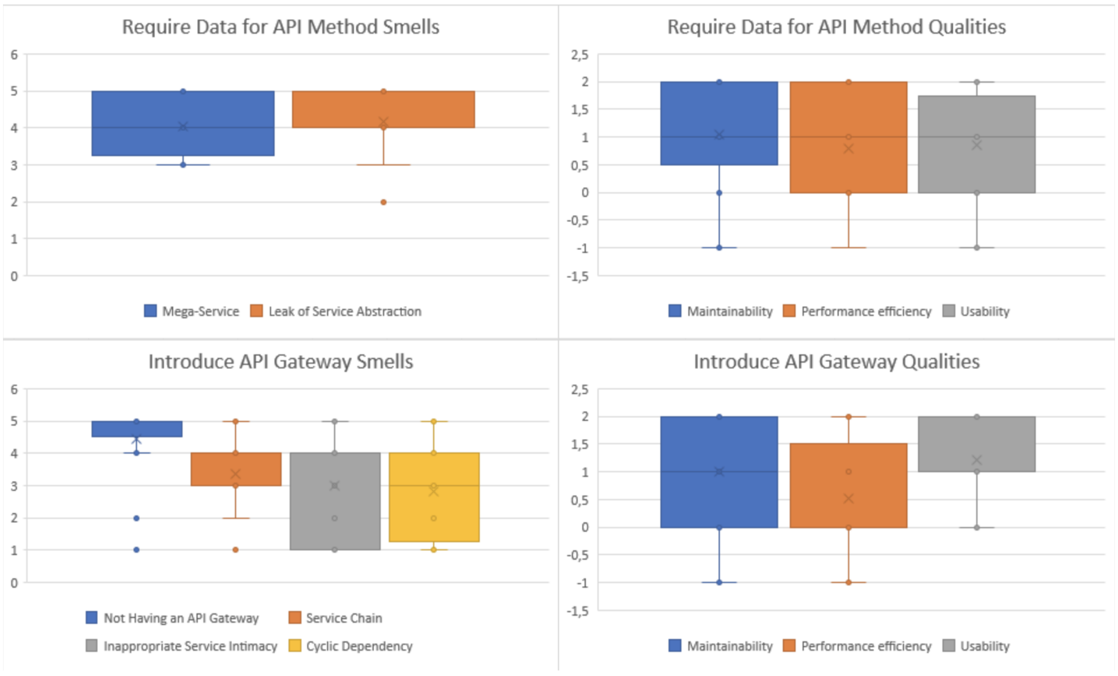


Figure 8.8.: Require Data for API Method Smells, and Qualities. Introduce API Gateway Smells and Qualities

On the chart 8.8 we see that the refactoring was assessed by its applicability to Mega-Service and Leak of Service Abstraction smells. We see that the technique is equally good applicable to tackle both smells. Even though the median values are the same, we see that respondents were not that sure about Mega-Service. There are quite a few respondents that belive that the refactoring is barely applicable to the smell.

While both of the smells are important to get rid of, but tackling mega-service would give more benefits. It is not expectable to see how good this refactoring is applicable to solve such an important smell while it was mostly derived to solve Leak of Service Abstraction. And it is slightly more applicable to this smell as can be seen on the box plot.

The refactoring increases maintainability of the service. It also leads to moderate increase of usability. Performance efficiency is not mentioned in the catalog. While we have a median value of one, there are still many respondents that believe the refactoring is not affecting performance efficiency. Nevertheless, having opinion with slight increase of the quality factor on average, it be considered as a positive side effect of refactoring but not the main driver of its application.

Introduce API Gateway

The refactoring technique has a clear purpose to introduce an API gateway to a system. The results depicted on the figure 8.8 are also not surprising. The technique is the most applicable to the corresponding smell - Not Having an API Gateway. Almost all of the respondents were positive about it and we have a maximum median value of five. However, based on the free text answers analysis to this refactoring we noticed that some people misunderstood the concept. The comments include a statement that it is not a good idea to implement a custom API Gateway. However, as any traditional refactoring, the technique does not provide a concrete solution and there is no advice given on implementation. So, it is completely fine to use existing solution for API Gateway that would still mean following the refactoring.

Respondents agreed that the technique considerably increases usability of the system. The graph 8.8 demonstrates that maintainability of the system is also increased. Based on respondents opinion, the performance efficiency slightly increased or not affected by the technique application. In the catalog we discussed that the benefits in building a response for the user in an API-gateway will overweight problems related to introducing a service in between of the user and the rest of the system.

The analysis of the comments gives the following opinion on affecting maintainability that has different perspective and therefore different implication "every change in each service needs to be reflected in the gateway which decreases maintainability and development speed.".

8.2.4. Refactorings to Tackle Smells

There is a different perspective on the gained results. One of this perspective is effectiveness of the proposed refactorings for each smell. This perspective is crucial, as software refactoring has an intention of paying technical debt. Technical dept can be detected with smells as was discussed in section 2.2. While the main purpose of the questionnaire is to assess proposed refactorings applicability, we can still have a good assessment and comparison of their applicability to each involved smell.

In the table presented on the figure 8.9, we see perspective of the refactorings applicability to the smells. Colors define how applicable they are and the number is actual applicability value.

We mark the highest applicability with dark green color, good applicability with light green, weak applicability with yellow and no applicability with dark orange.

On the figure 8.9 we see that top three most applicable refactorings to tackle Inappropriate Service Intimacy smell are inline service, Remove Middle Service and Move

Smells Perspective								
Smell/Refactoring	Encapsulate Resp.	Inline Service	Extract Service	Move Resp.	Remove Middle	Replace Param	Require Data	Introduce Gateway
Mega-Service			2	2			2	
Microservice Greedy		2		3				
Inapp. Service Intimacy	3	2	4	2	2			2
Wrong Cuts	4			2				
Leak of Service Abstr.						2	2	
No API Gateway	3							1
Service Chain			3	3				2
Cyclic Dependency		2	3					3
Chatty Service	1			2				
Low Cohesive Oper.			2					
Scattered Parasitic Func			2	2				
Bottleneck Service			3	3				

🟩 - Highly applicable
▭ - Applicable
🟨 - Barely applicable
🟧 - Not applicable

Figure 8.9.: Smells Perspective

Responsibility techniques. While inlining a service and moving responsibility are quite general techniques and have much in common, Remove middle man is specific to only one case. The least applicable refactoring to solve the mentioned issue is extract service.

Comparison of refactoring techniques for Chatty Service shows that encapsulate responsibility technique is considered to be more applicable to solve the issue than move responsibility.

For wrong cuts smell there is a clear separation in respondents opinion on high applicability of encapsulate responsibility technique and only little applicability of move responsibility to tackle the smell.

On the figure 8.9 we see that there is only one refactoring in the catalog that claims to tackle low cohesive operation smell. While there is no comparison to other refactoring, it is seen with green color that this technique helps to solve the issue effectively. The next smell - cyclic dependency has three different refactorings claimed to tackle it. The most applicable refactoring is inline service. Two other techniques - extract service and introduce api gateway are much less applicable having similar level of relevance.

We can immediately see a clear leader for Not Having an API Gateway smell - introduce an api gateway. There is a less clear picture for Service Chain smell, however. All the three smells are not highly applicable to the smell with better results for introduce api gateway, but no clear leader. The smell is complex and refactorings were derived indirectly in the catalog. We see that we could not find a specific technique that can tackle this smell effectively.

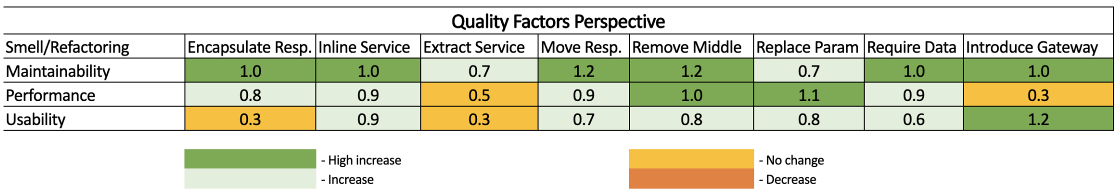
Scattered parasitic functionality is solved better by move responsibility refactoring as we can see in the table. Leak of service abstraction can be effectively solved by both proposed refactorings - replace parameter with explicit API method and require data for API method. It is an interesting case because both of the techniques work on the same level - level of service interface. Such changes are less complex and can be performed with less effort.

We can see that inline service can solve the Microservice Greedy smell effectively, while Move Responsibility is less effective. For the Bottleneck Service there is no good technique found. We can see that both of the refactorings have only yellow color that means weak applicability. This can be due to the the fact that this smell is common and often necessary evil with its tradeoffs. The most obvious example is pattern API gateway. While API gateway has all the issues of bottleneck service smell, it provides many important benefits.

Applicability of the proposed refactorings to Mega Service smell is quite sound. Extract service and require data for API method refactoring are equally good applicable to the smell. While move responsibility technique is less applicable - it still has sound results to be considered as one of the techniques for the problem solution.

8.2.5. Refactorings to Improve Quality

Another perspective that we find useful to assess is effectiveness of the refactorings to improve quality factors. There are only three qualities that were present to be assessed in the questionnaire for each technique - maintainability, performance efficiency and usability. We have a look at each of the quality factor and all the refactorings. We can see how good or bad each of the technique helps to improve the quality.



Quality Factors Perspective

Smell/Refactoring	Encapsulate Resp.	Inline Service	Extract Service	Move Resp.	Remove Middle	Replace Param	Require Data	Introduce Gateway
Maintainability	1.0	1.0	0.7	1.2	1.2	0.7	1.0	1.0
Performance	0.8	0.9	0.5	0.9	1.0	1.1	0.9	0.3
Usability	0.3	0.9	0.3	0.7	0.8	0.8	0.6	1.2

🟩 - High increase
▨ - Increase
🟨 - No change
🟧 - Decrease

Figure 8.10.: Quality Factors Perspective

We use weighted average value for presenting quality factors matrix 8.10. We mark great quality improvements (over 1.0) with dark green, weak improvement with light green and minimum change in quality with yellow.

We will first look at maintainability quality factor. Maintainability is the most important factor that is often the main driver to perform refactoring. On the figure 8.10 we that there all the eight refactorings influence maintainability positively. We can see than the quality factor is improved the most by refactorings remove middle service and move responsibility. Indeed, as wan mentioned in the catalog, move responsibility is very effective technique that, applied wisely, leads to great maintainability improvements.

The least effective refactorings in terms of improvement system's maintainability are extract service and replace parameter with explicit API method. The two techniques are very different. In case of extract service - maintainability can be decreased on operation level. We had comments on the technique in the survey results about having more communication between teams and obligation to support one additional service that indeed decreases maintainability that can decrease system's maintainability. As was mentioned in the catalog, maintainability of the extracted service will be increased, though. It might be still useful to use the refactoring for important services. Other refactorings mentioned on the graph moderately improve maintainability approximately to the same degree.

The next quality factor is performance efficiency. The results are also demonstrated on figure 8.10. We see clearly that there are two refactorings that do not affect or can even decrease performance of the system - introduce API gateway and extract service. It is obvious why extract service can decrease the performance but for introduce api gateway technique the situation is more complex. As we discussed in the refactoring results analysis subsection, performance could potentially be increased by wisely combining several requests into one.

We see that other refactoring techniques slightly improve the quality factor. Replace parameter with explicit API method is leader on the chart, though the results might be inadequate due to questionnaire problems described later in the discussion section. Remove middle service, extract service and move responsibility can be named as the most effective technique to increase performance.

Usability is the last quality factor that was presented to participants.It is complex to assess the quality as there are different perspectives. There is service usability that some of the respondents had in mind due to the comments and also system usability. Top three of proposed refactorings that improve the quality factor the most are introduce API gateway, inline service and remove middle service. We see two refactorings that do not affect or can even decrease usability. Those are encapsulate responsibility and extract service. Having inline service as refactoring improving usability and extract service as one that does not affect or decreases it, we can imply that it is easier for developers to use bigger services that serve many different requests. We also see that refactorings that have specific goal of increasing usability, as replace parameter with explicit API method is not considered that effective and radical changes in arrangement of services.

8.3. Discussion

Conducting a survey we tried to attract people with different occupation. We also reached people with different levels of experience in their career. As we discussed in the questionnaire description, the main reason for it is to receive opinions from different perspectives. It might be interesting to analyse how different were the results given by people with different background.

On the figure 8.11 we demonstrate respondents ratio within each occupation groups evaluating techniques applicability. On the axis x we give different options of applicability level that interviewees could chose and on the axis y there is percentage of people within each group that chose the option. Options of technique applicability are ranging from 1 - not applicable to 5 - highly applicable. We can see that respondents from different stratas show approximately the same evaluation ratio in terms of refactorings applicability. There are a little more people having management positions that chose highest applicability level to smells for presented techniques and also more people that gave applicability level four among students.

The next groups that we derived have different experience level. On the chart 8.12 we see that we have 5 different groups from those having one to three years of experience

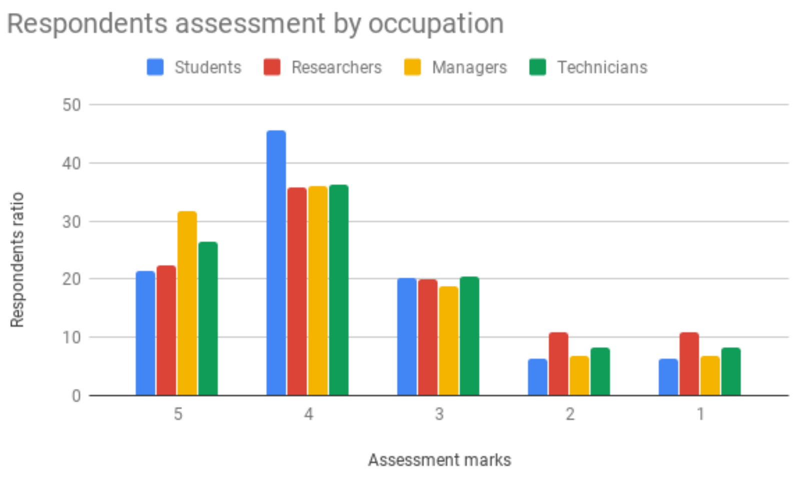


Figure 8.11.: Respondents Assessment by Occupation

to those people that have been working for more than 15 years. While we provided an option with less that one year of experience we did not reach any people within this group.

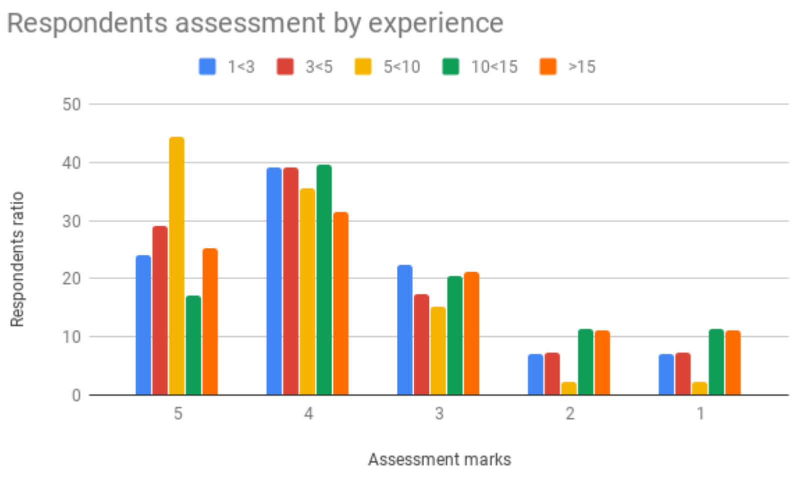


Figure 8.12.: Respondents Assessment by Experience

Responses received from group exclusively chosen with specific background might lead to bias results. We do not have this situation as we were able to reach different groups. Moreover, their analysis and comparison shows that the responses received from each of the groups are not radically different.

Besides not having variety among respondents problems can arise because of the questionnaire structure and content. We noticed tendency of people giving better assessment to refactorings that are meant to tackle ony one or two smells. It can be related to the fact that assessing a refactoring's applicability to one smell - interviewee mentally creates a concept that is specific to tackling this smell with the proposed refactoring. Asking the participant to assess the technique's applicability to further smells we risk to receive bias results because they already have a picture of the refactoring aiming to solve the first smell presented.

The same problem might arise due to a different reason that we consider as a more valuable factor. For any refactoring with very limited (one or two) smells presented that it is meant to tackle we were able to give a more specific example for smells, while for refactorings presented with several smells - only one general example was given that cannot demonstrate technique's applicability to every smell equally good.

The last but not least aspect that influences the questionnaire quality of the smells used. While microservices architectural style is still fresh - the smells are not yet formed and the catalog that we used cannot be proven to be fully valid. It is also a valid point that those smells are not common and still not known by most of the participants.

8.4. Summary

In the chapter we presented a method that we used for evaluation of the research results. The first section demonstrated the developed questionnaire. We described questions that were asked to respondents. We were concerned about their current experience with microservices architecture style. We also presented how we were trying to reach different types of interviewees - with varied background and occupation.

In the results section we discussed what background our actual participants have, what experience do they have with microservices and how many years they are working in software engineering. Then, we demonstrated results on the approach that prove results validity and reliability. Afterwards, we described results on each of the refactoring presented. We included graphs with applicability of each technique to smells and degree with which the technique affects quality factors. In the refactorings to tackle smells and refactorings to improve quality we presented different perspectives on the techniques assessment. We looked at the most effective and least effective refactorings to tackle each smell and to improve each quality factor. The last subsection - discussion - revealed potential problems of the questionnaire that might lead to bias results towards some of the presented refactorings. We deduced that variety of the respondents did not lead to bias responses, while structure of the survey might have introduced partially unfair responses.

