Universidade do Minho
Escola de Engenharia

Francisco José Oliveira Freitas

Refactoring Java Monoliths into Executable Microservice-Based Applications

December, 2021

Universidade do Minho
Escola de Engenharia

Francisco José Oliveira Freitas

Refactoring Java Monoliths into Executable Microservice-Based Applications

Master dissertation
Integrated Master in Informatics Engineering

Dissertation supervised by:
Jácome Cunha
André Ferreira

December, 2021

Acknowledgements

I would like to thank my supervisor Dr. Jácome Cunha and my co-supervisor Dr. André ferreira for all the advice and continuous support during this last year.

I also want to thank INESTEC for the scholarship I was awarded that helped me to support my personal costs during this last period.

A special thanks to my family who provided me with all the means that allowed me to get here today. Without them, this would not have been possible.

Finally, I want to thank all my friends who were present in the most difficult moments and who provided me with unforgettable moments during the last 5 years.

This work is supported by the national funds through the Portuguese Funding Agency (FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020).

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

________________________, _______________________________________

(Place) (Date)

____

(Francisco José Oliveira Freitas)

Resumo

Refactoring de Monólitos em Java para Aplicações Baseadas em Microsserviços Executáveis

Nos últimos anos assistiu-se a uma mudança drástica na forma como o *software* é desenvolvido. Os projectos de *software* de grande escala estão a ser construídos através de uma composição flexível de pequenos componentes possivelmente escritos em diferentes linguagens de programação e com os processos de *deploy* independentes – as chamadas aplicações baseadas em microsserviços. Isto tem sido motivado pelos desafios associados ao desenvolvimento, manutenção e evolução de grandes sistemas de software, mas também pelo aparecimento da *cloud* e pela facilidade que trouxe em termos de escalabilidade horizontal, reutilização e flexibilidade na propriedade e no *deploy*. O crescimento dramático da popularidade dos microsserviços levou várias empresas a aplicar grandes *refactorings* aos seus sistemas de software. Contudo, esta é uma tarefa desafiante que pode demorar vários meses ou mesmo anos.

Esta dissertação propõe uma metodologia capaz de transformar automaticamente aplicações desenvolvidas em Java sob uma arquitetura monolítica em aplicações baseadas em microserviços. A metodologia proposta é direccionada para as aplicações que tiram partido da técnica ORM para relacionar classes com as entidades da base de dados, através de anotações no código fonte. A nossa abordagem recebe como *input* o código fonte e uma proposta de microsserviços, e aplica técnicas de *refactoring* às classes para tornar cada microsserviço independente. Esta metodologia cria uma API para cada chamada de métodos de classes que se encontram noutros serviços, e as entidades da base de dados também sofrem *refactoring* para serem incluídas no serviço correspondente. A metodologia proposta foi implementada através da construção de uma ferramenta que suporta o *refactoring* de aplicações desenvolvidas em Java *Spring* e que utilizam as anotações da *JPA* para o mapeamento entre as classes e as entidades.

Realizou-se um análise quantitativa e qualitativa em 120 projetos *open-source* aleatoriamente recolhidos do *GitHub*. Na avaliação quantitativa procurou-se perceber a aplicabilidade da metodologia e na analise qualitativa, através da execução de testes unitários, procurou-se avaliar se aplicação original e a aplicação baseada em microserviços gerada são funcionalmente equivalentes.

Os resultados são promissores sendo a metodologia capaz de realizar o refactoring em 69% dos projetos, sendo o resultado da execução dos testes unitários igual em ambas as versões dos projetos.

Palavras-chave: arquitetura de microsserviços, aplicações baseadas em microsserviços, decomposição de monólitos, Java, refactoring

Abstract

Refactoring Java Monoliths into Executable Microservice-Based Applications

In the last few years we have been seeing a drastic change in the way software is developed. Large-scale software projects are being assembled by a flexible composition of many (small) components possibly written in different programming languages and with independent *deploy* processes – the so-called microservice-based applications. This has been motivated by the challenges associated with the development, maintenance, and evolution of large software systems, but also by the appearance of the cloud and the ease it brought in terms of horizontal scaling, reusability and flexibility in ownership and deployment. The dramatic growth in popularity of microservice-based applications has pushed several companies to apply major refactorings to their software systems. However, this is a challenging task that may take several months or even years.

This dissertation proposes a methodology to automatically evolve a Java monolithic application into a microservice-based one. The proposed methodology is directed to the applications that take advantage of the ORM technique to relate Java classes to database entities, through annotations in the source code. The methodology receives as input the Java source code and a proposition of microservices and refactors the original classes to make each microservice independent. The proposed methodology creates an API for each method call to classes that are in other services. The database entities are also refactored to be included in the corresponding service. The proposed methodology was implemented by building a tool that supports the refactoring of Java *Spring* applications that use *JPA* annotations for mapping between Java classes and database entities.

We performed a quantitative and qualitative analysis on 120 open-source projects randomly selected from *GitHub*. In the quantitative evaluation we tried to understand the applicability of the methodology and in the qualitative analysis, through the execution of unit tests, we tried to evaluate if the original application and the application based on micro-services generated are functionally equivalent.

The results are promising, with the methodology being able to refactor 69% of the projects, with the unit test results being the same in both versions of the projects.

Keywords: microservice architecture, microservice-based applications, monolithic decomposition, Java, refactoring

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

“The death of big software” has been announced by Andriole, 2017. This has been motivated by the challenges associated with the development, maintenance, and evolution of large software systems, but also by the appearance of the cloud and the ease it brought in terms of horizontal scaling, reusability and flexibility in ownership and deployment. Therefore, in the last few years, there has been a dramatic growth in the use of cloud computing leading to companies like Amazon, Microsoft and IBM creating their cloud platforms providing various services for application development (Cito et al., 2015). Indeed, many software systems are currently being developed as a set of loosely-coupled components, possibly written in different programming languages, eventually deployed anywhere in the cloud, and communicating through the internet, creating an architectural style usually termed microservices (S. Newman, 2015). These pieces can be used to mix-and-match as to create new or even to evolve existing software (Andriole, 2017)

While microservices are not new, there is still much to investigate and therefore there are several investigations ongoing to improve their understanding in an attempt to formalize processes associated with them (Hamzehloui et al., 2019).

One of the main motivations of a microservice architecture is that it has the potential to increase the flexibility and agility of software development as each service can be developed and deployed individually using different technologies.

Although microservices are currently standard in industry, the fact is that there are still many applications that were (and still are) built as monoliths, that is, applications composed of all the core logic related to the domain of the problem contained in a single process (S. Newman, 2015). However, applications developed with this architecture, tend to grow in size as well as complexity, resulting in very complex monolithic systems leading to the disadvantages, such as scalability and team organization, starting to have more impact than the advantages of this paradigm, with the detection of bugs and the addition of new features being more complicated (Chen et al., 2017).

The manual process of migrating monoliths to this new paradigm is complex and, depending on the project's complexity, may take months or even years to complete (Fritzsch et al., 2019; Mazzara et al., 2018). The decomposition of software systems is one of the main struggles, and as shown in the work of Fritzsch et al., 2019, none of the participants in the study was aware of automated techniques that could assist the migration of a monolithic application to a microservice-based one. Migrating from a monolithic application to a microservices architecture is a costly process and full of unique challenges, as each system is itself unique with its own particularities (Kazanavicius & Mazeika, 2019), and this transition is mostly dependent on the experience and knowledge of the systems by the experts (Pahl & Jamshidi, 2016). Thus, the research community has been working on techniques and methodologies to aid in this migration, i.e.in transforming a monolithic application into a microservice-based one, while preserving the semantics of the original application (Chen et al., 2017; Gysel et al., 2016; Jin et al., 2021; Jin et al., 2018; Kamimura et al., 2018; Mazlami et al., 2017). Moreover, several companies have also applied major refactorings of their backend systems to transform their applications (Mazzara et al., 2018).

Most of the previous works are focused on the identification of the services, but lack the step of actually refactoring the application to make it a microservice-based one. The works that propose refactoring the application do not take into consideration issues such as the data decomposition of a monolithic system and the user interface.

In this dissertation we propose a methodology to refactor monolithic applications developed in Java which receives as input a list and composition of microservices. Our methodology analyzes the source code and the services proposed and refactors the classes that have method calls to other classes that are part of other services. Each of these calls is replaced by a call to a new method we automatically generate, implementing a REST call to the original method which now is a different service. We also refactor the database classes, applying patterns, as they may need to be spread by the different services too. We create a tool, MicroRefact, as a proof of concept which supports the refactoring of applications developed in Java Spring. To evaluate the applicability of our tool we did a quantitative evaluation, by measuring what percentage of applications the tool is able to refactor. We also perform a qualitative evaluation where we try to understand if the generated microservices-based application is functionally equal to the monolithic one. The applications used in the evaluation were extracted from *GitHub* and their unit tests were used for qualitative evaluation. The results obtained are promising, being the tool capable of producing results in 69% of the applications. Moreover, the execution of the unit tests had the same result in the generated application as in the original one for all applications.

1.1 Contributions

Our work has the following contributions to the area of migration to microservices:

- A methodology capable of refactoring a monolithic application into microservices and capable to decompose the database for the services, being the user free to choose which microservices to form.

- Implementation of a tool capable of applying the proposed methodology to Java *Spring* web applications that delivers to the user a microservices-based application functionally equivalent to the original.

- An extensive study of 120 open source applications and the use of unit tests to validate refactoring.

During this work, we also published a scientific article (Freitas et al., 2021):

Francisco Freitas, André Ferreira, Jácome Cunha.
Refactoring Java Monoliths into Executable Microservice-Based Applications.
In 25th Brazilian Symposium on Programming Languages, pp. 100–107, 2021.
New York, NY, USA: Association for Computing Machinery (ACM).
DOI: https://doi.org/10.1145/3475061.3475086.

1.2 Document Organization

The rest of this dissertation is structured as follows: Chapter 2 introduces the concepts of monolithic architectures and microservices; Chapter 3 describes the current state of the art in the microservices migration process; Chapter 4 presents in detail the proposed methodology for refactoring monoliths; Chapter 5 describe the implementation of MicroRefact; Chapter 6 presents in detail our evaluation; in Chapter 7 we draw our conclusions and describe some possible future work.

2 Background

To migrate monolithic applications to microservices it is necessary to know both architectures and how they work. In this chapter we present the core ideas of each architecture, their benefits and challenges.

2.1 Monoliths

This architecture is characterized by all functions are encapsulated in a single service, with the application contained in a single process (Kazanavicius & Mazeika, 2019). It is an architecture strongly recommended for new projects where the domain as well as the bounded context are not yet well defined. Over time, applications tend to grow leading to increased complexity. An approach well recognized by developers to mitigate the complexity inherent to application growth is to split applications over several layers (Fowler et al., 2002).

Typically, an application is divided into three layers: the presentation layer that interacts with the user or, through an API, exposes functionality; the business layer that contains all the business logic and implements all the system's functionality; the persistence layer that is responsible for making requests to the database and, in applications developed with the object-oriented programming paradigm, mapping classes to database tables. This layer usually only accesses a database that contains all the information about the application.

This architecture has multiple advantages, for new projects and projects of short duration or low complexity. It is simple to develop since the *IDEs* and other development tools are oriented to develop a single application (Richardson, 2018) and code reuse is simplified compared to a distributed system since the code already resides in the monolith (S. Newman, 2019). Also it is simple to test, the deployment process is easy since there is only one component, and it is easy to scale horizontally using multiple replicas of the system and a load balance to balance the load between the replicas (Kharenko, 2015).

However, as the size and complexity of the code grows it becomes more difficult to resolve bugs and add new features since it becomes so large that no developer can get a full understanding of the code, leading to the mistakes when changes are made to the code, leaving the monolith even more complex resulting in slower development cycles, eventually becoming a "big ball of mud" (Richardson, 2018). Regarding scalability, all components of the application are replicated although only some need more resources. As for the deployment although simple, in each software upgrade the deployment of the entire application is required. In terms of resilience, it is not very resilient to failure since a failure in one of the application modules could result in the failure of the entire system.

In general, monolithic architectures are sufficient to handle a large set of problems when one intends to decrease dependencies and keep coupling at a low level. However, the deployment, resiliency, and flexibility are the big drawbacks when one wants to make some changes to the monoliths.

2.2 Microservices

There are several definitions of microservices, some use the word's prefix and claim that services should have a small number of lines of code (e.g. 100 lines), while others claim that a service should only take two weeks to develop (Richardson, 2018).

S. Newman, 2019 describes microservices as services with independent deployment process that are modeled around the business domain, that use the network to communicate with each other, and expose their functionality through endpoints.

On the other hand, Fowler and Lewis, 2014 describe microservices as an application architecture composed of a set of small services, each one having its own process, its own technology stack and its own well-resolved set of tasks to execute, and each one being independent relatively to the process of deployment and scalability, and with a well-defined bounded context. Fowler and Lewis, 2014 also affirm a communication layer is needed between services because they are isolated and therefore typically each service exposes its own API, which will receive HTTP requests and respond according to the logic defined in the service

Although microservices are based on the SOA architectural style (Service Oriented Architecture), which also divides the system into a set of services, they have significant differences. These differences include the way data is manipulated, since in SOA there is a global data model and shared database, and communication between services using protocols such as SOAP (Simple Object Access Protocol) and smart pipes.

Advantages

The microservices architecture has multiple advantages over monolithic architectures. The advantages are inherited from the characteristics of distributed systems adopted and from taking to the extreme the philosophy of service-oriented architectures.

Enables the continuous delivery and deployment of large, complex applications.

Microservices follow the culture of devOps and enable continuous software delivery and integration (Richardson, 2018). In other words, they take advantage of automated pipelines responsible for performing tasks from automated testing to deploy to production, resulting in more frequent deploys since they are independent and service-focused, causing less downtime and allowing easier identification of the source of problems that reach production (S. Newman, 2015).

Maintenance

When a monolithic system becomes large and complex, the system comprehension curve for team members who do not know the source code is proportional to its size, reflecting in a delay in the delivery of new features and in the detection and resolution of bugs. However, a microservices architecture is composed of several small services, which are potentially easier to maintain since there is no need for familiarization with the whole system. Thus, the focus of the developers is on one or more services, being the process of detecting bugs and adding new features more efficient than the monolithic application.

Scalability

In monolithic applications when a certain module needs to be scaled, the entire application has to be scaled, reflecting a waste of computational resources since the other modules that compose the system are stable. In contrast, microservices allow resource-focused scalability, i.e. if a service is being overloaded with requests, it can be scaled horizontally, i.e. replicate the service, in a way that is transparent to all other services. This way, only the services that need it are scaled, resulting in a considerable saving of resources compared to a monolithic system. Due to the independence of service development, each service can be implemented on the most suitable hardware, which is not possible in a monolithic architecture (Richardson, 2018).

Autonomy

Managing a large team is challenging, but with microservices this process can be simplified. Usually large teams are divided into sub-teams and each of these is assigned at least one service. Each team is responsible for developing and deploying its service. Since the services are independent, these teams become autonomous because they do not depend on others to develop their service.

Heterogeneity of technologies

As mentioned before, the services are independent. The architecture of microservices does not have a commitment to a technological stack, being each service developed with the most appropriate technology as long as the contracts established between services are not affected, and there may be several technologies in the application's technological stack. Technological heterogeneity promotes the use of new technologies, and, in case of failure, one can always choose another one without putting the whole system in question, which is impossible in a monolithic system.

Fault tolerance

Dealing with failures is a key issue, especially in distributed systems. Microservices, by following a distributed system approach, have all the advantages associated with distributed systems regarding fault handling and are able to limit faults in the application to certain services, which is not possible in a monolithic system, where a fault can result in the collapse of the entire application. To mitigate failures in microservices, design patterns such as timeouts, retry and circuit breaker (Kumar, 2020) are used.

Disadvantages

The disadvantages associated with microservices are also found in distributed systems.

Communication overhead

In a microservices architecture, services communicate with each other over the network, while in a monolith the communication between components is in memory. Microservices can use two types of communication: synchronous communication, usually HTTP communication in REST APIs; asynchronous communication, typically *rabbitMQ* or *ZeroMQ*. These communications add a overhead that does not exist in a monolithic application, and can increase the response time (Hubbell, 2020).

Tests

Performing tests, such as integration tests, becomes more complex because of the need to create stubs of the remaining services to guarantee isolation of the fraction being tested from the rest of the system (S. Newman, 2015).

Logs

Like testing, the process of debugging by various services in the application is also more complicated, since the developer has to deal with the logs of the different services.

Deploy functionality across multiple services

Launching new functionality that spans different services requires careful coordination between the various development teams, necessitating a plan based on the dependencies between services for the deployment process. This process is different from that used in a monolithic applications since, in these, it is possible to deliver updates of multiple components atomically (Richardson, 2018).

Granularity of services

One of the challenges of migrating to a microservices architecture is that there is no concrete, well-defined algorithm for decomposing a system into services. To make things worse, if the decomposition of a system is done incorrectly, a distributed monolith will be built. A distributed monolith is a system in which sets of services must perform the implementation process together. A distributed monolith has the disadvantages of monolithic architecture and microservices architecture (Richardson, 2018).

3 State of the Art

The decomposition of systems into modules started to be explored by Parnas, 1972, who tried to demonstrate that the efficiency of modularization of a system depends on the criteria used to divide the system.

Since then, functional decomposition has remained an important topic in software engineering. As software systems grew and became more complex, there was a need to distribute the systems across networked infrastructures such as *web services* and remote objects (Kamimura et al., 2018).

The main challenges of this decomposition are the manual and tedious work accompanied by the difficulty in identifying the functional units given the need for a detailed analysis of several dimensions of the software architecture, and the quality of the final result is often strongly linked to the experience and knowledge of the specialist performing the decomposition (Kamimura et al., 2018).

In the next sections we describe several works discussing different kinds of decomposition of applications into microservices.

3.1 Manual Migration of Monoliths to Microservices

In this section we conduct a literature review of migration processes of monolithic architectures performed manually by experts in large-scale projects. The aim is to identify knowledge about patterns and methodologies used with the objective of collecting useful information for the automation of the process.

Balalaie et al., 2018, in order to improve the migration planning process and combat the *ad-hoc* approach, conducted a survey of *design patterns* through the analysis of industrial caliber application migration processes. In this work, an analysis of the entire migration process is performed, from architecture identification to the *deploy* process.

Since a *one-size-fits-all* methodology would not be functional, because each project has its own particularities, Balalaie et al., 2018 adopted an approach called *Situational Method Engineering* (SME) (Henderson-Sellers et al., 2014) , which allows adaptation of methods according to the specific situation (Balalaie et al., 2018).

From the analysis and application of the SME resulted a repository with 15 patterns and, through a case study, emerged a migration plan, Figure 1, able to handle the possible dependencies between the steps.

The identified patterns and the migration plan both have a significant impact on the work to be done because they introduce best practices into the migration process.

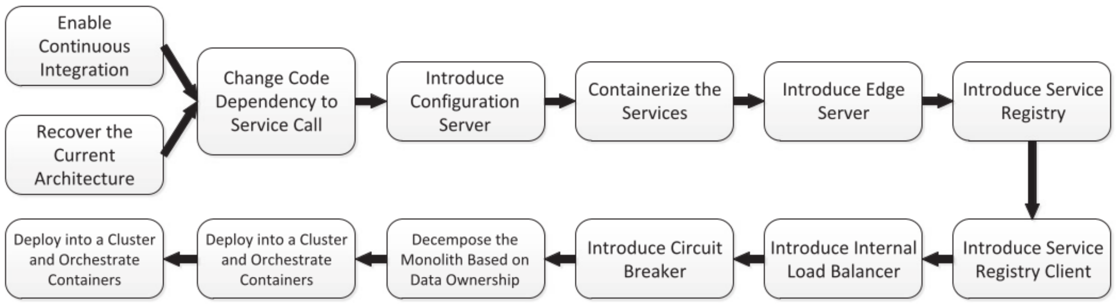

Enable Continuous Integration
Recover the Current Architecture
Change Code Dependency to Service Call
Introduce Configuration Server
Containerize the Services
Introduce Edge Server
Introduce Service Registry
Introduce Service Registry Client
Introduce Internal Load Balancer
Introduce Circuit Breaker
Decempose the Monolith Based on Data Ownership
Deploy into a Cluster and Orchestrate Containers
Deploy into a Cluster and Orchestrate Containers

Figure 1: Proposed Migration Plan

Source: (Balalaie et al., 2018)

Fowler, 2004 suggests an incremental migration, which consists of gradually building a new application by extracting functionality from the monolith avoiding a *big bang* rewrite. Thus, the generated application consists of a set of microservices that interact with the monolithic application. Over time, the number of functionalities implemented by the monolith tends to decrease as they are migrated to microservices, until the monolith disappears and becomes a microservice (Richardson, 2018). Fowler terms this pattern as *Strangler application*.

The main advantage of an incremental migration is the beginning of the return on investment with the first extraction of a service, since there is a partial migration of the monolith, which is not possible in a *big bang rewrite* that only delivers benefits when it is complete.

A partial migration implies major modifications to the source code to make it possible for microservices and monolith to interact, as well as maintain data consistency. To prevent extensive modifications Richardson, 2018 presents three strategies to “strangle” the monolith following an incremental approach which we now describe.

The first, implementation of new functionality through services, proposes adding new functionality to the monolith by creating new services, preventing the monolith from growing, becoming more complex, difficult to understand, and ungovernable over time (Richardson, 2018). For the new service to be able to collaborate with the monolith, it is necessary to create adapters in both the monolith and the service that use a mechanism for communication between processes, typically *representational state transfer* (REST).

The second, separation of *frontend* from *backend*, suggests the separation of the presentation layer from the logic and database layers, which allows to develop, perform the deployment process, and scale the two services independently. After the separation, the *frontend* service makes remote calls to the *backend*. However, this is a partial solution since the *backend* remains complex.

The third, splitting the monolith by extracting functionality to services, proposes the decomposition of the monolith by extracting features for services. By using this strategy the monolith tends to disappear over

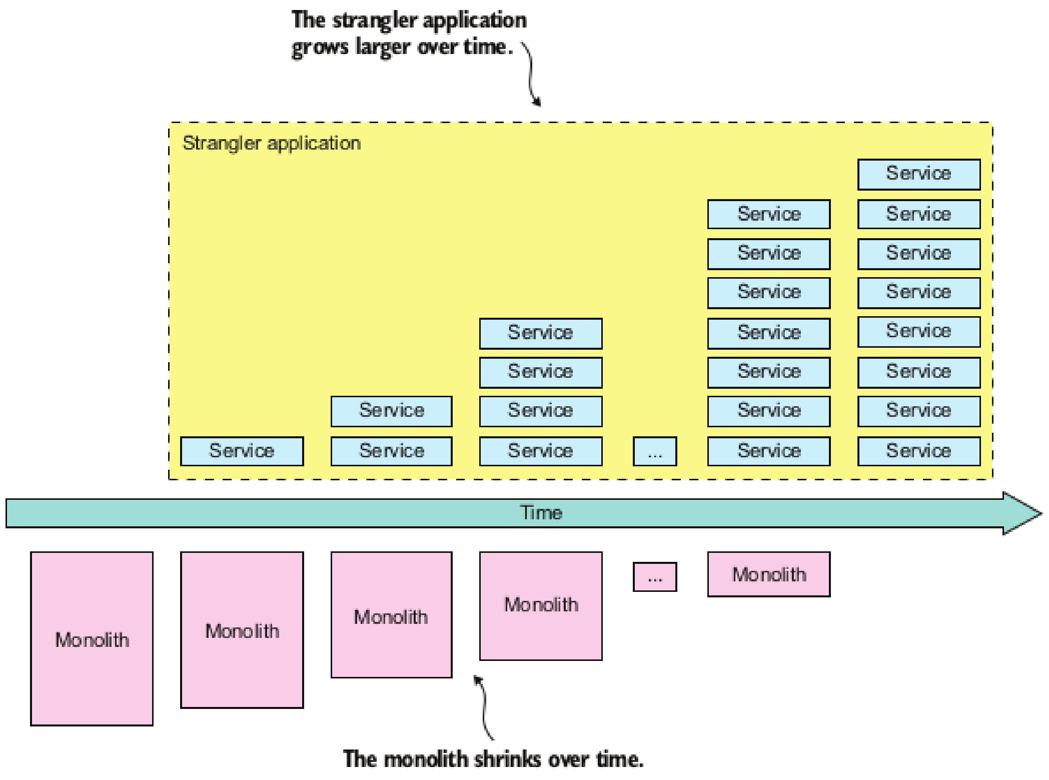

The strangler application grows larger over time.
Strangler application
Service
...
Time
Monolith
The monolith shrinks over time.

Figure 2: Strangler application pattern by Fowler

Source: Richardson, 2018

time and the number of microservices grows. To extract a feature it is imperative to make a vertical cut in the monolith (Richardson, 2018), because it may have dependencies on all layers of the monolith, however this task is not trivial. Problems such as splitting the domain model, breaking dependencies between objects, and *database refactoring* appear, and their solutions are tightly coupled to the system being migrated. Regarding domain model splitting and consequently breaking dependencies between objects, based on *Domain-driven design* (Evans, 2004) , Richardson, 2018 suggests the use of the *aggregate* pattern that replaces the direct reference between objects with a reference based on the primary key. The replacement of the direct reference between objects by a reference based on the primary key plays a central role in the solution we propose. For the *refactoring of the database*, in order to avoid major changes in the code, he proposes a solution of Ambler and Sadalage, 2006 which consists in preserving the original logical schema of the monolith for a transition period and using *triggers* for synchronization between the original schema and the new schemas. Preserving the original schema of the monolith significantly reduces the work that needs to be done immediately and over time the code can be migrated to the new schema.

3.2 Source-code oriented solutions

Since the main goal is to transform a monolithic application into a microservices-based application through a process of *refactoring* the source code of the monolith, the analysis of source code oriented solutions plays a central role in obtaining knowledge about existing *refactoring* techniques in the literature that could be useful for our solution.

In order to mitigate the problems during the transition of applications to services in the *cloud*, Kwon and Tilevich, 2013 propose *Cloud Refactoring*, a set of automated *refactoring* techniques, implemented in the *Eclipse* IDE, that facilitate the process of transforming applications to support services in the *cloud*. These techniques provide feature extraction for services, remote access to services, fault handling, and replacement of client accessed services with equivalent services in the cloud.

To assist the decision of which features to migrate, *Cloud Refactoring* provides a tool that, through coupling metrics, indicates which classes are the least coupled. For this it has two recommendation mechanisms available: profile-based recommendation and cluster-based recommendation.

This tool allows the developer to have full control of the features that will be migrated by accepting as input the set of methods or fields that are to be migrated, using *proxys* and interfaces, Figure 3 and 4, to break dependencies between classes.