9. Conclusion & Future Work

Contents

9.1. Conclusion

This thesis presented a catalog of refactorings for systems with microservices architectural style. We organized the work starting from background. The background is meant to make reader familiar with all the basis that we built our work on. We presented the main points of microservices architecture style, its main characteristics, benefits and tradeoffs. Then, we discussed a concept of technical debt together with bad smells, as one of the crucial parts for the further created concept. We also discussed refactoring and its importance in software development.

After the background was presented (chapter 2), we defined problem statement (chapter 3). We defined three challenges - selection, description and evaluation of refactorings. We described them in the section to give better understanding of the thesis scope.

The following chapter (chapter 4) presented related work. The main part in the chapter is the one with microservice smells catalog. It is also part of the further work where we used the smells extensively. The second section of the chapter describes microservices patterns and practices used in the research.

The next chapter is called concept (chapter 5). We defined the whole process for building the catalog. Afterwards, we defined a technique's description template. We also discussed initial refactoring selection process where three different approaches were presented and one of them was chosen after analysis. The last step in the concept was choosing a way of evaluation.

Chapter Initial Refactoring Selection (chapter 6) contains results of performing the process defined in concept. We performed matching between existing bad smells and microservices smells. Then, we demonstrate discovering of the refactoring based on the matching. The results were presented in tables.

Chapter Refactoring Catalog (chapter 7) contains the main results of the thesis. After the selection process was done, in the chapter we described all the refactorings following the template defined in the concept chapter. Each refactoring was presented with an example.

In the last chapter - evaluation (chapter 8) - we discussed the last step of the process.

To speak of validity and applicability of the refactorings in the catalog, we conducted a questionnaire as evaluation method. The first part of the chapter describes the created questionnaire. The following section describes the results collected. We demonstrated their analysis and attached charts with statistical information. Besides results on the catalog, we analysed respondents' opinion on chosen approach as well as deduced rating of the techniques assessed from different perspectives. In the last section we presented discussion of the questionnaire potential problems that might have led to bias results for some of the techniques.

Defined research questions were answered in the thesis. We also reached the goal of building a valuable and reliable catalog of refactoring techniques for microservices systems. We found that the results of the thesis are applicable in practice through evaluation. We also reached a concise description of each technique that contains all the necessary information and easy to follow. The catalog is technology agnostic and can be used by developers in practice.

9.2. Future Work

The topic of refactoring of microservices is still barely discovered. Microservices architectural style is relatively new in software engineering. We made the first steps in building a catalog of refactoring techniques. The results appeared to be valuable and applicable due to analysis of the evaluation.

There are already two research works on discovering microservices smells "On the Definition of Microservice Bad Smells" [TL18] and "Towards a Collaborative Repository for the Documentation of Service-Based Antipatterns and Bad Smells" [Bog+19] that were used heavily in the thesis. There is still much to do in discovering new microservices smells. There are different perspectives that can be useful to separate and organize in separate catalogs. We noticed that there is still very little separation between smells in services and in the microservice system. The separation, however, is very important as it is required to use different techniques to tackle those smells. There is still a room for improvement in differentiating between microservices smells and other service-oriented architecture smells. Some of them are the same, while others are different because of the different approaches in organizing the services, their communication and work together.

The catalog itself can be extended by deducing refactorings using different than in the thesis approaches. With time there is a possibility to extend the catalog with already used and proven new techniques for microservices instead of building on existing ones.

One of the interesting aspects for practical use in industry in the topic is automation. There are existing tools that help to detect smells. We can see potential in building such tools for microservices smells. The analysis might be not that easy to perform, as it will require monitoring of running system. However, as monitoring is quite common for microservices systems and the architectural style gains only more attention, there would be a great benefit in having such tools.

Besides detecting smells, we can also envision tools that help to refactor microservices systems following described in the catalog techniques. This might be a bigger problem than detecting smells and for sure more difficult to implement than automation of traditional techniques such as move method and extract class. Nevertheless, we see potential in the automation that would help developers in industry.

Aside of the refactorings and smells we discussed another important aspect in the thesis - how the techniques affect quality of the system. Until now there is no commonly used quality model specifically for microservices systems. It is a complex problem, as there are two different perspectives - quality of the system in general and quality of each service that should also be considered. We had examples of refactoring techniques that led to increased quality factor for system but in the same time it decreases quality of the service under refactoring.

We also see potential changes in refactoring process. It is common for proposed refactorings to involve several services. While different services are maintained by different teams - we assume that refactoring process might include more operational and organizational work. However, the refactoring process will be changes only on the system level. Changes in each service might still be performed following existing process.

Testing as a crucial part of refactoring is affected a lot by the architectural style. While unit testing of units in each service stays the same, integration testing gets more complicated. It becomes more difficult to isolate only parts of the system that are under testing. Testing process on the system level might also change as it would involve several teams.

A. Bad Smells Aliases

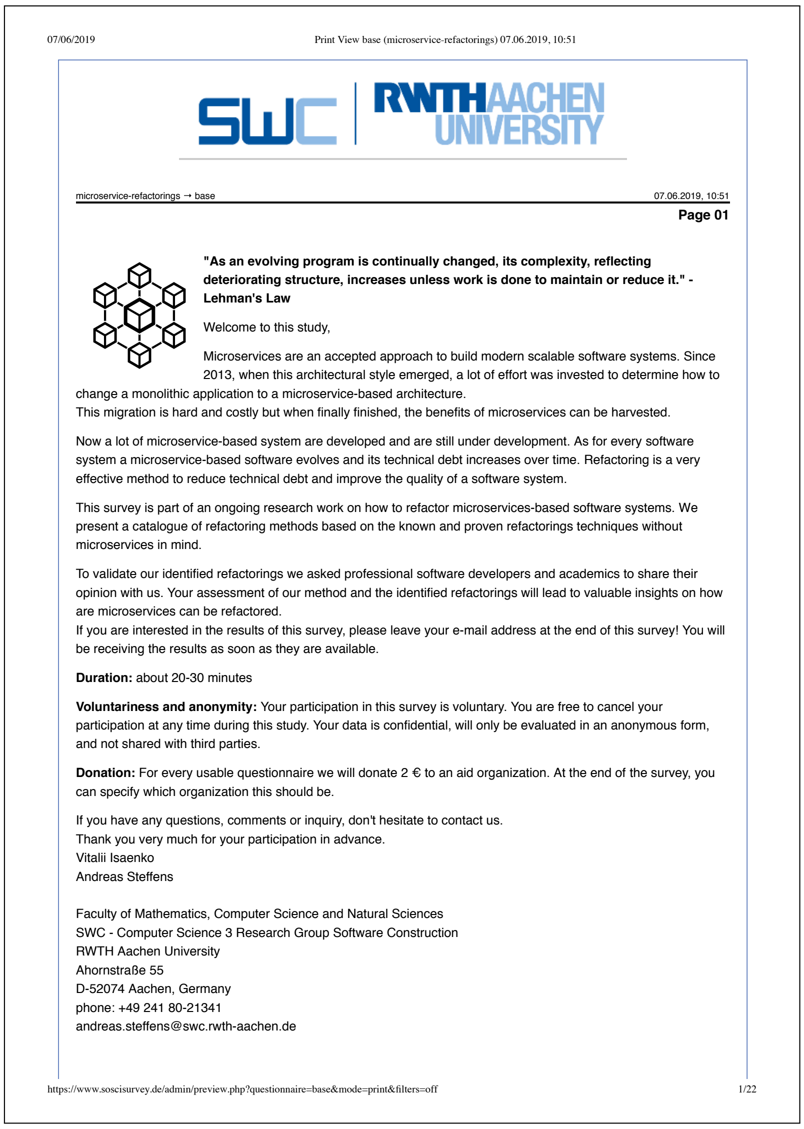
Bad Smells Aliases	
Bad Smell	Aliases
Cyclically-dependent Modularization [SSS14]	Dependency cycles [M.07] Cyclic dependencies [PJ88] Cycles [BL03] Bidirectional relation [CU06] Cyclic class relationships [MPC99]
Deficient Encapsulation	Hideable public attributes/methods [Son] Unencapsulated class [CU06] Class with unparametrerized methods [CU06]
Unnecessary Abstraction [SSS14]	Irrelevant class [Uml] Lazy Class [Fow99] Freeloader [Kho+12] Small class [JR92] [CU06] Mini-class [SSM06] No responsibility [Bud91] Agent class [Uml]
Deficient Encapsulation [SSS14]	Hideable public attributes/methods [Son] Unencapsulated class [CU06] Class with unparametrerized methods [CU06]
Multifaceted Abstraction [SSS14]	Divergent Change [Fow99] Conceptualization abuse [Tri05] Large class [Mey88] [Fow99] [CU06] [Moh+10] Lack of Cohesion [Sem]
Insufficient Modularization [SSS14]	God class [Rie96] Fat interface [Mar03] Blob class [Inf] Classes with complex control flow [Sem] Too much responsibility [Bud91] Local breakable [Str]
Change Function Declaration [Fow19]	Rename Function [Fow19]

Bad Smell	Aliases
	Rename Method [Fow99] Add Parameter [Fea04] Remove Parameter [Fow19] [Ref] Change Signature [Fow19]
Replace Param with Explicit Methods [Fow19]	Remove Flag Argument [Fow19]
Broken Modularization [SSS14]	Feature Envy [Fow99] Class passively stores data [Sem] Data Class [Fow99] [Mar03] [CU06] [Inf] Data records [Str] Record (class) [Moh+10] Data Container [RFG05] Misplaced operations [DDN02] Misplaced control [Tri05]
Hub-like Modularization [SSS14]	Bottlenecks [Son] [PJ88] Local hubs [Str] Man-in-the-middle [RFG05]

B. Microservices Smells Aliases

Microservices Smells Aliases	
Microservices Smell	Aliases
Leak of Service Abstraction [TL18]	Ambiguous Interface [Onl]
Mega-Service [TL18]	Bloated Service [Onl]
Low Cohesive Operations [TL18]	Bloated Service [Onl]
Microservice Greedy [TL18]	Nanoservices [Onl] Golden Hammer [Onl] Silver Bullet [Onl] When in doubt, make it a service [Onl] Tiny Service [Onl] Refactor Mercilessly [Onl] Fine Grained Web Service [Onl] Fine-Grained Services [Onl] Fine-Grained Interfaces [Onl] Web Services Will Fix Our Problems [Onl]
Sloth [TL18]	The Knot [Onl]
Shared Persistency [TL18]	Data ownership [TL18] Data-Driven Migration [Onl] Shared Persistence [TL18]
Ambiguous Service [Onl]	Ambiguous Web Service [Onl] Ambiguous Name [Onl]
API Versioning [TL18]	Static Contract Pitfall [TL18]
Business Process Forever [Onl]	No Businessman Involvement [Onl]
Chatty Service [Onl]	Chatty Web Service [Onl]
Hardcoded Endpoints [TL18]	Hardcoded IPs and Ports [Onl] Point to Point Web Services [Onl]
Mega-Service [TL18]	Multiservice [Onl] The God Object [Onl] God Object Web Service [Onl] Big Ball of Mud [Onl]
Service Chain [Onl]	Percolating Process [Onl]
Too Many Standards [TL18]	Lust and Gluttony bad practices [TL18]
Timeout [TL18]	Dogpiles [TL18]

C. Questionnaire



07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

microservice-refactorings → base 07.06.2019, 10:51

Page 01

"As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it." - Lehman's Law

Welcome to this study,

Microservices are an accepted approach to build modern scalable software systems. Since 2013, when this architectural style emerged, a lot of effort was invested to determine how to change a monolithic application to a microservice-based architecture.
This migration is hard and costly but when finally finished, the benefits of microservices can be harvested.

Now a lot of microservice-based system are developed and are still under development. As for every software system a microservice-based software evolves and its technical debt increases over time. Refactoring is a very effective method to reduce technical debt and improve the quality of a software system.

This survey is part of an ongoing research work on how to refactor microservices-based software systems. We present a catalogue of refactoring methods based on the known and proven refactorings techniques without microservices in mind.

To validate our identified refactorings we asked professional software developers and academics to share their opinion with us. Your assessment of our method and the identified refactorings will lead to valuable insights on how are microservices can be refactored.
If you are interested in the results of this survey, please leave your e-mail address at the end of this survey! You will be receiving the results as soon as they are available.

Duration: about 20-30 minutes

Voluntariness and anonymity: Your participation in this survey is voluntary. You are free to cancel your participation at any time during this study. Your data is confidential, will only be evaluated in an anonymous form, and not shared with third parties.

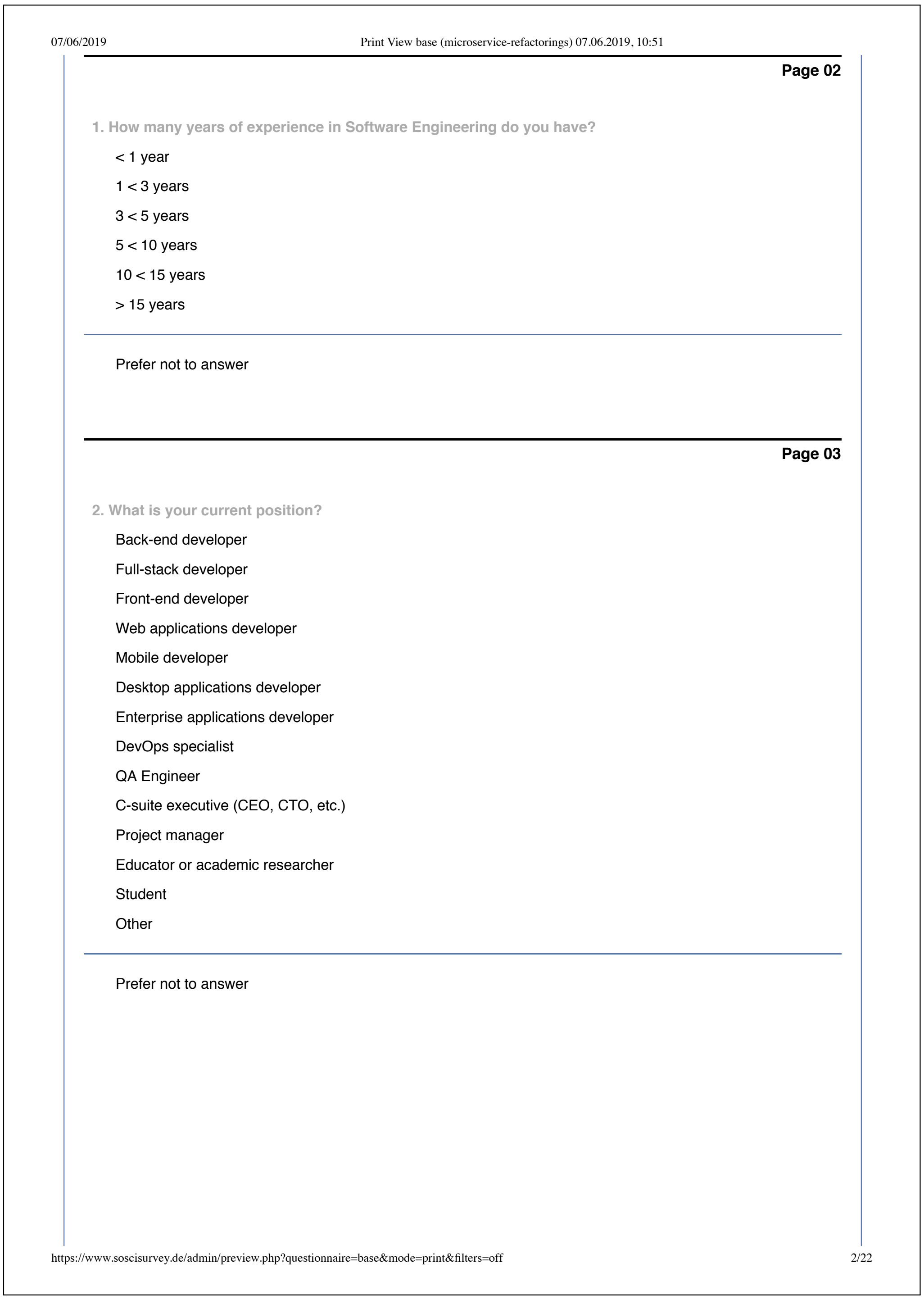
Donation: For every usable questionnaire we will donate 2 € to an aid organization. At the end of the survey, you can specify which organization this should be.

If you have any questions, comments or inquiry, don't hesitate to contact us.
Thank you very much for your participation in advance.
Vitalii Isaenko
Andreas Steffens

Faculty of Mathematics, Computer Science and Natural Sciences
SWC - Computer Science 3 Research Group Software Construction
RWTH Aachen University
Ahornstraße 55
D-52074 Aachen, Germany
phone: +49 241 80-21341
andreas.steffens@swc.rwth-aachen.de

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 1/22

Figure C.1.: Questionnaire page 1 of 22



07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

Page 02

1. How many years of experience in Software Engineering do you have?

< 1 year

1 < 3 years

3 < 5 years

5 < 10 years

10 < 15 years

> 15 years

Prefer not to answer

Page 03

2. What is your current position?

Back-end developer

Full-stack developer

Front-end developer

Web applications developer

Mobile developer

Desktop applications developer

Enterprise applications developer

DevOps specialist

QA Engineer

C-suite executive (CEO, CTO, etc.)

Project manager

Educator or academic researcher

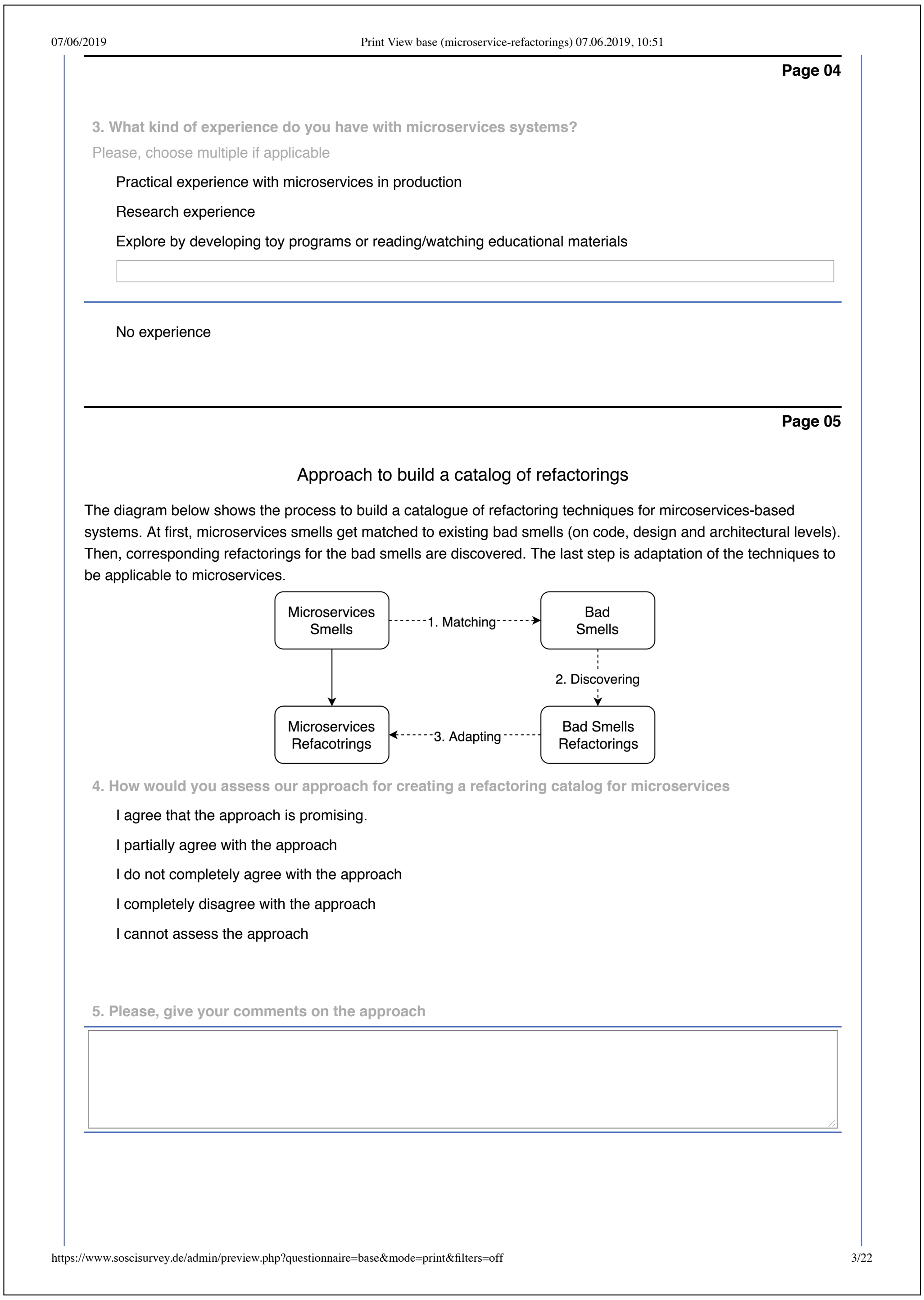
Student

Other

Prefer not to answer

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 2/22

Figure C.2.: Questionnaire page 2 of 22



07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

Page 04

3. What kind of experience do you have with microservices systems?

Please, choose multiple if applicable

Practical experience with microservices in production

Research experience

Explore by developing toy programs or reading/watching educational materials

No experience

Page 05

Approach to build a catalog of refactorings

The diagram below shows the process to build a catalogue of refactoring techniques for mircoservices-based systems. At first, microservices smells get matched to existing bad smells (on code, design and architectural levels). Then, corresponding refactorings for the bad smells are discovered. The last step is adaptation of the techniques to be applicable to microservices.

$$\begin{tikzcd}
\boxed{\text{Microservices Smells}} \arrow[r, dashed, "\text{1. Matching}"] \arrow[d] & \boxed{\text{Bad Smells}} \arrow[d, dashed, "\text{2. Discovering}"] \\
\boxed{\text{Microservices Refacotrings}} & \boxed{\text{Bad Smells Refactorings}} \arrow[l, dashed, "\text{3. Adapting}"]
\end{tikzcd}$$

4. How would you assess our approach for creating a refactoring catalog for microservices

I agree that the approach is promising.

I partially agree with the approach

I do not completely agree with the approach

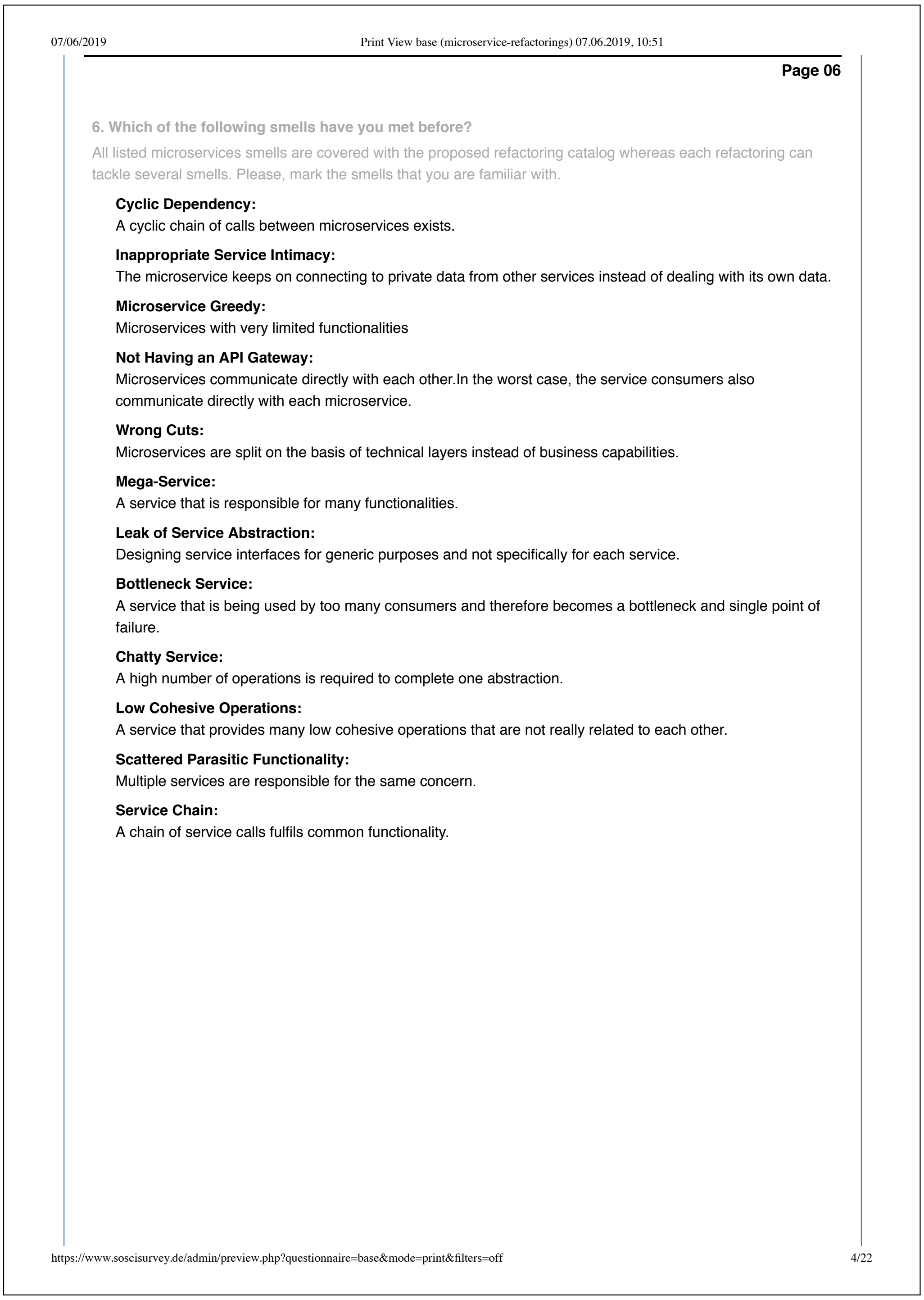
I completely disagree with the approach

I cannot assess the approach

5. Please, give your comments on the approach

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 3/22

Figure C.3.: Questionnaire page 3 of 22



07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

Page 06

6. Which of the following smells have you met before?

All listed microservices smells are covered with the proposed refactoring catalog whereas each refactoring can tackle several smells. Please, mark the smells that you are familiar with.

Cyclic Dependency:
A cyclic chain of calls between microservices exists.

Inappropriate Service Intimacy:
The microservice keeps on connecting to private data from other services instead of dealing with its own data.

Microservice Greedy:
Microservices with very limited functionalities

Not Having an API Gateway:
Microservices communicate directly with each other.In the worst case, the service consumers also communicate directly with each microservice.

Wrong Cuts:
Microservices are split on the basis of technical layers instead of business capabilities.

Mega-Service:
A service that is responsible for many functionalities.

Leak of Service Abstraction:
Designing service interfaces for generic purposes and not specifically for each service.

Bottleneck Service:
A service that is being used by too many consumers and therefore becomes a bottleneck and single point of failure.

Chatty Service:
A high number of operations is required to complete one abstraction.

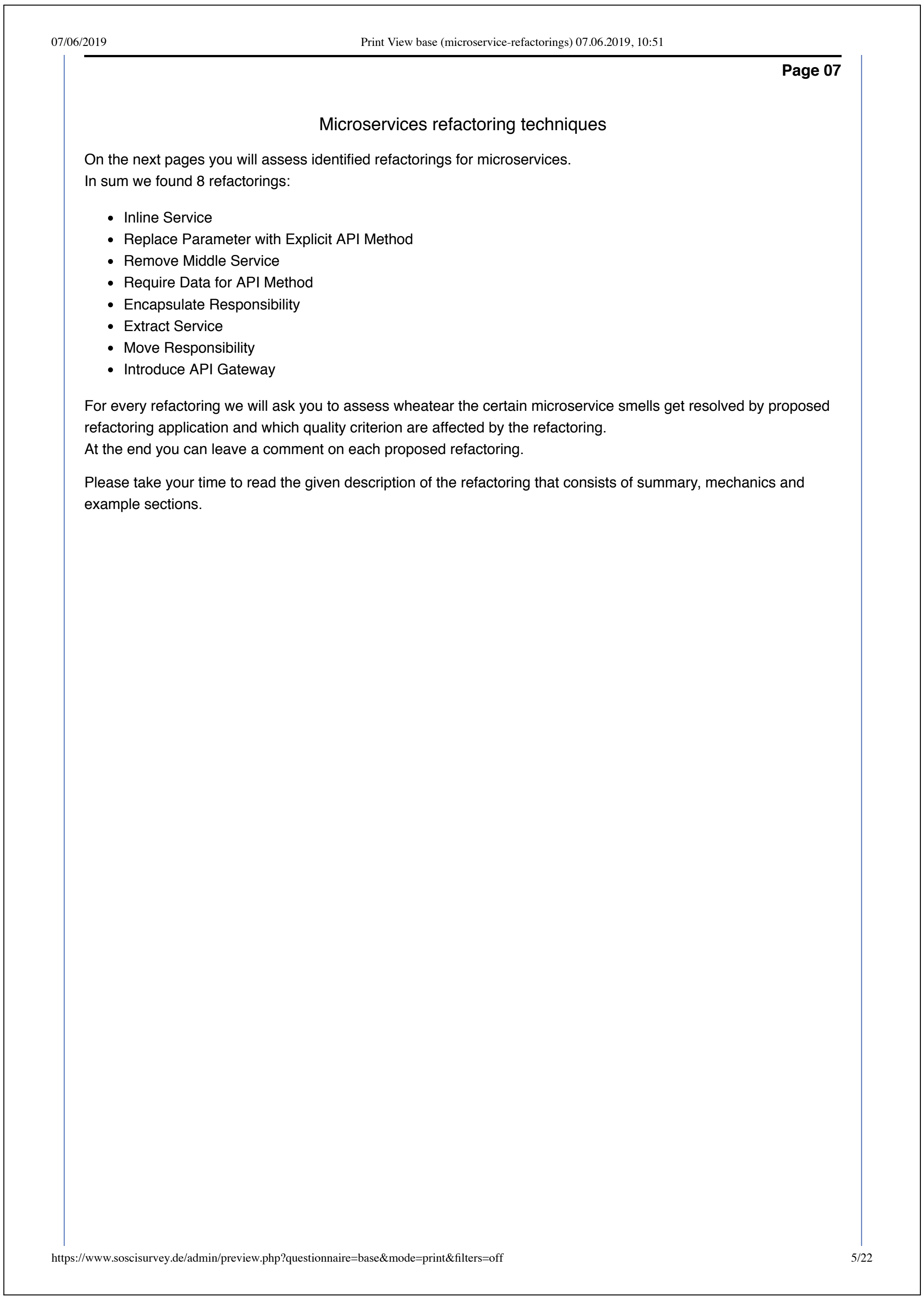
Low Cohesive Operations:
A service that provides many low cohesive operations that are not really related to each other.

Scattered Parasitic Functionality:
Multiple services are responsible for the same concern.

Service Chain:
A chain of service calls fulfils common functionality.

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 4/22

Figure C.4.: Questionnaire page 4 of 22


07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

Page 07

Microservices refactoring techniques

On the next pages you will assess identified refactorings for microservices.
In sum we found 8 refactorings:

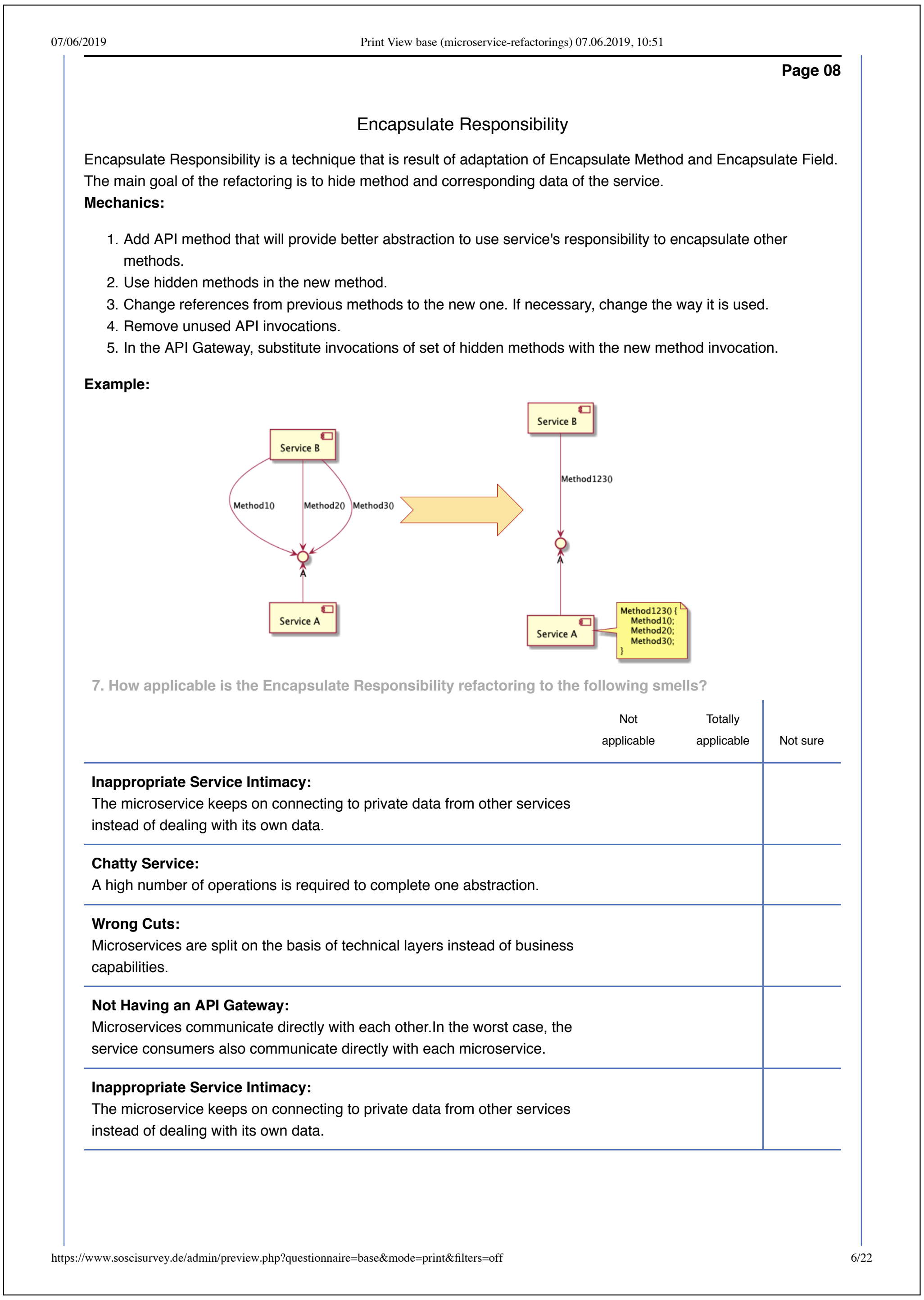
- Inline Service
- Replace Parameter with Explicit API Method
- Remove Middle Service
- Require Data for API Method
- Encapsulate Responsibility
- Extract Service
- Move Responsibility
- Introduce API Gateway

For every refactoring we will ask you to assess wheatear the certain microservice smells get resolved by proposed refactoring application and which quality criterion are affected by the refactoring.
At the end you can leave a comment on each proposed refactoring.