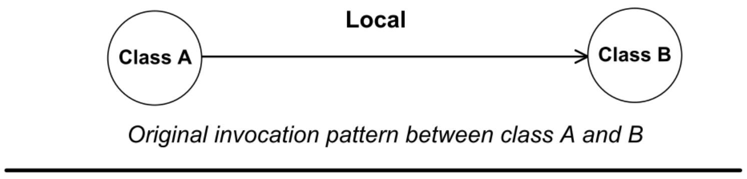


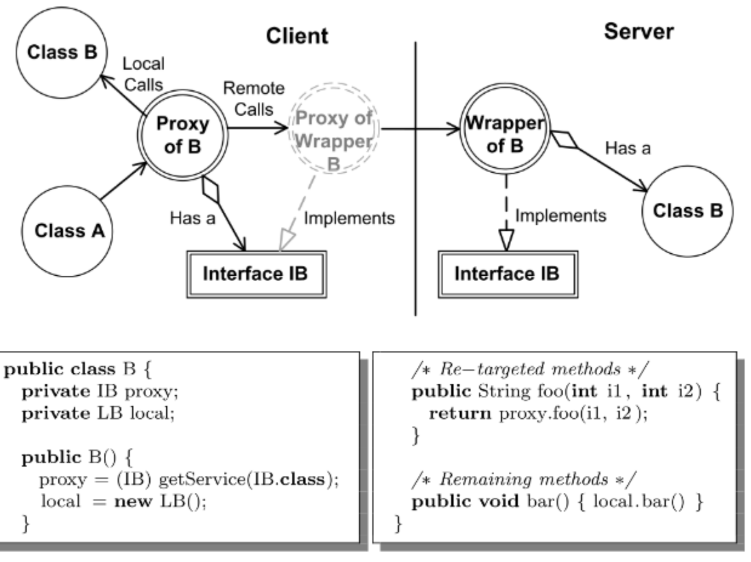
Figure 3: Original communication between classes.

Source: Kwon and Tilevich, 2013

To make the application resilient to failures, *Cloud Refactoring* provides three strategies with recognized value: *retry*, active replication and passive replication. In addition, new strategies can be added through a *Fault Tolerance Description Language* script (*services in the cloudFTDL*) (Edstrom & Tilevich, 2012).

Kwon and Tilevich, 2013's work proposes a partial solution to the problem that is intended to be addressed since it transforms a monolithic application into an application with a SOA architecture. However it presents some limitations: it uses classes as the atomic unit of identification and extraction of microservices, making it impossible to segregate components of a class for several microservices; it does not allow the full decomposition of the system into services since there may be strongly coupled classes; it does not support bidirectional communication between client and server.

With the goal of proposing microservices, Kamimura et al., 2018 developed a method to identify microservices through source code using *SArF*, a *clustering* algorithm proposed by Kobayashi et al., 2012. The extraction process begins by identifying the *endpoints* of the application API, using the *@Controller*



```
public class B {
  private IB proxy;
  private LB local;

  public B() {
    proxy = (IB) getService(IB.class);
    local  = new LB();
  }
```

```
  /* Re−targeted methods */
  public String foo(int i1, int i2) {
    return proxy.foo(i1, i2);
  }

  /* Remaining methods */
  public void bar() { local.bar() }
}
```

Figure 4: Communication after extraction of the methods from B to a cloud service.

Source: Kwon and Tilevich, 2013

annotation that identifies the *endpoints* exposed in the REST API. Next, the classes responsible for defining the persistence and processing of the data are identified through the *@Entity* and *@Table* annotations. Our solution also uses the annotations to extract information that helps identify the role of the classes. In this work of Kamimura et al., 2018 only a microservices proposal is performed, there is no extraction or *refactoring* of the code. As opposed, our solution refactors the code.

Abdullah et al., 2019 propose an automatic method for decomposing the monolith into microservices that aims to improve scalability and application performance through a black-box approach based on *logs* and machine learning algorithms that do not require supervision. This method also helps identify the most appropriate types of virtual machines for each microservice. The solution is based on to identifying uniform resource identifiers (URIs) of a given monolithic application in order to form groups of URIs and subsequently microservices. To do this, an analysis of the *access logs* is performed in order to collect the size of the response document and response time for each request for URIs and then group them using this information as criteria. Despite being a work in which there is an automatic decomposition of the monolith, Abdullah et al., 2019 not address which *refactoring* techniques are used and neither how the *deploy* process is done, giving preference to explaining the algorithm for selecting the most suitable virtual machine for each microservice.

Brito et al., 2021 propose a methodology capable of identifying sets of components candidates to form microservices automatically using topic modeling and clustering techniques. To do this, a lexical extraction of all relevant terms is performed to identify what a given component represents in the context of the project domain and an extraction of structural dependencies between components is performed by analyzing an *Abstract Syntax Tree* (AST). Next, the *Latent Dirichlet Allocation* algorithm is applied (Blei et al., 2003), to identify the most relevant k-topics in the source code. Finally, a graph is constructed based on the structural dependencies and the distribution of the k-topics by the components of the monolith with the final objective of applying the *Louvain* algorithm, proposed by Blondel et al., 2008, on the graph with the intention of grouping the components and consequently arriving at the microservices proposal. This work has as one of its great advantages being one of the few source code oriented solutions that offers a tool capable of identifying potential microservices, however this only supports projects developed in Java.

Asseldonk, 2021 tried to understand what effect have the use of various viewpoints of the system on the decomposition of the monolith into microservices. For that he created a tool that generates a proposed decomposition of monoliths developed in Python through the combination of static, dynamic and semantic analysis. The tool extracts the semantic, static and dynamic dependencies from the application under analysis and creates four edge-weighted graph, one for each type of dependencies and another in which it combines all the dependencies. Then, for each graph the *Louvain* algorithm (Blondel et al., 2008) is applied to find potential microservices. To compare the quality of the proposed microservices Asseldonk, 2021 uses metrics proposed by the work of Jin et al., 2021 regarding independence of functionality and modularity. This work ends up going further than Brito et al., 2021 because it generates microservices proposals based on several types of analysis.

3.3 Model-based Solutions

Model-based solutions allow the use of models to support migrations since models also represent a view over the interactions between system's components (Chen et al., 2017).

Gysel et al., 2016 propose a tool, *Service Cutter*, that through artifacts, such as use cases and domain models, extracts coupling information that is represented by a graph. Weights are added to the edges of the graph according to a set of prioritized criteria, in order to identify clusters of components, and consequently good candidates for microservices. For this purpose, at this point, two clustering techniques are available: *Epidemic Label Propagation* by Leung et al., 2009 and the Girvan and Newman, 2001 algorithm. The criteria available in the *Service Cutter* were obtained through a literature review and the authors' experience leading to the construction of a catalog with 16 coupling criteria, Figure 5, being these divided into 4 categories. The *Service Cutter* presents a promising technique for migrating to microservices but has some limitations: it is highly dependent on the quality of the software artifacts provided by the user, and this fact has a direct impact on the solution; the artifacts have to be imported into the tool through *JSON* files, leading to a conversion process that may result in degradation of the artifact quality.

Tyszberowicz et al., 2018 propose an approach based on a use case specification of the software requirements and a functional decomposition of these requirements. Through tools such as *easyCRC* (Tyszberowicz and Raman, 2007) and *Text Analysis Online* the nouns and verbs of the use case specifications are extracted in order to obtain information about the system operations as well as the state variables. To organize the data obtained, a table is created in which the operations and their relationship with the variables are stored. With the intention of proposing a decomposition in high cohesion and

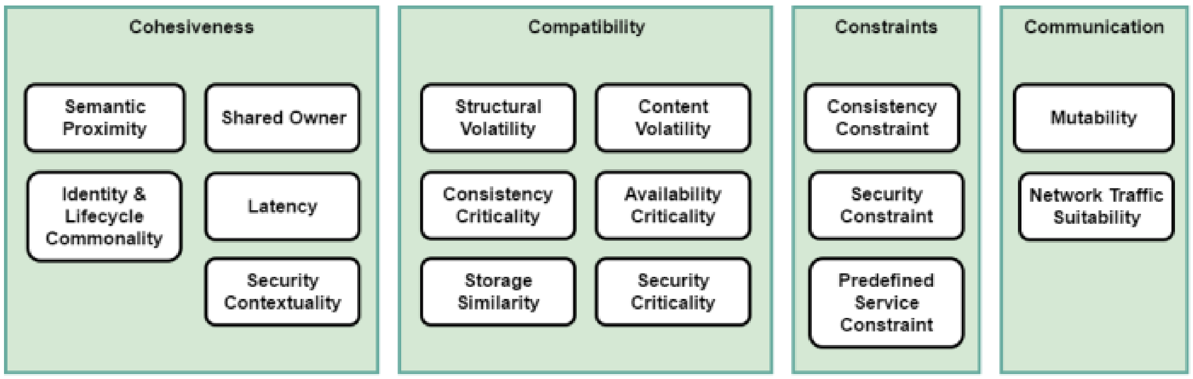


Figure 5: Coupling criteria by Gysel et al., 2016

Source: Gysel et al., 2016

low coupling components, through the operation/relationship table, a bipartite graph is built in which the nodes represent state variables and operations. The edges of the graph connect operations to the state variables they manipulate, with writes having a greater weight than reads. By visualizing the graph they identify clusters of components, and consequently candidates for microservices. Finally, this approach defines the API for each microservice and the stored data. The API is built by joining the operations that edit the microservice state variables and the microservice operations invoked by others. For the database, on the other hand, they follow the ideology of Yanaga, 2017 and Fowler and Lewis, 2014 which defend the sharing of data between microservices, with the data from external microservices being available through the API that contains it, not allowing any direct access to the database outside the service.

Like *Service Cutter*, this approach is highly dependent on the quality of *input*, yet it addresses issues such as API and database that are vital to the our work.

3.4 Summary

Based on the research and analysis of the bibliography presented in the previous sections, we conclude that there are already some works dedicated to the migration from monoliths to microservices. However, most of these works focus on the identification of microservices and not on monolith refactoring process. Instead, there is an indication of how the monolith should be decomposed. However, there are also some works that propose a methodology accompanied by a tool that performs the refactoring process like "Cloud Refactoring" and Abdullah et al., 2019's work but both have some limitations. The "Cloud Refactoring" cannot refactor classes that use parameter passing, serialization, and local resources such as databases and disk files. On the other hand, Abdullah et al., 2019's work uses dynamic analysis through the analysis of access logs and this has some limitations: the first is the overhead on the system to collect execution logs, which can be a problem on large scale systems; secondly, their approach is limited by the quality and coverage of the test cases performed to execute and collect their logs. Creating tests in an incomplete manner and with inadequate coverage will also result in an inadequate microservices proposal (Brito et al., 2021). Manual approach, despite being manual, introduce concepts and ideas that help in building an automated approach for refactoring monoliths into microservices. Since there are already several proposals for identifying microservices, our focus is on the next step, i.e. refactoring the monolith. Our goal is to propose a methodology for refactoring a monolithic application into a microservices-based application. We intend to address problems such as database decomposition and communication between services. Since our focus is on refactoring the monolith we do not address the identification of microservices problem, instead our methodology receives as input a microservice proposal that can be generated manually or can be generated by a tool for this purpose.

4

A Methodology for Refactoring Java Monoliths into Microservice-Based Applications

Given the high complexity of creating a generalized solution for multiple languages and frameworks, we reduced the scope to something more specific. In terms of web languages, *PHP, JavaScript, Java, C# and Python* stand out. Despite being a widely used language, *PHP* has not had a significant adaptation in the microservices world. In recent years, the *JavaScript* through *frameworks* such as *Express* has gained notoriety in this world since it is used in projects of small and large dimensions, however, many of these are already developed under a microservices architecture. Of the others, the *Java* stands out for being a language that over the years has remained as one of the most used for web development and there are numerous monolithic applications and therefore we chose to create a methodology to evolve monoliths developed in Java. In this chapter, we describe the proposed methodology for refactoring monolithic applications developed in Java into microservice-based applications.

The proposed methodology is directed to applications that take advantage of object-relational mapping (ORM) technique, which converts the data between the database entities and Java classes. This technique, in the Java language, can be applied through XML files or by annotations in the application's source code. The methodology is oriented to applications that use annotations to apply ORM technique. We chose this subset of applications because the use of ORM through annotations is a modern technique and to keep the focus of refactoring on the Java classes and avoid pre-processing configuration files for each project. Furthermore, since the entities and the relationships between entities are defined in the source code, through annotations, it is possible to decompose the database by microservices since the changes made in the source code will directly impact the logical database schema. Figure 6 depicts an overview of the steps that compose the refactoring process.

Our methodology receives, as input, the source code of the application under analysis and a set of microservices, in which each microservice is described by the set of classes. Each class must belong to just one microservice. Our methodology has as its output a microservices-based application that from the functional point of view is the same as the monolithic application under analysis.

The methodology is divided into 3 phases, the Information Extraction, the Database Refactoring and the Code Refactoring. In the Information Extraction phase, we extract the structural information from the

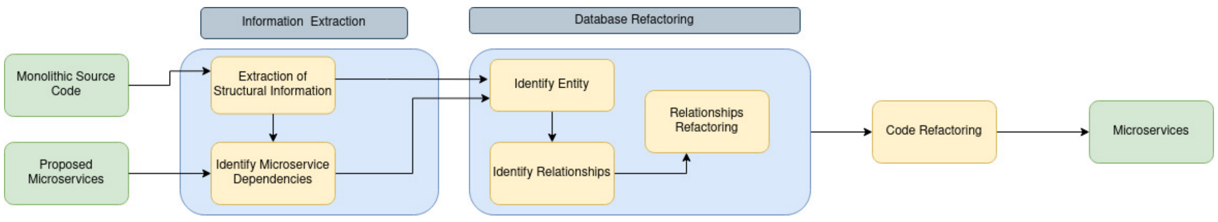


Figure 6: Overview of the proposed methodology

source code of the project under analysis and combine it with the microservices proposal to identify the dependencies between microservices. In the next phase, Database Refactoring, we use the structural information and the dependencies between microservices to identify entities, relationships between entities, and to identify which relationships need refactoring, proceeding to the refactoring of those relationships. Finally, in Code Refactoring, we use the structural information and the dependencies between microservices to analyze the class variables and the dependencies between classes. This analysis allows us to identify and refactor the classes that have dependencies with classes that belong to different microservices. To demonstrate what transformations our methodology makes to monolithic projects, we use as an example a Java *Spring* application called *restaurantServer*[1].

In the following sections we present the *restaurantServer* application and explain in detail each of the phases of our methodology accompanied with examples of the transformations performed on the monoliths.

4.1 *restaurantServer*

In this section we present the *restaurantServer*, a backend application for restaurant management. Although it is a small project, we chose to use this application as example because it covers a large portion of the cases we consider. Our goal is to present extracted examples from this application to demonstrate what happens in each phase of our methodology, and to demonstrate the concrete modifications that each phase of our methodology makes to monolithic applications. To achieve this we present examples of interactions between classes in the original application and the same interactions in the generated microservices based application.

To generate a microservices proposal for *restaurantServer*, to be used as input for our methodology, we used the tool developed by Brito et al., 2021 available at https://github.com/miguelfbrito/microservice-identification. Although the tool allows the customization of the input, we use the default parameters. The microservices proposal generated by the tool is present in Table 1. The generated proposal consists in 7 microservices and by the name of the classes we can understand what is the domain of each microservice. The examples presented in this chapter will be based on this microservices proposal. In the next section we present the first phase of our methodology, the Information Extraction .

[1] https://github.com/asledziewski/restaurantServer

#Microservice	Classes[a]
1	security.WebSecurityConfiguration security.jwt.JwtAuthEntryPoint security.jwt.JwtAuthTokenFilter security.jwt.JwtProvider
2	security.service.UserDetailsServiceImpl repository.UserRepository entity.User security.service.UserPrinciple service.UserService controller.UserController
3	entity.Bill entity.BillPosition service.BillService controller.BillController service.BillPositionService controller.BillPositionController repository.BillPositionRepository repository.BillRepository
4	service.RTableService entity.RTable controller.RTableController repository.RTableRepository
5	entity.Dish RestaurantServerApplication repository.DishRepository service.DishService controller.DishController
6	repository.ReservationRepository entity.Reservation service.ReservationService controller.ReservationController
7	form.response.JwtResponse email.MailService controller.AuthController entity.Role repository.RoleRepository form.LoginForm form.SignUpForm

Table 1: Microservices proposal for restaurantServer

[a]All classes have the prefix pl.edu.wat.wcy.pz.restaurantServer. in their qualified name

4.2 Information Extraction

The building blocks of our methodology are the structural information and proposed microservices. Since our methodology receives as part of the input a microservices proposal, which in practice is the decomposition of the monolith classes by microservices, these classes may have interactions with classes that belong to different microservices, i.e. have a dependency with a class that after refactoring will exist in a different environment. It is these dependencies that we need to identify and refactor. Therefore, to identify the dependencies between classes that belong to different microservices we need to extract information from the application source code and cross-reference it with the microservices proposal.

In the Information Extraction phase, structural information is extracted from the source code and dependencies between microservices are identified. Next, we describe how to obtain this information.

4.2.1 Extraction of structural information

Since the interactions between the classes are present in the structural information of the source code, we extract it from the source code of the system under analysis.

We extract the structural information through the *Abstract Syntax Tree* (AST) of the application under analysis. We chose to extract from AST because the class interactions are straightforward to obtain working on the AST and because we obtain the information contained in each class, such as name, imports, classes from which it extends, if applicable, implemented interfaces, annotations, variables, methods and invoked methods which is information that we need to build the classes into their respective microservices. To identify the classes that a class interacts with, we combine the list of invoked methods, the list of implemented interfaces and the list of extends, which are returned by the AST. The Algorithm 1 presents how the identification of classes which a class interacts with is performed. Receives as input a class C and returns a class C with the list of dependencies with other classes filled in. An iteration through the lists mentioned above is performed to extract the names of the classes that class C interacts with. Throughout the iteration, the classes' names are stored in Dep_C and finally the information contained in Dep_C is stored in C which results in the list of dependencies of C with other classes. The dependency list contains the name of the classes from which the class invokes methods, the name of the interfaces it implements and the name of the super class, if applicable.

Algorithm 1 Identification of classes where a class interacts with.

```
Input: $C$
Output: $C$
  $i = 0;$
  $Dep_C = [];$
  $methods = C.getInvoked\_methods();$
  $interfaces = C.getInterfaces();$
  $extends = C.getExtends();$
  for $(i; i < size(methods); i++)$ do
      if $(methods[i].targetClasse\ not\ in\ Dep_C)$ then
          $Dep_C.add(methods[i].targetClasse);$
      end if
  end for
  $i = 0;$
  for $(i; i < size(interfaces); i++)$ do
      if $(interfaces[i].name\ not\ in\ Dep_C)$ then
          $Dep_C.add(interfaces[i].name);$
      end if
  end for
  $i = 0;$
  for $(i; i < size(extends); i++)$ do
      if $(extends[i].name\ not\ in\ Dep_C)$ then
          $Dep_C.add(extends[i].name);$
      end if
  end for
  $C.setDependencies(Dep_C);$
```

For example, from the AST of *restaurantServer* the information extracted for the `ReservationController` class is presented in Listing 4.1.

```
{"name": "pl.edu.wat.wcy.pz.restaurantServer.controller.
   ReservationController",
 "imports":["lombok.AllArgsConstructor", "org.springframework.http.
    HttpStatus", "org.springframework.web.bind.annotation", "org.
    springframework.web.server.ResponseStatusException", "pl.edu.wat.
    wcy.pz.restaurantServer.entity.Reservation", "pl.edu.wat.wcy.pz.
    restaurantServer.service.ReservationService", "java.text.
    SimpleDateFormat","java.util.Collection", "java.util.Date","java.
    util.Optional"],
 "extendedTypes":[],
 "implementedTypes":[],
 "annotations":["@AllArgsConstructor","@RestController","@CrossOrigin
    "],
 "instance_variables":[{"annotations":[], "modifier":"private ", "
    identifier":[], "type":"ReservationService", "variable":"
    reservationService", "lineBegin":20,"lineEnd":20}],
 "myMethods":{"getReservationById":{...}, "addReservation":{...}, "
    updateReservation":{...}, "getReservations":{...}, "
    deleteReservation":{...}, "getCurrentReservations":{...}
 },
 "methodInvocations":[{"methodName":"getReservations","
    targetClassName":"pl.edu.wat.wcy.pz.restaurantServer.service.
    ReservationService"},{"methodName":"getCurrentReservations","
    targetClassName":"pl.edu.wat.wcy.pz.restaurantServer.service.
    ReservationService",{"methodName":"getReservationById","
    targetClassName":"pl.edu.wat.wcy.pz.restaurantServer.service.
    ResrvationService"},{"methodName":"getDate","targetClassName":"pl
    .edu.wat.wcy.pz.restaurantServer.entity.Reservation"},{"
    methodName": "setDateDays", "targetClassName":"pl.edu.wat.wcy.pz.
    restaurantServer.entity.Reservation"},{ "methodName":"setDateTime
    ", "targetClassName":"pl.edu.wat.wcy.pz.restaurantServer.entity.
    Reservation"},{"methodName":"addReservaion", "targetClassName":"
    pl.edu.wat.wcy.pz.restaurantServer.service.ReservationService"},{
    "methodName":"updateReservation","targetClassName":"pl.edu.wat.
    wcy.pz.restaurantServer.service.ReservationService"},{"methodName
    ":"deleteReservationById","targetClassName":"pl.edu.wat.wcy.pz.
    restaurantServer.service.ReservationService"
 }]
}
```

Listing 4.1: Information extracted from the AST about *ReservationController* class

And after the identification of the classes which the class `ReservationController` interacts with, the list presented in Listing 4.2 is added to the `ReservationController`.

```
{
"dependencies": ["pl.edu.wat.wcy.pz.restaurantServer.service.
   ReservationService", "pl.edu.wat.wcy.pz.restaurantServer.entity.
   Reservation"]
}
```

Listing 4.2: List of classes that interact with *ReservationController* class

This process is repeated for all classes of the monolith under analysis, and at the end of this process all interactions between classes are known, which is fundamental to identify which of these interactions are between classes of different microservices, which is the next step.

4.2.2 Identify Microservice Dependencies

We define dependencies between microservices as a reference to a certain non-primitive type that does not belong to the microservice, i.e., if microservice A has a class that depends on data type T and microservice B has the class of type T then A depends on B.

The process of identification of dependencies between microservices and the identification of dependencies between classes from different microservices is presented in Algorithm 2. To obtain the dependencies between microservices and the dependencies between classes of different microservices, we used the proposed microservices, $Microsercices\_Proposal$, by iterating over each microservice ms to check if the classes that belong to the dependency list of the classes that belong to ms exist in the classes that belong to ms. Through this comparison, we observe that the classes that are in the list of dependencies of a class that belongs to ms and that do not belong to ms are classes that belong to another microservice resulting in a dependency between microservices. When a dependency between microservices is detected, a pair is added to $dep\_microservice$ to indicate which microservices are involved in the dependency and the class name that class c depends on is added to update the dependency list. At the end of the process, the dependency list of each class only stores the dependencies with classes of other microservices.

Algorithm 2 Identification of dependencies between microservices

```
Input: $Microsercices\_Proposal = [ms_1, ms_2, ..., ms_n]$,
       $Classes = [c_1, c_2, ..., c_m]$,
Output: $dep\_microservice$                    ▷ List of dependencies between microservices
  $dep\_microservice = set();$
  for $(i = 0; i < size(Microservices\_Proposal); i++)$ do
      $ms = Microservices\_Proposal[i];$
      for $(j = 0; j < size(ms); j++)$ do
          $class\_name = ms[j];$
          $c = Classes.getClasse(class\_name);$
          $dep\_class\_microservices = [];$
          $dependencies\_list = c.getDependencies();$
          for $(t = 0; t < size(dependencies\_list); t++)$ do
              $dep\_name = dependencies\_list[t];$
              if ($dep\_name$ not $in\ ms\ \&\&\ dep\_name$ not $in\ dep\_class\_microservices$) then
                  $dep\_class\_microservices.add(dep\_name);$
                  for $(k = 0; i < size(Microservices\_Proposal); k++)$ do
                      if $dep\_name\ in\ Microservices\_Proposal[k]$ then
                          $dep\_microservice.add((i, k));$
                      end if
                  end for
              end if
          end for
          $c.setDependencies(dep\_class\_microservices);$
      end for
  end for
```

Using as an example the proposed microservices for the *restaurantServer* presented in Table 1 and the information obtained for the class `ReservationController` in the previous section, we can observe that the classes which the `ReservationController` class depends on belong to the same microservice as the `ReservationController`, so there is no dependency between microservices detected by this class. Regarding the `ReservationController` dependencies list, it is empty since there are no dependencies with classes of other microservices. For an overview of the proposed microservices and *restaurantServer* we found 16 dependencies, presented in Table 2, between microservices. The list of dependencies of each class with classes of other microservices can be observed in Appendix A.1, where in bold is highlighted the analyzed class followed by the classes it depends on .

Microservice	Depends on
1	2
2	6;7
3	4;5
4	3;6
5	2;4;7
6	2;4;7
7	1;2;6

Table 2: Dependencies between microservices

Since the composition of microservices given as input may not have followed any microservice identification methodology, the composition of microservices proposed can have some incongruities. Sometimes it is necessary to make some adjustments to the microservice proposal given as input. If the microservices proposal indicates that an interface implemented by a class is in different microservices, we replicate the interface and place the copy in the microservice where the implementing class belongs, since an interface only provides the signature of the methods that a class must implement, not having a significant impact on the microservice domain. On the other hand, regarding inheritance, our methodology does not allow the super class and sub classes to be in different microservices because they have an “is a” relationship and must belong to the same domain and therefore the same microservice.

4.3 Database Refactoring

One of the big challenges of migrating a monolithic system to microservices is database refactoring. Typically, each microservice has its own database which, from the perspective of migrating a monolithic system to microservices, leads to the need to decompose the monolithic database into several databases. The database decomposition is not a trivial process, so we have to consider issues such as transactional integrity, referential integrity, joins and latency (S. Newman, 2019).

The database refactoring phase aims to identify entities, relationships between entities, and refactoring the relationships between entities that belong to different microservices.

Having computed the structural information and the dependencies between classes from different microservices, we can now refactor the application under analysis starting with the database. We use the structural information extracted in the previous phase to identify the classes that are mapped as entities and the relationships between the entities. We use the annotation list of each class to identify the classes that are mapped as entities and through the annotations of the instance variables we identify the relationships. Table 3 shows the entities and relationships present in *restaurantServer*. The logical schema of the database is defined by 7 classes and 6 relationships.

Entity	Relationship	Entity
User	Many-to-Many	Role
User	One-to-Many	Reservation
Bill	One-to-Many	BillPosition
BillPosition	Many-to-One	Dish
Rtable	One-to-Many	Reservation
Rtable	One-to-Many	Bill

Table 3: Relationship between Entities

With the breakdown of the monolith into microservices, it is necessary to verify the integrity of the relationships between entities. As we are in the scope of applications that take advantage of the use of annotations to apply the ORM technique, when relationships between entities are identified, in terms of code this translates into a dependency between classes that needs to be handled. By refactoring the classes involved in the relationship we refactor the relationship of the database entities. Our methodology maintains the relationships between entities that belong to different microservices, using foreign keys to secure these relationships.

After a comprehensive and exhaustive review of the state of the art on migration patterns, we decided to apply the following patterns for database refactoring:

- *Data Transfer Object (DTO)*
- *Move Foreign-Key Relationship to Code*
- *Database Wrapping Service*

Next, we present each pattern and in which scenarios they are applied through examples.

Data Transfer Object

The data transfer object pattern (DTO) is a distribution pattern used to reduce the number of calls when working with remote interfaces. When working with remote interfaces (e.g web services), each service call is an expensive operation. Indeed the majority of the cost of each call is associated with the round-trip time between the client and the server. Therefore the solution is to transfer more data within each call. So the solution is to create a data transfer object that can hold all the data for the call. It needs to be serializable to go across the connection. Usually it is used an assembler on the server side, as shown in the Figure 7, to transfer data between the DTO and any domain objects. Another advantage associated with this pattern is encapsulating the serialization mechanism for transferring data over the wire, keeping this logic out of the rest of the code and also providing a clear point to change the serialization if desired (Fowler et al., 2002).

How do we use it in the refactoring process.

When a relationship between entities that belong to different microservices is identified, it means that at least one of the classes that are mapped as entities has an instance variable with the same data type as the other entity/class involved in the relationship. As the entities/classes involved in the relationship come to exist in different environments since they are in different microservices, one of the classes has an instance variable with a data type unknown to its domain. We apply the DTO pattern to create this data type. In this way, the unknown data type comes to exist, avoiding changes in the classes that have references to this data type.

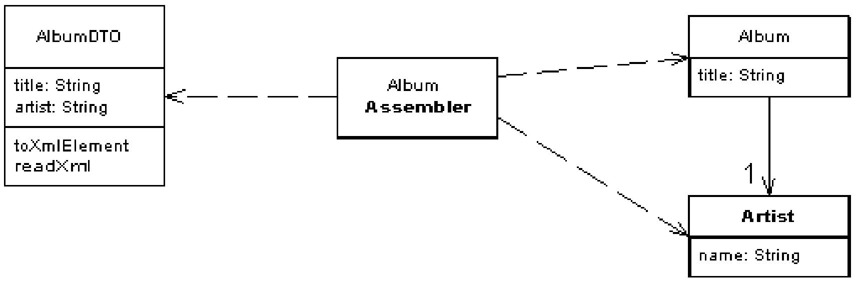


Figure 7: Data transfer Object Pattern

Source: Fowler, 2002

To demonstrate how this pattern is used we use the relationship between the classes `User` and `Reservation` of *restaurantServer*. By the microservice proposal in Table 1 and the class dependencies we verify that the `User` class depends on the `Reservation` class and the classes are in different microservices. The relationship between the `User` class and the `Reservation` class is characterized by the `User` class having an instance variable of type list for `Reservation`. To create the `Reservation` data type in the microservice that the `User` class belongs to, the DTO pattern is applied. This way the `User` class remains unchanged and does not depend directly on the `Reservation` class that resides in another microservice.

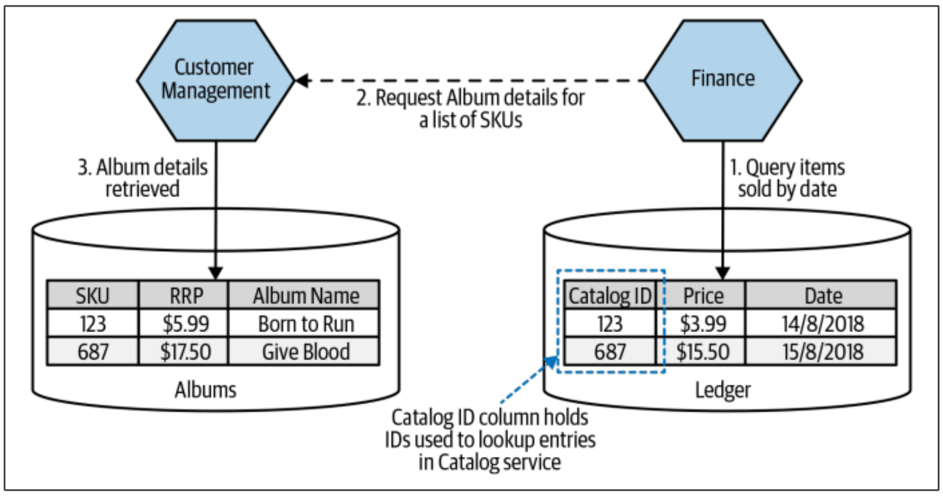
Move Foreign-Key Relationship to Code

The move foreign-key relationship to code pattern is a database decomposition pattern, and like its name indicates, this pattern is used to move the foreign key from the database into the source code of the application.

The relationships between entities are denoted by foreign-key relationships. Defining this relationship in underlying database engine ensures data consistency and lets the database engine execute performance optimizations to ensure that the join operation is as fast as possible. However, when one wants to split the database and the entities involved in the relationship end up living in different schemas two problems emerge: the join of information cannot be performed via database join and the data inconsistency is now possible.

The move foreign-key relationship to code pattern resolves the first problem by moving the join of information to code. By moving the join operation to the code, Figure 8, the database calls are replaced by service calls and the primary key is used to filter the information that is retrieved.

This pattern incurs some performance costs because latency increases. We pass from a local *SELECT* in the database to a *SELECT* in the one database, a service call and a *SELECT* in another database.


Customer Management
Finance
2. Request Album details for a list of SKUs
3. Album details retrieved
1. Query items sold by date
Albums
Ledger
Catalog ID column holds IDs used to lookup entries in Catalog service

SKU	RRP	Album Name
123	$5.99	Born to Run
687	$17.50	Give Blood

Catalog ID	Price	Date
123	$3.99	14/8/2018
687	$15.50	15/8/2018

Figure 8: Replacing a database join operation with service calls

Source: S. Newman, 2019

How do we use it in the refactoring process.

We apply this pattern when we find relationships of the types One-to-One, Many-to-One, and One-to-Many between entities that will belong to different microservices. To do that the first step is to remove the annotation from source code which creates the relationship. This annotation is typically present in one of the classes involved in the relationship. With the removal of the annotation that created the relationship, the table that stored the foreign-key lost the column for that purpose.

Next, we add an instance variable to the class that represents the other entity involved in the relationship. We add this new variable to create a column in the table that is responsible for storing the foreign-key. In this way, the table has the same attributes that it had before the database was split. Furthermore, this new variable will be used as a filter to retrieve data from the database through *SELECT*.

As we already have the tables involved in the relationship with the right attributes and each one in its schema, the next step is to identify the methods that manipulate the data that belongs to the other microservice. A search for methods that have a reference to that data is made. These methods will then have an internal service call to manipulate this data using the primary key as a parameter.

To create the service calls and to make the generated code loosely coupled the following steps are performed:

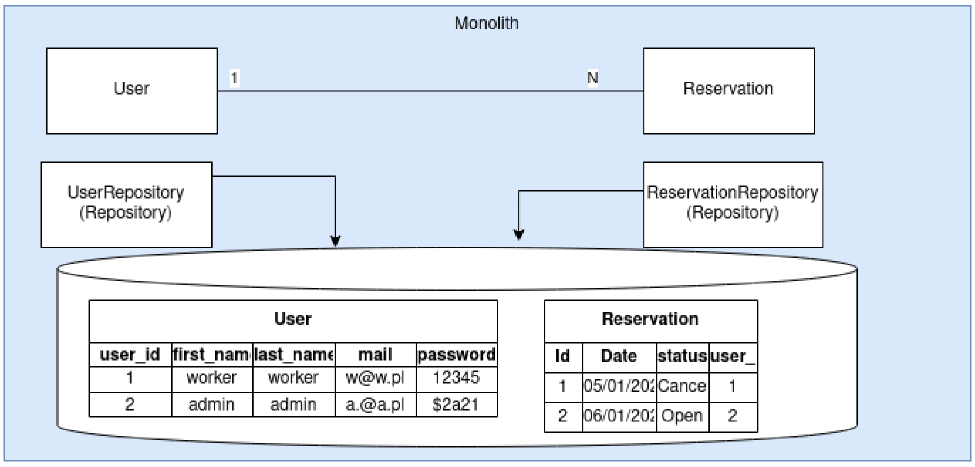
- A interface is created where the signature of the identified methods is declared.
- A class is created that implements the interface created that is responsible for making the service calls
- A variable of the type of the interface created is added to the class that is mapped as an entity.

Finally, to respond to the service calls it is necessary to create a API in the other microservice involved in the relationship. To do that, we create a class where we define the resource paths for the requests, another class for process the request and we add methods to data access object (DAO) for querying the database. These methods added to the class DAO are for retrieving the information required by the services call from the database.

To demonstrate the transformations made by this pattern we use the relationship between the class `User` and the class `Reservation`. Figure 9 shows the One-to-Many relationship between `User` and `Reservation` in the original application. We omit some attributes from `Reservation` because it has no impact on the refactoring of the relationship. Both classes have the `Entity` annotation that indicates that they are mapped as database entities and their instance variables are mapped as attributes. The `Reservation` table has a foreign key that corresponds to the `User's` primary key. Moreover, both classes have their respective `Repository` class that allows the manipulation of the data present in the database.

In terms of code, Figure 10 and 11, the One-to-Many relationship is represented by the `User` class having an instance variable of type list of `Reservation` with `OneToMany` annotation and the `Reservation` class having an instance variable to store a primary key of `User`.

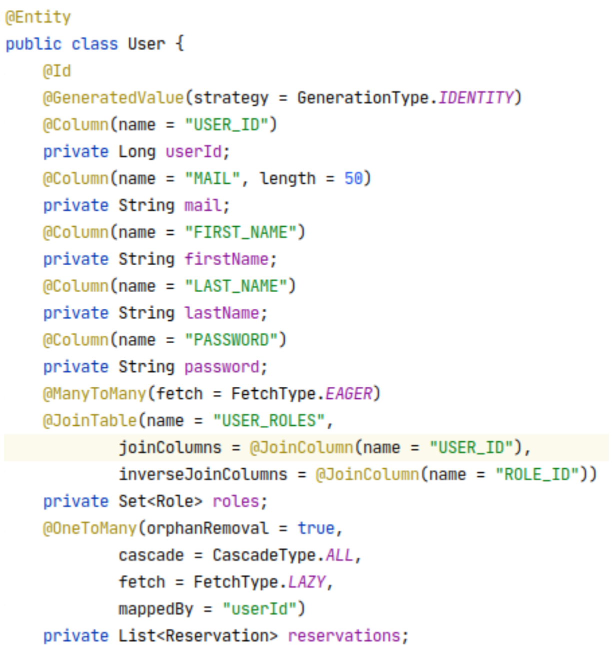
Figure 12 gives an overview of the relationship between the `User` and `Reservation` after refactoring and the transformations made as a result of the refactoring. One interface and one class were generated, `ReservationRequest` and `ReservationRequestImpl` respectively, in the `User's` microservice, which are responsible for making requests to the microservice where the `Reservation` information is and, by applying the DTO pattern, it was also created a class called `Reservation` that has the same attributes as the original `Reservation` class. In microservice #6, where the original `Reservation` class



User				
user_id	first_nam	last_name	mail	password
1	worker	worker	w@w.pl	12345
2	admin	admin	a.@a.pl	$2a21

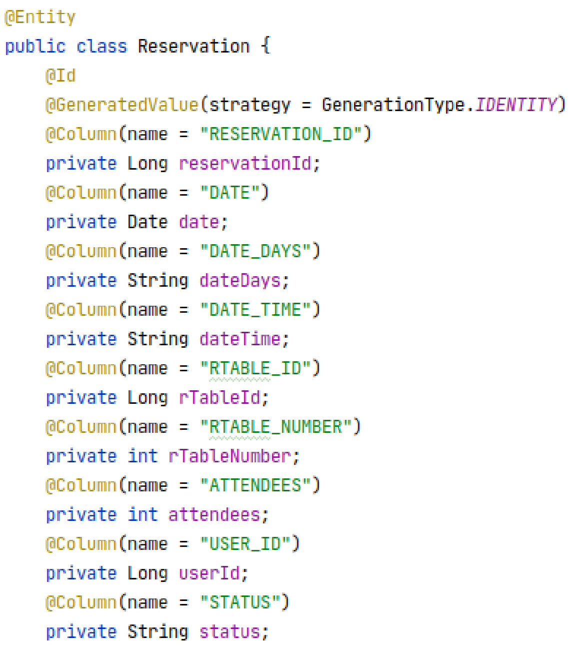
Reservation			
Id	Date	status	user_
1	05/01/202	Cance	1
2	06/01/202	Open	2

Figure 9: One-to-Many relationship between User and Reservation in monolithic application



```java
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "USER_ID")
    private Long userId;
    @Column(name = "MAIL", length = 50)
    private String mail;
    @Column(name = "FIRST_NAME")
    private String firstName;
    @Column(name = "LAST_NAME")
    private String lastName;
    @Column(name = "PASSWORD")
    private String password;
    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name = "USER_ROLES",
            joinColumns = @JoinColumn(name = "USER_ID"),
            inverseJoinColumns = @JoinColumn(name = "ROLE_ID"))
    private Set<Role> roles;
    @OneToMany(orphanRemoval = true,
            cascade = CascadeType.ALL,
            fetch = FetchType.LAZY,
            mappedBy = "userId")
    private List<Reservation> reservations;
```

Figure 10: Instance variables of the User class in monolithic application

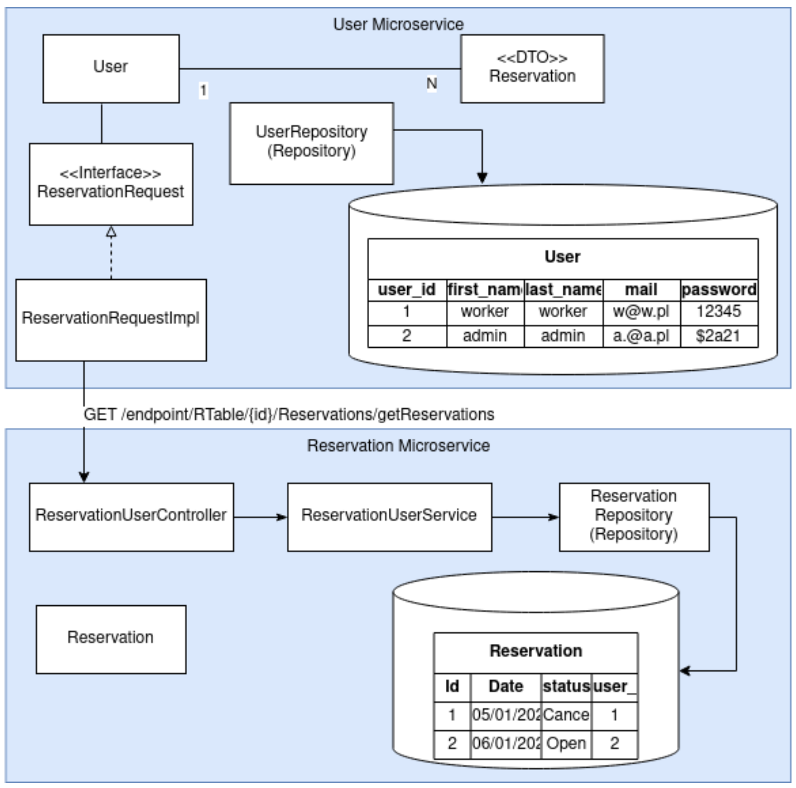


```java
@Entity
public class Reservation {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "RESERVATION_ID")
    private Long reservationId;
    @Column(name = "DATE")
    private Date date;
    @Column(name = "DATE_DAYS")
    private String dateDays;
    @Column(name = "DATE_TIME")
    private String dateTime;
    @Column(name = "RTABLE_ID")
    private Long rTableId;
    @Column(name = "RTABLE_NUMBER")
    private int rTableNumber;
    @Column(name = "ATTENDEES")
    private int attendees;
    @Column(name = "USER_ID")
    private Long userId;
    @Column(name = "STATUS")
    private String status;
```

Figure 11: Instance variables of the Reservation class in monolithic application

belongs, the `ReservationUserController` and `ReservationUserService` classes were created to receive the requests and to process the requests, respectively. Also, in the `ReservationRepository` class the `setReservations` and `getReservations` methods were added.

In terms of code, the class `User`, Figure 13, now has an instance variable of type `ReservationRequest` and the instance variable list of `Reservation` that had the annotation `OnetoMany` now has the annotation `Transient` which indicates that this information is not to be stored in the database. The `ReservationRequest` interface, Figure 14, has the signature of `setReservations` and `getReservations` methods that have as parameter the user id and the `ReservationRequestImpl` class, Figure 15, implements the `ReservationRequest` interface and is responsible for calling the reservation



User

user_id	first_nam	last_name	mail	password
1	worker	worker	w@w.pl	12345
2	admin	admin	a.@a.pl	$2a21

Reservation

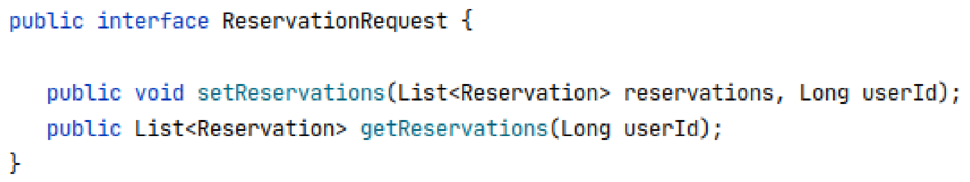
Id	Date	status	user_
1	05/01/202	Cance	1
2	06/01/202	Open	2

Figure 12: One-to-Many relationship between User and Reservation in microservice-based application

microservice.

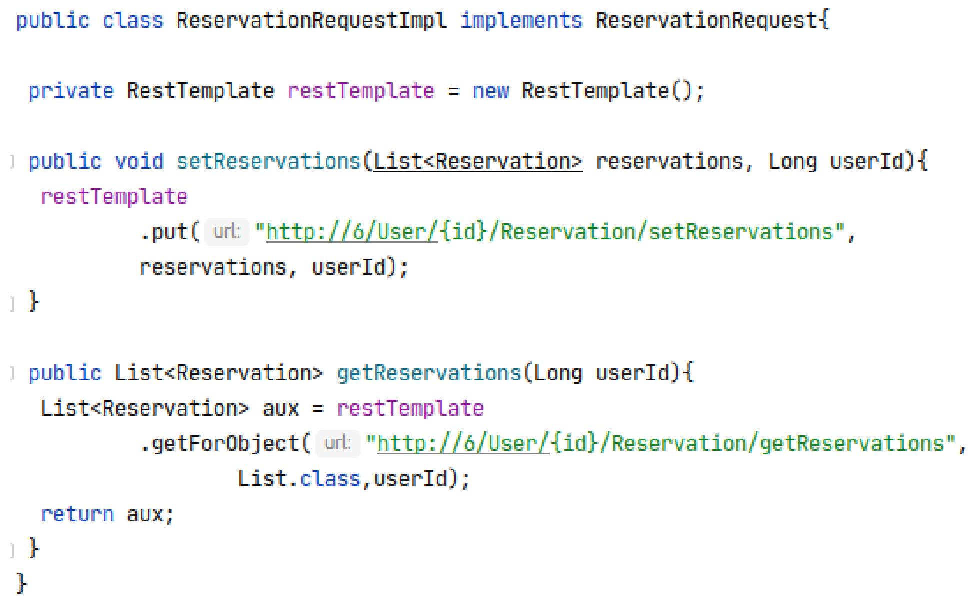
```
@Entity
public class User {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "USER_ID")
 private  Long userId;
@Column(name = "MAIL", length = 50)
 private  String mail;
@Column(name = "FIRST_NAME")
 private  String firstName;
@Column(name = "LAST_NAME")
 private  String lastName;
@Column(name = "PASSWORD")
 private  String password;
@ManyToMany(fetch = FetchType.EAGER)
@JoinTable(name = "USER_ROLES",
        joinColumns = @JoinColumn(name = "USER_ID"),
        inverseJoinColumns = @JoinColumn(name = "ROLE_ID"))
 private  Set<Role> roles;
@Transient
 private  List<Reservation> reservations;
@Transient
 private ReservationRequest reservationrequest = new ReservationRequestImpl();
```

Figure 13: Instance variables of the User class in microservice-based application



```java
public interface ReservationRequest {

   public void setReservations(List<Reservation> reservations, Long userId);
   public List<Reservation> getReservations(Long userId);
}
```

Figure 14: Interface generated by the database refactoring

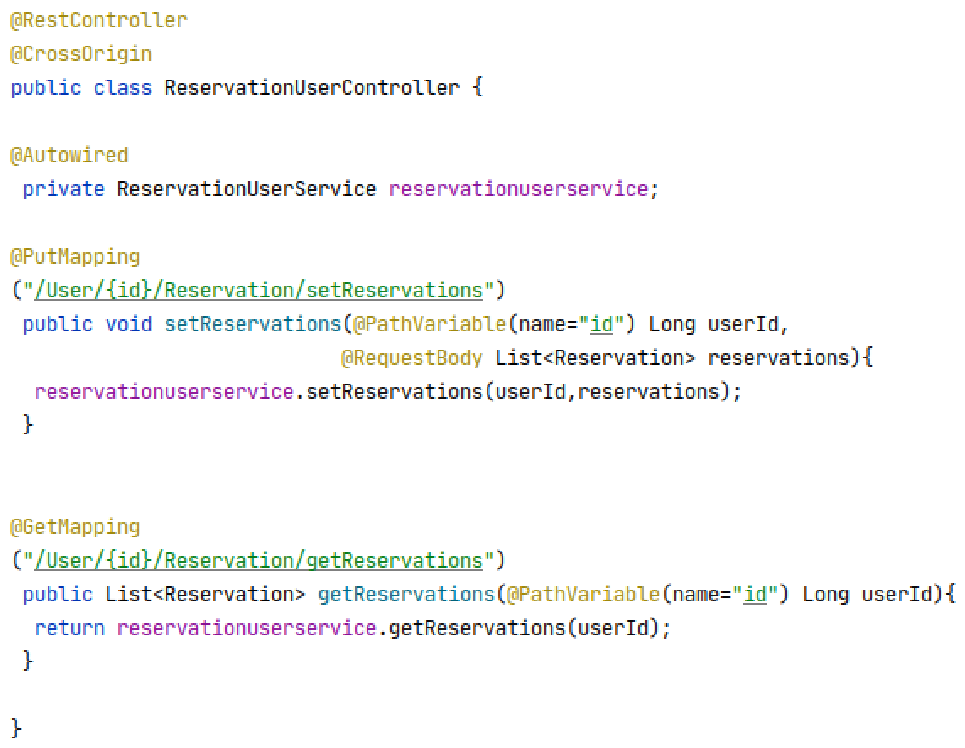


```java
public class ReservationRequestImpl implements ReservationRequest{

 private RestTemplate restTemplate = new RestTemplate();

 public void setReservations(List<Reservation> reservations, Long userId){
  restTemplate
          .put( url: "http://6/User/{id}/Reservation/setReservations",
          reservations, userId);
 }

 public List<Reservation> getReservations(Long userId){
  List<Reservation> aux = restTemplate
          .getForObject( url: "http://6/User/{id}/Reservation/getReservations",
                  List.class,userId);
  return aux;
 }
}
```

Figure 15: Generated class that is responsible for the calls to the Reservation microservice

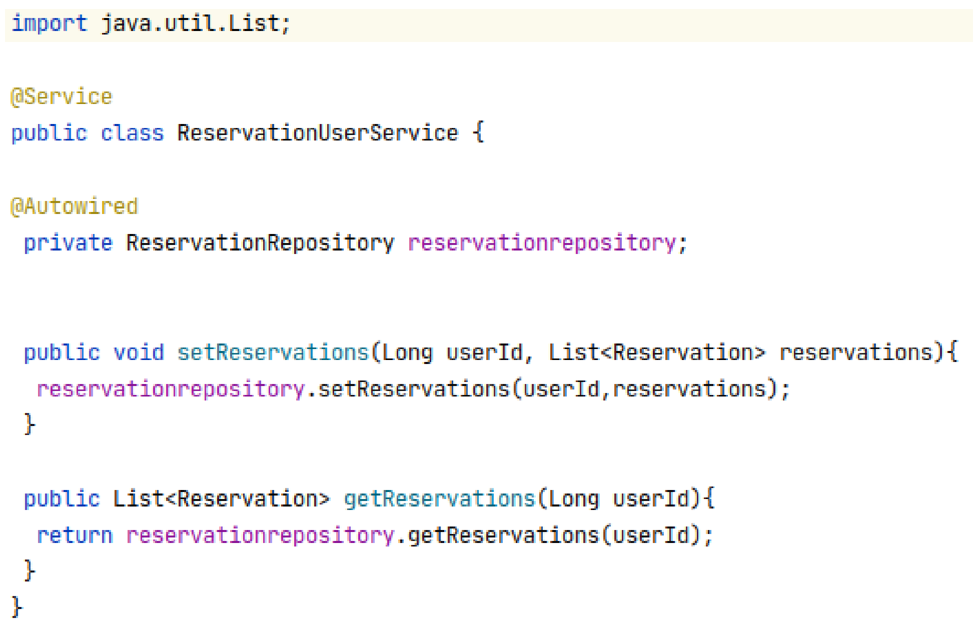
In the `Reservation` microservice the `ReservationUserController` class, Figure 16, is created, where we use the `RestController` and `CrossOrigin` annotations that indicate that this class exposes an API, where we define the resource path for each method and where we annotate the methods with `PutMapping` or `GetMapping` according to the type of operation they represent.

Finally, the class `ReservationUserService`, Figure 17, is created where the request is directed to the `Repository` class where the `getReservations` and `setReservations` methods were declared using the `User` id as a filter.



```
@RestController
@CrossOrigin
public class ReservationUserController {

@Autowired
 private ReservationUserService reservationuserservice;

@PutMapping
("/User/{id}/Reservation/setReservations")
 public void setReservations(@PathVariable(name="id") Long userId,
                             @RequestBody List<Reservation> reservations){
  reservationuserservice.setReservations(userId,reservations);
 }


@GetMapping
("/User/{id}/Reservation/getReservations")
 public List<Reservation> getReservations(@PathVariable(name="id") Long userId){
  return reservationuserservice.getReservations(userId);
 }

}
```

Figure 16: Generated class that exposes the API to handle Reservations



```
import java.util.List;

@Service
public class ReservationUserService {

@Autowired
 private ReservationRepository reservationrepository;


 public void setReservations(Long userId, List<Reservation> reservations){
  reservationrepository.setReservations(userId,reservations);
 }

 public List<Reservation> getReservations(Long userId){
  return reservationrepository.getReservations(userId);
 }
}
```

Figure 17: Generated class that processes and directs the request to the repository class

The transformations made to the relationship between `User` and `Reservation` are the same for the relationships `BillPosition` - `Dish`, `Rtable` - `Reservation` and `Rtable` - `Bill`. The `BillPosition` - `Bill` relationship did not need refactoring as the two classes belong to the same microservice. In Appendix A.2 are all the transformations made by database refactoring.