Please take your time to read the given description of the refactoring that consists of summary, mechanics and example sections.

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 5/22

Figure C.5.: Questionnaire page 5 of 22



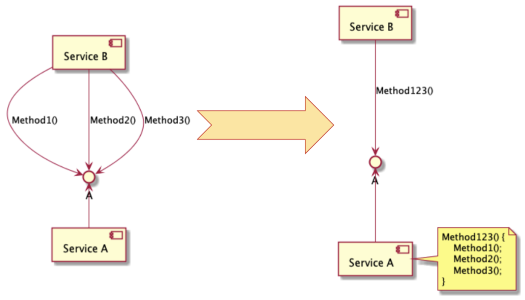
Page 08

Encapsulate Responsibility

Encapsulate Responsibility is a technique that is result of adaptation of Encapsulate Method and Encapsulate Field. The main goal of the refactoring is to hide method and corresponding data of the service.
Mechanics:

1. Add API method that will provide better abstraction to use service's responsibility to encapsulate other methods.
2. Use hidden methods in the new method.
3. Change references from previous methods to the new one. If necessary, change the way it is used.
4. Remove unused API invocations.
5. In the API Gateway, substitute invocations of set of hidden methods with the new method invocation.

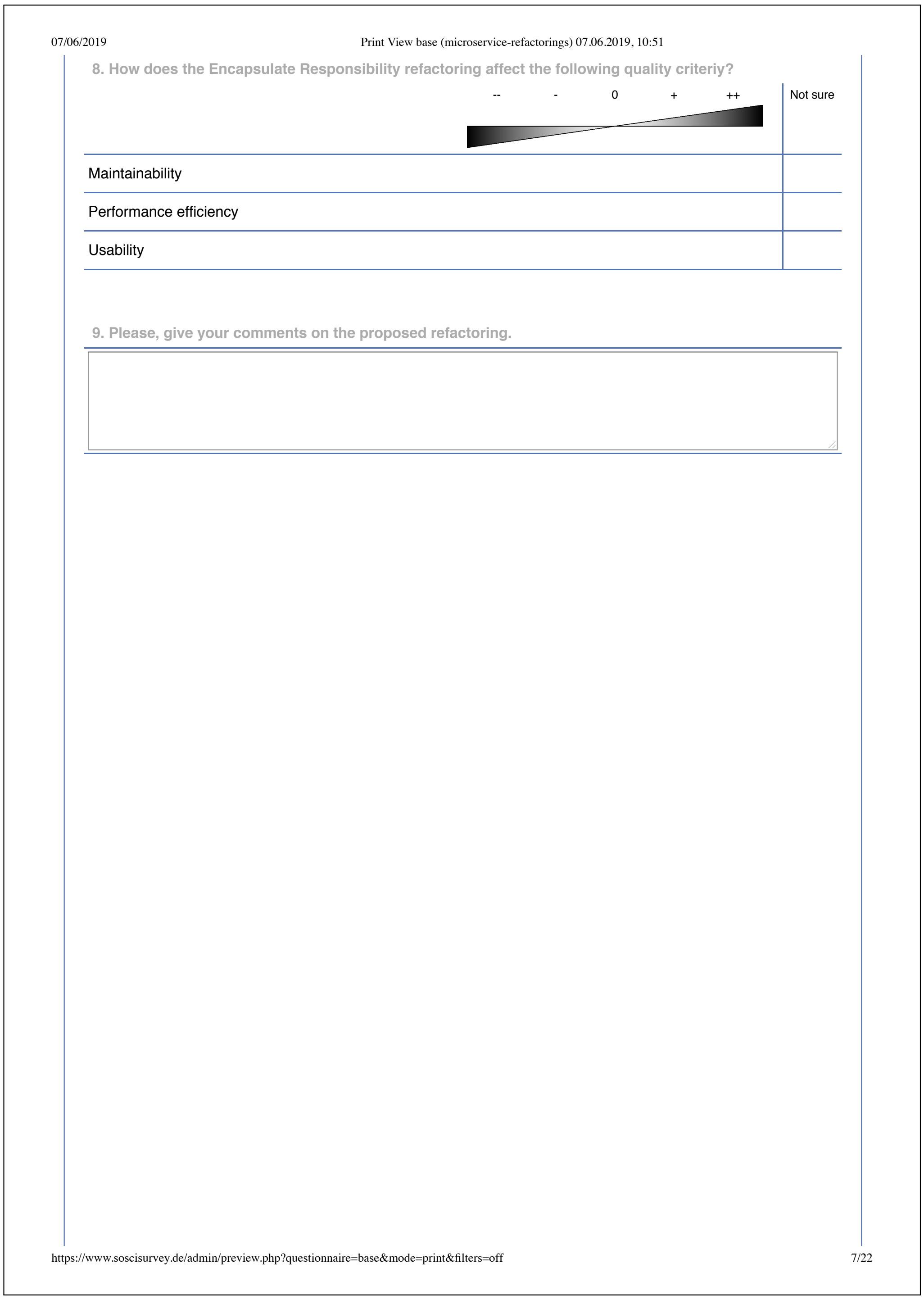
Example:



7. How applicable is the Encapsulate Responsibility refactoring to the following smells?

	Not applicable		Totally applicable	Not sure
Inappropriate Service Intimacy: The microservice keeps on connecting to private data from other services instead of dealing with its own data.				
Chatty Service: A high number of operations is required to complete one abstraction.				
Wrong Cuts: Microservices are split on the basis of technical layers instead of business capabilities.				
Not Having an API Gateway: Microservices communicate directly with each other.In the worst case, the service consumers also communicate directly with each microservice.				
Inappropriate Service Intimacy: The microservice keeps on connecting to private data from other services instead of dealing with its own data.				

Figure C.6.: Questionnaire page 6 of 22



07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

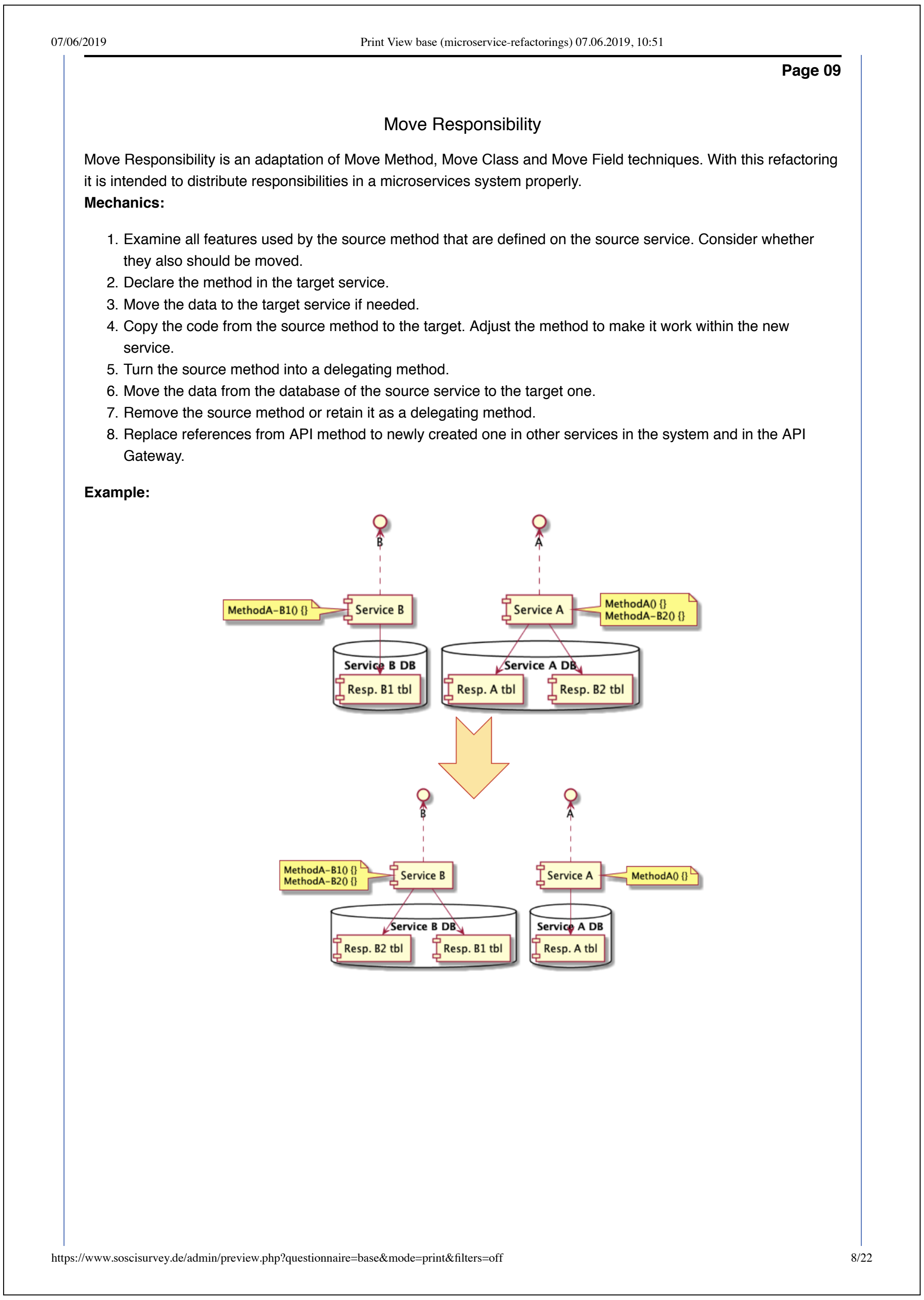
8. How does the Encapsulate Responsibility refactoring affect the following quality criteriy?

	--	-	0	+	++	Not sure
Maintainability						
Performance efficiency						
Usability						

9. Please, give your comments on the proposed refactoring.

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 7/22

Figure C.7.: Questionnaire page 7 of 22



Page 09

Move Responsibility

Move Responsibility is an adaptation of Move Method, Move Class and Move Field techniques. With this refactoring it is intended to distribute responsibilities in a microservices system properly.
Mechanics:

1. Examine all features used by the source method that are defined on the source service. Consider whether they also should be moved.
2. Declare the method in the target service.
3. Move the data to the target service if needed.
4. Copy the code from the source method to the target. Adjust the method to make it work within the new service.
5. Turn the source method into a delegating method.
6. Move the data from the database of the source service to the target one.
7. Remove the source method or retain it as a delegating method.
8. Replace references from API method to newly created one in other services in the system and in the API Gateway.

Example:

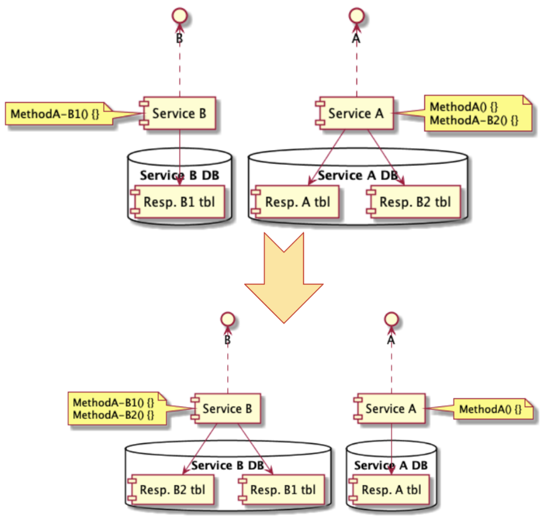
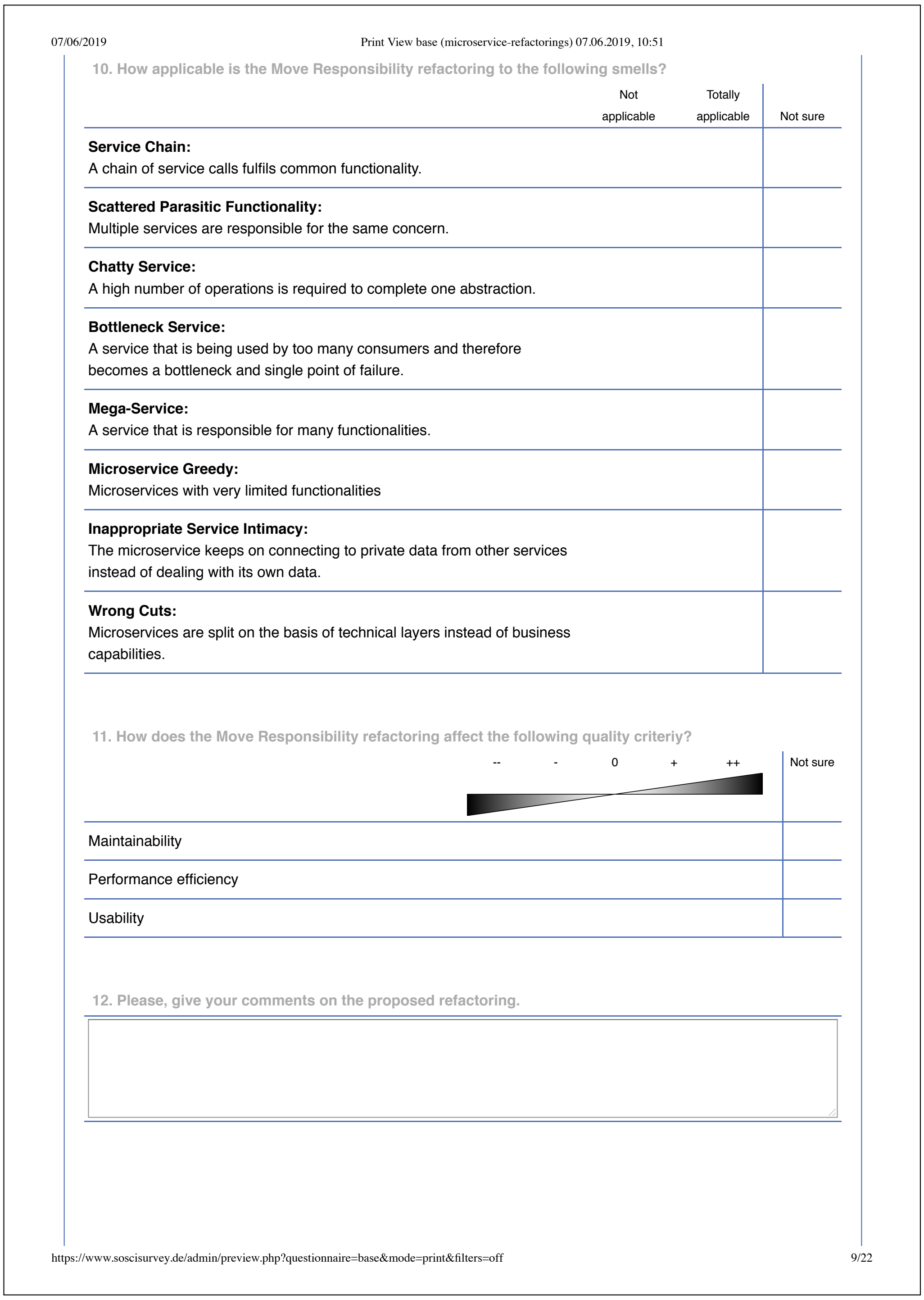


Figure C.8.: Questionnaire page 8 of 22



07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

10. How applicable is the Move Responsibility refactoring to the following smells?

	Not applicable	Totally applicable	Not sure
Service Chain: A chain of service calls fulfils common functionality.			
Scattered Parasitic Functionality: Multiple services are responsible for the same concern.			
Chatty Service: A high number of operations is required to complete one abstraction.			
Bottleneck Service: A service that is being used by too many consumers and therefore becomes a bottleneck and single point of failure.			
Mega-Service: A service that is responsible for many functionalities.			
Microservice Greedy: Microservices with very limited functionalities			
Inappropriate Service Intimacy: The microservice keeps on connecting to private data from other services instead of dealing with its own data.			
Wrong Cuts: Microservices are split on the basis of technical layers instead of business capabilities.			