Database Wrapping Service

The database wrapping service pattern is a coping pattern, used to add a service as wrapper of database schema.

As Newman, 2019 says, when something is too complicated to handle, hiding the mess makes sense. Sometimes breaking down the database of sensitive systems like banks and hospitals can lead to catastrophic consequences. One possible solution is to share the database across all microservices but it goes against the principles of microservices - each microservice has its own data and loose coupling. The database wrapping service pattern appears as a better solution and it is also seen as a stepping stone to more fundamental changes, giving you time to break apart the schema underneath your API layer (S. Newman, 2019). This pattern creates a new service to "hide" the database, Figure 18, providing an API to access the data. In this way the services replace access to the database with requests to the service that owns the database. This ensures the database schema is unchanged and the services exhibit low coupling.

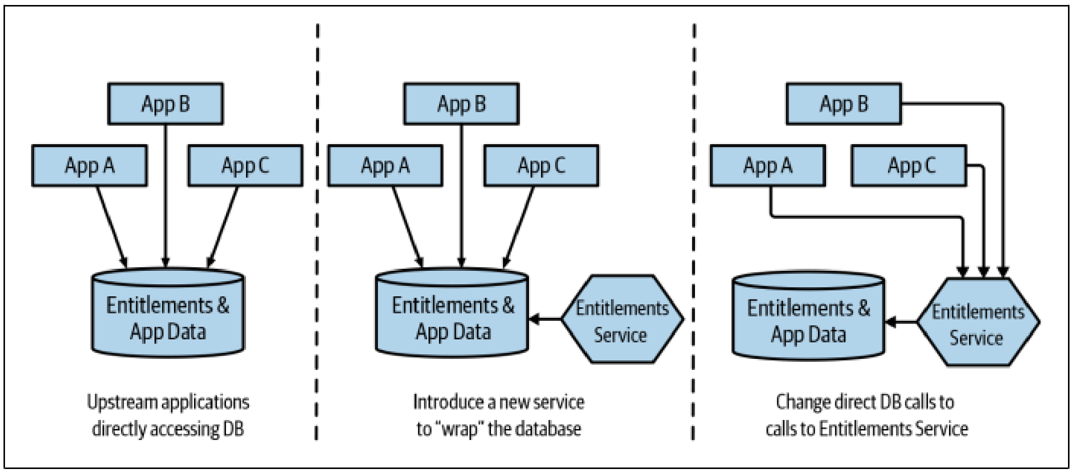


Figure 18: Using a service to wrap a database

Source: S. Newman, 2019

How do we use it in the refactoring process.

We apply this pattern when we find a Many-to-Many relationship between entities that will belong to different microservices. In Many-to-Many relationships a "join table" is created, being formed by the two foreign keys (i.e. copies of the primary keys of the entities involved). If we apply the move foreign-key relationship to code pattern we would lose this table. Instead we apply this pattern and the relationship keeps intact.

We apply this pattern by extracting the classes that are mapped to the entities that are involved in the relationship, and their respective DAO classes for a new service. This new service provides an API to access the data stored in the database and, therefore, the services that need to access that data, replace direct database calls with calls to the new service.

To demonstrate how this pattern is applied we use the relationship between `User` and `Role`. By the microservices proposal of table 1 we verify that the classes `Role` and `User` belong to different microservices and by table 3 we verify that these classes have a Many-to-Many relationship. We create an extra microservice to store the `User` and `Role` classes and their respective `Repository` classes to access the database. This way the relationship between the entities is maintained and the refactoring process continues with this new microservice being added to the microservices proposal and the classes dependency lists are recalculated taking into account this new scenario.

4.4 Code Refactoring

Having already refactored the classes that create the database logical schema, it is now necessary to analyze the other classes. The code refactoring phase aims to refactor the classes that have method calls to other classes that are part of other services. Each of these methods calls is replaced by a service call that is responsible for call the original method in the microservice where the method belongs. Figure 19 illustrates an overview of the steps that compose the code refactoring phase.

The code refactoring process starts with the analysis of the variables of the classes. The information about the variables is straightforwardly obtained from the structural information of the class. We analyze the data type of each instance variable and we check if the data type is in the list of dependencies that the class has with other microservices. If the data type of the instance variable under analysis is not in the list of dependencies, it is not necessary to continue the refactoring process for this variable, because it corresponds to a data type that exists in the microservice to which the class belongs or it is a primitive data type of the language. However, if the data type is among the dependencies with other microservices it is necessary to search for method invocations of the class that corresponds to the data type, because these methods, with the decomposition of the monolith in microservices, will no longer be invoked locally, they will be replaced by calls to the service that owns the data type and consequently the methods.

The identification of the invoked methods is also simple to obtain through structural information. Since the identified methods may have as a parameter or return a data type that is present in the class's list of dependencies with other microservices, it is necessary to check the data type of the parameters and the return. If in the parameters or in the return of the method we found data types that are in the list of dependencies of the class with other microservices we apply the DTO pattern to create these data types in the microservice. The reason for doing this is that the parameters and the method return will be sent in the service call so they are data transfer objects.

After having the invoked methods identified and their parameters and returns checked, we can create the service calls. We create the data type of the instance variable by creating an interface with the same name as the data type. In this way the modifications made to the code will be transparent to the class under analysis. The interface created contains the signature of all the invoked methods identified that will become service calls. To define the service calls we create a class, termed `Request`, that implements the interface created. This new class is responsible for making the remote service calls and therefore we define the REST calls for each method in this class.

Finally, in the microservice that owns the class that corresponds to the data type that triggered the whole refactoring process, a REST API is created that allows the invocation of the original methods.

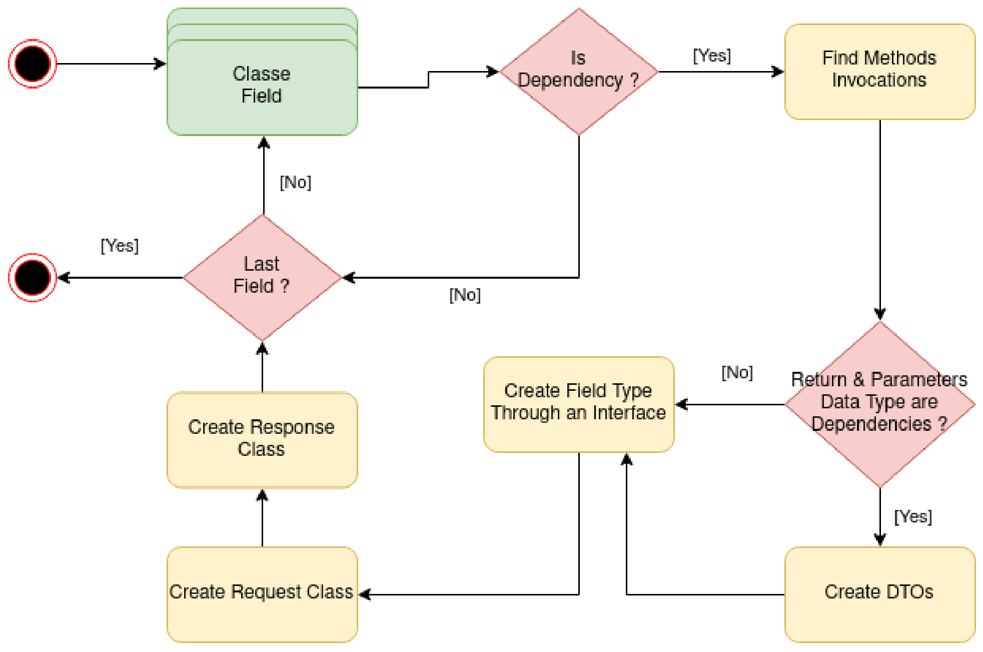


Figure 19: Overview of code refactoring

In order to complete the code refactoring process it is necessary to check the data types of the local variables, which are the variables created within each method. We check if the data type of the local variables are in the list of dependencies with other microservices of the class under analysis. If the data type of the local variables is in the list of dependencies we create the data type by applying the DTO pattern and a search is made for invoked methods of the class that corresponds to the data type. We apply the DTO pattern because these variables typically, when they are not of a language primitive type, are the result of invoking a method of a class that has defined an instance variable. In fact, the refactoring of the instance variables already applies the DTO pattern to create the return data type of the invoked methods which corresponds, for the most part, to the local variables. To avoid creating classes that already exist when a data type is created by applying the DTO pattern, this data type is removed from the list of dependencies with other microservices of the class under analysis. The identified invoked methods of the local variables are defined as service calls in the class generated by the DTO pattern application. As in the analysis of the instance variables a REST API is created, in the microservice that owns the data type, that allows the invocation of the original method.

To show the changes made by the code refactoring phase we use the dependency between the `BillService` class and the `RtableRepository` class, as an example. The Figure 20 gives us an overview of the interaction of the class `BillService` with the class `RTableRepository` in the monolithic application. The `BillService` class has two instance variables, one of type `BillRepository` and another of type `RTableRepository`, and six methods: `getBills`, `getBillById`, `addBill`, `updateBill`, `deleteBill` and `getBillPositions`.

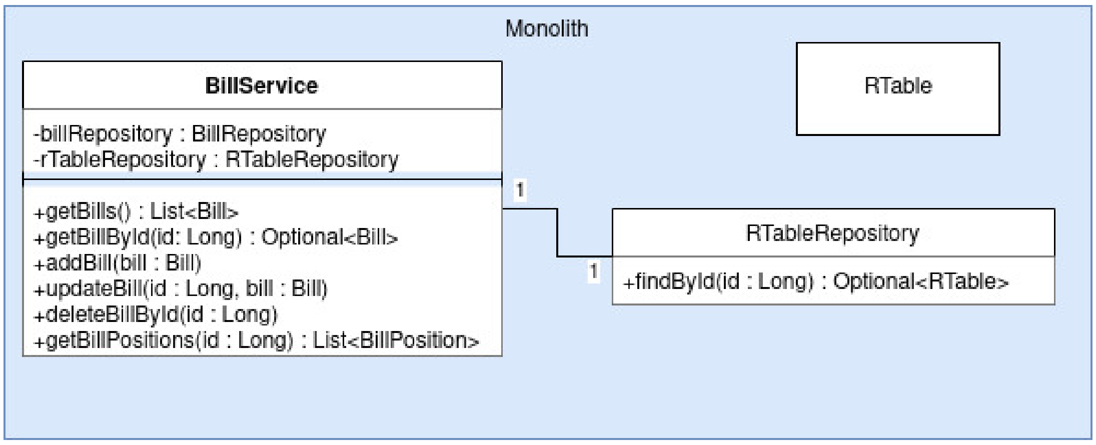

Monolith
BillService
-billRepository : BillRepository
-rTableRepository : RTableRepository
+getBills() : List<Bill>
+getBillById(id: Long) : Optional<Bill>
+addBill(bill : Bill)
+updateBill(id : Long, bill : Bill)
+deleteBillById(id : Long)
+getBillPositions(id : Long) : List<BillPosition>
1
1
RTable
RTableRepository
+findById(id : Long) : Optional<RTable>

Figure 20: Interaction between Billservice and RTableRepository in monolithic application

In the monolithic version the relationship between the `BillService` class and the `RTableRepository` class is characterized by the `BillService` class having an instance variable of type `RTableRepository` and by invoking the `findbyId` method, which returns a `RTable`, from the `RtableRepository` class within the `addBill` method.

The relationship between the two classes and the invocation of the `findById` method after code refactoring phase is presented in Figure 21. In the code refactoring phase an interface was created in `BillService's` microservice called `RTableRepository` to represent this type in the microservice where the `findById` method signature was declared since it is only this method that is invoked from the original `RTableRepository` class. Besides that, it was created a class called `RtableRepositoryImpl` that is responsible for making the REST call to invoke the original method and a class called `RTable` to represent the `finById` return type. In `RTable's` microservice a class called `RtableRepositoryController` is created where the REST API is defined and that invokes the `findById` method in the original `RTableRepository` class.

All the transformations made by the code refactoring phase can be seen in https://github.com/MicroRefact/restaurantServerMs where we provide the source code of the refactored application.

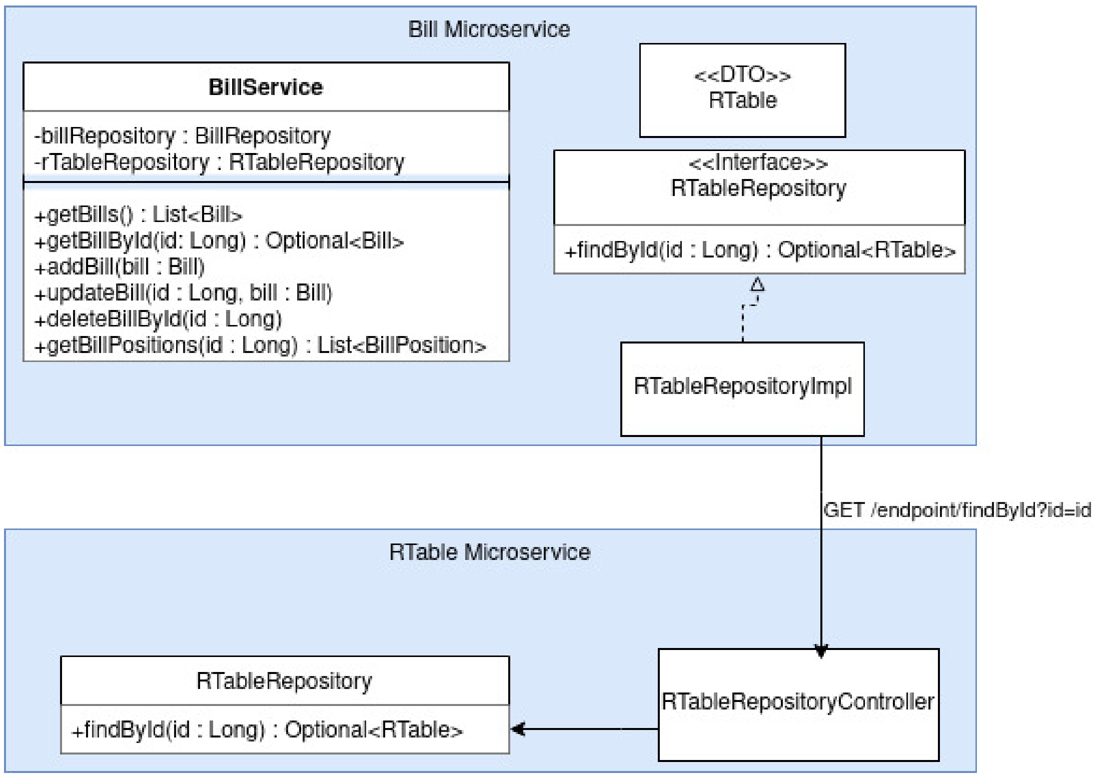

Bill Microservice
BillService
-billRepository : BillRepository
-rTableRepository : RTableRepository
+getBills() : List<Bill>
+getBillById(id: Long) : Optional<Bill>
+addBill(bill : Bill)
+updateBill(id : Long, bill : Bill)
+deleteBillById(id : Long)
+getBillPositions(id : Long) : List<BillPosition>
<<DTO>>
RTable
<<Interface>>
RTableRepository
+findById(id : Long) : Optional<RTable>
RTableRepositoryImpl
GET /endpoint/findById?id=id
RTable Microservice
RTableRepository
+findById(id : Long) : Optional<RTable>
RTableRepositoryController

Figure 21: Interaction between Billservice and RTableRepository in microserices-based application

5

MicroRefact - A tool for refactoring Java monoliths

In this chapter we present the implementation of our tool, MicroRefact, which is a shell based-tool developed in *Python* that serves as a proof of concept to validate the methodology. The MicroRefact is designed for Java applications, in particular for the Spring[1] framework. Although the methodology indicates the manipulation of data access objects the developed tool only supports applications that use the repository pattern[2] instead of the DAO pattern since the identification of these classes is straightforward through the annotations and the identification of which entity the repository class refers to is directly obtained. Furthermore, both patterns have the same goal: to serve as an abstraction to persist the data so the methodology covers both patterns.

Regarding the ORM, there are several ORM frameworks. MicroRefact supports the refactoring of applications that use the Java Persistence API[3] (JPA) to do the mapping. We decided cover JPA since it describes a common interface to data persistence frameworks. The full implementation is available at https://github.com/FranciscoFreitas45/MicroRefact. Figure 22 represents an overall flow of MicroRefact.

In the following sections we detail each part of MicroRefact.

5.1 Information Extraction

The information extraction phase is responsible for processing the input provided by the user in order to identify the proposed microservices and extract information from the source code of the monolith. MicroRefact receives as input a *JSON* file with the composition of the microservices and the path to the source code of the monolith. The microservices are represented in the *JSON* file by documents in which each document contains an array with the full name of the classes that form the microservice.

[1] https://spring.io/
[2] https://martinfowler.com/eaaCatalog/repository.html
[3] https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html

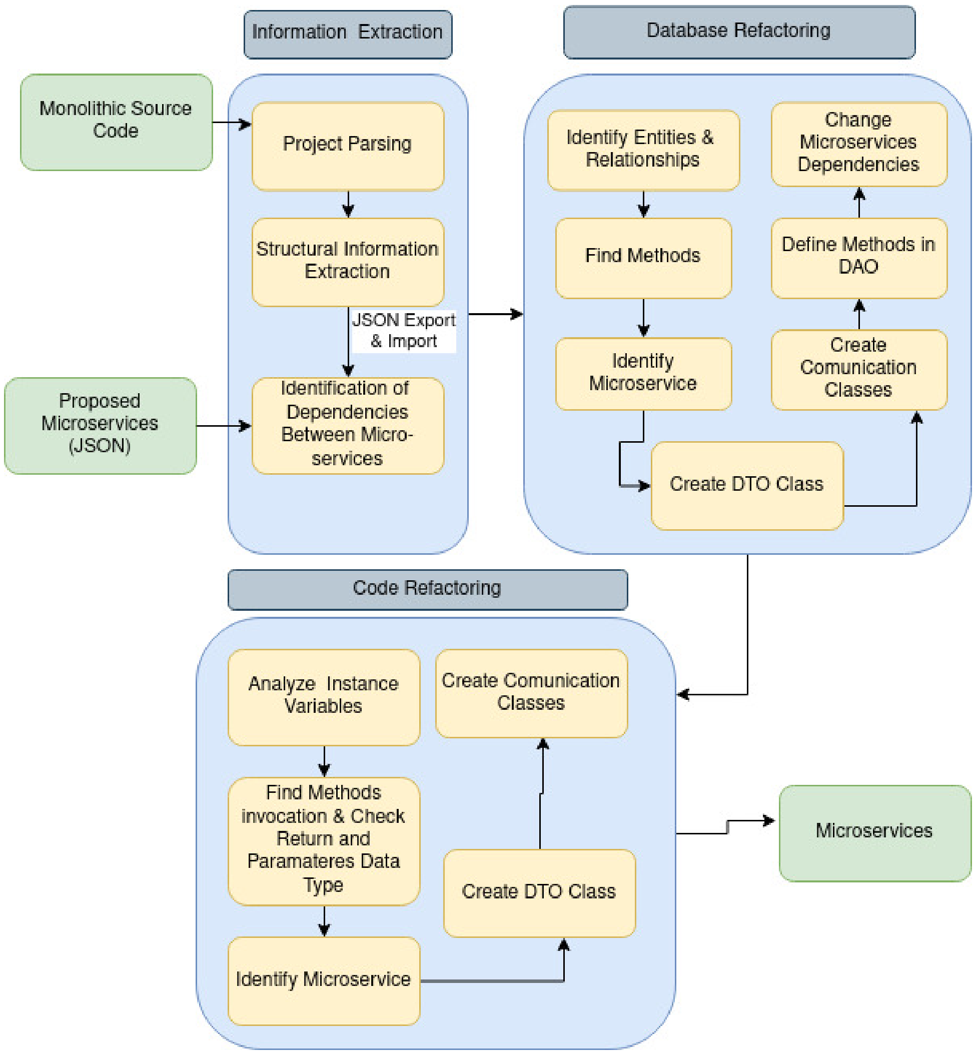

Information Extraction
Database Refactoring
Monolithic Source Code
Project Parsing
Structural Information Extraction
JSON Export & Import
Proposed Microservices (JSON)
Identification of Dependencies Between Micro-services
Identify Entities & Relationships
Find Methods
Identify Microservice
Create DTO Class
Create Comunication Classes
Define Methods in DAO
Change Microservices Dependencies
Code Refactoring
Analyze Instance Variables
Find Methods invocation & Check Return and Paramateres Data Type
Identify Microservice
Create DTO Class
Create Comunication Classes
Microservices

Figure 22: Overall flow of MicroRefact

5.1.1 Project Parsing

In order to extract structural information from the source code of the software project under analysis, we parsed the source code of the software project to an AST. For parsing we decided to use the *Java Parser*[4] library since it is the most used parser for the Java language, having weekly releases.

The extraction of the structural information is performed through the following steps:

- Firstly, the *Java Parser* identifies the source root of the software project under analysis and from it, for each java file, it creates a *CompilationUnit*. It is through the *CompilationUnits* that we will obtain the structural information of each class.

- For each of the *CompileUnits* a class is identified, as well as its full qualified name, its range, its declared constructors, its imports and its direct dependencies of inheritance and implementation.

[4]https://javaparser.org/

- Next, a set of visitors are declared to extract the following information from the class: method declarations (*MethodDeclarationVisitor*), annotations (*AnnotationVisitor*), variable declarations (*VariableDeclaratorVisitor*), mehtods calls (*MethodCallExprVisitor*) and instance variable declarations (*FieldDeclarationVisitor*). The *MethodDeclarationVisitor* identifies all the methods declared in a class. For each method it is identified its name, annotations, return type, the type of its parameters, the local variables, its body and the methods called within it. *AnnotationVisitor* just identifies annotations attached to a class. *VariableDeclaratorVisitor* identifies all variables both instance and local. *MethodCallExprVisitor* is responsible to identify all the method calls from other classes. *FieldDeclarationVisitor* is used to identify all the instance variables. For each instance variable its name, annotations, type and modifier are identified. To create the dependency list for a class, we analyze all the declared variables and methods. We identify the type of variables and the types of parameters and method returns. Then, we generate a list with the names of the classes that a given class depends on.

- An object is created to store the information identified and collected by the *Java Parser* for each class. At the end of the identification and extraction process, the information about the classes of the project under analysis is exported to a *JSON* file to be processed by the *Python* program, along with a *JSON* file which contains the composition of the microservices.

5.1.2 Identification of Dependencies between Microservices

To identify the dependencies between classes that belong to different microservices we need to process both JSON files, so these files are read by the *Python* program. A data structure is created to represent the structural information of each class as well as to represent the microservices in the program.

For the data structure, we create a class called *Class* to represent a class and a class called *Cluster* to represent a microservice. The class *Class* is responsible for storing the structural information of each class. The *Cluster* class is composed by:

- A dictionary, where the key is the name of a class and the value is an object of type *Class*.

- A list for adding new classes to the microservice.

- The path to the folder where the microservice's Java files are created.

The data structure created is composed of a list of *Cluster* that represents the number of microservices into which the monolith has been decomposed and the compositions of the microservices.

The process of loading the structure begins by reading the *JSON* file that contains the composition of the microservices. For each microservice a *Cluster* object is instantiated and for each class a *Class* object is instantiated that is stored in the dictionary inside the *Cluster* object. The *Class* objects at this stage do not yet have the structural information loaded. Next, to load the *Class* objects with the structural information, the *JSON* file containing the structural information of all classes of the project under analysis is read and, through setters the structure is loaded. While loading the structure it is checked if the proposed decomposition of the monolith respects one of our assumptions i.e. if the super classes and the sub classes belong to the same microservices. If it does not, the program throws an exception notifying the user that there is a bad division of the monolith. Also, while loading the structure, some adaptations are made to the microservices proposal, regarding the implementation of interfaces. It is checked if all the interfaces that a given class implements are in the same microservice as the class. To achieve this, an iteration over the list of interfaces that a given class implements is made and it is checked if the name of the interfaces are key in the dictionary that represents the microservice that the class in analysis is in. If they are not key, a copy of the *Class* object that represents the interface is made and added to the dictionary that represents the microservice of the class under analysis. This assures that all the interfaces that a class implements exist in the microservice.

After having the structure loaded and the microservices proposal validated, the dependencies between the classes of different microservices are identified. This is accomplished by iterating over the dictionary, contained in the *Cluster*, that contains the *Class* objects that represent the classes of the microservice, and for each *Class* it is checked if the name of the classes that are in the list of dependencies of the *Class* are keys in the dictionary that the *Class* under analysis belongs to. If it is not key the name of the class that the *Class* depends on, is added to the list of dependencies with other microservices of the *Class* through a setter method. This process is repeated for all objects *Cluster*, so all objects *Class* have a list of dependencies with other microservices fulfilled.

5.2 Database Refactoring

With the data structure loaded, it is used to refactor the database of the software system under analysis. This process begins with the identification of the classes that are mapped as entities by analyzing the annotation list of each *Class* object. A search is made for the keyword `Entity` in the annotation list. For each *Class* that is identified as an entity, the algorithm retrieves the list of instance variables to identify relationships between entities. The relationships are denoted by keywords `OneToMany`, `ManyToOne`, `ManyToMany` and `OneToOne`. An iteration through the list of instance variables is performed to check whether the instance variables have in their annotation list one of the keywords that indicate a relationship.

If relationships are found, it is checked if the type of the instance variable, which has the annotation, is in the list of dependencies with other microservices of the object *Class* under analysis. If the type of the instance variable is not in the list of dependencies with other microservices of the *Class*, it means that the classes involved in the relationship are in the same microservice and move on to analyze the next instance variable. However, if it is, the refactoring follows different paths according to the relationship identified. For the `OneToMany`, `OneToOne` and `ManyToOne` relationships a search is performed on the list of methods of the *Class* object under review to identify methods that use the instance variable. We identify methods that use the type of the instance variable through the return type, or through parameters, or in the declaration of variables. With the methods identified a list, named *methods*, is created and this list will be used for the REST API definition.

For the application of the DTO pattern to create the type of the instance variable in the *Cluster* where the *Class* under analysis belongs, the type of the instance variable is used to identify the position (index) in the *Cluster* list, of the *Cluster* object that has in its dictionary a key with the same name as the type of the variable. Then, the index and the type of the variable are used to do a get of the corresponding *Class* object to make a clone of it, named *ClassDepend*. In this cloned object some changes are made in its attributes. The prefix "DTO" is added to the qualified name of the *ClassDepend*, the annotation list associated with the *ClassDepend* is cleared and an iteration through the instance variables of the *ClassDepend* object is performed to identify possible types that do not exist in the *Cluster* that the *Class* under analysis is in. The identified types are marked as "dragged" by the *ClassDepend* and the objects associated with these types are cloned and stored together with the *ClassDepend* in the list of new classes of the *Cluster* object to which *Class* belongs.

To apply the move foreign-key relationship to code pattern, two functions are created that are responsible for creating the interface and the class that implements the interface, *createInterface* and *createClass_callInterface*, respectively. A class called *MyInterface* is also created to represent the interfaces created and its attributes are the name, the method modifier, the signature of the methods and the imports.

The *createInterface* function receives as one of its parameters the list of identified methods, *methods*. The *createInterface* function extracts the signature from the identified methods and creates a *MyInterface* object. The name given to the generated interfaces is composed of the type of the instance variable plus the word Request (e.g. ClassXREQUEST). The function *createClass_callInterface* receives as one of the inputs the generated *MyInterface* object and its goal is to create a class where the calls to the REST API that will be created later are defined.

To make REST calls it is used the `RestTemplate` class from the *org.springframework.web.client* package and therefore a variable of this type is added to the list of instance variables of the class to be created. Next, the get of the signature of the methods present in the *MyInterface* is obtained and the methods that are responsible for making REST calls are created. For the REST calls it is used two methods from the `RestTemplate`, *getForObject* for methods that have a return type and *put* for methods where the return type is void. We chose to use the *put* method instead of *postForObject* since most of the methods are setters, i.e. there is no creation of new objects but rather an update of certain attributes.

After all the methods are declared an object of type *Class* is instantiated that contains the information of this new class and is added to the list of new classes of the *Cluster* to which the class in analysis belongs as well as the interface.

For the application of the move foreign-key relationship to code pattern to be complete, a REST API is created. The list of identified methods (*methods*) is used to create an object of type Class, called *Controller*, where the resource paths are defined. The *Controller* object's annotation list is formed by the annotations

`RestController` and `CrossOrigin` because these are the annotations that create the REST API and the imports list by `org.springframework.web.bind.annotation.*`, `org.springframework.beans.factory.annotation.Autowire` and the import of the data type of the instance variable under analysis. The list of instance variables is composed of a variable of type *Service* that will be created next. For each method present in the list *methods* a method with the same name is created and a resource path is defined as follows:

" */type_of_the_instance_variable/{foreign-key}/name_of_Class_under_analysis/Name_of_method* ".

Each resource path has associated an Hypertext Transfer Protocol (HTTP) verb. Methods that have return type have the `GetMapping` annotation because they are gets and those that do not have return type have `PutMapping` annotation because they are puts.

Then another *Class* object is created, called *Service*, where the requests are processed. The *Service* also uses the *methods* list to define the methods. The *Service* object's annotation list is formed by the annotation `Service` because this new class is responsible for providing the business functionalities. This class aims to invoke the functions declared in the Repository class to query the database. The *Service* and *Controller* objects are added to the list of new classes of Cluster to which the type of the instance variable under analysis belongs.

Finally methods are added to the Repository class. For this the identification of the *Class* object that represents Repository is carried out. For the identification we used regular expressions through the *re.py*[5]. After the identification, the signature of the methods from the *methods* list is added with the addition of a parameter to represent the foreign-key. These methods require the user, after refactoring the whole application, to define the queries to the database.

For the `ManyToMany` relationship the refactoring is different. For the application of the database wrapping service pattern, a search is made for the object *Class* that represents the Repository of the *Class* under analysis. To identify this object *Class* we also use regular expressions. Next, the type of the instance variable is used to identify the position (index) in the Cluster list, of the Cluster object that has in its dictionary a key with the same name as the type of the variable. Then the index and variable type are used to get the corresponding *Class* object and to identify the Repository corresponding to the object.

Next, a *Cluster* object is created in which the two objects that represent the entities and the two objects that represent the respective Repositories are stored in its dictionary. When these objects are stored in the new *Cluster* object, they are deleted from the dictionary of the *Cluster* object that they belonged to before. Besides, it is also checked if the *Class* objects have classes marked as "dragged" and if so, a clone of these objects is made and stored in the dictionary as well.

Then an iteration is made through the dictionaries of the *Clusters* where the objects belong to check if the objects have interactions with other classes. If interactions are identified, the name of the object it depends on is added to the list of dependencies with other services of the *Class* object that had the interaction. Finally the *Cluster* object is added to the list of *Cluster*.

[5] https://docs.python.org/3/library/re.html

5.3 Code Refactoring

After the refactoring of the classes that define the database logical schema is finished, it is time to analyze the other classes. For this purpose, two functions were created, *code_refactoring_instance_var* which is responsible for analyzing the instance variables of each object *Class* and the *code_refactoring_local_var* which is responsible for analyzing the local variables.

The *code_refactoring_instance_var* function starts with an iteration through the dictionary of each *Cluster* to refactor the classes that do not form the database logical schema. These classes are identified through the annotation list, i.e., if the annotation list does not contain the `Entity` annotation then it is a class that will be analyzed by the function. For each analyzed *Class*, it is performed the get of instance variables and for each instance variable it is checked if its type belongs to the list of dependencies with other microservices of the *Class* under analysis. If the type of the instance variable does not belong to the list of dependencies with other microservices, it means that the type of the variable exists in the microservice. Otherwise, a search is executed in the list of invoked methods of the *Class* object under analysis, for the methods invocation by the instance variable, and a list with the name of the identified methods is created, called *methodsInvocations*. For each method present in *methodsInvocations* it is checked if its return type or parameter types are among the dependencies with other microservices of the *Class* object under analysis and if so the DTO pattern is applied. For this purpose, each parameter type and return type is used to identify the position (index) in the *Cluster* list of the *Cluster* object that has in its dictionary a key with the same name as the type. Then the index and type are used to get the corresponding *Class* object to clone it.

To avoid adding a class to a microservice where it already exists, it is checked if the name of the cloned object *Class* exists in the list of new classes of the *Cluster* to which the *Class* object in analysis belongs. If it already exists the cloned object is destroyed, otherwise, the object's annotation list is cleaned and it is checked if the cloned object's instance variables have dependencies. If they have dependencies, the *Class* objects representing their types are cloned and are marked as “dragged” by the *Class* object representing the DTO, and both the cloned objects marked as “dragged' and the cloned object representing the DTO are stored in the list of new classes of the *Cluster* to which the *Class* object under analysis belongs.

Next, an interface is created to represent the type of the instance variable that triggered the refactoring. To achieve this, and to avoid replications, it is verified if already exists a *MyInterface* object with the name of the type of the instance variable in the list of new classes of the Cluster which the object *Class* under analysis belongs. If it already exists it means that already exists an interface that represents the type of the instance variable and a class that implements the interface, and therefore only the methods that are not yet declared in the interface are added. To do this, an iteration is performed over the identified invoked methods and it is checked if they are already in the list of methods declared in the interface and if not, their signature is added to the interface and they are added to the *Class* object that represents the class which implements the interface. If it does not exist, a *MyInterface* object is instantiated with the name of the instance variable where the identified invoked methods are added to the method list and a *Class* object is instantiated that represents the class that implements the interface and will make the remote invocations of the original methods. As in database refactoring, we also use the `RestTemplate` class for HTTP requests. The *MyInterface* object and the *Class* object created are added to the list of new classes of the *Cluster* to which the object *Class* in analysis belongs.

Finally, a REST API is created that allows the invocation of the original method. The process is similar to database refactoring. It is verified if it already exists an object *Class* with the name of the *Class* object that intends to create and if so, it is performed the get of this object and added to its method list the methods present in the object *MyInterface*, generated in the previous step, that the *Class* does not have. If a *Class* object with the same name does not exist, a *Class* object with that name is instantiated and added to the list of new classes of the *Cluster* to which the orginal method belongs and all the methods present in the *MyInterface* object are declared in this object.

In the *code_refactoring_local_var* function, like in *code_refactoring_instance_var*, an iteration over each *Cluster* is executed and the *Class* objects that do not have the `Entity` annotation are analyzed, but this function analyzes the local variables. For each class an iteration over the methods is performed to check, for each method, if there are methods invoked from a class that does not belong to the *Cluster*. If the invocation of methods from classes that do not belong to the *Cluster* is identified, it is verified if the variable that makes the invocation is an instance variable and if it is the case it is not necessary to refactor because it was done in the previous function. However, if it is not the case, the DTO pattern (Fowler et al., 2002) is applied through the clone of the *Class* object that represents the type of the variable and their "dragged" classes.

Finally, the information contained in each object *Class* is used to write Java files. The *Class* class has a method called *create*, which generates a Java file with the class information. To do this, *Python's* built-in *open* function is used to create a file object and an iteration over all the fields in the *Class* is performed to write them to the file using the *write* method of file object. A folder is created for each microservice where the files are written.

6

Evaluation

In order to quantitatively assess the applicability of our methodology and qualitatively assess whether the generated microservices-based applications are functionally equal to the original version we collected 120 projects from *GitHub*. In the qualitative evaluation we use the unit tests of the monolithic projects to execute them in both versions of the application, the original and the microservices based one, and if the result of the unit tests is the same in both applications we conclude that the applications are functionally equal. In this chapter we describe this evaluation in detail.

6.1 Project collection

To evaluate the proposed methodology it is necessary to test it in several projects of different sizes and complexity. To do this, we created a list of projects for which we executed the MicroRefact. We decided to extract the projects from *GitHub* because it is the most popular platform for hosting source code and files.

We used the *GitHub* search API for code to find repositories that contained the terms `org.springframework.data.jpa` and `org.springframework.data.jpa.repository.JpaRepository`, since these are very common terms in applications that use JPA annotations to do object relational mapping and are terms exclusive to applications built with the Spring framework. The query used was as follows:

https://api.github.com/search/code?q=org.springframework.data.jpa+org.springframework.data.jpa.repository.JpaRepository+language:java

Since the *GitHub* search API limits each request to 1000 results and the query is to identify repositories through code there are repeated results. Executing this query and after removing duplicate repositories, we identified 686 repositories. To ensure that only monolithic applications are used, we only consider projects with one 'src' folder. We also discard projects with less than 25 classes because they may represent "toy" projects and we use filters to exclude demo and test projects using the following stop words: "release", "framework", "learn", "source","spring", "study", "demonstration", "test", "practice". That reduced the 686 projects to 353. We decided to use about one third of the projects since we feel that one third is representative of the population and also because of time reasons. From the identified projects we randomly selected 120 projects. The histogram of the projects collected by the number of classes is presented in Figure 23. The biggest one has 1339 classes and the smallest 27 classes

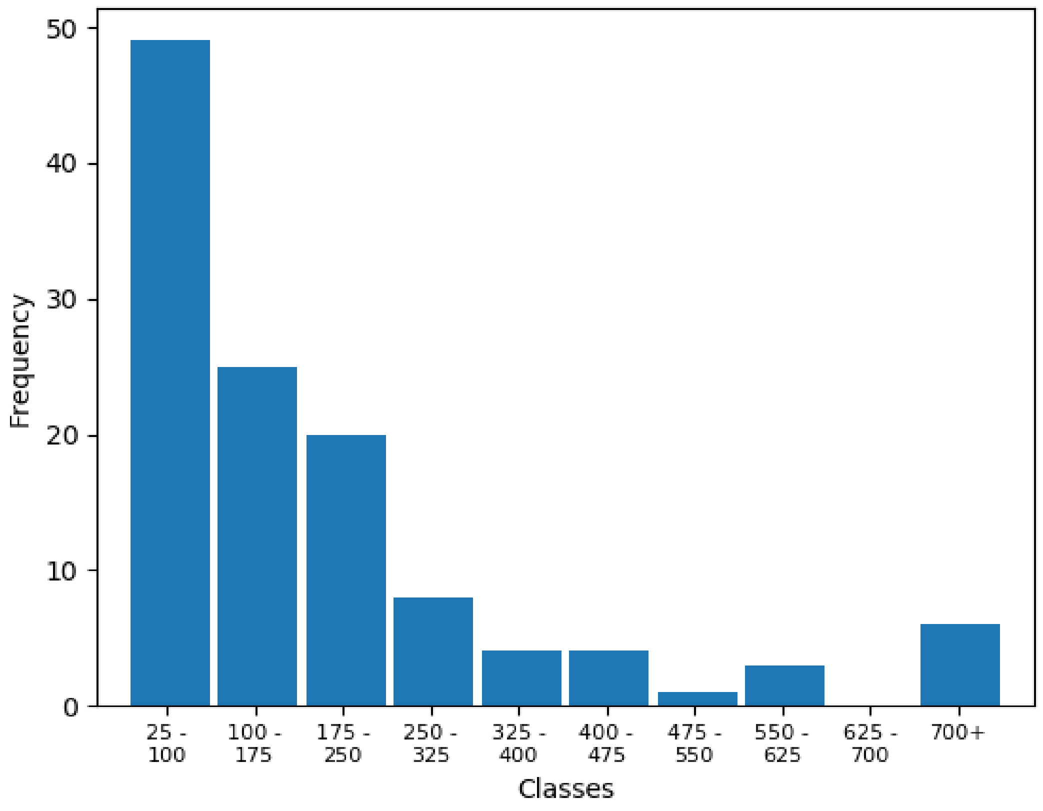


Figure 23: Histogram of collected projects by class count

6.2 Setup

We implement the methodology presented by building MicroRefact. We use the tool to test the collected projects to evaluate the percentage of projects that the tool is able to refactor and to evaluate from a functional point of view if the generated microservices based application is equal to the original monolithic application through unit tests of the monolithic application. The setup was divided in two parts: the first part where it was necessary to obtain a microservices proposal for each of the collected projects and the second part where it was necessary to adapt the unit tests to run on the microservices based version, since they were designed for monolithic application.

To obtain a microservice proposal for each project collected, we used the tool developed by Brito et al., 2021 available at https://github.com/miguelfbrito/microservice-identification. Although the tool allows the customization of the input, we use the default parameters. For each project the tool generates several proposals for decomposition of the monolith and we always choose the one that reveals the greatest value in the metrics that the tool uses to evaluate the proposed decomposition.

After running MicroRefact on the project under analysis, the result is analyzed. If successful, and if the project has unit tests, these are run on the generated microservices-based application. However, the testing process has not been automated because, although the code generation is automated, it is necessary to define *SQL* queries for the methods that were added to the `Repository` classes and because the unit tests need some adjustments that depend on the context. Typically they are simple *SELECT* queries or data *UPDATE* queries.

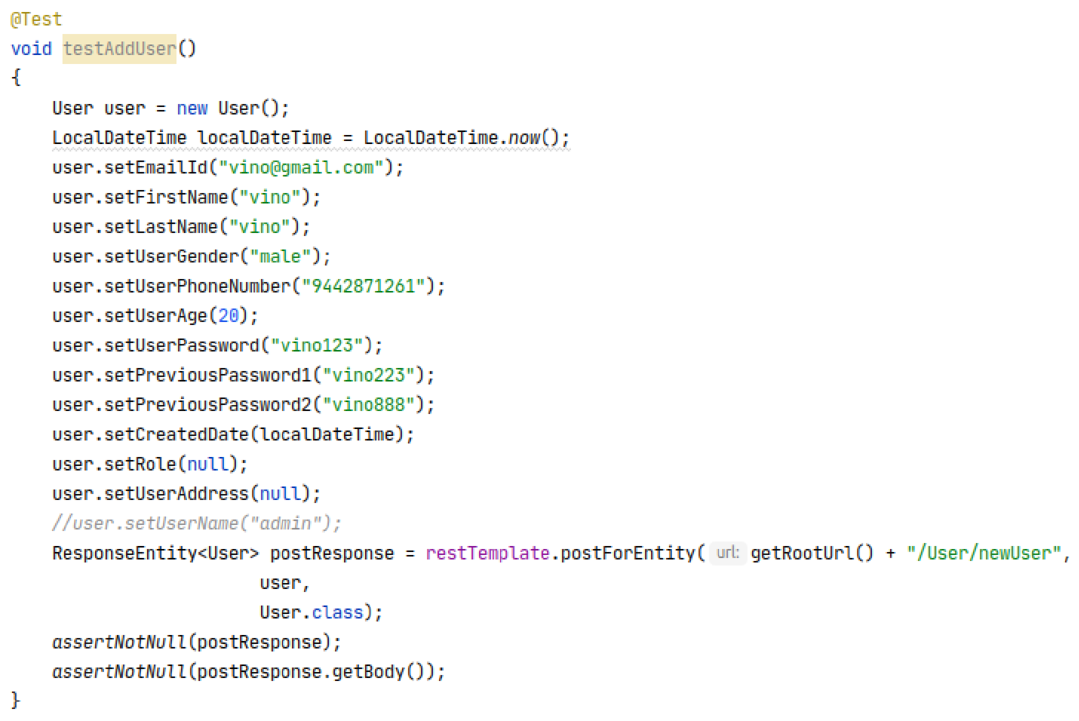
As the tests do not enter into the refactoring process, it was necessary to identify to which microservices the test classes should be transferred. This process was manual and had a detailed analysis of the test classes as well as the domain of the generated microservices. In projects with a good design and developed with good practices the test class name served to identify to which microservice it should be transferred. In more disorganized projects it was done an analysis of the imports of the test class as well as the declared variables in order to identify to which microservice it should belong. In addition, and given that the tests were designed for monolithic application, it was necessary to adapt the tests for microservices. Among the modifications made to the test classes are imports because the test classes contained references to classes that do not belong to the microservice. These imports were replaced with imports that refer to types created to replace the reference to classes that do not belong to the microservice, i.e. the DTO and interfaces created.

The other change made was in tests that manipulate the database. As unit tests are designed for monolithic application, there are unit tests that test the interactions between classes that are mapped as database entities. In the unit tests where new records are inserted in the database tables, and as we are in the scope of applications that use annotations to define the database logical schema, these create objects and use the `Repository` classes to insert them. However the objects created have as attributes objects that refer to other entities, so the objects that refer to other entities also have to be instantiated. In the monolithic version when the test is run, insertions are made in all tables referring to the classes involved in the test. However, in microservices and with database refactoring, the references to classes that are mapped as entities that do not belong to the microservice have been replaced by DTOs with the same name and therefore the unit tests create the objects through the instantiation of a DTO class except for the entities that belong to the microservice that are created in the database. The objects created through the instantiation of a DTO class exist in memory in the microservice where the unit test is executed but the record corresponding to the object does not exist in the microservice database where the entity that is represented by the DTO belongs, which causes methods like *setters* to fail. In addition there are also unit tests where the creation of a record in the database is tested by instantiating an object and using setters to set the values of other entities to null. In this way the test is focused only on the entity that belongs to the microservice.

6.2.1 Example

To illustrate the modifications made to the unit tests we use a practical example from one of the projects present in the evaluation. Figure 24 shows a unit test for creating a `User` in the *Online-medicine-shopping-ecommerce*[1]. The `User` class has the following instance variables: `userId`, `emailId`, `firstName` ,`lastName`, `userAge`, `userGender`, `userPhoneNumber`, `userPassword`, `previousPassword1`, `previousPassword2`, `createdDate` `role` and `userdAdress`. Except for the last two variables, all variables have primitive language types. However, `role` is of type `Role` which is a class that is mapped as an entity and `userAddress` is of type `Address` which is also a class that is mapped as an entity and both entities have a one-to-one relationship with `User`. The microservices proposal generated for *Online-medicine-shopping-ecommerce* indicates that the `User`, `Address` and `Role` classes belong to different microservices.

In this test a `User` object is instantiated that through setters sets the values for each attribute of the object. The setters `setRole` and `setUserAdress` receive a null as input instead of an object. This test as it is fails in the microservices based application because the relationship between `User` and `Address` and `User` and `Role` has been refactored. The setters `setRole` and `setUserAddress` now call the respective services to update the data. These calls have as parameter the primary key of the entities in question, so when making a service call with `null` it results in a 404 not found error because the passage of the primary key in the request is made through the URL.

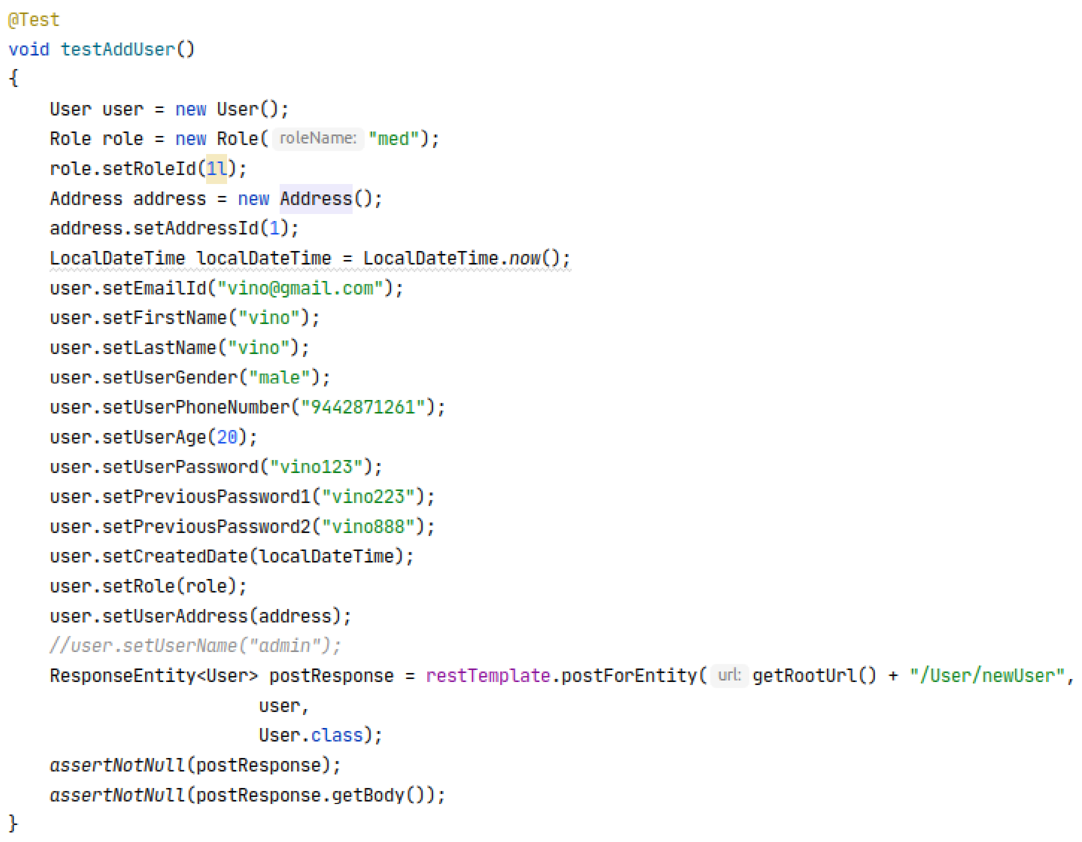


```java
@Test
void testAddUser()
{
    User user = new User();
    LocalDateTime localDateTime = LocalDateTime.now();
    user.setEmailId("vino@gmail.com");
    user.setFirstName("vino");
    user.setLastName("vino");
    user.setUserGender("male");
    user.setUserPhoneNumber("9442871261");
    user.setUserAge(20);
    user.setUserPassword("vino123");
    user.setPreviousPassword1("vino223");
    user.setPreviousPassword2("vino888");
    user.setCreatedDate(localDateTime);
    user.setRole(null);
    user.setUserAddress(null);
    //user.setUserName("admin");
    ResponseEntity<User> postResponse = restTemplate.postForEntity( url: getRootUrl() + "/User/newUser",
                        user,
                        User.class);
    assertNotNull(postResponse);
    assertNotNull(postResponse.getBody());
}
```

Figure 24: Original unit test for creating a user in Online-medicine-shopping-ecommerce

[1] https://github.com/ariv98/Online-medicine-shopping-ecommerce

A possible solution is to instantiate objects of type `Address` and `Role` and set them in the `User` as shown in Figure 25. However, these would be instantiated through DTO classes which would cause the information stored in the objects not to exist in the `Address's` microservice database and in the `Role's` microservice database, causing the `setRole` and `setUserAddress` methods to try to update records that do not exist.



```java
@Test
void testAddUser()
{
    User user = new User();
    Role role = new Role( roleName: "med");
    role.setRoleId(1l);
    Address address = new Address();
    address.setAddressId(1);
    LocalDateTime localDateTime = LocalDateTime.now();
    user.setEmailId("vino@gmail.com");
    user.setFirstName("vino");
    user.setLastName("vino");
    user.setUserGender("male");
    user.setUserPhoneNumber("9442871261");
    user.setUserAge(20);
    user.setUserPassword("vino123");
    user.setPreviousPassword1("vino223");
    user.setPreviousPassword2("vino888");
    user.setCreatedDate(localDateTime);
    user.setRole(role);
    user.setUserAddress(address);
    //user.setUserName("admin");
    ResponseEntity<User> postResponse = restTemplate.postForEntity( url: getRootUrl() + "/User/newUser",
                    user,
                    User.class);
    assertNotNull(postResponse);
    assertNotNull(postResponse.getBody());
}
```

Figure 25: Unit test for creating a user in Online-medicine-shopping-ecommerce

We decided to adapt this solution for the cases of setters set `null` and we executed the unit test on both versions of the project. To avoid having an object instantiated in a microservice and no corresponding record in the database of the microservice that owns the entity, before running the unit test a record is inserted in the database that will correspond to the object instantiated during the test run. This way, there is a small adaptation in the test to simulate the monolith test environment in the microservices based application

To run the microservices, we set the environment properties port and *MySQL*[2] database connection for each microservice. The remaining environment properties were copied from the monolithic version of the project under analysis.

[2]https://www.mysql.com/

6.3 Results

The results for each project can be found at Appendix B.1 . For each project the following data is presented: project name, number of classes of the monolithic version of the project, number of proposed microservices (#PM), if a microservices based application was generated, the number of microservices generated by refactoring (#MG), the number of classes of the microservices based version of the project (#CAR), the number of relationships between entities refactored (#RR), the number of dependencies between classes of different microservices (#DR), percentage of new classes in the microservices-based application (%NC),if the project has tests, the result of the test execution and the justification why the refactoring failed, if applicable.

MicroRefact was able to refactor approximately 69% of the applications and within the universe of refactored projects 33% have unit tests. We used the unit tests to evaluate the equity of the applications by comparing the output of the tests in the two versions of the application with the input of unit tests being the same in both versions. All the unit tests executed had the same output in both versions of the application, which shows that the refactoring was successful.

The blox plot of the percentage of new classes in the projects is shown in Figure 26. The percentage of generated classes for the projects is bound between -40 and 234 with the median around 73.