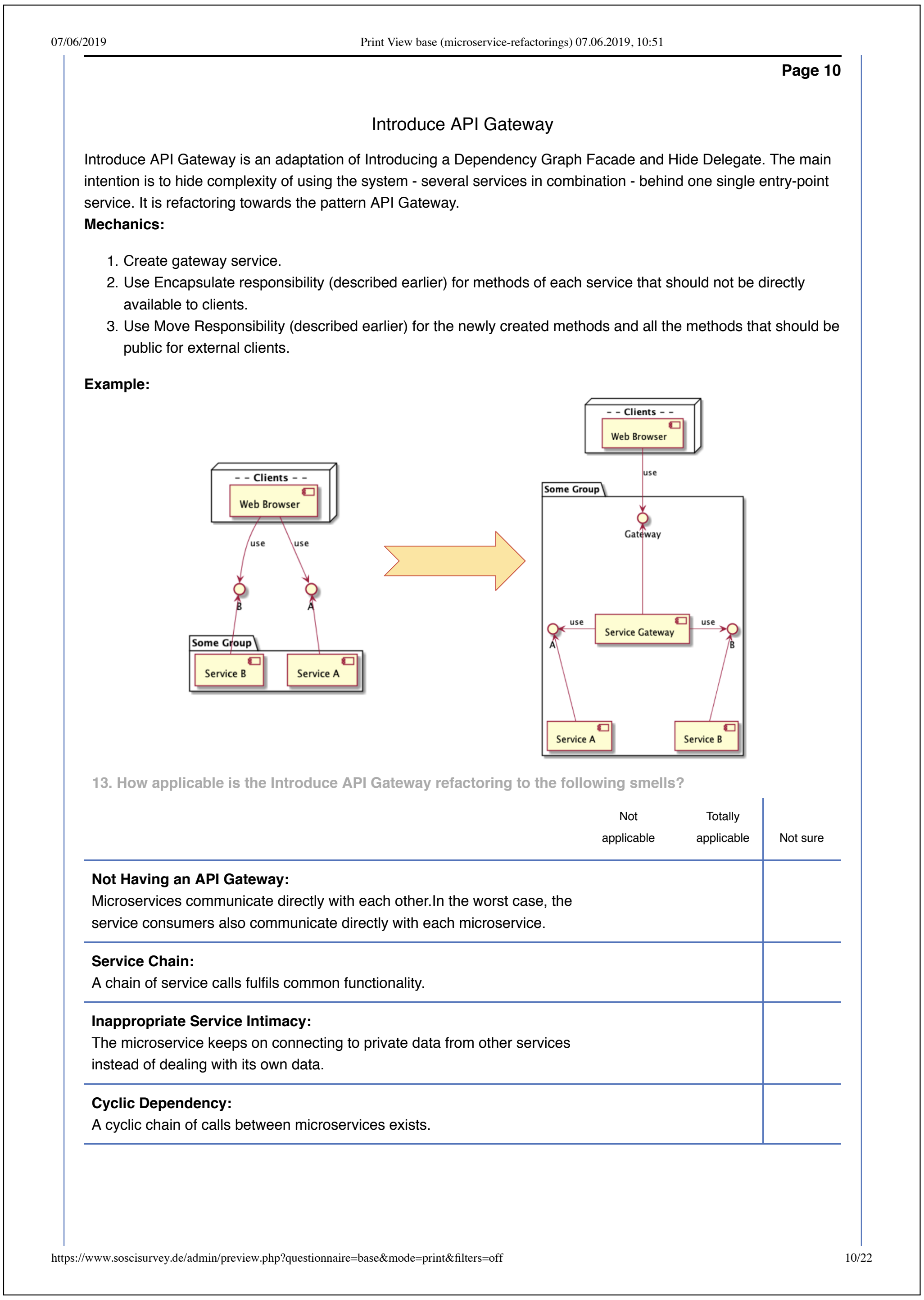
11. How does the Move Responsibility refactoring affect the following quality criteriy?

	--	-	0	+	++	Not sure
Maintainability						
Performance efficiency						
Usability						

12. Please, give your comments on the proposed refactoring.

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 9/22

Figure C.9.: Questionnaire page 9 of 22



07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

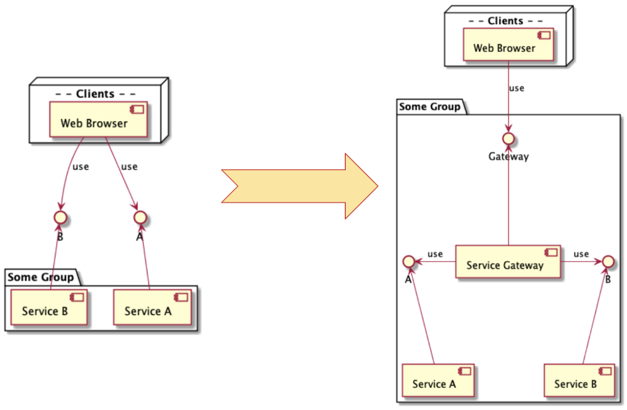
Page 10

Introduce API Gateway

Introduce API Gateway is an adaptation of Introducing a Dependency Graph Facade and Hide Delegate. The main intention is to hide complexity of using the system - several services in combination - behind one single entry-point service. It is refactoring towards the pattern API Gateway.
Mechanics:

1. Create gateway service.
2. Use Encapsulate responsibility (described earlier) for methods of each service that should not be directly available to clients.
3. Use Move Responsibility (described earlier) for the newly created methods and all the methods that should be public for external clients.

Example:

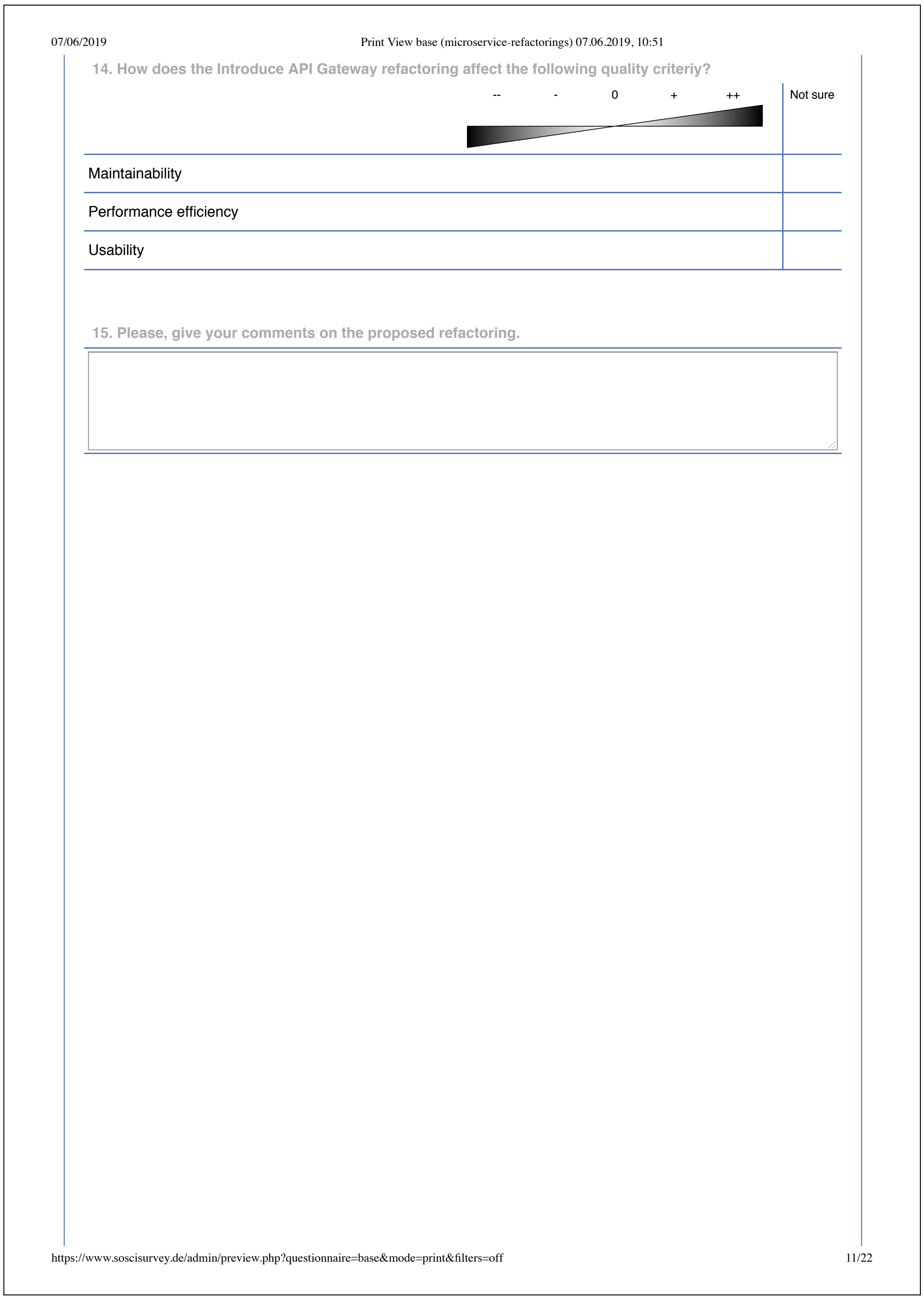


13. How applicable is the Introduce API Gateway refactoring to the following smells?

	Not applicable	Totally applicable	Not sure
Not Having an API Gateway: Microservices communicate directly with each other.In the worst case, the service consumers also communicate directly with each microservice.			
Service Chain: A chain of service calls fulfils common functionality.			
Inappropriate Service Intimacy: The microservice keeps on connecting to private data from other services instead of dealing with its own data.			
Cyclic Dependency: A cyclic chain of calls between microservices exists.			

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 10/22

Figure C.10.: Questionnaire page 10 of 22



07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

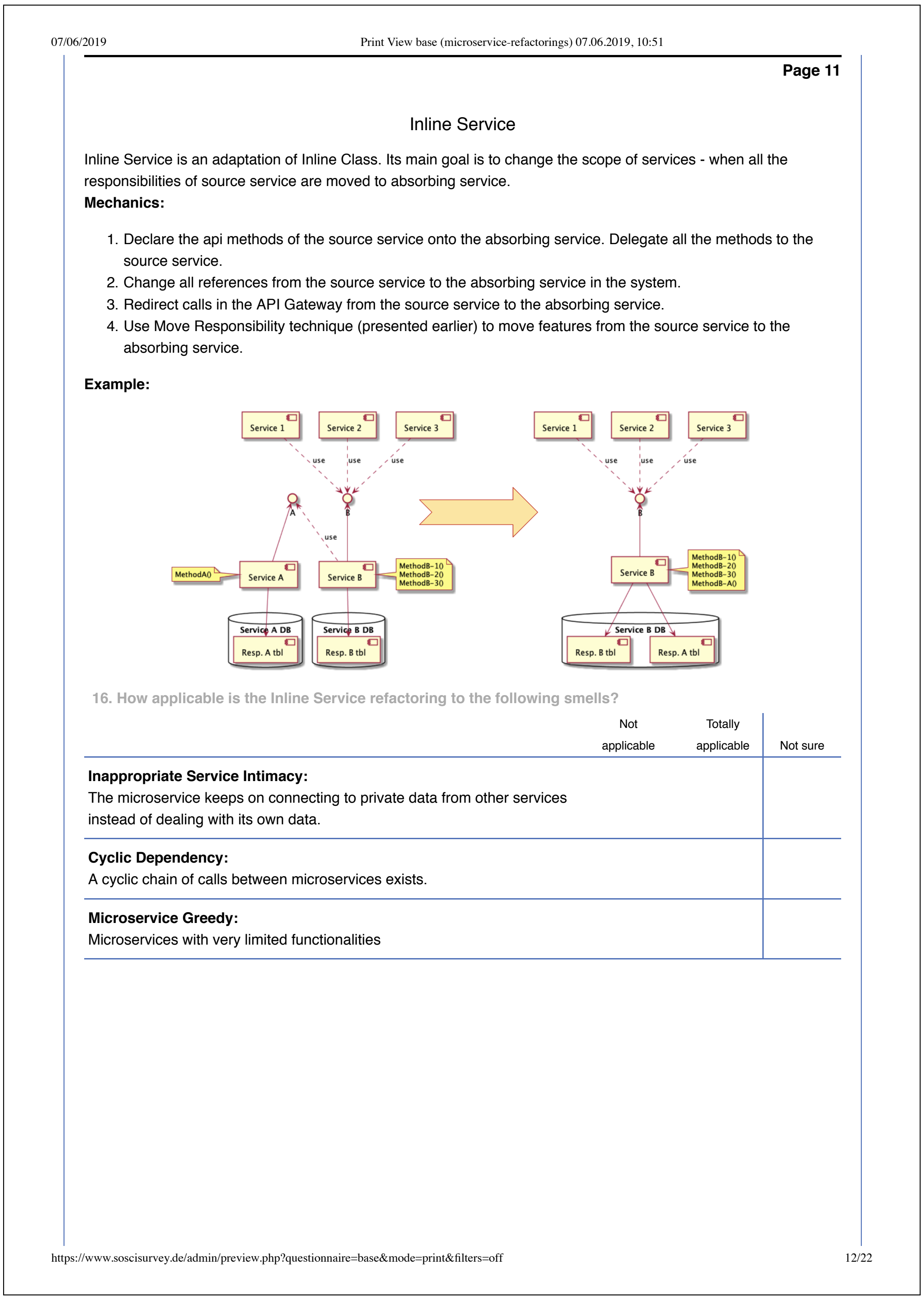
14. How does the Introduce API Gateway refactoring affect the following quality criteriy?

	--	-	0	+	++	Not sure
Maintainability						
Performance efficiency						
Usability						

15. Please, give your comments on the proposed refactoring.

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 11/22

Figure C.11.: Questionnaire page 11 of 22



07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

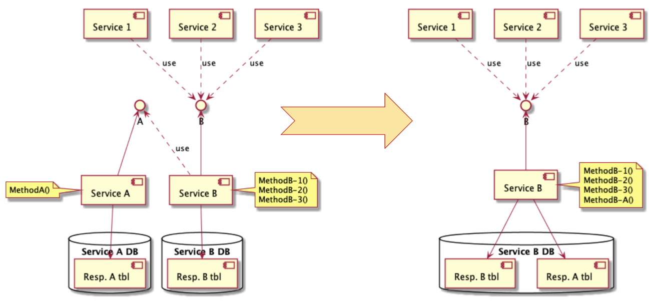
Page 11

Inline Service

Inline Service is an adaptation of Inline Class. Its main goal is to change the scope of services - when all the responsibilities of source service are moved to absorbing service.
Mechanics:

1. Declare the api methods of the source service onto the absorbing service. Delegate all the methods to the source service.
2. Change all references from the source service to the absorbing service in the system.
3. Redirect calls in the API Gateway from the source service to the absorbing service.
4. Use Move Responsibility technique (presented earlier) to move features from the source service to the absorbing service.

Example:

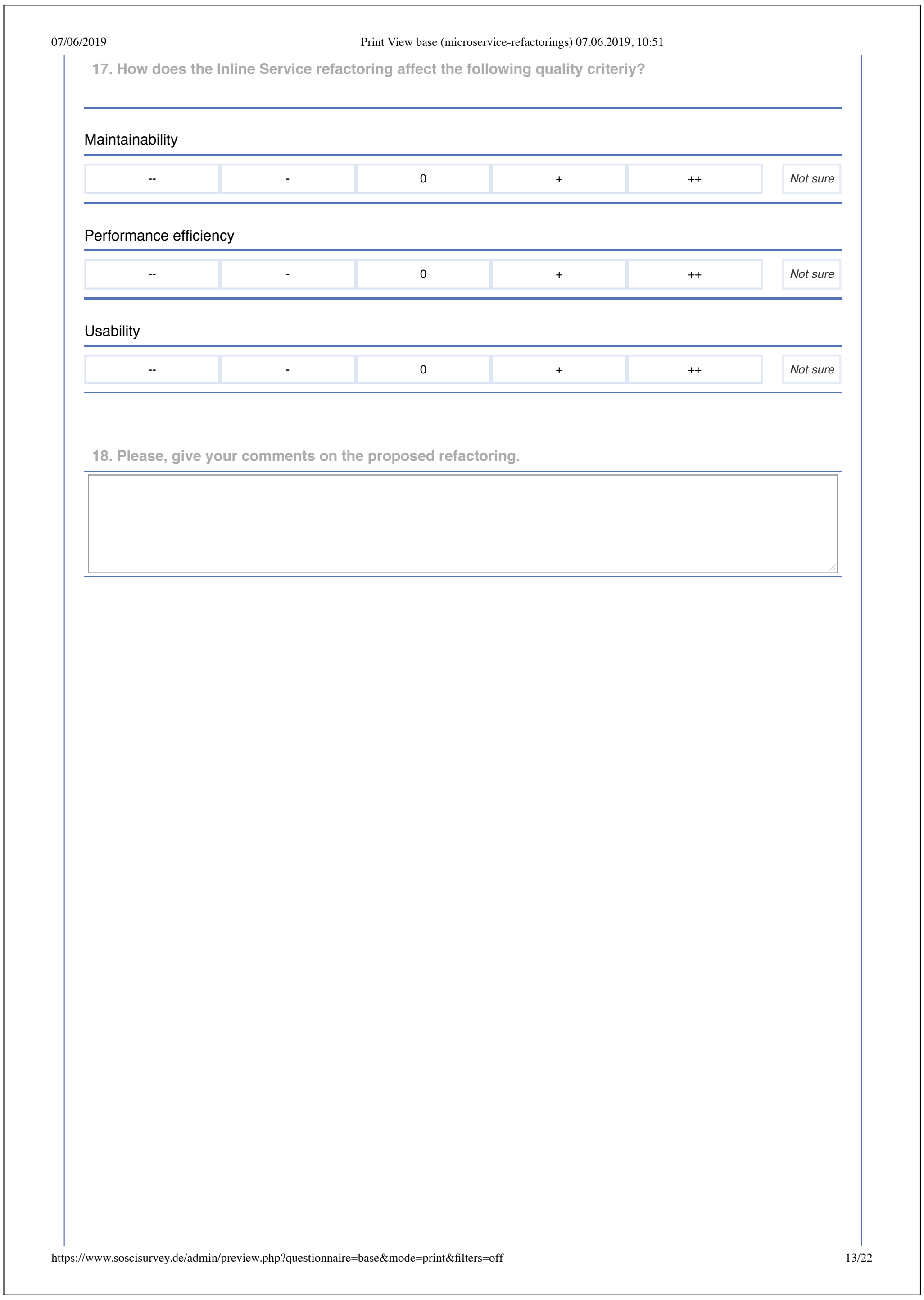


16. How applicable is the Inline Service refactoring to the following smells?

	Not applicable	Totally applicable	Not sure
Inappropriate Service Intimacy: The microservice keeps on connecting to private data from other services instead of dealing with its own data.			
Cyclic Dependency: A cyclic chain of calls between microservices exists.			
Microservice Greedy: Microservices with very limited functionalities			