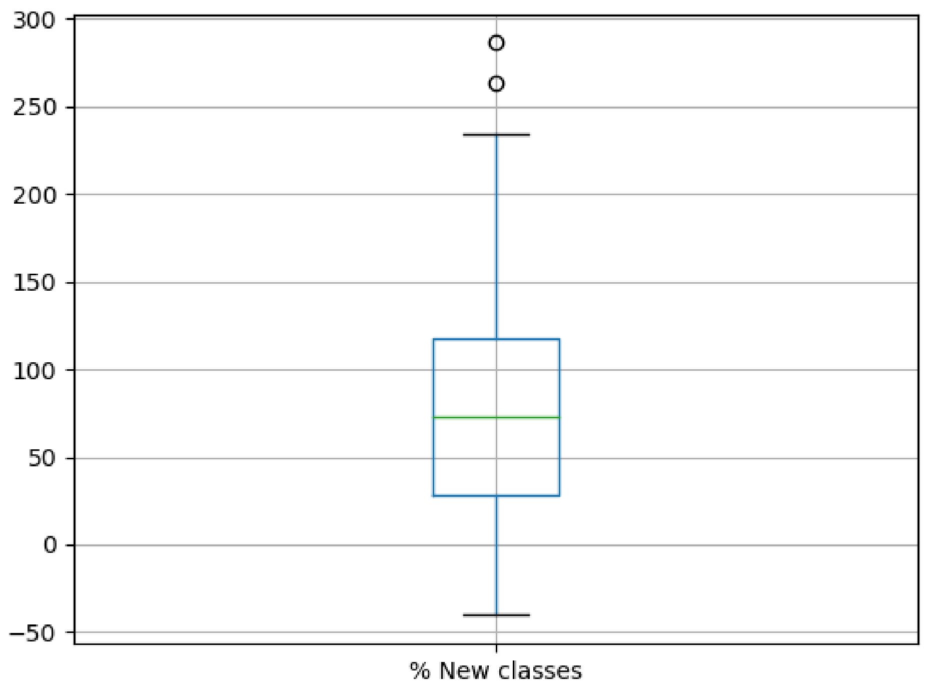


Figure 26: Box plot of percentage of new classes across the projects

6.4 Analysis

The dataset used for the study consists of projects with a wide range of class numbers and different domains. Furthermore, there are projects where the percentage of new classes is negative, which seems strange since the proposed methodology foresees the creation of classes. In this section we introduce a more extensive analysis of the results obtained.

6.4.1 Degree of complexity and size of the applications

In this chapter we try to understand the impact of application size and complexity on the percentage of refactored applications. A large percentage, approximately 41%, of the projects used in the evaluation are projects where the range of classes is between 25 and 100. One might think that these projects are small and or have a low degree of complexity. As the class range in which the most projects were refactored was this range, one could think that the tool is not prepared for projects of great magnitude and high complexity. However, we must understand the context of the study. Since we are using applications that use annotations to apply the ORM technique, the number of classes in these projects tends to be smaller than projects that do not use this technique because they do not need to declare classes to define DAO's, being these projects complex but with a low number of classes. Furthermore, as shown in Figure 27, the tool was able to refactor projects in all ranges of class numbers, which shows that the tool is prepared to support all degrees of complexity present in these projects.

6.4.2 Unrefactored projects

Regarding the unrefactored projects, there are different reasons why refactoring did not happen. Table 4 shows the reasons why projects were not refactored and the number of projects that fit into it.

Reason	#Projects
Missing a repository class	18
Missing @id annotation	6
Use DAOs	10
@Id annotation in method instead of variable	1
Use other ORM frameworks	2

Table 4: Reasons for unrefactored

About 50% of the unrefactored projects are projects where the class `Repository` is missing. We can divide the reason why refactoring fails due to the lack of the `Repository` class into two types: the `Repository` class exists but in the microservices proposal it is in a different microservice than the `Entity` class that the `Repository` class refers to; an `Entity` class exists that does not have a corresponding `Repository` class. For the first case a different microservice proposal where the `Entity`

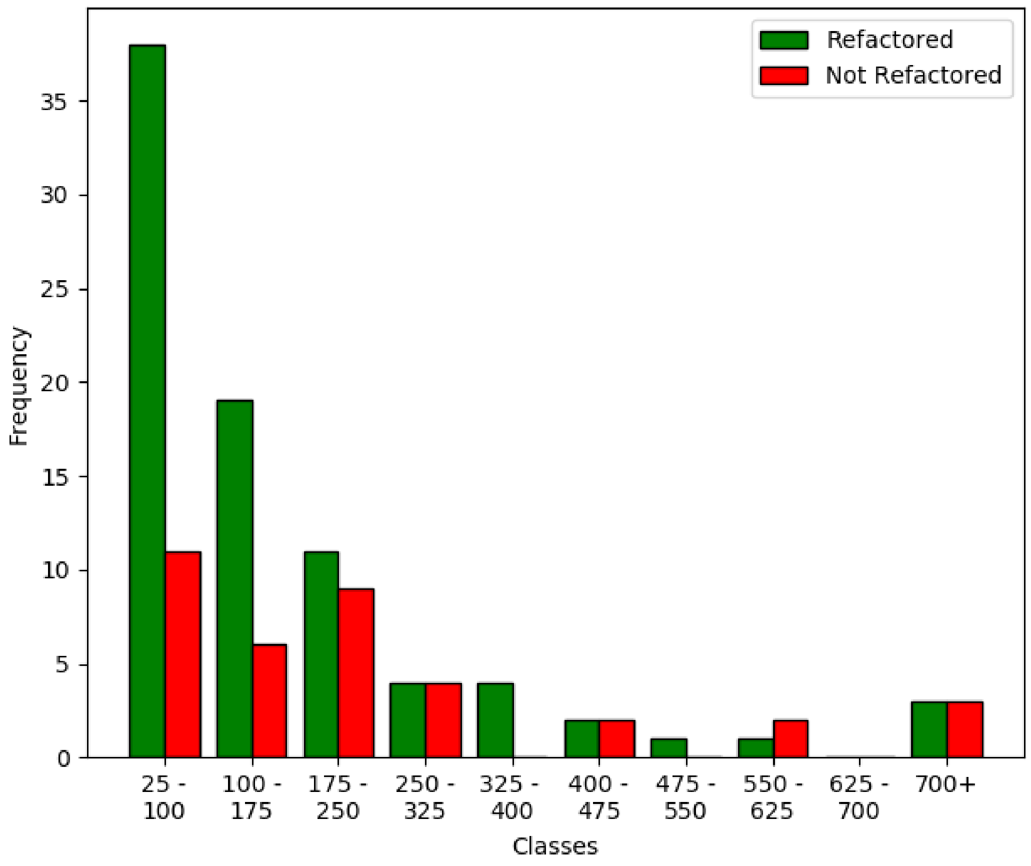


Figure 27: Histogram of number of refactored projetcts by class count

class and the corresponding `Repository` class are in the same microservice would solve the problem. In the second case refactoring is not possible since there is an `Entity` class where there is no `Repository` class that allows the retrieval of data from the database, which may be an error in the code or dead code. We identify the `Repository` classes to apply the *Move Foreign-Key Relationship to Code Pattern* (S. Newman, 2019) , because it implies adding methods to these classes so that the join is possible.

Another reason why refactoring was not possible is the lack of the `Id` annotation in classes that are mapped to database entities. In the database refactoring phase it is necessary to identify the primary key of the entities involved in the relationship that is under analysis because in the application of the *Move Foreign-Key Relationship to Code Pattern* the primary key of one of the entities is used as a parameter in the invocation of the REST API generated to filter the information retrieved during the join. The identification of the primary key is accomplished through the `Id` annotation on the class instance variables. We also cover the cases in which the class `Entity` under analysis is a sub class and in which the `Id` annotation is in the super class, which means, if the `Id` annotation in the class `Entity` under analysis is not found in one of the instance variables it is checked if in the super class from which the class `Entity` extends there is the `Id` annotation in one of the instance variables. Without the identification of the primary key, the refactoring is not possible, because the calls to the service to make the join of the data would not have a filter, being able to access information that should not be accessed.

The use of DAOs by some projects together with `Repository` classes led to refactoring in these projects failing, since some `Entity` classes have a corresponding `Repository` class and others have a DAO class. MicroRefact only supports refactoring of `Entity` classes that have a corresponding `Repository` class.

The other reasons why refactoring is not possible are also implementation reasons. MicroRefact only searches for the primary key in instance variable annotations and only supports JPA annotations.

Overall, MicroRefact can be extended to support some of the cases where it failed, namely the use of DAOs, other ORM frameworks and annotations on methods. The remaining cases cannot be included in MicroRefact because they go against the proposed methodology because the lack of `Id` annotation on the classes mapped as entities and the lack of `Repository` classes for some entities may compromise the refactoring of the application.

6.4.3 Refactored projects with fewer classes than the original projects

First of all, we must understand how the class count of each project was made and how the percentage of new classes in the project was calculated. For the class count of each project we used the *find* command, Listing 6.1, from the shell with the *-regex* and *-f* options combined with the *wc* command with the *-l* argument.

```
$ find {path_to_project} -regex .*java -type f | wc -l
```

Listing 6.1: Number of classes in a project.

To calculate the percentage of new classes we use the following formula:

$$PNC = \frac{100 * (NCMS - NCM)}{NCM}$$

where PNC is the percentage of new classes, NCMS is the number of classes of the microservices based application and NCM is the number of classes of the monolithic application.

Looking at the results it can be observed that in some projects the number of classes of the original application is larger than the generated microservices based application. This happens because the tool used to generate the microservices proposals sometimes does not use all the classes of the projects. As it uses clustering techniques to group the classes by microservices, the classes that do not have dependencies with other classes nor have dependent classes end up not being used in the microservices proposal. Besides that, MicroRefact only handles at the classes that are in the microservices proposal. Thus, the refactoring is performed based on the classes that are in the proposed microservices and it is possible that the number of new classes created during refactoring is smaller than the number of classes that are outside the microservices proposal reflecting in a negative percentage of new classes.

6.4.4 Percentage of new classes and analysis

In this section, we perform a correlation analysis between the number of classes generated, the number of relationships between entities, and the number of dependencies between classes using some examples to understand the impact these have in the number of new classes generated.

The wide range of the percentage of new classes in the generated applications could be justified by the fact that the microservices proposal does not contemplate all the classes of the projects. However, we decided to better understand the impact of the number of relationships between entities that need refactoring and the number of dependencies between classes of different microservices on the percentage of new classes. First, it is necessary to understand how many classes are typically generated for each refactoring case. Starting with the relationships between entities, the relationships `OneToMany`, `OneToOne`, and `ManyToOne` generate at least five classes: an interface (`Request`), a class that implements the interface (`RequestImpl`), a class where the API is defined (`Controller`), another to process the request (`Service`) and a DTO class to represent the type of entity in the foreign microservices that if it has dependencies are also reflected in the creation of more DTO classes, so the number of classes generated may be greater than five, which happens in projects where there are several relationships between entities from different microservices. In terms of the percentage of new classes, if we look at the classes that trigger the refactoring, it is an addition of at least 250% (ratio 2.5) . On the other hand, the `ManyToMany` relationship indirectly generates new classes, since it is the original classes that are dragged to the new microservice, however, if the entities involved have another type of relationship with entities that belonged to the same microservice these relationships end up being refactored leading to the generation of new classes.

As for the dependencies at the source code level between classes of different microservices, at least three classes are generated: an interface to represent the foreign type, a class that implements the interface and makes the call to the service, and a class that creates the API that allows the invocation of the original method. In addition, DTO classes can also be generated if the invoked methods have parameter or return types that are foreign to the calling microservice resulting in a percentage of new class added of at least 150% compared to the number of classes that trigger the refactoring (ratio 1.5).

The results of the percentage of new classes created in each project by the range of classes of the project are presented in Figure 28. The projects that have a higher percentage of new classes are the projects between 25 and 100 classes, which is expected because the addition of a class in these projects causes a higher percentage of new classes because they are smaller projects. The percentage of new classes tends to decrease along the class range. In the range from 325 classes to 400 there is an outlier with a value close to 219% corresponding to the *EUSURVEY*[3] project, which is justified by the high number of dependencies between classes of different microservices (6949 dependencies).

The projects that present the highest percentage value of new classes are the projects where the relationships between entities were refactored, which is also expected since it is in the database refactoring

[3]https://github.com/EUSurvey/EUSURVEY

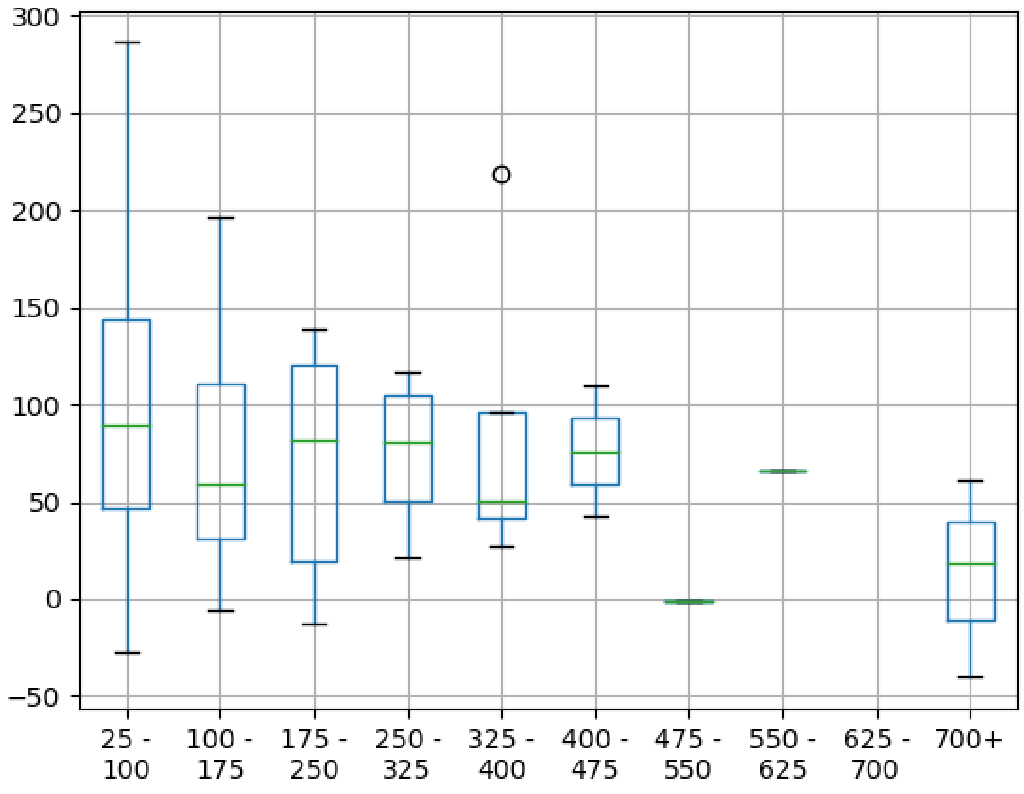


Figure 28: Box plot for the percentage of new classes across ranges of classes

where more classes are created. Several projects did not have relationships between entities refactored because they are projects that do not use the annotations in the instance variables that denote the relationships,for example, *Online_Movie_Ticket_System*[4] (111%) and *zammc-manage*[5] (29%). Others projects did not have relationships between entities because there are no relationships between entities of different microservices, for example, *mwebsvr*[6] (48%), which is translated into a lower percentage of new classes relative to projects of the same dimension in which there was a refactoring of the relationships, for example, *Champik*[7] (197%), *ProyectoUNAM*[8] (149%), and *OA_system*[9] (125%) .

[4] https://github.com/samyumaddikunta/Online_Movie_Ticket_System
[5] https://github.com/SupermePower/zammc-manage
[6] https://github.com/YangLeeThomson/mwebsvr
[7] https://github.com/lionsdamajectim/Champink
[8] https://github.com/CocayUNAM/ProyectoUNAM
[9] https://github.com/wn150509/OA_system

6.5 Threats to validity

In this section we discuss the threats to the work presented.

A possible threat is related to the use of only open source applications in the evaluation. However, it is common to find companies that make their code available. Nevertheless, it is possible that the results may vary for proprietary software.

Other threat is the quality of the microservice proposals generated for the projects. The tool used to generate microservice proposals does not guarantee that the decomposition it proposes is the best possible. Furthermore, the developers did not perform an extensive study of how the number of topics and the resolution directly affect the microservice proposal. Instead, they defined arbitrary ranges for each parameter. However they evaluated the quality of the proposed solution by using the microservice architecture metrics proposed by Jin et al., 2021 and the results were positive.

Another threat is the unit tests available by the projects. Some projects present no tests and others present few tests. For the projects that do not present tests it was not possible to evaluate if from the functional point of view the generated application is equal to the original. For the applications that present few tests, these can be class-focused, not testing the interactions between classes, leaving several interactions between classes of different microservices to be tested.

7 Conclusion

We present a methodology for refactoring original Java monolithic applications into microservices-based applications. The proposed methodology is directed to the applications that take advantage of ORM technique to relate Java classes to database entities, in particular applications that make the mapping between classes and entities through annotations in the source code which allows refactoring the database by refactoring the application source code.

The proposed methodology receives as input the source code of the application under analysis and a set of microservices, in which each microservice is described by the set of classes that composes that microservice. Each class must belong to just one microservice. Our methodology has as its output a microservices-based application that from the functional point of view is the same as the monolithic application under analysis.

We built a tool, MicroRefact, as a proof of concept, that supports Java Spring applications that use JPA annotations for mapping between Java classes and database entities and performed a quantitative evaluation where we tried to understand the applicability of the methodology and a qualitative evaluation where we tried to understand if the microservice-based applications generated are functionally equivalent to the original monolithic Java applications through the execution of unit tests provided by the monolithic application. The evaluation was conducted against the collection of 120 open-source Java Spring applications from GitHub.

The results show that 69% of the applications were automatically refactored. Regarding the qualitative evaluation all tests performed had the same result in both versions of the application.

7.1 Future work

The work presented has some limitations and so there is potential for improvement and extension of the work. Some points for improvement and future work are presented below:

- The methodology allows inheritance across different microservices, i.e. the super-class and sub-classes being in different microservices in the microservice proposal received as input. Just as we do for interfaces, we could make a copy of the super-class and place it in the microservice where the sub-class belongs.

- Extend the MicroRefact to applications that use XML files to map Java classes to database entities. Although applications that use XML files can be converted to applications that use annotations there is still a considerable amount of applications that use XML. Extending the methodology to support these applications requires a thorough analysis of the XML files and the relationship they have with Java classes.
- Extend MicroRefact to support other object-oriented languages like C# and Python. The methodology covers the OOP concepts so it is possible to extend the developed tool to automatically refactor projects developed in other object-oriented languages.
- Extend MicroRefact enabling the choice of the language to which one wants to migrate the microservices. Sometimes what triggers the migration to microservices is the desire to change technology.

Bibliography

Abdullah, M., Iqbal, W., & Erradi, A. (2019). Unsupervised learning approach for web application auto-decomposition into microservices. *Journal of Systems and Software*, *151*, 243–257. https://doi.org/https://doi.org/10.1016/j.jss.2019.02.031 (cit. on pp. 13, 15)

Ambler, S. W., & Sadalage, P. J. (2006). *Refactoring databases: Evolutionary database design*. Addison-Wesley Professional. (Cit. on p. 11).

Andriole, S. J. (2017). The death of big software. *Commun. ACM*, *60*(12), 29–32. https://doi.org/10.1145/3152722 (cit. on p. 1)

Asseldonk, L. v. (2021). *From a monolith to microservices: The effect of multi-view clustering* (Doctoral dissertation). Utrecht University. (Cit. on p. 14).

Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D. A., & Lynn, T. (2018). Microservices migration patterns. *Software: Practice and Experience*, *48*(11), 2019–2042. https://doi.org/https://doi.org/10.1002/spe.2608 (cit. on pp. 9, 10)

Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent dirichlet allocation. *J. Mach. Learn. Res.*, *3*, 993–1022 (cit. on p. 13).

Blondel, V., Guillaume, J.-L., Lambiotte, R., & Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of Statistical Mechanics Theory and Experiment*, *2008*. https://doi.org/10.1088/1742-5468/2008/10/P10008 (cit. on p. 14)

Brito, M., Cunha, J., & Saraiva, J. (2021). Identification of microservices from monolithic applications through topic modelling. *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 1409–1418. https://doi.org/10.1145/3412841.3442016 (cit. on pp. 13, 14, 16, 18, 47)

Chen, R., Li, S., & Li, Z. (2017). From monolith to microservices: A dataflow-driven approach. *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, 466–475. https://doi.org/10.1109/APSEC.2017.53 (cit. on pp. 1, 2, 14)

Cito, J., Leitner, P., Fritz, T., & Gall, H. C. (2015). The making of cloud applications: An empirical study on software development for the cloud. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 393–403. https://doi.org/10.1145/2786805.2786826 (cit. on p. 1)

Edstrom, J., & Tilevich, E. (2012). Reusable and extensible fault tolerance for restful applications. *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, 737–744. https://doi.org/10.1109/TrustCom.2012.244 (cit. on p. 12)

Evans, E. (2004). *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley. (Cit. on p. 11).

Fowler, M. (2002). Localdto [(Accessed on 10/02/2021)]. (Cit. on p. 26).

Fowler, M. (2004). Stranglerfigapplication [(Accessed on 11/20/2020)]. (Cit. on p. 10).

Fowler, M., & Lewis, J. (2014). Microservices [(Accessed on 11/30/2020)]. (Cit. on pp. 5, 15).

Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., & Stafford, R. (2002). *Patterns of enterprise application architecture*. Addison-Wesley Professional. (Cit. on pp. 4, 26, 45).

Freitas, F., Ferreira, A., & Cunha, J. (2021). Refactoring java monoliths into executable microservice-based applications. *25th Brazilian Symposium on Programming Languages*, 100–107. https://doi.org/10.1145/3475061.3475086 (cit. on p. 3)

Fritzsch, J., Bogner, J., Wagner, S., & Zimmermann, A. (2019). Microservices migration in industry: Intentions, strategies, and challenges. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. https://doi.org/10.1109/ICSME.2019.00081 (cit. on p. 1)

Girvan, M., & Newman, M. (2001). Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of the United States of America*, *99*, 7821–7826 (cit. on p. 14).

Gysel, M., Kölbener, L., Giersche, W., & Zimmermann, O. (2016). Service cutter: A systematic approach to service decomposition. In M. Aiello, E. B. Johnsen, S. Dustdar, & I. Georgievski (Eds.), *Service-oriented and cloud computing* (pp. 185–200). Springer International Publishing. (Cit. on pp. 2, 14, 15).

Hamzehloui, M., Sahibuddin, S., & Ashabi, A. (2019). A study on the most prominent areas of research in microservices. *International Journal of Machine Learning and Computing*, *9*, 242–247. https://doi.org/10.18178/ijmlc.2019.9.2.793 (cit. on p. 1)

Henderson-Sellers, B., Ralyte, J., Ågerfalk, P., & Rossi, M. (2014). *Situational method engineering*. https://doi.org/10.1007/978-3-642-41467-1. (Cit. on p. 9)

Hubbell, D. (2020). Top 4 pros and cons of microservices architecture [(Accessed on 7/12/2020)]. (Cit. on p. 7).

Jin, W., Liu, T., Cai, Y., Kazman, R., Mo, R., & Zheng, Q. (2021). Service candidate identification from monolithic systems based on execution traces. *IEEE Transactions on Software Engineering*, *47*(5), 1–1. https://doi.org/10.1109/TSE.2019.2910531 (cit. on pp. 2, 14, 57)

Jin, W., Liu, T., Zheng, Q., Cui, D., & Cai, Y. (2018). Functionality-oriented microservice extraction based on execution trace clustering. *2018 IEEE International Conference on Web Services (ICWS)*, 211–218 (cit. on p. 2).

Kamimura, M., Yano, K., Hatano, T., & Matsuo, A. (2018). Extracting candidates of microservices from monolithic application code. *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, 571–580. https://doi.org/10.1109/APSEC.2018.00072 (cit. on pp. 2, 9, 12, 13)

Kazanavicius, J., & Mazeika, D. (2019). Migrating legacy software to microservices architecture. *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, 1–5. https://doi.org/10.1109/eStream.2019.8732170 (cit. on pp. 2, 4)

Kharenko, A. (2015). Monolithic vs. microservices architecture [(Accessed on 25/11/2020)]. (Cit. on p. 4).

Kobayashi, K., Kamimura, M., Kato, K., Yano, K., & Matsuo, A. (2012). Feature-gathering dependency-based software clustering using dedication and modularity. https://doi.org/10.1109/ICSM.2012.6405308 (cit. on p. 12)

Kumar, S. (2020). Fault-tolerant patterns for microservice [(Accessed on 12/07/2020)]. (Cit. on p. 7).

Kwon, Y.-W., & Tilevich, E. (2013). Cloud refactoring: Automated transitioning to cloud-based services. *Automated Software Engineering*, *21*. https://doi.org/10.1007/s10515-013-0136-9 (cit. on pp. 12, 13)

Leung, I. X. Y., Hui, P., Liò, P., & Crowcroft, J. (2009). Towards real-time community detection in large networks. *Physical review. E, Statistical, nonlinear, and soft matter physics*, *79 6 Pt 2*, 066107 (cit. on p. 14).

Mazlami, G., Cito, J., & Leitner, P. (2017). Extraction of microservices from monolithic software architectures. *2017 IEEE International Conference on Web Services (ICWS)*, 524–531. https://doi.org/10.1109/ICWS.2017.61 (cit. on p. 2)

Mazzara, M., Dragoni, N., Bucchiarone, A., Giaretta, A., Larsen, S. T., & Dustdar, S. (2018). Microservices: Migration of a mission critical system. *IEEE Transactions on Services Computing*, 1–1. https://doi.org/10.1109/TSC.2018.2889087 (cit. on pp. 1, 2)

Newman, S. (2015). *Building microservices: Designing fine-grained systems*. O'Reilly Media. https://books.google.pt/books?id=jjl4BgAAQBAJ. (Cit. on pp. 1, 6, 7)

Newman, S. (2019). *Monolith to microservices: Evolutionary patterns to transform your monolith*. O'Reilly Media, Incorporated. https://books.google.pt/books?id=iuI3wQEACAAJ. (Cit. on pp. 4, 5, 24, 27, 33, 53)

Pahl, C., & Jamshidi, P. (2016). Microservices: A systematic mapping study. *Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2*, 137–146. https://doi.org/10.5220/0005785501370146 (cit. on p. 2)

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM*, *15*(12), 1053–1058. https://doi.org/10.1145/361598.361623 (cit. on p. 9)

Richardson, C. (2018). *Microservices patterns: With examples in java*. Manning Publications. https://books.google.pt/books?id=UeK1swEACAAJ. (Cit. on pp. 4–8, 10, 11)

Tyszberowicz, S., & Raman, A. (2007). The easycrc tool. *2007 International Conference on Software Engineering Advances*, 52. https://doi.org/10.1109/ICSEA.2007.72 (cit. on p. 14)

Tyszberowicz, S., Heinrich, R., Liu, B., & Liu, Z. (2018). Identifying microservices using functional decomposition. In X. Feng, M. Müller-Olm, & Z. Yang (Eds.), *Dependable software engineering. theories, tools, and applications* (pp. 50–65). Springer International Publishing. (Cit. on p. 14).

Yanaga, E. (2017). *Migrating to microservice databases: From relational monolith to distributed data*. O'Reilly Media. https://books.google.pt/books?id=5D9UwAEACAAJ. (Cit. on p. 15)

A Appendix

A.1 Dependencies of restaurantServer classes with classes that belong to different microservices.