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 12/22

Figure C.12.: Questionnaire page 12 of 22



07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

17. How does the Inline Service refactoring affect the following quality criteriy?

Maintainability

--	-	0	+	++	*Not sure*

Performance efficiency

--	-	0	+	++	*Not sure*

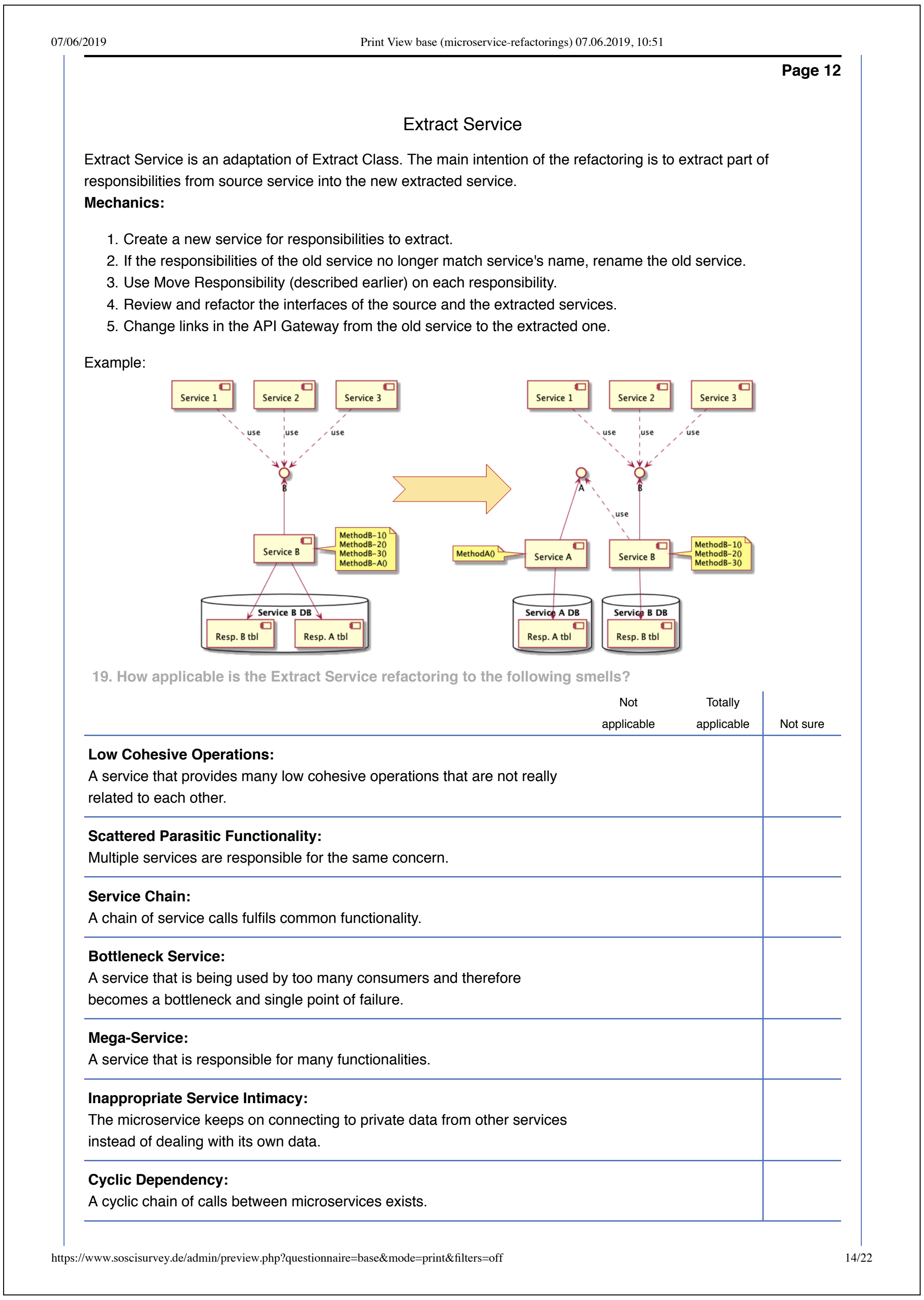
Usability

--	-	0	+	++	*Not sure*

18. Please, give your comments on the proposed refactoring.

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 13/22

Figure C.13.: Questionnaire page 13 of 22



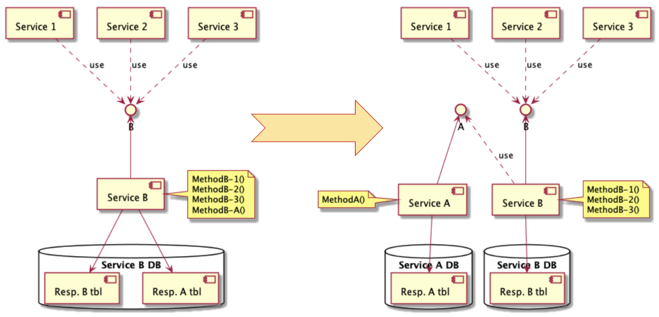
Page 12

Extract Service

Extract Service is an adaptation of Extract Class. The main intention of the refactoring is to extract part of responsibilities from source service into the new extracted service.
Mechanics:

1. Create a new service for responsibilities to extract.
2. If the responsibilities of the old service no longer match service's name, rename the old service.
3. Use Move Responsibility (described earlier) on each responsibility.
4. Review and refactor the interfaces of the source and the extracted services.
5. Change links in the API Gateway from the old service to the extracted one.

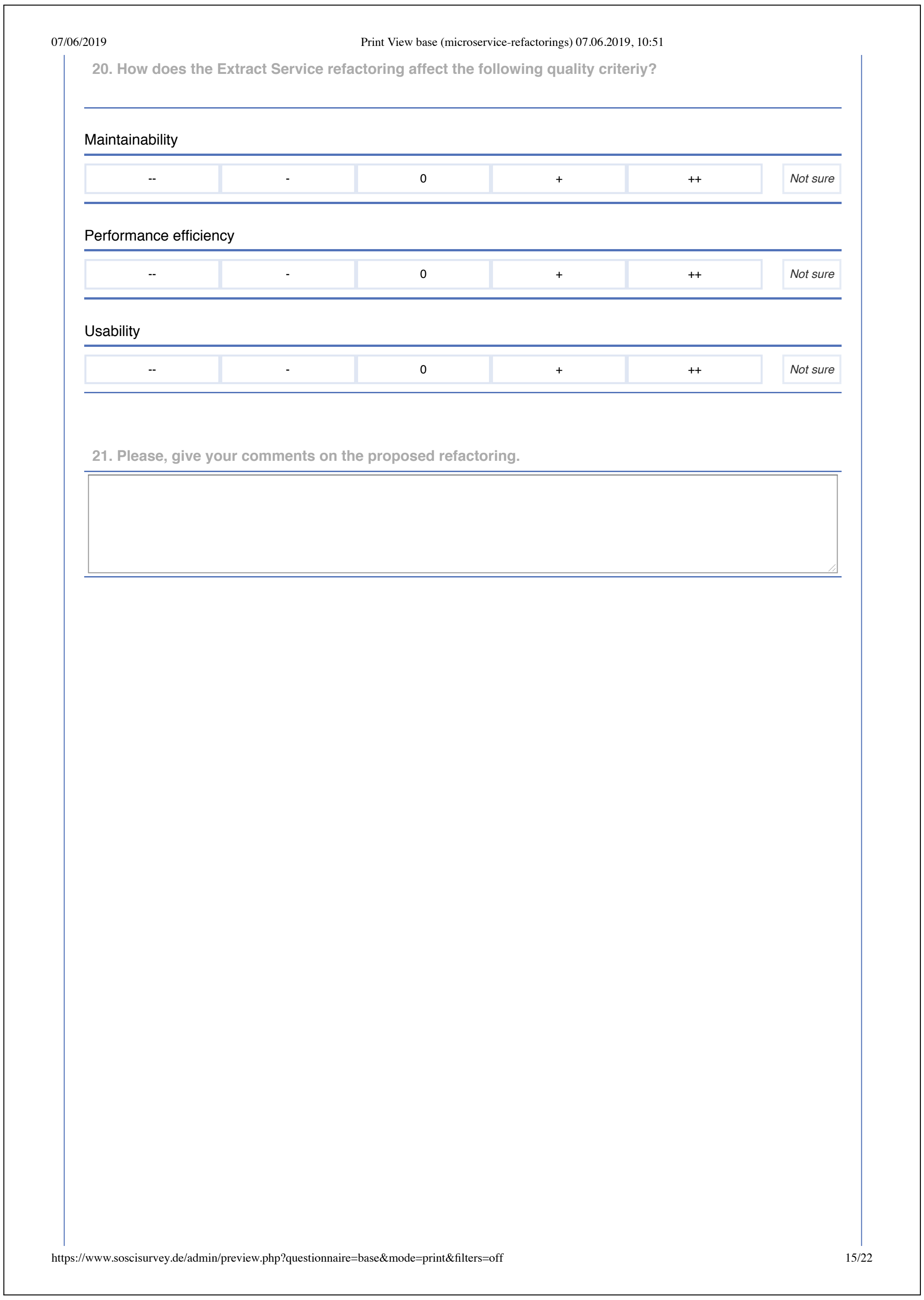
Example:



19. How applicable is the Extract Service refactoring to the following smells?

	Not applicable	Totally applicable	Not sure
Low Cohesive Operations: A service that provides many low cohesive operations that are not really related to each other.			
Scattered Parasitic Functionality: Multiple services are responsible for the same concern.			
Service Chain: A chain of service calls fulfils common functionality.			
Bottleneck Service: A service that is being used by too many consumers and therefore becomes a bottleneck and single point of failure.			
Mega-Service: A service that is responsible for many functionalities.			
Inappropriate Service Intimacy: The microservice keeps on connecting to private data from other services instead of dealing with its own data.			
Cyclic Dependency: A cyclic chain of calls between microservices exists.			

Figure C.14.: Questionnaire page 14 of 22


07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

20. How does the Extract Service refactoring affect the following quality criteriy?

Maintainability

--	-	0	+	++	*Not sure*

Performance efficiency

--	-	0	+	++	*Not sure*

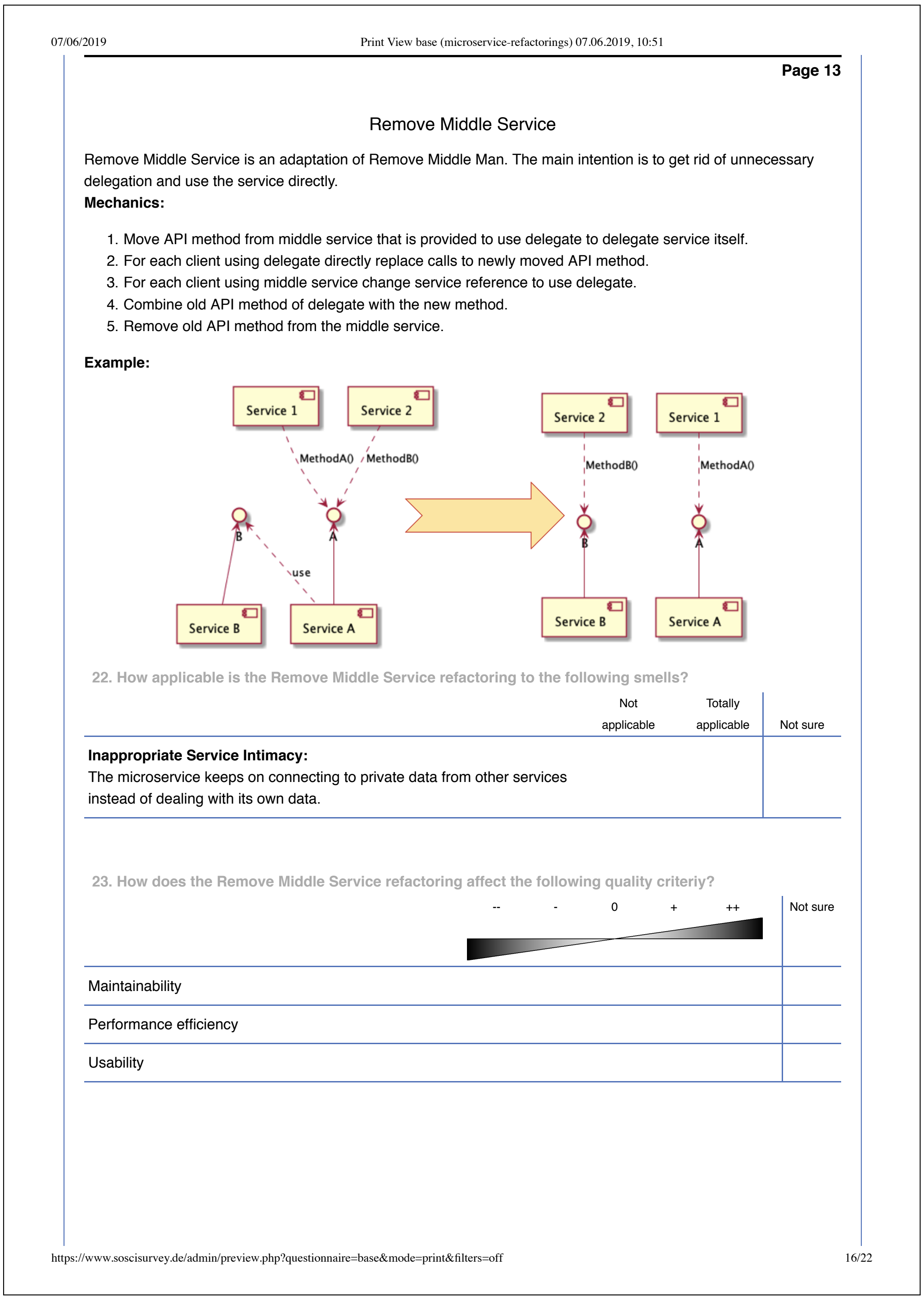
Usability

--	-	0	+	++	*Not sure*

21. Please, give your comments on the proposed refactoring.

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 15/22

Figure C.15.: Questionnaire page 15 of 22



07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

Page 13

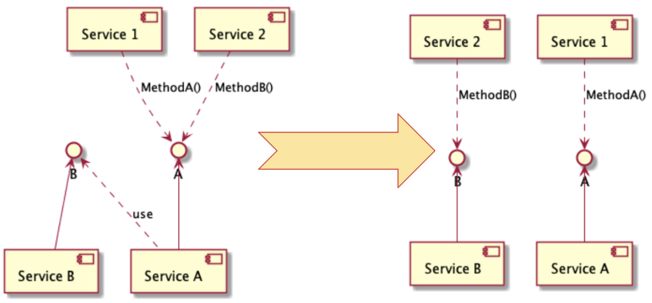
Remove Middle Service

Remove Middle Service is an adaptation of Remove Middle Man. The main intention is to get rid of unnecessary delegation and use the service directly.

Mechanics:

1. Move API method from middle service that is provided to use delegate to delegate service itself.
2. For each client using delegate directly replace calls to newly moved API method.
3. For each client using middle service change service reference to use delegate.
4. Combine old API method of delegate with the new method.
5. Remove old API method from the middle service.

Example:



22. How applicable is the Remove Middle Service refactoring to the following smells?

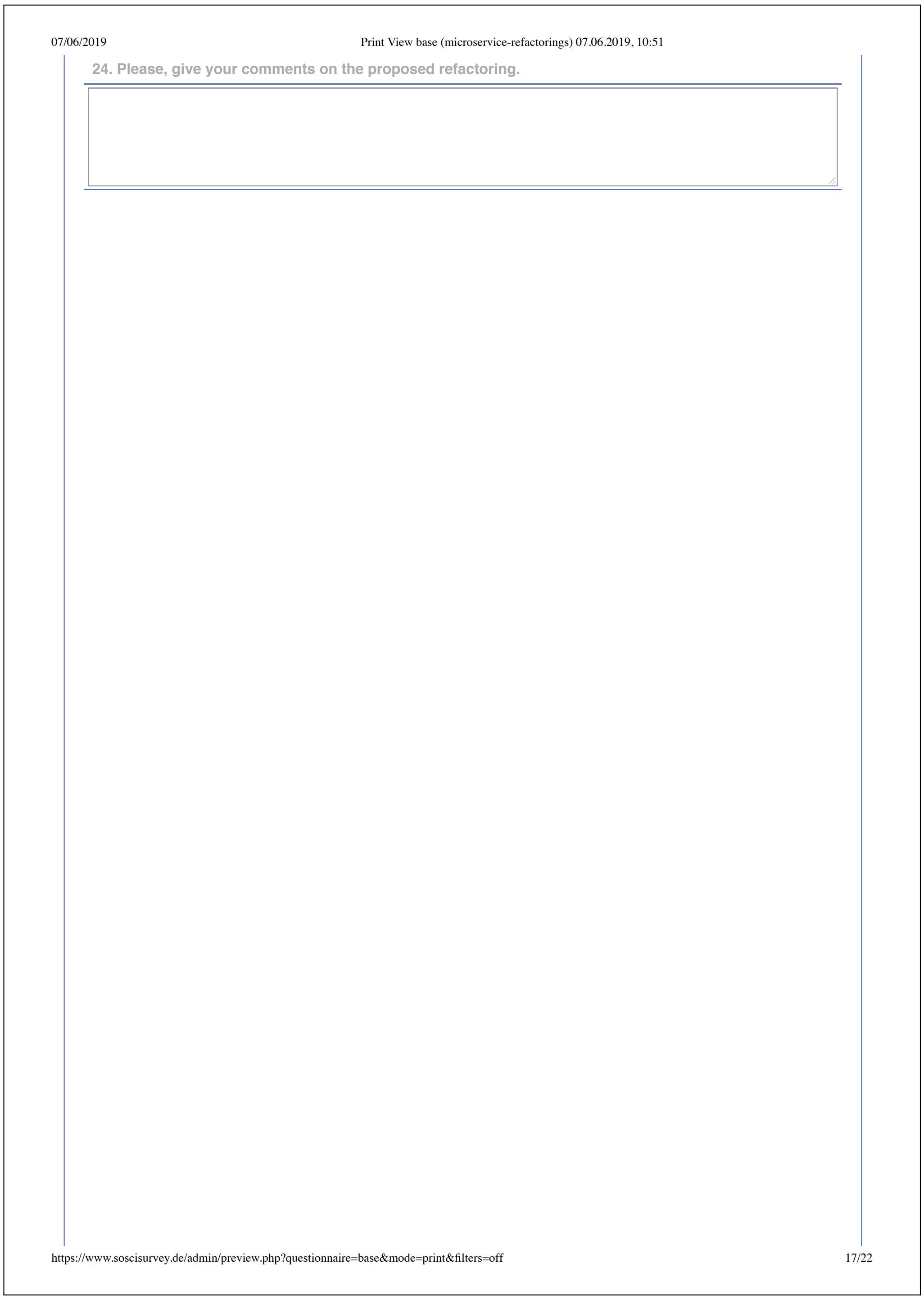
	Not applicable	Totally applicable	Not sure
Inappropriate Service Intimacy: The microservice keeps on connecting to private data from other services instead of dealing with its own data.			

23. How does the Remove Middle Service refactoring affect the following quality criteriy?

	--	-	0	+	++	Not sure
Maintainability						
Performance efficiency						
Usability						

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 16/22

Figure C.16.: Questionnaire page 16 of 22

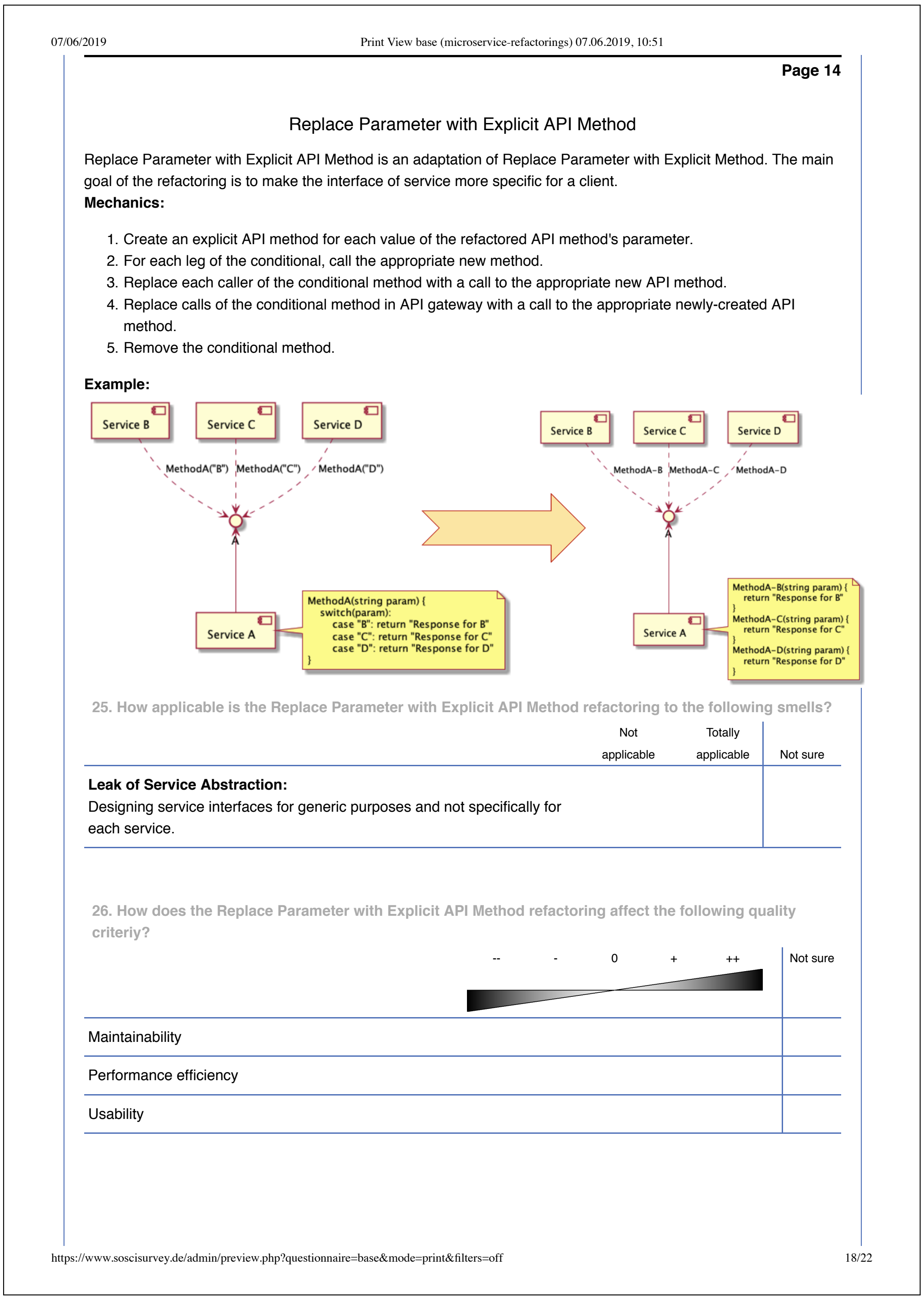


07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

24. Please, give your comments on the proposed refactoring.

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 17/22

Figure C.17.: Questionnaire page 17 of 22



07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

Page 14

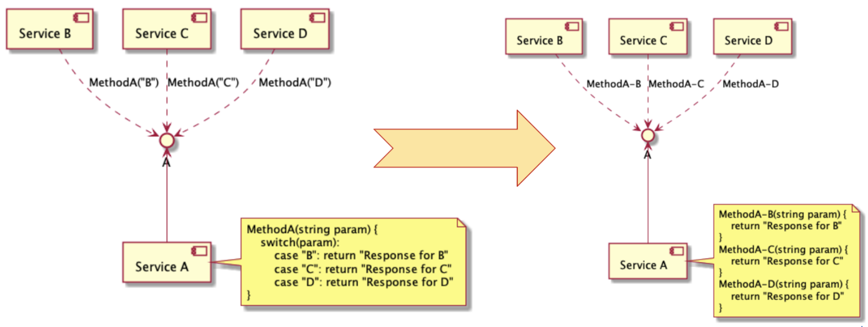
Replace Parameter with Explicit API Method

Replace Parameter with Explicit API Method is an adaptation of Replace Parameter with Explicit Method. The main goal of the refactoring is to make the interface of service more specific for a client.

Mechanics:

1. Create an explicit API method for each value of the refactored API method's parameter.
2. For each leg of the conditional, call the appropriate new method.
3. Replace each caller of the conditional method with a call to the appropriate new API method.
4. Replace calls of the conditional method in API gateway with a call to the appropriate newly-created API method.
5. Remove the conditional method.

Example:



```
MethodA(string param) {
    switch(param):
        case "B": return "Response for B"
        case "C": return "Response for C"
        case "D": return "Response for D"
}
```

```
MethodA-B(string param) {
    return "Response for B"
}
MethodA-C(string param) {
    return "Response for C"
}
MethodA-D(string param) {
    return "Response for D"
}
```

25. How applicable is the Replace Parameter with Explicit API Method refactoring to the following smells?

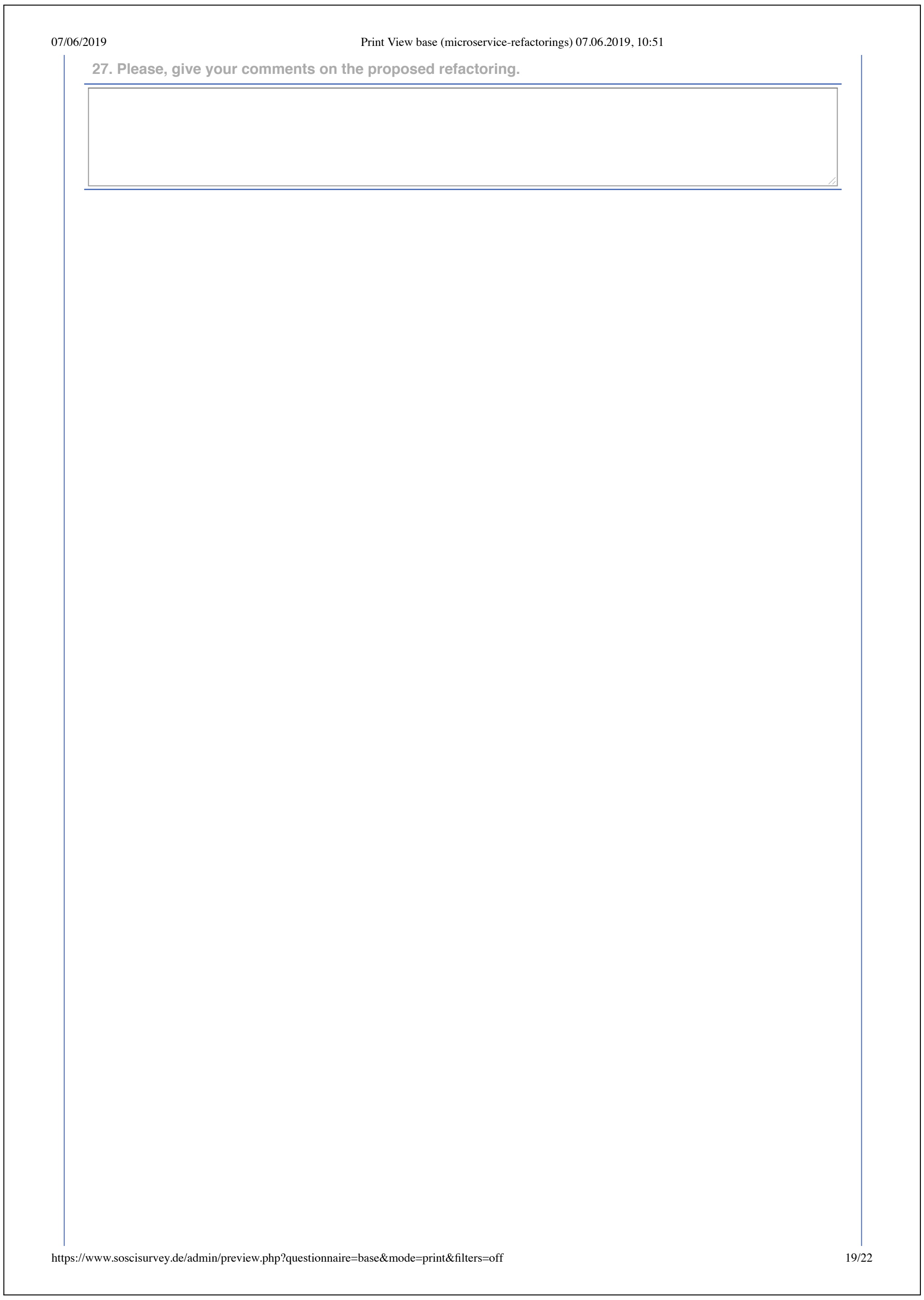
	Not applicable	Totally applicable	Not sure
Leak of Service Abstraction: Designing service interfaces for generic purposes and not specifically for each service.			

26. How does the Replace Parameter with Explicit API Method refactoring affect the following quality criteriy?

	--	-	0	+	++	Not sure
Maintainability						
Performance efficiency						
Usability						

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 18/22

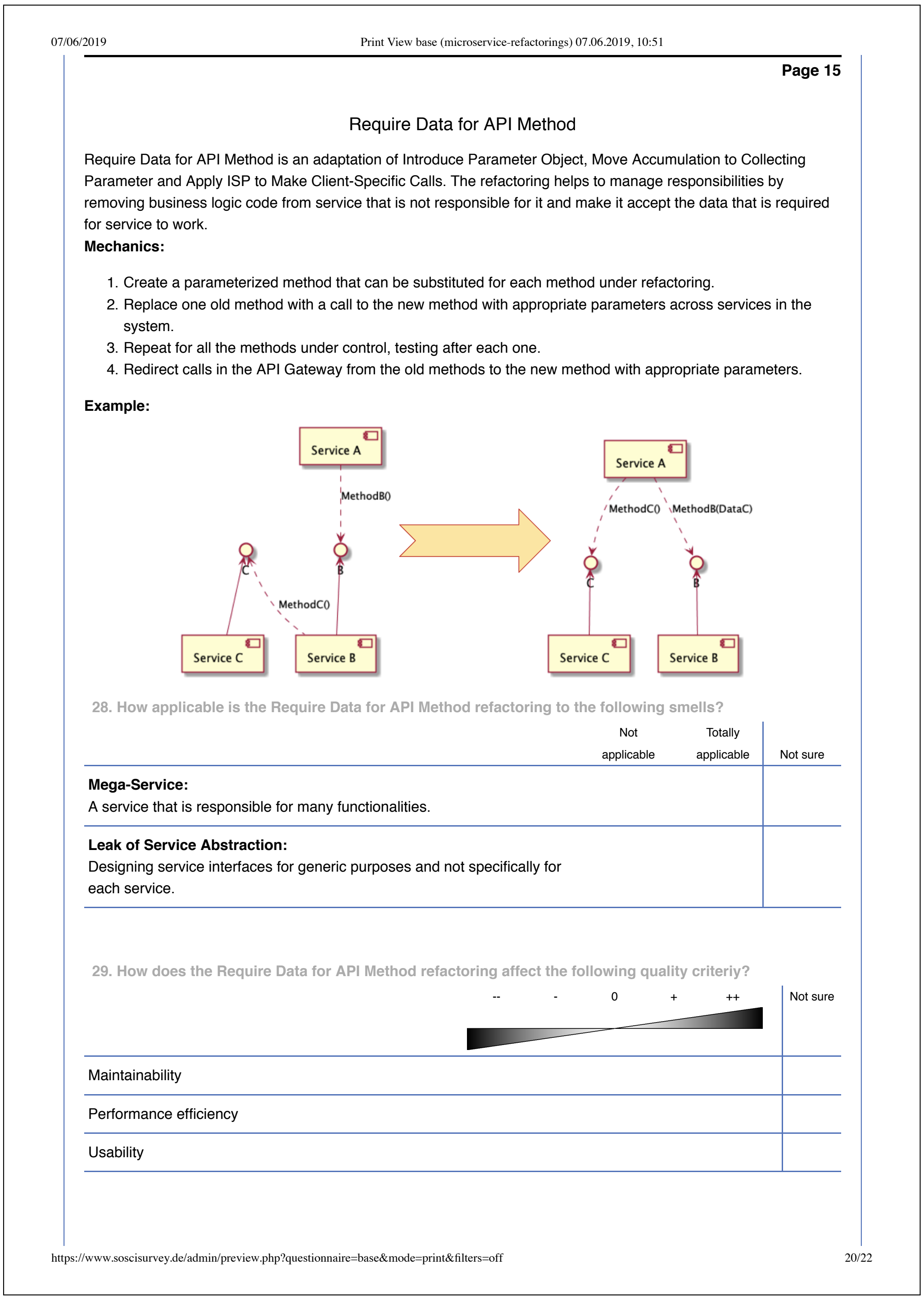
Figure C.18.: Questionnaire page 18 of 22


07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

27. Please, give your comments on the proposed refactoring.

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 19/22

Figure C.19.: Questionnaire page 19 of 22



07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

Page 15

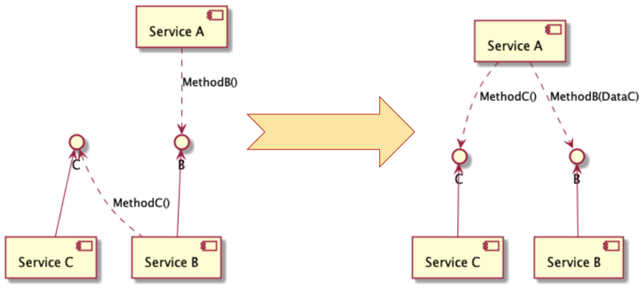
Require Data for API Method

Require Data for API Method is an adaptation of Introduce Parameter Object, Move Accumulation to Collecting Parameter and Apply ISP to Make Client-Specific Calls. The refactoring helps to manage responsibilities by removing business logic code from service that is not responsible for it and make it accept the data that is required for service to work.

Mechanics:

1. Create a parameterized method that can be substituted for each method under refactoring.
2. Replace one old method with a call to the new method with appropriate parameters across services in the system.
3. Repeat for all the methods under control, testing after each one.
4. Redirect calls in the API Gateway from the old methods to the new method with appropriate parameters.