Microservice 1

pl.edu.wat.wcy.pz.restaurantServer.security.WebSecurityConfiguration
pl.edu.wat.wcy.pz.restaurantServer.security.service.UserDetailsServiceImpl

pl.edu.wat.wcy.pz.restaurantServer.security.jwt.JwtAuthTokenFilter
pl.edu.wat.wcy.pz.restaurantServer.security.service.UserDetailsServiceImpl

pl.edu.wat.wcy.pz.restaurantServer.security.jwt.JwtProvider
pl.edu.wat.wcy.pz.restaurantServer.security.service.UserPrinciple

Microservice 2

pl.edu.wat.wcy.pz.restaurantServer.entity.User
pl.edu.wat.wcy.pz.restaurantServer.entity.Role
pl.edu.wat.wcy.pz.restaurantServer.entity.Reservation

pl.edu.wat.wcy.pz.restaurantServer.service.UserService
pl.edu.wat.wcy.pz.restaurantServer.entity.Reservation

pl.edu.wat.wcy.pz.restaurantServer.controller.UserController
pl.edu.wat.wcy.pz.restaurantServer.email.MailService
pl.edu.wat.wcy.pz.restaurantServer.entity.Reservation

Microservice 3

pl.edu.wat.wcy.pz.restaurantServer.entity.BillPosition
pl.edu.wat.wcy.pz.restaurantServer.entity.Dish

pl.edu.wat.wcy.pz.restaurantServer.service.BillService
pl.edu.wat.wcy.pz.restaurantServer.repository.RTableRepository
pl.edu.wat.wcy.pz.restaurantServer.entity.RTable

pl.edu.wat.wcy.pz.restaurantServer.controller.BillController
pl.edu.wat.wcy.pz.restaurantServer.repository.RTableRepository

pl.edu.wat.wcy.pz.restaurantServer.service.BillPositionService
pl.edu.wat.wcy.pz.restaurantServer.service.DishService

Microservice 4

pl.edu.wat.wcy.pz.restaurantServer.service.RTableService
pl.edu.wat.wcy.pz.restaurantServer.entity.Bill
pl.edu.wat.wcy.pz.restaurantServer.entity.Reservation

pl.edu.wat.wcy.pz.restaurantServer.entity.RTable
pl.edu.wat.wcy.pz.restaurantServer.entity.Reservation
pl.edu.wat.wcy.pz.restaurantServer.entity.Bill

pl.edu.wat.wcy.pz.restaurantServer.controller.RTableController
pl.edu.wat.wcy.pz.restaurantServer.entity.Bill
pl.edu.wat.wcy.pz.restaurantServer.entity.Reservation

Microservice 5

pl.edu.wat.wcy.pz.restaurantServer.RestaurantServerApplication
pl.edu.wat.wcy.pz.restaurantServer.entity.Role
pl.edu.wat.wcy.pz.restaurantServer.entity.User
pl.edu.wat.wcy.pz.restaurantServer.entity.RTable
pl.edu.wat.wcy.pz.restaurantServer.repository.RoleRepository
pl.edu.wat.wcy.pz.restaurantServer.repository.UserRepository
pl.edu.wat.wcy.pz.restaurantServer.repository.RTableRepository

Microservice 6

pl.edu.wat.wcy.pz.restaurantServer.service.ReservationService
pl.edu.wat.wcy.pz.restaurantServer.service.UserService
pl.edu.wat.wcy.pz.restaurantServer.email.MailService
pl.edu.wat.wcy.pz.restaurantServer.service.RTableService
pl.edu.wat.wcy.pz.restaurantServer.entity.User
pl.edu.wat.wcy.pz.restaurantServer.entity.RTable

Microservice 7

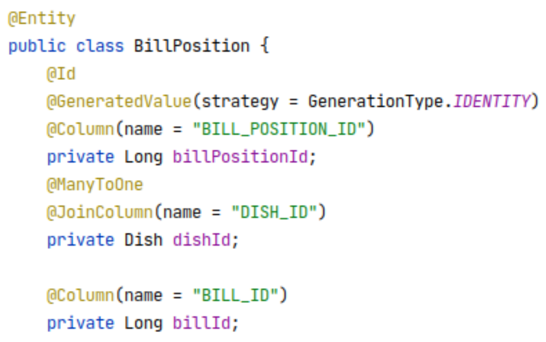
pl.edu.wat.wcy.pz.restaurantServer.form.response.JwtResponse
pl.edu.wat.wcy.pz.restaurantServer.entity.Reservation

pl.edu.wat.wcy.pz.restaurantServer.controller.AuthController
pl.edu.wat.wcy.pz.restaurantServer.repository.UserRepository
pl.edu.wat.wcy.pz.restaurantServer.security.jwt.JwtProvider
pl.edu.wat.wcy.pz.restaurantServer.entity.User

A.2 All restaurantServer transformations made by database refactoring phase

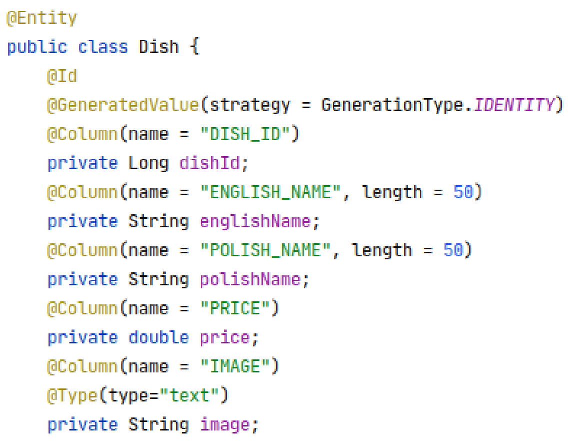
A.2.1 BillPosition - Dish

A.2.1.1 Relationship in monolithic application



```
@Entity
public class BillPosition {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "BILL_POSITION_ID")
    private Long billPositionId;
    @ManyToOne
    @JoinColumn(name = "DISH_ID")
    private Dish dishId;

    @Column(name = "BILL_ID")
    private Long billId;
```

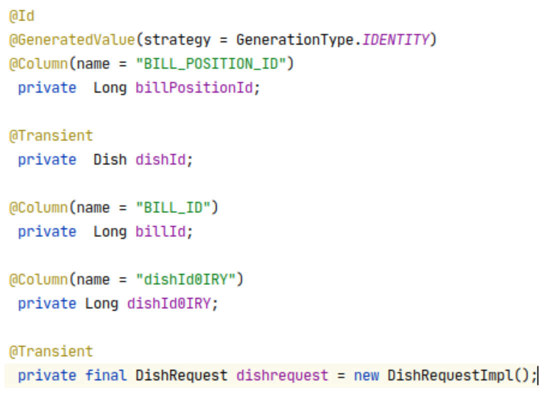
Figure 29: Instance variables of the BillPosition class in monolithic application



```
@Entity
public class Dish {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "DISH_ID")
    private Long dishId;
    @Column(name = "ENGLISH_NAME", length = 50)
    private String englishName;
    @Column(name = "POLISH_NAME", length = 50)
    private String polishName;
    @Column(name = "PRICE")
    private double price;
    @Column(name = "IMAGE")
    @Type(type="text")
    private String image;
```

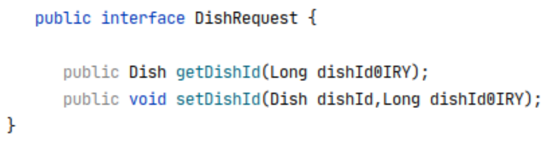
Figure 30: Instance variables of the Dish class in monolithic application

A.2.1.2 Relationship in microservice-based application



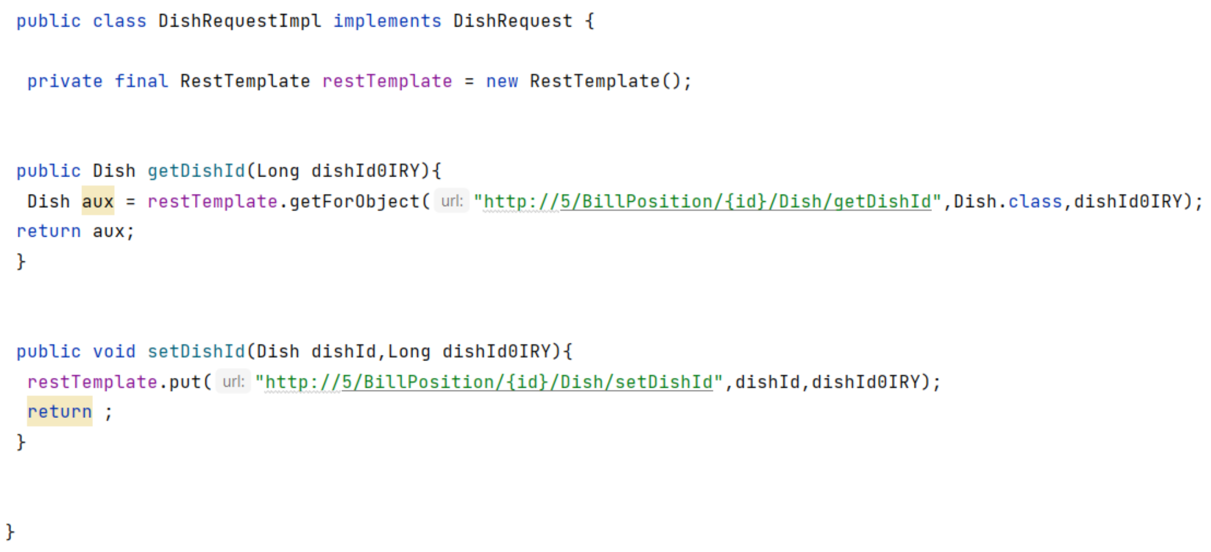
```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "BILL_POSITION_ID")
 private  Long billPositionId;

@Transient
 private  Dish dishId;

@Column(name = "BILL_ID")
 private  Long billId;

@Column(name = "dishId0IRY")
 private Long dishId0IRY;

@Transient
 private final DishRequest dishrequest = new DishRequestImpl();
```

Figure 31: Instance variables of the BillPosition class in microservice-based application



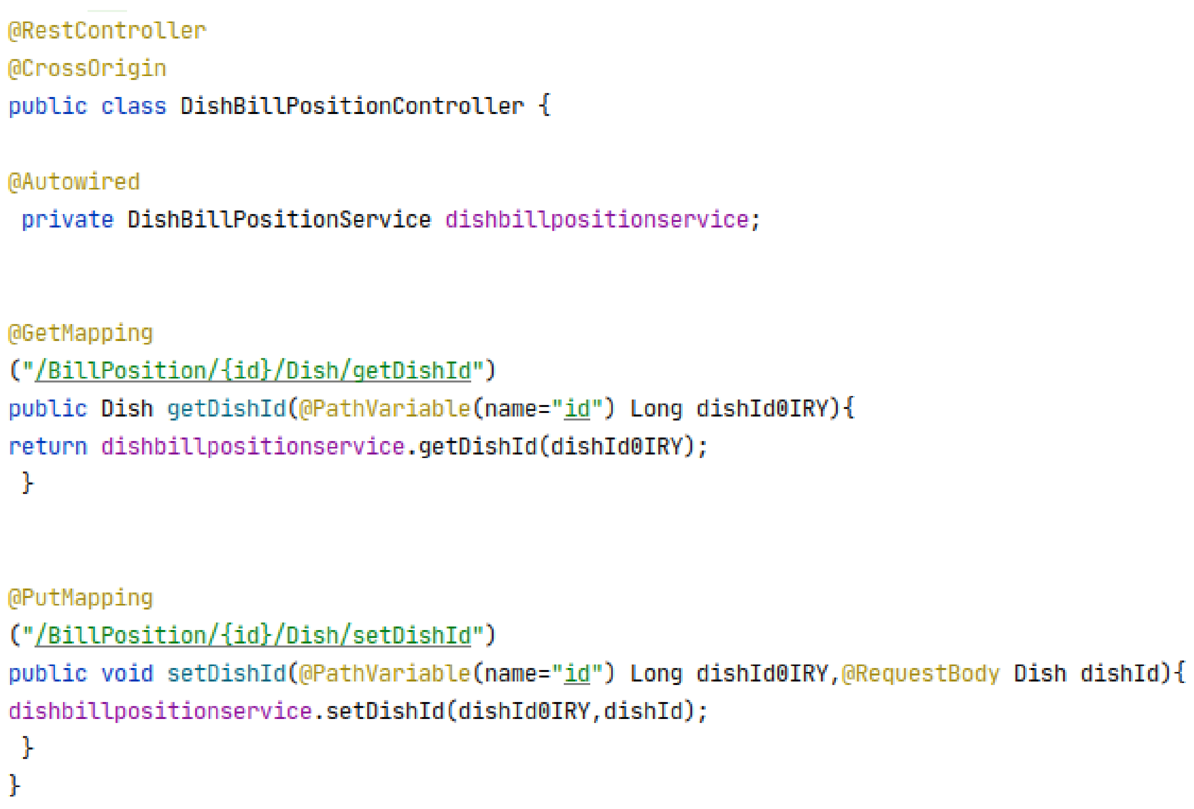
```
    public interface DishRequest {

        public Dish getDishId(Long dishId0IRY);
        public void setDishId(Dish dishId,Long dishId0IRY);
}
```

Figure 32: Interface DishRequest generated by the database refactoring



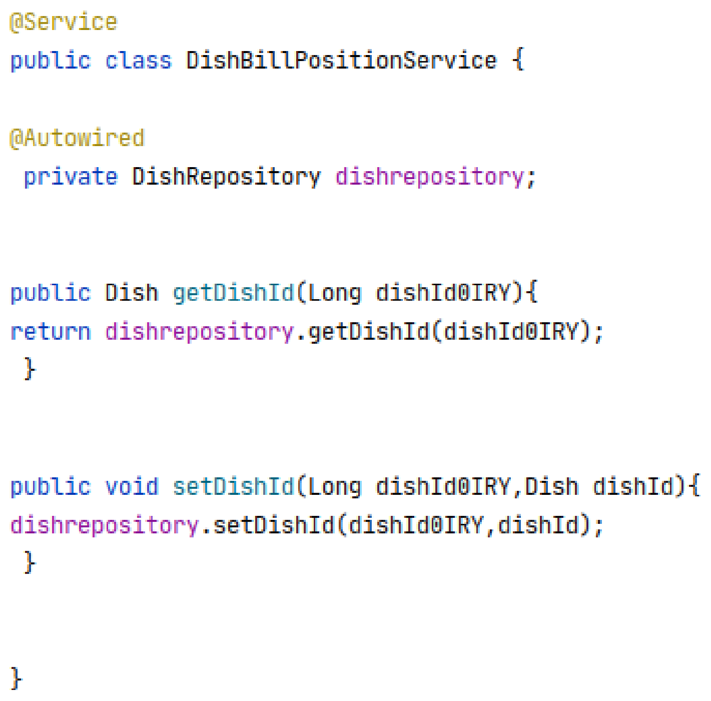
```
public class DishRequestImpl implements DishRequest {

 private final RestTemplate restTemplate = new RestTemplate();


public Dish getDishId(Long dishId0IRY){
 Dish aux = restTemplate.getForObject( url: "http://5/BillPosition/{id}/Dish/getDishId",Dish.class,dishId0IRY);
return aux;
}


public void setDishId(Dish dishId,Long dishId0IRY){
 restTemplate.put( url: "http://5/BillPosition/{id}/Dish/setDishId",dishId,dishId0IRY);
 return ;
}


}
```

Figure 33: Generated class DishRequestImpl that is responsible for the calls to the Dish microservice



```
@RestController
@CrossOrigin
public class DishBillPositionController {

@Autowired
 private DishBillPositionService dishbillpositionservice;


@GetMapping
("/BillPosition/{id}/Dish/getDishId")
public Dish getDishId(@PathVariable(name="id") Long dishId0IRY){
return dishbillpositionservice.getDishId(dishId0IRY);
 }


@PutMapping
("/BillPosition/{id}/Dish/setDishId")
public void setDishId(@PathVariable(name="id") Long dishId0IRY,@RequestBody Dish dishId){
dishbillpositionservice.setDishId(dishId0IRY,dishId);
 }
}
```

Figure 34: Generated class that exposes the API to handle Dishes

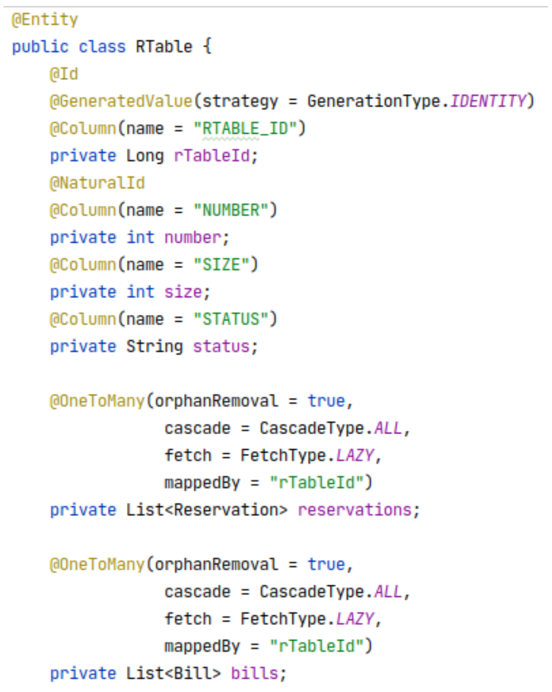


```
@Service
public class DishBillPositionService {

@Autowired
 private DishRepository dishrepository;


public Dish getDishId(Long dishId0IRY){
return dishrepository.getDishId(dishId0IRY);
 }


public void setDishId(Long dishId0IRY,Dish dishId){
dishrepository.setDishId(dishId0IRY,dishId);
 }


}
```

Figure 35: Generated class that processes and directs the request to the DishRepository class

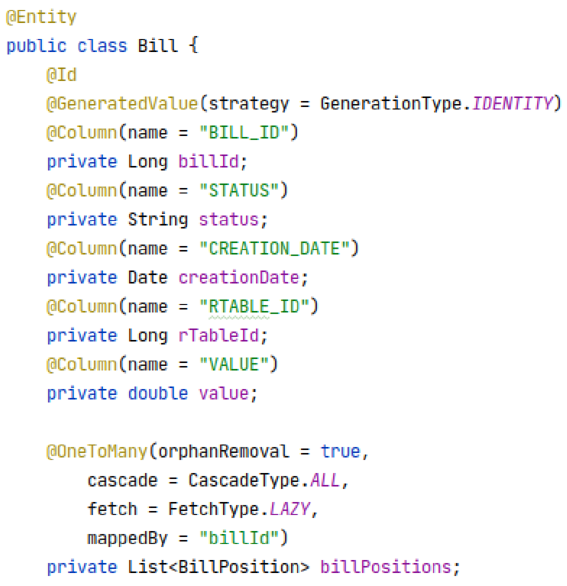
A.2.2 RTable - Reservation & RTable - Bill

A.2.2.1 Relationship in monolithic application



```java
@Entity
public class RTable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "RTABLE_ID")
    private Long rTableId;
    @NaturalId
    @Column(name = "NUMBER")
    private int number;
    @Column(name = "SIZE")
    private int size;
    @Column(name = "STATUS")
    private String status;

    @OneToMany(orphanRemoval = true,
               cascade = CascadeType.ALL,
               fetch = FetchType.LAZY,
               mappedBy = "rTableId")
    private List<Reservation> reservations;

    @OneToMany(orphanRemoval = true,
               cascade = CascadeType.ALL,
               fetch = FetchType.LAZY,
               mappedBy = "rTableId")
    private List<Bill> bills;
```

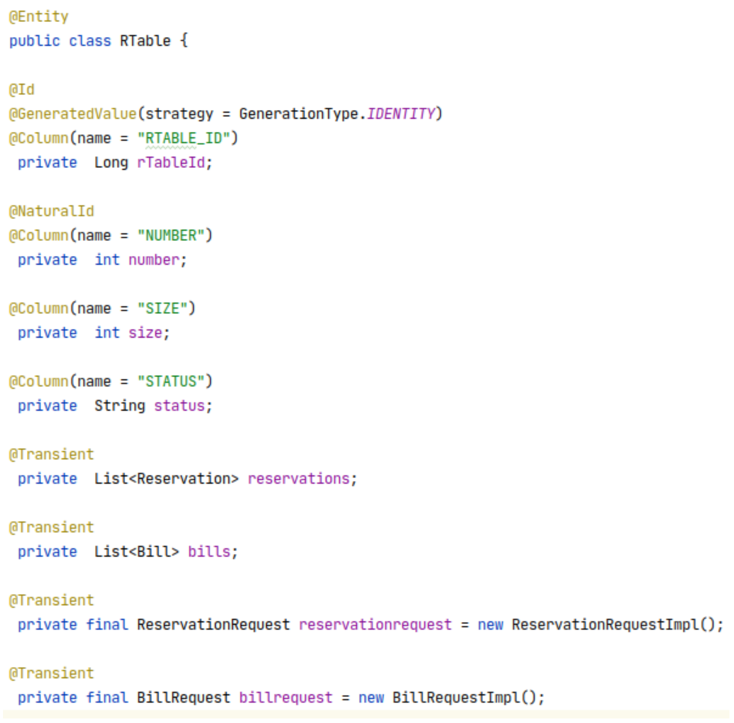
Figure 36: Instance variables of the RTable class in monolithic application



```java
@Entity
public class Bill {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "BILL_ID")
    private Long billId;
    @Column(name = "STATUS")
    private String status;
    @Column(name = "CREATION_DATE")
    private Date creationDate;
    @Column(name = "RTABLE_ID")
    private Long rTableId;
    @Column(name = "VALUE")
    private double value;

    @OneToMany(orphanRemoval = true,
        cascade = CascadeType.ALL,
        fetch = FetchType.LAZY,
        mappedBy = "billId")
    private List<BillPosition> billPositions;
```

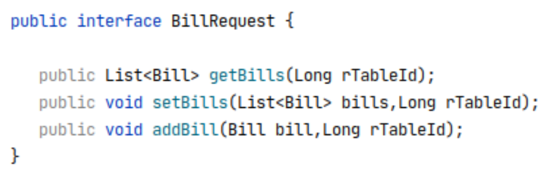
Figure 37: Instance variables of the Bill class in monolithic application

A.2.2.2 Relationship in microservice-based application



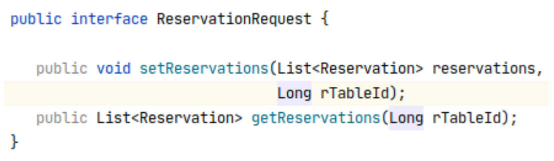
```java
@Entity
public class RTable {

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "RTABLE_ID")
 private  Long rTableId;

@NaturalId
@Column(name = "NUMBER")
 private  int number;

@Column(name = "SIZE")
 private  int size;

@Column(name = "STATUS")
 private  String status;

@Transient
 private  List<Reservation> reservations;

@Transient
 private  List<Bill> bills;

@Transient
 private final ReservationRequest reservationrequest = new ReservationRequestImpl();

@Transient
 private final BillRequest billrequest = new BillRequestImpl();
```

Figure 38: Instance variables of the RTable class in microservice-based application



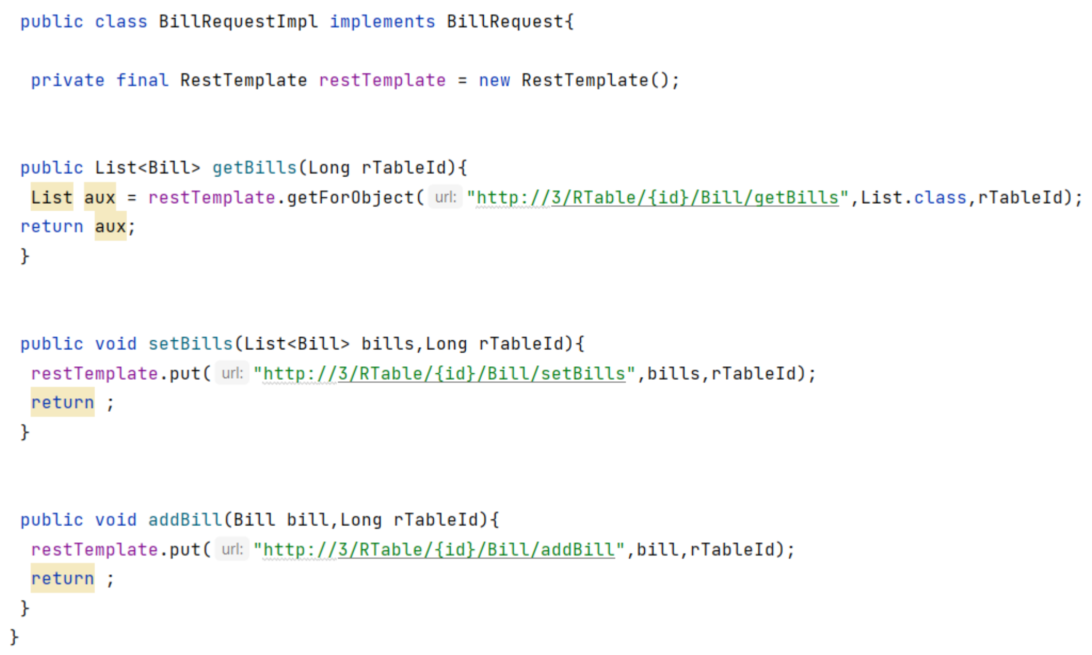
```java
public interface BillRequest {

   public List<Bill> getBills(Long rTableId);
   public void setBills(List<Bill> bills,Long rTableId);
   public void addBill(Bill bill,Long rTableId);
}
```

Figure 39: Interface BillRequest generated by the database refactoring



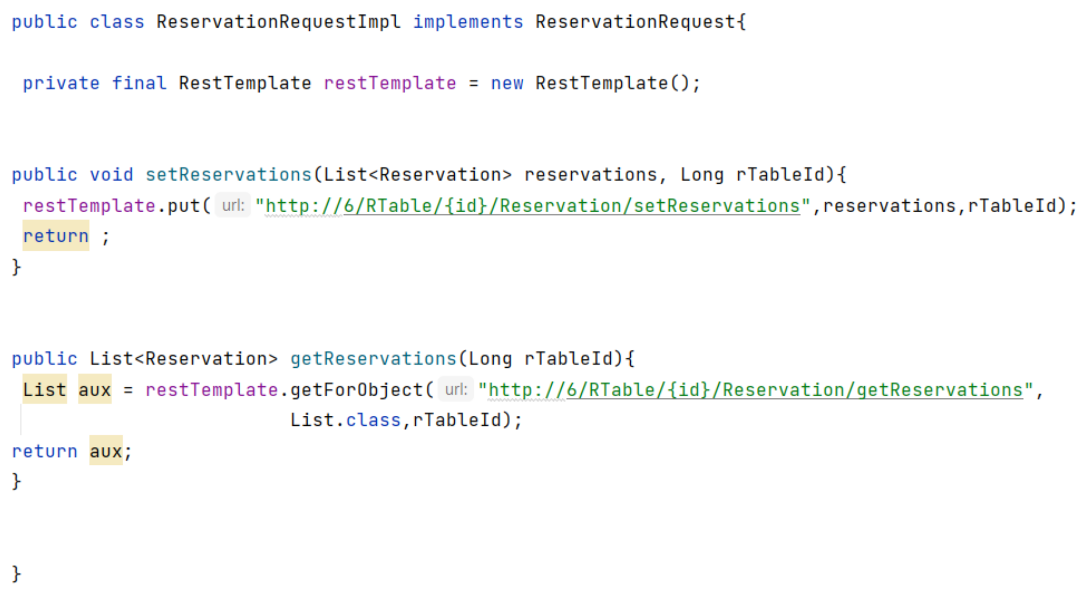
```java
public interface ReservationRequest {

   public void setReservations(List<Reservation> reservations,
                                          Long rTableId);
   public List<Reservation> getReservations(Long rTableId);
}
```

Figure 40: Interface ReservationRequest generated by the database refactoring



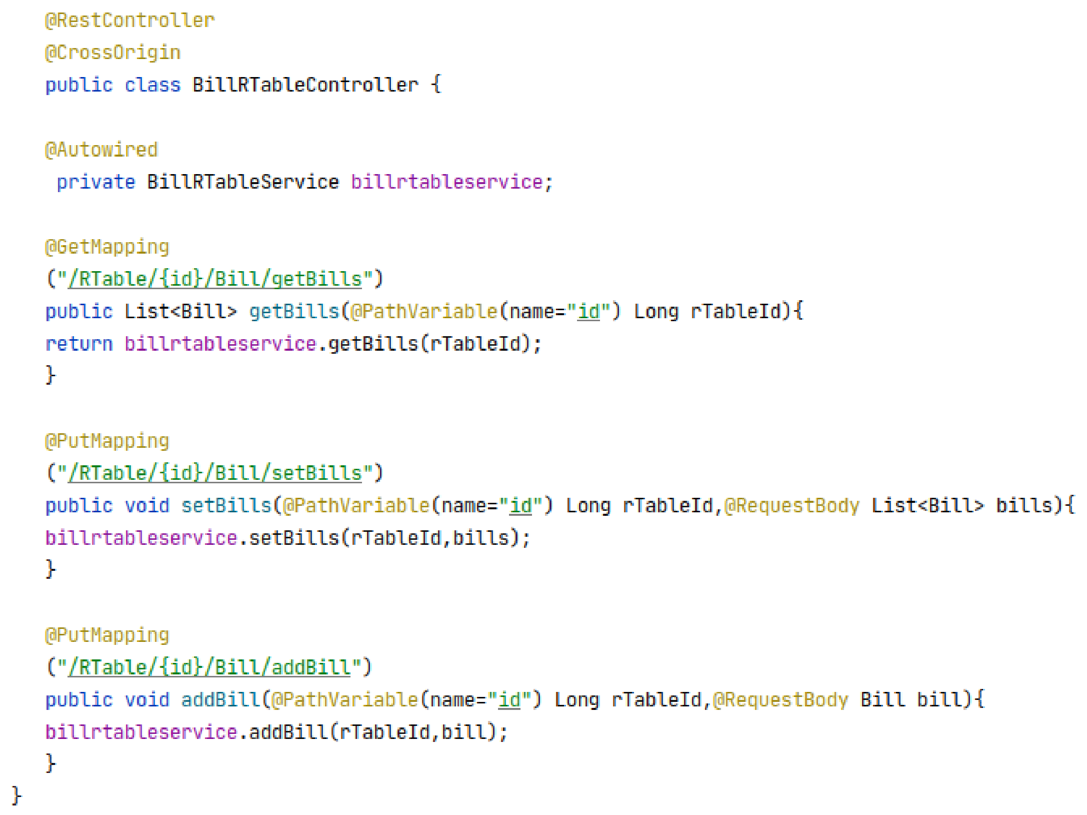
```java
public class BillRequestImpl implements BillRequest{

 private final RestTemplate restTemplate = new RestTemplate();


public List<Bill> getBills(Long rTableId){
 List aux = restTemplate.getForObject( url: "http://3/RTable/{id}/Bill/getBills",List.class,rTableId);
return aux;
}


public void setBills(List<Bill> bills,Long rTableId){
 restTemplate.put( url: "http://3/RTable/{id}/Bill/setBills",bills,rTableId);
 return ;
}


public void addBill(Bill bill,Long rTableId){
 restTemplate.put( url: "http://3/RTable/{id}/Bill/addBill",bill,rTableId);
 return ;
}
}
```

Figure 41: Generated class BillRequestImpl that is responsible for the calls to the Bill microservice



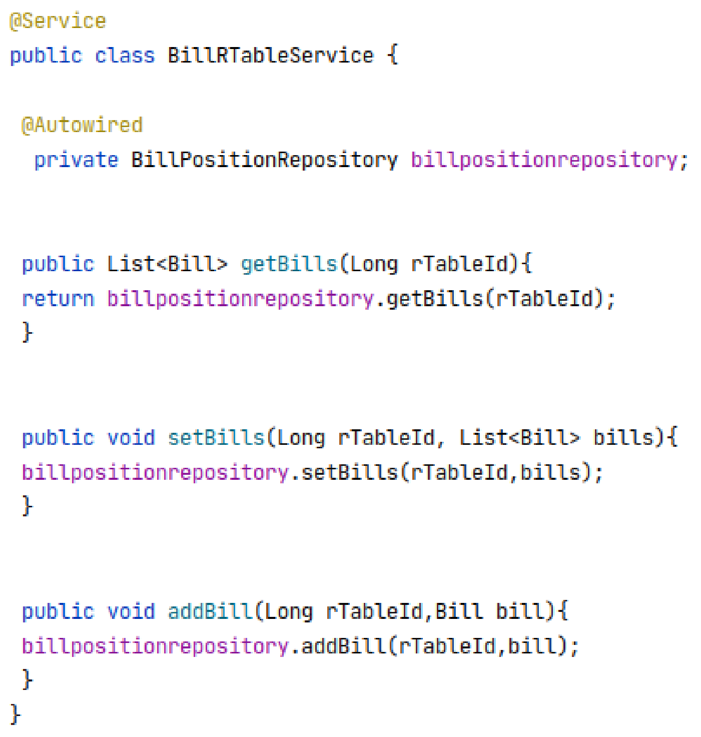
```
public class ReservationRequestImpl implements ReservationRequest{

 private final RestTemplate restTemplate = new RestTemplate();


public void setReservations(List<Reservation> reservations, Long rTableId){
 restTemplate.put( url: "http://6/RTable/{id}/Reservation/setReservations",reservations,rTableId);
 return ;
}


public List<Reservation> getReservations(Long rTableId){
 List aux = restTemplate.getForObject( url: "http://6/RTable/{id}/Reservation/getReservations",
                          List.class,rTableId);
return aux;
}


}
```

Figure 42: Generated class ReservationRequestImpl that is responsible for the calls to the Reservation microservice



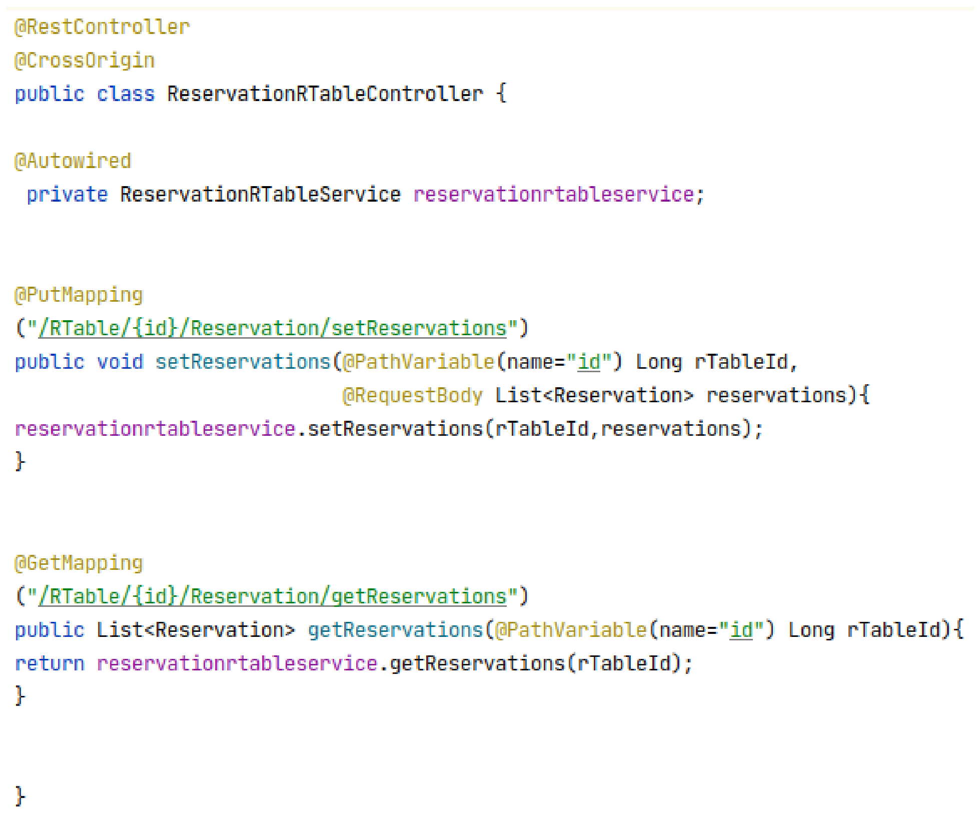
```
   @RestController
   @CrossOrigin
   public class BillRTableController {

   @Autowired
    private BillRTableService billrtableservice;

   @GetMapping
   ("/RTable/{id}/Bill/getBills")
   public List<Bill> getBills(@PathVariable(name="id") Long rTableId){
   return billrtableservice.getBills(rTableId);
   }

   @PutMapping
   ("/RTable/{id}/Bill/setBills")
   public void setBills(@PathVariable(name="id") Long rTableId,@RequestBody List<Bill> bills){
   billrtableservice.setBills(rTableId,bills);
   }

   @PutMapping
   ("/RTable/{id}/Bill/addBill")
   public void addBill(@PathVariable(name="id") Long rTableId,@RequestBody Bill bill){
   billrtableservice.addBill(rTableId,bill);
   }
}
```

Figure 43: Generated class that exposes the API to handle Bills



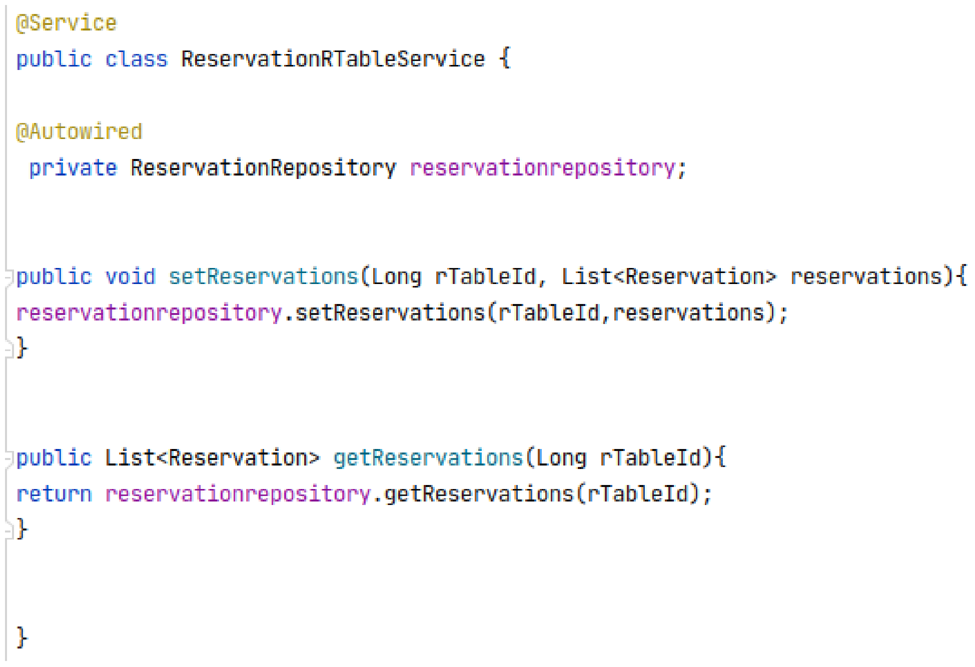
```java
@Service
public class BillRTableService {

 @Autowired
  private BillPositionRepository billpositionrepository;


 public List<Bill> getBills(Long rTableId){
 return billpositionrepository.getBills(rTableId);
 }


 public void setBills(Long rTableId, List<Bill> bills){
 billpositionrepository.setBills(rTableId,bills);
 }


 public void addBill(Long rTableId,Bill bill){
 billpositionrepository.addBill(rTableId,bill);
 }
}
```

Figure 44: Generated class that processes and directs the request to the BillRepository class



```java
@RestController
@CrossOrigin
public class ReservationRTableController {

@Autowired
 private ReservationRTableService reservationrtableservice;


@PutMapping
("/RTable/{id}/Reservation/setReservations")
public void setReservations(@PathVariable(name="id") Long rTableId,
                            @RequestBody List<Reservation> reservations){
reservationrtableservice.setReservations(rTableId,reservations);
}


@GetMapping
("/RTable/{id}/Reservation/getReservations")
public List<Reservation> getReservations(@PathVariable(name="id") Long rTableId){
return reservationrtableservice.getReservations(rTableId);
}


}
```

Figure 45: Generated class that exposes the API to handle Reservations



```
@Service
public class ReservationRTableService {

@Autowired
 private ReservationRepository reservationrepository;


public void setReservations(Long rTableId, List<Reservation> reservations){
reservationrepository.setReservations(rTableId,reservations);
}


public List<Reservation> getReservations(Long rTableId){
return reservationrepository.getReservations(rTableId);
}



}
```

Figure 46: Generated class that processes and directs the request to the ReservationRepository class

B Appendix 2

B.1 Evaluation results from GitHub projects

Project Github	#Classes	#PM	Refact?	#MG	#CAR	#RR	#DR	%NC	tests?	Results	Reason
benceburom/smart-kitchen	85	7	Yes	8	147	14	15	73	Yes	Same result in both versions	
pibigstar/parsevip	90	12	No	-	-	-	-	-	-	-	Missing Repository
Vino007/javaEEScaffold	45	5	No	-	-	-	-	-	-	-	Create DAO
514840279/danyuan-application	277	15	No	-	-	-	-	-	-	-	Missing Id annotation
cym1102/nginxWebUI	127	12	Yes	12	315	0	269	148	No	-	
jaselvam/Sprint2_Forestmanagement	65	8	Yes	8	95	0	8	46	Yes	Same result in both versions	
INCF/eeg-database	1007	29	No	-	-	-	-	-	-	-	Create DAO
Haseoo/courier-company-system	312	11	Yes	12	380	14	115	22	Yes	Same result in both versions	
st52542/eshopSemPrace	59	7	Yes	8	197	9	32	234	Yes	Same result in both versions	
BarCzerw/inTeams	57	5	Yes	6	188	8	52	230	Yes	Same result in both versions	

Project Github	#Classes	#PM	Refact?	#MG	#CAR	#RR	#DR	%NC	tests?	Results	Reason
YangLeeThomson/mwebsvr	174	13	Yes	13	258	0	128	48	Yes	Same result in both versions	
derekreynolds/onleave	149	13	Yes	13	145	4	9	-3	Yes	Same result in both versions	*
tangdu/smh2	112	11	Yes	11	212	0	90	89	No	-	
jdmr/mateo	704	28	No	-	-	-	-	-	-	-	Create DAO
leluque/university-site-cms	254	15	No	-	-	-	-	-	-	-	Id annotation in methods
OpenGeoportal/OGP2	330	18	Yes	18	482	0	244	46	No	-	
mrabusalah/Qr-Students-Attendance	91	9	Yes	9	99	0	5	9	Yes	Same result in both versions	
kai8406/cmop	237	13	Yes	13	557	0	1096	135	No	-	
gliderwiki/glider	329	20	Yes	20	511	0	705	55	Yes	Same result in both versions	
mozammel/mNet	148	16	Yes	16	438	38	78	196	Yes	Same result in both versions	
HIIT/dime-server	91	7	No	-	-	-	-	-	-	-	Missing Id annotation

Project Github	#Classes	#PM	Refact?	#MG	#CAR	#RR	#DR	%NC	tests?	Results	Reason
justinbaby/my-paper	463	19	Yes	19	971	0	771	110	No	-	
cruzerblade95/school-attendance-system-api	91	8	Yes	8	102	0	5	12	Yes	Same result in both versions	
zhangyanbo2007/youkefu	830	60	Yes	60	1337	0	2221	61	No	-	
EUSurvey/EUSURVEY	356	11	Yes	11	1133	0	6949	218	Yes	Same result in both versions	
suyeq/steamMall	120	12	Yes	12	245	0	138	104	No	-	
Prasad108/TutesMessanger	109	11	Yes	11	238	0	208	118	No	-	
busing/circle_web	268	26	Yes	26	428	0	399	60	No	-	
bbaibb1009/wxcrm	95	12	Yes	12	206	0	165	117	No	-	
khasang/delivery	218	16	Yes	16	225	0	13	0	Yes	Same result in both versions	
TexnologiaLogismikou/Fiz	199	13	Yes	13	174	0	80	-13	Yes	Same result in both versions	*
AI-2020-2021-LAB1/stock-exchange-app-backend	119	9	Yes	9	242	7	29	103	Yes	Same result in both versions	
wang007/live-server	238	18	No	-	-	-	-	-	-	-	Create DAO

Project Github	#Classes	#PM	Refact?	#MG	#CAR	#RR	#DR	%NC	tests?	Results	Reason
ushahidi/SwiftRiver-API	249	12	No	-	-	-	-	-	-	-	Create DAO
softservedata/lv257	171	9	No	-	-	-	-	-	-	-	Create DAO
bao17634/Warehouse-system	76	11	Yes	11	134	0	101	76	No	-	
Seenck/jeecg-bpm-3.8	810	23	Yes	23	961	0	1946	19	Yes	Same result in both versions	
GraffiTab/GraffiTab-Backend	188	12	No	-	-	-	-	-	-	-	Use Orika for ORM
surajcm/Poseidon	113	7	Yes	7	106	0	83	-6	Yes	Same result in both versions	*
WilsonHu/sinsim	264	16	No	-	-	-	-	-	-	-	Use MyBatis for ORM
dp2-g56/Dp2-L02	245	12	Yes	12	585	0	119	139	Yes	Same result in both versions	
wangwang1230/te-empl	156	8	Yes	8	215	0	277	38	No	-	
Glamdring/welshare	241	16	No	-	-	-	-	-	-	-	Create DAO
OHDSI/ArachneCentralAPI	841	19	No	-	-	-	-	-	-	-	Missing Id annotation

Project Github	#Classes	#PM	Refact?	#MG	#CAR	#RR	#DR	%NC	tests?	Results	Reason
Eliathen/sharelibrary	182	13	Yes	14	393	20	53	116	Yes	Same result in both versions	
momoplan/dataanalysis	195	12	No	-	-	-	-	-	-	-	Missing Repository
isa-group/ideas-studio	115	10	Yes	10	160	0	63	39	Yes	Same result in both versions	
smartcommunitylab/smartcampus.vas.corsi.web	65	8	No	-	-	-	-	-	-	-	Missing Repository
XMFBee/AuthPlatform1	281	22	Yes	22	609	0	1258	117	No	-	
scrumtracker/scrumtracker2017	44	6	Yes	7	163	5	97	270	No	-	
racem-cherni/KinderGartenProject	209	12	No	-	-	-	-	-	-	-	Missing Repository
shangtech/WeiXinPlatform	88	8	Yes	8	165	0	84	88	No	-	
longyzkd/wj-web-ext-enhancer	154	12	Yes	12	205	0	140	33	No	-	
gvSIGAssociation/gvsig-web	127	7	Yes	7	146	0	30	15	No	-	
immime/shop-2	292	17	Yes	17	588	0	114	101	No	-	
hongqiang/shopb2b	465	23	No	-	-	-	-	-	-	-	Create DAO
devopsrepo973/switchProject	501	14	Yes	14	497	0	182	-1	Yes	Same result in both versions	*
SupermePower/zammc-manage	119	13	Yes	13	154	0	102	29	No	-	

Project Github	#Classes	#PM	Refact?	#MG	#CAR	#RR	#DR	%NC	tests?	Results	Reason
dtdhehe/ptu-life	66	10	Yes	10	125	0	103	89	No	-	
Haseoo/TeleLearn-Backend	229	13	Yes	13	309	16	67	35	Yes	Same result in both versions	
398907877/AppPortal	470	22	No	-	-	-	-	-	-	-	Missing Id annotation
xabaohui/zis	553	23	No	-	-	-	-	-	-	-	Missing Repository
Russel-JX/OUC-Family	276	33	No	-	-	-	-	-	-	-	Create DAO
Maxcj/Maxcj	177	15	Yes	15	256	0	108	45	No	-	
assertmyself/gweb-v2	365	29	Yes	29	464	0	183	27	No	-	
MarceloMansilla-zz/Veterinaria	132	10	Yes	10	182	5	36	27	Yes	Same result in both versions	
hyperaeon/CrazyAndOptimize	1399	155	Yes	155	836	0	23	- 40	No		*
veerappanvs/elae.cap	224	18	No	-	-	-	-	-	-	-	Missing Repository
pollishulya/agency	82	7	No	-	-	-	-	-	-	-	Missing Repository
xirui0920/bishe	97	12	Yes	12	109	2	8	12	Yes	Same result in both versions	
ncontrerasn/PetSuite	123	7	No	-	-	-	-	-	-	-	Missing Repository

Project Github	#Classes	#PM	Refact?	#MG	#CAR	#RR	#DR	%NC	tests?	Results	Reason
SamuelBiazotto/Desafio-Lanches-Backend	27	4	No	-	-	-	-	-	-	-	Missing Repository
Fzshuai/FzshuaiBlog	56	7	Yes	8	113	7	38	102	No	-	
mscaldas2012/chronicApps	29	3	Yes	3	36	2	2	24	No	-	
FabricadeSoftwareUNIVILLE/ProjCOLABAssistant	63	7	No	-	-	-	-	-	-	-	Missing Repository
xxn520/yitao	112	9	No	-	-	-	-	-	-	-	Missing Id annotation
paulosanchezh/lwRepository	50	6	Yes	8	144	11	32	188	No	-	
mauricedibbets1986/qienurenappgroep2	69	5	No	-	-	-	-	-	-	-	Missing Id annotation
siwar20/consomi_tounsi	112	14	No	-	-	-	-	-	-	-	Missing Repository
restockDAW/restockBackend	48	6	Yes	6	92	0	43	92	No	-	
bau03/E-Ticaret-Backend-Springboot	97	10	Yes	11	236	13	123	143	No	-	
xhieu2206/bankingapp	187	11	No	-	-	-	-	-	-	-	Create DAO
kkminyoung/DSC-Hackathon	45	6	Yes	6	67	0	9	49	No	-	
saeedshokoohi/tikon	553	32	No	-	-	-	-	-	-	-	Missing Repository
galimsarov/java-thesis-project	73	7	Yes	7	108	0	18	48	No	-	
PepperNoodles/PepperNoodles	184	13	No	-	-	-	-	-	-	-	Missing Repository

Project Github	#Classes	#PM	Refact?	#MG	#CAR	#RR	#DR	%NC	tests?	Results	Reason
eduardosukeda/vendas-api	47	5	Yes	5	79	5	4	68	No	-	
SLCricketLeaderBoard/SLCricketLeaderBoard_BackEnd	140	14	Yes	14	385	24	220	175	No	-	
kiva12138/ITBlogSiteDesignAndCode	82	6	Yes	6	60	0	26	-27	No	*	
SliferDMC/Backup-GRI-Frontend	48	6	Yes	7	91	9	39	90	No	-	
Khrustal/coursework-backend	34	4	Yes	5	83	5	16	144	No	-	
WinwindH/CareerBridge	98	11	Yes	11	123	0	9	26	No	-	
carlan92/Hospital_Management_System	129	8	Yes	8	255	18	152	59	No	-	
flaviohlm/segue_me	133	11	No	-	-	-	-	-	-	-	Missing Repository
wn150509/OA_system	237	12	Yes	13	535	33	510	126	No	-	
MenkeTechnologies/puffride	41	9	Yes	9	103	11	1	151	No	-	
namndph10091/VINFAST	45	5	Yes	5	97	4	12	116	No	-	
pratyush-sdrc/SDRC-Collect-Web	193	27	Yes	27	233	0	444	21	No	-	
gigarthan/sliit-procurement-system	53	7	Yes	7	75	0	8	42	No	-	
CocayUNAM/ProyectoUNAM	112	9	Yes	9	279	12	184	149	No	-	
Himani-Bonde/BugTracker	29	5	No	-	-	-	-	-	-	-	Missing Repository
Fordclifford/mregister	59	6	Yes	7	71	1	42	20	No	-	
offway/athena	196	18	Yes	18	357	0	165	82	No	-	
LorenaAcosta/spa-restful-api	69	11	No	-	-	-	-	-	-	-	Missing Repository
balagaff/eComm-master-DR	34	6	Yes	6	63	0	14	3	No	-	

Project Github	#Classes	#PM	Refact?	#MG	#CAR	#RR	#DR	%NC	tests?	Results	Reason
OlivierBrinkman/Eindopdracht_Backend	48	4	No	-	-	-	-	-	-	-	Missing Repository
Jtagirova/Weflors	54	7	Yes	7	80	0	25	48	No	-	
NiuPiFiveTeam/HotelManageSystem	224	16	Yes	16	407	0	238	82	Yes	Same result in both versions	
samyumaddikunta/Online_Movie_Ticket_System	36	5	Yes	5	76	0	14	111	No	-	
jixs/coolweather/daqi	402	30	Yes	30	573	0	260	43	No	-	
lionsdamajectim/Champink	35	5	Yes	5	104	18	39	197	No	-	
ariv98/Online-medicine-shopping-ecommerce	69	8	Yes	8	204	8	6	196	Yes	Same result in both versions	
Aaryatech/HrEsayWebApiPune	586	28	Yes	28	975	0	2221	66	No	-	
ihlys/musicstore	119	10	No	-	-	-	-	-	-	-	Missing Repository
Lavanya-Annadi/PetAdapt	40	5	Yes	5	84	7	18	110	No	-	
navinsharma25/Hotel-Booking-Management-System	43	8	Yes	8	140	NA	NA	226	Yes	Same result in both versions	
MandemHarshithaReddy527/HMS	53	8	Yes	8	79	6	0	49	No	-	
asledziewski/restaurantServer	38	7	Yes	8	138	5	37	263	No	-	

Project Github	#Classes	#PM	Refact?	#MG	#CAR	#RR	#DR	%NC	tests?	Results	Reason
altsora/SocialNetwork	119	9	Yes	10	206	8	75	73	Yes	Same result in both versions	
yu-gotcha/time-attack-web	32	4	Yes	4	78	8	5	144	No	-	