Example:



28. How applicable is the Require Data for API Method refactoring to the following smells?

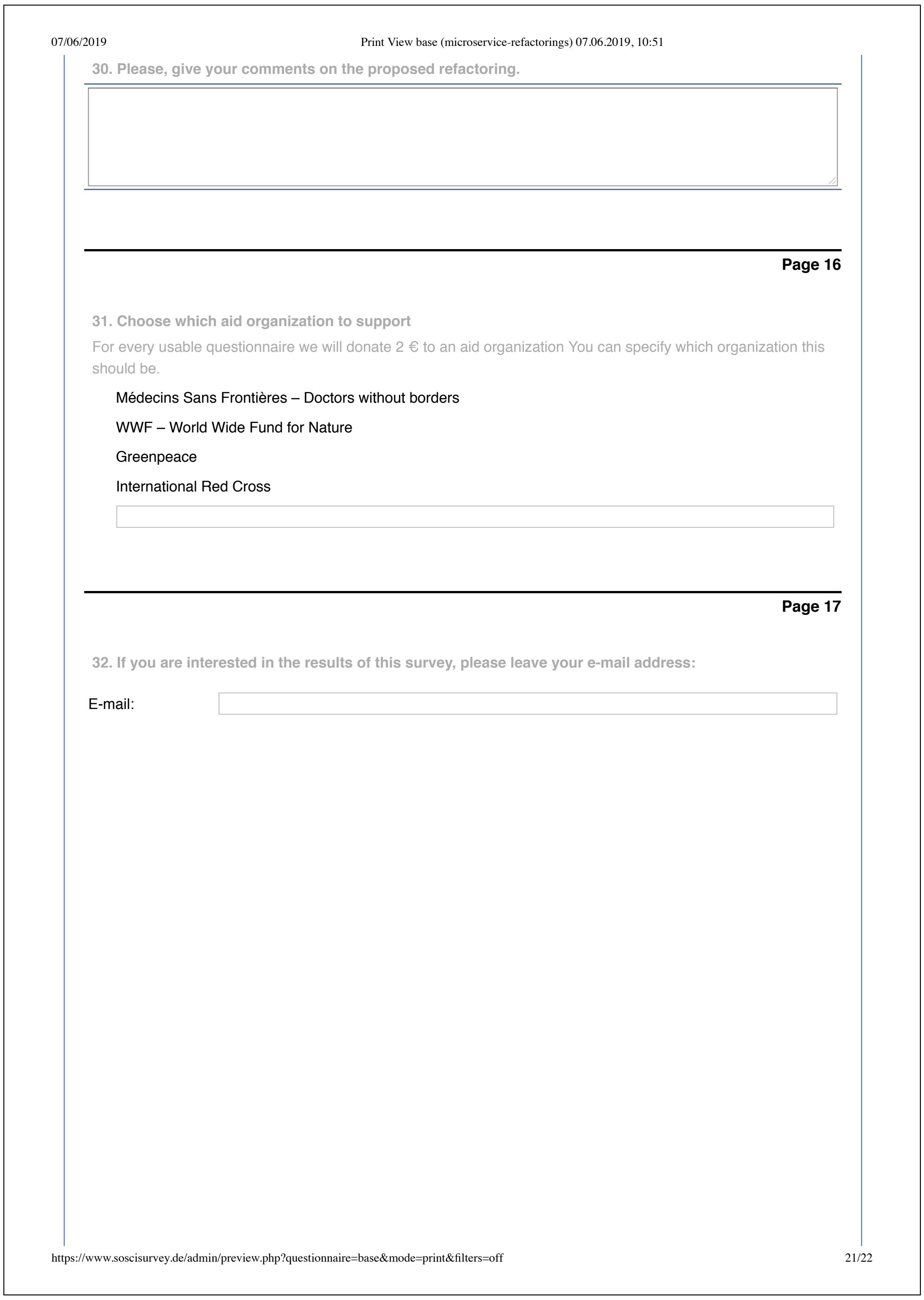
	Not applicable	Totally applicable	Not sure
Mega-Service: A service that is responsible for many functionalities.			
Leak of Service Abstraction: Designing service interfaces for generic purposes and not specifically for each service.			

29. How does the Require Data for API Method refactoring affect the following quality criteriy?

	--	-	0	+	++	Not sure
Maintainability						
Performance efficiency						
Usability						

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 20/22

Figure C.20.: Questionnaire page 20 of 22


07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

30. Please, give your comments on the proposed refactoring.

Page 16

31. Choose which aid organization to support

For every usable questionnaire we will donate 2 € to an aid organization You can specify which organization this should be.

Médecins Sans Frontières – Doctors without borders

WWF – World Wide Fund for Nature

Greenpeace

International Red Cross

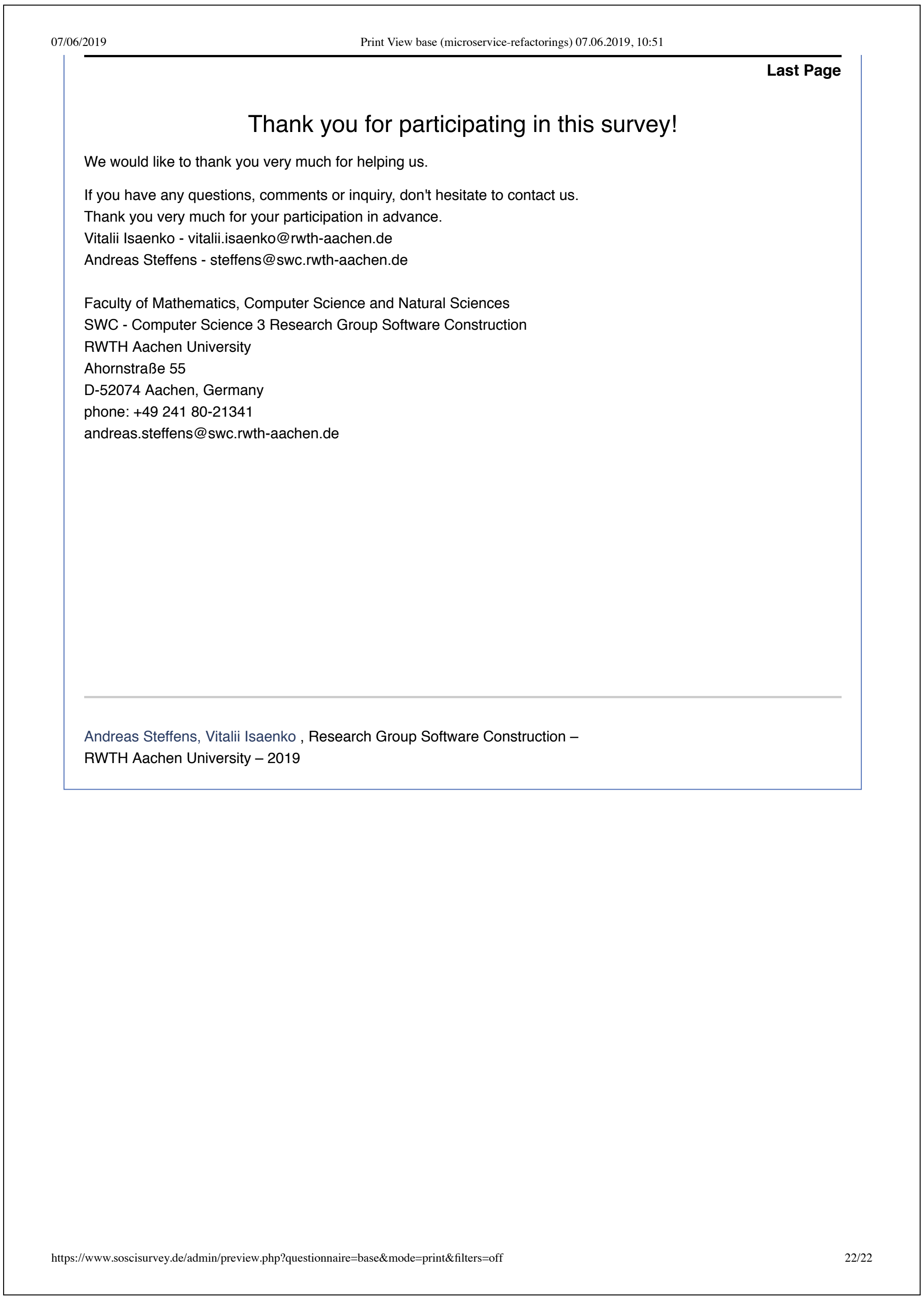
Page 17

32. If you are interested in the results of this survey, please leave your e-mail address:

E-mail:

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 21/22

Figure C.21.: Questionnaire page 21 of 22



07/06/2019 Print View base (microservice-refactorings) 07.06.2019, 10:51

Last Page

Thank you for participating in this survey!

We would like to thank you very much for helping us.

If you have any questions, comments or inquiry, don't hesitate to contact us.
Thank you very much for your participation in advance.
Vitalii Isaenko - vitalii.isaenko@rwth-aachen.de
Andreas Steffens - steffens@swc.rwth-aachen.de

Faculty of Mathematics, Computer Science and Natural Sciences
SWC - Computer Science 3 Research Group Software Construction
RWTH Aachen University
Ahornstraße 55
D-52074 Aachen, Germany
phone: +49 241 80-21341
andreas.steffens@swc.rwth-aachen.de

Andreas Steffens, Vitalii Isaenko , Research Group Software Construction –
RWTH Aachen University – 2019

https://www.soscisurvey.de/admin/preview.php?questionnaire=base&mode=print&filters=off 22/22

Figure C.22.: Questionnaire page 22 of 22

Bibliography

[Bae] *Versioning a REST API.* 2019. URL: https://www.baeldung.com/rest-versioning (visited on 04/28/2019) (cited on page 17).

[BL03] D. Beyer and C. Lewerentz. “CrocoPat: efficient pattern analysis in object-oriented programs”. In: *11th IEEE International Workshop on Program Comprehension, 2003.* 2003, pp. 294–295. DOI: 10.1109/WPC.2003.1199220 (cited on page 91).

[Bog+19] J. Bogner et al. “Towards a Collaborative Repository for the Documentation of Service-Based Antipatterns and Bad Smells”. In: Mar. 2019 (cited on pages 13, 21, 33, 88).

[Bud91] T. Budd. *An Introduction to Object-oriented Programming.* Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1991. ISBN: 0-201-54709-0 (cited on page 91).

[CU06] M. Choinzon and Y. Ueda. “Detecting Defects in Object Oriented Designs Using Design Metrics.” In: Jan. 2006, pp. 61–72 (cited on pages 91, 92).

[DDN02] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object Oriented Reengineering Patterns.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN: 1558606394 (cited on page 92).

[Fea04] M. Feathers. *Working Effectively with Legacy Code.* Martin, Robert C. Prentice Hall PTR, 2004. ISBN: 9780131177055 (cited on pages 6, 92).

[Fow19] M. Fowler. *Online catalog of refactorings.* 2019. URL: https://refactoring.com/catalog/ (visited on 01/27/2019) (cited on pages 38, 91, 92).

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Boston, MA, USA: Addison-Wesley, 1999. ISBN: 0-201-48567-2 (cited on pages 1, 5, 6, 20, 27, 34–42, 44, 48, 50, 53, 56, 59, 63, 66, 91, 92).

[Gar+09] J. Garcia et al. “Toward a Catalogue of Architectural Bad Smells”. In: *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems.* QoSA ’09. East Stroudsburg, PA, USA: Springer-Verlag, 2009, pp. 146–162. ISBN: 978-3-642-02350-7. DOI: 10.1007/978-3-642-02351-4_10 (cited on pages 5, 21, 33).

[Inc05] I. L. Inc. *Smells to Refactorings Cheatsheet.* 2005. URL: https://www.industriallogic.com/blog/smells-to-refactorings-cheatsheet/ (visited on 11/26/2018) (cited on page 35).

[Inf] *InFusion Hydrogen design flaw detection tool.* URL: `http://www.intooitus.com/products/infusion` (cited on pages 91, 92).

[JR92] P. Johnson and C. Rees. “Reusability Through Fine-grain Inheritance”. In: *Softw. Pract. Exper.* 22.12 (Dec. 1992), pp. 1049–1068. ISSN: 0038-0644. DOI: `10.1002/spe.4380221203` (cited on page 91).

[Ker04] J. Kerievsky. *Refactoring to Patterns.* Pearson Higher Education, 2004. ISBN: 0321213351 (cited on pages 27, 34, 36–40, 42, 53).

[Kho+12] F. Khomh et al. “An exploratory study of the impact of antipatterns on class change- and fault-proneness”. In: *Empirical Software Engineering* 17.3 (2012), pp. 243–275. ISSN: 13823256. DOI: `10.1007/s10664-011-9171-y` (cited on page 91).

[LD17] K.-K. Lau and S. Di Cola. *An Introduction to Component-based Software Development.* English. Singapore: World Scientific Publishing Co. Pte. Ltd., 2017. ISBN: 978-981-3221-87-1 (cited on page 24).

[LR06] M. Lippert and S. Roock. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully.* 1st ed. Wiley, May 2006. ISBN: 0470858923 (cited on pages 1, 20, 34–36, 38–42).

[M.07] S. M. *Software architecture refactoring. Tutorial.* 2007 (cited on page 91).

[Mar03] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003. ISBN: 0135974445 (cited on pages 91, 92).

[Mar17] R. C. Martin. *Clean Architecture: A Craftsman’s Guide to Software Structure and Design.* Robert C. Martin Series. Boston, MA: Prentice Hall, 2017. ISBN: 978-0-13-449416-6 (cited on page 55).

[Mey88] B. Meyer. *Object-Oriented Software Construction.* 1st. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. ISBN: 0136290493 (cited on page 91).

[Mfi] *Martin Fowler: "Refactoring ist heute relevanter als vor zwanzig Jahren".* 2018. URL: `https://www.heise.de/developer/artikel/Martin-Fowler-Refactoring-ist-heute-relevanter-als-vor-zwanzig-Jahren-4249380.html` (visited on 04/25/2019) (cited on page 1).

[Mic] *Microservices.* 2019. URL: `https://martinfowler.com/articles/microservices.html` (visited on 04/06/2019) (cited on page 3).

[Moh+10] N. Moha et al. “DECOR: A Method for the Specification and Detection of Code and Design Smells”. In: *IEEE Transactions on Software Engineering* 36 (Jan. 2010), pp. 20–36. DOI: `10.1109/TSE.2009.50` (cited on pages 91, 92).

[Mot] *Motivation for Software Architecture Refactoring.* 2017. URL: `https://dzone.com/articles/motivation-for-software-architecture-refactoring` (visited on 04/20/2019) (cited on page 6).

[MPC99] B. K. Miller, Pei Hsia, and Chenho Kung. “Object-oriented architecture measures”. In: *Proceedings of the 32nd Annual Hawaii International Conference on Systems Sciences. 1999. HICSS-32. Abstracts and CD-ROM of Full Papers.* Vol. Track8. 1999, 10 pp.–. DOI: `10.1109/HICSS.1999.773101` (cited on page 91).

[New15] S. Newman. *Building Microservices.* 1st. O’Reilly Media, Inc., 2015. ISBN: 1491950358, 9781491950357 (cited on page 3).

[Onl] *Service-Based Antipatterns.* 2019. URL: `https://xjreb.github.io/service-based-antipatterns/` (visited on 02/17/2019) (cited on pages 13, 15, 93).

[PJ88] M. Page-Jones. *The Practical Guide to Structured Systems Design: 2Nd Edition.* Upper Saddle River, NJ, USA: Yourdon Press, 1988. ISBN: 0-13-690769-5 (cited on pages 91, 92).

[Que] *Questionnaire.* URL: `https://en.wikipedia.org/wiki/Questionnaire` (visited on 04/08/2019) (cited on page 30).

[Ref] *Refactoring Guru. Remove Parameter.* URL: `https://refactoring.guru/remove-parameter` (cited on page 92).

[Res] *REST API Versioning.* 2019. URL: `https://restfulapi.net/versioning/` (visited on 04/28/2019) (cited on page 17).

[RFG05] J. Ratzinger, M. Fischer, and H. Gall. “Improving Evolvability Through Refactoring”. In: *SIGSOFT Softw. Eng. Notes* 30.4 (May 2005), pp. 1–5. ISSN: 0163-5948. DOI: `10.1145/1082983.1083155` (cited on pages 6, 92).

[Ric19] C. Richardson. *Microservices Patterns.* English. Shelter Island, NY 11964: Manning Publications Co., 2019. ISBN: 978-1-61729-454-9 (cited on pages 4, 15, 16).

[Rie96] A. J. Riel. *Object-Oriented Design Heuristics.* 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN: 020163385X (cited on pages 35, 91).

[Sem] *Semmle Code Tool.* URL: `https://semmle.com/` (cited on pages 91, 92).

[Son] *Sonargraph-quality: A tool for assessing and monitoring technical quality.* URL: `https://www.hello2morrow.com/products/sonargraph/quality` (cited on pages 91, 92).

[SSM06] F. Simon, O. Seng, and T. Mohaupt. *Code-Quality-Management - technische Qualität industrieller Softwaresysteme transparent und vergleichbar gemacht.* dpunkt.verlag, 2006, pp. I–XVII, 1–340. ISBN: 978-3-89864-388-7 (cited on page 91).

[SSS14] G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for Software Design Smells: Managing Technical Debt.* 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014. ISBN: 0128013974, 9780128013977 (cited on pages 5, 20, 27, 34–42, 53, 66, 91, 92).

[Str] *Structural Analysis for Java Tool (Stan4J.* URL: `http://stan4j.com/` (cited on pages 91, 92).

[Szy02] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201745720 (cited on page 23).

[TL18] D. Taibi and V. Lenarduzzi. “On the Definition of Microservice Bad Smells”. In: *IEEE Software* 35.3 (2018), pp. 56–62. ISSN: 0740-7459. DOI: `10.1109/MS.2018.2141031` (cited on pages 13, 14, 21, 33–39, 88, 93).

[Tri05] A. Trifu. “Automated Strategy Based Restructuring of Object Oriented Code”. In: (Jan. 2005) (cited on pages 91, 92).

[Uml] *UML Specification.* URL: `https://www.omg.org/spec/UML/2.2/About-UML/` (cited on page 91).

[Wha] *What are microservices?* 2019. URL: `https://microservices.io/` (visited on 04/06/2019) (cited on page 3).